Presné funkcie - Zápočtový program na Programování II

Michal Szabados

2. augusta 2008

Obsah

1 Špecifikácia

Program umožní vypísať hodnotu funkcií e^x a $\sin(x)$ pre zadané x. Navyše si užívateľ bude môcť zvoliť číselnú sústavu, v ktorej sa výsledok vypíše a taktiež aj počet desatinných miest. Do programu sa budú dať ľahko doplniť ďalšie funkcie.

2 Analýza

2.1 Vstup

Účelom programu je vypisovať istý výraz s ľubovoľnou presnosťou. Preto je nutné, aby užívateľov vstup bol presný. Z toho dôvodu na vstupe od užívateľa parameter dosadzovaný do zvolenej funkcie je buď číslo v desiatkovej sústave s konečným desatinným rozvojom alebo zlomok.

Vnútorne sa vstup prevedie na zlomok a Euklidovým algoritmom upraví na základný tvar tak, aby bol menovateľ nezáporný.

2.2 Reprezentácia funkcií

Počítač dokáže s číslami robiť len pár operácií, ako napr. sčítanie alebo násobenie. Ako pomocou nich môžeme reprezentovať funkcie?

Dobrú odpoveď nám dáva matematická analýza s teóriou Taylorovho polynómu. Tá tvrdí, že isté funkcie sa dajú zapísať v tvare

$$f(x) = f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f^{(2)}(a)}{2!}(x-a)^2 + \cdots + \frac{f^{(n)}(a)}{n!}(x-a)^n + o((x-a)^n).$$

Pre nás to prakticky znamená, že funkciu dokážeme rozvinúť v rad, pričom zvyšok bude „malý“ (viď ďalej). Rad je už súčet konečne mnoho zlomkov, s ktorým sa dá pracovať.

2.3 Vyčíslenie Taylorovho radu

Problém sme si teda previedli do nasledujúceho tvaru.

Užívateľ nám zadá hodotu $x = p/q$, číselný základ b a požadovaný počet desatinných miest d. Našou úlohou je potom vyčísliť na výstup výraz

$$k_0 + k_1\frac{x}{1!} + k_2\frac{x^2}{2!} + \cdots + k_n\frac{x^n}{n!},$$

kde k_i sú koeficienty Taylorovho rozvoja.

Program samozrejme na základe počtu desatinných miest d musí určiť, aký dlhý rozvoj musíme zobrať. Túto podúlohu preberieme v kapitole Funkcie. Pre túto kapitolu však predpokladajme, že potrebnú dĺžku rozvoja poznáme.

V nasledujúcich podkapitolách sa pomocou riešenia jednoduchších problémov dopracujeme k tomu, ako mocninný rozvoj vyčísliť.

2.3.1 Vyčíslenie zlomku

Možnosť vypočítať zlomok priamo vydelením do premennej typu `double` nám nevyhovuje. `double` má totiž obmedzenú presnosť a pri požiadavke užívateľa vypísať výsledok na už len 100 desatinných miest by zlyhal.

Preto použijeme nasledovné dve myšlienky:

- Každý (nezáporný) zlomok sa dá napísať v tvare $r + s/t$, kde r je celé číslo a s/t je zlomok menší než 1.

- Desatinný rozvoj zlomku môžeme získavať postupne.[1] N-tá cifra desatinného rozvoja je totiž zvyšok po delení b čísla $x \cdot b^n$.

Takže ak máme vyčísliť zlomok menší než 1, stačí ho vynásobiť číselným základom, vypísať celú časť, odpočítať ju a napokon rekurzívne dať vypísať zlomok, ktorý nám vznikol na konci. Pre ilustráciu uvádzam algoritmus výpočtu $9/7 \approx 1.2857\ldots$, svorky označujú práve spracúvaný zlomok.

$$\underbrace{\frac{9}{7}} = 1+ \underbrace{\frac{2}{7}} = 1+ \underbrace{\frac{20}{7}}\,\frac{1}{10} = 1+\frac{2}{10}+ \underbrace{\frac{6}{7}}\,\frac{1}{10} = 1+\frac{2}{10}+ \underbrace{\frac{60}{7}}\,\frac{1}{100} = 1+\frac{2}{10}+\frac{8}{100}+ \underbrace{\frac{4}{7}}\,\frac{1}{100}$$

$$\begin{aligned}
\frac{9}{7} &= 1+\frac{2}{7} && \to 1\\
\frac{20}{7} &= 2+\frac{6}{7} && \to 2\\
\frac{60}{7} &= 8+\frac{4}{7} && \to 8\\
\frac{a}{b} &= (a \text{ div } b)+\frac{(a \bmod b)}{b}
\end{aligned}$$

Môžme si všimnúť, že zakaždým, až na prvý krok, musí byť vypísaná cifra menšia než číselný základ b (v tomto prípade 10). Takže ak by sme nemuseli vypisovať celú časť výsledku, vystačili by sme si s normálnymi celými číslami. Avšak veľkosť celej časti nevieme ovplyvniť a už pri celkom malých parametroch môže byť príliš veľká (napr. e^{50}), preto sa nevyhneme použitiu dlhých čísel.

2.3.2 Vyčíslenie e

Tento odstavec vychádza z článku S. Rabinowitza a S. Wagona [1].

Tentokrát máme zadaný rozvoj čísla $e = 1 + \frac{1}{1!} + \frac{1}{2!} + \ldots + \frac{1}{n!}$ pre vopred dané n. Nápad previesť rozvoj na zlomok, ktorý už vieme vyčísliť, znova zlyhá – faktoriál rastie príliš rýchlo a menovateľ sa nám nezmestí do premennej.

Riešenie využíva rovnaký postreh ako pri vyčísľovaní zlomkov, akurát s tým rozdielom, že si v každom kroku pamätáme celý rozvoj:

$$\begin{aligned}
1+1+\frac{1}{2!}+\frac{1}{3!}+\frac{1}{4!}+\frac{1}{5!}+\frac{1}{6!} &&&\to 2\\
\frac{10}{2!}+\frac{10}{3!}+\frac{10}{4!}+\frac{10}{5!}+\frac{10}{6!} &= 7+\frac{0}{2!}+\frac{1}{3!}+\frac{0}{4!}+\frac{1}{5!}+\frac{4}{6!} &&\to 7\\
\frac{0}{2!}+\frac{10}{3!}+\frac{0}{4!}+\frac{10}{5!}+\frac{40}{6!} &= 1+\frac{1}{2!}+\frac{1}{3!}+\frac{3}{4!}+\frac{1}{5!}+\frac{4}{6!} &&\to 1
\end{aligned}$$

Kľúčom k pochopeniu algoritmu je pochopiť prvú rovnosť. Presnejšie postup, ako vypočítať koeficienty pri jednotlivých faktoriáloch. Postupuje sa smerom sprava. $10/6!$ môžme napísať ako

$$\frac{10}{6!} = \frac{6}{6!} + \frac{4}{6!} = \frac{1}{5!} + \frac{4}{6!}.$$

[1] Istý program na výpočet čísla π na viacerých počítačoch naraz dokonca dokáže vypočítať n-tú cifru rozvoja bez znalosti predošlých cifier.

Teda výraz na ľavej strane sa nezmení, ak namiesto 10 bude pri 6! koeficient 4 a pri 5! bude koeficient o 1 väčší, teda 11.

Takto ale môžeme pokračovať ďalej – $11/5! = 10/5! + 1/5!$ = $2/4! + 1/5!$ atď., až kým nad každým faktoriálom $n!$ bude číslo menšie než n a pred celý rad nedostaneme celú časť.

Keďže tento algoritmus je veľmi dôležitý, ukážeme si ešte jeden pohľad na vec, ktorý sa nám bude hodiť aj neskôr. Všimnime si, že rozvoj čísla e vieme zapísať aj v zlomkovom tvare:

$$1+1+\frac{1}{2!}+\frac{1}{3!}+\frac{1}{4!}+\frac{1}{5!}+\frac{1}{6!} \quad = \quad 1+1+\frac{1+\frac{1+\frac{1+\frac{1}{6}}{5}}{4}}{3}}{2}$$

Celá časť je teda 2 a ideme násobiť 10, aby sme získali ďalšiu cifru desatinného rozvoja. Pri tom sa nám vďaka analógii s predošlým postupom vynásobia všetky jednotky 10. Teraz je úplne prirodzené upraviť zlomky tak, aby neobsahovali celé časti – a tým dostaneme presne ten postup, ktorý sme používali v predošlom spôsobe.

$$\frac{10+\frac{10+\frac{10+\frac{10}{6}}{5}}{4}}{3}}{2} = \frac{10+\frac{10+\frac{11+\frac{4}{6}}{5}}{4}}{3}}{2} = \frac{10+\frac{10+\frac{12+\frac{1+\frac{4}{6}}{5}}{4}}{3}}{2} = \cdots = 7+\frac{0+\frac{1+\frac{0+\frac{1+\frac{4}{6}}{5}}{4}}{3}}{2}$$

Znovu pri úprave zlomku v čitateli necháme zvyšok po delení a pred zlomok vyjde celočíselný podiel. Preto koeficient v čitateli každého zlomku bude vždy menší než číslo v menovateli.[2]

Z matematického hľadiska, aby nám postup fungoval, musí vždy platiť, že „celá časť“, ktorú dostávame, je naozaj celou časťou – teda že zvyšok (resp. zvyšný zlomok v druhom pohľade) je vždy menší než 1. To našťastie vždy platí, pre dôkaz viď prílohu A. Len poznamenám, že v úplnom algoritme budeme potrebovať vedieť, že platí o niečo silnejšie tvrdenie.

2.3.3 Zlomkové mocniny e

Rozšíriť predchádzajúci algoritmus na celé čísla je jednoduché – stačí, aby sa pri inicializácii namiesto koeficientu 1 dosadzovali skutočné Taylorove koeficienty tvaru x^n. Vzniknú tým síce znovu problémy s pretekaním a so zápornými koeficientami, tie však vyriešime neskôr. Nateraz nám bude stačiť, že vieme vyčísliť mocninný rozvoj vtedy, keď koeficient nad $n!$ je celý, nezáporný a menší než n.

Chceli by sme rozšíriť náš algoritmus pre zlomky. Teda pre daný parameter $x = p/q$ a danú dĺžku rozvoja (napr. 6) potrebujeme vyčísliť

$$1+\frac{p}{q}+\frac{p^2}{q^2 2!}+\frac{p^3}{q^3 3!}+\frac{p^4}{q^4 4!}+\frac{p^5}{q^5 5!} = 1+\frac{p+\frac{p^2+\frac{p^3+\frac{p^4+\frac{p^5}{5q}}{4q}}{3q}}{2q}}{q}$$

Uvedený šikovný zápis nás navádza na správnu myšlienku „zase robiť to isté“ – tentokrát však v menovateľoch nemáme predom zadané čísla 1 až 5, ale čísla závislé na parametri: q až $5q$. Teda znovu budeme upravovať zlomky smerom „sprava“ (resp. zhora) tak, aby sme vždy vyňali celú časť.

Na tomto mieste by matematika znovu mohlo zaujímať, či po každom kroku bude vyššie uvedený zlomok menší než jeden. Áno, naozaj bude, pre dôkaz viď prílohu A.

[2] Zložený zlomok sa nám sladá z viacerych zlomkov, pričom každý z nich má v čitateli číslo a menší zložený zlomok. V menovateli je zakaždým len jedno číslo.

2.3.4 Algoritmus výpisu a terminológia

Zhrňme si teda, čo doteraz vieme. Keď zovšeobecníme našu úlohu naspäť na ľubovoľnú funkciu, vieme vyčísliť výraz

$$k_0 + k_1\frac{x}{1!} + k_2\frac{x^2}{2!} + \cdots + k_n\frac{x^n}{n!} = k_0 + \frac{k_1 p}{1!q} + \frac{k_2 p^2}{2!q^2} + \cdots + \frac{k_n p^n}{n!q^n} =$$

$$= k_0 + \frac{k_1 p + \frac{k_2 p^2 + \cdots^{+\frac{k_n p^n}{nq}}}{2q}}{q} = k_0' + \frac{k_1' + \frac{k_2' + \cdots^{+\frac{k_n'}{nq}}}{2q}}{q}$$

za nasledujúcej podmienky:

Podmienka 1 *Každý z koeficientov k_i' $(1 \leq i \leq n)$ je nezáporné celé číslo menšie než iq.*

Rád by som sa vyjadril k názvosloviu. Algoritmus je postavený na tzv. „spigot algoritme“ podľa [1], takže ho tiež budem nazývať *spigot(ový) algoritmus*. Pritom rozvoj funkcie – pole čísel $k_0', k_1', \ldots, k_n'$ nazývame *spigotom*.[3] Spigot nazveme *normálny*, pokiaľ bude spĺňať Podmienku 1. A napokon *sčesaním* spigotu nazveme proces úpravy spigotu, keď zvyšky z jednotlivých zlomkov sú prenášané viac a viac doľava.

Zistime náročnosti programu. Pamäťová je zrejme $O(n)$, pretože si musíme pamätať všetky koeficienty. Z časového hľadiska pri každom zistení cifry desatinného rozvoja potrebujene celý spigot prenásobiť číselnou bázou – na to potrebujeme čas $O(n)$. Navyše ho potrebujeme upraviť naspäť do normálneho tvaru (sčesať), čo vieme spraviť na jeden prechod „sprava“, t.j. v čase $O(n)$. Ak teda označíme d počet vypisovaných „desatinných“ miest výsledku, celková časová zložitosť je $O(dn)$. Ako neskôr zistíme, d bude pri $\exp(x)$ a $\sin(x)$ rádu $O(n)$, takže zložitosť bude $O(n^2)$.

V nasledujúcich kapitolách sa budeme zaoberať tým, ako na začiatku z čísel $k_i p^n$ dostať čísla k_i', čiže upraviť spigot tak, aby spĺňal Podmienku 1.

2.4 Inicializácia

Už vieme vyčísliť spigot keď je normálny. Našou úlohou teda zostáva zostaviť ho z parametrov užívateľa.

Priblížme si, v čom je problém. Keď máme do spigotu dosadené, dá sa ho normalizovať tým, že ho raz „prečešeme“ sprava doľava. Pri Taylorovom rozvoji však dosadzované členy rastú exponenciálne a ľahko sa nám stane, že sa nám nezmestia do premennej. Preto potrebujeme nájsť taký algoritmus, ktorý sa bude exponenciálnemu rastu vyhýbať. Pre ilustráciu spigot $e^{\frac{100}{7}}$ dĺžky 6:

$$1 + \frac{100}{7} + \frac{100^2}{7^2 2!} + \frac{100^3}{7^3 3!} + \frac{100^4}{7^4 4!} + \frac{100^5}{7^5 5!} + \frac{100^6}{7^6 6!} = 1 + \frac{100 + \frac{100^2 + \frac{100^3 + \frac{100^4 + \frac{100^5 + \frac{100^6}{42}}{35}}{28}}{21}}{14}}{7}$$

[3] spigot [en] – *čap, zátka, kohútik*. Zvyšky „stekajú“ dole spigotom aby nakoniec mohla „odkvapnúť“ ďalšia cifra desatinného rozvoja.

2.4.1 Riešenie

Preveďme následnú úpravu predošlého výrazu:

$$\begin{aligned} &= 1+100(\frac{1}{7}+100(\frac{1}{7^2 2!}+100(\frac{1}{7^3 3!}+100(\frac{1}{7^4 4!}+100(\frac{1}{7^5 5!}+100(\frac{1}{7^6 6!}))))))\\ &= 1+\frac{100}{7}(1+\frac{100}{2\cdot 7}(1+\frac{100}{3\cdot 7}(1+\frac{100}{4\cdot 7}(1+\frac{100}{5\cdot 7}(1+\frac{100}{6\cdot 7}))))) \end{aligned}$$

v zlomkovom tvare si ukážme všeobecný tvar:

$$= k_0 + p\frac{k_1 + p\frac{k_2 + \cdot^{\cdot^{\cdot^{+p\frac{k_n}{nq}}}}}{2q}}{q}$$

Vidíme, že myšlienka je rovnaká ako pri Hornerovej schéme – čitateľ môžeme vyňať z celého zvyšku keďže delí každý člen.

Tým už máme návod na algoritmus. Na začiatku inicializujeme spigot číslami k_i. Potom budeme jeho koniec postupne násobiť p, pričom ho pri každom násobení sčešeme. Ukážme si beh algoritmu pri výpočte e^3 s dĺžkou spigotu 5 (prvý riadok v bloku je násobenie, druhý sčesanie):

$\frac{1}{0!}$	$\frac{1}{1!}$	$\frac{1}{2!}$	$\frac{1}{3!}$	$\frac{1}{4!}$	$\frac{1}{5!}$
1	1	1	1	1	1
1	1	1	1	1	3
1	1	1	1	1	3
1	1	1	1	3	9
1	1	1	2	0	4
1	1	1	6	0	12
1	1	**3**	0	2	2
1	1	9	0	6	6
1	**5**	1	1	3	1
1	15	3	3	9	3
18	0	0	2	1	3

Tento algorimus má však jeden skrytý problém. Síce sa vyhneme veľkým číslam na práve spracúvanom úseku, ale na pozícii tesne pred ním nám budú vznikať veľké čísla[4] (zvýraznené čísla v tabuľke). Veľkému číslu na prvej pozícii sa však nevyhneme, pretože veľkosť celej časti zlomku nevieme ovplyvniť.

Na riešenie nás navedie myšlienka sčesávať spigot až úplne doľava. Vo vyššie uvedenom algoritme to však nie je možné, pretože v každom kroku potrebujeme vynásobiť aj to „veľké číslo“ koeficientom p. Takže ak by sme sčesávali úplne, museli by sme násobiť celý spigot. Tým pádom nemôžeme mať v ňom na začiatku inicializované koeficienty k_i (v prípade e^3 jednotky), pretože tie potrebujeme násobiť až od nejakého okamihu. Takže ich budeme pripočítavať v tom správnom momente:

[4]Pri implementácii tohto algoritmu mi dochádzalo stále k pretekaniu.

celá časť	$\frac{1}{0!}$	$\frac{1}{1!}$	$\frac{1}{2!}$	$\frac{1}{3!}$	$\frac{1}{4!}$	$\frac{1}{5!}$
0	0	0	0	0	0	$0+1$
0	0	0	0	0	0	1
0	0	0	0	0	$0+1$	3
0	0	0	0	0	1	3
0	0	0	0	$0+1$	3	9
0	0	0	0	2	0	4
0	0	0	$0+1$	6	0	12
$0+1$	1	0	1	0	2	2
3	0	$0+1$	3	0	6	6
$3+2$	2	0	1	1	3	1
15	$0+1$	0	3	3	9	3
$15+3$	3	0	0	2	1	3
18	0	0	0	2	1	3

V tabuľke som už naznačil prácu s celou časťou. Tá musí byť uložená ako dlhé číslo, takže nemôže byť súčasťou spigotu. Hoci plní rovnakú úlohu ako koeficient pri 1/0!, nechávam v spigote aj tento koeficient kvôli kompatibilite. Pritom v každom kroku jeho obsah prenášam do celej časti.

2.4.2 Záporné čísla

Zatiaľ sme sa venovali len prípadu, keď čísla v spigote sú nezáporné. K úplnému dokončeniu algoritmu nám chýba doriešiť prípad, keď užívateľ chce vypočítať hodnotu funkcie v zápornom bode alebo keď koeficienty Taylorovho rozvoja sú záporné. Znova pritom ale chceme, aby po inicializácii nám vznikol nezáporný normalizovaný zlomok, ktorý bežným spôsobom vyčíslime.

Pozrime sa, čo sa stane, ak do spigotu dosadíme na začiatku záporné čísla. V prvom rade musíme dodefinovať operáciu „celočíselné delenie“ a „zvyšok po delení“. To spravíme prirodzeným spôsobom, n div k definujeme ako *najväčšie číslo menšie než n deliteľné k vydelené k*. Potom n mod k bude rozdiel $n - k(n \text{ div } k)$. Pritom k pripúšťame stále len kladné.

$$-5 \text{ div } 3 = -2 \qquad -5 \bmod 3 = 1 \qquad -10 \text{ div } 7 = -2 \qquad -10 \bmod 7 = 4$$

To, že pripúšťame len kladné k nám stačí, pretože členy, ktorými delíme, sú vždy násobky menovateľa q, ktorý nezáporný je.

Naša definícia má dve podstatné vlastnosti. Jednak rozširuje klasickú definíciu celočíselného delenia a zvyšku po delení a navyše vždy platí, že **zvyšok po delení je nezáporný**. To má okamžitý dôsledok v tom, že pri sčesávaní nám vznikajú nezáporné koeficienty, takže po inicializácii popísanej v predchádzajúcej kapitole dostaneme číslo tvaru

$$c + z,$$

kde c je celá časť a z je zlomok uložený v spigote, ktorý ale bude v normálnom tvare.

Môžu nastať dva prípady. Ak je c nezáporné, všetko je v poriadku. c vypíšeme a z môžeme vyčísľovať normálnym spôsobom.

Prípad, keď je c záporné, ilustrujme na príklade $c + z = -3.14$. Keďže z je z intervalu $[0, 1)$, musí platiť $c = -4$ a $z = 0.86$. Zrejme ako celú časť potrebujeme vypísať $c+1$. Čo však so z? Spravíme nasledovný trik. Vynásobime z číslom -1. z je spigot, takže po tejto úprave bude pri 1/0! koeficient -1 a vo zvyšku zlomku bude číslo 0.14. Teda stačí nám tú -1 zahodiť a sme hotoví.

2.4.3 Zložitosť

Z pamäťového hľadiska znovu potrebujeme pamäť len na spigot, takže zložitosť je $O(n)$.

Z časového hľadiska potrebujeme n-krát prenásobiť a sčesať spigot, čo nám dáva celkovú zložitosť $O(n^2)$. Môžeme si všimnúť, že algoritmus je takmer rovnaký ako pri vyčísľovaní spigotu. Rozdiel je akurát v tom, že namiesto násobenia číselnou bázou násobime čitateľom zadaného parametru a ešte navyše v správny moment pričítame koeficienty k_i.

2.5 Funkcie

Užívateľ bude mať možnosť zvoliť si funkciu, ktorú chce vyčísliť. Preto pre každú funkciu v ponuke musíme pripraviť dve veci:

1. koeficienty Taylorovho rozvoja,
2. vypočítať dĺžku spigotu v závisloti na vstupe užívateľa.

Pre bod 1. nám stačí len zvoliť vhodný bod, v ktorom spravíme Taylorov rozvoj.

Bod 2. je však zložitejší. Dĺžka rozvoja určite bude závisieť na požadovanom počte desatinných miest. Ujasnime si ale, čo chápeme pod pojmom „presnosť na d desatinných miest“. V našom ponímaní to bude znamenať, že *presná hodnota sa od vypísanej líši o menej než* b^{-d}, kde b je číselná báza. [5]

Význam pojmu si ukážme pre ilustráciu na výpise Eulerovho čísla $e \approx 2.71828$ v desiatkovej sústave s presnosťou na dve desatinné miesta. Výsledok sa má od skutočnej hodnoty líšiť maximálne o $10^{-2} = 0.01$. Teda prípustné hodnoty sú napr. 2.71, 2.72, 2.718 či 2.709. Budeme ale vypisovať len dve desatinné cifry, takže jediné dva možné výsledky sú 2.71 a 2.72. [6]

Potrebujeme teda dosiahnuť, aby zvyšok Taylorovho rozvoja (t.j. časť, ktorú zahodíme) bol v absolútnej hodnote menší než b^{-d}. Aby to vôbec bolo možné, musíme vedieť, že Taylorov rozvoj konverguje na celom $\mathbb{R}$. Našťastie z matematickej analýzy vieme, že pre funkcie $\exp(x)$ a $\sin(x)$ tomu tak je. Už zostáva len odhadnúť zvyšok. Cauchyho alebo Lagrangeov tvar zvyšku nám však nepomôže, pretože v jeho vyjadrení používame funkciu, ktorú sa snažíme vyčísliť a navyše závisí exponenciálne na vzdialenosti vyčísľovaného bodu od bodu, v ktorom robíme rozvoj. Preto budeme musieť použiť primitívnejšie techniky.

[5] Keďže báza nemusí byť desiatková, pojem „destinný rozvoj“ je trochu zavádzajúci. Žiaľ neviem o žiadnom vhodnejšom termíne.

[6] Vyvstáva otázka, ktorú hodnotu z týchto dvoch vypíše náš program. Bude to hodnota 2.71 napriek tomu, že hodnota 2.72 je presnej hodnote bližšie. Náš algoritmus nám totiž dáva postupne cifry výsledku. Ak by sme chceli druhý výsledok, museli by sme teda zaokrúhľovať, čo je vo fyzikálnej praxi bežné, ale z matematického hľadiska je to nepekná operácia (môj subjektívny názor) a znateľne by to predĺžilo program.

Poznamenám však, že ak by sme chceli pripúšťať len „fyzikálny“ výsledok, pri definícii presnosti by sme volili toleranciu $b^{-d}/2$.

2.5.1 exp(x)

Taylorov rozvoj funkcie $\exp(x)$ budeme robiť v bode $a = 0$, pretože vyjde jednoduchý tvar

$$exp(x) = 1 + x + \frac{x^2}{2!} + \cdots + \frac{x^n}{n!} + z_n(x).$$

Na odhad zvyšku sa pustíme do hlbšej analýzy.

Absolútnu hodnotu argumentu x vieme určite zhora odhadnúť nejakým číslom m. Keďže zrejme platia nerovnosti $\frac{x^k}{k!} \leq \frac{m^k}{k!}$, stačí nám nájsť potrebnú dĺžku rozvoja pre m.

K-tý člen rozvoja vieme tiež napísať ako

$$\frac{m^k}{k!} = \frac{m}{1} \cdot \frac{m}{2} \cdots \frac{m}{k}.$$

Od nejakého členu k_0 teda bude platiť, že $\frac{m}{k} < \frac{m}{k_0} = q < 1$. Takže pre $k > k_0$ dostávame

$$\frac{m^{k+1}}{(k+1)!} = \frac{m}{k+1} \cdot \frac{m^k}{k!} < q\frac{m^k}{k!}$$

a z toho už ľahko indukciou

$$\frac{m^k}{k!} < q^{k-k_0}\frac{m^{k_0}}{k_0!} \quad \Longrightarrow \quad z_{k_0}(x) = \sum_{k=k_0+1}^{\infty} \frac{m^k}{k!} < \sum_{k=k_0+1}^{\infty} q^{k-k_0}\frac{m_{k_0}}{k_0!} = \frac{m^{k_0}}{k_0!}\frac{q}{1-q}.$$

Pre $q \leq 1/3$ navyše platí $q/(1-q) \leq 1/2$, takže dostávame odhad zvyšku $z_{k_0}(x)$ pomocou posledného členu:

$$z_{k_0}(x) < \frac{1}{2} \cdot \frac{m^{k_0}}{k_0!} \quad \text{pre } k_0 \geq 3m$$

Toto je veľmi príjemné zistenie, pretože nám stačí odhadnúť len členy rozvoja. To spravíme nasledovným spôsobom. Zo známej nerovnosti $\left(\frac{n}{3}\right)^n < n!$ dostávame

$$\frac{m^k}{k!} < \frac{m^k}{\left(\frac{k}{3}\right)^k} = \left(\frac{3m}{k}\right)^k < 1 \qquad \text{pre } k \geq 3m.$$

Takže môžeme navhnúť nasledujúci algoritmus. Ako počiatočnú dĺžku rozvoja zvolíme $l = \lceil 3|x| \rceil$. Vieme, že $\frac{|x|^l}{l!} < 1$. Tiež vieme odhadnúť zhora mocninou dvojky činiteľ, ktorým musíme vynásobiť aktuálny člen, aby sme dostali ten ďalší. Tým vieme zhora odhadnúť mocninou dvojky ďalší člen a tento postup môžeme opakovať, kým nedostaneme požadovanú presnosť.

To, že sme odhadovali mocninou dvojky, nám umožňuje pamätať si iba dvojkový logaritmus aktuálneho člena. Tým sa zjednoduší aj odhad nasledujúceho, keďže namiesto násobenia používame sčítanie.

Odhadnime ešte, rádovo aký dlhý rozvoj musíme zobrať pri presnosti na d desatinných miest. Minimálne $l = \lceil 3|x| \rceil$, čo je $O(|x|)$. Ďalej s každým ďalším členom získame presnosť na aspoň jednu dvojkovú cifru, čo je $O(d)$, takže celkovo vyjde dĺžka $O(|x| + d)$.

2.5.2 sin(x)

Taylorov rozvoj tejto funkcie budeme robiť rovnako v bode $a = 0$, pretože znovu vyjde jednoduchý rozvoj:

$$sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots + (-1)^n \frac{x^{2n+1}}{(2n+1)!} + z_n(x).$$

Tentokrát však vieme, že absolútna veľkosť zvyšku pri sínuse je určite menšia, než veľkosť zvyšku pri exponenciále (sínus obsahuje len vybrané členy rozvoja exponenciály). Takže stačí nám jednoducho zobrať rovnako dlhý rozvoj, ako pri exponenciále.

3 Užívateľská dokumentácia

Presné funkcie je program na výpočet hodnoty funkcií $\exp(x)$ a $\sin(x)$ pre zadané x s presnosťou na ľubovoľný počet desatinných miest v zvolenej číselnej sústave.

Program má nasledujúcu syntax:

```
spigot.exe [funkcia [argument [počet_des_miest [číselná_sústava]]]] [-d]
```

kde parametre v hranatých zátvorkách sú nepovinné.

funkcia	Určuje funkciu. Pokiaľ parameter nie je zadaný, program si ho vypýta. Prípustné hodnoty sú: `1`, `e` exponenciála `2`, `sin` sínus
argument	Argument dosadzovaný do zvolenej fukcie. Pokiaľ parameter nie je zadaný, program si ho vypýta. Musí byť v tvare zlomku alebo desatinného čísla s desatinnou bodkou, prípadne mu môže predchádzať znamienko mínus.
počet_des_miest	Počet miest za (desatinnou) bodkou vo výsledku. Pokiaľ parameter nie je zadaný, východzia hodnota je 32.
číselná_sústava	Číselná sústava, v ktorej je vypísaný výsledok. Musí to byť číslo medzi 2 a 36 vrátane. Pokiaľ parameter nie je zadaný, východzia hodnota je 10.
-d	V prípade, že je parameter uvedený, program vypisuje tzv. „debugovacie“ výpisy – informácie o behu programu.

3.1 Ukážka programu

Pomocou vzorca $e = \exp(1)$ vypočítame hodnotu Eulerovho čísla:

```
: spigot.exe e 1
2.71828182845904523536028747135266
```

Vypočítame hodnotu $\sin(3.14)$ s presnosťou na 100 desatinných miest. Necháme si pritom vypísať ladiace výpisy. Môžeme vidieť, že vnútorne sa 3.14 prevedie na 157/50:

```
: spigot.exe sin 3.14 100 -d
DBG: Ratam 157/50.
Running...
DBG: Dlzka spigotu: 115.
0.00159265291648695254054143632444326144324052781902
26874184880508367128341969726816553665119281901636 6
```

Podľa vzorca $\sqrt{e} = \exp(1/2)$ vypočítame hodnotu odmocniny z e. Výsledok si necháme vypísať v dvojkovej sústave na 30 miest za bodkou. Pritom necháme program, aby si od nás parametre vypýtal:

```
: spigot.exe
Zvol si funkciu:
        1 - e^x
        2 - sin(x)
> 1
Zadaj argument (x):
> 1/2
Zvol pocet 'desatinnych' miest:
> 30
Zvol ciselnu sustavu:
> 2
1.10100110000100101001100011100 0
```

4 Programátorská dokumentácia

Program je napísaný v jazyku C. Skladá sa zo štyroch súborov:

global.h	Obsahuje globálne deklarácie programu.
funkcie.h	Interface na komunikáciu s modulom funkcie.c
funkcie.c	Modul obsahujúci funkcie exp a sin.
spigot.c	Hlavný súbor

Navyše program používa mierne upravenú knižnicu XySSL MPI na prácu s dlhými číslami:

bignum.c
xyssl\bignum.h
xyssl\bn_mul.h
xyssl\config.h

4.1 global.h

Definuje typ `LLINT`. Tiež definuje makrá `FOR` a `DOWNFOR` pre jednoduchšie používanie konštrukcie `for`.

4.2 funkcie.h

Obsahuje interface na komunikáciu s modulom `funkcie.c`.

4.3 funkcie.c

`LLINT lg(LLINT c)`

Horná celá časť dvojkového logaritmu z c. Pre $c \leq 0$ vracia -1.

`LLINT calculate_e_spigot_length(LLINT arg_numerator, LLINT arg_denominator, int num_base, LLINT dec_places)`

Vypočíta dĺžku spigotu potrebnú na dosiahnutie požadovanej presnosti pre funkciu exp.

`LLINT e_taylor_koeficient(LLINT term)`

Vráti *term*-ty člen Taylorovko rozvoja funkcie exp v bode 0.

`LLINT calculate_sin_spigot_length(LLINT arg_numerator, LLINT arg_denominator, int num_base, LLINT dec_places)`

Vypočíta dĺžku spigotu potrebnú na dosiahnutie požadovanej presnosti pre funkciu sin.

`LLINT sin_taylor_koeficient(LLINT term)`

Vráti *term*-ty člen Taylorovko rozvoja funkcie sin v bode 0.

4.4 spigot.c

4.4.1 Globálne premenné

```
LLINT *spigot;
LLINT sp_length;
```

Smerník na spigot a jeho dĺžka. *spigot* sa alokuje vo funkcii main.

```
LLINT arg_numerator, arg_denominator;
int num_base;
LLINT dec_places;
```

Vstupné parametre – čitateľ a menovateľ argumentu, číselný základ a počet vypisovaných „desatinných“ miest.

```
int function;
LLINT (* taylor_koeficient)(LLINT term);
```

$function$ má hodnotu `FN_E` alebo `FN_SIN` podľa toho, ktorá funkcia je vybraná. $taylor\_koeficient$ je smerník na funkciu vracajúcu *term*-ty koeficient Taylorovho rozvoja pre zvolenú funkciu. Nastavuje sa vo funkcii main.

```
int debug = 0;
```

Má hodnotu pravda alebo nepravda podľa toho, či sa majú vypisovať debugovacie výpisy. Východzia hodnota je nepravda.

4.4.2 Funkcie

`LLINT signmod(LLINT dividend, LLINT divisor)`

Funkcia modulo aj pre záporné delence *dividend*. Predpoklad: $divisor > 0$.

`LLINT signdiv(LLINT dividend, LLINT divisor)`

Funkcia celočíselné delenie aj pre záporné delence *dividend*. Predpoklad: $divisor > 0$.

`void multiply_spigot(LLINT number)`

Vynásobenie spigotu číslom *number*.

`void comb_spigot()`

Úprava spigotu do normálneho tvaru.

`mpi *ll_to_mpi(LLINT sour, mpi *dest)`

Konverzia `LLINT` do `mpi`. Vracia smerník na výsledok (t.j. *dest*) pre jednoduchšiu manipuláciu.

`void prepare_spigot()`

Dosadí do *spigot*u argument a vypíše celú časť výsledku.

`LLINT get_next_digit()`

Vráti ďalšiu cifru rozvoja v zvolenej číselnej sústave.

`void euclid_simplify(LLINT *nominator, LLINT *denumerator)`

Vykrátenie zlomku pomocou Euklidovho algoritmu.

`void read_point_fraction(char *buf, LLINT *numerator, LLINT *denominator)`

Načítanie zlomku v tvare s desatinnou bodkou z buf.

`int set_parameters(int argc, char *argv[])`

Načítanie a spracovanie parametrov zo vstupu.

`int main(int argc, char *argv[])`

Hlavná funkcia. Načíta parametre a spustí výpočet. Taktiež nechá vypísať výstup.

A Dôkaz správnosti algoritmu

Dokážem, že keď zlomok v spigote spĺňa Podmienku 1 (t.j. ak je normálny), tak je menší než 1. Všeobecne, formálne:

Veta 1 *Nech $n \in \mathbb{N}$; $k_i, m_i \in \mathbb{Z}_0^+$; $k_i < m_i$ pre $i \in \{1, \ldots, n\}$. Potom*

$$\frac{k_1 + \frac{k_2 + \cdot^{\cdot^{\cdot^{+\frac{k_n}{m_n}}}}}{m_2}}{m_1} < 1.$$

Zlomok v ktorom vystupuje m_i v menovateli označme z_i. Z definície teda

$$z_n = \frac{k_n}{m_n} \qquad z_i = \frac{k_i + z_{i+1}}{m_i} \text{ pre } i \in \{1, \ldots, n-1\}$$

Indukčne dokážem, že $z_i < 1$, čím bude dôkaz hotový, pretože náš zlomok je z_1.

1. $k_n < m_n$, takže $k_n/m_n = z_n < 1$.
2. Máme indukčný predpoklad $z_{i+1} < 1$. Keďže k_i aj m_i sú celé čísla a k_i je menšie, musí platiť $k_i + 1 \leq m_i$, z čoho vyplýva, že $k_i + z_{i+1} < m_i$. Takže $z_i = \frac{k_i + z_{i+1}}{m_i} < 1$ pre $i \in \{1, \ldots, n-1\}$, čím sme hotoví.

B Beh algoritmu pre sin(−5/3)

Ukážeme si výpočet sin(−5/3) s dĺžkou spigotu 6.

$$\sin(-5/3) \approx 0 + \frac{-5}{3} + 0 - \frac{(-5)^3}{3^3 3!} + 0 + \frac{(-5)^5}{3^5 5!}$$

Najprv prebehne inicializácia. Prvý riadok je násobenie -5 s pripočítavaním k_i, v druhom je podiel z celočíselného delenia „vyššieho“ zlomku a v treťom je zvyšok po delení.

celá časť	0	3	6	9	12	15
0	0	0	0	0	0	**1** + 0
	0	0	0	0	0	0
0		0	0	0	0	1
0	0	0	0	0	**0** + 0	−5
	−1	−1	−1	−1	−1	0
−1		2	5	8	11	10
5	0	−10	−25	**-1** − 40	−55	−50
	−6	−6	−6	−5	−4	0
−1		2	5	8	1	10
5	0	−10	**0** − 25	−40	−5	−50
	−5	−5	−5	−1	−4	0
0		0	0	4	3	10
0	0	**1** + 0	0	−20	−15	−50
	0	−1	−3	−2	−4	0
0		0	3	5	5	10
0	**0** + 0	0	−15	−25	−25	−50
	−2	−4	−4	−3	−4	0
−2		2	5	8	7	10
0	0	−2	−5	−8	−7	−10
	−1	−1	−1	−1	−1	0
−2	−1	0	0	0	4	5

Keďže v poslednom kroku vyšla celá časť záporná, vypíše sa číslo o 1 väčšie, než je uložené v celej časti (t.j. -1). Nakoniec sa spigot prenásobi -1. Momentálne teda vyzerá situácia v spigote nasledovne:

$$\frac{0 + \frac{0 + \frac{4 + \frac{5}{15}}{12}}{9}}{6}}{3}$$

Nasleduje výpis spigotu. Prvý riadok v bloku je násobenie číselnou bázou 10. V druhom je celočíselný podiel z vyššieho zlomku a v treťom zvyšok po delení.

0	3	6	9	12	15
	0	0	0	4	5
	0	0	0	40	50
0	0	0	3	3	0
	0	0	3	7	5
	0	0	30	70	50
0	0	4	6	3	0
	0	4	0	1	5
	0	40	0	10	50
2	6	0	1	3	0
	0	4	1	1	5
	0	40	10	10	50
2	6	1	1	3	0
	0	5	2	1	5
	0	50	20	10	50
2	8	2	1	3	0
	2	4	3	1	5
	20	40	30	10	50
9	7	3	1	3	0
	0	1	4	1	5

So šiestimi desatinnými miestami a dĺžkou spigotu 6 sme dostali výsledok -1.002229. Pre porovnanie, $\sin(-5/3) \approx -0.9954079577$.

Referencie

[1] Stanley Rabinowitz and Stan Wagon: *A Spigot Algorithm for the Digits of Pi.*