

7. Обчислення з заданою точністю

На практиці не менше значення ніж методи сортування даних мають числові методи розв'язування математичних задач. Сама назва «комп'ютер» означає *обчислювач*, а ще кілька років назад широко використовувався термін ЕОМ – електронно-обчислювальна машина. Такий наголос на здатності виконувати обчислення не випадковий: комп'ютер і було винайдено саме для автоматизації обчислень, а перші комп'ютери використовували виключно для розв'язування складних математичних задач, що вимагало виконання величезної кількості обчислень. Винайдення комп'ютера зумовило бурхливий розвиток цілої галузі математичної науки – обчислювальної математики, яка вивчає і розробляє спеціальні (числові) методи відшукування наближених розв'язків різноманітних задач: сумування рядів, числового інтегрування та диференціювання, обчислення значень коренів, розв'язків систем рівнянь тощо. Ми не маємо змоги в межах цього посібника докладно викласти теорію методів обчислень (ми і не ставили перед собою такої мети), проте познайомити читача з алгоритмами хоча б кількох з них – просто зобов'язані. Кожен освічений програміст знає, що означає для його програми обчислення синуса, кореня чи іншої стандартної функції. До того ж числові методи дають прекрасну нагоду продемонструвати декілька важливих прийомів програмування.

7.1. Сумування рядів

Практично всі компілятори мов програмування високого рівня надають програмістові достатній перелік можливостей для обчислення математичних функцій, проте всі процесори використовують спеціальні засоби, щоб обчислити, наприклад, експоненту чи синус кута. У давніших моделях комп'ютерів кожне звертання до математичної функції компілятор замінював на виклик відповідної послідовності команд (процедури), яка обчислювала значення цієї функції певним наближеним методом. Сучасні комп'ютери обладнують процесорами з вбудованими математичними співпроцесорами, в яких такі процедури запрограмовані «в камені». Розглянемо один зі способів наближеного обчислення функції $\sin(x)$ – за допомогою розкладу в ряд Тейлора. Відомо, що він має вигляд:

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

і для невеликих за модулем значень x значення синуса можна обчислювати як суму цього ряду.

Задача 30. *Задано дійсне значення x ($|x| \leq \pi/4$). Обчислити $\sin(x)$ з точністю 10^{-6} , використовуючи розклад синуса до ряду Тейлора.*

Одразу виникає декілька запитань: як обчислювати члени цього ряду; коли закінчувати обчислення; що гарантує досягнення заданої точності?

Ряд Тейлора функції синус є знакозмінним, а для значень x з вказаного діапазону послідовність абсолютних величин членів цього ряду є монотонно спадною і збіжною до нуля. Тому, як відомо з курсу математичного аналізу, за ознакою Лейбніца цей ряд збігається, причому модуль його залишку не перевищує першого відкинутого члена. Це означає, що для обчислень умовою досягнення заданої точності є виконання нерівності $|d_n| \leq 10^{-6}$, де d_n – черговий член ряду. Як тільки на якійсь ітерації виконається ця умова, обчислення можна припиняти.

Кожен член ряду містить відповідні факторіал і степінь x . Зрозуміло, що обчислювати їх необхідно за допомогою множення, тим більше, що мова Pascal не має засобів піднесення до степеня. Чи можна використовувати дві різні змінні з метою накопичення степеня x і факторіала? Наприклад, так:

```

d = x; s = 0;           // перший член ряду і початкове значення суми
p = x; f = 1;          // початкові значення степеня x і факторіала
n = 1;                 // номер члена ряду
while (abs(d) > 1e-6)
{
    // У знакозмінного ряду
    if (n % 2 == 1) s += d; // непарні члени додаємо,
    else s -= d;           // а парні - віднімаємо.
    ++n;                   // Починаємо рахувати наступний член ряду:
    p *= pow(x, 2);        // степінь зріс на x^2,
    f *= (2 * n - 2) * (2 * n - 1); // факторіал - на два множники,
    d = p / f;             // маємо член ряду.
}

```

У цьому фрагменті і логічно, і синтаксично все написано добре, а все ж так виконувати обчислення суми зазначеного ряду не варто. Вся справа в тому, що черговий член ряду є часткою двох потенційно великих чисел: для великих за модулем x його степінь зростає дуже швидко (добре, якщо його величина обмежена, як в нашій задачі, а якщо ні!), а факторіал, взагалі, є найшвидше зростаючою математичною функцією. Що ж станеться, якщо «дуже велике» поділити на «дуже велике»? Сучасні комп'ютери відображають дійсні числа і виконують обчислення зі скінченною точністю. Наприклад, значення типу *double* у мові C++ містять близько 15-ти точних десяткових знаків мантиси. Припустимо, що у виразі $d = p/f$ чисельник є величиною порядку 10^{16} («дуже велике»), а знаменник – порядку 10^{20} (ще більше), і кожен з них містить до чотирнадцяти точних знаків – точність втрачається в результаті виконання арифметичних дій. Тоді порядок точних цифр мантис чисельника і знаменника є від 10^6 до 10^{20} . За правилами виконання обчислень з плаваючою крапкою точні знаки результату теж будуть у цьому діапазоні, але d є величиною порядку 10^{-4} і тому не містить ні одного (!) точного знака.

З метою ефективного обчислення суми заданого ряду доцільно використати інший підхід і вивести рекурентну формулу для обчислення членів ряду. Легко бачити, що $d_n = (-1)^{n+1} x^{2n-1} / (2n-1)!$, а $d_{n+1} = (-1)^n x^{2n+1} / (2n+1)!$. Тоді $d_{n+1}/d_n = -x^2 / (2n(2n+1))$, і шукані рекурентні формули мають вигляд $d_1 = x$, $d_{n+1} = -x^2 / (2n(2n+1)) d_n$, $n=1,2,\dots$. Зауважимо також, що величину $-x^2$ можна обчислити ще перед циклом, а значення n краще збільшувати не на 1, а на 2: не потрібно буде виконувати множення на 2 в знаменнику рекурентної формули:

```

void sinSeriesSum()
{
    cout << "\n *Обчислення суми ряду Тейлора - розкладу sin(x)*\n\n"
         << "Введіть дійсне число x = ";
    double x; cin >> x;
    // Приготування до обчислень
    double y = -x*x; // часто вживаний множник
    double d = x;    // перший член ряду
    double s = x;    // сума ряду
    int n = 2;       // черговий множник для факторіала
    while (abs(d) > 1e-6)
    {
        // черговий член ряду обчислюємо за рекурентною формулою
        d *= y / (n*(n + 1));
        s += d; // враховуємо його в сумі ряду
        n += 2;
    }
    cout << "S = " << s << "    sin(x) = " << sin(x) << '\n';
    return;
}

```

Пропонуємо читачеві самостійно простежити, як змінюватимуться значення d і s на кількох перших кроках циклу.

Продемонструємо використання описаного підходу на ще одному прикладі.

Задача 31. *Задано дійсні числа x , ε ($x \neq 0, \varepsilon > 0$). Обчислити з точністю ε значення*

$$s = \sum_{k=0}^{\infty} \frac{(-1)^k x^{4k+3}}{(2k+1)!(4k+3)}.$$

Ряд в умові задачі є знакозмінним, тому для перевірки досягнення заданої точності, як і раніше, використаємо умову $|d_n| \leq \varepsilon$. Для обчислення членів ряду виведемо рекурентні

формули: $d_0 = x^3/3$, $d_k = d_{k-1} \cdot \frac{-x^4}{2k(2k+1)} \cdot \frac{4k-1}{4k+3}$, $k = 1, 2, \dots$. Видно, що остання формула містить

«зайвий» громіздкий співмножник $(4k-1)/(4k+3)$. Звідки він взявся? Кожен з членів ряду містить у знаменнику множник вигляду $(4k+3)$. У різних d_k ці множники різні, тому їх не доцільно зачислювати до рекурентної формули. Залишимо в ній тільки множники «відповідальні» за обчислення степеня і факторіала, а ділення на $(4k+3)$ виконуватимемо перед додаванням d_k до суми:

```
void sumOfSeriesB()
{
    cout << "\n *Обчислення суми ряду S( -1^k x^4k+3 / (2k+1)!(4k+3) )*\n\n";
    << "Введіть дійсне число x = ";
    double x; cin >> x;
    cout << "Задайте точність обчислень eps = ";
    double eps; cin >> eps;
    // Приготування до обчислень
    double y = x*x;
    double d = x*y; // перший член ряду без дільника (4k+3)
    double s = d/3; // сума ряду
    y = -y*y; // часто вживаний множник
    int k = 0; // номер врахованого доданка
    while (abs(d) > eps)
    {
        // черговий член ряду обчислюємо за рекурентною формулою
        ++k;
        d *= y / (2 * k*(2 * k + 1));
        s += d / (4 * k + 3); // враховуємо його в сумі ряду
    }
    cout << "S(" << x << ") = " << s << '\n';
    return;
}
```

7.2. Обчислення кореня алгебричного рівняння

Одним з найпростіших методів обчислення кореня рівняння $f(x)=0$ на проміжку $[a; b]$ є метод поділу відрізка навпіл. Його можна використовувати тоді, коли відомо, що $f(x)$ – неперервна функція, яка один раз змінює свій знак на заданому проміжку. Алгоритм методу дуже простий: обчислюють середину проміжку і значення функції на ній, визначають, на лівій чи правій половині відрізка функція змінює свій знак, і для цієї половини повторюють описані дії. Обчислення припиняють тоді, коли довжина чергового проміжку стає меншою за задану величину точності. За наближене значення кореня рівняння приймають будь-яке

значення з останнього проміжку, наприклад, його середину. На рис. 6 зображено декілька перших кроків методу для рівняння $f(x) = 0$.

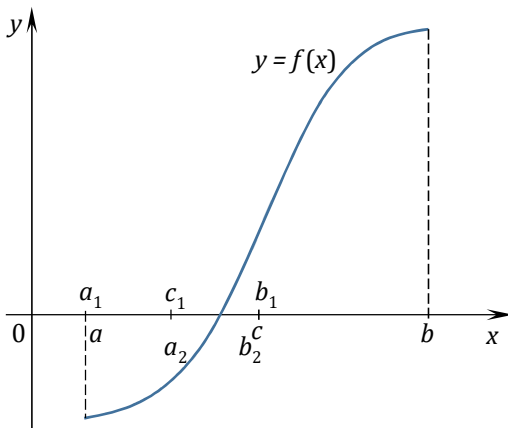


Рис. 6. Метод дихотомії

Задача 32. Обчислити методом поділу відрізка навіпів корені рівняння $x^2 + 4x - 32 = 0$ на проміжках $[-8, 4; -7, 7]$ та $[3, 7; 4, 2]$ з точністю 10^{-4} і корінь рівняння $e^{2x} = 3$ на проміжку $[0, 5; 1]$ з точністю 10^{-8} .

Метод реалізуємо у вигляді функції. Її вхідними параметрами будуть: a, b – межі відрізка; ϵ – точність обчислень; f – функція, що описує ліву частину заданого рівняння (такий параметр повинен мати тип «вказівник на функцію, що приймає один параметр дійсного типу і повертає значення дійсного типу»). Використаємо також локальні змінні: c – середина відрізка $[a; b]$; fa, fc – значення функції $f(x)$ у точках a, c відповідно (їх необхідно

зберігати, щоб не виконувати повторних обчислень $f(x)$). Щоб перевірити зміну знаку функції, визначимо знак добутку $fa*fc$: добуток множників протилежних знаків – від’ємний.

```
typedef double(*func)(double);
double dychotom(func f, double a, double b, double eps = 1e-8)
// Ітеративна реалізація методу дихотомії
{
    double fa = f(a);
    while (b - a > eps)
    {
        double c = (a + b)*0.5; // середина проміжку
        double fc = f(c);
        if (fa*fc < 0) b = c;    // корінь - у лівій половині проміжку
        else                  // корінь - праворуч
        {
            a = c; fa = fc;
        }
    }
    return (a + b)*0.5;
}
```

Тут для зміни меж проміжку відшукування кореня виконуємо присвоєння $b = c$ – посуваємо праву межу ліворуч, або $a = b$ – посуваємо ліву межу праворуч. Наступне повторення ітераційного циклу виконає такі ж дії для оновлених меж – як сказано в описі алгоритму на початку параграфу. Проте ми могли б реалізувати цей алгоритм інакше: фразу «і для цієї половини повторюють описані дії» можна перекласти мовою C++ за допомогою рекурсивного виклику функції обчислення кореня.

```
double dychotomRec(func f, double a, double b, double eps = 1e-8)
{
    // Рекурсивна реалізація методу дихотомії
    double c = (a + b)*0.5;
    if (b - a <= eps)          // проміжок достатньо малий
        return c;
    if (f(a)*f(c) < 0)         // шукати корінь на лівій половині
        return dychotomRec(f, a, c, eps);
    else                       // шукати корінь на правій половині
        return dychotomRec(f, c, b, eps);
}
```

Тепер, щоб знайти корінь деякого рівняння за допомогою однієї з описаних функцій, достатньо задати це рівняння функцією аргумента дійсного типу і передати її разом з іншими параметрами у виклику *dychotom()* чи *dychotomRec()*. Для розв'язування задачі 35 можна записати

```
// ліві частини рівнянь
double parabola(double x)
{
    return (x + 4.)*x - 32.;
}
double exponenta(double x)
{
    return exp(2.*x) - 3.;
}

// Фрагмент головної програми
cout << "\n *Обчислення кореня рівняння методом дихотомії*\n\n";
double x1 = dychotom(parabola, -8.4, -7.7, 1e-4);
cout << " Ітеративно: x1 = " << x1 << '\n';
x1 = dychotomRec(parabola, -8.4, -7.7, 1e-4);
cout << " Рекурсивно: x1 = " << x1 << '\n';
```

Функція *dychotom* відрізняється від усіх, які ми писали раніше: серед її параметрів є один функціональний, а відповідним аргументом є не число, а інша функція! Функції, параметризовані функціями, називають *функціями вищих порядків*. У нашому прикладі такими є *dychotom* і *dychotomRec*. Їхніми аргументами могли б стати і стандартні функції. Наприклад, виклик *dychotom(cos, -0.2, 0.2)* знайде корінь рівняння $\cos x = 0$.

Описаний підхід розв'язує поставлену задачу, але має один недолік: занадто багато оголошень функцій. Такі «дрібні» функції, як *parabola* та *exponenta* і місце в тексті програми займають, і простір імен засмічують. Новий стандарт мови C++ надає кращу можливість – лямбда-вирази. Вони прийшли у C++ з мов функціонального програмування і дають змогу задавати аргумент функції вищого порядку за допомогою анонімною функції. Схематично оголошення лямбда-виразу можна зобразити так:

[] (формальні параметри) -> тип_результату { тіло функції }

Тип результату можна не вказувати, якщо в тілі функції є одна інструкція *return*, і компілятор сам може вивести його.

Остаточно програму обчислення коренів алгебричних рівнянь запишемо у вигляді:

```
void dichotomy()
{
    cout << "\n *Обчислення кореня рівняння методом дихотомії*\n\n";
    func par = [](double x) -> double { return (x + 4.)*x - 32.; };
    double x1 = dychotom(par, -8.4, -7.7, 1e-4);
    double x2 = dychotom(par, 3.7, 4.2, 1e-4);
    double x3 = dychotom([](double x){ return exp(2.*x) - 3.; }, 0.5, 1.);
    cout<<" Ітеративно:\nкорені параболи    x1 = "<< x1 <<"    x2 = "<< x2 <<'\n'
        << "корінь експоненти    x = " << x3 << '\n';
    x1 = dychotomRec(par, -8.4, -7.7, 1e-4);
    x2 = dychotomRec(par, 3.7, 4.2, 1e-4);
    x3 = dychotomRec([](double x){ return exp(2.*x) - 3.; }, 0.5, 1.);
    cout<<"\n Рекурсивно:\nкорені параболи    x1 = "<< x1 <<"    x2 = "<< x2 <<'\n'
        << "корінь експоненти    x = " << x3 << '\n';
    return;
}
```

У результаті виконання цієї програми отримано такі результати:

Обчислення корення рівняння методом дихотомії

Ітеративно:

корені параболи $x_1 = -7.99997$ $x_2 = 4.00002$
 корінь експоненти $x = 0.549306$

Рекурсивно:

корені параболи $x_1 = -7.99997$ $x_2 = 4.00002$
 корінь експоненти $x = 0.549306$

Очевидно, що обчислені наближені корені збігаються з точними у межах заданої точності.

7.3. Числове інтегрування

Існує багато аналітичних методів обчислення визначених інтегралів, складено багато довідкових таблиць зі значеннями часто вживаних інтегралів, однак на практиці завжди

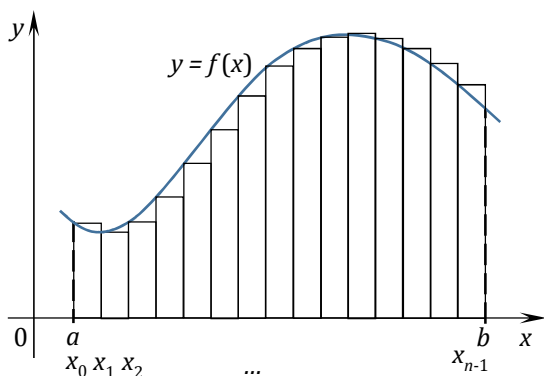


Рис. 7. Метод лівих прямокутників

зустрічаються такі, яких нема в таблицях, і для яких аналітичні методи не працюють. Тоді на допомогу приходять методи числового інтегрування, які дають змогу обчислити наближене значення визначеного інтеграла. Ми розглянемо один з найпростіших методів числового інтегрування – *метод лівих прямокутників*. За

цим методом $\int_a^b f(x)dx \approx h \sum_{i=0}^{n-1} f(x_i) = I_n$, де $h = \frac{b-a}{n}$,

$x_i = a + ih$. З метою досягнення заданої точності задають деяке значення n , обчислюють і порівнюють I_n та I_{2n} . За правилом Рунге точності ϵ досягнуто, якщо виконується $|I_n - I_{2n}|/3 \leq \epsilon$.

Якщо ж ні, то обчислюють I_{4n} і порівнюють його з I_{2n} і т. д.

Обчислення такої суми, як у формулі лівих прямокутників, легко запрограмувати з використанням одного простого циклу (використаємо тип *func* з попередньої програми):

```
double leftR(double a, double b, func f, int n)
{
    double h = (b - a) / n;
    double s = 0.;
    for (int i = 0; i < n; ++i) s += f(a + i*h);
    return s*h;
}
```

Здавалося б, тепер достатньо викликати цю функцію потрібну кількість разів, щоб виконати обчислення з заданою точністю:

```
// Фрагмент програми інтегрування
int n = 10; // задали початкове значення n
double I1 = leftR(a, b, f, n); // обчислили значення I(n)
double I2 = leftR(a, b, f, n * 2); // та I(2n)
while (abs(I1-I2)/3. > 1e-6 )
{
```

```

    I1 = I2; // запам'ятали старе значення,
    n *= 2; // подвоїли кількість доданків
    I2 = leftRect(a, b, f, n * 2); // порахували нове значення
}
cout << "Integral = " << I2 << '\n';

```

Справді, тут нема ні синтаксичних, ні логічних помилок, однак такий фрагмент програми дуже нераціональний. Він примушує функцію *leftR* виконувати багато зайвих обчислень. Яких? Функція *leftR* щоразу обчислює значення $f(x_i)$. Нехай $a=0$, $b=1$, тоді під час першого виклику *leftR* буде обчислено значення f у точках 0, 0.1, 0.2, ..., 0.9, а під час другого – в точках 0, 0.05, 0.1, 0.15, ..., 0.9, 0.95. Бачимо, що друга множина точок містить усі точки першої, і саме в них обчислення $f(x_i)$ будуть виконані удруге. Така ж ситуація буде повторюватися під час кожного наступного виклику *leftR*: якщо кількість точок x_i щоразу подвоювати (як ми це робимо), то тільки половина з них буде новою, і тільки для них потрібно обчислювати $f(x_i)$, а для «старої» половини можна використати значення, обчислені на попередній ітерації. У наведеному ж вище фрагменті функція *leftR* для кожного нового значення n обчислює всі значення $f(x_i)$.

Щоб уникнути зайвих обчислень, удосконалимо функцію *leftR* і «заховаємо» перевірку точності в її тілі. Обмежимо також кількість подвоєнь n : якщо цього не зробити, існуватиме великий ризик зациклення програми для тих підінтегральних функцій, для яких важко досягти високої точності обчислення інтеграла за допомогою методу лівих прямокутників:

```

double leftRect(double a, double b, func f, double eps)
// інтеграл від f на проміжку [a;b] методом лівих прямокутників
// p - значення I(2n); q - значення I(n); s - використовується
// для постійного накопичення суми значень f(i)
// дозволено не більше 12 подвоєнь сітки інтегрування
{
    int n = 10;
    double h = (b - a) / n;
    double s = 0.;
    for (int i = 0; i < n; ++i) s += f(a + i*h);
    double p = s*h; // перше значення інтеграла
    double q = 0.; // наступне значення
    int k = 0; // кількість подвоєнь вузлів інтегрування
    while (abs(p - q) / 3. > eps && k < 13)
    {
        ++k; q = p; // запам'ятали попереднє I(n)
        double h2 = h * 0.5;
        double a2 = a + h2;
        for (int i = 0; i < n; ++i)
            s += f(a2 + i*h); // доповнили суму
        p = s * h2; // отримали нове значення I(2n)
        h = h2; n += n; // подвоїли кількість доданків
    }
    if (k > 12) cout << "leftRec::WARNING: Accuracy loss is possible\n";
    return p;
}

```

Таку підпрограму вже можна використовувати на практиці. Покажемо, як з її допомогою обчислюють означені інтеграли: одновимірні, одновимірні залежні від параметра, а також подвійні.

Задача 33. Обчислити методом лівих прямокутників $\int_0^{\pi} \sin x dx$ з точністю 10^{-6} , серію

інтегралів $S(t) = \int_{-1}^1 (t^2 x^2 + 1) dx$ для $t = 0(0,5)3$ з точністю 10^{-5} і значення

$$R = \int_0^1 \int_0^x (x + 2y)^2 dy dx \text{ з точністю } 10^{-4}.$$

З першим інтегралом не виникне жодних труднощів: усього один виклик `leftRect(0., M_PI, sin)`, і все.

Підінтегральну функцію у `leftRect` передають за допомогою параметра `func f`, тобто, за допомогою функції з одним аргументом. Як обчислити інтеграл $S(t)$, коли підінтегральна функція залежить від числового параметра t ? Його не можна включити до формальних параметрів, тому використаємо для цього глобальну змінну (у програмі це змінна `double t`).

Як пристосувати `leftRect` для обчислення подвійних інтегралів? Можна використати зведення подвійного інтеграла до повторного. Тоді зовнішній інтеграл (по аргументу x) можна обчислити за допомогою `leftRect` зі «спеціальною» підінтегральною функцією, яка в свою чергу містить виклик `leftRect` для обчислення внутрішнього інтеграла (по аргументу y). У цьому випадку в програмі буде неявна рекурсія, проте C++ не накладає обмежень на такі виклики підпрограм:

```
// глобальні змінні
double t;
double X;

// спеціальні підінтегральні функції
double funParam(double x)
{
    // для інтеграла, залежного від параметра t
    return pow(x * t, 2) + 1;
}
// для подвійного інтеграла
double twoArg(double y)
{
    return pow(X + 2.*y, 2);
}
double firstInt(double x)
//функція, що обчислює внутрішній інтеграл у повторному
{
    X = x; // передавання другої змінної через глобальну
    return leftRect(0., x, twoArg, 0.0001);
}
void numIntegration()
{
    cout << "\n *Обчислення інтегралів методом лівих прямокутників*\n\n";
    // обчислимо одновимірний інтеграл
    cout << "Integral(sin,0,Pi) = " << leftRect(0., M_PI, sin) << '\n';
    // а тепер - серію S(t)
    cout << "\n  t      S(t)\n-----\n";
    for (int i = 0; i <= 6; ++i)
    {
        t = i*0.5;
        cout << "  " << t << '\t' << leftRect(-1., 1., funParam, 1e-5) << '\n';
    }
    // і нарешті - подвійний
```



```

    cout << "\n Подвійний інтеграл повторним числовим інтегруванням\n";
    cout << "Double integral = " << leftRect(0., 1., firstInt, 0.0001) << '\n';
    return;
}

```

Після виконання програми отримаємо такі результати.

Обчислення інтегралів методом лівих прямокутників

Integral(sin,0,Pi) = 2

t	S(t)
0	2
0.5	2.16667
1	2.66667
1.5	3.50001
2	4.66667
2.5	6.16667
3	8.00001

Подвійний інтеграл повторним числовим інтегруванням
Double integral = 1.08292

Легко перевірити, що вони збігаються з точними із заданою точністю. Задачу ми розв'язали, проте кидається в очі велика кількість допоміжних функцій, обчислення подвійного інтеграла дуже заплутане, розділене на декілька функцій у різних місцях програми, а використання глобальних змінних узагалі є порушенням канонів надійного програмування. Спробуємо виправити ситуацію за допомогою вже знайомих нам лямбда-виразів.

У тілі лямбда-виразу можна використовувати статичні змінні, оголошені в блоці функції, всередині якої «живе» цей вираз. Отже, замість глобальних змінних програми використаємо статичні змінні функції, а замість окремих підінтегральних функцій – відповідні лямбда-вирази.

```

void lambdaIntegration()
{
    cout << "\n *Обчислення інтегралів з використанням лямбда-виразів*\n\n";
    // обчислимо одновимірний інтеграл
    cout << "Integral(sin,0,Pi) = " << leftRect(0., M_PI, sin) << '\n';
    // а тепер - серію S(t)
    static double t; // параметр підінтегральної функції
    cout << "\n  t      S(t)\n-----\n";
    for (int i = 0; i <= 6; ++i)
    {
        t = i*0.5;
        cout << "    " << t << '\t'
              << leftRect(-1., 1.,
                          [](double x) {return pow(x * t, 2) + 1; }, 1e-5) << '\n';
    }
    // і нарешті - подвійний
    static double X; // другий аргумент підінтегральної функції
    cout << "\n Подвійний інтеграл повторним числовим інтегруванням\n";
    double I = leftRect(0., 1.,
                        [](double x) { X = x; return leftRect(0., x,
                                                                [](double y){ return pow(X + 2.*y, 2); }, 0.0001); },
                        0.0001);
}

```

```
    cout << "Double integral = " << I << '\n';  
    return;  
}
```

Такий код компактніший, надійніший (спеціальні змінні t і X локалізовані в тілі *lambdaIntegration*), зрозуміліший (наприклад, легко читається подвійне інтегрування) і видає такі самі результати, як попередня функція.