Об аппроксимации некоторых трансцендентных функций в компьютерной арифметике

Гаврилов К. В.

Новосибирский государственный технический университет, Новосибирск, Россия

Аннотация. В статье предложены алгоритмы быстрого вычисления некоторых трансцендентных функций из числа элементарных (синус, косинус, тангенс, логарифм и показательная функция), приведены соответствующие программные коды, обеспечивающие одинарную точность вычислений (стандарта *IEEE* 754), с использованием языка ассемблера для x86-совместимых компьютеров. Построение аппроксимаций основано на разложении функций в ряд Чебышёва (разложение по многочленам Чебышёва первого рода). Однако прямое применение данной техники не всегда приводит к достаточно эффективным решениям. Причины могут крыться как в недостаточной скорости сходимости ряда, так и в необходимости выполнения дополнительных вычислений для приведения аргумента функции в промежуток аппроксимации. В связи с этим в статье иллюстрируются некоторые приемы повышения эффективности вычислительных алгоритмов. Дополнительным средством повышения производительности служит использование при написании кода расширений *AVX* и *FMA* набора инструкций x86. Для оценки эффективности полученного кода проведено сравнение по времени работы предложенных функций с функциями, реализованными на базе команд математического сопроцессора, а также стандартными функциями языка *C++* в среде *Visual Studio* 2013.

Ключевые слова: аппроксимация функций, полиномы Чебышёва, компьютерная арифметика, схема Горнера, язык ассемблера, математический сопроцессор.

Введение

В современных компьютерах на базе архитектуры *x*86 аппаратно поддерживаются одинарная, двойная и расширенная точности вычислений с плавающей точкой. Данные форматы описаны в стандарте *IEEE* 754 и занимают в памяти соответственно 4, 8 и 10 байт [1]. Исторически вычисления с плавающей точкой производились средствами математического сопроцессора (*FPU*), функции которого сейчас интегрированы в центральный процессор [2]. Хотя математический сопроцессор поддерживает все перечисленные выше форматы данных, внутренние вычисления в нем производятся всегда с расширенной точностью. Данное обстоятельство, а также иногда недостаточно быстрая аппаратная реализация алгоритмов, заставляет разработчиков создавать более производительные алгоритмы для нужд тех приложений, где не важны компактность кода и высокая точность вычислений, но важна скорость. Задача создания быстрых вычислительных алгоритмов с одинарной и двойной точностью стала особенно актуальной после появления таких расширений набора инструкций, как 3*DNow*! (фирмы *AMD*), *SSE*, затем *SSE2* (фирмы *Intel*) и дальнейших, с помощью которых можно производить расчеты с одинарной и двойной точностью быстрее, чем средствами математического сопроцессора. Кроме того, данные расширения (их также называют *SIMD*-расширения) позволяют повысить производительность кода за счет векторизации вычислений [3]. Между тем большинство трансцендентных математических функций не были аппаратно реализованы в *SIMD*-расширениях, в связи с чем появились и продолжают появляться библиотеки подпрограмм, реализующие ускоренные версии таких функций, включая и стандартные библиотеки для языков программирования.

Одним из наиболее эффективных методов построения аппроксимаций функций является разложение функций в ряд по многочленам Чебышёва первого рода [4, 5]

$$T_n(t) = \frac{1}{2}\left[(t+\sqrt{t^2-1})^n + (t-\sqrt{t^2-1})^n\right] =$$

$$= \cos(n \arccos t),\ -1 \le t \le 1,\ n = 0, 1, 2, \ldots .$$

Данные многочлены являются ортогональными в весовом L_2 -пространстве с весом

$$1/\sqrt{1-t^2}\ .$$

Они также обладают свойством наименьшего уклонения от нуля на отрезке $[-1;1]$, благодаря чему аппроксимация на их основе по точности лишь незначительно уступает наилучшему равномерному приближению (среди многочленов той же степени), однако коэффициенты ряда Чебышёва вычисляются проще, чем для наилучшего равномерного приближения [6].

Запишем ряд Чебышёва для некоторой достаточно гладкой на отрезке $[-1;1]$ функции $f(t)$:

$$f(t) = \frac{c_0}{2} + \sum_{i=1}^{\infty} c_i T_i(t)\ .$$

Умножим это равенство скалярно на $T_n(t)$. При этом все слагаемые правой части обратятся в 0, кроме слагаемого с номером $i = n$, которое будет равно $c_n \pi/2$. Отсюда получаем выражение для коэффициентов разложения

$$c_n = \frac{2}{\pi}\int_{-1}^{1}\frac{f(t)T_n(t)}{\sqrt{1-t^2}}dt =$$

$$= \frac{2}{\pi}\int_{0}^{\pi} f(\cos u)\cos(nu)du\ ,\ n = 0, 1, 2, \ldots .$$

Поскольку $|T_n(t)| \le 1$, для достижения требуемой точности отбрасываем члены ряда, начиная с того, у которого величина c_n по модулю оказалась меньше допустимой погрешности.

Завершающим шагом в получении аппроксимации является приведение подобных в частичной сумме ряда Чебышёва. В результате получаем аппроксимирующий многочлен

$$f(t) \approx b_0 + b_1 t + \ldots + b_{n-1}t^{n-1},$$

который имеет ту же степень, что и номер последнего выписанного члена ряда. Если функция $f(t)$ четная, ее разложение будет содержать только четные степени аргумента; в этом случае для удобства нумерацию коэффициентов будем вести только по четным степеням.

В рамках статьи изложенные выкладки не будут подробно расписываться, будут только приводиться готовые коэффициенты b_i (или функции от них) в программном коде. Для их вычислений использовались системы *Wolfram Alpha* [7], *MathCad* [8] и онлайн-калькулятор многочленов Чебышёва [9]. В программной реализации функций вычисления аппроксимирующего полинома производятся по схеме Горнера [10].

Далее в статье речь будет идти, главным образом, о числах с одинарной точностью. Остановимся подробнее на том, какая это точность. Данный формат служит для записи чисел в двоичном экспоненциальном представлении (за исключением нуля и других специальных значений), и содержит знаковый бит, 8 битов порядка и 23 бита мантиссы. Фактически мантисса имеет 24 бита, поскольку старший ее бит присутствует неявно – в нормализованном числе он всегда установлен, и его нет необходимости хранить явно. Таким образом, ошибка величиной в единицу младшего разряда мантиссы дает относительную погрешность δ такую, что $2^{-24} < \delta \le 2^{-23}$ в зависимости от числа. В среднем можно считать $\delta \approx 2^{-23,5}$. Теперь рассмотрим десятичное число, содержащее m значащих цифр. Аналогично находим, что ошибка в единицу младшего разряда дает относительную погрешность около $10^{0,5-m}$. Приравнивая относительные погрешности, находим

$$m \approx 0{,}5 + 23{,}5\lg 2 \approx 7{,}574 .$$

Т.е. число с одинарной точностью эквивалентно по относительной погрешности числу, содержащему в среднем 7,6 значащих десятичных цифр. Обычно говорят о точности в 7-8 значащих цифр.

Заметим, что в обоих случаях можно было бы потребовать, чтобы ошибка составляла половину младшего разряда, как бывает при правильном округлении чисел. Нетрудно видеть, что это не влияет на результат. В общем случае для мантиссы размером k двоичных разрядов эквивалентное среднее количество значащих десятичных цифр определяется по формуле

$$m \approx 0{,}5 + (k - 0{,}5)\lg 2 .$$

1. Аппроксимация функций

1. Синус. Рассмотрим задачу вычисления функции $\sin x$. Учитывая периодичность функции, удобно аппроксимировать один ее полупериод. Это достигается приведением аргумента в диапазон $x \in [-\pi/2; \pi/2]$ и делением на $\pi/2$, т.е. полагаем

$$t = (x - k\pi)\,2/\pi ,$$

где $k \approx x/\pi$ – округленная до целых величина x/π. Также удобно понизить степень ряда, разделив функцию на t. Таким образом, будем строить аппроксимацию функции

$$f(t) = \frac{1}{t}\sin\frac{\pi t}{2},\ t \in [-1; 1] .$$

В силу ее четности ряд будет содержать только четные степени (всего понадобится 5 членов). Результат будет вычисляться по формуле

$$\sin x = (-1)^k \sin(x - k\pi) = (-1)^k t\, f(t) .$$

Стоит заметить, что при возрастании по модулю значения x будет происходить снижение точности функции, поскольку для аргумента тригонометрических функций важно не количество значащих цифр, а количество знаков дробной части. При $x \ge 2^{23}$ число не будет иметь дробной части, и тогда возвращаемые значения функции становятся похожими на случайные числа. К сожалению, в рамках фиксированного формата числа с этой ситуацией практически невозможно бороться, и мы оставим это свойство алгоритма как есть. Код функции показан на *Рис.* 1.

Между тем в алгоритме обработана ситуация, когда $|x| \ge \pi \cdot 2^{31}$ (приблизительно) или x не число. В этом случае функция возвращает без изменения ее аргумент. Частично проблема потери точности при больших значениях $|x|$ может быть решена за счет более точного вычисления t (например, с двойной точностью), правда, это не спасет от влияния ошибок округления исходных данных, представленных все равно с одинарной точностью. Сказанное относится ко всем рассмотренным тригонометрическим функциям.

Для сокращения количества вычислений в качестве аргумента аппроксимирующего многочлена берется не t, а $t/2$, поскольку вычисляется величина $x/\pi - k \in [-0{,}5; 0{,}5]$, в то время как аппроксимирующий многочлен определен на отрезке $[-1; 1]$. Поэтому все

коэффициенты b_i заранее умножены на 2^{2i+1}. Заметим также, что для построения аналогичной аппроксимации с двойной точностью требуется вычисление 9 коэффициентов.

```
float sin1(float x)
{
  static const float ct[6] =          // Таблица констант
  {
    0.318309886f,                     // 1/pi
    3.14159264f,                      // 2*b0
    -5.16771008f,                     // 8*b1
    2.55007752f,                      // 32*b2
    -0.598291286f,                    // 128*b3
    0.0776576603f                     // 512*b4
  };
  _asm
  {
    vmovss xmm0, [x]                  // В xmm0 загрузить x
    mov edx, offset ct                // В edx - адрес таблицы констант
    vmulss xmm2, xmm0, [edx]          // Получить x/pi в xmm2
    vcvtss2si ecx, xmm2               // Перевести x/pi в целое число с округлением
    vcvtsi2ss xmm1, xmm1, ecx         // xmm1=k - округлённое до целых x/pi
    dec ecx                           // of=1, если ecx=0x80000000 (т.е. переполнение или др. ошибки)
    jo sin_end                        // В случае слишком большого |x| вернуть аргумент без изменений
    shl ecx, 31                       // При чётном k установить знаковый бит ecx
    vsubss xmm1, xmm1, xmm2           // xmm1=-t/2 - минус дробная часть от x/pi в диапазоне [-0,5; 0,5]
    vmovd xmm2, ecx                   // В xmm2 знаковая маска результата
    mov ecx, 4                        // ecx=4 - инициализировать счётчик цикла
    vxorps xmm1, xmm1, xmm2           // xmm1=(-1)^k*t/2 - выставить правильный знак будущего результата
    vmovss xmm0, [edx+20]             // Инициализировать результат младшим коэффициентом 512*b4
    vmulss xmm2, xmm1, xmm1           // xmm2=t^2/4 - подготовить аргумент для вычисления многочлена
      sin_loop :                      // Цикл вычисления многочлена
    vfmadd213ss xmm0, xmm2, [edx+ecx*4] // Выполнить очередной шаг вычислений по схеме Горнера
    loop sin_loop                     // Учесть все 5 коэффициентов. В xmm0 формируется 2f(t)
    vmulss xmm0, xmm0, xmm1           // Умножить xmm0 на xmm1, получаем готовый результат (-1)^k*t*f(t)
      sin_end :                       // Перенос результата в стек математического сопроцессора
    vmovss [x], xmm0                  // Сохранить результат на месте переменной x
    fld [x]                           // Затем загрузить его в стек FPU
  }
}
```

Рис. 1. Листинг функции для вычисления синуса

2. Косинус. Рассмотрим задачу вычисления функции $\cos x$. Как и в предыдущем случае, положим

$$t = (x - k\pi)\,2/\pi\,,$$

где $k \approx x/\pi$ – округленная до целых величина x/π. Строится аппроксимация функции

$$f(t) = \cos\frac{\pi t}{2}\,,\ t \in [-1;1]\,.$$

В силу ее четности ряд будет содержать только четные степени (всего понадобится 6 членов). Результат будет вычисляться по формуле

$$\cos x = (-1)^k \cos(x - k\pi) = (-1)^k f(t)\,.$$

В остальном для данной аппроксимации справедливо все сказанное в отношении функции синус. Код функции показан на *Рис*. 2.

3. Тангенс. Рассмотрим задачу вычисления функции $\operatorname{tg} x$. Как и в случае других тригонометрических функций, в качестве промежутка аппроксимации выберем полупериод функции $x \in [-\pi/4;\,\pi/4]$. Тогда

$$t = 4x/\pi - 2k\,,\ t \in [-1;1]\,.$$

Здесь $k \approx 2x/\pi$ – округленная до целых величина $2x/\pi$. На первый взгляд было бы удобно разложить в ряд Чебышёва функцию

$$f(t) = \frac{1}{t}\operatorname{tg}\frac{\pi t}{4}\,.$$

Однако расчеты показывают, что для достижения одинарной точности в этом случае потребовалось бы вычислить 7 коэффициентов. Но если взять обратную величину от этой функции, т.е. положить

$$f(t) = t\operatorname{ctg}\frac{\pi t}{4}\,,$$

тогда достаточно будет вычислить лишь 5 коэффициентов. Таким образом, более удачный выбор функции $f(t)$ позволил ускорить сходимость ряда. Результат вычисляется по формуле

$$\operatorname{tg} x = (-1)^k [\operatorname{tg}(x - k\,\pi/2)]^{[(-1)^k]} =$$
$$= \begin{cases} -f(t)/t\,, & k \text{ нечетное}\,; \\ t/f(t)\,, & k \text{ четное}\,. \end{cases}$$

Правда, при этом для четных k вместо одной операции умножения потребуется более медленная операция деления. Временны́е потери от такой замены приблизительно соответствуют одной итерации цикла вычисления полинома, однако экономия у нас составляет 2 итерации, т.е. более существенная. Зато для нечетных k деление требуется в любом случае, и в результате мы получаем более заметный выигрыш. Код функции показан на *Рис*. 3.

```
float cos1(float x)
{
  static const float ct[7] =          // Таблица констант
  {
    0.318309886f,                     // 1/pi
    -1.00000000f,                     // -b0
    4.93480214f,                      // -4*b1
    -4.05870916f,                     // -16*b2
    1.33521200f,                      // -64*b3
    -0.234937314f,                    // -256*b4
    0.0243963244f                     // -1024*b5
  };
  _asm
  {
    vmovss xmm0, [x]                  // В xmm0 загрузить x
    mov edx, offset ct                // В edx - адрес таблицы констант
    vmulss xmm2, xmm0, [edx]          // Получить x/pi в xmm2
    vcvtss2si ecx, xmm2               // Перевести x/pi в целое число с округлением в ecx
    vcvtsi2ss xmm1, xmm1, ecx         // xmm1=k - округлённое до целых x/pi
    dec ecx                           // of=1, если ecx=0x80000000 (переполнение или др. ошибки)
    jo cos_end                        // В случае слишком большого |x| вернуть аргумент без изменений
    shl ecx, 31                       // Сформировать в ecx знаковую маску результата (0 - "минус")
    vsubss xmm2, xmm2, xmm1           // xmm2=t/2 - дробная часть от x/pi в диапазоне [-0,5; 0,5]
    vmovd xmm1, ecx                   // В xmm1 знаковая маска результата
    mov ecx, 5                        // ecx=5 - инициализировать счётчик цикла
    vmovss xmm0, [edx+24]             // Инициализировать результат младшим коэффициентом -1024*b5
    vmulss xmm2, xmm2, xmm2           // xmm2=t^2/4 - подготовить аргумент для вычисления многочлена
      cos_loop :                      // Цикл вычисления многочлена
    vfmadd213ss xmm0, xmm2, [edx+ecx*4] // Выполнить очередной шаг вычислений по схеме Горнера
    loop cos_loop                     // Учесть все 6 коэффициентов. Формируется xmm0=-f(t)
    vxorps xmm0, xmm0, xmm1           // xmm0=(-1)^k*f(t) - выставить правильный знак результата
      cos_end :                       // Перенос результата в стек математического сопроцессора
    vmovss [x], xmm0                  // Сохранить результат на месте переменной x
    fld [x]                           // Затем загрузить его в стек FPU
  }
}
```

Рис. 2. Листинг функции для вычисления косинуса

Для уменьшения количества операций в программе в качестве аргумента аппроксимирующего многочлена вместо t берется величина $t/2 = 2x/\pi - k \in [-0{,}5; 0{,}5]$, поэтому все коэффициенты b_i заранее умножены на 2^{2i-1}. Константы $b_0/2$ и $2/\pi$ совпали, поэтому для них в программе используется одна константа. Функция может быть легко переделана под вычисление котангенса: для этого достаточно поменять местами делимое и делитель в операции деления. Для построения аналогичной аппроксимации с двойной точностью потребовалось бы вычислить 10 коэффициентов разложения.

4. Логарифм. Рассмотрим задачу вычисления функции $\ln x$. Поскольку в компьютере используется двоичное представление данных, удобно производить расчет двоичного логарифма $\mathrm{lb}\, x$, а логарифмы по другим основаниям выражать через двоичный логарифм умножением на соответствующую константу.

Процедура деления или умножения вещественных чисел в компьютерных форматах на степень двойки очень простая и не приводит к потере точности – она сводится к изменению порядка числа. По этой причине промежуток аппроксимации двоичного логарифма наиболее экономично выбирать так, чтобы левая его граница была в 2 раза меньше правой. Так, данную функцию нередко аппроксимируют на отрезке $[1; 2]$. Но здесь есть важный нюанс. Аппроксимирующий многочлен обычно имеет свободный член, содержащий фиксированное количество знаков после запятой. Поэтому полученное приближенное значение логарифма будет содержать в лучшем случае столько же верных знаков после запятой. Но при $x \approx 1$ логарифм становится близким к 0, и при неизменном количестве знаков после запятой количество верных значащих цифр может

существенно уменьшаться, т.е. падает точность. Для борьбы с этим явлением используется аппроксимация функции $\mathrm{lb}(y+1)$ в окрестности точки $y=0$. Далее мы будем строить такую аппроксимацию, поэтому найденные коэффициенты аппроксимирующего многочлена можно использовать, в том числе, и для реализации функции $\mathrm{lb}(y+1)$.

Перейдем к построению аппроксимации. С учетом сказанного представляется разумным аппроксимировать функцию $\mathrm{lb}(y+1)$ на отрезке $y\in[-1/3;1/3]$. Тогда соответствующий отрезок для функции $\mathrm{lb}\,x$ имеет вид $x\in[2/3;4/3]$, т.е. границы отрезка отличаются в 2 раза, как и требуется. В этом случае полагаем

$$f(t)=\mathrm{lb}(t/3+1)/t\ ,\ t=3y\ ,$$

а результат получается по формуле

$$\mathrm{lb}\,x=\mathrm{lb}(y+1)=t\ f(t)+k\ ,$$

где k – число, на которое понадобилось уменьшить порядок величины x, чтобы привести ее в диапазон $[2/3;4/3]$. Здесь мы видим два недостатка. Во-первых, для приведения числа в указанный диапазон нужно хранить еще одну константу $4/3$ и сравнивать с ней число. Во-вторых, для достижения одинарной точности требуется 10 членов разложения, т.е. ряд Чебышёва для данной функции $f(t)$ сходится слишком медленно.

```
float tg1(float x)
{
  static const float ct[5] =            // Таблица констант
  {
    -0.00585941709f,                    // 128*b4
    0.636619770f,                       // b0/2 = 2/Pi
    -0.523598213f,                      // 2*b1
    -0.0861463538f,                     // 8*b2
    -0.0200433814f                      // 32*b3
  };
  _asm
  {
    vmovss xmm0, [x]                    // В xmm0 загрузить x
    mov edx, offset ct                  // В edx - адрес таблицы констант
    vmulss xmm1, xmm0, [edx+4]          // Разделить x на pi/2 и поместить в xmm1
    vcvtss2si ecx, xmm1                 // Перевести x*2/pi в целое число с округлением
    vcvtsi2ss xmm2, xmm2, ecx           // xmm2=k - округлённое до целых x*2/pi
    dec ecx                             // of=1, если ecx=0x80000000 (т.е. переполнение или др. ошибки)
    jo tg_end                           // В случае слишком большого |x| вернуть аргумент без изменений
    shl ecx, 31                         // При чётном k установить знаковый бит ecx и zf=0, иначе zf=1
    vsubss xmm2, xmm2, xmm1             // xmm2=-t/2 - минус дробная часть от x*2/pi в диапазоне [-0,5; 0,5]
    vmovd xmm1, ecx                     // В xmm1 знаковая маска результата (0 - "минус")
    vxorps xmm1, xmm1, xmm2             // xmm1=(-1)^k*t/2 - выставить правильный знак будущего результата
    vmovss xmm0, [edx]                  // Инициализировать результат младшим коэффициентом 128*b4
    vmulss xmm2, xmm2, xmm2             // xmm2=t^2/4 - подготовить аргумент для вычисления многочлена
    mov ecx, 4                          // ecx=4 - инициализировать счётчик цикла
      tg_loop :                         // Цикл вычисления многочлена
    vfmadd213ss xmm0, xmm2, [edx+ecx*4] // Выполнить очередной шаг вычислений по схеме Горнера
    loop tg_loop                        // Учесть все 5 коэффициентов. В xmm0 формируется f(t)/2
    jnz tg_div                          // Перейти к делению t/f(t), если k чётное, т.е. |tg x|<=1
    vmovaps xmm2, xmm0                  // Для нечётного k обменять местами регистры xmm0 и xmm1
    vmovaps xmm0, xmm1                  // В этом случае |tg x|>1, и нужно делить f(t)/t
    vmovaps xmm1, xmm2                  // Теперь в xmm0 - t/2, в xmm1 - f(t)/2
      tg_div :                          // Получение окончательного результата
    vdivss xmm0, xmm1, xmm0             // Найти f(t)/t, если k нечётное, или t/f(t), если k чётное
      tg_end :                          // Перенос результата в стек математического сопроцессора
    vmovss [x], xmm0                    // Сохранить результат на месте переменной x
    fld [x]                             // Затем загрузить его в стек FPU
  }
}
```

Рис. 3. Листинг функции для вычисления тангенса

Более высокой скоростью сходимости обладает ряд Чебышёва для функции

$$f(t)=\frac{1}{t}\mathrm{lb}\frac{a+t}{a-t}\ ,\ t=\frac{ay}{y+2}=a\frac{x-1}{x+1}\ ,$$

где $a>1$ – некоторая константа, $t\in[-1;1]$. Наиболее экономичный выбор: $a=3+\sqrt{8}$, которому соответствует диапазон $x\in[\sqrt{2}/2;\sqrt{2}]$. Такое значение a целесообразно использовать для вычислений с двойной или более высокой точностью, когда эффект от уменьшения количества членов ряда перевешивает расходы на операцию сравнения числа с константой $\sqrt{2}$. Для вычислений с одинарной точностью требуется 4

члена ряда, и, оказывается, что применение значения $a = 3 + \sqrt{8}$ не уменьшает необходимое количество членов разложения по сравнению с выбором $a = 5$, которому соответствует диапазон $y \in [-1/3; 1/2]$. Этот диапазон несколько избыточен: нам достаточно, чтобы $y \in [-1/4; 1/2]$, тогда аргумент функции $\mathrm{lb}\, x$ будет принадлежать отрезку $x \in [0{,}75; 1{,}5]$. Но зато сравнение числа, приведенного в диапазон $[1; 2]$, со значением 1,5 не составляет труда: достаточно проверить его второй по убыванию старшинства бит мантиссы. Таким образом, остановимся на последнем варианте. Конечный результат получается по формуле

$$\ln x = (t\ f(t) + k) \ln 2,$$

где k – число, на которое понадобилось уменьшить порядок величины x, чтобы привести ее в диапазон $[0{,}75; 1{,}5]$. Код функции показан на *Рис*. 4.

```
#define salc _emit 0xD6                 // Определяем команду salc её машинным кодом
float ln1(float x)
{
  static const float ct[6] =            // Таблица констант
  {
    1.0f,                               // 1
    0.576395561f,                       // 5^5*b2
    0.961802196f,                       // 5^3*b1
    2.88539007f,                        // 5*b0
    0.439077714f,                       // 5^7*b3
    0.693147181f                        // ln 2
  };
  _asm
    {
    mov eax, [x]                        // Прочитать в eax двоичное представление числа x
    cmp eax, 0x7F800000                 // Сравнить число со значением +Inf
    jge ln_inf                          // Если x=+Inf или +NaN, вернуть аргумент x без изменений
    ror eax, 23                         // Циклически сдвинуть число так, чтобы порядок оказался в al
    movzx ecx, al                       // Получить в ecx порядок числа со смещением +127
    jecxz ln_break                      // Перейти, если x=0 или x денормализованное (т.е. близко к 0)
    salc                                // al=-1, если установлен старший бит явной части мантиссы; иначе al=0
    sub cl, al                          // ecx=k+127, где k - добавка к двоичному логарифму
    add al, 127                         // Изменить порядок x так, чтобы привести его к диапазону 0.75<=x<1.5
    sub ecx, 127                        // ecx=k
    ror eax, 9                          // Получить в eax двоичное представление числа и проверить его знак
    vcvtsi2ss xmm3, xmm3, ecx           // xmm3=k, это добавка к двоичному логарифму
    sbb ecx, ecx                        // ecx=-1 (значение NaN), если x<0; или ecx=0, если x>0
    jz ln_ok                            // Перейти, если x>0, т.е. аргумент корректный. Ошибка, если x<=0
      ln_break :                        // Точка входа для случая x=0 или денормализованного x
    xchg eax, ecx                       // Далее результат будет формироваться в eax
    or eax, 0xFF800000                  // Сформировать в eax значение -Inf, если x=0 или x денормализованное
      ln_inf :                          // Точка входа для обработки случая x=Inf
    vmovd xmm0, eax                     // Перенести результат из eax в xmm0
    jmp ln_end                          // Перейти в конец функции
      ln_ok :                           // Продолжаем для случая x>0
    vmovd xmm0, eax                     // xmm0=x (приведённое значение)
    mov eax, offset ct                  // В eax адрес таблицы констант
    vaddss xmm1, xmm0, [eax]            // xmm1=x+1
    vsubss xmm0, xmm0, [eax]            // xmm0=x-1
    vdivss xmm0, xmm0, xmm1             // xmm0=(x-1)/(x+1)=t/5
    vmovss xmm1, [eax+16]               // Инициализировать сумму младшим коэффициентом: xmm1=5^7*b3
    vmulss xmm2, xmm0, xmm0             // xmm2=t^2/25 - подготовить аргумент для вычисления многочлена
      ln_loop :                         // Цикл вычисления многочлена (ecx пробегает от 1 до 3)
    inc ecx                             // Увеличить счётчик цикла на 1. Флаг pf установится только при ecx=3
    vfmadd213ss xmm1, xmm2, [eax+ecx*4] // Выполнить очередной шаг вычислений по схеме Горнера
    jnp ln_loop                         // Учесть все 4 коэффициента. В xmm1 формируется 5*f(t)
    vfmadd213ss xmm0, xmm1, xmm3        // xmm0=t*f(t)+k - получили готовый двоичный логарифм
    vmulss xmm0, xmm0, [eax+20]         // Перевести двоичный логарифм в натуральный
      ln_end :                          // Перенос результата в стек математического сопроцессора
    vmovss [x], xmm0                    // Сохранить результат на месте переменной x
    fld [x]                             // Затем загрузить его в стек FPU
  }
}
```

Рис. 4. Листинг функции для вычисления натурального логарифма

Для уменьшения количества операций в программе в качестве аргумента аппроксимирующего многочлена вместо t берется величина t/a, поэтому все коэффициенты b_i заранее умножены на a^{2i+1}. Также в программе используется непосредственная работа с порядком числа, и для ускорения работы исключена поддержка денормализованных чисел (такие числа считаются нулем). В случае вычисления логарифма с двойной точностью при $a = 3 + \sqrt{8}$ потребовалось бы 8 членов ряда.

5. Показательная функция. Рассмотрим задачу вычисления функции e^x. Прежде всего, заметим, что в компьютерной арифметике данную функцию следует вычислять через двоичную показательную функцию. Данная операция имеет тот нюанс, что при $x \approx 0$ теряются значащие цифры аргумента: они практически не влияют на результат, который лежит вблизи 1. Это может быть проблемой в тех формулах (например, в формулах для гиперболических функций), где прямо или косвенно используется выражение вида $e^x - 1$, которое не удалось бы хорошо вычислить с помощью функции e^x. Поэтому наряду с показательной функцией бывает полезно иметь функцию $e^x - 1$. Для ее вычисления строится аппроксимация

$$f(t) = t/(2^t - 1),\ t = x \operatorname{lb} e \in [-1;1].$$

Затем по ней вычисляется функция

$$e^x - 1 = 2^t - 1 = t/f(t).$$

Функция $f(t)$ здесь выбрана из соображений повышения скорости сходимости ряда Чебышёва. Такой выбор оправдан, несмотря на наличие сравнительно медленной операции деления. Отметим, что используемый здесь диапазон аппроксимации является избыточным, хотя это и полезно для прикладных целей. Так, подобная функция аппаратно реализована в математическом сопроцессоре (команда *f2xm1*).

Для нашей задачи достаточно аппроксимировать функцию

$$f(t) = t/(2^{t/2} - 1),\ t = 2(x \operatorname{lb} e - k) \in [-1;1],$$

где $k \approx x \operatorname{lb} e$ – округленное до целых значение $x \operatorname{lb} e$. Результат формируется по формуле

$$e^x = 2^k (t/f(t) + 1).$$

Для получения одинарной точности достаточно взять 4 члена разложения со степенями 0, 1, 2 и 4 (коэффициенты при нечетных степенях, начиная со степени 3, обращаются в 0). В случае использования двойной точности потребовалось бы 7 коэффициентов (при степенях 0, 1, 2, 4, 6, 8 и 10). Код функции показан на *Рис.* 5.

```
float exp1(float x)
{
  static const float ct[4] =          // Таблица констант
  {
    1.44269504f,                      // lb e
    2.88539009f,                      // b0
    0.115524160f,                     // 4*b2
    -0.000921114014f                  // 16*b4
  };
  _asm
  {
    vmovss xmm0, [x]                  // В xmm0 загрузить x
    mov edx, offset ct                // В edx - адрес таблицы констант
    vmulss xmm0, xmm0, [edx]          // Получить x*lb e в xmm0
    vcvtss2si eax, xmm0               // Перевести x*lb e в целое число с округлением
    vcvtsi2ss xmm1, xmm1, eax         // xmm1=k - округлённое до целых x*lb e
    dec eax                           // of=1, если eax=0x80000000 (т.е. переполнение или др. ошибки)
    jno exp_cont                      // Перейти для формирования 2^k, если |x| не слишком большой
    vmovmskps eax, xmm0               // Прочитать в младший бит eax знак числа x
    ror eax,1                         // Установить cf, если x<0. Флаг zf здесь всегда сброшен
      exp_break :                     // Далее формируется 0 (если x<0) или Inf (если x>0)
    vxorps xmm0, xmm0, xmm0           // Инициализировать результат нулём
    jbe exp_end                       // Перейти в конец, если результат действительно близок к 0
    vrcpss xmm0, xmm0, xmm0           // В случае x>0 поместить в xmm0 бесконечность (Inf)
    jmp exp_end                       // Перейти в конец
      exp_cont :                      // Продолжить для не слишком большого |x|
    sub eax, -128                     // Получить в eax порядок числа 2^k плюс смещение 127
    jle exp_break                     // Перейти для установки 0, если порядок большой отрицательный
    cmp eax, 254                      // Убедиться, что положительный порядок не слишком большой
    ja exp_break                      // Перейти для установки бесконечности в случае переполнения
    vsubss xmm0, xmm0, xmm1           // xmm0=t/2 - дробная часть от x*lb e в диапазоне [-0,5; 0,5]
    vmulss xmm2, xmm0, xmm0           // xmm2=t^2/4 - подготовить аргумент для вычисления многочлена
    shl eax, 23                       // Сформировать в eax двоичное представление числа 2^k
    vmovss xmm1, [edx+12]             // Инициализировать результат младшим коэффициентом 16*b4
    vfmadd213ss xmm1, xmm2, [edx+8]   // xmm1=4*b2+4*b4*t^2
    vfmsub213ss xmm1, xmm2, xmm0      // xmm1=b1*t+b2*t^2+b4*t^4
    vaddss xmm0, xmm0, xmm0           // xmm0=t
    vaddss xmm1, xmm1, [edx+4]        // xmm1=b0+b1*t+b2*t^2+b4*t^4=f(t)
    vmovd xmm2, eax                   // xmm2=2^k
    vdivss xmm0, xmm0, xmm1           // xmm0=t/f(t)
    vfmadd213ss xmm0, xmm2, xmm2      // xmm0=2^k*(t/f(t)+1)=e^x
      exp_end :                       // Перенос результата в стек математического сопроцессора
    vmovss [x], xmm0                  // Сохранить результат на месте переменной x
    fld [x]                           // Затем загрузить его в стек FPU
  }
}
```

Рис. 5. Листинг функции для вычисления показательной функции

При реализации алгоритма учитывалось, что константа $b_1 = -0{,}5$. Также, поскольку в качестве аргумента аппроксимирующего многочлена вместо t берется величина $t/2$, коэффициенты b_i были умножены на 2^i. Как и для тригонометрических функций, погрешность вычисления показательной функции несколько увеличивается с ростом $|x|$, поскольку точность результата здесь тоже зависит от количества знаков после запятой в исходных данных.

2. Оценка производительности

Для сравнения производительности различных алгоритмов использовался компьютер на базе процессора *AMD FX*-8350, работающий под управлением операционной системы *Microsoft Windows* 7. Тестовая программа выполняет 10000000 вызовов функции, и возвращает время выполнения в миллисекундах. Компилировалась программа в среде *Microsoft Visual Studio* 2013 с ключом «/o2», предназначенным для повышения производительности. Были рассмотрены 3 варианта функции: «*SIMD*» – предложенный в работе код на базе *SIMD*-инструкций, «*C++*» – встроенная стандартная функция *C++*, «*FPU*» – функция на базе команд математического сопроцессора. Запуск повторялся несколько раз, и типичный полученный результат для серии однородных испытаний заносился в *Таблицу* 1.

Таблица 1

Функция	SIMD	C++	FPU
sin	109	234	328
cos	109	234	328
tg	109	265	468
ln	109	156	764
exp	94	125	375

Заключение

В статье рассмотрены особенности и приемы построения на компьютере эффективных аппроксимаций функций синус, косинус, тангенс, логарифм и показательная функция, относящихся к трансцендентным функциям. Описанные подходы могут также применяться для аппроксимации других трансцендентных функций. Предложены алгоритмы на основе разложений в ряд Чебышёва, и программы, реализующие данные алгоритмы на языке *C++* с использованием вставок на языке ассемблера. Проведены исследования производительности предложенных алгоритмов. В результате исследования установлено, что все предложенные алгоритмы демонстрируют эффективность, превышающую эффективность соответствующих стандартных функций *C++* и функций, написанных с использованием команд математического сопроцессора.

Литература

[1] Зыков А. Г., Поляков В. И. Арифметические основы ЭВМ. – СПб: Университет ИТМО, 2016. – 140 с.

[2] Гагарина Л. Г., Кононова А. И. Архитектура вычислительных систем и Ассемблер с приложением методических указаний к лабораторным работам. Учебное пособие. – М.: СОЛОН-Пресс, 2019. – 368 с.

[3] Чередов А. Д., Мальчуков А. Н. Организация ЭВМ и систем: Учебное пособие. – Томский политехнический университет. – 4-е изд., перераб. и доп. – Томск: Изд-во Томского политехнического университета, 2016 – 236 с.

[4] Тиман А. Ф. Теория приближения функций действительного переменного. – М.: Гос. изд-во физ.-мат. лит., 1960. – 624 с.

[5] Дзядык В. К. Введение в теорию равномерного приближения функций полиномами. – М.: Наука, 1977, 512 с.

[6] Математика, ее содержание, методы и значение. Под ред. Александрова А. Д., Колмогорова А. Н., Лаврентьева М. А. – М.: Изд. Академии наук СССР, 1956; т. 2 – 397 с.

[7] URL: https://www.wolframalpha.com

[8] URL: https://www.mathcad.com/ru

[9] URL: https://www.kontrolnaya-rabota.ru/s/polinom/chebisheva

[10] Волков Е. А. Численные методы: Учеб. пособие для вузов. – 2-е изд., испр. – М.: Наука, 1987. – 248 с.

Константин Викторович Гаврилов, к.т.н., доцент каф. Автоматики НГТУ. Имеет 12 научных публикаций.
E-mail: got@ngs.ru

Статья получена 02.09.2020.

On Approximation of Some Transcendental Functions in Computer Arithmetic

Konstantin V. Gavrilov

Novosibirsk State Technical University, Novosibirsk, Russia

Abstract: The article offers algorithms for fast calculation of some transcendental functions from the number of elementary ones (sine, cosine, tangent, logarithm, and exponential function), and corresponding program codes are given that provide single-precision calculations (IEEE 754 standard) using the Assembly language for x86-compatible computers. The construction of approximations is based on the expansion of functions in the Chebyshev series (expansion in Chebyshev polynomials of the first kind). However, direct application of this technique does not always lead to sufficiently effective solutions. The reasons may lie both in the insufficient rate of convergence of the series, and in the need to perform additional calculations to bring the function argument into the approximation interval. In this regard, the article illustrates some techniques for improving the efficiency of computational algorithms. An additional way to improve performance is to use the x86 instruction set when writing AVX and FMA extensions. To evaluate the efficiency of the resulting code, the proposed functions were compared with functions implemented by the mathematical coprocessor commands, as well as standard C++ functions in Visual Studio 2013.

Key words: function approximation, Chebyshev polynomials, computer arithmetic, Gorner scheme, Assembly language, mathematical coprocessor.

REFERENCES

[1] Zikov A. G., Polyakov V. I. Arifmeticheskie osnovi EVM. – SPb: Universitet ITMO, 2016. – 140 s.

[2] Gagarina L. G., Kononova A. I. Arkhitektura vichislitelnikh sistem i Assembler s prilozheniem metodicheskikh ukazaniy k laboratornim rabotam. Uchebnoye posobiye. – M.: SOLON-Press, 2019. – 368 s.

[3] Cheredov A. D., Malchukov A. N. Organizatsiya EVM i sistem: Uchebnoye posobiye. – Tomskiy politeknicheskiy universitet. – 4-e izd., pererab. i dop. – Tomsk: Izd-vo Tomskogo politeknicheskogo universiteta, 2016 – 236 s.

[4] Timan A. F. Teoriya priblizheniya funktsiy deystvitelnogo peremennogo. – M.: Gos. izd-vo fiz.-mat. lit., 1960. – 624 s.

[5] Dzyadik V. K. Vvedeniye v teoriyu ravnomernogo priblizheniya funktsiy polinomami. – M.: Nauka, 1977, 512 s.

[6] Matematika, ee soderzhaniye, metodi i znacheniye. Pod red. Aleksandrova A. D., Kolmogorova A. N., Lavrentyeva M. A. – M.: Izd. Akademii nauk SSSR, 1956; t. 2 – 397 s.

[7] URL: https://www.wolframalpha.com

[8] URL: https://www.mathcad.com/ru

[9] URL: https://www.kontrolnaya-rabota.ru/s/polinom/chebisheva

[10] Volkov E. A. Chislenniye metodi: Ucheb. posobiye dlya vuzov. – 2-e izd., ispr. – M.: Nauka, 1987. – 248 s.

Konstantin V. Gavrilov, Candidate of Technical Sciences, Associate Professor of the Department of Automation of NSTU. Author of 12 scientific publications.
E-mail: got@ngs.ru

The paper has been received on 02/09/2020.