Государственное образовательное учреждение высшего профессионального образования
Московский государственный технический университет имени Н.Э. Баумана
Домашнее задание N1 по курсу «***Архитектура Компьютеров***»
Вариант: 2 от 2013-10-24
Студент: ***ИУ9-11 Борнев Антон***

Листинг 1: Наивная реализация

```
#define _USE_MATH_DEFINES
#define _GNU_SOURCE
#include <string.h>
#include <stdio.h>
#include <math.h>
#include <float.h>
#include <stdlib.h>
#include <stdbool.h>

double f(double x)
{
    return cos(x);
}

double F(double x)
{
    return sin(x);
}

double integrate(double left, double right, ulong steps)
{
    double sum = 0;
    double x   = left;
    double dx  = (right-left)/steps;
    while(x < right)
    {
        sum += 0.5 * (f(x) + f(x + dx)) * dx;
        x   += dx;
    }
    return  sum;
}
```

f — функция

F — первообразная

steps — число шагов

left — левая граница интегрирования

right — правая граница интегрирования

dx — высота

x — левая граница на *i-ом* шаге.

sum — интеграл

Обозначения, используемые в работе:

$a \oplus b$ — операция сложения двух чисел с плавающей точкой.

$a \ominus b$ — операция вычитания двух чисел с плавающей точкой.

- + / * — точные математические операции.

Определение 1.

*Пусть a и b — числа с плавающей точкой, * — какая-то бинарная операция в арифметике с плавающей точкой, ○ - операция * в поле* $\mathbb{R}$ *, тогда:*

$$\forall a, b : a * b = a \circ b(1 \pm \varepsilon)$$

Определение 2.

Машинный эпсилон (ε), является наименьшим таким числом, что $1+\varepsilon \neq 1$ *. Таким образом, числа a и b с отношением* $1 < \frac{a}{b} < 1+\varepsilon$ *считаются одинаковыми.*

(Для типа данных double $\varepsilon = 2^{-53} \approx 1.11 * 10^{-16}$ *)*

Предварительная подготовка:

Первое, в чем следует убедиться, так это в том, что данный компилятор генерирует код, соответствующий стандарту *IEEE 754*. В рамках данной работы был использован компилятор *gcc*. Для полного соответствия *IEEE 754* на официальной wiki рекомендуют использовать следующие флаги компиляции: *-frounding-math -fsignaling-nans*. Таким образом, переменная *QMAKE_CFLAGS*, отвечающая за флаги компиляции при использовании *qmake*, примет значения: *-O0 -std=c99 -frounding-math -fsignaling-nans*. Теперь, когда компилятор генерирует код соответствующий *IEEE 754*, можно приступать к дальнейшему анализу.

Запустим программу и посмотрим результаты выполнения:

Результаты 1: «Наивная реализация»

Количество шагов	*Результат программы*	*Ожидаемый результат*	*Абсолютная погрешность*
1	-6.283185307179586	0.000000000000000	6.283185307179586477e+00
2	0.000000000000000	0.000000000000000	2.449293598294706414e-16
3	-0.523598775598299	0.000000000000000	5.235987755982992826e-01
4	0.000000000000000	0.000000000000000	2.288475490443933335e-17
5	0.000000000000000	0.000000000000000	1.339070573669549874e-16
6	-0.785398163397448	0.000000000000000	7.853981633974485239e-01
7	0.000000000000000	0.000000000000000	1.991598500205919747e-16
8	0.000000000000000	0.000000000000000	2.449293598294706414e-16
9	0.000000000000000	0.000000000000000	8.813754755807632069e-17
10	0.000000000000000	0.000000000000000	1.339070573669549874e-16
11	-0.000000000000000	0.000000000000000	5.779962672170176036e-16
12	-0.488524308328426	0.000000000000000	4.885243083284261052e-01
13	0.000000000000000	0.000000000000000	1.436486987893341477e-16
14	0.000000000000000	0.000000000000000	1.991598500205919747e-16
15	0.000000000000001	0.000000000000000	6.987602111019124179e-16
16	0.000000000000000	0.000000000000000	7.839590613569716037e-17
17	-0.357120032891942	0.000000000000000	3.571200328919425744e-01
18	-0.000000000000000	0.000000000000000	4.114628135232441225e-16
19	-0.000000000000000	0.000000000000000	5.779962672170176036e-16
20	0.000000000000000	0.000000000000000	1.436486987893341477e-16
21	-0.000000000000001	0.000000000000000	8.555520233733067387e-16
22	-0.279814942303097	0.000000000000000	2.798149423030970813e-01
23	-0.268116805632010	0.000000000000000	2.681168056320099808e-01
24	0.000000000000001	0.000000000000000	8.652936647956858990e-16
25	-0.247379455879290	0.000000000000000	2.473794558792904948e-01
26	-0.238149859010715	0.000000000000000	2.381498590107149989e-01
27	0.000000000000000	0.000000000000000	1.894182085982128144e-16
28	0.000000000000000	0.000000000000000	2.288475490443933335e-17
29	-0.000000000000000	0.000000000000000	5.779962672170176036e-16
30	-0.207151132339387	0.000000000000000	2.071511323393869182e-01

Заметим, что результаты на некотором количестве шагов (выделены красным) сильно отличаются от ожидаемых. Рассмотрим количество шагов, равное одному и трем:
По формуле метода трапеций:

$$\int_{-\pi}^{\pi} \cos(x) = 0.5 * 2\pi(\cos(-\pi) + \cos(\pi)) \approx -6.2831853071795864769252$$

Следовательно, для количества шагов, равных одному, результат является верным (входит в погрешность метода). Для трех шагов:

$$\int_{-\pi}^{\pi} \cos(x) = 0.5 * \frac{2\pi}{3}(\cos(-\pi) + 2\cos(\frac{-\pi}{3}) + 2\cos(\frac{\pi}{3}) + \cos(\pi)) = 0$$

Что не совпадает с результатом программы, следовательно, можно сделать предположение, что допущена грубейшая ошибка в реализации. Так как в «наивной реализации» используется цикл *while*, а все операции с плавающей точкой выполняются с какой-то погрешностью, то сделав предположение, что на момент третьего шага x не равен правой границе, внесём изменение в программе: будем возвращать x после завершения цикла.

Листинг 2: Изменения в программе для отладки

```
double integrate(double left, double right, ulong steps,
                                                  double* real_right)
{
      double sum = 0;
      double x   = left;
      double dx  = (right-left)/steps;
      while(x < right)
      {
         sum += 0.5 * (f(x) + f(x + dx)) * dx;
         x   += dx;
      }
      (*real_right) = x; /* new */
      return  sum;
}
```

Результаты 2: (Листинг 2)

Количество шагов	*Правая граница после завершения программы*	*Результат программы*	*Ожидаемый результат*	*Абсолютная погрешность*
1	3.141592653589793	-6.283185307179586	0.000000000000000	6.283185307179586477e+00
2	3.141592653589793	0.000000000000000	0.000000000000000	2.449293598294706414e-16
3	5.235987755982988	-0.523598775598299	0.000000000000000	5.235987755982992826e-01
4	3.141592653589793	0.000000000000000	0.000000000000000	2.288475490443933335e-17
5	3.141592653589793	0.000000000000000	0.000000000000000	1.339070573669549874e-16
6	4.188790204786390	-0.785398163397448	0.000000000000000	7.853981633974485239e-01
7	3.141592653589793	0.000000000000000	0.000000000000000	1.991598500205919747e-16
8	3.141592653589793	0.000000000000000	0.000000000000000	2.449293598294706414e-16
9	3.141592653589794	0.000000000000000	0.000000000000000	8.813754755807632069e-17
10	3.141592653589793	0.000000000000000	0.000000000000000	1.339070573669549874e-16
11	3.141592653589794	-0.000000000000000	0.000000000000000	5.779962672170176036e-16
12	3.665191429188090	-0.488524308328426	0.000000000000000	4.885243083284261052e-01
13	3.141592653589793	0.000000000000000	0.000000000000000	1.436486987893341477e-16
14	3.141592653589793	0.000000000000000	0.000000000000000	1.991598500205919747e-16
15	3.141592653589793	0.000000000000001	0.000000000000000	6.987602111019124179e-16
16	3.141592653589793	0.000000000000000	0.000000000000000	7.839590613569716037e-17
17	3.511191789306239	-0.357120032891942	0.000000000000000	3.571200328919425744e-01
18	3.141592653589794	-0.000000000000000	0.000000000000000	4.114628135232441225e-16
19	3.141592653589794	-0.000000000000000	0.000000000000000	5.779962672170176036e-16
20	3.141592653589793	0.000000000000000	0.000000000000000	1.436486987893341477e-16
21	3.141592653589793	-0.000000000000001	0.000000000000000	8.555520233733067387e-16
22	3.427191985734319	-0.279814942303097	0.000000000000000	2.798149423030970813e-01
23	3.414774623467165	-0.268116805632010	0.000000000000000	2.681168056320099808e-01
24	3.141592653589794	0.000000000000001	0.000000000000000	8.652936647956858990e-16
25	3.392920065876975	-0.247379455879290	0.000000000000000	2.473794558792904948e-01
26	3.383253626942852	-0.238149859010715	0.000000000000000	2.381498590107149989e-01
27	3.141592653589794	0.000000000000000	0.000000000000000	1.894182085982128144e-16
28	3.141592653589793	0.000000000000000	0.000000000000000	2.288475490443933335e-17
29	3.141592653589794	-0.000000000000000	0.000000000000000	5.779962672170176036e-16
30	3.351032163829112	-0.207151132339387	0.000000000000000	2.071511323393869182e-01

Гипотеза 1:

Как видно из результатов (2), на тестах, на которых происходит «выброс», *x > right*, следовательно, происходит выход за правую границу интегрирования (например, для трёх шагов).

1. Оценим накапливаемую погрешность при суммировании $x = x + dx$. Так как каждая операция суммирования выполняется с некоторой ошибкой, то:

$$\sigma = ((((x+dx)(1\pm\varepsilon)+dx)(1\pm\varepsilon)...)+dx)(1\pm\varepsilon)$$

где σ — сумма, посчитанная с ошибкой. Вычитая из данной формулы точное значение суммы для *n-ого* шага *x + ndx*, приводя подобные, пренебрегая слагаемыми с *ε* больше первой степени, так как $\varepsilon \gg \varepsilon^2 \gg ... \gg \varepsilon^n$, упрощая и сделав грубую оценку, получим следующую формулу абсолютной ошибки:

$$\delta(n) = \pm \frac{\varepsilon(right(n+1) - left(n-1))}{2}$$

Следовательно, ожидается примерно линейный рост ошибки. По данной формуле для количества шагов, равных трём:

$\delta(3) \approx 1.04 \times 10^{-15}$, что не объясняет ошибку порядка $\Delta \approx 0.5$, отсюда можно сделать вывод, что проблема заключена не в суммировании $x = x + dx$.

2. Выход происходит при примерном равенстве *x* и *right*. Так как *x* все ещё считается меньше, чем *right*, то совершается ещё одна итерация цикла. Попробуем оценить, к чему это приведёт. Пусть *x* = *right*, тогда абсолютная ошибка от выхода за границу будет примерно:

$$0.5 * (f(x) + f(x+dx)) * dx$$

или для количества шагов, равных трём:

$$dx = 2\pi/3, x = \pi, f(x) = \cos(x)$$
$$S_\delta = 0.5(f(x) + f(x+dx))dx = -0.5\cos(2\frac{\pi}{3}) * 2\frac{\pi}{3} \approx -0.5235987755982988$$

Из результатов (2) видно, что это число совпадает с абсолютной ошибкой, следовательно, наша гипотеза подтвердилась. Если бы использовалось фиксированное количество шагов, то мы могли бы не дойти до правой границы или опять получить выход за неё, поэтому учтём это в следующей программе: теперь цикл будет останавливаться при *right < x2* (10 строка, листинг 3), а оставшаяся часть площади будет считаться за циклом (15 строка, листинг 3). Используя данную реализацию, мы также избавимся от потенциальной ошибки: если бы функция была не определена дальше правой границы.

Листинг 3: Изменение в программе после гипотезы 1

```
double integrate(double left, double right, ulong steps,
                                                double* real_right)
{
      double sum   = 0;
      double x2, x = left;
      double dx    = (right - left)/steps;
      for(;;)
      {
          x2 = x + dx;
          if (x2 > right) break;
          sum += 0.5 * (f(x) + f(x2)) * dx;
          x = x2;
      }
      sum += 0.5 * (f(right) + f(x)) * (right - x);
      (*real_right) = x + (right - x);
      return  sum;
}
```

Опыт 1:

Запустим программу и посмотрим на результаты:

Результаты 3: (Листинг 3)

Количество шагов	*Правая граница после завершения программы*	*Результат программы*	*Ожидаемый результат*	*Абсолютная погрешность*
1	3.141592653589793	-6.283185307179586	0.000000000000000	6.283185307179586477e+00
2	3.141592653589793	0.000000000000000	0.000000000000000	2.449293598294706414e-16
3	3.141592653589793	0.000000000000000	0.000000000000000	2.449293598294706414e-16
4	3.141592653589793	0.000000000000000	0.000000000000000	2.288475490443933335e-17
5	3.141592653589793	0.000000000000000	0.000000000000000	1.339070573669549874e-16
6	3.141592653589793	-0.000000000000000	0.000000000000000	4.669739647545019495e-16
7	3.141592653589793	0.000000000000000	0.000000000000000	1.991598500205919747e-16
8	3.141592653589793	0.000000000000000	0.000000000000000	2.449293598294706414e-16
9	3.141592653589793	0.000000000000001	0.000000000000000	3.101821524831076288e-16
10	3.141592653589793	0.000000000000000	0.000000000000000	1.339070573669549874e-16
11	3.141592653589793	0.000000000000000	0.000000000000000	1.991598500205919747e-16
12	3.141592653589793	-0.000000000000000	0.000000000000000	5.779962672170176036e-16
13	3.141592653589793	0.000000000000000	0.000000000000000	1.436486987893341477e-16
14	3.141592653589793	0.000000000000000	0.000000000000000	1.991598500205919747e-16
15	3.141592653589793	0.000000000000001	0.000000000000000	6.987602111019124179e-16
16	3.141592653589793	0.000000000000000	0.000000000000000	7.839590613569716037e-17
17	3.141592653589793	-0.000000000000001	0.000000000000000	9.665743258358223927e-16
18	3.141592653589793	0.000000000000000	0.000000000000000	7.839590613569716037e-17
19	3.141592653589793	0.000000000000000	0.000000000000000	1.894182085982128144e-16
20	3.141592653589793	0.000000000000000	0.000000000000000	1.436486987893341477e-16
21	3.141592653589793	-0.000000000000001	0.000000000000000	8.555202337330673 87e-16
22	3.141592653589793	-0.000000000000001	0.000000000000000	9.665743258358223927e-16
23	3.141592653589793	-0.000000000000000	0.000000000000000	7.445297209107910846e-16
25	3.141592653589793	-0.000000000000000	0.000000000000000	5.779962672170176036e-16
26	3.141592653589793	-0.000000000000001	0.000000000000000	1.327396808838998268e-15
30	3.141592653589793	-0.000000000000001	0.000000000000000	1.160863355145224787e-15

Опыт 2:

Проинтегрируем кусочно-заданную функцию:

$$f(x)=\begin{cases} g(x), x \le right \\ NaN, x > right \end{cases}$$

Так как все операции, выполняемые с *NaN*, возвращают *NaN*, то было бы сразу заметно, если бы интегрирование дало сбой. В ходе данного опыта было подтверждено, что операции не выводят нас за правую границу интегрирования (хотя это и было очевидно).

Вывод: после внесённых изменений интеграл на тех шагах, на которых были «выбросы», считается достаточно точно, серьёзная ошибка была устранена, следовательно, данные изменения целесообразно внести в программу.

Гипотеза 2

«Суммирование x += dx может привести к зацикливанию»

Пусть x — левая граница, dx — шаг интегрирования, тогда из определения 2:

$$\boxed{\text{Если } 1<\frac{x\oplus dx}{x}<1+\varepsilon \text{, то } x \text{ и } x\oplus dx \text{ представляются одинаково}}$$

Отсюда не сложно вывести формулу зависимости числа шагов от левой и правой границы, при которых будет происходить зацикливание, с точностью до коэффициента z:

$$z*\frac{(x+right)}{max(|x|,|right|)\varepsilon}<steps_{max}$$

Эксперимент:

Попробуем экспериментально найти верхнюю и нижнюю границу коэффициента z.

Листинг 4: Экспериментальная оценка коэффициента z.

```
#define ERROR(a,b) (((a) - (b))/(a) + 1)
#define SWAP(a,b,x) {x tmp; tmp = a; a = b; b = tmp; }
for (i = 1; i < 100000; i = i++) {
    double left  = fRand(DBL_MIN,DBL_MAX);
    double right = fRand(DBL_MIN,DBL_MAX);
    if (left > right) SWAP(left, right, double);
    unsigned long long N = (long double)(((right - left))) *
     (long double)(1llu << 53llu)/MAX(fabs(right),fabs(left));
    unsigned long long dN = N;
    double dx = (right - left) / N;
    while (left + dx != left) {
        dN = dN + pow(10,10);
        dx = (right - left) / dN;
    }
    double a = (double)(dN);
    double b = (double)(N);
    val = ERROR(a,b);
    if (val > max) max = val;
    if (val < min) min = val;
}
```

После нескольких запусков получены примерно следующие результаты:

Максимальное значение *z* стремится к 2, минимальное к 1.

Данный результат является чисто субъективным, но он позволяет проверить, что:

1) оценка действительно точна, с точностью до коэффициента $z \in [1,2)$

2) действительно происходит зацикливание.

Замечание: формула была проверена на функции интегрирования из листинга 1. В результате программа уходила в бесконечный цикл.

Заключение:

Можно сделать вывод, что гипотеза верна. Следует задокументировать, что использовать данную функцию на шагах больших $steps_{max}$, вычисленных при $z = 1$, не рекомендуется.

Изменим функцию интегрирования так, чтобы цикл не становился бесконечным, а именно добавим следующие строчки, осуществляющие проверку реальности выполнения расчёта с данным количеством шагов. Если это невозможно, то функция не зациклится, а вернёт *Not a Number*.

```
if (steps >= (right - left) * ((ulong)(1) << (ulong)(53)) *
MAX(fabs(left),fabs(right)))
        return NAN;
```

Замечание: *проверку можно реализовать и таким образом:*

```
double dx  = (right-left)/steps;
if (left + dx == left) return NAN;
```

Гипотеза 3

Рассмотрим суммирование вектора чисел $\{x_n\}$ в числах с плавающей точкой:

$$S(n) = x_0 \oplus x_1 \oplus x_3 \oplus \ldots \oplus x_n =$$
$$= x_0 + n x_0 \varepsilon + x_1 + n x_1 \varepsilon + x_2 + (n-1) x_2 + \ldots + x_n + x_n \varepsilon =$$
$$= \sum_{i=0}^{n} x_i + \sum_{i=0}^{n} (n - max(1, i) + 1) x_i \varepsilon$$

Из второй строки видно, что коэффициенты при первых *x*, например, x_1, x_0 — εn, в то время как коэффициент при x_n — ε. Отсюда можно сделать предположение, что при наивном суммировании выгодно складывать числа, начиная с малых по модулю, чтобы относительная ошибка суммирования была наименьшей. Также положительные числа лучше складывать отдельно от отрицательных.

Опыт:

Напишем программу для суммирования вектора случайных чисел и посчитаем сумму различными алгоритмами:

1) Наивный метод

2) Суммирование отсортированного массива по убыванию

3) Суммирование отсортированного массива по возрастанию

4) Для суммирования чисел будем использовать пирамиду: при каждой операции суммирования будем доставать два минимальных элемента из пирамиды, суммировать их и класть назад. Таким образом, каждый раз будут суммироваться два наименьших числа из вектора $\{x_n\}$. В ходе работы программы на векторе случайных чисел размером $n = 741777$ получены следующие результаты:

Таблица 1: Сравнение четырёх алгоритмов

	Result	*Absolute error*	*Result (second)*	*Absolute error (second)*
Naive	-5.176118010672767e+11	0.000122338533401489258	1.343202116610425e+06	2.97998212772654369e-08
Sorted(less)	-5.176118010672769e+11	6.07669353485107422e-05	1.343202116610455e+06	-2.50111042987555265e-12
Sorted(more)	-5.176118010672816e+11	0.00482150912284851074	1.343202116610420e+06	3.44564341503428295e-08
Heap sum	-5.176118010672768e+11	2.68220901489257812e-07	1.343202116610455e+06	-2.50111042987555265e-12
Long double naive	-5.176118010672767947e+11		1.343202116610454859e+06	

Почти во всех тестах наблюдаются равные результаты, даваемые *Heap sum* и сортировкой по возрастанию, но на некоторых замечено преимущество *Heap sum*. Также почти во всех тестах суммирование от большего к меньшему было хуже наивного.

Вывод: в ходе этого и других тестов можно подтвердить гипотезу 3: результат суммирования действительно улучшается, если сначала суммировать малые числа. Для дальнейшего анализа будем считать, что наша сумма считается достаточно точно. Для этого изменим тип переменной *sum* (листинг 3) на *long double*. Также добавим флаг компиляции *-m128bit-long-double*, чтобы получить более точные вычисления при использовании *long double*.

Гипотеза 4.

Погрешность возникает из-за неправильного вычисления площадей элементарных трапеций. Действительно, так как каждая операция выполняется с некоторой погрешностью, то из гипотезы 3 делаем предположение, что *dx* на момент шага *i* равен не *i*dx*.

В реализации из листинга 3 площадь трапеций считается по формуле:

$S_{\varphi}(i)=0.5*(f(x_i+x_{i_\delta})+f(x_{i+1}+x_{i+1_\delta}))*dx$, где $x_{i_\delta}, x_{i+1_\delta}$ — какая-то погрешность на *i-ом* шаге, в то время как точная формула:

$$S(i)=0.5*(f(x_i)+f(x_{i+1}))*dx$$

Посчитаем абсолютную погрешность по *Y*:

$$y_\Delta=\sum_{i=0}^{N-1} S(i)-\sum_{i=0}^{N-1} S_\varphi(i)$$

Так как метод трапеций аппроксимирует функцию на участке $[x_i, x_{i+1}]$ до прямой, то упростим формулу:

$$y_\Delta=\sum_{i=0}^{N-1} 0.5*dx(k_i x_i-k_i(x_i+x_{i_\delta})+k_i x_{i+1}-k_i(x_{i+1}+x_{i+1_\delta}))=-0.5*dx*\sum_{i=0}^{N-1} k_i*(x_{i_\delta}+x_{i+1_\delta})$$

Как видно из последней формулы, погрешность по *Y* линейным образом зависит от ошибки по *X* и от первой производной, следовательно, компенсируя ошибки по *X*, также компенсируется влияние от первой производной по *Y*. Поэтому будем считать высоту не константой, а *x2 - x*, где *x* — значение на *i-ом* шаге, а x2 — значение на *(i+1)-ом*.

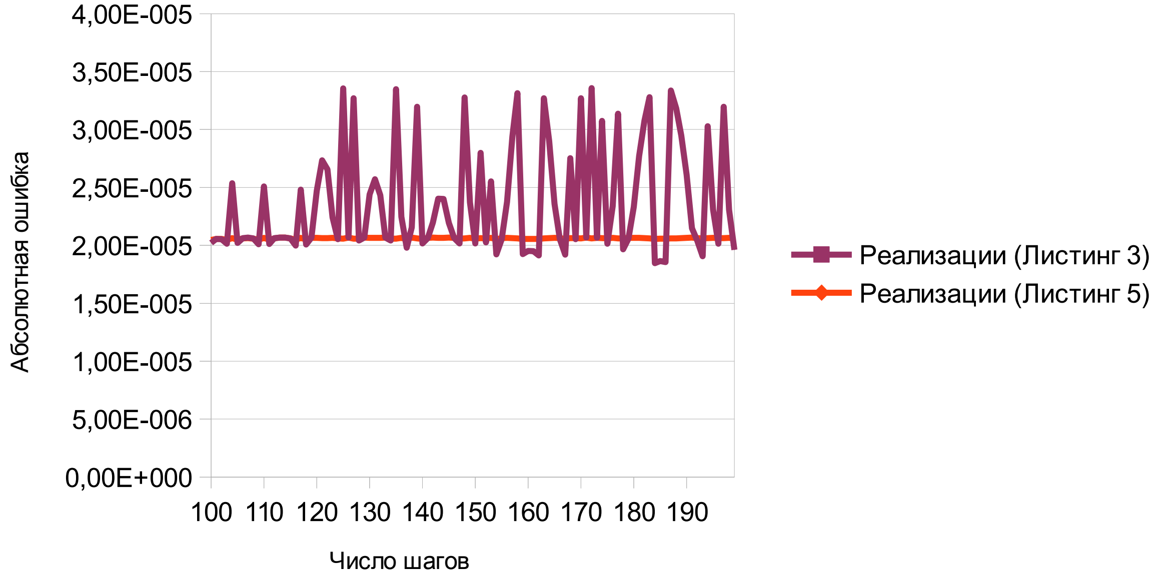
Перепишем функцию интегрирования с учётом всех гипотез.

```
Листинг 5: Финальная реализация
typedef double T;
typedef long double LT;
//Last version. Will return NAN when "steps" value two big
#define MAX(a,b) (((a) > (b))? (a) : (b))
T integrate(T left, T right, ulong steps, T* real_right) {
    if (steps >= ((long double)right - (long double)left) * ((ulong)(1) << (ulong)(53)) * MAX(fabsl((long double)left),fabsl((long double)right)))
        return NAN;
    LT sum = 0;
    T x2 = left, x = left;
    T dx  = (right  - left) / steps;
    for(; x2 < right; x = x2) {
        x2 = x + dx;
        sum += ((f(x) + f(x2)) * (x2-x)) * 0.5;
    }
    sum += (LT)(((f(right) + f(x)) * (right - x) * 0.5));
    (*real_right) = x + (right - x);
    return  sum;
}
```

Опыт:

Проинтегрируем функцию *y = cos(x)* на промежутке [100*π*;101*π*] двумя реализациями (из листинга 3 и листинга 5). Абсолютная погрешность метода больше $8*10^{-5}$ на всём промежутке интегрирования. Используем тип *float* для *X*-ов, так как проблемы для *float* будут видны намного раньше, чем для *double*. Для суммирования всё также будем использовать *long double*, чтобы максимально исключить накопление погрешности в суммировании по *Y*. Построим график по результатам абсолютной погрешности обеих реализаций:

График 1: Сравнение реализаций



Вывод:

Как можно видеть из последнего опыта, реализация из листинга 5 не совершает «выбросов» относительно прямой $y = 2*10^{-5}$, в то время как реализация из листинга 3 показывает намного худшие результаты. Так как в реализации из листинга 5 были учтены все ошибки, исследованные в предыдущих гипотезах, то целесообразно использовать её.