Софийски Университет “Св. Климент Охридски”
Факултет по математика и информатика
Катедра “Математическа логика и приложенията ѝ”

Дипломна работа

на тема

Един практически поглед върху изчислимостта на реални числа

на

Бранимир Здравков Ламбов

Специалност “Информатика”, ф. №42104

Научен ръководител:
проф. дмн. Д. Скордев

Ръководител катедра:
доц. дмн. И. Сосков

София,
Октомври 2001

Съдържание

Съдържание 2
1 Въведение 4
2 Теория 5
2.1 Изчислимост на реални числа 5
2.2 Примитивно рекурсивна изчислимост 6
2.3 Изчислимост на функции върху реални числа 9
2.4 Представяне (на число и грешка) 12
2.5 Пресмятане на елементарните функции (число и грешка) 12
2.5.1 Грешка при събиране 13
2.5.2 Грешка при умножение 13
2.5.3 Реципрочно 13
2.5.4 Квадратен корен 14
2.5.5 Логаритъм 15
2.5.6 Експонента 16
2.5.7 Константата π 17
2.5.8 Тригонометрични функции 17
2.6 Извеждане на резултат 17
3 Реализация 19
3.1 Плаваща запетая с произволна точност (*class LongFloat*) 19
3.2 Грешката (*class ErrorEst*) 20
3.3 Представяне на термове (*class RealObject*) 21
3.3.1 Видове обекти 22
3.3.2 Множествено обръщение към обект 23
3.3.3 Кеширане на пресмятанията 24
3.3.4 Ефективност на представянето 24
3.4 Потребителската страна (*class Real*) 24
3.4.1 Инициализация и приключване 25
3.4.2 Типът *Real* 25
3.4.2.1 Конструктори, деструктори, присвояване 25
3.4.2.2 Константи 26
3.4.2.3 Операции 26
3.4.2.4 Изискване на резултат 28

3.4.3 Пример на потребителска програма ... 28
3.5 Други избрани моменти в реализацията ... 30
3.5.1 Управление на блоковете с данни ... 30
3.5.2 Умножение чрез конволюция ... 31
3.5.3 Регулиране на дълбочината на спускане в рекурсия ... 32
3.5.4 Скриване на детайлите на реализацията ... 32
3.5.5 Преносимост на кода ... 33
4 Резултати ... 34
4.1 Пресмятане на константи и елементарни функции ... 34
4.2 Възможност за справяне с по-сложни задачи ... 34
4.2.1 Безпроблемно използване на шаблони ... 34
4.2.2 Конволюция на реални числа ... 35
5 Какво може да бъде добавено ... 36
Библиография ... 37

1 Въведение

Математическото понятие “реално число” не е представимо с изчислима функция в която и да е от множеството еквивалентни дефиниции за изчислимост. Няма как едно неизброимо поле да се представи в рамките на изброимо множество, каквото са изчислимите функции. В литературата съществуват и примери на реални числа, които не са представими с изчислима функция.

Въпреки това, множеството от изчислимите реални числа изглежда достатъчно за решаване на практическите проблеми. В повечето случай те се свеждат до някаква последователност от прилагане на елементарни функции върху рационални числа. И тъй като рационалните числа са лесно представими в света на изчислимите функции, а елементарните функции върху реални числа имат своето представяне в т. нар. Тип 2 Теория на Изчислимостта (Type-2 Theory of Effectivity, TTE, [15]), е разумно да се помисли за система за представяне на изчислими реални числа чрез термове, описващи начина за получаване на числото.

Реализации на подобни системи съществуват, въпреки че обикновено задачата, която те решават, е поставена по друг начин, като например “точно пресмятане на реални числа” [9] — като в тези случаи реалното число, което се пресмята, всъщност е представено с програмата, която използва въпросната система. Системата няма контрол върху числата, тя просто ги преобразува в смисъла на машините в TTE. За съжаление обикновено тя е ограничена конструктивно да използва за основа рационални числа и няма пълния потенциал на TTE машини, които биха могли да работят и върху неизчислими реални числа.

Въпросното ограничение изглежда разумно, защото за да се подаде едно реално число на системата, то трябва да бъде получено, а вътрешно за компютъра няма как да се получат неизчислими числа.

И все пак реалните числа биха могли да идват и отвън, например от интерфейса с човек, или от връзка с друга машина от различен тип. Както възможностите на една изчислима функция, работеща релативно някакво множество, не са еднакви с тези на изчислимата спрямо празно множество, така и една TTE машина с произволен вход е по-добра от ограничената до рационална основа.

Системата, представена в този текст, представя реалните числа чрез термове, манипулира ги като такива, и позволява произволен брой оракули, представящи външни реални числа.

Тя има и още едно съществено различие — всички операции в нея са примитивно рекурсивни, което гарантира приключване на всяка операция и демонстрира достатъчността на примитивната рекурсия за преработване на реални числа.

Ще разгледаме теорията, която позволява това; ефективен начин за представяне на термовете, описващи едно изчисление; а също и алгоритмите, реализиращи самите елементарни функции при най-ниска известна сложност. Ще се спрем и на едно разумно примитивно рекурсивно ограничение на въпроса за извеждане на търсено от потребителя свойство.

2 Теория

2.1 Изчислимост на реални числа

В Канторовата теория определение за реално число се дава така: класът на еквивалентност на редици рационални числа с регулатор на сходимост, като две редици рационални числа *a* и *b* са еквивалентни, ако $\lim_{n\to\infty}(a_n - b_n) = 0$. Така, за да представим едно реално число α, е достатъчно да намерим редица рационални числа с регулатор на сходимост, такава че $\lim_{n\to\infty}(a_n - \alpha) = 0$.

Рационалните числа са изброими и съществуват много начини за представянето им с помощта на рекурсивна функция. Може би най-често използваната номерация $q: \mathbb{N} \to \mathbb{Q}$ на множеството на рационалните числа е с помощта на тройка функции $I_1, I_2, I_3: \mathbb{N} \to \mathbb{N}$, такива, че $q(n) = \frac{I_1(n) - I_2(n)}{I_3(n)+1}$. Ако се избере такава тройка, за която докато *n* пробягва $\mathbb{N}$, $I_1(n)$, $I_2(n)$ и $I_3(n)$ пробягват всички възможни тройки естествени числа, ще получим номерация, за която на всяко рационално число съответства поне едно естествено число, което го представя. Знаем, че съществува такава тройка рекурсивни функции. За следващите разсъждения ще предполагаме, че $q: \mathbb{N} \to \mathbb{Q}$ е функция с тези свойства.

Дефиниция 1. Представяне на реално число α ще наричаме такава рекурсивна функция $\varphi: \mathbb{N} \to \mathbb{N}$, за която

$$(1) \qquad \left|q(\varphi(n)) - \alpha\right| < \frac{1}{n+1}$$

за всяко $n \in \mathbb{N}$. Едно число ще наричаме изчислимо тогава и само тогава, когато такава φ съществува.

Нека да разгледаме един пример. Да вземем за кодираща функция на тройка естествени числа

$$(2) \qquad \Pi_3(u,v,w) = 2^u 3^v (6w+1), I_1(n) = (n)_2, I_2(n) = (n)_3, I_3(n) = \frac{\frac{n}{2^{(n)_2} 3^{(n)_3}} - 1}{6}$$

Тези функции са примитивно рекурсивни като композиция на примитивно рекурсивни, а всяка тройка $u, v, w \in \mathbb{N}$ има някакъв индекс *n*. Тогава реалното число *e* е изчислимо например с помощта на функцията

$$(3) \qquad e(n) = \Pi_3\left(\sum_{i=0}^{n+1} \frac{(n+1)!}{i!}, 0, (n+1)!\right).$$

Това е така, защото за всяко $n \in \mathbb{N}$:

$$(4) \qquad q(e(n)) = \frac{I_1(e(n)) - I_2(e(n))}{I_3(e(n))} = \frac{\sum_{i=0}^{n+1} \frac{(n+1)!}{i!} - 0}{(n+1)!} = \sum_{i=0}^{n+1} i!$$

$$\left|q(e(n)) - e\right| = \sum_{i=n+2}^{\infty} i! < \frac{1}{(n+1)!} \le \frac{1}{n+1}$$

□

Не всяко реално число е изчислимо: тъй като изчислимите функции върху естествени числа са изброимо много, възможните реални числа, представени от тях, също са изброимо много, недостатъчно да покрият цялото множество на реалните числа.

Пример за неизчислимо реално число е сумата $x_A = \sum_{i \in A} 2^{-i}$ на редът на Шпекер за нерекурсивно A, например K. Ако допуснем, че x_K е изчислимо например с рекурсивната $\varphi_K: \mathbb{N} \to \mathbb{N}$, то за да построим рекурсивна функция, разрешаваща стоп проблема, е достатъчно да проверим дали 2^{-i} присъства в x_K. Съществува $l > i$, за което $l \notin K$, и $j > l$, за което $j \in K$. Тогава $\left|q(\varphi_K(2^{j+1})) - x_K\right| < 2^{-j-1}$. Тогава $x_K - 2^{-j} < q(\varphi_K(2^{j+1})) - 2^{-j-1} < x_K$. Тъй като

$$(5) \qquad x_K = \sum_{k \in K, k<l} 2^{-k} + 0.2^{-l} + \sum_{k \in K, l<k<j} 2^{-k} + 1.2^{-j} + \sum_{k \in K, j<k} 2^{-k},$$

получаваме

$$(6) \qquad \sum_{k \in K, k<l} 2^{-k} < q(\varphi_K(2^{j+1})) - 2^{-j-1} < \sum_{k \in K, k<l} 2^{-k} + 2^{-l} + 2^{-j}$$

следователно за всяко $m < l$

$$(7) \qquad \begin{gathered} 2^m(q(\varphi_K(2^{j+1}))) = a \Rightarrow \\ \left(2^m(q(\varphi_K(2^{j+1})) - (2^{j+1}+1)^{-1}) > \lfloor a \rfloor \wedge 2^m(q(\varphi_K(2^{j+1})) + (2^{j+1}+1)^{-1}) < \lceil a \rceil\right) \end{gathered}$$

Това доказва, че съществува такова n, че

$$(8) \qquad 2^i(q(\varphi_K(n)) = a \Rightarrow \left(2^i(q(\varphi_K(n)) - (n+1)^{-1}) > \lfloor a \rfloor \wedge 2^i(q(\varphi_K(n)) + (n+1)^{-1}) < \lceil a \rceil\right)$$

откъдето $\lfloor 2^i x_K \rfloor = \lfloor 2^i q(\varphi_K(n)) \rfloor$, следователно $\lfloor 2^i q(\varphi_K(n)) \rfloor \bmod 2 = 1 \Leftrightarrow i \in K$

Условието (8) е рекурсивно проверимо. За решаване на проблема дали дадено естествено число i принадлежи на K достатъчно е чрез пресмятане на все по-точни приближения $q(\varphi_K(n))$ на x_K да стигнем до такова, което удовлетворява въпросното условие. □

2.2 Примитивно рекурсивна изчислимост

Тук ще изложим няколко резултата от [13], които ще са ни нужни за по-нататъшните разсъждения.

Ако в Дефиниция 1 за изчислимост на реално число заменим "рекурсивна" с "примитивно рекурсивна", ще получим дефиниция, която не е еквивалентна с нея.

Например, за всяка рекурсивна функция t: $\mathbb{N} \to \{0, 1\}$ е дефинирано числото

$$(9) \qquad \alpha_t = \sum_{i=0}^{\infty} 4^{-i} t(i)$$

и то е изчислимо. Да разгледаме произволно негово представяне φ_t.

Тогава

$$(10) \qquad 4^m \varphi_t(4^{m+1} - 1) - \tfrac{1}{4} < 4^m \alpha_t < 4^m \varphi_t(4^{m+1} - 1) + \tfrac{1}{4},$$

а

$$(11) \qquad \lfloor 4^m \alpha_t \rfloor \le 4^m \alpha_t \le \lfloor 4^m \alpha_t \rfloor + \sum_{i>0} 4^{-i} = \lfloor 4^m \alpha_t \rfloor + \tfrac{1}{3},$$

откъдето

$$(12) \qquad \lfloor 4^m \alpha_t \rfloor \le \lfloor 4^m \varphi_t(4^{m+1} - 1) + \tfrac{1}{4} \rfloor \le \lfloor 4^m \alpha_t + \tfrac{1}{2} \rfloor \le \lfloor \lfloor 4^m \alpha_t \rfloor + \tfrac{1}{3} + \tfrac{1}{2} \rfloor = \lfloor 4^m \alpha_t \rfloor,$$

и следователно

$$(13) \qquad t(m) = \lfloor 4^m \varphi_t(4^{m+1} - 1) + \tfrac{1}{4} \rfloor \bmod 4.$$

Така при примитивно рекурсивна φ_t бихме могли да намерим примитивно рекурсивен начин за пресмятане на стойностите на t, което не е възможно за всички t. □

В [13] последният извод е направен с помощта на равенство, подобно на (13), но с по-сложен израз в дясната страна.

Въпреки това дефиниция на изчислимо реално число с помощта на примитивно рекурсивни функции е възможна. Това може да стане ако например разрешим дясната страна на (1) да бъде заменена с по-слабо условие, което зависи от реалното число.

Дефиниция 2. Ще дефинираме **тотална апроксимация** на едно реално число α като такава двойка примитивно рекурсивни функции (A, E), за които

$$(14) \qquad |q(A(n)) - \alpha| < q(E(n))$$

и измежду стойностите на $q(E(n))$ има произволно близки до 0.

Следващото твърдение е доказано по същество в [8]. Изложеното тук доказателство не се различава от предложеното в [13].

Твърдение 1. Едно число има тотална апроксимация тогава и само тогава, когато то е изчислимо.

Ако имаме тотална апроксимация (A, E) на реалното число α, то функцията f: $\mathbb{N} \to \mathbb{N}$, дефинирана чрез

$$(15) \qquad f(n) = A\big(\mu m\big((n+1)\big(I_1(E(m)) - I_2(E(m))\big) < I_3(E(m)) + 1\big)\big)$$

е рекурсивна, защото измежду стойностите на $q(E(n))$ има произволно близки до 0, и изпълнява условието

$$(16) \qquad \left|q(f(n))-\alpha\right| < (n+1)^{-1}$$

за всяко $n \in \mathbb{N}$.

Обратно, ако f е рекурсивна функция, изпълняваща условията на Дефиниция 1:

Съществува четворка примитивно рекурсивни функции, които номерират множеството на всички четворки от вида $(n, I_1(f(n)), I_2(f(n)), I_3(f(n)))$. Нека тези функции да са съответно s, t, u ,v. Тогава двойката

$$(17) \qquad \begin{aligned} A(n) &= \Pi_3(t(n),u(n),v(n)), \\ E(n) &= \Pi_3(1,0,s(n)+1) \end{aligned}$$

е примитивно рекурсивна и изпълнява

$$(18) \qquad \left|q(A(n))-\alpha\right| < q(E(n)).$$

f е дефинирана за произволно големи n, затова измежду стойностите на така получената $q(E(n))$ има произволно близки до 0. □

Дефиниция 2 обаче има сериозен недостатък — по тотална апроксимация на едно ненулево реално число α не е възможно примитивно рекурсивно да се получи тотална апроксимация на неговото реципрочно.

Нека допуснем, че това е възможно, т.е. съществува двойка примитивно рекурсивни оператори, която трансформира тотална апроксимация до тотална апроксимация на нейното реципрочно. Нека $t: \mathbb{N} \to \mathbb{N}$ е рекурсивна функция с примитивно рекурсивна графика, но която не е примитивно рекурсивна. Да вземем за всяка двойка естествени k и n:

$$(19) \qquad a_k = \frac{1}{t(k)+1}, A_k(n) = \Pi_3(1,0,\min\{t(k),n\}+1), E(n) = \Pi_3(1,0,n+1)$$

(A_k, E) е примитивно рекурсивна тотална апроксимация на числото a_k. По нашето предположение съществува двойка (A_k', E'), тотална апроксимация на $1/a_k$, която изпълнява условието

$$(20) \qquad \left|q(A_k'(n)) - (t(k)+1)\right| < (E'(n))$$

за всяка двойка естествени k и n. В частност, имаме

$$(21) \qquad \left|q(A_k'(0)) - (t(k)+1)\right| < q(E'(0))$$

и това дава примитивно рекурсивна граница за $t(k)$, което е достатъчно, за да намерим примитивно рекурсивна функция, пресмятаща t, което не е възможно. □

Този проблем може да се разреши ако на примитивно рекурсивните функции, представящи апроксимации на реални числа, се позволи стойност неопределеност. Ако в (14) дясната част на неравенството бъде 0, то няма как да бъде изпълнено; следователно за коя да е тотална апроксимация $q(E(n))$ не може да бъде 0 за кое да е n. Ще се възползваме от тази излишна стойност.

Дефиниция 3. Частична апроксимация на едно реално число α ще наричаме двойката (A, E), такава че

$$(22) \qquad I_1(E(n)) = I_2(E(n)) \vee \left|q(A(n)) - \alpha\right| < q(E(n))$$

и измежду стойностите на $q(E(n))$ има произволно близки до 0 различни от 0.

Твърдение 2. Едно число има частична апроксимация ако и само ако то има тотална апроксимация.

За всяка частична апроксимация модификация на правилото (15):

$$(23)$$
$$f(n) = A\Big(\mu m\big(I_1(E(m)) \neq I_2(E(m)) \wedge (n+1)\big(I_1(E(m)) - I_2(E(m))\big) < I_3(E(m)) + 1\big)\Big)$$

и тук дава валидна рекурсивна функция, която изпълнява условията на Дефиниция 1, от която по (17) можем да получим тотална апроксимация.

От друга страна всяка тотална апроксимация е и коректна частична апроксимация. □

2.3 Изчислимост на функции върху реални числа

Знаем, че имаме кодирания в естествени числа на двойка естествени числа и на произволна крайна редица естествени числа. Където се налага, ще използваме стандартните означения L, R, $(n)_k$ на функциите, намиращи съответно ляв и десен елемент от двойка и елемента с индекс k от редицата, представена чрез n, и $\langle a_i \rangle_{i=1,..k}$ на функцията, кодираща крайна редица с дължина k.

Нека имаме частична функция f: $\mathbb{R} \to \mathbb{R}$.

Дефиниция 4. Представяне на f ще наричаме такава рекурсивна функция φ: $\mathbb{N}^* \to \mathbb{N}^*$, за която при всеки избор на $\alpha \in \mathrm{dom}(f)$ са изпълнени следните условия:

а) за всяка крайна редица $a_1, a_2, a_3, \ldots, a_k$ от естествени числа, такива че

$$(24) \qquad |q(a_n) - \alpha| < (n+1)^{-1},$$

$n=1, 2, \ldots, k$, φ дава редица $b_1, b_2, b_3, \ldots, b_l$, такава че $|q(b_n) - f(\alpha)| < (n+1)^{-1}$, $n=1, 2, \ldots, l$;

б) за произволно l съществува k, такова че за произволна изпълняваща (24) редица $a_1, a_2, a_3, \ldots, a_k$, φ дава редица с дължина поне l.

f се нарича изчислима ако и само ако такава φ съществува.

Изчислимите в този смисъл функции са изчислими в смисъла на [15]. Не сме сигурни дали обратното е вярно.

Дефиниция 5. Частична апроксимация на f ще наричаме такава примитивно рекурсивна функция ψ: $\mathbb{N}^2 \to \mathbb{N}^2$, за която при всеки избор на $\alpha \in \mathrm{dom}(f)$ са изпълнени следните условия:

а) за всяка двойка естествени числа (a, e), където

$$(25) \qquad I_1(e) = I_2(e) \text{ или } |q(a) - \alpha| < q(e),$$

ψ дава двойка $(a^*, e^*) = \psi(a, e)$, такава че $I_1(e^*) = I_2(e^*)$ или $|q(a^*) - f(\alpha)| < q(e^*)$,

б) за всяко $\varepsilon > 0$ съществува $\delta > 0$, такава че за всяко двойка естествени (a, e), изпълняващи (25), имаме

$$(26) \quad 0 < q(e) < \delta \Rightarrow 0 < q(R(\psi(a, e))) < \varepsilon.$$

Твърдение 3. Една функция върху реални числа е изчислима тогава и само тогава, когато тя има частична апроксимация.

Под приближение a от тук нататък ще разбираме номер на приближение, изпълняващ (24), а под приближение (a, e) ще разбираме номер, изпълняващ (25).

Нека да забележим, че примитивно рекурсивно можем да преобразуваме между крайни редици, използвани в първата дефиниция и поредица от двойки, всяка от които отговаря на условията на втората: от редица $<a_n>_{n=1,..k}$, изпълняваща (24) можем да образуваме редицата $\tau_{\rightarrow}(<a_n>_{n=1,..k}) = <a_n, \Pi_3(1, 0, n+1))>_{n=1,..k}$, всеки член от която изпълнява (25).

Обратното преобразуване става чрез

$$(27) \quad \tau_{\leftarrow}(<a_m, e_m>_{n=1,..k}) = <a_{i_p}>_{p=1,..l}, \text{ където}$$

$$(28) \quad i_p = \mu n \le k(I_1(e_n) \ne I_2(e_n) \wedge (p+1) \mid I_1(e_n) - I_2(e_n) \mid < I_3(e_n) + 1), p = 1,..l\,.$$

Тази трансформация също е примитивно рекурсивна, защото редицата е крайна и минимизацията в този случай е ограничена.

Нека f има частична апроксимация ψ и $a_1, a_2, a_3, \ldots a_k$ е редица от приближения на $\alpha \in \mathrm{dom}(f)$, такива че $|q(a_n) - \alpha| < (n+1)^{-1}$ за $n = 1, \ldots, k$. Тогава $\varphi(<a_n>_{n=1,..k}) = \tau_{\leftarrow}(\psi^*(\tau_{\rightarrow}<a_n>_{n=1,..k}))$, където ψ^* е такава, че $\psi^*(<a_n, e_n>_{n=1,..k}) = <\psi(a_n, e_n)>_{n=1,..k}$ за всяка крайна редица $<a_n, e_n>_{n=1,..k}$. Всички функции, използвани в дефиницията на φ са примитивно рекурсивни, следователно и самото φ е примитивно рекурсивно. Тъй като при достатъчна продължителност на входната редица условието в (26) ще бъде изпълнени за произволно близки до нула δ, редицата $\psi^*(<a_n, e_n>_{n=1,..k})$ може да изпълни минимизацията за $\tau_{\leftarrow}$ за произволно големи i.

Обратно, нека f има представяне φ и (a, e) е приближение, удовлетворяващо (25). Ако е изпълнено първото условие в дизюнкцията (25), полагаме $\psi(a, e) = (a, e)$.

В противен случай, примитивно рекурсивно можем да пресметнем

$$(29) \quad l = \left\lfloor \frac{I_3(e)}{I_1(e) - I_2(e)} \right\rfloor.$$

Ще построим l на брой редици с дължини от 1 до l, изпълняващи (24). Достатъчно е за всяка от тях да повторим a нужния брой пъти. Тази процедура също е примитивно рекурсивна.

Съществува примитивно рекурсивна процедура, с помощта на която можем да изпълним едновременно φ върху тези l редици за l стъпки. Ако никое от тези ограничени изпълнения на φ не даде резултат, полагаме ψ неопределено. Ако φ даде поне един резултат, използваме за ψ да бъде най-точното получено приближение.

Тъй като l неограничено расте с намаляването на e, всеки от резултатите, които φ дава, ще се получи за някое l и за всички по-големи от него. Тъй като φ дава произволно добри приближения, които зависят от дължината на подадената като

параметър редица, а тя от (29) расте с намаляването на $q(e)$, ψ изпълнява условието (26) за частична апроксимация. □

Като страничен резултат това доказателство показва, че примитивно рекурсивните функции са достатъчни за представяне на функции върху реални числа по Дефиниция 4.

Аналогично на изчислимите реални числа, не всички функции върху реални числа са изчислими. Едно от условията, които изчислимите функции изпълняват, е непрекъснатостта им във всяка точка от дефиниционното им множество.

Твърдение 4. Всяка изчислима функция е непрекъсната във всяка точка от дефиниционната си област.

Едно изображение f: $\mathbb{R}\to\mathbb{R}$ е непрекъснато в т. $x_0\in\mathrm{dom}(f)$, ако

$$(30)\qquad \forall\varepsilon>0\,\exists\delta>0\,\forall x\left(|x-x_0|<\delta\Rightarrow|f(x)-f(x_0)|<\varepsilon\right)$$

Нека ψ е частична апроксимация на f. За всяко $\varepsilon>0$ съществува δ, която изпълнява (26), и за всяко реално число α съществува достатъчно добро приближение (a, e), което изпълнява лявата страна на импликацията в (26). Да изберем такова приближение за числото x_0 в качеството на α и нека $(a^*, e^*) = \psi(a, e)$ за това приближение. То удовлетворява $q(e^*) < \varepsilon$. Тогава (a, e) изпълнява условието (25) за всяко число y в интервала $(x_0 - (q(e) - |x_0 - q(a)|), x_0 + (q(e) - |x_0 - q(a)|))$, откъдето получаваме $|q(a^*) - f(y)| < q(e^*)$. Но $|q(a^*) - f(x_0)| < q(e^*)$, следователно $|f(x_0) - f(y)| < 2\varepsilon$. □

Това условие показва, че коя да е прекъсната функция не е изчислима. Така например функцията $sign(x)$, дефинирана като

$$(31)\qquad sign(x)=\begin{cases}-1, x<0\\0, x=0\\1, x>0\end{cases}$$

не е изчислима, защото $sign(x)$ е прекъсната в точката 0. От друга страна, функцията $sign'(x)$, дефинирана чрез

$$(32)\qquad sign'(x)=\begin{cases}-1, x<0\\\neg!, x=0\\1, x>0\end{cases}$$

може да бъде изчислима, защото единствената точка на прекъсване на $sign(x)$ тук не е от дефиниционната област на функцията.

Ще продължим с описанието на примитивно рекурсивните функции, които реализират частични апроксимации на елементарните функции. Ще въведем понятието "работна точност", което ще бъде някаква цяла оценка на $-\log q(e)$ (брой приблизително верни цифри на входа) и ще бъде аналог на l, получено чрез (29) в доказателството на Твърдение 3. Ще представим алгоритми за пресмятането на елементарните функции с намаляваща грешка, зависеща от работната точност и от отклонението на входа.

Ще работим с относително представяне на грешката, което е еквивалентно на абсолютното, използвано досега, но е по-удобно за практическа работа.

2.4 Представяне (на число и грешка)

Нека имаме реалното число α и рационална оценка за интервал $(q - e,\ q + e)$, в който то със сигурност се намира. Тогава можем да изберем такива рационални числа $a \neq 0$ и $\varepsilon > 0$, че

$$(33) \qquad \left|\frac{\alpha}{q} - 1\right| < \varepsilon,$$

тъй като

$$(34) \qquad a = \begin{cases} q, \text{ако } \alpha \neq 0 \\ e, \text{в пр. случай} \end{cases}$$
$$\varepsilon = \begin{cases} e/|q|, \text{ако } \alpha \neq 0 \\ 2, \text{в пр. случай} \end{cases}$$

изпълняват условието (33).

Ако за някое α сме намерили $a \neq 0$ и $\varepsilon > 0$, изпълняващи (33), е ясно, че съществува $e \in \mathbb{R}$, такова че $\alpha = a(1+e)$ и $|e| < \varepsilon$.

2.5 Пресмятане на елементарните функции (число и грешка)

В тази глава ще изложим методите, по които числено могат да бъдат пресметнати елементарните функции при определена работна точност n. Ще използваме мярката сложност на изчислението, която ще базираме на някаква функция за сложността на операциите на по-ниско ниво, зависеща от n.

Ще опишем реализации на функциите, които ще гарантират, че при вход $(a,\ \varepsilon)$ грешката, получена при крайното оценяване на $f(a)$, ще бъде по-малка от 2^{-n}. Ще намерим оценка за броя итерации, които ще са нужни това да се постигне спрямо работната точност n. Този брой итерации ще ни даде сложността на реализацията.

Ще приемем, че основните функции събиране и умножение вече са реализирани, като сложността на събирането е пренебрежима спрямо тази на умножението, а последната ще означим с $M(n)$. Методите за пресмятане на елементарните функции са избирани така, че общата сложност за пресмятане на коя да е функция да е по-малка от $O(nM(n))$. Тъй като $\lim_{n\to\infty} \frac{\log^{\alpha} n}{n} = 0$ за произволно $\alpha \in \mathbb{R}$, ще изискваме сложността на пресмятане да е най много $O(M(n)log^{c} n)$ за някоя константа c.

Освен това ще намерим граници за относителната грешка ε', получена в резултат на неточно приближение на аргумента и ще гарантираме, че стойността на абсолютната грешка $|a'|\varepsilon'$ намалява за намаляващо $|a|\varepsilon$ и аргументи в дефиниционната област на функцията.

Когато подаденото на функцията приближение се окаже извън дефиниционната област на функцията, от реализацията се изисква да изследва подаденото приближение и ако то допуска стойност на реалния аргумент в границите на дефиниционната област да оцени сечението на допустимите стойности на аргумента с дефиниционната област. Тази процедура би могла да се държи

странно при реални аргументи извън дефиниционната област, но изискванията в Дефиниция 5 не предполагат каквото и да било за поведението на функцията в тези случаи.

2.5.1 Грешка при събиране

За грешката при събиране на реалните числа α и β, имайки техни приближения $a(1+e_a) = \alpha$ и $b(1+e_b) = \beta$, получаваме:

$$(35) \qquad \varepsilon' = \left|\frac{\alpha+\beta}{a+b} - 1\right| = \left|\frac{a(1+e_a)+b(1+e_b)}{a+b} - 1\right| = \left|\frac{ae_a+be_b}{a+b}\right| < \frac{|a|\varepsilon_a+|b|\varepsilon_b}{|a+b|}$$

Ако $a+b$ се окаже нула, ще оценим абсолютната грешка и, използвайки (34), по нея можем получим коректни стойности за приближение и грешка.

С намаляване на $|a|\varepsilon_a+|b|\varepsilon_b$ получената абсолютна грешка намалява произволно близко до 0.

2.5.2 Грешка при умножение

Отново при $a(1+e_a) = \alpha$ и $b(1+e_b) = \beta$, $ab \neq 0$:

$$(36) \qquad \varepsilon' = \left|\frac{\alpha\beta}{ab} - 1\right| = \left|\frac{ab(1+e_a)(1+e_b)}{ab} - 1\right| = |e_a+e_b+e_ae_b| < |\varepsilon_a| + |\varepsilon_b| + |\varepsilon_a\varepsilon_b|$$

Получената оценка за $|a'|\varepsilon'$ намалява с намаляването на $|a|\varepsilon_a+|b|\varepsilon_b$.

2.5.3 Реципрочно

Съвременен метод за намиране на реципрочно с експоненциална сходимост е методът на Нютон на последователни приближения за намиране на нула на функция:

$$(37) \qquad x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}.$$

Приложен за $f(x) = a - 1/x$, той дава

$$(38) \qquad x_{i+1} = x_i - \frac{a-1/x_i}{1/x_i^2} = 2x_i - ax_i^2.$$

Ако грешката при i-тата итерация е ε:

$$(39) \qquad \frac{1}{a} = x_i(1+e) \Rightarrow \varepsilon' = \left|\frac{a^{-1}}{x_{i+1}} - 1\right| = \left|\frac{1}{ax_i(2-ax_i)} - 1\right| = \left|\frac{1+e}{2-(1+e)^{-1}} - 1\right| = \left|\frac{e^2}{1+2e}\right| = \overset{|e|<1/2}{=} \left|e^2\sum_{i=0}^{\infty}(-2e)^i\right| \overset{|e|\le 1/4}{<} 2\varepsilon^2,$$

откъдето ако $\varepsilon \le 2^{-e} \rightarrow \varepsilon' \le 2^{-2e+1}$, т.е. на всяка итерация броят на верните цифри почти се удвоява. Двете условия за e изискват добро начално приближение, което в реални условия лесно може да се намери (например с *double*, което дава $\varepsilon < 2^{-50}$). Броят итерации за постигане на резултат с исканите характеристики е по-малък от $2\log n$. Сложността на така реализираната операция е $O(M(n)\log n)$.

Ако грешката на входният параметър е ε:

$$(40)\qquad \alpha = a(1+e) \Rightarrow \varepsilon' = \left|\frac{a^{-1}(1+e)^{-1}}{a^{-1}} - 1\right| = \left|\frac{1}{1+e} - 1\right| = \left|\frac{e}{1+e}\right| \overset{|e|<1}{=} \left|e\sum_{i=0}^{\infty}(-e)^i\right| \overset{|e|\le ½}{<} \varepsilon + 2\varepsilon^2$$

Ако входното приближение не е достатъчно добро, за да се изпълнят двете условия за $|e|$, резултатът може да се смята за неопределен. Ако $\alpha \neq 0$ е сигурно, че някое по-добро приближение ще изпълнява условията. Ненулево α също гарантира и ограниченост на $1/\alpha$, откъдето и намаляваща оценка за абсолютната грешка при намаляване на $|a|\varepsilon$.

2.5.4 Квадратен корен

Квадратният корен също се може да се пресметне по метода на Нютон, приложен за функцията $f(x) = a - 1/x^2$, чиято нула е реципрочното на квадратен корен от x. Получаваме

$$(41)\qquad x_{i+1} = x_i - \frac{a - x^{-2}}{2x^{-3}} = x_i\left(1 - \frac{ax_i^{\,2} - 1}{2}\right) = x_i\frac{3 - ax_i^{\,2}}{2}\ .$$

Всяка итерация променя грешката така:

(42)

$$a^{-½} = x_i(1+e) \Rightarrow \varepsilon' = \left|\frac{a^{-½}}{x_{i+1}} - 1\right| = \left|\frac{2x_i(1+e)}{x_i(3 - ax_i^{\,2})} - 1\right| = \left|\frac{2+2e}{3-(1+e)^{-2}} - 1\right| = \left|\frac{2(1+e)^3}{2+6e+3e^2} - 1\right| =$$

$$= \left|\frac{e^2(3+2e)}{2+6e+3e^2}\right| \le \left|\frac{e^2(3+2e)}{2(1+3e)}\right| \overset{|e|<⅓}{=} \left|e^2\frac{3+2e}{2}\sum_{i=0}^{\infty}(-3e)^i\right| \overset{|e|\le ⅙}{\le} \frac{e^2}{2}\,|3+2e|2 < 4\varepsilon^2$$

Ако $\varepsilon \le 2^{-e} \rightarrow \varepsilon' \le 2^{-2e+2}$, отново имаме удвояване на броя верни битове, брой итерации, по-малък от $\log n$, и сложност $O(M(n)\log n)$. Както и при предишното разсъждение, началното приближение получено с *double* аритметика е достатъчно точно за да изпълни условията за e.

Оттук и $x^{1/2} = x.x^{-1/2}$ има сложност $O(M(n)\log n)$. Тъй като предполагаме, че оценката никога не е нула, делението тук не би могло да създаде проблеми.

Грешката на резултатът спрямо входът за квадратен корен се пресмята чрез:

$$(43)\qquad \alpha = a(1+e) \Rightarrow \varepsilon' = \left|\frac{\sqrt{a(1+e)}}{\sqrt{a}} - 1\right| = \left|\sqrt{(1+e)} - 1\right| =$$

$$= \begin{cases} \sqrt{1+e} - 1 < 1 + e - 1 < \varepsilon, \text{ при } e \ge 0 \\ 1 - \sqrt{1+e} \overset{e>-1}{<} 1 - 1 - e < \varepsilon, \text{ при } e < 0 \end{cases}$$

а грешката при реципрочен квадратен корен може да се пресметне като реципрочно на квадратен корен и е $\varepsilon' \le \varepsilon + 2\varepsilon^2$.

И тук важат забележките към грешката при реципрочно.

2.5.5 Логаритъм

Добра сложност при пресмятане на логаритъм дава известен резултат на Гаус за границата на аритметично-геометричната редица (a_i, b_i):

$$(44)\qquad \begin{aligned} a_0 &= a, b_0 = b \\ a_{i+1} &= \frac{a_i + b_i}{2}, b_{i+1} = \sqrt{a_i b_i} \end{aligned}$$

Ако означим с $AGM(a,b) = \lim_{i\to\infty} a_i = \lim_{i\to\infty} b_i$, то $\ln y = \frac{\pi}{2AGM(1,4y^{-1})}(1+h)$, където $h < y^{-2}$. ([5], също и [4], [2], и алтернативен алгоритъм в [3]).

При положение, че имаме предварително пресметната стойност за π (2.5.7), ако изберем y достатъчно голямо, например $y > 2^{n+1}$, ще получим резултат с грешка по-малка от 2^{-2n}.

$y > 2^{n+1}$ лесно може да се осигури, понеже $\ln ma = \ln m + \ln a$ за произволно m, например 2^{n+1}. Ако комбинираме и с предварително пресметнат $\ln 2 = (\ln 2^{n+1})/(n+1)$, можем да пресметнем логаритъм от произволно число.

Да разгледаме сходимостта на метода:

Ако $0 < b_i \ll a_i$: $b_{i+1}/a_{i+1} = \frac{2\sqrt{b_i/a_i}}{1+b_i/a_i} < 2\sqrt{b_i/a_i}$, т.е. необходими са не повече от $\lceil \log_2 (a_0/b_0) \rceil$ итерации за да станат a_i и b_i близки. След това ако $b_i/a_i = 1+e_i$, $|e_i| < \varepsilon_i$ то

$$(45)\qquad \begin{aligned} \varepsilon_{i+1} = |1 - b_{i+1}/a_{i+1}| &= \left|1 - \frac{2\sqrt{1+e_i}}{2+e_i}\right| = \frac{(1-\sqrt{1+e_i})^2}{|2+e_i|} \le \\ &\le \begin{cases} \frac{(1+e_i-1)^2}{2+e_i} \le \frac{\varepsilon_i^{\,2}}{2} \le \varepsilon_i^{\,2}, \text{при } e_i \ge 0 \\ \frac{(1-1-e_i)^2}{2+e_i} \overset{e_i>-1}{\le} \varepsilon_i^{\,2}, \text{при } e_i < 0 \end{cases} \end{aligned},$$

т.е отново за логаритмичен брой стъпки получаваме нужната точност. Тъй като *AGM* използва квадратен корен, чиято сложност е $O(M(n)\log n)$, за сложността на ln получаваме $O(M(n)\log^2 n)$.

Ако грешката на входният параметър е ε:

$$(46)\qquad \begin{aligned} \alpha = a(1+e) \Rightarrow \varepsilon' &= \left|\frac{\ln(a(1+e))}{\ln a} - 1\right| = \left|\frac{\ln(1+e)}{\ln a}\right| = \left|\frac{-\ln(1+e)^{-1}}{\ln a}\right| \overset{|e|<1}{=} \\ &= \left|\frac{\ln \sum_{i=0}^{\infty} (-e)^i}{\ln a}\right| \overset{|e|\le 1/2}{\le} \left|\frac{\ln(1+|e|+2e^2)}{\ln a}\right| < \frac{\varepsilon + 2\varepsilon^2}{|\ln a|} \end{aligned}$$

За да получим рационално ограничение е достатъчно в знаменател да поставим някое по-малко по абсолютна стойност от ln a. Двете условия за e могат да бъдат удовлетворени от по-добро приближение за кое да е число от дефиниционната област на ln. Стойността за абсолютната грешка $\varepsilon+2\varepsilon^2$ намалява с намаляването на $|a|\varepsilon$. Ако ln $a = 0$, можем да използваме (34) за да получим относителни стойности от абсолютната грешка.

2.5.6 Експонента

След като вече можем да пресмятаме ln x, можем да използваме метода на Нютон за пресмятане на e^y:

$$(47) \qquad f(x) = a - \ln x \Rightarrow x_{i+1} = x_i + (a - \ln x_i).x_i$$

За да сметнем сходимостта на метода, ще изведем един по-общ резултат за метода на Нютон:

Ако искаме да пресметнем $x = \psi(a)$ чрез $\varphi(y) = \psi^{-1}(y)$, използваме формулата

$$(48) \qquad x_{i+1} = x_i + \frac{a - \varphi(x_i)}{\varphi'(x_i)} = x_i + (\varphi(x) - \varphi(x_i))\psi'(\varphi(x_i)).$$

Ако използваме развитието в ред на Тейлър на $\varphi(x)$ около точката x_i, и формулата за остатъчния член, получаваме:

$$(49) \qquad x - x_{i+1} = \frac{(\varphi(x_i) - \varphi(x))^2}{2!}\psi''(\varphi(x_i) + (\varphi(x) - \varphi(x_i))\theta)), 0 < \theta < 1.$$

За конкретния случай $\psi(y) = e^y$ и $\varphi(x) = \ln x$, ако $x = x_i(1+h)$

$$\left|\frac{x - x_{i+1}}{x_{i+1}}\right| = \left|\frac{(\ln x - \ln x_i)^2}{2x_i(1 + a - \ln x_i)} e^{\ln x_i} e^{\ln x_i \theta h}\right| \overset{x_i \le 2, h \le 1}{<} \left|\frac{\ln^2(1+h)}{(1 + \ln(1+h))}\right| =$$

$$(50) \qquad = \left|\ln(1+h) - 1 + \frac{1}{(1 + \ln(1+h))}\right| \overset{|\ln(1+h)|<1}{=} \left|\ln(1+h) - 1 + \sum_{j=1}^{\infty} (-\ln(1+h))^j\right| \overset{|\ln(1+h)| \le ½}{\le}$$

$$\le (2\ln^2(1+h)) \overset{|h| \le ½}{\le} 4\varepsilon^2$$

Резултатът е коректен, тъй като от $e^y = 2^{\lfloor y/\ln 2 \rfloor} . e^{y - \lfloor y/\ln 2 \rfloor}$ можем да си осигурим $x_i<2$, а началното приближение, пресметнато с *double* аритметика, е достатъчно добро да изпълни условията за h. Сложността е $O(M(n)\log^3 n)$.

Грешката, получена от неточни входни данни, е:

$$(51) \qquad \alpha = a(1+h) \Rightarrow \varepsilon' = \left|\frac{e^{a(1+h)}}{e^a} - 1\right| = \left|e^{ha} - 1\right| < e^{\varepsilon|a|} - 1$$

Тъй като $|a|$ е ограничен, а e^y се приближава неограничено до 1 с приближаването на y към 0, дясната част на (51), а с нея и $|a'|\varepsilon'$, се приближава неограничено до 0.

2.5.7 Константата π

Реализиран е резултат на Борвайн и Борвайн [2], който гласи:

Ако $\alpha_0 = 6 - 4\sqrt{2}$ и $y_0 = \sqrt{2} - 1$,

$$(52) \qquad y_{n+1} = \frac{1 - \sqrt[4]{1 - y_n^4}}{1 + \sqrt[4]{1 - y_n^4}}$$

$$(53) \qquad \alpha_{n+1} = (1 + y_{n+1})^4 \alpha_n - 2^{2n+3} y_{n+1}(1 + y_{n+1} + y_{n+1}^2),$$

то $0 < \alpha_n - 1/\pi < 16.4^n e^{-2.4^n \pi}$.

На всяка стъпка $\ln \varepsilon$ намалява почти четири пъти. Достатъчни са по-малко от $2\log_4 n$ стъпки за достигане на пълна точност. Сложността е $O(M(n)\log^2 n)$.

Други алгоритми за пресмятане на π с логаритмична сложност могат да се намерят в [2] и [1]. Исторически първият от тях е използва *AGM* итерации и е открит независимо от Брент [4] и Саламин [11] през 1975.

2.5.8 Тригонометрични функции

По методите, описани по-горе, могат да бъдат пресметнати логаритъм и експонента и на комплексни числа. Резултатите за сходимостта и абсолютната стойност на грешката също са коректни, защото при изчисляването им не са използвани свойства, верни само за реални числа.

Като използваме тъждествата

$$(54) \qquad \begin{aligned} &\ln x = \ln|x| + i \arg x, \arg x = arctg(imag(x)/real(x)) \text{ за } real(x) > 0 \\ &e^y = e^{real(y)}.(\cos imag(y) + i \sin imag(y)) \end{aligned}$$

можем да пресметнем всяка от тригонометричните функции.

Формулата (33) $\left|\frac{\alpha}{a} - 1\right| < \varepsilon$, при комплексни α и a означава

$$(55) \qquad \alpha = a(1+e) \Rightarrow \begin{aligned} &\left|\frac{\alpha_r}{a_r} - 1\right| = \left|\frac{a_r(1+e_r) - a_i e_i - a_r}{a_r}\right| = \left|\frac{a_r e_r - a_i e_i}{a_r}\right| < \frac{|\varepsilon|\left(|a_r| + |a_i|\right)}{|a_r|} \\ &\left|\frac{\alpha_i}{a_i} - 1\right| = \left|\frac{a_i(1+e_r) + a_r e_i - a_i}{a_i}\right| = \left|\frac{a_i e_r + a_r e_i}{a_i}\right| < \frac{|\varepsilon|\left(|a_r| + |a_i|\right)}{|a_i|} \end{aligned}$$

Т.е. за да получим добро приближение за реалната (имагинерната) част трябва да си осигурим тя да бъде по-голяма по абсолютна стойност от имагинерната (респ. реалната).

2.6 Извеждане на резултат

Извеждането на резултат се реализира чрез последователно изискване на все по-точни приближения до достигане на резултат, който задължително удовлетворява дадено свойство. Такива свойства могат да бъдат например неравенства между две реални числа и намиране на приближение с дадена точност.

Процесът на извеждане на резултат изисква неограничена минимизация, която в много случаи може и да не завърши (например $\alpha = \beta$ не може да бъде изведено за никои α и β).

В реалния свят тази минимизация е ограничена от определени ресурси като памет и време. По тази причина ако функциите за изискване на резултат се заменят с примитивно рекурсивни, работещи само до определена граница на точността, могат да се изберат такива стойности на тази граница, че резултатите да се различават от теоретичните, но да са еднакви с реализируемите с неограничена точност.

3 Реализация

Реализацията на системата за работа с реални числа е написана на *C++* и е разделена в няколко основни модула:

- спомагателни структури и функции;
- модул за представяне на рационални числа, реализиран като представяне с плаваща запетая с произволна точност в класа *LongFloat*;
- модул за представяне на приближенията с оценяване на грешката при пресмятания в класовете *Estimate* и *ErrorEst*;
- модул за представяне на термове състоящ се от класа *RealObject* и неговите наследници;
- модул за интерфейс с потребителя в класа *Real*.

3.1 Плаваща запетая с произволна точност (*class LongFloat*)

Представянето на приближения на едно реално число изисква възможност за работа с рационални числа. Теоретично предпочитаното представяне на число като числител и знаменател като точни цели числа е много неудобно за работа в реални условия, тъй като всяка една операция в такъв случай е свързана с умножение, с непрекъснато нарастване на дължината и на числител, и на знаменател.

Предпочитан подход е използването на приближение на рационалното число, представено с точност, достатъчна за текущото ниво на пресмятането. Едно такова приближение, имащо за знаменател степен на някаква предварително определена основа, и числител с дължина, която може при нужда да се променя, обикновено се нарича число с плаваща запетая и произволна точност.

Съществуват множество готови пакети за работа с такива числа. С оглед изчерпателността на дипломната работа, беше предпочетено написването на собствен код за нуждите на тази система.

Модулът за работа с плаваща запетая с произволна точност в тази система използват за основа 2^{32} вместо класическото 2. Предпочитаната дължина на думата на съвременните компютри е 32 бита. Така, избирайки 32 бита за дължина на цифрата в нашите числа с плаваща запетая, можем да си осигурим максимална ефективност на изчисленията.

Преди всяко изчисление за намиране на приближение, модулът за плаваща запетая се инициализира за дадена работна точност. Тази точност важи за всички резултати при това изчисление. Когато то завърши, модулът се затваря, което освобождава всяка предварително заета памет и самия модул, който отново може да бъде използван, евентуално с по-висока точност.

Точността се задава в брой думи на мантисата. Това грубо означава, че по-висока с единица точност предлага нови десет коректни десетични цифри.

Пълният формат на числата с плаваща запетая в системата е следният:

- мантиса с фиксирана от работната точност дължина;

- 32-битова експонента;
- флаг за отрицателност на числото;
- флаг за специални стойности.

Мантисата се записва като последователност от 32-битови думи без знак. Избран е low endian формат на записване, т.е. първата в паметта дума има най-малко значение. Мантисата изпълнява и едно допълнително условие, свързано с точността — всяка нулева дума в най-крайна (най-значеща) позиция не носи никаква информация и би довела до загуба на точността при някои операции. По тази причина мантисата винаги е нормализирана така, че най-значещата ѝ дума да е ненулева.

Експонентата е 32-битово цяло число със знак. По-дълга експонента не е нужна, тъй като в реални условия много големи или много малки числа се получават когато точността на пресмятане е голяма; при съвременните машини обемът на паметта ще се е изчерпал значително преди достигане на подобна точност.

Флагът за **отрицателност** определя знака на числото.

Флагът за **специални стойности** гарантира коректно обслужване на случаите, в които текущата точност не е достатъчна. Най-простата специална стойност е нула; останалите се получават от нея при някои операции: безкрайност, при деление на нула, и неопределеност, при деление на нула с нула или някои операции с безкрайности. Нулата е присъединена към списъка специални стойности, защото не може да се запише в мантиса по правилата, описани по-горе.

Общата формула за пресмятане на представеното число, с изключение на специалните стойности, е:

$$(56) \qquad \textit{стойност} = (-1)^{\textit{знак}} 2^{32.\textit{експонента}} \left(\sum_{i=0}^{\textit{точност}-1} 2^{32(i-\textit{точност})} \textit{мантиса}(i) \right)$$

Формулите за пресмятане на сбор, разлика и произведение на числа с плаваща запетая се получават директно от предходната. Реализацията обръща внимание на постигането на максимална възможна точност, т.е. след коя да е от операциите резултатът да не се различава от истинския с повече от $2^{32.(\textit{експонента-точност})-1}$.

Сложността им в брой операции върху думи е пропорционална на точността за събиране и изваждане и пропорционална на квадрата на точността при умножение. Както ще видим по-нататък, умножението може да се реализира и със сложност $O(\textit{точност}.\log^2 \textit{точност})$.

Делението също има директна реализация, подобна на изучаваната в училище процедура за десетично деление, със сложност, пропорционална на квадрата на точността. Предпочетен, като по-ефективен при добро умножение, е методът на Нютон (2.5.3), който може да достигне сложност, равна на тази на умножението.

3.2 Грешката (*class ErrorEst*)

Важна част от задачата е и представянето на грешката. Тя трябва да бъде число, което да може да покрие целия интервал възможни точности на приближението; трябва да бъде сравнително точна, но да не добавя прекалено много към общото време за пресмятане; още по-важно е грешката да се подчинява на други закони

на пресмятането — такива, които да гарантират, че пресметната грешка никога няма да бъде по-малка от реалната.

Методът на представяне на приближенията не е подходящ за грешката поради последните две изисквания. Разхищение е използването на проста експонента, която да показва в кой бит е грешката в приближение, тъй като всяка една операция удвоява една такава експонента. Не може да бъде представена и с някой от хардуерно реализираните типове, които имат много ниски максимални стойности за експонентата.

С оглед ефективност е за предпочитане грешката да се представя като число с плаваща запетая с мантиса, дълга една дума. Тук основата 2^{32} води до съществена загуба на точност, тъй като в този случай има числа, чийто представяния имат само 1-2 значещи бита.

Избраното представяне е чрез 32-битова експонента при основа 2, и 32-битова мантиса, която има винаги 1 в най-старшия си бит. Знак тук не е нужен, защото грешката винаги се взима по абсолютна стойност.

Операциите с грешки се използват единствено от комбиниращия клас *Estimate*, който се грижи за правилното ѝ пресмятане. Правилата за закръгляване в *ErrorEst* са изцяло подчинени на нуждите на операторите в *Estimate*.

Формулите, използвани за пресмятане на грешката при елементарните операции, вече бяха описани в 2.5.

Estimate реализира пресмятанията на грешка така, че закръглянето винаги да бъде в правилната посока. Например при събиране аргументите в числителя се закръглят нагоре, умножението се закръгля нагоре и сумата в числителя също се закръгля нагоре. Но преобразуването на знаменателя се прави при закръгляне надолу.

Някои операции са с напълно неясен резултат при определени стойности на грешката, например реципрочно при грешка, по-голяма или равна на приближението. За такива случаи *ErrorEst* дефинира и стойност плюс безкрайност. Полученият резултат при такава грешка вероятно няма нищо общо с истинския, и изчислението трябва да бъде направено отново с по-голяма точност.

Друг интересен момент, свързан с пресмятането на грешката, е резултатът нула. Тъй като грешката се запомня относително приближението, не е възможно да се представи каквато и да било реално пресметната грешка при приближение нула. От друга страна, по-важно за нас не е полученото приближение, а въпросът дали истинският резултат попада в границите, определени от приближение и грешка. Затова е и абсолютно коректно в такъв случай да се отмести приближението, а грешката да се увеличи, така че новият интервал да покрива изцяло стария. Това например може да стане като приближението получи стойността на абсолютната грешка, а за стойност на относителната грешка се приеме 2.

3.3 Представяне на термове (*class RealObject*)

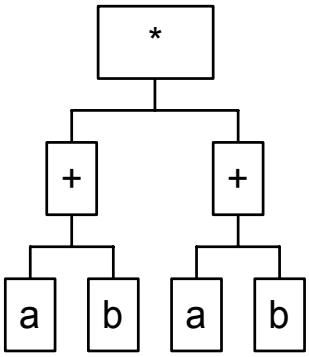
В основата на представянето на термовете в тази система стои понятието обект. Обектите са константи и операции. Всички те са наследници на класа *RealObject*, който предоставя на системата единен начин за обработка на всички обекти.

Структурата на свързване на обектите в тази система е подобна на дървовидната структура за представяне на израз, но с някои съществени разлики.

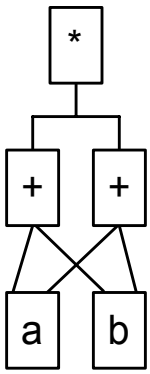
Всички пресмятания в системата се представят като един общ граф. Може да бъде свързан или не в зависимост от характера на изчислението. Графът се строи отдолу нагоре.

В основата на всяко изчисление стоят константите. Те могат да бъдат няколко вида, но имат една отличителна черта — не зависят от нищо друго в системата и затова стоят на най-ниско ниво; над тях са операциите, които зависят от аргументи. За аргумент на операциите могат да служат други операции, с което изчисленията се навързват едно над друго в подобна на дърво структура.

Тук идва основната разлика с дърво: всеки един обект на определено ниво може да бъде във връзка с множество обекти на по-високо ниво, а може да бъде и стойността на дадена потребителска променлива. Например, изразът (*a* + *b*) * (*a* + *b*) може да бъде представен чрез дърво така:



Структурата, с която системата би представила този израз, е:



Процесът на получаване на приближение е рекурсивен: всяка операция поисква приближения от своите аргументи, за да може да бъде приложена; те от своя страна продължават процедурата до достигане на константа или получен предварително резултат.

Тази процедура е аналогична на стандартната процедура на оценяване на израз, представен с дърво; но тук също има разлики, които позволяват да се сведе до минимум сложността на тази процедура.

3.3.1 Видове обекти

Видовете обекти отговарят на видовете термове, които представя системата, а те са:

- константи, подадени като *double* от потребителя (*RealFromDouble*)
- константи, подадени като текстов низ от потребителя (*RealFromString*)

- константи, подадени като оракул от потребителя (*RealFromOracle*)
- естествени константи (π^{-1}, π, $\ln 2$, $\ln 10$), наречени още безаргументни операции (*RealNullary*).
- унарни операции (*RealUnary*)
- бинарни операции (*RealBinary*)

Първите четири са константи и винаги са на най-ниско ниво в йерархията. Последните две могат да са на произволно ниво.

Естествените константи са използвани често от системата и е много вероятно във всеки един момент те да са вече пресметнати. Затова чрез представянето им като отделен клас обекти може да се избегне излишно повторно изчисляване. Освен това, алгоритмите, по които се пресмятат тези константи, са по-добри от еквивалентните им стойности на функции.

Всеки от тези видове обекти може да бъде стойността на дадена потребителска променлива, или аргумент на дадена операция на по-високо ниво. Във всеки един момент потребителят може да реши да продължи пресмятането, с което да въведе нови обекти в структурата, и да промени обекта, към който дадена негова променлива сочи.

3.3.2 Множествено обръщение към обект

Вече споменахме, че всеки обект може да бъде свързан с повече от една връзка към обекти от по-високо ниво. Защо се налага това?

Нека в резултат на някакво изчисление потребителят получи за резултат променливата *a*. Представен в системата, този резултат представлява структура навързани един над друг обекти. В този момент потребителят може да реши да продължи изчислението си и например да поиска сумата на *a* със *a*. В едно стандартно дърво на терм, това би означавало копиране на цялото дърво на *a*, за да може първото копие да се събере със второто. Това копиране означава и копиране на процедурата на получаване на приближения на числото, когато потребителят поиска резултат.

Не така обаче стоят нещата при пресмятане с вградени за един обичаен компютърен език числени типове. Когато потребителят има стойността на *a*, със сумирането на *a* със *a* той прави само една допълнителна операция; тъй като тук представената система има за цел да предложи възможно най-естествен начин за боравене с реални числа, е важно тя да не променя качествено сложността. Ако начинът за боравене с термове е подобен на представения в предния параграф, всяка допълнителна стъпка би удвоила едновременно времевата сложност на процедурата за извеждане на резултат и пространствената сложност на представянето на реални числа, т.е. те биха растели експоненциално спрямо броя операции, поискани от потребителя.

Алтернатива, която не предизвиква увеличаване на сложността, е всеки обект да има възможност да приеме повече от една връзка. Така описаният по-горе случай би предизвикал създаване само на един обект, този за операцията, който се обръща два пъти към сочения от променливата *a* обект. Този подход изисква контрол върху броя връзки, които даден обект има, за да не може свързани обекти да изчезват преди да са освободени от всички свои връзки, и, от друга страна, да няма разхищение на ресурси от изоставени несвързани обекти.

3.3.3 Кеширане на пресмятанията

Аналогичен подход е възприет и по отношение на процеса на пресмятане на приближения. Тъй като един вече съществуващ в системата обект има точно определен брой връзки с други обекти и потребителски променливи, може да се познае колко пъти ще бъде поискано от него приближение. Ако този брой е повече от веднъж е разумно след всяко пресмятане полученият резултат да се запомни (кешира), за да не се повтаря отново пресмятането, когато и следващият свързан обект поиска резултат.

След като броят поискани приближения от други обекти (за разлика от потребителски запитвания) стане равен на броя на връзките на обекта кешираната стойност вече не е нужна и може да се освободи. Сигурно е, че ако това стане, някъде на по-високо ниво стои друга кеширана стойност, чийто брой връзки не е изчерпан — най-малкото е сигурно, че обектът, представящ променливата, за която потребителят пита, няма да е загубил своето приближение, защото потребителските запитвания не се приспадат от броя връзки (потребителят може да поиска повече от един резултат; също така може да поиска резултат от обект, свързан единствено с променлива, след това да прехвърли връзката на друг обект, след което да загуби променливата — ясно е, че в тези случаи полученият веднъж резултат би се загубил, ако потребителските запитвания се броят, но това не трябва да става).

Кешираните приближения са временни обекти. При промяна на работната точност те се унищожават, защото техните резултати вече са безсмислени. След един такъв момент пресмятанията започват отново от най-ниско ниво.

3.3.4 Ефективност на представянето

Важен въпрос е дали наистина това представяне е достатъчно ефективно, за да се справи с евентуалните изисквания на потребителя.

Както вече бе споменато, извършването на нова операция от потребителя увеличава броя обекти, представени в системата, точно с един. От друга страна всеки обект, вече представен в системата, пресмята приближението си точно веднъж за дадена работна точност.

Ако една потребителска програма, работеща с вградени типове, извършва n на брой операции за постигане на даден резултат, то същата програма, работеща с реални числа, би заела допълнително пространство, пропорционално на n, и при дадена работна точност ще извежда приближения за брой единични операции, пропорционален на n.

3.4 Потребителската страна (*class Real*)

Начинът на използване на системата е прост. Потребителят има единствено достъп до класа *Real*, който осигурява нужните методи и операции. Единственият заглавен файл, нужен за работа със системата, е *real.h*.

Всички класове, функции и оператори в системата са отделени в собствен *namespace*, *RealLib*. Това се прави с цел ненатрапване на операциите върху реални числа на други части на потребителската програма. При желание на програмиста той може да направи всички имена от системата локални чрез използване на директивата *using namespace RealLib*.

3.4.1 Инициализация и приключване

Преди всяка работа със системата, тя трябва да бъде инициализирана чрез извикване на процедурата *Initialize*, декларирана така:

(57) *void RealLib*::*Initialize*(*unsigned precStart* = 10, *unsigned precMax* = 1000, *unsigned numEstAtStart* = 100);

Параметърът *precStart* определя началната точност на пресмятане, *precMax* максималната, а *numEstAtStart* броя приближения, за който се заема памет предварително. Работата с по-ниска точност от избраната по подразбиране няма смисъл, защото при по-ниска точност времето за обработка на представянето е повече от времето за същинското пресмятане. Избраната по подразбиране максимална точност отговаря на около 10000 верни десетични цифри и е вероятно достатъчна за много изчисления. Параметърът *numEstAtStart* има малко по-различна функция, той определя памет за колко приближения ще бъде заета в началото и с колко тя ще расте при нужда. Влияе върху съотношението заемана памет / производителност. Променя се обратно пропорционално на работната точност, защото ефектът му намалява с увеличаване на времето за обработка.

За освобождаване на заетите за пресмятане ресурси, след завършване на работа със системата тя трябва да бъде затворена с функцията *Finalize*, декларирана като:

unsigned RealLib::*Finalize*();

Връщаната стойност е текущата работна точност. Тя дава оценка за това какво е било нужно, за да се постигне търсеният резултат.

3.4.2 Типът *Real*

Класът *Real* наподобява вграден тип. Той има същите естествени операции като вграден тип и същите преобразувания от вградени типове, но за да се избегне загуба на точност, не предлага естествено конвертиране до вграден тип.

Публично достъпни членове на *Real* и операции за работа с класа са:

3.4.2.1 Конструктори, деструктори, присвояване

RealLib::*Real*::*Real*(*const Real* &*src*);

Копиращ конструктор, за да може един *Real* да се инициализира с друг *Real*.

RealLib::*Real*::*Real*(*const double src* = 0);

Конвертиращ конструктор от тип *double*. Приема представената от *val* стойност за точна.

RealLib::*Real*::*Real*(*const char* **src*);

Конвертиращ конструктор от текстов низ. Прави копие на низа и при всяка итерация за пресмятане оценя представеното число възможно най-добре.

RealLib::*Real*::*Real*(*OracleFunction* **src*);

Конвертиращ конструктор от функция оракул. Тук *OracleFunction* е дефинирана чрез *typedef const char* * (**RealLib*::*OracleFunction*)(*unsigned precision*);

OracleFunction връща десетичен низ за приближението, за който се приема, че се различава от представеното число с не повече от единица в най-малко значещата позиция от низа. Към *OracleFunction* се поставя изискването да дава все по-точни приближения с нарастване на параметъра *precision.*

Real& *RealLib*::*Real*::*operator* = (*const Real* &*rhs*);

Оператор за присвояване на стойност. Изтрива текущата стойност и задава нова.

RealLib::*Real*::~*Real*();

Деструктор, унищожава връзката между променливата и нейното представяне.

3.4.2.2 Константи

Real RealLib::*rpi*();

Константата $1/\pi \approx .31830988618379067153776752674502872406891929$1.

Real RealLib::*pi*();

Константата $\pi \approx 3.14159265358979323846264338327950288419716939$.

Real RealLib::*ln2*();

Константата $\ln 2 \approx .69314718055994530941723212145817656807550013$4.

Real RealLib::*ln10*();

Константата $\ln 10 \approx 2.30258509299404568401799145468436420760110148$.

3.4.2.3 Операции

Real RealLib::*Real*::*operator* – () *const*;

Real RealLib::*operator* + (*const Real* &*lhs, const Real* &*rhs*);

Real RealLib::*operator* - (*const Real* &*lhs, const Real* &*rhs*);

Real RealLib::*operator* * (*const Real* &*lhs, const Real* &*rhs*);

Real RealLib::*operator* / (*const Real* &*lhs, const Real* &*rhs*);

Унарен минус, събиране, изваждане, умножение и деление. Последните са реализирани като външни за класа функции за да позволяват повече автоматични преобразувания.

Real RealLib::*recip*(*const Real* &*arg*);

Реципрочно. Предполага, че аргументът не е нула. Ако е, оценяването му ще предизвиква ре-итерации с увеличаване на точността до достигане на максимума ѝ.

Real RealLib::*sqrt*(*const Real* &*arg*);

Квадратен корен. Може да даде неправилен резултат или *NaN* ако аргументът е по-малък от нула.

Real RealLib::*rsqrt*(*const Real* &*arg*);

Реципрочен квадратен корен. Предизвиква ре-итерации ако аргументът е равен на нула, а при некоректен вход резултатът, както и при *sqrt*, не е дефиниран.

Real RealLib::*ln*(*const Real &arg*);

Натурален логаритъм. Предизвиква ре-итерации ако аргументът е равен на нула, и не е определен при аргумент по-малък от нула.

Real RealLib::*exp*(*const Real &arg*);

Експонента. Резултатът е неопределен ако надхвърли възможностите за представяне от системата.

Real RealLib::*abs*(*const Real &arg*);

Абсолютна стойност. Ако аргументът включва интервал около нулата, резултатът се ограничава само до положителната си част.

Real RealLib::*atan2*(*const Real &y, const Real &x*);

Аргумент на комплексното число $x + iy = atan(y/x)$ за x > 0. Резултатът зависи от знака и на двата аргумента и е в интервала $(-\pi, +\pi)$. Функцията е ограничена до непрекъснатата част на дефиниционната си област, затова поведението на функцията е неопределено за $y = 0$ и $x \leq 0$.

Real RealLib::*tan*(*const Real &arg*);

Тангенс. Предизвиква ре-итерации ако аргументът е $k\pi + \pi / 2$ за някое цяло k.

Real RealLib::*cos*(*const Real &arg*);

Косинус. Резултатът е в интервала [-1, 1].

Real RealLib::*sin*(*const Real &arg*);

Синус. Резултатът е в интервала [-1, 1].

Real RealLib::*acos*(*const Real &arg*);

Аркускосинус. Не е дефиниран ако аргументът не е в интервала [-1, 1]. Интервалът на резултатът е [0, π].

Real RealLib::*asin*(*const Real &arg*);

Аркуссинус. Дефиниран за интервала [-1, 1]. Интервалът на резултатът е [-π/2, π/2].

Real RealLib::*atan*(*const Real &arg*);

Аркустангенс. Интервалът на резултатът е (-π/2, π/2).

Real RealLib::*cosh*(*const Real &arg*);

Хиперболичен косинус.

Real RealLib::*sinh*(*const Real &arg*);

Хиперболичен синус.

Real RealLib::*tanh*(*const Real &arg*);

Хиперболичен тангенс.

Real RealLib::*sq*(*const Real &arg*);

Квадрат.

3.4.2.4 Изискване на резултат

double RealLib::*Real*::*AsDouble*() *const*;

Конверсия до тип *double*. Тук системата може да промени текущата точност на пресмятане за да достигне достатъчна за точно представяне в *double* на резултата. Възможно е максималната точност на пресмятане да не стигне за получаване на резултат. В този случай функцията връща *NaN*.

char RealLib*::*Real*::*AsDecimal*(*char *buffer*, *unsigned lenwanted*) *const*;

Конверсия до текстов низ. Като параметри се подават буфер, във който стойността се записва, и брой "верни" десетични цифри на исканото число, т.е. които представят числото с разлика не по-голяма от 1 в най-малко значещата позиция. Също както при *AsDouble*, системата се опитва да достигне нужна точност за представяне. Когато това е невъзможно, връща резултат "*n/a*". Има и още един специален случай, който системата разглежда: близките до нула числа се изчисляват само до абсолютна грешка $10^{-lenwanted}$, след което се връща резултат "*probable zero*". Въпреки че този резултат не отговаря точно на изискванията, в практиката често възникват случаи, в които и истински нули трябва да могат да бъдат изведени като резултат. Изискването за верни цифри тук отговаря на ниска относителна грешка, която при нулев резултат не може да се постигне и би довел до непрекъснато увеличаване на точността до достигане на нейния максимум. От друга страна, работейки с абсолютна грешка в подобни случаи, отговор "*probable zero*" осигурява известна вероятност резултата да е нула, а от друга страна, възможност да бъдат извеждани истински нули.

char RealLib*::*NonZeroRealAsDecimal*(*const Real& arg*, *char *buffer*, *unsigned lenwanted*) *const*;

Ако потребителят знае, че даден резултат със сигурност не е нула, той може да използва тази функция за да получи стойността му с относителна грешка, по-малка от $10^{-lenwanted}$. Функцията е реализирана като сравнение с нула, последвано от *AsDecimal*.

bool operator < (const Real &lhs, const Real &rhs);

bool operator > (const Real &lhs, const Real &rhs);

bool operator != *(const Real &lhs, const Real &rhs);*

Сравнение, полуразрешимо. Връща резултат след като достигне достатъчна точност за да определи със сигурност истина или неистина, и *false*, ако максималната точност не е достатъчна.

Равенството никога не би могло да даде положителен отговор, а поради това и нестрогите сравнения са еквивалентни със строгите. Това прави реализация на другите три булеви оператора безсмислено.

3.4.3 Пример на потребителска програма

Ще представим един пример за потребителска програма и неин резултат.

Да предположим, че имаме за цел да пресметнем разликата между истинската стойност на константата *e* и нейното приближение чрез хиляда члена на представящия я ред на Тейлър.

Програмата има следния вид:

```
(1)   #include <stdio.h>
(2)   #include <time.h>
(3)   #include "real.h"
(4)
(5)   using namespace RealLib;
(6)
(7)   int main(int argc, char* argv[])
(8)   {
(9)     char buf[100];
(10)    clock_t starttime = clock();
(11)    Initialize();
(12)
(13)    {
(14)            Real ex(exp(Real(1.0)));
(15)            Real s(0.0), m(1.0);
(16)
(17)            for (int i=1; i<=1000; ++i) {
(18)                    s += m;
(19)                    m /= i;
(20)            }
(21)
(22)            Real diff(ex - s);
(23)
(24)            if (diff > 0.0) {
(25)                    printf("exp(1.0):\t%s\n", ex.AsDecimal(buf, 50));
(26)                    printf("taylor:\t%s\n", s.AsDecimal(buf, 50));
(27)                    printf("difference:\t%s\n", diff.AsDecimal(buf, 50));
(28)            }
(29)
(30)    }
(31)
(32)    unsigned pr = Finalize();
(33)
(34)    clock_t endtime = clock();
(35)    printf("prec: %d time elapsed: %lf\n", pr,
(36)                    double(endtime - starttime) / CLOCKS_PER_SEC);
(37)
(38)    getchar();
(39)    return 0;
(40)  }
```

Кои са моментите, които различават тази програма от нейно подобие, работещо с вградени типове?

- системата може да се използва единствено от *C++* програми, тъй като *C* не разбира от класове;
- *(3)* вмъква заглавния файл за системата;
- *(5)* прави локални за този файл имената в системата;
- *(11)* инициализира системата, *(32)* я затваря;
- *(13)* и *(30)* ограждат използването на реални променливи. Това не е задължително, но е препоръчително, защото гарантира освобождаването на реалните променливи при преминаване на *(30)*;
- на *(14)* и *(15)* е предпочетен вариантът на инициализиране на променлива чрез конструктор пред *C*-подобния с присвояване. По този начин се избягва излишно нулево инициализиране на променливите с последващо освобождаване. Това е предпочитания начин за инициализация в *C++* дори и за вградени типове ([14]). Също не е задължително;
- Пресмятането на редът на Тейлър *(17) – (20)* няма разлики със съответния си вариант с вградени типове;
- Извеждането на резултат *(24) – (28)* е специфично за работата със системата. Тук се прави изрична проверка *(24)* за ненулевост, в която потребителят е сигурен, но която е нужна на системата, за да се откаже от резултат "*probable zero*".

Примерен получен резултат при изпълнение на тази програма е:

```
exp(1.0):    .271828182845904523536028747135266249775724709e+1
taylor:      .271828182845904523536028747135266249775724709e+1
difference:  .248765330892737362051512455739417194009745e-2567
prec: 373 time elapsed: 286.440000
```

3.5 Други избрани моменти в реализацията

Тук ще опишем част от спомагателните модули и някои от използваните подходи за подобряване на реализацията.

3.5.1 Управление на блоковете с данни

За бързодействие на системата при ниска работна точност на пресмятания са от особена важност реализациите на операциите по създаване на нов обект с плаваща запетая и операциите по присвояване на стойност на обект.

Те са свързани с често заемане и освобождаване на памет, операции обикновено доста сложни, значително надминаващи по времетраене реализацията на произволна аритметична операция при ниска точност, и имащи ясно изразена тенденция на усложняване и забавяне при многократна употреба в следствие на получената фрагментация на паметта.

Блоковете памет, които се заделят в тази система за представяне на числа с плаваща запетая, са с фиксиран размер за текущата работна точност. Работата по заемане на памет за такива фиксирани по размер блокове може да бъде

значително по-проста, и да се ограничава до една-две минимални по сложност операции в голямата част от случаите.

За това се грижи класът *DataManager* и неговите подструктури *DataBuffer* и *FreeStack*.

При инициализация на модулът за работа с плаваща запетая предварително се заема голям буфер памет (*DataBuffer)* и се създава един стек с индекси на свободни блокове за числа с плаваща запетая (*FreeStack*).

Така при заявка за нов блок е достатъчно да се изтегли нов индекс от стека; освобождаването просто прибавя върнатия индекс обратно в стека. Ако стекът се окаже празен при операция по заемане на памет, той и буферът порастват, за да направят място за нови блокове. Последната операция е доста сложна, но се налага извънредно рядко.

Копирането на една променлива в друга също може да се извърши без нужда от копиране на данните, чрез използването на множество връзки към един реален обект. Този подход също е използван в системата.

3.5.2 Умножение чрез конволюция

Времето за умножение на две числа с плаваща запетая при точност n обикновено се оценява на $O(n^2)$. Съществуват обаче и по-добри алгоритми за умножение, като например алгоритъмът на Шьонхаге–Щрасен [12], който има сложност $O(n\log n\log\log n)$. Тук ще разгледаме едно опростяване на този алгоритъм.

Ако отделим обработката на преносът и закръгляването от операцията по умножение на две поредици 32-битови числа $\langle a_i\rangle$ и $\langle b_i\rangle$, то придобива следния вид:

$$c_k = \sum_{i=0}^{n} a_i^{*} b_{k-i}^{*}, k = 1..2n-1, \tag{58}$$

където

$$a_i^{*} = \begin{cases} 0, \text{ако } i < 0 \vee i \geq n \\ a_i, \text{ако } i \geq 0 \wedge i < n \end{cases} \tag{59}$$

и аналогично за b. Това е точно дефиницията на конволюция, използвана при дискретната обработка на сигнали. В теорията на дискретната обработка на сигнали ([10], [7]) тази операция е често срещана и добре изследвана. Един от резултатите, получени там, позволява да представим конволюцията като последователност от четири операции — две дискретни трансформации на Фурие в права посока, поелементно умножение, и една трансформация на Фурие в обратна посока.

Конволюцията има сложност $O(n^2)$, същата е и сложността на дискретната трансформация на Фурие.

Смисълът на това представяне се появява с така наречените бързи трансформации на Фурие [6], които изпълняват същата задача като дискретните, но за $O(n\log n)$ стъпки. Така общата сложност на конволюцията се свежда до

$$O(n\log n) + O(n\log n) + O(n) + O(n\log n) = O(n\log n). \tag{60}$$

Последващата обработка на преносът, за да завършим умножението, е със сложност $O(n)$ и не се отразява на общата сложност на умножението.

Реализацията на трансформация на Фурие изисква работа с числа с плаваща запетая с подходящ размер, за да могат точно да представят сумата от произведенията на цифрите на всяко представимо число. Коректна реализация изисква постепенно увеличаване на точността на пресмятане на трансформациите в зависимост от n или намаляване на дължината на обработваната дума, което добавя още един $\log n$ към сложността на процедурата.

За целите на тази система обработването на точности по-високи от определена граница не е нужно, тъй като времевите изисквания при високи такива не са по силите на съвременните компютри.

Затова дължината на думата, обработвана с конволюция, е фиксирана на 16 бита, а самите трансформации се извършват с точност *double*. Това е достатъчно за коректна обработка на точности до $2^{53-2.16} = 2^{21}$ 16-битови думи, или работна точност, в смисъла на използваната в системата, до над един милион 32-битови думи.

3.5.3 Регулиране на дълбочината на спускане в рекурсия

Извеждането на резултат и изтриването на ненужни структури на представяне на числа е свързано със рекурсивно спускане в дълбочина до достигане на обект с определени свойства.

Ако реализацията на това спускане се остави на компилатора, съществува вероятност паметта, запазена за стек на потребителската програма, да не достигне за извършване на операцията.

За да се избегне този ефект системата използва алгоритъм на ограничено спускане в рекурсия. Той се състои в спускане до определена дълбочина, запомняне на достигнатите обекти, и последващо спускане от запомнените обекти.

Паметта за временно запомняне на обектите не е ограничена така, както стека, и това позволява достигане на значително по-голяма дълбочина на спускане.

Важно е да се отбележи, че както при пресмятане на резултата без контрол на рекурсията, така и при реализиране на самия контрол вече посетените обекти трябва да се отбелязват; в противен случай сложността на процеса лесно може да стане експоненциална.

3.5.4 Скриване на детайлите на реализацията

Всяка реализация на модул в тази система зависи от възможно най-малко характеристики на станалите модули. Както и потребителският интерфейс няма достъп до вътрешното представяне на данните, така и модулите нямат контрол върху прекалено много аспекти на реализацията на използвани от тях модули на по-ниско ниво.

Ако модулите от по-високо ниво е работят с някаква абстракция на модулите от по-ниско, която включва само необходимите им методи и данни, можем да бъдем сигурни, че при замяна на съответния модул от по-ниско ниво системата ще работи коректно. Достатъчно е абстракцията, която заменящият предлага, да е същата като тази на заменения.

Този подход е реализиран на всички нива в системата. Цели модули от нея могат лесно да бъдат заменени с еквивалентни.

Така например модулът за управление на блоковете с данни предлага на използващия го модул единствено операции за заемане на памет за нова мантиса, освобождаване на вече заета, и адресиране на дума в нея. Това позволява лесно неговата реализация да бъде заменена с друга, евентуално по-ефективна или по-проста, например чрез директно извикване на *new* и *delete*.

3.5.5 Преносимост на кода

Голямо внимание при реализирането на системата е обърнато на преносимостта на написания код. Не се използват специфични за използваната операционна система, компилатор или процесорна платформа функции или код, включително и асемблер. Системата е изпробвана и работи еднакво добре под следните платформи (компилатор, операционна система, процесор):

- *MS Visual C++, Windows, Intel x86*
- *Cygwin GNU C++, Windows, Intel x86*
- *Intel C++ Optimizing Compiler, Windows, Intel x86*
- *CodeWarrior, Windows, Intel x86*
- *GNU C++, Linux, Intel x86*
- *CodeWarrior, MacOS, PowerPC*
- *GNU C++, Linux, PowerPC*
- *GNU C++, Solaris, Alpha*

За съжаление това прави кода за пресмятане на елементарните преобразувания събиране, изваждане, умножение и конволюция върху поредици от 32-битови цели числа значително по-бавни от евентуални техни реализации на асемблер или близко до асемблер ниво.

От друга страна модулният характер на системата позволява този недостатък да бъде избегнат и дори да бъде превърнат в преимущество чрез свързване на системата със специфични, специално оптимизирани за платформата, модули, като разработването им е значително улеснено от изобилието от платформено зависими силно оптимизирани решения за векторни операции и бързи трансформации на Фурие (например *Math Kernel Library* на *Intel*[1], *vDSP* на *Apple*[2], безплатната и преносима *FFTW* [3], реализациите по преносимия *VSIPL*[4] стандарт, и много други).

Повече информация за споменатите пакети може да се намери на:

[1] http://developer.intel.com/software/products/mkl/

[2] http://developer.apple.com/hardware/ve/vector_libraries.html

[3] http://www.fftw.org

[4] http://www.vsipl.org

4 Резултати

4.1 Пресмятане на константи и елементарни функции

Времената за пресмятане на елементарните функции с определена точност на Athlon 600MHz са представени в тази таблица:

десетични цифри	100, ms	1000, ms	2500, s	10000, s
събиране	0.032	0.044	0.0001	0.00022
умножение	0.035	0.55	0.0017	0.0149
реципрочно	0.140	7.1	0.033	0.307
квадратен корен	0.245	16	0.071	0.654
π	4	20	0.935	8.65
логаритъм	4.8	24.44	1.1	10.38
експонента	22	3390	20.33	211.13
аркустангенс	13	1275	5.93	55.09
косинус	61.4	8350	42.95	451.54

(кодът на програмата, използвана за тестване, се намира в *examples/benchops.cpp*)

Тази таблица позволява да заключим, че пресметнатите стойности за сложностите на елементарните функции отговарят на получените реални резултати.

4.2 Възможност за справяне с по-сложни задачи

Системата работи добри и при по-сложни за решаване задачи, включващи множество итерации и пресмятания в дълбочина. Няма проблем да бъде пресметната разликата между приближение с ред на Тейлър на *e* и неговата точна стойност (примерът от 3.4.3, *examples/taylor.cpp*), както и по-сложни задачи.

4.2.1 Безпроблемно използване на шаблони

Реализацията на интерфейсният клас се държи като естествен тип, което я прави добър кандидат за използване в шаблонни функции.

Най-естествен пример за шаблонно използване на *Real* е образуването на комплексни числа с помощта на класа *complex* от стандартната *C++* библиотека.

За съжаление някои от реализациите на този клас, разпространявани с известни компилатори, не са коректни. Например реализацията в *Microsoft Visual C++ 6* конвертира параметрите до *double* преди изчисленията. Това има унищожителен ефект върху точното представяне на числата.

Из изходния код на тази система се намира проста реализация на този клас, която се използва за реализиране на комплексните операции *ln* и *exp*. Тази реализация работи идеално с *Real*. Пример за това може да бъде намерен в *examples/complex.cpp*.

4.2.2 Конволюция на реални числа

Значително по-сложна операция, чиято реализация също присъства в кода на тази система, е конволюцията.

Модулът за конволюция, използван от системата, е също реализиран като шаблонен клас. Това позволява лесно да се изпробва и конволюция върху реални числа.

Времето, за което тази конволюция се изпълнява, заедно с времето за извеждане на ненулевите резултати с точността на *double* е представено в тази таблица:

дължина на векторите	64	256	1024	4096
време за конволюция, s	0.93	3.85	15.16	63.05

(*examples/convolution.cpp*)

Изчисленията с реални числа имат доста различно поведение и времева сложност от изчисленията с прости типове, затова и най-добрите алгоритми за вградени типове не отговарят на най-добрите за реални числа.

Така например бързите *Split-Radix* трансформации на Фурие [6] имат оптимална известна сложност в брой умножения, което ги прави оптимални за реални числа, въпреки че за прости типове те предлагат доста слаби резултати.

Създаването на вектор с ъгловите коефициенти при реални числа също може да бъде значително опростено. Тъй като $e^{\frac{i\omega}{2}} = \sqrt{e^{i\omega}}$, те могат да бъдат пресмятани без да се използват функциите синус и косинус въобще. При вградени типове това не може да се направи, защото точността се губи много бързо. При реални числа това не е така.

5 Какво може да бъде добавено

От теоретична гледна точка е важно да се докаже еквивалентността на класа функции, разглеждани тук с класа функции, дефинирани в TTE, или да се намери пример на функция, която е изчислима само в термините на TTE.

В реализацията има място за реализиране на операциите събиране и умножение с по-малка от работната точност, след което с модификации на изложените алгоритми за пресмятане на елементарните функции тяхната сложност може да бъде свалена до $M(n)$ за деление и квадратен корен и $M(n)\log n$ за останалите.

Реализацията работи в пъти по-бавно от други подобни решения поради платформената си независимост и неоптимизирания код; едно разширение към тази система, което да реализира зависими от платформата операции на най-ниско ниво може да разреши този проблем.

Разбира се, може да се помисли и за разширение на набора функции, които предлага системата; поради вече споменатия проблем с използване на класа *complex* от стандартната *C++* библиотека може да се предлага негова алтернатива заедно с пакета. Биха могли да се намерят и посочат и пакети за линейна алгебра и обработка на сигнали, които могат да работят върху представени от системата реални числа.

Библиография

[1] Bailey, D.H., Borwein, J.M. and Borwein, P.B., *Ramanujan, Modular Equations, and Approximations to Pi or How to compute One Billion Digits of Pi*, American Mathematical Monthly 96, (no. 3) 1989, 201–219
Достъпна в електронен вид на
http://www.cecm.sfu.ca/organics/papers/borwein/index.html

[2] Borwein, J.M., and Borwein, P.B., *Pi and the AGM — A study in analytic number theory and computational complexity*, Wiley, N.Y., 1987.

[3] Brent, R.P., *Fast multiple-precision evaluation of elementary functions*, Journal of the ACM 23 (1976) 242–251.
Достъпна в електронен вид на
http://web.comlab.ox.ac.uk/oucl/work/richard.brent/pub/pub034.html

[4] Brent, R.P., *Multiple-precision zero-finding methods and the complexity of elementary function evaluation*, Analytic Computational Complexity, Academic Press, New York, 1975, 151–176.
Достъпна в електронен вид на
http://web.comlab.ox.ac.uk/oucl/work/richard.brent/pub/pub028.html

[5] Gauß, C.F., *Werke*, Göttingen 1866-1933, Be 3, 361–403

[6] Loan, C.V., *Computational Frameworks for the Fast Fourier Transform*, Society for Industrial and Applied Mathematics, 1992

[7] Lyons, R.G., *Understanding Digital Signal Processing*, Addison Wesley Longman, 1997

[8] Mostowski, A., *A Lemma concerning recursive functions and its applications*, Bull. Acad. Polon. Sci. Cl. III 1 (1953) 277–280

[9] Müller, N., *The iRRAM: Exact Arithmetic in C++*, Workshop on Constructivity and Complexity in Analysis, Swansea, 2000
Достъпна в електронен вид на
http://www.informatik.uni-trier.de/~mueller/Forschung/irram.ps

[10] Oppenheim, A.V., Schafer, R.W., *Discrete-time Signal Processing*, Second Edition, Prentice-Hall, 1999

[11] Salamin, E., *Computation of π using arithmetic-geometric mean*, Mathematics of Computation 30 (1976) 565–570.

[12] Schönhage, A. and Strassen, V., *Schnelle multiplikation Grosser Zahlen*, Computing 7 (1971) 281–292.

[13] Skordev, D., *Characterization of the Computable Real Numbers by means of Primitive Recursive Functions*, Computability and Complexity in Analysis, Lecture Notes in Computer Science, Vol. 2064, Springer-Verlag, Berlin/Heidelberg 2001, 296–309

[14] Stroustrup, B., *The C++ Programming Language*, Third Edition, AT&T, 1997

[15] Weihrauch, K., *Computable Analisys. An Introduction*. Springer–Verlag, Berlin/Heidelberg 2000.