O predstavitvi podatkov v računalniku: decimalna števila

↓↓↓
Jure Slak

→ **Z računalniki lahko dandanes izvajamo kompleksne matematično-fizikalne izračune in simulacije. V marcu 2020 je omrežje Folding@Home, ki se uporablja za izračune zlaganja proteinov, preseglo 1.5 exaFLOPS-a, to je več kot 1,500,000,000,000,000,000 računskih operacij na sekundo. Že v enoti sami, ki jo uporabljamo za merjenje hitrosti FLOPS, se skriva glavni podatkovni tip, ki stoji za vsemi simulacijami. FLOPS namreč pomeni *floating point operation per second* in pove, koliko računskih operacij z decimalnimi števili lahko naredimo v eni sekundi. V tem prispevku se bomo poglobili v to, kako decimalna števila sploh predstavimo v računalniku in kako z njimi računamo.**

Decimalna ali realna števila

Velikokrat se pogovorno reče, da v računalniku hranimo realna števila. Nekateri programski jeziki, npr. različne verzije SQL, tudi uporabljajo besedo `real` za oznako tipa. Vendar kljub temu v računalniku nekaterih realnih števil ne moremo predstaviti zelo enostavno. Iracionalna števila, kot npr. $\sqrt{2}$ ali π, običajno le aproksimiramo. Ravno število π je v programskem jeziku C definirano kot

- `#define M_PI 3.14159265358979323846`,

torej »le« na 20 decimalk, precej manj kot trenutni slovenski rekord 3333 decimalk, ki jih je znal na pamet povedati zmagovalec zadnjega π-dneva. Kot bomo videli, računalnik uporablja le racionalna števila omejene natančnosti – morebitna realna števila so ustrezno zaokrožena. To nam omogoča enostavnost računanja in hitrost, natančnost pa ni največja. A brez skrbi, večinoma so decimalna števila, ki jih uporablja računalnik, povsem dovolj natančna.

Fiksna in plavajoča pika

FLOPS se v prvih dveh črkah nanaša na operacije s plavajočo piko.[1] Predstavitev decimalnih števil s plavajočo piko je ena izmed dveh osnovnih načinov predstavitve števil. Druga, enostavnejša možnost je predstavitev s fiksno decimalno piko. Pri slednji imamo na voljo nekaj mest za del pred piko in nekaj mest za decimalno piko. Denimo, da imamo dve mesti pred in dve mesti po piki. Števila, ki jih lahko zapišemo, so torej od 00.00 do 99.99. Interval od $[0, 100]$ smo enakomerno pokrili s 10000 števili na razdalji 0.01. Tem številom rečemo *predstavljiva*, vseh ostalim pa nepredstavljiva. Vsako realno število, s katerim želimo računati, moramo zaokrožiti; ponavadi izberemo najbližje predstavljivo število. Za število x bomo z $\mathrm{fl}(x)$ označili predstavljivo število, kjer zaokrožimo x.

Seštevanje in odštevanje predstavljivih števil je enostavno in natančno, lahko pa pride do prekoračitve ali podkoračitve; to pomeni, da rezultat leži izven razpona možnih vrednosti. Pri množenju in deljenju pridemo do novih težav. Če pomnožimo 0.5 in 0.41, pri točnem računanju dobimo 0.205, kar pa ni predstavljivo; rezultat je potrebno zaokrožiti na najbližje predstavljivo število. V našem primeru imamo dve izbiri: zaokrožimo lahko na 0.20 ali na 0.21. Čeprav je v resničnem življenju pogosto, da polovice zaokrožamo navzgor, v računalništvu pona-

[1] Pogosto se jim v slovenščini reče tudi števila s plavajočo vejico, saj je vejica v slovenščini ločilo, ki se uporablja za označevanje decimalnih mest. Vendar je v računalništvu precej bolj pogosta pika, zato jo bomo uporabljali tudi v tem prispevku.

vadi velja, da polovice zaokrožamo proti »sodemu številu«. V našem primeru bi zaokrožili k 0.20, saj je zadnja števka 0 soda. Tako pravilo uporabljamo, da bolj enakomerno razporedimo realna števila k njihovim decimalnim številom, saj se nekatere polovice zaokrožijo navzgor, nekatere pa navzdol. Računanje s fiksno decimalno piko zagotavlja določeno absolutno natančnost: v našem primeru bo zaokrožitvena napaka kvečjemu 0.005, polovica ločljivosti števil, s katerimi delamo. Napisano z enačbo:

- $|x - \mathrm{fl}(x)| \leqslant 0.005$.

Absolutna natančnost pa prinese tudi svoje težave. Kaj če želimo izračunati 0.05×0.08? Točen rezultat bi bil 0.04, kar se zaokroži na 0. Podobno ne moremo izračunati 30.12×40.23, saj je rezultat večji od 99.99. Računanje s fiksni decimalno piko preprosto ni dovolj fleksibilno za delo z velikimi ali majhnimi števili; dobro deluje le, če so vsa števila, s katerimi delamo, med 0 in 99.99. Tukaj pa je prednost drugega sistema predstavitve števil, predstavitev s plavajočo piko (angl. floating point). V tem sistemu števila predstavimo v obliki $m \times 10^e$, za število $m \in [0, 1)$, ki se imenuje *mantisa*, in naravni eksponent e. Kako sistem deluje, si poglejmo na podobnem primeru kot pri predstavitvi s fiksno decimalno piko. Denimo, da imamo na voljo dve mesti za mantiso m in dve za eksponent e. Nekaj možnih števil je tako 0.23, $0.54 \cdot 10^{12}$, $0.10 \cdot 10^1$. Za eksponente bomo namesto dvomestnih števil od 0 do 99 dovolili števila od -50 do 49. Še vedno obdržimo enako količino števil, le malo jih zamaknemo, da lahko delamo tudi z negativnimi eksponenti. Tako so tudi števila $0.1 \cdot 10^{-12}$ in $0.43 \cdot 10^{-33}$ veljavna.

Ker lahko števila zapišemo na več različnih načinov (npr. $1 = 1 = 0.1 \cdot 10^1 = 0.01 \cdot 10^2$), se dogovorimo, da vedno pišemo v *normalizirani* obliki, to je v obliki, kjer se število začne z $0.x$, za neničelno decimalko x. Število 0.0062 bi napisali kot $0.62 \cdot 10^{-2}$, število 123 pa bi zaokrožili na $0.12 \cdot 10^2$. Razpon števil, ki ga s plavajočo piko lahko dosežemo s štirimi mesti, je precej večji kot pri fiksni piki: najmanjše normalizirano pozitivno število je $0.1 \cdot 10^{-50}$, torej 0.00000000000000000000000000000000000000000000000000001, največje $0.9 \cdot 10^{49}$, torej 900000000000000000000000000000000000000000000000. Seveda nimamo povsod enake absolutne natančnosti, saj imamo še vedno na voljo samo 10000 števil – pri številih okoli 500 bo razlika med sosednjimi števili 10 (npr. med $0.5 \cdot 10^3$ in $0.51 \cdot 10^3$), pri številih okoli 700000 bo ločljivost 10000, pri številih okoli 0.0003 pa 0.00001. Vidimo, da se ločljivost prilagaja velikosti števil. To je tudi smiselno: če računamo nekaj v kilometrih, pričakujemo tudi natančnost v redu velikosti kilometrov, če pa računamo v mikrometrih, prav tako pričakujemo natančnost v mikrometrih. Za števila, predstavljena s plavajočo piko, pravimo, da imajo približno enako *relativno* natančnost. Če število x zapišemo s plavajočo piko, bo absolutna napaka pri zaokrožitvi morda velika, relativna napaka

- $\dfrac{|\mathrm{fl}(x) - x|}{|x|}$

pa bo vedno manjša od 0.05. Tej vrednosti rečemo osnovna zaokrožitvena napaka.

Za konec si razliko med fiksno in plavajočo vejico poglejmo še grafično. Na sliki 1 vidimo primerjavo predstavljivih števil v fiksni in premični piki.

Računanje s premično piko

Premična pika je bolj fleksibilna pri računskih operacijah. Računalnik nam za osnovne računske operacije ($+$, $-$, $\times$, $/$) zagotavlja, da bo rezultat operacije zaokrožen na najbližje predstavljivo število. Če seštejemo 6.3 in 7.4 bi pri točnem izračunu dobili 13.7, vendar se rezultat zaokroži na 14, saj imamo na voljo le dve mesti natančnosti. Formalno izračun poteka tako:

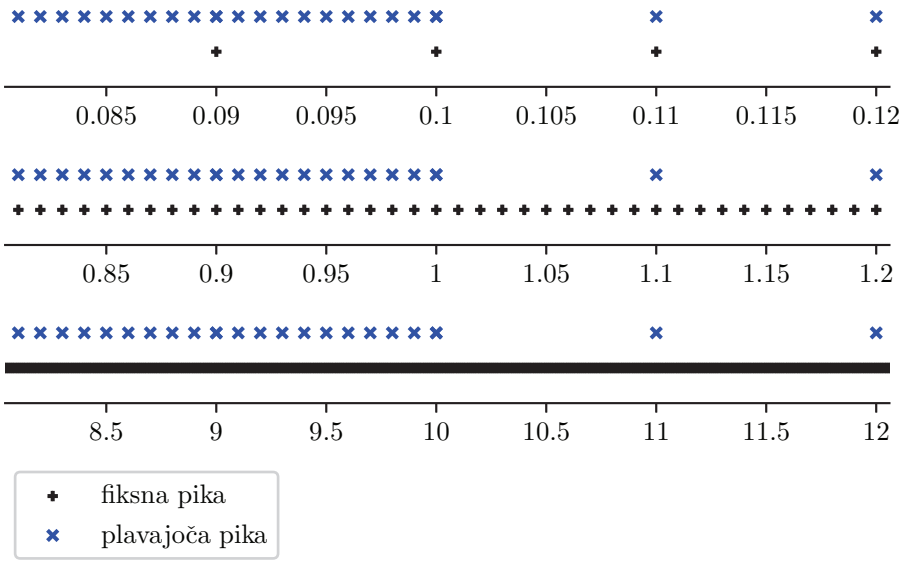
- $0.63 \cdot 10^1 + 0.74 \cdot 10^1 = \mathrm{fl}(1.37 \cdot 10^1) = 0.14 \cdot 10^2$.

Podobno se zgodi pri množenju: 6.3 pomnoženo s 7.4 bi s točnim računanjem dalo 46.62, v premični piki, pa se izračuna v

- $0.63 \cdot 10^1 \times 0.74 \cdot 10^1 = \mathrm{fl}(4.662 \cdot 10^1) = 0.46 \cdot 10^2$.

Ker imamo le dve mesti natančnosti, moramo pri rezultatu zavreči dve decimalki, toda relativna napaka $(46.62 - 46)/46.62 \approx 0.013$ je manjša od osnovne zaokrožitvene napake 0.05. Poglejmo še primer 0.05×0.08, ki je pri fiksni decimalni piki povzročal težave. Točen rezultat je 0.004, v premični piki dobimo

- $0.5 \cdot 10^{-1} \times 0.8 \cdot 10^{-1} = \mathrm{fl}(0.04 \cdot 10^{-1}) = 0.4 \cdot 10^{-2}$,



SLIKA 1.
Primerjava pogostosti predstavljivih števil na treh različnih delih realne osi. Števila s fiksno piko imajo enako absolutno natančnost ne glede na lokacijo, števila s premično piko pa postajajo čedalje bolj redka, toda ohranjajo enako relativno natančnost. Na grafih se spreminjajo enote, zato števila s fiksno piko izgledajo, kot da so čedalje bolj gosta, števila s premično piko pa se zdijo enako gosta.

kar je popolnoma točno. Če bi dodali malo več decimalk in izračunali 0.053×0.082, bi izračun potekal tako:

- $0.53 \cdot 10^{-1} \times 0.82 \cdot 10^{-1} = \mathrm{fl}(0.4346 \cdot 10^{-1})$
  $= 0.43 \cdot 10^{-2}$.

V tem primeru rezultat ni točen, je pa precej boljši kot pri fiksni piki. Njegova relativna napaka je približno 1%.

Poučno je tudi,kaj se zgodi, če izračunamo npr. $100 + 0.1$. Dobimo

- $0.1 \cdot 10^3 \times 0.1 = \mathrm{fl}(0.1001 \cdot 10^3) = 0.1 \cdot 10^3$.

Rezultat je zopet točno 100, saj vrednost 0.1 ni bila dovolj velika, da bi jo upoštevali, in se je zgubila pri zaokroževanju. Še vedno pa je to znotraj zaokrožitvene napake: 0.1 predstavlja le 0.1% od 100, kar je močno znotraj dovoljene 5% napake.

Dejanska števila v računalniku

Pri delu z decimalnimi števili v računalniku ne uporabljamo le dveh decimalnih mest, kot smo jih mi do sedaj, a principi računanja kljub temu ostanejo enaki. Delo z decimalnimi števili predpisuje standard IEEE 754. Glavni tip, ki ga najpogosteje uporabljamo, se imenuje `double`. Velik je 64 bitov in se imenuje dvojna natančnost – tako ime ima, ker je dvakrat večji od 32-bitnega tipa `single`, ki predstavlja enojno natančnost. Decimalna števila so shranjena v dvojiškem sistemu, ne v desetiškem. Primer števila bi bilo npr.

- 0.101101×2^3,

kar pretvorimo v

- $(1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4} + 0 \cdot 2^{-5} + 1 \cdot 2^{-6})$
  $\times 2^3 = 5.625$.

Izmed 64 bitov, ki so na voljo, jih je 11 rezerviranih za eksponent, 52 za mantiso (decimalke), in 1 za predznak števila (+ ali −). 11-bitni prosto za eksponent nam omogoča eksponente od −1022 do 1023, najmanjši in največji eksponent −1023 in 1024 pa sta rezervirana za posebna števila, o katerih bomo več povedali kasneje. V dvojiškem sistemu velja omeniti tudi posebno obliko normaliziranega zapisa. Če število zapišemo v običajnem normaliziranem zapisu, bo oblike $0.x$, kjer je x neničelna števka. Toda v dvojiškem je ta števka lahko le 1, zato je nepotrebno, da jo shranjujemo, in se raje dogovorimo, da kot normalizirano obliko za dvojiška decimalna števila vzamemo $1.x$, kjer je x poljuben, 0 ali 1. Tako najbolje izkoristimo 52 dvojiških decimalk, ki jih imamo na voljo, kar je enako približno 16 desetiškim decimalkam natančnosti. V tem sistemu je osnovna zaokrožitvena napaka enaka $1.11 \cdot 10^{-16}$.

Decimalno število s 64 biti je v računalniku predstavljeno kot:

- $s\ \underbrace{eee\ldots eee}_{\substack{11\text{ bitov}\\ \text{za eksponent}}}\ \underbrace{mmm\ldots mmm}_{52\text{ bitov za mantiso}}$,

kjer s predstavlja en bit za predznak.

Poglejmo si konkreten primer. Bitni zapis števila 4269.6842 je enak

- 0 10000001011 0000101011011010111100100 1111011101100101111111101100,

kjer so vrinjeni presledki za lažje branje. Prvi bit je enak 0, kar nam pove, da je število pozitivno. Sledi eksponent; če 10000001011 pretvorimo v desetiško, dobimo 1035. Vendar so eksponenti zamaknjeni, saj nimajo razpona od 0 do 2048, temveč od -1023 do 1024. Z upoštevanjem zamika je iskani eksponent enak $1035 - 1023 = 12$. Zapisano število je tako enako

- $1.0000101011011010111100100111101110110010111111101100 \times 2^{12}$,

kar je enako

- 1000010101101.1010111100100111101110110010111111101100

oz. približno 4269.6841999999996915. Kot vidimo, rezultat ni točno 4269.6842, temveč je za približno $3.09 \cdot 10^{-13}$, kar (relativno gledano) ustreza osnovni zaokrožitveni napaki.

Posledice zaokroževanja

Zaokroževanje na najbližjo vrednost pomeni, da običajna pravila računanja ne držijo več. Če v računalniku izračunamo $(a + b) + c$, to ni več nujno enako kot $a + (b + c)$. Poglejmo primer: vzemimo $a = 0.88$, $b = 0.56$ in $c = 0.13 \cdot 10^1$. Če izračunamo $a + b + c$ od leve proti desni, dobimo

- $$\begin{aligned} a + b + c &= 0.88 + 0.56 + 0.13 \cdot 10^1 \\ &= \mathrm{fl}(1.44) + 0.13 \cdot 10^1 \\ &= 0.14 \cdot 10^1 + 0.13 \cdot 10^1 \\ &= \mathrm{fl}(0.27 \cdot 10^1) = 0.27 \cdot 10^1. \end{aligned}$$

Če pa izračunamo $a + b + c$ od desne proti levi, dobimo

- $$\begin{aligned} a + b + c &= 0.88 + 0.56 + 0.13 \cdot 10^1 \\ &= 0.88 + \mathrm{fl}(1.86) = 0.88 + 0.19 \cdot 10^1 \\ &= \mathrm{fl}(2.78) = 0.28 \cdot 10^1. \end{aligned}$$

Razlika je majhna, vendar števili nista enaki. To je tudi eden izmed razlogov, da decimalna števila redko neposredno primerjamo, ali imajo popolnoma enako vrednost. Že majhne razlike v načinu izračuna namreč lahko prinesejo napake pri zadnjih decimalkah. Poslužimo se raje primerjanja s *toleranco*: namesto da bi pogledali, ali je $a = b$, pogledamo, ali je absolutna vrednost razlike med a in b manjša od tolerance t: $|a - b| \leqslant t$, za npr. $t = 0.00001$. Z izbiro vrednosti t lahko tudi določimo, kako velike napake so še sprejemljive.

Še ena zanimivost se pojavi pri računanju aritmetične sredine dveh števil. V matematiki smo navajeni, da aritmetična sredina $\frac{a+b}{2}$ dveh števil a in b leži natanko na sredini med številoma. Pri decimalnih številih temu ni tako: vzemimo npr. $a = 0.21$ in $b = 0.24$. Njuna izračunana aritmetična sredina je

- $$\begin{aligned} \frac{a+b}{2} &= \frac{1}{2}(0.21 + 0.24) = \frac{1}{2}\,\mathrm{fl}(0.43) = \\ &= \frac{1}{2}(0.43) = \mathrm{fl}(0.215) = 0.22, \end{aligned}$$

kar ni točen rezultat 0.215, toda je (eden izmed) najboljših možnih približkov.

Oglejmo si še en primer izračuna aritmetične sredine, tokrat za $a = 0.66$ in $b = 0.67$:

- $$\begin{aligned} \frac{a+b}{2} &= \frac{1}{2}(0.66 + 0.67) = \frac{1}{2}\,\mathrm{fl}(1.33) = \\ &= \frac{1}{2}(0.13 \cdot 10^1) = \mathrm{fl}(0.65) = 0.65. \end{aligned}$$

Tokrat smo dobili, da je sredina števil 0.66 in 0.67 enaka 0.65, kar seveda leži izven intervala $[0.66, 0.67]$! Če pogledamo izračun, vidimo, da je bil glavni krivec za izgubo natančnosti to, da smo zašli v prevelika števila. Pri predstavitvi 1.33 smo lahko obdržali le dve mesti in smo bili prisiljeni zadnjo zavreči, da smo shranili 1.3.

Izračun bi lahko popravili tako, da bi ga namesto $\frac{a+b}{2}$ napisali kot $a + \frac{b-a}{2}$. V tem primeru ne bi prišlo

do dela s tako velikimi števili in rezultat bi bil

$$\begin{aligned} a + \frac{b-a}{2} &= 0.66 + \frac{1}{2}(0.67 - 0.66) \\ &= 0.66 + \frac{1}{2}\,\mathrm{fl}(0.01) \\ &= 0.66 + \frac{1}{2}(0.1 \cdot 10^{-1}) \\ &= 0.66 + \mathrm{fl}(0.05 \cdot 10^{-1}) \\ &= 0.66 + 0.5 \cdot 10^{-2} = \mathrm{fl}(0.665) = 0.66. \end{aligned}$$

V tem primeru je rezultat do zadnjega koraka kljub vmesnim zaokrožitvam točen, na koncu pa se zaokroži na najboljši možni približek.

Pokažimo še nekaj zanimivih primerov dejanske dvojne natančnosti. V izbranem programskem jeziku (npr. Python) izračunajmo $s = 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1$. Če bi bilo računanje točno, bi rezultat moral biti enak 1. Vendar, ko preverimo `s == 1`, dobimo odgovor `False`, da enakost ne drži. Res, če izpišemo vrednost s na dovolj decimalk, dobimo $s = 0.9999999999999988898$, oz. razliko $|s - 1| = 1.110223 \cdot 10^{-16}$. Če bi pri preverjanju uporabili toleranco npr. 10^{-15}, bi dobili pravilni rezultat, da sta odgovora enaka.

Ponovimo eksperiment z 0.5 namesto 0.1. Rezultat bi v tem primeru moral biti 5, in vrednost s je 5.0000000000000000000 ter primerjava `s == 5` vrne `True`. Zakaj tokrat nismo dobili 4.99999999999 999999 namesto 5? Odgovor je, da so bila vsa števila, ki so v računu nastopala, torej 0, 0.5, 1.0, 1.5, 2.0, 2.5 itd. . . , predstavljiva in nikoli ni prišlo do zaokroževanja. Tako je bil tudi rezultat točen. Če pazljivo izbiramo števila, lahko sicer konstruiramo primere, kot je zgornji, kjer napak ni. Vendar v praksi na kaj takega ne moremo računati – sprijazniti se je potrebno z napakami in znati z njimi ustrezno ravnati.

Posebne vrednosti

Prej smo omenili, da sta eksponenta -1023 in 1024 rezervirana za posebne vrednosti. To je potrebno, ker poleg decimalnih števil standard vsebuje tudi posebna števila $\pm\infty$ in NaN, ki predstavljajo (pozitivno ali negativno) neskočnost, in pa posebno število »not a number«, ki označuje, da je rezultat neveljaven ali nedoločen. Najlažji način, da dobimo vrednost ∞, ki ga običajno napišemo z besedilom kot »`inf`«, je, da poskusimo izračunati npr. `1.0 / 0.0`. Večina programskih jezikov (z izjemo Pythona) bo vrnila vrednost `inf`. Za računanje z neskončnostjo veljajo naslednja pravila:

- $\infty + a = \infty$
  $\infty - a = \infty$

$$\infty \cdot a = \begin{cases} \infty; & \text{če } a > 0 \\ \mathrm{NaN}; & \text{če } a = 0 \\ -\infty; & \text{če } a < 0 \end{cases}$$

$$\infty / a = \begin{cases} \infty; & \text{če } a \geqslant 0 \\ -\infty; & \text{če } a < 0 \end{cases}$$

$$a^{\infty} = \begin{cases} 0; & \text{če } |a| < 1 \\ 1; & \text{če } |a| = 1 \\ \infty; & \text{če } |a| > 1 \end{cases}$$

$$\infty^{a} = \begin{cases} 0; & \text{če } a < 0 \\ 1; & \text{če } a = 0 \\ \infty; & \text{če } a > 0 \end{cases}$$

Pri tem smo predpostavili, da je a običajno število (torej ni ∞ ali NaN). Definirano pa je npr. tudi $\infty + \infty = \infty$ in $\infty \cdot \infty = \infty$. Definicije so smiselne z vidika matematičnih limit: če imamo dve zaporedij števil, ki gresta proti ∞, gre proti ∞ tudi njun produkt ali vsota. Za razliko ali kvocient pa to ne velja, zato so rezultati izrazov ∞/∞, $\infty - \infty$, $0/0$ ali $0 \cdot \infty$ enaki NaN. Računanje z NaN pa je enostavno definirati: čim je eden od operandov enak NaN, je tudi rezultat NaN: torej $a + \mathrm{NaN} = \mathrm{NaN}$, $a \cdot \mathrm{NaN} = \mathrm{NaN}$, $a^{\mathrm{NaN}} = \mathrm{NaN}$, . . . S posebnimi vrednostmi pa znajo delati tudi elementarne funkcije. Velja npr. $\sqrt{\infty} = \infty$, $\sqrt{-2} = \mathrm{NaN}$, $\sin(\infty) = \mathrm{NaN}$, $\arctan(\infty) = \pi/2$.

Za NaN pa velja še ena posebnost: če ga primerjamo samega s sabo, vrne `False`. Torej, če npr. naredimo `x = 0.0/0.0` in nato izračunamo `x == x`, dobimo `False`. Ta posebna lastnost velja samo za NaN in ne drži za nobeno drugo decimalno število. Kot smo videli, je to le ena izmed pasti, v katero se lahko ujamemo pri delu z decimalnimi števili. S tem, kako računati, da se v naše izračune ne prikrade preveč napak, se ukvarja *numerična matematika*. Ta tudi neveščemu računalničarju omogoča zanesljivo delo z decimalnimi števili.

× × ×