АННОТАЦИЯ

В работе предлагаются и обосновываются легко реализуемые численные методы приближённого поиска глобального минимума непрерывной на отрезке функции и непрерывной на брусе функции двух переменных, на которые не накладываются никакие дополнительные условия типа липшицевости. Единственным требованием к функции является её непрерывность на выпуклом компакте (отрезке или прямоугольнике). Приводятся результаты численных экспериментов. Методы могут быть полезны при решении задач негладкого математического программирования.

ANNOTATION

The paper proposes and proves an easily implemented numerical methods for the approximate search for a global minimum of a continuous function on an interval and of a continuous function of two variables on a rectangle without any additional constraints such as conditions of Lipschitz type. The only requirement for the function is its continuity on a convex set (interval or rectangle). The results of numerical experiments are given. Methods can be useful in solving problems of non-smooth mathematical programming.

ВВЕДЕНИЕ

Уже достаточно давно исследователями ставится задача нахождения глобального минимума функции одной или нескольких переменных. Ясно, что методы отыскания локального минимума не решают поставленную задачу, поскольку не дают уверенности в том, что найдены все локальные минимумы функции. Нужны какие-либо глобальные свойства функции на множестве ограничений. Одним из таких свойств является выпуклость функции на выпуклом множестве. Однако, это достаточно узкий класс функций. Другой, более широкий класс, это класс липшицевых функций на выпуклом множестве. В настоящее время известно достаточно много численных методов отыскания глобального минимума липшицевой функции. По-видимому, первым из таких методов является хорошо известный ныне метод Пиявского (метод ломаных). В след за указанным методом появились методы Евтушенко, Стронгина и другие.

Разумеется, что эти методы не решают задачу глобальной оптимизации для произвольной непрерывной функции, которая необязательно является липшицевой. Такой, например, является функция на отрезке , очевидно, имеющая глобальный минимум в точке .

Как выяснилось в работе [1], любая непрерывная на выпуклом компакте функция обладает некоторым глобальным свойством, обобщающим свойство липшицевости, которое может позволить создавать приближённые методы отыскания глобального минимума любой непрерывной на выпуклом компакте функции.

В предлагаемой работе строятся и обосновываются два приближённых алгоритма минимизации функции непрерывной на отрезке и функции непрерывной на замкнутом прямоугольнике.

Поясним вышесказанное.

В работе [1] было введено понятие ε-липшицевости:

Пусть функция определена на выпуклом компактном множестве из нормированного пространства . Она называется ε-липшицевой, если выполняется следующее условие:

(1)

Там же было доказано, что выполнение условия (1) **необходимо и достаточно** для непрерывности функции на . Оказалось, что свойство ε-липшицевости функции можно с успехом применять к созданию методов оптимизации негладких(может быть существенно негладких) функций. Так, в указанной работе [1], была предложена модификация метода Пиявского (метода ломаных) поиска приближённого минимума ε-липшицевой функции. Таким образом, открывается возможность минимизации непрерывных на компакте функций без добавления каких-либо дополнительных условий (кроме непрерывности) на минимизируемую функцию. Ясно, что в силу известной теоремы Вейерштрасса этот минимум существует, осталось его найти.

Следует заметить, что из определения (1) не следует положительности постоянной , которую будем называть ε-константой Липшица.

Для дальнейших выкладок для нас будет существенной её строгая положительность. В [2] было показано, что неравенство при заданном выполняется, если найдутся такие, что

(2)

Условие (2) было названо согласованностью функции и . Легко понять, что это условие не сильно сужает класс рассматриваемых функций, и мы будем всюду в дальнейшем полагать это условие выполненным.

Свойство ε-липшицевости вызвало появление ряда работ [3, 4, 5, 6], в каждой из которых приходилось решать вспомогательную задачу минимизации ε-липшицевой функции. При этом предполагалось, что значение оценки было известно или известна функциональная зависимость от ε. Отыскание указанных зависимостей является непростой задачей. Можно сказать, что непростой задачей является уже задача нахождения постоянной Липшица для липшицевых функций, задача же нахождения постоянной является задачей ещё более сложной. В последнее время появилась работа [2], в которой предложен один из возможных алгоритмов приближённого отыскания постоянной по заданному . В данной работе считается, что эта задача решена и нам известна минимальная оценка .

Одним из известных простых в реализации методов минимизации липшицевой функции является метод перебора по равномерной сетке. В предлагаемой работе строится и обосновывается обобщение упомянутого метода на случай ε-липшицевой функции.

В работе принята сквозная нумерация формул.

# Глава 1. Алгоритм равномерного перебора отыскания глобального минимума функции одной переменной непрерывной (ε-липшицевой) на отрезке

## Описание алгоритма и его обоснование

Здесь мы сформулируем и обоснуем алгоритм минимизации непрерывной функции на отрезке.

Пусть непрерывна на отрезке , её -постоянная Липшица, которая считается известной. Предполагается выполненным условие согласованности и (2): найдутся такие, что

Введём параметры алгоритма:

* ε – параметр, выбираемый из условия (1)
* – погрешность, с которой отыскивается приближённое значение минимума функции

Всюду предполагается выполнение неравенства .

Обозначим:

* - наименьшее значение функции на отрезке (которое, очевидно, существует);
* положим , где n-количество отрезков, на которые разбивается отрезок .

Потребуем выполнение неравенства

отсюда . Полученное условие позволяет найти количество пробных точек – узлов равномерной сетки, по которой ведётся перебор.

Построим последовательность пробных точек отрезка по следующему правилу:

Нашей задачей будет показать, что

или, что то же самое

.

Очевидно, для этого будет достаточно показать, что среди членов последовательности найдётся точка , для которой выполняется условие:

(3)

Обозначим и пусть . Тогда из неравенства (1) следует

или, что то же самое

(4)

Но поскольку , то выполняется следующее неравенство

Тогда из неравенства (4) следует:

(5)

Пусть - точка глобального минимума функции на отрезке . Очевидно, найдётся отрезок такой, что .

Тогда, из неравенства (5) следует:

Что и требовалось доказать.

## Замечание к алгоритму

Использование алгоритма предполагает априорное знание ε-константы Липшица для заданного ε.

Как видно из цитируемой литературы - лишь для некоторых элементарных функций удалось построить функциональную зависимость от ε. Последняя зависимость, из известных автору, получена в работе [6] для функции , что, как следует из работы, было сопряжено с трудностями. Однако, если дополнить приведённый алгоритм предварительным нахождением минимальной оценки ε-константы Липшица с помощью алгоритма, предложенного в [2], то указанная трудность снимается (правда ценой увеличения времени нахождения глобального минимума).

## Численный пример

В качестве тестовой задачи приближённого нахождения глобального минимума функции была рассмотрена функция вида

для которой в [5] была получена оценка

Для двух различных наборов постоянных приведём результаты численного эксперимента.

*Таблица 1.Результаты работы алгоритма*

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Отрезок | ε\* | ε |  |  |  |  |  |  | , *м*с |  |
| Набор 1. *b*1= *b*2= *b*3= 0; *a*1=–4, *a*2=–1, *a*3= 3 | | | | | | | | | |  |
|  |
|  | 0.01 | 0.005 | 50 | 0.0001 | –0.999999 | | 0.000001 | 99999 | 4 |  |
|  | 0.001 | 250 | 0.000036 | –1.000004 | | 0.002000 | 277777 | 11 |  |
| [–5; 5] |  |
| 0.001 | 0.0005 | 500 | 0.000001 | –3.999999 | | 0.000012 | 9999999 | 398 |  |
|  |  |
|  | 0.0001 | 2500 | 0.0000004 | –1.000000 | | 0.000202 | 27777777 | 1101 |  |
|  | 0.01 | 0.005 | 50 | 0.000100 | 2.999999 | | 0.000003 | 200000 | 9 |  |
|  | 0.001 | 250 | 0.000036 | –1.000000 | | 0.000007 | 555555 | 35 |  |
| [–10; 10] |  |
| 0.001 | 0.0005 | 500 | 0.000001 | 2.999999 | | 0.000026 | 20000000 | 751 |  |
|  |  |
|  | 0.0001 | 2500 | 0.0000004 | –1.000000 | | 0.000081 | 55555555 | 2059 |  |
|  | 0.01 | 0.005 | 50 | 0.000100 | 2.999999 | | 0.000004 | 300000 | 12 |  |
|  | 0.001 | 250 | 0.000036 | 2.999999 | | 0.000009 | 833333 | 32 |  |
| [–15; 15] |  |
| 0.001 | 0.0005 | 500 | 0.000001 | 2.999999 | | 0.000066 | 30000000 | 1148 |  |
|  |  |
|  | 0.0001 | 2500 | 0.0000004 | 2.999999 | | 0.000110 | 83333333 | 3059 |  |

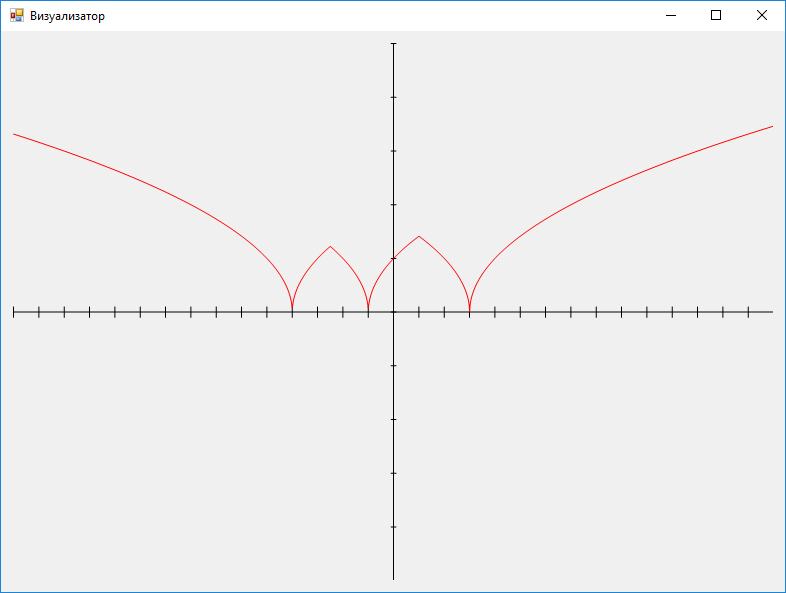


Рисунок 1 График функции с первым набором параметров

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Набор 2. *b*1=–1, *b*2=–1.005, *b*3= 0.5; *a*1=–4, *a*2=–1, *a*3= 3 | | | | | | | | |  |
|  |
|  | 0.01 | 0.005 | 50 | 0.000100 | –0.999999 | –1.004999 | 99999 | 3 |  |
| [–5; 5] | 0.001 | 250 | 0.000036 | –1.000004 | –1.003000 | 277777 | 9 |  |
|  |
| 0.001 | 0.0005 | 500 | 0.000001 | –0.999999 | –1.004982 | 9999999 | 367 |  |
|  |  |
|  | 0.0001 | 2500 | 0.0000004 | –1.000000 | –1.004798 | 27777777 | 997 |  |
|  | 0.01 | 0.005 | 50 | 0.000100 | –1.000000 | –1.004997 | 200000 | 7 |  |
|  | 0.001 | 250 | 0.000036 | –1.000000 | –1.004993 | 555555 | 23 |  |
| [–10; 10] |  |
| 0.001 | 0.0005 | 500 | 0.000001 | –1.000000 | –1.004973 | 20000000 | 711 |  |
|  |  |
|  | 0.0001 | 2500 | 0.0000004 | –1.000000 | –1.004919 | 55555555 | 1971 |  |
|  | 0.01 | 0.005 | 50 | 0.000100 | –1.000000 | –1.004995 | 300000 | 10 |  |
|  | 0.001 | 250 | 0.000036 | –0.999996 | –1.003000 | 833333 | 43 |  |
| [–15; 15] |  |
| 0.001 | 0.0005 | 500 | 0.000001 | –1.000000 | –1.004933 | 30000000 | 1056 |  |
|  |  |
|  | 0.0001 | 2500 | 0.0000004 | –1.000000 | –1.004834 | 83333333 | 2871 |  |

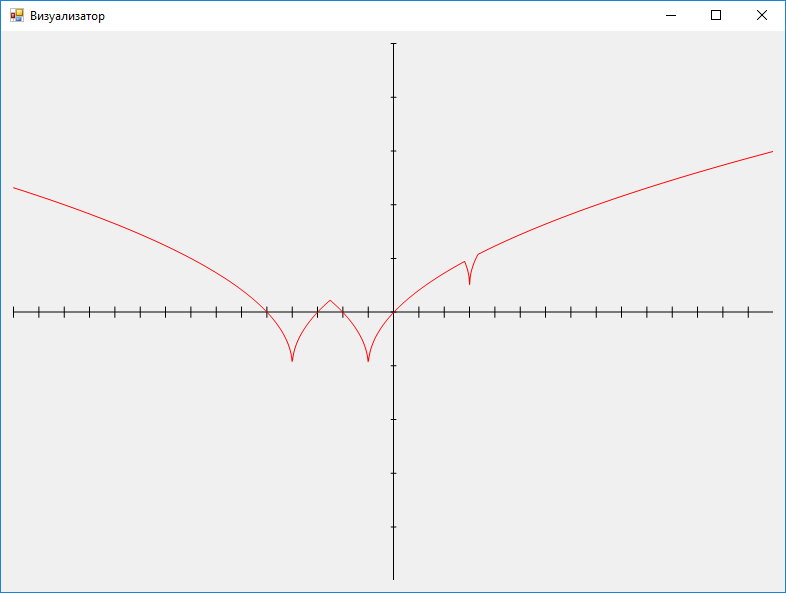


Рисунок 2 График функции со вторым набором параметров

Сравнивая с результатами, полученными в [5], мы видим, что при более простой реализации алгоритма требуется больше пробных точек, однако время, необходимое на выполнение алгоритма, значительно уменьшается.

Другие примеры работы алгоритма будут показаны ниже в главе 3.

# Глава 2. Случай функции двух переменных

Далее рассмотрим случай применения метода равномерного перебора поиска глобального минимума ε-липшицевой функции двух переменных.

Дана функция непрерывная на , её -постоянная Липшица, которая считается известной. Предполагается выполненным условие согласованности и (2): найдутся такие, что

Прямоугольник, на котором будет производиться поиск минимума, разделим сеткой размера . Введём следующие обозначения:

* ε – параметр, выбираемый из условия (1)
* – погрешность, с которой отыскивается приближённое значение минимума функции
* – шаг алгоритма по оси ;
* – шаг алгоритма по оси ;
* - наименьшее значение функции на области (которое, очевидно, существует);
* - узел сетки, являющийся нижней левой точкой ячейки , где , на которые разбивается прямоугольник .

Определим значение . Для этого потребуем выполнения неравенства

Тогда, очевидно

И, преобразовав это неравенство, получаем нижнюю границу величины шага по оси :

Аналогичным образом получаем оценку числа разбиений по оси :

Нашей задачей будет доказать истинность следующего неравенства:

Найдётся такой, что

(6)

(смотри рисунок 1)



*Рисунок 3 Геометрическое пояснение неравенства (6)*

Или, что то же самое

В качестве нормы удобно будет выбрать покоординатную норму

Тогда:

Так как соседние узлы сетки не могут располагаться друг от друга больше, чем на величину шага, то

А так как и и ограничены сверху одной величиной , можно записать

Откуда получаем требуемое неравенство

Что и требовалось доказать.

## Замечание

Строго говоря, -константы Липшица по осям и могут различаться, но мы можем взять большую из них и использовать как универсальную

И тогда величина шага по обеим осям станет одинаковой

В прочем, небольшое усложнение алгоритма возможно и для различающихся констант и .

## Численный пример

В качестве тестовой задачи приближённого нахождения глобального минимума функции была рассмотрена функция вида

для которой в [1] была получена оценка

*Таблица 2.Результаты работы алгоритма*

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Брус | ε\* | ε |  |  |  |  |  |  |  | , *м*с |  |
|  | 0.05 | 0.01 | 26 | 0.001538 | 0.000000 | -0.000027 | 0.005210 | 1300 | 2043 | 117 |  |
| 0.001 | 251 | 0.000195 | -0.000088 | -0.000063 | 0.008044 | 10245 | 16093 | 7191 |  |
|  |
| 0.01 | 0.005 | 51 | 0.000098 | 0.000000 | -0.000012  0.000008 | 0.003472 | 20400 | 32045  87616 | 28118 |  |
|  |
| 0.001 | 251 | 0.000036 | 0.000004 | 0.002904 | 55778 | 221548 |  |
|  | 0.05 | 0.01 | 26 | 0.001538 | 0.000000 | 0.000000 | 0.000000 | 650 | 2043 | 65 |  |
| 0.001 | 251 | 0.000195 | -0.000044 | 0.000000 | 0.000044 | 5123 | 16093 | 3627 |  |
|  |
| 0.01 | 0.005 | 51 | 0.000098 | 0.000000 | 0.000000  0.000000 | 0.000000 | 10200 | 32045  87616 | 14249 |  |
|  |
| 0.001 | 251 | 0.000036 | -0.000016 | 0.000016 | 27889 | 106840 |  |

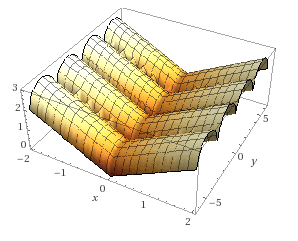


Рисунок 4 График рассматриваемой функции

# Глава 3. Расчёт численных примеров

Помимо приведённых численных примеров в предыдущих главах, был проведён ряд численных экспериментов над разработанными алгоритмами с другими функциями, некоторые из которых будут приведены ниже.

на интервале [–2; 3], если известна её -константа Липшица

*Таблица 3.Результаты работы алгоритма*

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Отрезок | ε\* | ε |  |  |  |  |  |  | , *м*с |  |
|  | 0.01 | 0.005 | 51 | 0.000098 | 0.000044 | | 0.006647 | 2796 | 0 |  |
|  | 0.001 | 251 | 0.000036 | –0.000013 | | 0.0036 | 7645 | 0 |  |
| [–2; 3] |  |
| 0.001 | 0.0005 | 501 | 0.000001 | 0.000000 | | 0.000636 | 274703 | 12 |  |
|  |  |
|  | 0.0001 | 2501 | 0.0000004 | 0.000000 | | 0.00031 | 761848 | 35 |  |

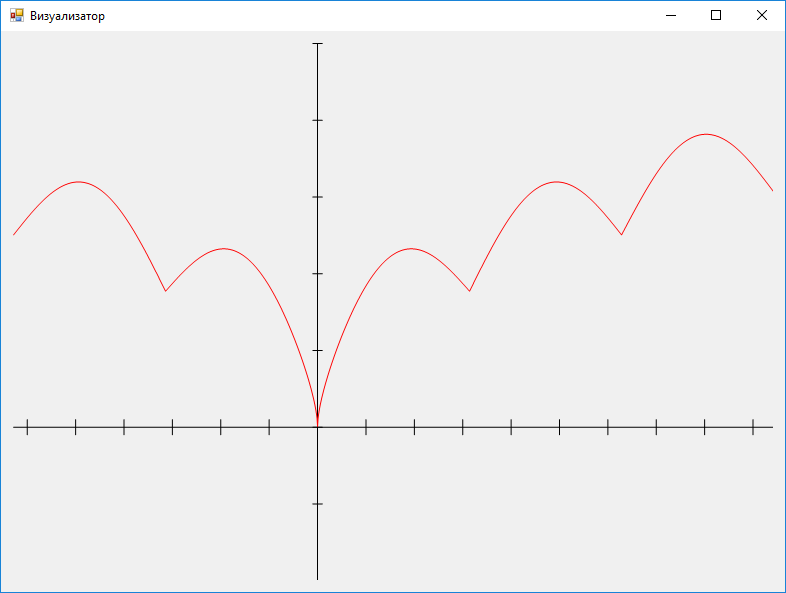


Рисунок 5 График исследуемой на минимум функции

на интервале [–2; 3], если известна её -константа Липшица

*Таблица 4.Результаты работы алгоритма*

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Отрезок | ε\* | ε |  |  |  |  |  |  | , *м*с |  |
|  | 0.01 | 0.005 | 51 | 0.000098 | 0.000044 | | 0.006647 | 2796 | 0 |  |
|  | 0.001 | 251 | 0.000036 | –0.000013 | | 0.0036 | 7645 | 0 |  |
| [–2; 3] |  |
| 0.001 | 0.0005 | 501 | 0.000001 | 0.000000 | | 0.000636 | 274704 | 14 |  |
|  |  |
|  | 0.0001 | 2501 | 0.0000004 | 0.000000 | | 0.00031 | 761848 | 32 |  |

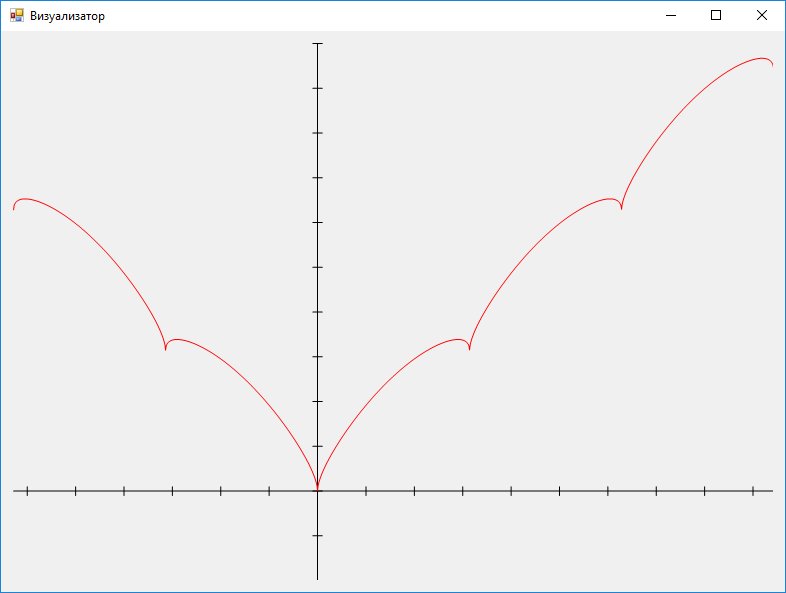


Рисунок 6 График исследуемой на минимум функции

на брусе , если известна её -константа Липшица

*Таблица 5.Результаты работы алгоритма*

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Брус | ε\* | ε |  |  |  |  |  |  |  | , *м*с |  |
|  | 0.05 | 0.01 | 26 | 0.001538 | -0.000027 | 0.000000 | 0.005210 | 2043 | 1300 | 143 |  |
| 0.001 | 251 | 0.000195 | -0.000063 | -0.000088 | 0.008044 | 16093 | 10245 | 7166 |  |
|  |
| 0.01 | 0.005 | 51 | 0.000098 | -0.000012 | 0.000000  0.000004 | 0.003472 | 32045 | 20400  55778 | 28302 |  |
|  |
| 0.001 | 251 | 0.000036 | 0.000008 | 0.002904 | 87616 | 212089 |  |

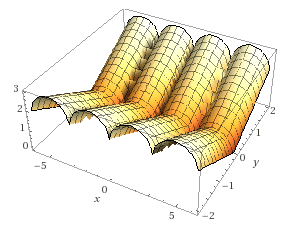
**

Рисунок 7 График исследуемой на минимум функции



на брусе , если известна её -константа Липшица

*Таблица 6.Результаты работы алгоритма*

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Брус | ε\* | ε |  |  |  |  |  |  |  | , *м*с |  |
|  | 0.05 | 0.01 | 26 | 0.001538 | 0.000000 | -0.000027 | 0.000027 | 1300 | 2043 | 122 |  |
| 0.001 | 251 | 0.000195 | -0.000088 | -0.000063 | 0.009425 | 10245 | 16093 | 7390 |  |
|  |
| 0.01 | 0.005 | 51 | 0.000098 | 0.000000 | -0.000012  0.000008 | 0.000012 | 20400 | 32045  87616 | 29085 |  |
|  |
| 0.001 | 251 | 0.000036 | 0.000004 | 0.002004 | 55778 | 217442 |  |

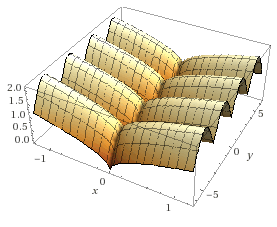


Рисунок 8 График исследуемой на минимум функции

# Глава 4. Программная реализация алгоритмов

## Обоснование выбора языка и среды программирования

Программная реализация алгоритмов была выполнена на языке программирования C#, поскольку это отлично зарекомендовавший себя на рынке во всём мире язык со множеством удобного функционала, работающего «из коробки». К числу его положительных черт можно отнести:

* это объектно-ориентированный, мультипарадигмальный, рефлективный язык программирования
* C# является «родным» языком программирования для среды Microsoft .Net, потому что он самым тесным и эффективным образом интегрирован с ней
* При компилировании программы CIL(Common Intermediate Language - «высокоуровневый ассемблер» виртуальной машины [.NET](https://ru.wikipedia.org/wiki/Microsoft_.NET)) – инструкции распространяются в специальных единицах – сборках, и сопровождаются своими метаданными, что делает её самодостаточным объектом. Благодаря тому, что в исполняемом файле, помимо самой программы, находятся метаданные, можно получать объекты, которые описывают типы, модули и сборки во время исполнения кода (run-time).
* Автоматическое управление памятью;
* Библиотека .NET Framework содержит богатый набор классов для работы с различными видами данными, такими как: ввод / вывод информации, работа с базами данным, работа с сетью, работа с графикой и так далее.
* Для C# кода предоставляется огромный набор «синтаксического сахара», позволяющего создавать лаконичный, понятный для чтения код. Также, сокращая код, необходимый для реализации одних и тех же действий, уменьшается шанс допустить ошибку. Примерами «синтаксического сахара» являются: лямбда выражения, анонимные типы, LINQ (набор появившихся в .NET функций, которые значительно расширяют возможности синтаксиса языков C# и Visual Basic).

В качестве среды программирования была выбрана MS Visual Studio 2017 Community, как последняя версия Visual Studio на момент начала разработки. Community версия, к тому же, распространяется абсолютно бесплатно и при этом сохраняет весь свой основной функционал от старших версий.

## Проектирование программного обеспечения

Разработка любых программных решений начинается с их проектирования. Данный процесс очень важен, так как любая ошибка может привести к тому, что на некотором этапе разработки придётся с нуля пересматривать архитектуру некоторых модулей или всего программного комплекса.

На этапе проектирования разрабатываются архитектура будущего программного обеспечения, устройство его компонентов и пользовательские интерфейсы. В зависимости от рода и сложности создаваемой программы, проектирование может обеспечиваться как «ручным», так и автоматизированным проектированием. Для выражения характеристик программного обеспечения разрабатываются документы и схемы в различных нотациях, такие как ER-диаграммы, UML-диаграммы, блок-схемы, структурные схемы, макеты и другие.

С точки зрения целей будущего использования, программное решение удобнее всего реализовать в формате библиотеки классов (DLL(англ. Dynamic Link Library — «библиотека динамической компоновки») в среде .Net). Далее будут описаны и обоснованы реализации разработанных алгоритмов минимизации.

## Реализация алгоритма равномерного перебора отыскания глобального минимума непрерывной (ε-липшицевой) на отрезке функции одной переменной

Для начала, удобнее всего будет представить данной в главе 1 словесное описание алгоритма в виде блок-схемы. Данный вид описания алгоритмов подчиняется Государственному стандарту - ГОСТ 19.701-90 «Схемы алгоритмов программ, данных и систем» [8]. Этот ГОСТ практически полностью соответствует международному стандарту ISO 5807:1985

На рисунке 9 изображена блок-схема алгоритма равномерного перебора отыскания глобального минимума непрерывной на отрезке функции одной переменной.

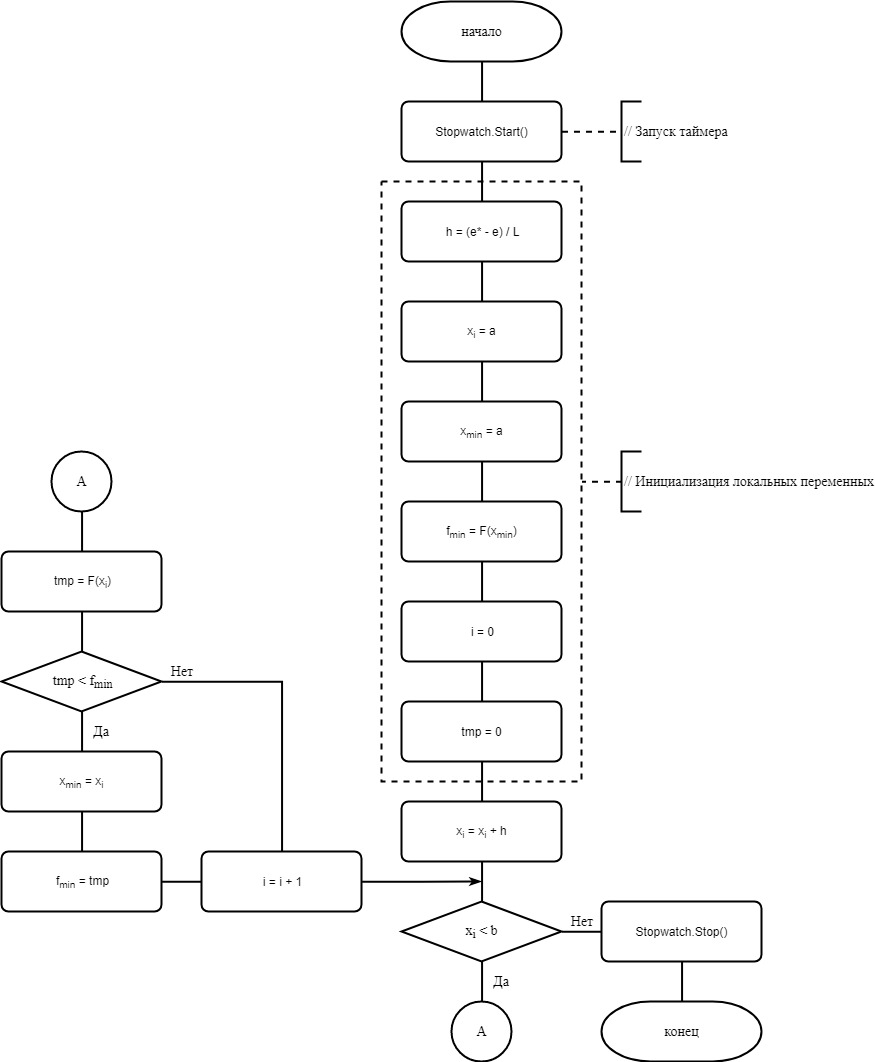


Рисунок 9 Блок-схема алгоритма случая функции одной переменной

По аналогии со стандартным пространством имён Math в среде .Net, было принято решение реализовывать алгоритм в виде статической функции, так как для работы алгоритма нет необходимости хранить никакие данные от запуска к запуску, а входными параметрами и выходными данными являются стандартные структуры среды. Получившийся исходный код функции приведён ниже.

/// <summary>

/// Модификация метода равномерного перебора поиска глобального минимума для эпсилон-липшецевых функций

/// </summary>

/// <param name="F">Исследуемая на глобальный минимум функция</param>

/// <param name="a">Левая граница отрезка</param>

/// <param name="b">Правая граница отрезка</param>

/// <param name="L">Константа Липшица</param>

/// <param name="e">Параметр, выбираемый из условия e-Липшицевости</param>

/// <param name="e2">Погрешность, с которой отыскивается приближённое значение минимума функции</param>

/// <returns>

/// x-значение на оси Ox, в котором достигается глобальный минимум;

/// F-глобальный минимум переданной функции на рассматриваемом отрезке;

/// n-количество пробных точек(итераций);

/// time-время, затраченное на выполнение поиска.</returns>

public static (double h, double x, double F, double n, long time) UniformSearchByBiryukov(Func<double, double> F, double a, double b, double L, double e, double e2)

{

// Таймер для приблизительного измерения производительности алгоритма

var sw = new Stopwatch();

sw.Start();

double h = (e2 - e) / L // шаг

, xi = a // координата на оси Ox, значение функции в которой рассмтаривается на текущей итерации

, xMin = xi // координата оси Ox, на которой достигается лучшее приближение к глобальному минимуму функции на текущей итерации

, fMin = F(xMin) // лучшее приближение к глобальному минимуму функции на текущей итерации

, tmp; // временное хранилище для подмены лучшего приближения к глобальному минимуму

int i = 0;

while ((xi += h) < b)

{

// если значение функции в текущей точке меньше последнего сохранённого - заменяем его

if ((tmp = F(xi)) < fMin)

{

xMin = xi;

fMin = tmp;

}

i++;

}

sw.Stop();

return (h, xMin, fMin, i, sw.ElapsedMilliseconds);

}

Входными параметрами алгоритма являются: исследуемая на минимум функция , координаты начала и конца отрезка, постоянная Липшица и параметры и .

Результатом выполнения функции являются: шаг , значение на оси , в котором достигается глобальный минимум переданной функции, глобальный минимум функции, количество пробных точек и затраченное на выполнение алгоритма время .

## Реализация алгоритма для случая функции двух переменных

Алгоритм равномерного перебора отыскания глобального минимума непрерывной (ε-липшицевой) на брусе функции двух переменных, что очевидно, очень похож на алгоритм для случая одной переменной.

При его реализации возникает задача выбора построения и обхода сетки, налагаемой на брус. В работе принят следующий алгоритм обхода:

* зафиксировать значение по оси ;
* обработать узлы сети при фиксированном значении ;
* перейти к следующему значению в соответствии с выбранным шагом.

Это не единственно возможный способ, однако от выбора способа обхода сетки суть алгоритма останется неизменной. На рисунке 10 приведена блок-схема алгоритма для случая функции двух переменных.

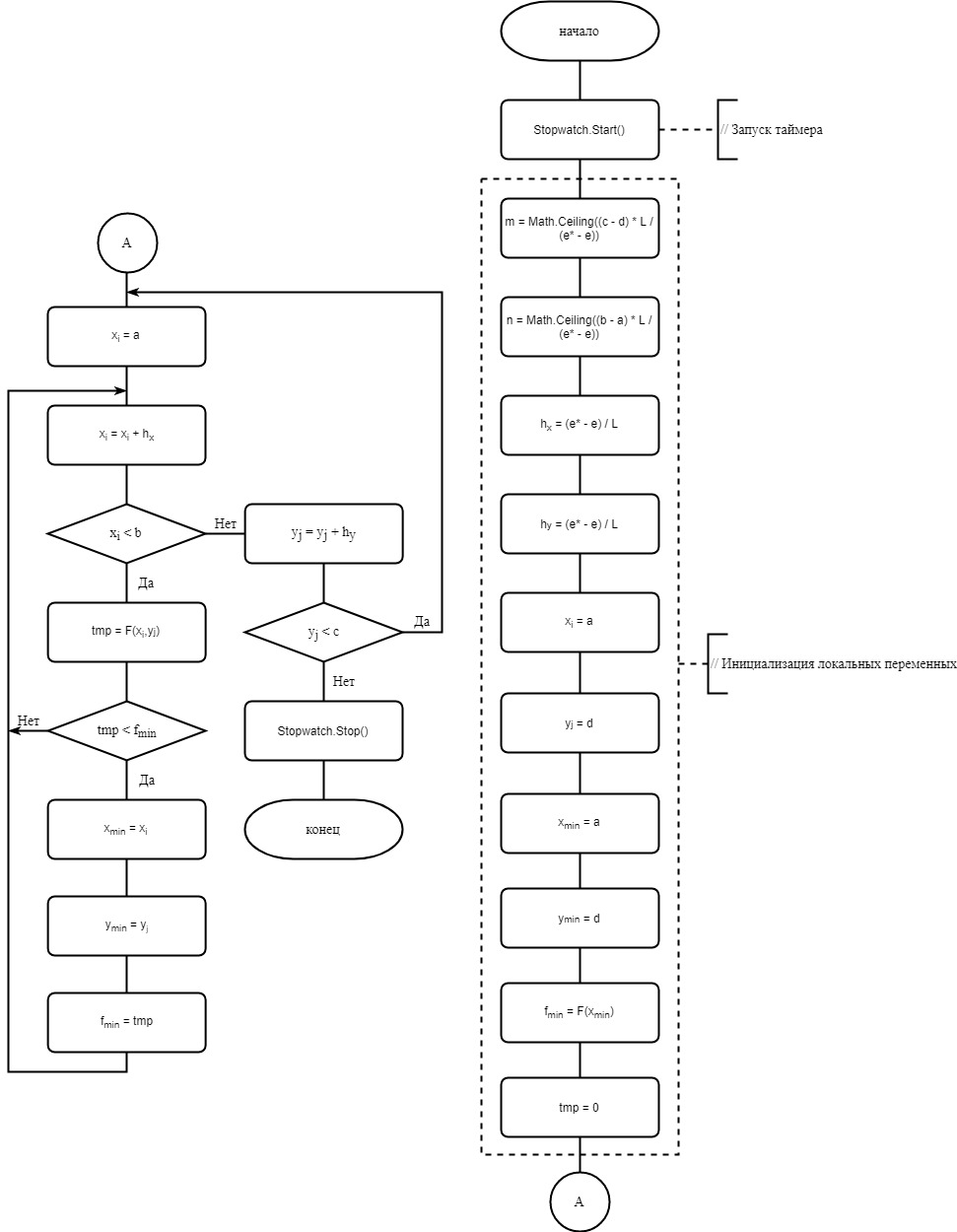


Рисунок 10 Блок-схема алгоритма случая функции двух переменных

Как и для предыдущего случая, алгоритм реализован в виде статической функции по тем же причинам. Причём случай функции двух переменных выполнен как перегрузка метода, решающего случай функции одной переменной, основываясь на принципе полиморфизма. Исходный код приведён ниже.

/// <summary>

/// Модификация метода равномерного перебора поиска глобального минимума для эпсилон-липшецевых функций двух переменных

/// </summary>

/// <param name="F">Исследуемая на глобальный минимум функция</param>

/// <param name="a">Левая граница бруса</param>

/// <param name="b">Правая граница бруса</param>

/// <param name="d">Нижняя граница бруса</param>

/// <param name="c">Верхняя граница бруса</param>

/// <param name="L">Константа Липшица</param>

/// <param name="e">Параметр, выбираемый из условия e-Липшицевости</param>

/// <param name="e2">Погрешность, с которой отыскивается приближённое значение минимума функции</param>

/// <returns>

/// x-значение на оси Ox, в котором достигается глобальный минимум;

/// y-значение на оси Oy, в котором достигается глобальный минимум;

/// F-глобальный минимум переданной функции на рассмтариваемом отрезке;

/// n-количество пробных точек по горизонтали;

/// n-количество пробных точек по вертикали;

/// time-время, затраченное на выполнение поиска.</returns>

public static (double hx, double hy, double x, double y, double F, double n, double m, long time) UniformSearchByBiryukov(Func<double, double, double> F, double a, double b, double d, double c, double L, double e, double e2)

{

// Таймер для приблизительного измерения производительности алгоритма

var sw = new Stopwatch();

sw.Start();

double m = Math.Ceiling((c - d) \* L / (e2 - e)), // число минимально необходимых пробных точек по оси Oy

n = Math.Ceiling((b - a) \* L / (e2 - e)); // число минимально необходимых пробных точек по оси Ox

double hx = (e2 - e) / L // шаг по оси Ox

, hy = (e2 - e) / L // шаг по оси Oy

, xi = a // координата на оси Ox, значение функции в которой рассмтаривается на текущей итерации

, yi = d // координата на оси Oy, значение функции в которой рассмтаривается на текущей итерации

, xMin = xi // координата оси Ox, на которой достигается лучшее приближение к глобальному минимуму функции на текущей итерации

, yMin = yi // координата оси Oy, на которой достигается лучшее приближение к глобальному минимуму функции на текущей итерации

, fMin = F(xMin, yMin) // лучшее приближение к глобальному минимуму функции на текущей итерации

, tmp; // временное хранилище для подмены лучшего приближения к глобальному минимуму

do

{

xi = a;

while ((xi += hx) < b)

{

// если значение функции в текущей точке меньше последнего сохранённого - заменяем его

if ((tmp = F(xi, yi)) < fMin)

{

xMin = xi;

yMin = yi;

fMin = tmp;

}

}

}

while ((yi += hy) < c);

sw.Stop();

return (hx, hy, xMin, yMin, fMin, n, m, sw.ElapsedMilliseconds);

}

Входными параметрами алгоритма являются: границы бруса , постоянная Липшица и параметры и .

Результатом выполнения функции являются: шаг по оси , шаг по оси , значения на оси и на оси , в которых достигается глобальный минимум переданной функции, глобальный минимум функции, количество пробных точек по горизонтали и по вертикали, и затраченное на выполнение алгоритма время .

# Глава 5. Тестирование программного продукта

## Основные понятия и принципы тестирования программного продукта

Тестирование программного любого продукта, вне зависимости от его назначения или объёма исходного кода, является важной частью разработки программного продукта. Без полноценно проведённого тестирования невозможно оценить, насколько качественно программный продукт выполнен. Чем качественнее будут проведены тесты, тем меньше вероятность, что будут упущены ошибки, которые в будущем придётся исправлять

Тестирование — это процесс испытания, исследования [программного продукта](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5), имеющий следующие основные цели:

* показать, что программный продукт выполнен качественно и в соответствии со всеми требованиями;
* обнаружить такие ситуации, при возникновении которых поведение программы является некорректным, нежелательным или вовсе непредсказуемым.

Информационные потоки процесса тестирования изображены на рисунке 11.

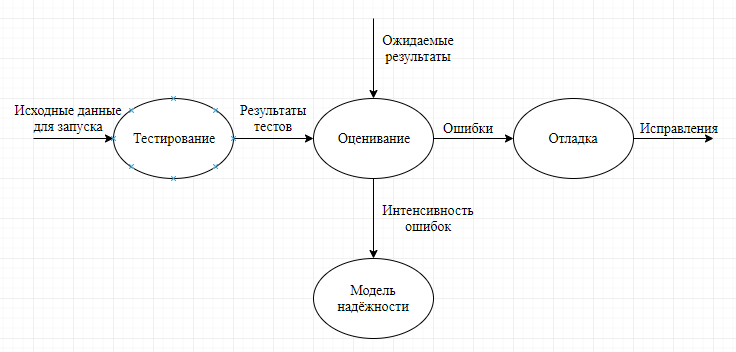


Рисунок 11 Информационные потоки процесса тестирования

В тех случаях, когда программный продукт разрабатывается для массового применения, отсутствие должного тестирования, а в следствии, упущение ряда ошибок, может вызвать негативное отношение к нему среди пользователей.

Сколь хорошо не были бы проведены тесты, их результаты не могут однозначно указывать на полное отсутствие возможности возникновения ошибок. Для наиболее полного определения качества программного обеспечения производится анализ совокупности следующих составляющих:

* [надёжность](https://ru.wikipedia.org/wiki/%D0%9D%D0%B0%D0%B4%D1%91%D0%B6%D0%BD%D0%BE%D1%81%D1%82%D1%8C);
* [сопровождаемость](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D0%BF%D1%80%D0%BE%D0%B2%D0%BE%D0%B6%D0%B4%D0%B5%D0%BD%D0%B8%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D1%8F);
* практичность;
* эффективность;
* мобильность;
* функциональность.

В процессе тестирования для каждого модуля программного обеспечения создаются наборы тестов с вариациями исходных значений и ожидаемыми результатами отработки модулей.

По выполнению каждого теста результаты их исполнения проверяются или, другими словами, реальные результаты исполнения тестов сравниваются с ожидаемыми, заранее подготовленными и проверенными результатами. При обнаружении несовпадений, фиксируется ошибка. Каждая найденная ошибка обязательно должна быть исправлена, поэтому начинается процесс поиска причин неверной работы модулей программного продукта и его отладка. На поиск места дефекта и исправление может потребоваться от пары минут до нескольких дней. Эта неопределенность приводит к затруднению планирования действий.

По окончании отладки на исправленные места исходных кодов пишутся новые, либо исправляются старые тесты и процесс проверки начинается заново.

Если функции программного обеспечения реализованы правильно, а обнаруженные ошибки легко исправляются, может быть сделан один из двух выводов:

* качество и надежность ПО удовлетворительны;
* тесты не способны обнаруживать серьезные ошибки.

Существуют 2 принципа тестирования программы:

* функциональное тестирование (тестирование «черного ящика»);
* структурное тестирование (тестирование «белого ящика»). [9]

## Тестирование, применённое к разрабатываемому программному обеспечению

На данный момент разработано большое число различных методологий проведения тестирования программных продуктов, такие как функциональное, нагрузочное, интеграционное и другие тестирования. Однако, все они подразумевают если не понимание «изнанки» процессов, то, как минимум, наличие этих самых процессов, на которые по окончании или во время разработки пишутся тесты. Однако существует другая методология, основанная на том, что первым делом разрабатываются именно тестовые наборы, а лишь потом пишется код, ими проверяемый. Такой подход, как никакой другой подходит для процесса разработки математической библиотеки, когда заранее известны численные примеры с их точными решениями. Такой подход используется в методологии TDD.

Разработка через тестирование (TDD, test-driven development) - это процесс разработки программного обеспечения, который основывается на повторении очень короткого цикла разработки: сначала разработчик пишет (изначально неудачный) автоматический тестовый пример, который определяет желаемое улучшение или новую функцию, затем выдает минимальное количество кода, чтобы пройти этот тест, и, наконец, рефакторит новый кода в соответствии с приемлемыми стандартами. [10]

Обычно выполняется следующая последовательность шагов:

* Добавить тест;
* Запустите все тесты и посмотрите, не сработает ли новый;
* Написать код;
* Запустить тесты;
* Рефакторинг кода;
* Повторение.

## Реализация тестовых примеров

Исходя из сказанного в предыдущем пункте, первым шагом при разработке программного продукта было создание автоматизированных тестов на основании рассчитанных заранее численных примеров. Однако эти тесты необходимо ещё реализовать. Тут на помощь может прийти любая из существующих библиотек разработки автоматизированных тестов: NUnit, xUnit, JUnit или даже встроенная в Visual Studio система построения и исполнения тестов.

В рамках данной дипломной работы для реализации тестовых примеров был использован фрэймворк xUnit совместно с библиотекой FluentAssertions. Оба этих средства доступны для скачивания и внедрения в своё решение через NuGet - систему управления пакетами для платформ разработки Microsoft.

xUnit.net - это бесплатный инструмент для модульного тестирования в среде .NET Framework с открытым исходным кодом. Написанная первоначальным изобретателем NUnit v2, xUnit - это новейшая технология для модульного тестирования C #, F #, VB.NET и других языков .NET. xUnit работает с ReSharper, CodeRush, TestDriven.NET и Xamarin. Он является частью .NET Foundation и действует в соответствии с их кодексом поведения. Лицензируется под Apache 2 (лицензия, утвержденная OSI) [11].

FluentAssertions – библиотека, предоставляющая обширный набор методов расширений, которые позволяют более естественно указать ожидаемый результат модульных тестов в стиле TDD и BDD.

Связка двух вышеуказанных инструментов позволяет очень удобно и продуктивно работать над написанием автоматизированных модульных тестов.

Пример одного из тестов, который на самом деле при запуске исполнительной среды xUnit разворачивается в три различных теста, приведён ниже.

[Theory]

[InlineData(-Math.PI, 3 \* Math.PI, 0.001, 0.01)]

[InlineData(-Math.PI / 2.0, Math.PI / 2.0, 0.0005, 0.001)]

[InlineData(0, Math.PI, 0.01, 0.1)]

public void OneDimensionF1(double a, double b, double e, double e2)

{

// Рассматриваемая функция

double F(double x)

=> Math.Abs(x) + Math.Sqrt(Math.Abs(Math.Sin(x)));

// ожидаемый результат

double fMin = 0;

// полученное решение

var result = MathStrategy.UniformSearchByBiryukov(

F,

a, b, // [a;b]

L(e), // L=L(e)=1/(4e)

e, e2); // e, e\*

// Проверка прохождения теста

(result.F - fMin).Should().BeLessOrEqualTo(e2);

}

Атрибут [Theory] обозначает параметризованный тест, который является истинным для подмножества данных. Эти данные могут быть предоставлены несколькими способами, но наиболее распространённым является атрибут [InlineData]. Таким образом происходит выполнение тела одного и того же модульного теста на различных наборах исходных данных.

# Руководство пользователя