Государственное образовательное учреждение высшего профессионального образования
Московский государственный технический университет имени Н.Э. Баумана

Домашнее задание N1 по курсу

« Архитектура Компьютеров »

Вариант: 17 от 2014-10-29

Студент: ИУ9-11 Разборщикова Анастасия

Задача:
Разработать подпрограмму численного определения длины замкнутой кривой, заданной в параметрическом виде.

Начальные условия:
Тип кривой: эпитрохоида.
Параметрические уравнения кривой:

$$x = r(m+1)\cdot(\cos(m\varphi) - h\cdot\cos((m+1)\cdot\varphi))$$
$$y = r(m+1)\cdot(\sin(m\varphi) - h\cdot\sin((m+1)\cdot\varphi))$$

Начальные параметры:
$\varphi \in [\,0;\, 4\cdot\pi\,]$, r = 1, m = 0.5, h = 1.1.

Считается, что в любой точке кривой её производная ($dx/d\varphi$, $dy/d\varphi$) может быть вычислена аналитически и представлена в программе формулой или подпрограммой;

Формула определения длины кривой на отрезке [a; b]:

$$S = \int_a^b \sqrt{\left(\frac{\partial x}{\partial \varphi}\right)^2 + \left(\frac{\partial y}{\partial \varphi}\right)^2}\,\partial\varphi \qquad (1)$$

Метод интегрирования: метод прямоугольников.
Тип данных: float.

Часть 1. Предварительные аналитические вычисления.

Для удобства введем следующие обозначения:

S — вычисляемая длина кривой;

steps — требуемое число шагов;

n — пройденое число шагов;

x_i — есть полученное значение переменной x на данной итерации, $0 \le i \le steps$;

dx — значение шага, с которым изменяется переменная x;

dx_{real} — величина, на которую на самом деле изменилась переменная x (может отличаться от ожидаемого шага);

a, left, x_0 — левая граница промежутка интегрирования;

b, right, $x_{steps+1}$— правая граница промежутка интегрирования;

$f(x_{i+½})$ — значение функции, вычисленное в точке $x_i + ½dx$;

L — требуемая длина промежутка интегрирования [a ; b];

L_{real} — промежуток, который покрыла программа интегрирования;

ε — машинный эпсилон, относительная погрешность для используемого типа данных (данная величина объявлена как константа FLT_EPSILON в заголовочном файле <float.h>).

Нам требуется найти

$$S=\int_a^b f(x)\partial\varphi$$

Где

$$f(x)=\sqrt{\left(\frac{\partial x}{\partial\varphi}\right)^2+\left(\frac{\partial y}{\partial\varphi}\right)^2} \qquad (2)$$

Вычислим производные от x и y по φ:

$$x_\varphi'=r(m+1)(-m\cdot\sin(m\varphi)+h(m+1)\sin((m+1)\cdot\varphi))$$
$$y_\varphi'=r(m+1)(m\cdot\cos(m\varphi)-h(m+1)\cos((m+1)\cdot\varphi))$$

Чтобы вычислить сумму квадратов производных, вынесем за скобки общий множитель и сделаем замену:

$$(x_\varphi')^2+(y_\varphi')^2 = r^2(m+1)^2 Z,$$

где:

$$\begin{aligned} Z &= m^2\sin^2(m\varphi)-2mh(m+1)\sin(m\varphi)\sin((m+1)\varphi)+h^2(m+1)^2\sin^2((m+1)\varphi)+\\ &+m^2\cos^2(m\varphi)-2mh(m+1)\cos(m\varphi)\cos((m+1)\varphi)+h^2(m+1)^2\cos^2((m+1)\varphi)=\\ &= m^2[\sin^2(m\varphi)+\cos^2(m\varphi)]+h^2(m+1)^2[\sin^2((m+1)\varphi)+\cos^2((m+1)\varphi)]\\ &-2mh(m+1)\sin(m\varphi)\sin((m+1)\varphi)\\ &-2mh(m+1)\cos(m\varphi)\cos((m+1)\varphi) \end{aligned}$$

Делаем замену по тригонометрическим формулам:

$$\begin{aligned} \sin(\alpha)^2+\cos(\alpha)^2 &= 1\\ 2\sin(\alpha)\sin(\beta) &= \cos(\alpha-\beta)-\cos(\alpha+\beta)\\ 2\cos(\alpha)\cos(\beta) &= \cos(\alpha-\beta)+\cos(\alpha+\beta) \end{aligned}$$

$$\begin{aligned} Z &= m^2+h^2(m+1)^2-mh(m+1)(\cos(\varphi)-\cos((2m+1)\varphi)+\cos(\varphi)+\cos((2m+1)\varphi))\\ &= m^2+h^2(m+1)^2-2mh(m+1)\cos(\varphi) \end{aligned}$$

Итого имеем:

$$(x_\varphi')^2+(y_\varphi')^2 = r^2(m+1)^2 Z = r^2(m+1)^2(m^2+h^2(m+1)^2-2mh(m+1)\cos(\varphi))$$

$$f(x) = \sqrt{(x_\varphi')^2+(y_\varphi')^2} = r(m+1)\sqrt{(m^2+h^2(m+1)^2-2mh(m+1)\cos(\varphi))} \qquad (3)$$

Подставляем в формулу начальные значения и получаем:

$$\begin{aligned} S &= \int_0^{4\pi} f(x)\partial\varphi = \int_0^{4\pi} 1{,}5\sqrt{0{,}5^2+1{,}1^2\cdot1{,}5^2-2\cdot0{,}5\cdot1{,}1\cdot1{,}5\cos(\varphi)}\,\partial\varphi\\ &= \int_0^{4\pi} 1{,}5\sqrt{2{,}9725-1{,}65\cos(\varphi)}\,\partial\varphi = 1{,}5\int_0^{4\pi}\sqrt{2{,}9725-1{,}65\cos(\varphi)}\,\partial\varphi \end{aligned} \qquad (4)$$

$S = 31.81995$

(Для представления типа float необходима точность 6 знаков, но аналитически удалось вычислить значение лишь с точностью до 5 знаков, однако на данном этапе работы погрешность так велика, что и с этой точностью видны явные ошибки.)

Часть 2. Запуск первой реализации программы

Листинг 1: Исходный код наивной реализации программы

```
#define _USE_MATH_DEFINES
#define _GNU_SOURCE
#include <stdio.h>
#include <math.h>
#include <locale.h>


struct Curve
{
    float (*dx)(float fi, struct Curve *);
    float (*dy)(float fi, struct Curve *);
    float r, m, h;
} epitrochoid;


float dx_epitr(float fi, struct Curve *ep)
{
    float r = ep->r;
    float m = ep->m;
    float h = ep->h;
    return r*(m+1)*(-m*sin(m*fi)+h*(m+1)*sin((m+1)*fi));
}

float dy_epitr(float fi, struct Curve *ep)
{
    float r = ep->r;
    float m = ep->m;
    float h = ep->h;
    return r*(m+1)*(m*cos(m*fi)-h*(m+1)*cos((m+1)*fi));
}

float f(float fi, void *params)
{
    struct Curve *c = params;
    float dx = c->dx(fi, c);
    float dy = c->dy(fi, c);
    return sqrt(pow(dx, 2)+pow(dy, 2));
}

float integrate(float left, float right, unsigned long steps, float (*func)
(float, void *), void *params)
{
    float res = 0;
    float x = left;
    float dx = (right-left)/steps;
    for(x = left; x < right; x += dx){
```

```
        res += func(x + dx/2, r, m, h)*dx;
    }
    return res;
}



int main()
{
    long n;
    float L = 0, R = 4*M_PI;
    float V, V0 = 31.81995;

    /*Инициализация кривой начальными значениями*/
    epitrochoid.r = 1;
    epitrochoid.m = 0.5;
    epitrochoid.h = 1.1;
    epitrochoid.dx = dx_epitr;
    epitrochoid.dy = dy_epitr;
    /* точное решение */
    setlocale( LC_ALL, "" );
    /* использовать , или . для отделения дробной части */

    printf("Число шагов;Результат программы;Ожидаемый результат;Абсолютная
погрешность;Относительная погрешность\n");
    for( n = 1; n < 100; n += 1){
        V = integrate( L, R, n, f, r, m, h );
        /* приближенное решение для n шагов */
        printf( "%ld; %f; %f; %f; %G\n", n, V, V0, V-V0, (V-V0)/V0 );
        /* n и относительная ошибка */
    }
    return 0;
}
```

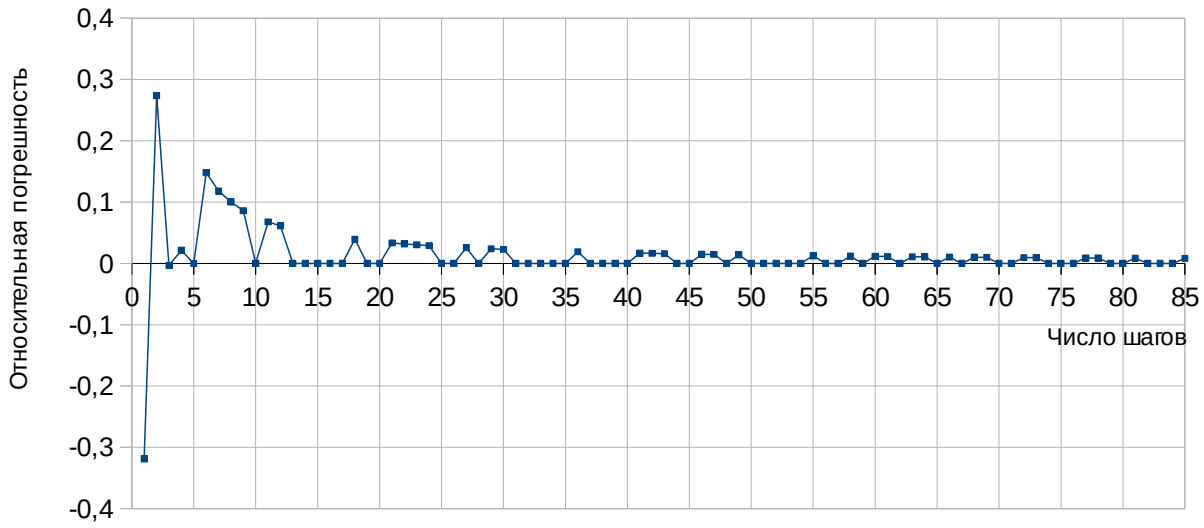

Рисунок 1: Результат выполнения наивной реализации программы

Таблица 1: Результат выполнения наивной реализации программы

Число шагов	Результат программы	Ожидаемый результат	Абсолютная погрешность	Относительная погрешность
1	21,676992	31,819950	-10,142958	-0,318761
2	40,526546	31,819950	8,706596	0,273621
3	31,713974	31,819950	-0,105976	-0,00333049
4	32,498413	31,819950	0,678463	0,0213219
...				
100	31,819958	31,81995	0,000008	2,40E-007

Попробуем проанализировать полученные данные и узнать, насколько отклоняются результаты выполнения программы от ожидаемой погрешности метода.

Из графика видно, что на первых шагах вычисленные значения имеют слишком большой разброс и отклонение не только от «эталонного» значения, но и от общей тенденции, что вызывает подозрение на ошибку реализации.

Убедимся в этом. Посчитаем вручную значение, которое должно получиться при используемом методе и ширине шага.

Выполним алгоритм (см. Листинг 1 на стр. 3), вычислив значение подынтегральной функции по формуле (4) на стр 2. Значение для steps равно 1. Тогда dx = right-left = 4*π, x + ½dx = 2* π

$$S = 1{,}5\sqrt{2{,}9725 - 1{,}64\cos(2\pi)}\cdot 4\pi = 1{,}5\sqrt{2{,}9725 - 1{,}64}\cdot 4\pi \approx 1{,}5\cdot 1{,}15130360896\cdot 1{,}5\cdot 4\pi = 21{,}7015617597$$

Что приблизительно объясняет полученный результат 21,676992.

Аналогично для 2-х шагов:

dx = 2π,

$x_1 = \pi$,

$x_2 = 3\pi$.

Заметим предварительно, что cos(π) = cos(3π) = -1;

$$S = S_1 + S_2 = 2\pi * 1{,}5 * \sqrt{2{,}9725 - 1{,}64 * \cos(\pi)} + 2\pi * 1{,}5 * \sqrt{2{,}9725 - 1{,}64 * \cos(3\pi)} = 4\pi * 1{,}5\left(\sqrt{2{,}9725 + 1{,}64}\right) = 4\pi * 1{,}5 * 2{,}14767315949 = 40{,}482685321$$

Результат выполнения программы - 40,526546.

Для 3-х шагов:

dx = 4/3*π = 4,18879020478639,

x_1 = 2/3* π = 2,09439510239,

x_2 = 2π = 6,28318530718,

x_3 = 3 1/3* π = 10,471975512.

cos(2π/3) = -0,5, cos(2π) = 1, cos(3 1/3* π) = cos(-π /3) = -0.5

$$S = S_1+S_2+S_3=4/3*\pi*1.5*\sqrt{2,9725-1,64*\cos(2\pi/3)}+$$
$$+4/3*\pi*1.5*\sqrt{2,9725-1,64*\cos(2\pi)}+4/3*\pi*1.5*\sqrt{2,9725-1,64*\cos(31/3*\pi)} =$$
$$= 12,2360900113+7,25292985335+12,2360900113 = 31,725109876$$

Результат программы — 31,713974.

Что означает, что скачки на первых итерациях объясняются погрешностью метода.

Как видно из таблицы 1 и рисунка 1 выше, в программе, помимо погрешности метода, присутствуют ошибки реализации. Явно прослеживающаяся тенденция нарушается в некоторых точках — очевидно, в некоторых точках прибавляется лишняя площадь.

Попробуем проанализировать, отчего это происходит.

Часть 3. Первичный анализ ошибок.

Пункт 1. Общий анализ и построение плана действий.

Посмотрим внимательно на код программы (Листинг 1) и попробуем понять, какие переменные вносят наибольший вклад в накопление ошибок, точность вычисления каких из них мы можем повысить.

В цикле многократно изменяются от итерации к итерации только две переменные: x и res. Переменная res зависит от правильности вычисления f(x) и dx, точнее от правомерности вычисления произведения от данной пары f(x) и dx (об этом позже).

На точность работы подынтегральной функции мы считаем, что повлиять не можем (аналитическое задание параметров для f(x) уникально для каждой кривой, следовательно, разрабатывая универсальную функцию вычисления длины кривой мы не углубляемся в оптимизацию этой функции, а пытаемся сделать устойчивой к накоплению ошибок функцию интегрирования).

Точность соответствия полученного x ожидаемому, от которого вычисляется f(x) на данной итерации, в свою очередь, зависит от точности представления x_0 = left и точности вычисления dx. Напомним, что в исходном коде присутствуют такие строки:

```
for(x = left; x < right; x += dx){
        res += func(x + dx/2, ...)*dx;
}
```

Пока имеем следующую цепочку взаимных зависимостей переменных (в дальнейшем этот список может быть уточнен):

1) res зависит dx и опосредованно зависит от x (от правомерности вычисления подынтегральной функции в данной точке);
2) x зависит от dx;
3) dx зависит от L, steps.

Поэтому наша задача на данном этапе:

1) исследовать ошибки, связанные с погрешностью вычисления и неправильным представлением переменной dx;

2) уменьшить величину ошибки, получаемой при вычислении суммы x+dx;
3) устранить погрешности, получающиеся при суммировании итоговых значений переменной res.

То есть, не имеет смысла бороться с ошибками при суммировании вычисленных на каждом шаге значений переменной res, если они изначально вычислялись на неправильном промежутке и не в тех точках, которые требовались, поэтому важно сначала рассмотреть ошибки, появляющиеся при вычислении переменной x.

Пункт 2. Анализ погрешностей при вычислении x.

Рассмотрим подробнее движение по оси Ox.

Цикл, в котором происходит суммирование площади, продолжается до тех пор, пока верно условие x < right (1) (см. Листинг 1 стр. 3). Это условие будет корректно для арифметики действительных чисел, где если dx вычисляется как: dx = (left-right)/steps, то число left-right кратно dx. Однако из-за ограничений точности представлений числа в машине используемое значение смещение аргумента на следующей итерации не равно ожидаемому, т.е.

$$\frac{L_{real}}{n} = dx_{real} = dx(1 \pm \varepsilon) \quad (5)$$

Тогда через n = steps итераций, когда ожидается, что вся площадь будет покрыта, действительный задействованный интервал будет равен:

$$x_n = x_0 + dx_{real} \cdot n = x_0 + dx(1 \pm \varepsilon) \cdot n$$

$$L_{real} = x_n - x_0 = dx_{real} \cdot n = dx\,(1 \pm \varepsilon) \cdot n = L\,(1 \pm \varepsilon) \neq L \ .$$

Более того, каждая операция суммирования также выполняется с ошибкой и следующий шаг вычисляется с погрешностью:

$$x_i = (x_{i-1} + dx)(1 \pm \varepsilon) \quad (6)$$

для $1 \le i \le steps$.

Откуда можно сразу же сделать следующий вывод.

Гипотеза 1.

Выход из цикла по условию x > right (1) не гарантирует, что на последней итерации правая граница участка, на котором вычисляется площадь, сумма x+dx будет не больше, чем right — правая граница интегрирования. То есть, если на последней итерации right-dx < x < right, то x+dx > right, и мы прибавляем лишнюю площадь.

Более того, можно сделать вывод, что если x таков, что x + ½dx > right, то будет произведена попытка вычислить значение подынтегральной функции на участке, где она может быть не определена.

Проверим это. Включим в код процедуры integrate (см. Листинг 1) счетчик количества реально выполненных итераций и вывод относительной ошибки правой границы. При запуске программы получаем следующий график (см. Рисунок 2):

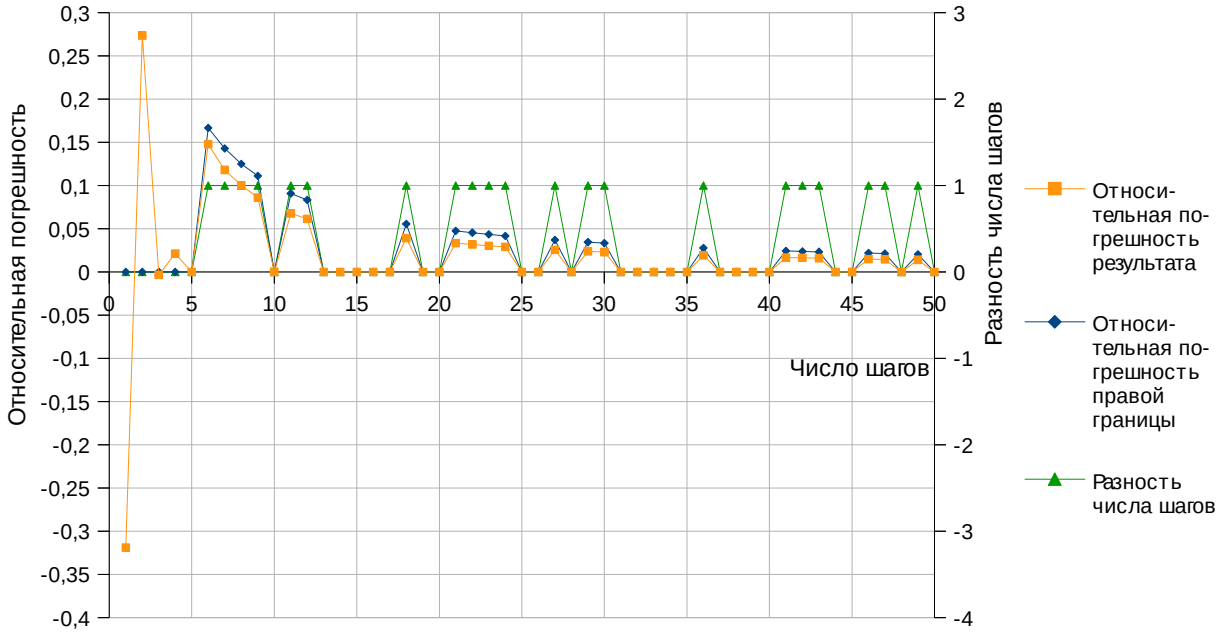

Рисунок 2: Результат запуска с выводом реальной правой границы интегрирования и числа выполненных итераций

На самом деле этот график не вполне точен: правая граница не равна ожидаемой на всех тестах, кроме 1-5 и 15-го. Но в остальных точках, где программа и LibreOffice Calc возвращают в качестве относительной погрешности 0, абсолютная погрешность меньше 10^{-5} степени, то есть они отличаются лишь на последний знак. Но эта погрешность присутствует и она всегда положительна. Однако и этот график достаточно показателен, чтобы увидеть зависимость между резкими выбросами на графике погрешности результата программы и большими выходами за правую границу интегрирования. В этих точках сделано больше шагов, чем требовалось, но не более, чем на единицу. Можно попытаться решить эту проблему наивно, в голову сразу приходят 2 решения:

1) сделать выход из цикла по счетчику,

2) усложнить условие, сделав выход за один шаг до правой границы (x+dx > right (2)), а оставшийся интервал посчитать за циклом на промежутке уже не dx, а right-x, что обеспечит полное покрытие функцией необходимого промежутка интегрирования.

Первый метод, как можно сразу заметить, сможет избавить только от возможности попытаться вычислить подынтегральную функцию на участке, где она не определена. Однако, в тех точках, где мы получали выход за правую границу интегрирования, очевидно, будет неполное покрытие, что может привести к заниженному результату.

Второй метод кажется более перспективным и при его реализации, действительно, убираются резкие скачки даже в случаях, когда все равно произведено большее количество итераций. Однако на данном этапе не будем забегать вперед и производить его внедрение, так как вначале нужно понять, как именно получается данное расхождение, отчего оно зависит и не выявятся ли прочие ошибки или более оптимальные пути решения проблемы. (Как оказалось впоследствии, это усовершенствование пришлось внедрить, см. Листинг 6 ниже).

Итак, возвращаясь к началу пункта, на данном этапе попытаемся проанализировать две ошибки, влияющие на вычисление аргумента на следующей итерации:

1) неправильно посчитанное смещение аргумента dx;

2) ошибки, накапливающиеся при суммировании (x+dx) на каждой следующей итерации;

3) итоговая погрешность, получаемая при наложении погрешностей 1) и 2).

В дальнейшем будем обозначать соответствующие абсолютные ошибки как $\Delta_{(1)}$, $\Delta_{(2)}$, и $\Delta_{(3)}$ соответственно.

Рассмотрим каждую из них, ее вклад в общую погрешность, установим, зависят ли ошибка (1) и ошибка (2) друг от друга. Только после этого можно будет ставить тесты на выявление каждой из них.

Во-первых, сразу можно сказать, что так как ошибки при суммировании в цикле накапливаются «как снежный ком», то вклад второй ошибки больший, поэтому сначала рассмотрим ее.

Выведение формул погрешностей, с которыми вычисляется x.

Во-первых, сразу можно сказать, что, так как ошибки при суммировании в цикле накапливаются «как снежный ком», то вклад второй ошибки больший, поэтому сначала рассмотрим ее.

Посчитаем, как данная ошибка влияет на вычисление аргумента на последней итерации:

Для одного шага согласно формуле (6) (см. стр. 7) погрешность составляет:

$$x_1 = (x_{i-1}+dx)(1\pm\varepsilon) = x_0+dx\pm\varepsilon(x_0+dx) .$$

Аналогично для 2-х шагов:

$$\begin{aligned} x_2 &= ((x_0+dx)(1\pm\varepsilon)+dx)(1\pm\varepsilon) = \\ &= (x_0+dx\pm\varepsilon(x_0+dx))+dx\pm\varepsilon(x+dx\pm\varepsilon(x_0+dx)+dx) = \\ &= x_0+2\,dx\pm\varepsilon(x_0+dx)\pm\varepsilon(x_0+2\,dx\pm\varepsilon(x_0+dx)) \end{aligned}$$

Заметим, что после раскрытия скобок в выражении

$$\varepsilon(x_0+2\,dx\pm\varepsilon(x_0+dx))$$

мы получим:

$$\varepsilon x_0+2\varepsilon\,dx\pm\varepsilon^2(x_0+dx)$$

Так как $\varepsilon \ll 1$ (порядка 2^{-23} для типа float), то $\varepsilon \gg \varepsilon^2 \gg \varepsilon^3 \gg \ldots \gg \varepsilon^n$ и эпсилон в степени выше единицы не окажет влияния на погрешность вычислений, поэтому слагаемыми, содержащими такие степени эпсилон, можно пренебречь, то есть, мы считаем, что:

$$\varepsilon x_0+2\varepsilon\,dx\pm\varepsilon^2(x_0+dx) \approx \varepsilon x_0+2\varepsilon\,dx$$

Короче:

$$\varepsilon(x_0+2\,dx\pm\varepsilon(x+dx)) = \varepsilon(x_0+2\,dx) \qquad (7)$$

И аналогично мы можем поступать со всеми выражениями, содержащими ε, которые расположены в скобках, умноженных, в свою очередь, на ε.

Окончательно получаем:

$$x_2 = x_0+2\mathrm{dx}\pm\varepsilon(x_0+dx)\pm\varepsilon(x_0+2\mathrm{dx})$$

Выразим погрешность для количества шагов, равного трем:

$$x_3 = (((x_0+dx)(1\pm\varepsilon)+dx)(1\pm\varepsilon)+dx)(1\pm\varepsilon) =$$
$$= x_0+3\,dx\pm\varepsilon(x_0+dx)\pm\varepsilon(x_0+2\,dx\pm\varepsilon(x_0+dx))$$
$$\pm\varepsilon[x_0+3\,dx\pm\varepsilon(x_0+dx)\pm\varepsilon(x_0+2\,dx\pm\varepsilon(x_0+dx))].$$

В последних квадратных скобках уберем все слагаемые, содержащие ε, аналогично (7). В итоге получаем:

$$x_3 = x_0+3\,dx\pm\varepsilon(x_0+dx)\pm\varepsilon(x_0+2\,dx)\pm\varepsilon[x_0+3\,dx] .$$

Прослеживается явная закономерность:

$$x_n = x_0+n\cdot dx\pm\sum_{k=1}^{n}\varepsilon(x_0+k\cdot dx) \qquad (8)$$

Первые шаги индуктивного предположения выполняются, докажем верность равенства для n+1.

$$x_{n+1} = (x_0+n\cdot dx\pm\sum_{k=1}^{n}\varepsilon(x_0+k\cdot dx)+dx)(1\pm\varepsilon) =$$
$$= x_0+(n+1)\,dx\pm\sum_{k=1}^{n}\varepsilon(x_0+k\cdot dx)\pm\varepsilon(x_0+(n+1)\,dx\pm\sum_{k=1}^{n}\varepsilon(x_0+k\cdot dx)) =$$
$$= x_0+(n+1)\,dx\pm\sum_{k=1}^{n}\varepsilon(x_0+k\cdot dx)\pm\varepsilon(x_0+(n+1)\,dx)=x_0+(n+1)\,dx\pm\sum_{k=1}^{n+1}\varepsilon(x_0+k\cdot dx)$$

Что и требовалось доказать.

Теперь мы можем оценить погрешность суммирования, используя формулу (8), но сначала упростим ее, заметив, что если вынести постоянный множитель, под знаком суммы останется сумма членов арифметической прогрессии:

$$x_n = x_0+n\cdot dx\pm\varepsilon\cdot n\,x_0\pm\varepsilon\cdot dx\sum_{k=1}^{n}k \qquad (9)$$

Формула для вычисления суммы первых n членов арифметической прогрессии:

$$S_n = \frac{2\,a_1+d(n-1)}{2}\cdot n$$

Подставим в нее значения $a_1 = 1$, $d = 1$:

$$S_n = \frac{2+(n-1)}{2}\cdot n = \frac{n(n+1)}{2} .$$

Тогда формула (9) преобразуется в:

$$x_n = x_0+n\cdot dx\pm\varepsilon\cdot n\,x_0\pm\varepsilon\cdot dx\sum_{k=1}^{n}k=$$
$$= x_0+n\cdot dx\pm\varepsilon(n\,x_0+dx\cdot\frac{n(n+1)}{2}).$$

Теперь мы можем оценить ожидаемую абсолютную погрешность вычислений Δ:

$$\Delta_{(2)}(n) = \varepsilon(n\,x_0+dx\cdot\frac{n(n+1)}{2}) \qquad (10)$$

Запись $\Delta_{(2)}(n)$ подчеркивает, что на каждой итерации переменная n изменяется, увеличивая абсолютную ошибку, в то время как x_0 и dx при этом остаются фиксированными.

Посмотрим, согласуются ли выводы с реально полученными результатами.

Посчитаем теоретическую абсолютную погрешность при суммировании для 6-ти шагов (первый значительный выброс, связанный с выходом за правую границу на рисунке 2со стр.8).

Имеем следующие начальные данные:

$x_0 = 0$, n = 6, dx = $4\pi/6$ = 2,09439510239.
Наблюдаемая абсолютная погрешность для шести шагов равна 4,707901.

$\Delta_{(2)}(6) = 2^{-23}(6*0 + 2{,}09439510239*21) = 1{,}19209*10^{-7} * 43{,}9822971503 =$
$52{,}4308566099*10^{-7} = 5{,}243085661*10^{-6} \approx 0{,}0000052431.$

а) для $+\Delta_{(2)}$:
Выход за границу равен $\Delta_{(2)}$,

$x_6 = R + \Delta/2 = 12{,}5663706144 + 0{,}0000026215 = 12{,}5663732359;$
$f(x_6) = 1{,}725;$
$S = f(x_6)*\Delta_{(2)} = 1{,}725*0{,}0000052431 = 0{,}0000090443475,$
что меньше наблюдаемой ошибки в 230 тысяч раз

б) для $-\Delta_{(2)}$:
Выход за границу равен $dx-\Delta_{(2)}$,

$x_7 = R + (dx-\Delta_{(2)})/2 = 12{,}5663706144 + 2{,}09438985929/2 = 13{,}613565544;$
$f(x_6) = 2{,}1981507162;$
$S = f(x_6) *(dx-\Delta_{(2)}) = 2{,}1981507162 * 2{,}09438985929 = 4{,}6037845692,$
что очень похоже на наблюдаемую абсолютную погрешность 4,707901 (разность составляет 2%).

Видимо, при данном наборе данных происходит *накопление ошибки «в минус»*.

Проведем те же выкладки для точки, где ошибка не столь явная, например, в точке x = 100.

Согласно таблице 1 абсолютная погрешность для 100 шагов равна 0,000008.

При $x_0 = 0$, n = 100, dx = $4\pi/100$ = 0,125663706144

$\Delta_{(2)}(100) = 2^{-23}(100*0 + 0{,}125663706144*5050) = 2^{-23}*634{,}6017160272 =$
$1{,}192092895508*10^{-7} * 634{,}6017160272 = 7{,}565041971531*10^{-5} \approx 0{,}0000756$

Тогда излишне посчитанная площадь на последней итерации:

а) для $+\Delta_{(2)}$:
Выход за границу равен $\Delta_{(2)}$,
$x_{100} = R + \Delta_{(2)}/2 = 12{,}5663706144 + 0{,}00003782 = 12{,}5664084344$
$f(x_{100}) = 1{,}5*sqrt(2{,}9725-1{,}65*cos(12{,}5664084344)) = 1{,}7250000007695936$
$S = f(x_{100})*\Delta_{(2)} = 1{,}7250000007695936 *0{,}0000756 = 0{,}00013041$

Наблюдаемая абсолютная погрешность для 100 шагов равна 0,000008, что примерно в 16 раз меньше, чем ожидаемая абсолютная ошибка.

б) для -Δ :
Выход за границу равен dx-Δ ,
x_{101} = R + (dx-$\Delta_{(2)}$)/2 = 12,5663706144 + 0,125588106144/2 = 12,6291646675,
f(x_{101}) = 1,727119556319885
S = f(x_{101})*(dx-$\Delta_{(2)}$)= 1,727119556319885 * 0,125588106144 = 0,216905674162,

что больше наблюдаемой абсолютной ошибки в 27 тысяч раз.

Из этого можно сделать следующие выводы:

1) при суммировании по оси Ox идет погрешность в итоге накапливается «в плюс» и «в минус» примерно в равном количестве случаев (если смотреть диапазон вплоть до 100 шагов);

2) данная ошибка проявляется неравномерно на разных наборах данных и может быть частично компенсирована другими ошибками;

Абсолютная ошибка при вычислении dx вычисляется куда быстрее, по (5) имеем:

$$\Delta_{(1)} = L - L_{real} = n\,dx - n\,dx(1 \pm \varepsilon) \;=\; \varepsilon\, n\, dx \qquad (11)$$

Сравнивая формулы (10) для вычисления ошибки $\Delta_{(2)}$ и (11) для вычисления $\Delta_{(1)}$ можно сделать вывод, что вторая ошибка действительно поглощает первую, как и предполагалось, и, при этом, они, с большой вероятностью, не смогут компенсировать друг друга. Чтобы увидеть это, раскроем скобки в (10) и сравним с (11) .

$$\Delta_{(2)}(n) \;=\; \varepsilon\left(n\,x_0 + dx\cdot\frac{n(n+1)}{2}\right) \;=\; \varepsilon\, n\, dx\cdot\frac{n+1}{2} + \varepsilon\, n\, x_0 = \varepsilon\, n\, dx + \varepsilon\, n\, dx\left(\frac{n+1}{2} - 1\right) + \varepsilon\, n\, x_0 =$$

$$= \;\varepsilon\, n\, dx + \varepsilon\, n\, dx\frac{n-1}{2} + \varepsilon\, n\, x_0$$

Итого итоговая абсолютная ошибка $\Delta_{(3)}$ равна:

$$\Delta_{(3)} = |\pm\Delta_{(2)} \pm \Delta_{(1)}| = \varepsilon\, n\, dx\frac{n+1}{2} + \varepsilon\, n\, x_0 \pm \varepsilon\, n\, dx \;=\; \varepsilon\, n\, dx\left(\frac{n+1}{2} \pm 1\right) + \varepsilon\, n\, x_0 =$$

$$= \;\varepsilon\, n\, dx\frac{(n+1)\pm 2}{2} + \varepsilon\, n\, x_{0,} \qquad (12)$$

что не на много отличается от ошибки $\Delta_{(2)}$.

Посмотрим, от каких величин зависят эти ошибки.

Сначала можно заметить, что во всех формулах можно заменить зависимую величину dx на L/n. Тогда получаем:

$$\Delta_{(1)} = \;\varepsilon\, L$$

$$\Delta_{(2)} \;=\; \varepsilon\left(n\,x_0 + L\cdot\frac{n+1}{2}\right)$$

$$\Delta_{(3)} \;=\; \varepsilon\left(n\,x_0 + L\frac{(n+1)\pm 2}{2}\right)$$

Теперь мы можем увидеть некоторые зависимости.

Утверждение 1:

1) $\Delta_{(1)}$ зависит только от соотношения длины промежутка интегрирования и требуемого числа шагов и не зависит от количества итераций;

2) $\Delta_{(2)}$ зависит от n, x_0 и L и не зависит от $\Delta_{(1)}$.

3) $\Delta_{(3)}$ при больших n не зависит от $\Delta_{(1)}$, так как, используя теорию пределов, несложно показать, что $\Delta_{(3)} \to \Delta_{(2)}$ при $n \to \infty$;

Теперь зная, от чего зависят данные ошибки (и выяснив, что они не зависят друг от друга), мы можем создать такие тесты, которые будут выявлять только исследуемую погрешность.

Для того, чтобы погрешность вычисления функции в точке не мешала увидеть ошибку смещения аргумента по оси Ox, возьмем в качестве кривой прямую, параллельную оси абсцисс. Очевидно, длина такой кривой равна длине промежутка интегрирования, а задается такая прямая уравнениями:

$$x = t$$
$$y = h,$$

где h — константа. В нашем случае возьмем h = 5, выбор константы никак не повлияет на результат программы, так как ее производная всегда равна нулю.
Производные:

$$x_t' = 1$$
$$y_t' = 0$$

При таких условиях ожидается, что подынтегральная функция равна:

$$f(x) = \sqrt{1^2+0^2} = 1$$

и вычисляется абсолютно точно.

Проверим это. Используем тип данных double вместо float, чтобы иметь запас точности и убедиться в отсутствии погрешности:

Листинг 2: Вспомогательная подпрограма: пример подынтегральной функции, вычисляемой без погрешности

```
#define _USE_MATH_DEFINES
#define _GNU_SOURCE
#include <stdio.h>
#include <math.h>
#include <locale.h>

struct Curve;

float dx_straight(float fi, struct Curve* ptr)
{
    return 1.0;
}

float dy_straight(float fi, struct Curve* ptr)
{
```

```
    return 0.0;
}

struct Curve
{
    float (*dx)(float fi, struct Curve *);
    float (*dy)(float fi, struct Curve *);
} straight = {
    dx_straight,
    dy_straight
};



int main()
{
    setlocale( LC_ALL, "" );
    /* использовать , или . для отделения дробной части */
    printf("%G %s\n", f(0,0,0,5), f(0,0,0,5) == (double)1? "true" :
"false");
    return 0;
}
```

Программа возвращает 1 без каких-либо дополнительных знаков, как и ожидалось (тот же результат получим и в случае, если заменить тип на long double, надо будет лишь заменить при выводе флаг `%G` на `%LG`, иначе на консоль будет выведено число 6,95314E-310, что сначало было воспринято автором как ошибка, не влияющая, впрочем, на результат эксперимента, так как точночти типа double уже было достаточно).

Тогда длина кривой, согласно формуле (1) равна:

$$S = \int_a^b 1 \cdot \partial x = b - a = L,$$

то есть, самому промежутку интегрирования, и мы сможем увидеть, как влияет на вычисление итоговой суммы погрешность в вычисляемом промежутке интегрирования.

Постановка тестов.

Тест 1. Влияние ошибки суммирования на результат программы.

Согласно утверждению 1, ошибка суммирования независима от ошибки вычисления dx, поэтому достаточно взять такие данные, чтобы смещение dx было представлено абсолютно точно, (в виде конечной двоичной дроби с мантиссой не длиннее 24 разрядов) и был ненулевым (этот случай будет рассмотрен позднее), а количество шагов было максимально возможным. Чтобы исключить ошибку вычисления подынтегральной функции, используем функцию из Листинга 2, а так же сделаем вывод итоговых значений интересующих нас переменных.

Чтобы погрешность была заметной, пусть переменная x имеет постоянный порядок и частично перекрывается с ненулевой частью мантиссы переменной dx так, чтобы при суммировании происходило округление всегда в одну сторону. Так как присутствует ошибка выхода за правую границу, пусть при суммировании происходило округление в меньшуу

сторону так, чтобы покрытие промежутка интегрирования происходило за большее число итераций. Тогда при выбранных данных ожидается, что будет происходить накопление погрешности в «плюс».

Сконструируем функцию, которая будет «собирать» нужное нам число из значений мантиссы, экспаненты и знакового бита, чтобы можно было легко видеть, что представляет из себя число в двоичном представлении.

Листинг 3: Вспомогательная программа, возвращающая число типа float с заданными значениями экспаненты и мантиссы.

```
union Num{
    struct {
        unsigned int mantissa : 23;
        unsigned int exp : 8;
        unsigned int sign : 1;
    } parts;
    float f;
};

float make_float(int sign, int exp, int mant)
{
    union Num new_float;
    new_float.parts.exp = exp + 127; //+BIAS
    new_float.parts.sign = sign;
    new_float.parts.mantissa = mant;
    return new_float.f;
}
```

Пусть, например, $x_0 = 1*2^{-1}$ и dx = make_float(0, -24, 0b01010101010101010101010) /*первый бит мантиссы опущен*/, то есть их мантиссы перекрываются так:

```
10000000000000000000000
+                       10101010101010101010101,
```

и после отбрасывания лишних разрядов их сумма будет округлена до

```
10000000000000000000001.
```

Чтобы от теста к тесту переменная dx изменялась простым смещением запятой в двоичном представлении, в качестве числа шагов будем брать степени двойки, а для того, чтобы округление происходило всегда в одном направлении, пусть смещение происходит на два разряда.

Вычислим, как приблизительно поведет себя программа с учетом того, что погрешность накапливается всегда в одну сторону и имеет место выход за правую границу (см. разбор формулы (10) на стр. 10 выше).

$$dx * n \le L_{real} \le dx * n + dx$$
$$dx * n \approx dx_{real} * n_{real}$$
$$\int_a^b f(x) \approx dx * n_{real} + dx = dx * n \frac{dx}{dx_{real}} + dx$$

То есть ожидается, что результат будет больше требуемого на величину, пропорциональную отношению ожидаемого и реального dx, а погрешность выхода за правую

границу тем больше, чем больше dx.

Листинг 4: Тест 1. Влияние ошибки суммирования на результат программы.

```
#define POW2(n) (1 << (n))          //Быстрое вычисление степени двойки

type integrate(type left, type right, unsigned long steps, type (*func)
(type, void *), void *params)
{
    type res = 0;
    type x = left;

    type last_x = x;
    type dx = (right-left)/steps;
    type dx_now, dx_last;   //Реальный шаг
    int dx_const = 1;       //Переменная, равная 1 после завершения, если
на каждой итераци реальное смещение было одинаковое
    unsigned long counter = 0;
    dx_now = dx_last = (x + dx) - x;

    for(x = left; x < right && counter < 10000000000; /*x += dx*/){
        res += func(x + dx/2, params)*dx;
        last_x = x;
        x += dx;
        dx_now = x-last_x;
        dx_const &= (dx_now == dx_last);
        dx_last = dx_now;
        last_x = x;
        ++counter;
    }
    printf("%lu; %lu; %G; %G; %s; %G",
           steps, counter, dx, dx_last, dx_const ? "true" : "0",
           res, dx*steps*dx/dx_last+dx);
    return res;
}

int main()
{
    ...
    printf("Ожидаемое число шагов; Реальное число шагов; dx ожидаемое; "
           "dx реальное; Разность постоянна?; "
           "Полученный результат; Ожидаемый результат с учетом погрешности;
Требуемый результат\n");
    unsigned long n;
    float left = 0.5, right;
    int d_radix = 2;     //На сколько разрядов в двоичном представлении
смещается мантисса переменной dx
    right = make_float(0, -1, 0b10101010101010101010101);
    for(d_radix = 2; d_radix < 23; d_radix += 2){
        n = POW2(d_radix);
        integrate(left, right, n, f, &straight);
        printf("; %G\n", right-left);
    }
    return 0;
}
```

Листинг 5: Результат запуска теста 1.

Ожидаемое число шагов	Реальное число шагов	dx ожидаемое	dx реальное	Разность постоянна?	Полученный результат	Ожидаемый результат с учетом погрешности	Требуемый результат
4	5	0,0833333	0,0833333	true	0,416667	0,416667	0,333333
16	17	0,0208333	0,0208333	true	0,354167	0,354167	0,333333
64	65	0,00520833	0,00520831	true	0,338542	0,338543	0,333333
256	257	0,00130208	0,00130206	true	0,334636	0,33464	0,333333
1024	1025	0,000325521	0,000325501	true	0,33366	0,333679	0,333333
4096	4097	8,13802E-05	8,13603E-05	true	0,333419	0,333496	0,333333
16384	16401	2,03451E-05	2,03252E-05	true	0,333696	0,333679	0,333333
65536	65793	5,08626E-06	5,06639E-06	true	0,334708	0,334646	0,333333
262144	266305	1,27157E-06	1,2517E-06	true	0,338921	0,338626	0,333333
1048576	1118481	3,17891E-07	2,98023E-07	true	0,357206	0,355556	0,333333
4194304	5592405	7,94729E-08	5,96046E-08	true	0,461056	0,444444	0,333333

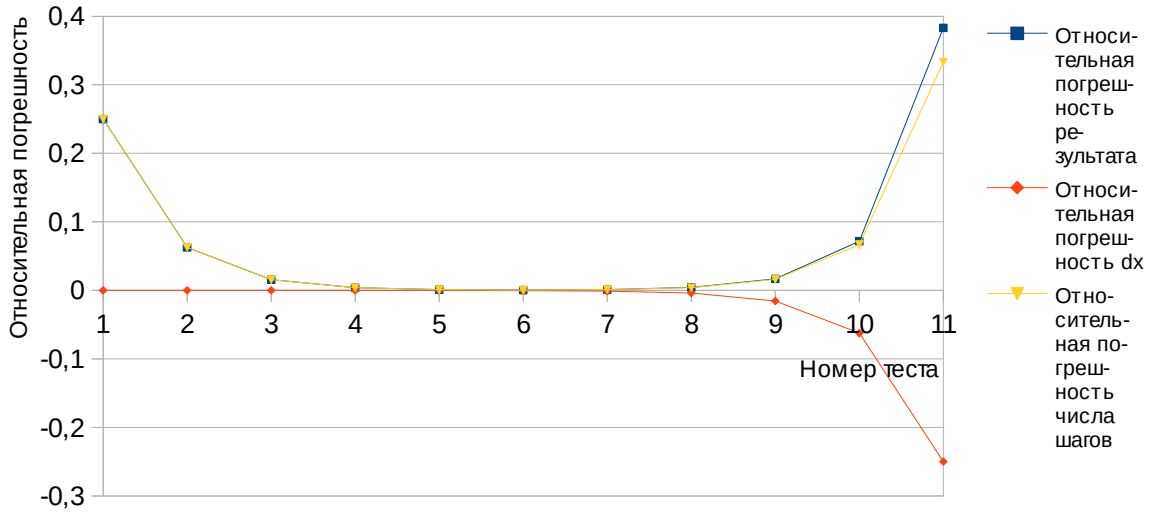

Рисунок 3: Результат запуска теста 1

Как мы видим, действительно, округление dx всегда происходило в меньшую сторону (это становится особено заметно при большом количестве итераций), с этим увеличивалось относительная ошибка числа шагов, и, соответственно, погрешность результата. При этом из-за того, что погрешность прибавления лишней суммы при выходе за правую границу участка интегрирования присутствовала, и она прямо пропорциональна dx (обратно пропорциаональна n), то она сильнее увеличивает погрешность при малом количестве итераций, то есть выброс на первых шагах можно объяснять ей. Попробуем убрать эту ошибку, чтобы видеть только погрешность, получаемую ошибкой суммирования внутри интервала интегрирования.

Изменим условие выхода из цикла с «x < right» (1) на «x + dx < right» (2), а на оставшемся участке посчитаем значение функции за пределами цикла.

Листинг 6: Изменения в функции интегрирования, блокирующие выход за правую границу

```
float integrate(float left, float right, unsigned long steps, float (*func)
(float, void *), void *params)
{
    ...
    for(x = left; x+dx < right; x += dx){ //Новое условие выхода из цикла
(2)
        …
    }
    dx_now = right-x; //Последний интервал считается отдельно
    res += func(x+dx_now/2, params)*dx_now;
    ++counter;
    printf("%lu; %lu; %G; %G; %s; %G; %G", …  dx*steps*dx/dx_last /*+dx*/);
}
```

Листинг 7: Результат запуска программы с блокировкой выхода за правую границу.

Ожидаемое число шагов	Реальное число шагов	dx ожидаемое	dx реальное	Разность постоянна?	Полученный результат	Ожидаемый результат с учетом погрешности	Требуемый результат
4	5	0,0833333	0,0833333	true	0,333333	0,333333	0,333333
16	17	0,0208333	0,0208333	true	0,333334	0,333334	0,333333
64	65	0,00520833	0,00520831	true	0,333335	0,333335	0,333333
256	257	0,00130208	0,00130206	true	0,333339	0,333338	0,333333
1024	1025	0,000325521	0,000325501	true	0,333355	0,333354	0,333333
4096	4097	8,13802E-05	8,13603E-05	true	0,333419	0,333415	0,333333
16384	16401	2,03451E-05	2,03252E-05	true	0,333676	0,333659	0,333333
65536	65793	5,08626E-06	5,06639E-06	true	0,334708	0,33464	0,333333
262144	266305	1,27157E-06	1,2517E-06	true	0,338921	0,338624	0,333333
1048576	1118481	3,17891E-07	2,98023E-07	true	0,357206	0,355556	0,333333
4194304	5592405	7,94729E-08	5,96046E-08	true	0,461056	0,444444	0,333333

При сравнении с листингом 5 сразу становится заметно, что при большом числе шагов вклад ошибки выхода за правую границу был небольшой, и результат практически не изменился. Зато проявилась тенденция роста относительной ошибки с увеличением отношения dx/dx_{real}.

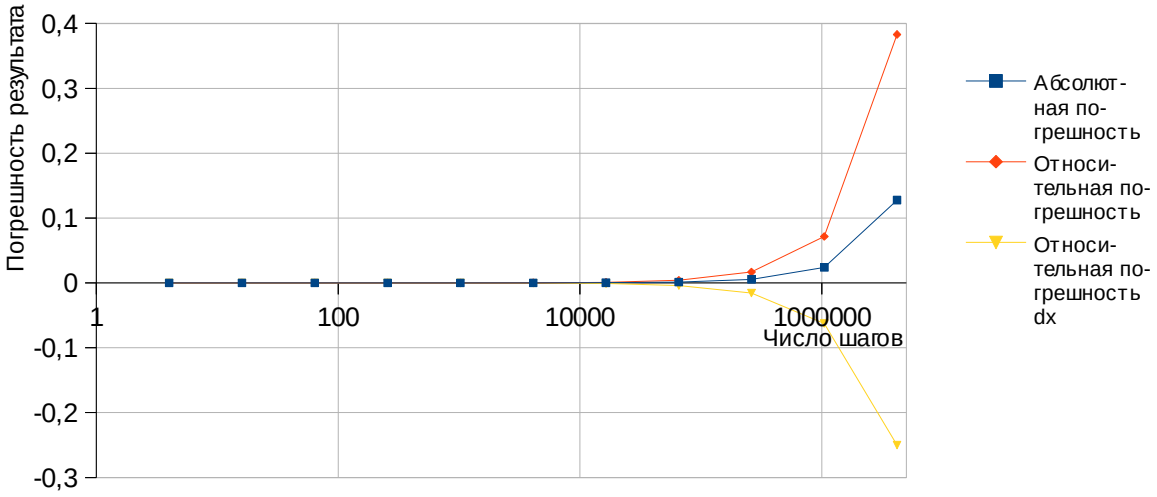

Рисунок 4: Результат запуска программы без выхода за правую границу

Теперь погрешность результата действительно стала меньше на первых шагах и растет с увеличением числа итераций.

Так как последнее изменение в программе (Листинг 6) привело к увеличению точности, оставим его.

Тест 2. Неправильное вычисление dx, приводящее к зацикливанию.

Согласно утверждению 1, ошибка вычисления dx зависит только от соотношения L и steps. Поэтому, чтобы максимально уменьшить влияние ошибки при сумировании x+dx, сделаем количество итераций равным одному, а левую границу нулевой, так как сложение с нулем происходит абсолютно точно. (Но при таких условиях будет сложно добиться того, чтобы x был вычислен неточно. В таком случае ошибка может проявиться, например, при сужении типов при передаче аргументов в функцию. Но этот тест не будет для нас показателен, так как в таком случае эта ошибка является внешней и не может быть устранена в теле функции.)

Аналогично, если предположить, что найдутся такие данные, что dx будет округлен до нуля, то погрешности суммирования также не будет. Однако если dx + x == x, то это значит, что x никогда не достигнет правой границы и программа зависнет, так как других условий выхода из цикла не предусмотрено. Приведем такой пример.

Возьмем такие L и n, что их частное будет столь малым (например, число 10^{-46}), что оно выходит за рамки представления типа float и считается машиной равным нулю. Поэтому возьмем такие исходые данные:

left = 0,

right = 1e-45,

steps = 10

и поставим вывод проверок на то, какие из данных величин считаются нулевыми. Так как ожидается, что программа зациклится, поставим также счетчик числа итераций, много больший, чем желаемое количество шагов.

Листинг 8: Пример входных данных, проводящих к равенству нулю переменной dx

```
float integrate(float left, float right, unsigned long steps, float (*func)
(float, void *), void *params)
{
    ...
    unsigned long counter = 0;        //Счетчик
    for(x = left; x < right && counter < 1000000; x += dx){
        res += func(x + dx/2, params)*dx;
        ++counter;         //Счетчик увеличивается на единицу после каждой
итерации
    }
    printf("dx = %G\ndx == 0: %s\nleft = %G\nright = %G\nright > left:
%s\nx = %G\nx == left: %s\ncounter = %lu\n",
           dx,
           dx == 0 ? "true" : "false",
           left, right,
           !(right - left == 0) ? "true" : "false",
           x,
           x == left ? "true" : "false",
           counter);
    return res;
}

int main()
{
    long n = 10;
    float L = 0, R = 1e-45;  //Малый интервал
    ...
    integrate(L, R, n, &straight);
    return 0;
}
```

Программа возвращает такой результат:

Листинг 9: Результат запуска программы (Листинг 1), приводящей к зацикливанию

```
dx = 0
dx == 0: true
```

```
left = 0
right = 1,4013E-45
right > left: true
x = 0
x == left: true
counter = 1000000
```

Программа повела себя, как и ожидалось: несмотря на то, что разность правой и левой границы была ненулевой, при делении на десять частей этот промежуток стал восприниматься машиной как нулевой, и выход из цикла не происходил вплоть до тех пор, пока счетчик не достиг установленного значения.

Этот пример показывает, во-первых, как незначительная, на первый взгляд, ошибка может оказаться фатальной, а во-вторых наводит на мысль о том, что могут существовать и другие ошибки, приводящие к зацикливанию. Такая ошибка может появиться в 2-х случаях:

1) при округлении dx до нуля (пример, рассмотренный выше);
2) $x_i >> dx$ и их мантиссы не пересекаются;

Зацикливание при непересекающихся мантиссах.

Так как ε — наименьшее число, обладающее свойством (1+ε) != 1, то, взяв δ = ½ε, мы получим число такое, что (1+δ) == 1.

В программе этим числом будет delta = FLT_EPSILON/2.

И если dx < max(|a|, |b|), то в какой-то момент в промежутке [a ; b] произойдет зацикливание.

Например, зацикливание происходит при таких данных:

left = 0;

right = 1;

$n = 10^8$.

Теперь можно сказать, что были рассмотрены все ошибки, заявленные в начале Пункта 2.