Research Article

Mikhail Posypkin* and Alexander Usov

# Implementation and verification of global optimization benchmark problems

https://doi.org/10.1515/eng-2017-0050
Received Aug 12, 2017; accepted Nov 19, 2017

**Abstract:** The paper considers the implementation and verification of a test suite containing 150 benchmarks for global deterministic box-constrained optimization. A C++ library for describing standard mathematical expressions was developed for this purpose. The library automate the process of generating the value of a function and its' gradient at a given point and the interval estimates of a function and its' gradient on a given box using a single description. Based on this functionality, we have developed a collection of tests for an automatic verification of the proposed benchmarks. The verification has shown that literary sources contain mistakes in the benchmarks description. The library and the test suite are available for download and can be used freely.

**Keywords:** optimization test problems, global optimization, interval analysis, function estimation, function gradient

## 1 Introduction

Test suites are inevitable to develop new algorithms for global optimization, as well as study existing optimization methods. Tests for optimization are available in many forms. There are several previous studies containing the collections of test problem. To the best of our knowledge the most comprehensive test suite for bound-constrained global optimization is considered in [1]. Some test suites are available online as modules for various programming languages. It is worth to mention CUTEr [2], which is a versatile testing environment for optimization and linear algebra solvers. The package contains a collection of test problems, along with Fortran 77, Fortran 90/95 and Matlab tools intended to help developers design, compare and improve new and existing solvers. Many Internet sources provide collections of global optimization benchmarks, for example [3]. In addition, there are automated generators of test functions in [4].

Also, there are several techniques for comparing global optimization algorithms. For example, [5] introduces a methodology allowing one to compare stochastic and deterministic methods. The article [6] is dedicated to a comparison between nature-inspired metaheuristic and deterministic algorithms. The systematic review of the benchmarking process of optimization algorithms is given in [7].

It is important to note that in [1], functions were collected from various literary sources. Our careful examination showed that in the process of rewriting errors were made in more than 30% of the tests. Thus, it is very important to verify the test suite. We used for these purposes the deterministic global optimization approach.

In addition to calculating the value of an objective function, various methods on estimating a function on a given box are used in the methods of global optimization. For example, the evaluation of the enclosing interval of a function is calculated by the interval analysis method [8] or by using Lipschitzian properties. Manual programming of these methods is time consuming and error-prone. We automated these tasks: the interval bounds are computed based on the same internal representation as for the value.

This article describes the approach to creating benchmark functions that calculate the value of the objective function at a given point and also to automatically obtain the interval bounds of the function on a given box using a single description of a mathematical expression. As a result of this approach, the test suite of 150 C++ template functions was created. Practically all the functions for the test suite were taken from [1] and checked with the original sources and automatically verified with the interval global optimization method. In total, four types of unit tests were developed based on the Google C ++ Testing Framework.

The distinguished feature of our approach is that we verified the test suite using global optimization methods and that we provide a flexible C++ interface to bench-

***Corresponding Author: Mikhail Posypkin:** Dorodnicyn Computing Centre, FRC CSC RAS, Russian Federation;
Email: mposypkin@gmail.com
**Alexander Usov:** Dorodnicyn Computing Centre, FRC CSC RAS, Russian Federation

marks. This interface supports standard methods for computing objective's values and gradients as well as more complex but highly demanded in global optimization interval estimates.

## 2 Description of the benchmark functions

Let's exemplify our approach on the `Rosenbrock` (1) and `DropWave` (2) functions.

$$f(x) = \sum_{i=1}^{D-1} \left[ 100 \cdot \left( x_{i+1} - x_i^2 \right)^2 + (x_i - 1)^2 \right] \quad (1)$$

$$f(x) = -\frac{1 + \cos\left(12 \cdot \sqrt{x_i^2 + x_2^2}\right)}{0.5 \cdot (x_1^2 + x_2^2) + 2} \quad (2)$$

The function (1) can have an arbitrary number of variables while the second one has exactly two parameters. The C ++ template function for a `Rosenbrock` function is depicted at Figure 1.

```
template <class T>
Expr<T> Rosenbrock(int n)
{
        Expr<T> x;
        Iterator i(0, n-2);
        Expr<T> t = i;
        return  loopSum(100*sqr(x[t+1]-sqr(x[i]))+sqr((x[i]-
        1)), i);
}
```

**Figure 1:** `Rosenbrock` function.

Let's look into this section of code in detail. The input parameter *n* is a number of function variables of the type *T*. The template parameter *T* is either a C++ standard real (double or float) type or an `Interval<>` type for working with interval estimates (Section III). The function returns an object of type `Expr<T>` which allows to calculate the value of a function at a given point or to obtain an interval estimate of the function over a given box (Section IV).

The implementation of the function body is based on the mathematical expressions library outlined in Section II. The variable at Figure 1 of type `Expr<T>` is the simplest possible expression that describes the function parameters vector. The variable of the `Iterator` type is used as an index to access vector's elements.

The iterator constructor has two input parameters meaning the range of the index variable. Note that the index of the variable does not necessary has an `Iterator` type. It can also be any integral expression. The example is given at the Figure 1: initially, the variable is initialized by the iterator and then the expression is used as an index to access the elements of . The `loopSum` returns the expression implementing the summation using the iterator *i*. Another method `sqr` returns a mathematical expression of type `Expr<T>` passed to the method as an input parameter to quadratic power. Both of these functions are described in the mathematical expressions library covered in the Section III.

`Rosenbrock` function depicted at the Figure 1 returns an object representing a mathematical expression to the calling code. Note that the function description is specified once. The resulting expression allows to calculate the value of the function/gradient at given points or calculate the interval estimate of the function/gradient over a box.

Besides a function a benchmark contains constraints. In this particular case we consider only interval constraints. The remaining information for the benchmark is stored as a meta description (Figure 2).

```
"Rosenbrock" : {
   "description": "Rosenbrock function",
   "anyDim" : true,
   "bounds": [
         {"a": -30.0, "b": 30.0}
   ],
   "globMin": [
         { "x" : [1.0] }
   ],
   "globMinY": 0.0
}
```

**Figure 2:** Meta description for the `Rosenbrock` benchmark in JSON format.

In the meta description the following JSON fields are used:

- `description` – name of the global optimization benchmark;
- `anyDim` – *true* or *false* flag describing the type of the dimensionality. If the flag is set to *true*, then the size of the parameter's vector can be arbitrary and must be set by a user. If the flag is *false*, then the size of the space is specified by the `dim` field Figure 4.

- dim – The number of an objective function parameters. This field should be empty if the flag anyDim is set to true.
- bounds – An array that describes the interval bounds on parameters. Each element of this array stores the left and right boundaries for the function parameters. These boundaries are denoted by *a* and *b* fields respectively. In case the flag anyDim is *true*, the array has only one element. All other elements are assumed to be equal to the first Figure 2.
- globMin – An array storing the coordinates of the global minimum point Figure 4. Note, the array is necessary because the function may have multiple global minima. Each element *x* consists of an array that stores the elements of the coordinate of the global minimum point. In case the flag anyDim is set to true, this array has only one element Figure 2. All other elements are assumed to be equal to the first.
- globMin – the global minimum value of the function.
- comment - an optional field with any additional information about a function.

Figure 3 shows the description of the DropWave function (2). This function has strictly defined dimension (equal to 2) and does not have an input parameter like Rosenbrock function (1).

```
template <class T>
Expr<T> DropWave()
{
        Expr<T> x;
        Expr<T> a = sqr(x[0])+sqr(x[1]);
        Expr<T> b = 1+cos(12*sqrt(a));
        Expr<T> c = 0.5*a + 2;
        return −b/c;
}
```

**Figure 3:** DropWave function.

It should be noted that if a description of a function formula is large, then it is more convenient to break it into several parts. For example, DropWave function (Figure 3) has intermediate variables *a*,*b* and *c*, which are initialized by some part of the formula. This makes it possible to reduce the number of parentheses and improve readability.

```
"DropWave" : {
   "description": "Drop-Wave function",
   "anyDim" : false,
   "dim": 2,
   "bounds": [
         {"a": -5.12, "b": 5.12},
         {"a": -5.12, "b": 5.12}
   ],
   "globMin": [
         { "x" : [0.0, 0.0] }
   ],
   "globMinY": -1.0
}
```

**Figure 4:** The meta description of the DropWave function in JSON format.

## 3 The mathematical expression library

Mathematical expression library was developed in C++ programming language. Though C++ is not that common in scientific computing as, for instance, Python and Fortran we believe it is still important to support it because a significant fraction of researchers and practitioners use this language in their work. C++ is significantly faster than Python and it supports templates and other advanced object-oriented capabilities. Such capabilities are crucial for processing polymorphic expressions and building extensible tools for computing function values and bounds. Another reason why we use C++ is a possibility of integration with existing code and libraries in our department.

The library of mathematical expressions is represented by the template class Expr<T>, where C++ standard real types double and float or the type Interval<> can be used as the template parameter *T*. This enables describing mathematical expressions based on real numbers or intervals.

The Expr<T> class has two constructors. The first constructor has one input parameter of type double. It allows to create a constant expression based on the value of this parameter. Also, this constructor is used to automatically convert real or integer numbers to Expr<T> type. It is worth noting that the automatic conversion of a number into an expression occurs if the number is to the right of an expression of type Expr<T>. If the number is on the left, then the C++ overload operator is used. An example of such an operator (binary “-“) is given in the Figure 5.

The second constructor of Expr<T> with no parameters can describe a vector of variables that is used to imple-

```
/**
* Subtracts the expression from the real number
* @param lv is left number
* @param rv is right expression
* @return expression
*/
template <class T2> friend
Expr<T2> operator-(double lv, const Expr<T2>& rv);
```

**Figure 5:** The operator of overloading the operation of subtraction in the class Expr<T>.

ment a function. A vector of variables with a single name (for example, $x$) whose size corresponds to the dimension of the space. In this case, an index is used to refer to a specific element of this vector. The index can be represented as an integer starting with 0, type `Iterator`, or an arbitrary expression of type `Expr<T>`. An access to individual parameters is provided by an overloaded square brackets operator Figure 6. For example, the vector variable $x$ was created to implement the `Rosenbrock` function Figure 1. Next, we refer to the elements of this vector using the iterator $i$ or the calculated expression $t + 1$. The size of the vector $x$ is determined by the input parameter $n$ of `Rosenbrock` function. The vector variable $x$ is also created in `DropWave` function Figure 3, but the elements of this vector are accessed using integers 0 or 1 since the dimension of the space equals to 2.

```
/**
* Index operator for vector variables
* @param i is integer index
* @return expression
*/
Expr<T> operator[](int i); /**
* Index operator for vector variables
* @param iterator
* @return expression
*/
Expr<T> operator[](const Iterator &iterator);
/**
* Index operator for vector variables
* @param expr is calculated expression
* @return expression
*/
Expr<T> operator[](const Expr<T> &expr);
```

**Figure 6:** The overloaded square brackets operators in the class `Expr<T>`.

Standard mathematical operations such as addition, subtraction, multiplication etc. are implemented using C++ standard operators overloading techniques. To describe various mathematical expressions, the library includes elementary mathematical functions. The library implements trigonometric functions `sin`, `cos`, `tg`, `ctg`, as well as inverse trigonometric functions `acos`, `asin`, `atg`, `actg`. In addition, the exponential function `exp`, the logarithmic functions `ln` and `log`, the function for computing the power `pow`, the absolute value `abs`, the minimum function `min` and the maximum function `max` are supported. The `IfThen` ternary operation is implemented for the organization of conditional logic in mathematical expressions. The functions `LoopSum` and `LoopMul` implement summation and multiplication respectively. To print a mathematical expression of type `Expr<T>`, an overloaded operator << is used.

The `calc` method Figure 7 was implemented in the `Expr<T>` class for calculating the value of a function at a given point. The parameter of the method is an algorithm – an object of the type `Algorithm<T>` that determines how to calculate the value of the function. The constructor of the algorithm is given a point of a box where the value of the function is calculated. This parameter has the type of vector `std::vector<T>`.

```
/**
* calculates expressions according to algorithm
* @param alg is algorithm. It allows to calculate either
* value of function or interval estimation of function.
* @return real number or object of Interval type or object
ValDer<>
* type or object IntervalDer<> type
*/
T calc(const Algorithm<T> & alg);
```

**Figure 7:** The `calc` method is required to calculate the value of a mathematical expression.

The idea of an algorithm in our library follows the concept of the design pattern strategy [9], where the same data (in our case, a mathematical expression) can be processed by different algorithms. Currently, four types of algorithms are implemented: `FuncAlg<T>` for calculating the value of the function, `InterEvalAlg<T>` for calculating the interval estimates of the function, `ValDerAlg<T>` for calculating the gradient of a function and `IntervalDerAlg<T>` for calculation of an interval estimation of the gradient of a function. All these algorithms are inherited from the base class `Algorithm<T>`. Figure 8 shows an example of calculating the value of a function at a given point. The `calc` method returns a scalar value of a function of type *T* for a given point.

```
Expr<double> expr = DropWave<double>();
double result = expr.calc(FuncAlg<double>({0.5, 1.5}));
std::cout << result;
```

**Figure 8:** Calling the calc method to get the value of the `DropWave` function at the given point (0.5, 1.5).

The library of mathematical expressions allows to calculate the interval estimation of a function on a given box. The `calc` method described above should be used for this purpose with `InterEvalAlg<T>` algorithm as a parameter Figure 9. Note that as the template parameter *T* in `std::vector<T>`, the type `Interval<double>` is passed because the vector stores the bounds of the box, and not the specified point as described in the first case. Type `InterEvalAlg<T>` takes double as a parameter of the template *T*, and then, by inheritance, the type `InterEvalAlg<T>` is converted to `Algorithm<Interval<T>>`. The `calc` method returns an interval estimate of a function of type `Interval<T>` (Section IV) over a given box.

```
Expr<Interval<double>> expr = DropWave<Interval<dou
ble>>();
std::vector<Interval<double>> box = {{ 0.5, 1.5}, {-0.5,
0.5}};
Interval<double> result = expr.calc(InterEvalAlg<dou
ble>(box));
std::cout << result;
```

**Figure 9:** Calculation of the interval estimate of `DropWave` function over the box [0.5, 1.5]x[-0.5, 0.5].

To calculate the gradient of a function, first it's necessary to create a mathematical expression by passing the type `ValDer<T>` as a template parameter Figure 10. The `ValDer<T>` type describes the function gradient and its value. Next, the `ValDerAlg<T>` algorithm is passed to the `calc` method. The point `std::vector<T>` in which the function gradient is calculated is passed to the constructor of this algorithm. As a result, method `calc` returns an object of type `ValDer<T>`.

The calculation of the interval estimate of the function gradient is shown in Figure 11. First, it's necessary to create the mathematical expression by passing the type `IntervalDer<T>` as a template parameter. This

```
Expr<ValDer<double>> expr = DropWave<ValDer<dou
ble>>();
ValDer<double> result = expr.calc( ValDerAlg<double>({
0.5, 1.5}));
std::cout << result;
```

**Figure 10:** Calculate the gradient of the `DropWave` function at the point (0.5, 1.5).

type describes the interval estimate of the function gradient and the interval estimation of the function itself. Then the `calc` method must be passed the algorithm `IntervalDerAlg<T>`. The constructor of this algorithm takes a box on which the interval estimation of the function gradient is calculated. A box is described using the type `std::vector<Interval<T>>`. As a result, the `calc` method returns an object of type `IntervalDer<T>`, characterizing the interval enclosure of the gradient.

```
Expr<IntervalDer<double>> expr = DropWave <Interval
Der<double>>();
std::vector<Interval<double>> box = {{ 0.5, 1.5}, {-0.5,
0.5}};
IntervalDer<double> result = expr.calc(IntervalDerAlg
<double>(box));
std::cout << result;
```

**Figure 11:** Calculations of the interval estimation of the gradient of the `DropWave` function on a box with boundaries [0.5, 1.5] and [-0.5, 0.5].

## 4 The Interval arithmetic library

As described in Section III, the library of mathematical expressions can calculate the interval estimations of a function. To achieve this, the algorithm implemented by the class `InterEvalAlg<T>` uses the library that implements the basic interval arithmetic [8].

The `Interval<T>` class is a template class in which the *T* parameter can be any real C++ type. This class is wrapper over a pair of real numbers representing the left and right boundaries of the interval respectively. Using the standard C++ operator overload techniques, the following operations on the intervals are implemented: addition, subtraction, multiplication and division. In addition, an interval estimation of elementary mathematical functions corresponding to library functions of mathematical expressions are also implemented. The Figure 12 shows an example of calculating the interval estimation of the `Ackley` function [1] for two interval variables *x* and *y*.

```
Interval<double> x(0.9, 1.1), y(-0.1, 0.1);
Interval<double> z = -20.0 * exp(-0.2 * sqrt(0.5 * (sqr(x) +
sqr(y)))) - exp(0.5*(cos(2.0 * M_PI * x) + cos(2.0 * M_PI *
y))) + 20.0 + M_E;
std::cout << z;
```

**Figure 12:** Interval estimation of `Ackley` function.

Currently, we have implemented extended version of `Interval<T>` library which allows to work with collection of intervals. Let's consider `1/x` function where `x` is interval [−1, 1]. Our library supports extended interval arithmetic and so outputs union of two intervals [−∞, −1] and [1, +∞]. Notice that existing Boost C++ library [10] doesn't support work with collections of intervals. That is why we have developed our own interval library.

The `IntervalDer<T>` class is implemented similarly to the `ValDer<T>` class, but it differs in that it is a wrapper not over a real number and an array, but over an interval and an array of intervals. The interval is an interval estimation of the function, and the interval array is an interval estimation of the function gradient. Figure 13 shows an example of calculating the interval estimation of the gradient of a function *f* and its interval estimation over a box [19.0, 21.0]x[43.0, 45.0]x[8.0,10.0].

```
IntervalDer<double> a({ 19.0, 21.0 }, { 1.0, 0.0, 0.0 });
IntervalDer<double> v({ 43.0, 45.0 }, { 0.0, 1.0, 0.0 });
IntervalDer<double> h({ 8.0, 10.0 }, { 0.0, 0.0, 1.0 });
IntervalDer<double> rad = (M_PI / 180.0) * a;
IntervalDer<double> t = sqr(v*cos(rad));
IntervalDer<double>   f   =   (t   /   32.0)*(tg(rad)   +
sqrt(sqr(tg(rad)) + 64.0*h / t));
std::cout << f;
```

**Figure 13:** Calculation of the interval estimation of the gradient of the function *f*.

## 5 The test set of functions and the test environment

The test suite contains 150 well known global optimization functions borrowed from [1]. The entire collection is implemented in the form of C++ template functions discussed above Figure 1 and 3. All necessary definitions are located in `testfuncs.h` file, which can be easily added to any C++ application. The `DescFuncReader` class implements reading the metadata from a JSON file whose format was considered earlier Figure 2 and 4. The class has `getdesr` method that returns `descfunc` structure with the benchmark metadata retrieved by a function key. A list of all the keys lies in the `keys.hpp` file in `Keys` structure. An example of using this class will be shown below when describing the environment for testing (Subsection A).

The entire test set has been thoroughly tested and verified. We developed four types of unit tests:

- Test for the equality of the calculated and expected value of a function.
- The test for the belonging of the value of a function to the interval.
- The test for the equality of the found and expected global minimum of a function.
- The test for the equality of the calculated and expected value of the function gradient.

We used Google testing framework (gtest) [11] to develop unit tests. About 600 tests of different types were implemented. All tests are characterized by the use of a dedicated common part and a short call of this common part from the body of a test function. Below we describe these tests in detail.

Figure 14 shows an example of testing the value of the `Rosenbrock` function. The goal of the test is to check the equality of the calculated and expected values at the global minimum point of the function. The `Rosenbrock` function is called in the body of the test function `TestRosenbrock` to create a mathematical expression for this function. Next, the `Test` method of `FuncsTest` class is called, which is common for all unit tests. The parameters of this method are the key (a unique name of a benchmark), a mathematical expression and optional the number of variables parameter. The latter is specified if `anyDim` flag is true (see Figure 2).

The `DescFuncReader` class object is created in the constructor of the `FuncsTest` class. This object is required to read the metadata function. The constructor of this class is passed the path to JSON file. The body of the `Test` function is given in Figure 14. First, `getdesr` method of the `DescFuncReader` class is called to get the metadata for the `Rosenbrock` benchmark. Next, we get the first point of the global minimum `globMinX`. Further, the global minimum value is calculated at that point. Then we get the expected value `globMinY` of the global minimum of the function. At the end of Test function, `ASSERT_NEAR` macro of gtest environment is called to compare the calculated value and the expected value of `globMinY`. These values should not differ more than the maximum permissible difference `EPSILON` equal to 0.001.

```
class FuncsTest : public ::testing::Test {
protected:
    FuncsTest() : dfr(JSONPATH)
    {
    }
    void Test(const std::string& key, const
    Expr<double>& expr, int dim = 1)
    {

            auto desc = dfr.getdesr(key, dim);
            std::vector<double>        globMinX        =
            desc.globMinX[0];
            double globMinY = expr.calc(globMinX,

                      FuncAlg<double>());

            double expected = desc.globMinY;
            double epsilon = EPSILON;
            ASSERT_NEAR(expected,   globMinY,   ep-
            silon);

    }
    DescFuncReader dfr;
};
TEST_F(FuncsTest, TestRosenbrock)
{
    int N = 3;
    auto expr = Rosenbrock<double>(N);
    Test(K.Rosenbrock, expr, N);
}
```

**Figure 14:** Testing the value of Rosenbrock function.

Figure 15 shows an example where it is checked that the particular value belongs to the enclosing interval. The test is organized as follows:

1. The value of a function is calculated at a random point in a box.
2. A new box is created around the generated point with a given length of edges.
3. The interval estimation of a function is calculated for the box obtained at the step 2.
4. Assertions are called to check that the value of a function obtained at a random point at the step 1 belongs to the interval estimation of the function computed at the step 3.

```
void   TestInterval(const   std::string&   key,   const
Expr<double> &exprFunc,
const Expr<Interval<double>> &exprInterval, int custom
Dim = 0)
{

    auto point = getRandomPoint(key, customDim);
    double funcValue = exprFunc.calc(point, FuncAlg
    <double>());
    auto intervals = getIntervals(point);
    auto interval = exprInterval.calc(intervals,
    InterEvalAlg<double>());
    double lowBound = interval.lb();
    double upperBound = interval.rb();
    ASSERT_GE(funcValue, lowBound);
    ASSERT_LE(funcValue, upperBound);
}
TEST_F(IntervalTest, TestIntervalRosenbrock)
{
    int N = 3;
    TestInterval(K.Rosenbrock,
    Rosenbrock<double>(N),

    Rosenbrock<Interval<double>>(N), N);
}
```

**Figure 15:** Testing whether the particular value of the Rosenbrock function belongs to the enclosing interval.

This next type of tests compares the global minimum value obtained with the help of the non-uniform covering method and the global minimum value documented in the literature [12]. The entire set of 150 functions was tested on this type of tests. We used the interval lower bounds for the non-uniform coverings method. The assertion checks whether the global minimum found differs from the expected value not more than the specified accuracy `EPSILON`.

The basis of the test comparing the computable and expected value of the gradient is the calculation of a derivative of a function by an approximate method of finite differences and an exact method based on the automatic differentiation [13]. The test compares the results of calculating the gradient of the function at an arbitrary point on a given box. It works as follows:

1. Calculate the value of the function and its' gradient at a random point of a given box using the `ValDerAlg<T>` algorithm.
2. Calculate the value of the function in the same random point using the `FuncAlg<T> algorithm`.

```
std::vector<double> getRandomPoint(const std::string& key, int customDim = 0)
{
	auto desc = dfr.getdesr(key);
	const int dim = desc.anyDim ? customDim : desc.dim;
	Box<double> box(dim);
	for (int i = 0; i < dim; i++)
	{
		int boundIndex = desc.anyDim ? 0 : i;
		box.mA[i] = desc.bounds[boundIndex].first;
		box.mB[i] = desc.bounds[boundIndex].second;
	}
	RandomPointGenerator<double> rg(box);
	std::vector<double> point(dim, 0.0);
	rg.getPoint(point.data());
	return point;
}
void TestDerivative(const std::string& key, const Expr<double>& exprFunc, const Expr<ValDer<double>>& exprDer, int
customDim=1)
{
	std::vector<double> point = getRandomPoint(key, customDim);
	auto der =	exprDer.calc(ValDerAlg<double>(point));
	double func_val = exprFunc.calc(FuncAlg<double>(point));
	double func_val_by_der = der.value();
	double epsilon = EPSILON;
	ASSERT_NEAR(func_val, func_val_by_der, epsilon);
	auto grad = der.grad();
	for(int i=0; i < point.size(); ++i)
	{
		auto new_point = std::vector<double>(point);
		new_point[i]+=DELTA;
		double new_func_val = exprFunc.calc(FuncAlg<double>(new_point));
		double partial_derivative = (new_func_val - func_val)/DELTA;
		std::cout << "func_val=" << func_val << "new_func_val="
		<< new_func_val << '\n';
		double expected = 100;
		double derivative_relative_value = (partial_derivative/grad[i])*100;
		double persent = PERCENT;
		ASSERT_NEAR(expected, derivative_relative_value, persent);
	}
}
TEST_F(DerivativeTest, TestDerivativeRosenbrock)
{
	int N = 3;
	TestDerivative(K.Rosenbrock, Rosenbrock<double>(N),
	Rosenbrock<ValDer<double>>(N), N);
}
```

**Figure 16:** Test the gradient of the `Rosenbrock` function.

3. The values of the function obtained by the algorithms `ValDerAlg<T>` and `FuncAlg<T>` should not differ more than `EPSILON`.
4. Next for each coordinate of the random point, get a new point by adding `DELTA` to the current coordinate. Calculate the value of the function at this point using the algorithm `FuncAlg<T>`.
5. Calculate the partial derivative by dividing by `DELTA` the difference between the value of the function at the random point and the new point.
6. Compare the value of the partial derivative obtained in step 1 and in step 5. The comparison is performed in relative terms expressed.
7. Return to step 4 until all coordinates are checked.

Figure 16 shows an example of testing the gradient of the `Rosenbrock` function.

## 6 Conclusions

In this paper, we studied the implementation and verification of tests for bound-constrained global optimization. A test suite of 150 functions was developed with the help of this approach. The suite was verified by a basic global optimization solver.

We plan to support the automatic calculation of the second derivatives of the function, as well as their interval estimates as a further development of the libraries described above. The techniques of fast automatic differentiation [14] will be used to achieve these goals. This functionality will allow the evaluation of alternative function estimates, for example, based on Lipschitz constant [15]. It is also planned to extend our approach to multi-objective problems to enable existing methods of deterministic global multi-criteria optimization [16] be employed.

The test suite can be used to compare various methods of global or local optimization. C++ source code of all libraries, as well as a test set of functions can be downloaded from GitHub at https://github.com/alusov/mathexplib.git

**Acknowledgement:** This study was supported by Ministry of Science and Education of Republic of Kazakhstan, project 0115PK00554, Russian Fund for Basic Research, project 17-07-00510 A, Leading Scientific Schools project NSH-8860.2016.1, Project III of the division of mathematics of RAS

## References

[1] Jamil, M., & Yang, X. S. (2013). A literature survey of benchmark functions for global optimisation problems. International Journal of Mathematical Modelling and Numerical Optimisation, 4(2), 150-194.
[2] Nicholas I.M. Gould, Dominique Orban, Philippe L. Toint. A Constrained and Unconstrained Testing Environment. Web: http://www.cuter.rl.ac.uk/
[3] Global Optimization Benchmarks and Adaptive Memory Programming for Global Optimization. Web: http://infinity77.net/global_optimization/genindex.html
[4] Gaviano, M., Kvasov, D. E., Lera, D., & Sergeyev, Y. D. (2003). Algorithm 829: Software for generation of classes of test functions github with known local and global minima for global optimization. ACM Transactions on Mathematical Software (TOMS), 29(4), 469-480
[5] Sergeyev, Y. D., Kvasov, D. E., & Mukhametzhanov, M. S. (2017). Operational zones for comparing metaheuristic and deterministic one-dimensional global optimization algorithms. Mathematics and Computers in Simulation, 141, 96-109.
[6] Kvasov, D. E., & Mukhametzhanov, M. S. (2017). Metaheuristic vs. deterministic global optimization algorithms: The univariate case. Applied Mathematics and Computation. 318, 245-259.
[7] Vahid Beiranvand, Warren Hare, Yves Lucet. (2017) Best practices for comparing optimization algorithms. Optimization and Engineering, 18 (4), pp 815–848.
[8] Hansen, Eldon, and G. William Walster, eds. Global optimization using interval analysis: revised and expanded. Vol. 264. CRC Press, 2003.
[9] Erich Gamma, Ralph Johnson, Richard Helm, John Vlissides Design Patterns Elements of Reusable Object-Oriented Software 2001.
[10] Interval Arithmetic Library. Boost C++ libraries. Web: http://www.boost.org/doc/libs/1_65_1/libs/numeric/interval/doc/interval.htm
[11] Google Test, Google's C++ test framework. Web: https://github.com/google/googletest
[12] Evtushenko, Y. G. (1971). Numerical methods for finding global extrema (case of a non-uniform mesh). USSR Computational Mathematics and Mathematical Physics, 11(6), 38-54.
[13] Kearfott, R. Baker. Rigorous Global Search: Continuous Problems. Nonconvex Optimization and Its Applications (1996)
[14] Evtushenko, Y. G., & Zubov, V. I. (2016). Generalized fast automatic differentiation technique. Computational Mathematics and Mathematical Physics, 56(11), 1819-1833.
[15] R.G. Strongin and Y.D. Sergeyev, Global Optimization with Non-convex Constraints: Sequential and Parallel Algorithms, Springer Science & Business Media, New York, 2013.
[16] Evtushenko, Y. G., & Posypkin, M. A. (2014). A deterministic algorithm for global multi-objective optimization. Optimization Methods and Software, 29(5), 1005-1019.