Решение задачи многомерной минимизации функции

Владимир Руцкий, 3057/2

1 Постановка задачи

Требуется найти с наперёд заданной точностью точку, в которой достигается минимум (локальный) многомерной функции $f(x)$ в некоторой области:

$$\min f(x), \quad x \in \mathbb{R}^n,$$

используя *метод градиентного спуска* и *генетический алгоритм*.

Исходная функция: $f(x) = x_1^3 + 2x_2 + 4\sqrt{2 + x_1^2 + x_2^2}$, заданная на $\mathbb{R}^2$.

2 Исследование применимости методов

2.1 Метод градиентного спуска

Исходная функция непрерывно дифференцируема:

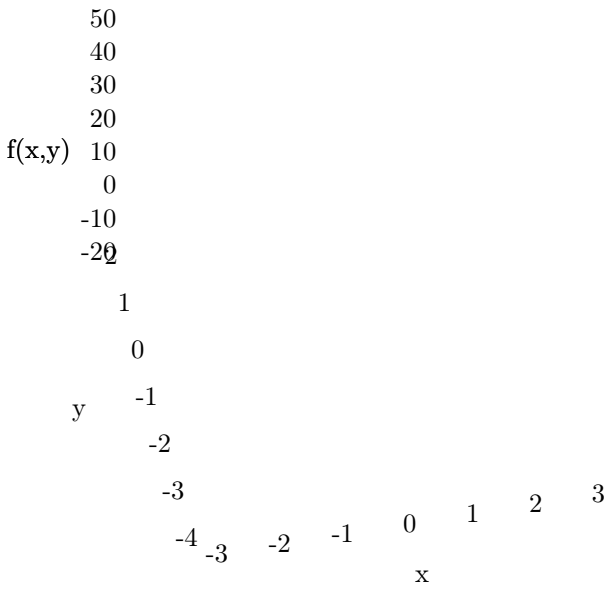
$$\frac{\partial f}{\partial x_1} = 3x_1^2 + \frac{4x_1}{\sqrt{2 + x_1^2 + x_2^2}},$$

$$\frac{\partial f}{\partial x_2} = 2 + \frac{4x_2}{\sqrt{2 + x_1^2 + x_2^2}}.$$

Функция не является ни выпуклой, ни ограниченной на $\mathbb{R}^2$, значит следует искать локальный минимум в некоторой области.

Построив график функции, можно попытаться найти достаточно близкую к локальному минимуму область, на которой функция будет выпуклой.

Рис. 1: График функции $f(x)$



Из графика видно, что минимум достигается близко к точке $(0, -1)$, будем исследовать функцию в окрестности этой точки.

Функция в исследуемой области не имеет особенных точек, ограничена и гладка, что предрасполагает к выполнению условия условия Липшица:

$$\exists R \in \mathbb{R}: \quad ||\nabla f(x) - \nabla f(y)|| \leqslant R||x - y||, \forall x, y \in \mathbb{R}^n,$$

Константа Липшица R была вычислена численно для дискретного набора точек из сетки $[-0.9; 2] \times [-3; 1]$ с шагом 0.01, и оказалась равной примерно 15.8. Следовательно итерационный процесс градиентного спуска будет сходиться.

2.2 Генетический алгоритм

Генетический алгоритм применим для поиска минимума выпуклой функции, а значит его можно использовать на области близкой к локальному минимуму функции, там где функция выпукла.

3 Описание алгоритма

3.1 Метод градиентного спуска

Метод градиентного спуска основывается на том, что для гладкой выпуклой функции градиент функции в точке направлен в сторону увеличения функции (в некоторой окрестности). Используя этот факт строится итерационный процесс приближения рассматриваемых точек области определения к точке минимума.

Выбирается начальное приближение минимума, далее строится последовательность точек, в которой каждая следующая точка выбирается на антиградиенте (луче, противоположном градиенту) в текущей точке:

$$x_{k+1} = x_k - \lambda_k \nabla f(x_k), \quad \lambda_k > 0$$

Шаг, на который "двигается" текущая точка за одну итерацию, выбирается следующим образом:

$$\lambda_k \in (0, q): \quad f(x_k - \lambda_k \nabla f(x_k)) = \min_{0<\lambda<q} f(x_k - \lambda \nabla f(x_k)),$$

значение λ_k ищется *методом золотого сечения.*

Константа q задаёт интервал поиска минимума на антиградиенте.

Условием остановки итерационного процесса является событие, когда следующая точка находится от предыдущей на расстоянии меньшим ε:

$$||x_{k+1} - x_k|| < \varepsilon.$$

3.2 Генетический алгоритм

Суть *генетического алгоритма* для поиска минимума состоит в моделировании процесса биологической эволюции таким образом, что в качестве наиболее приспособленных особей выступают объекты, соответствующие минимуму функции.

Точки области определения функции f выступают в роли особей. Первоначальная популяция выбирается как набор произвольных точек в исследуемой области определения функции.

Каждая итерация работы алгоритма — это смена поколения. Смена поколения определяется трёмя процессами:

- Отбор.

  Из текущей популяции выбираются наиболее приспособленные. В качестве функции приспособленности выступает f: особь (точка) x более приспособлена чем y, если $f(x) < f(y)$.

  Отобранная, более приспособленная часть текущего поколения, перейдёт в следующее поколение.

- Размножение.

  Особи популяции в произвольном порядке скрещиваются друг с другом. Скрещивание особей (точек) x_1, x_2 порождает третью точку $y = \lambda x_1 + (1 - \lambda)x_2$, где λ выбирается произвольным образом из отрезка $[0, 1]$.

Такое скрещивание обеспечивает в некоторой степени передачу потомству признаков родителей: положения в пространстве.

- Мутация.

  В свойства потомков текущей популяции вносятся хаотические изменения, это обеспечивает стабильное разнообразие каждой новой популяции.

  Мутация реализована как смещение особи (точки) на некоторый произвольный вектор: $y_{\text{mutated}} = y + \text{RandomVector}(||x_1 - x_2||)$. Модуль произвольного вектора линейно связан с расстоянием между родителями особи.

В результате новое поколение будет составлено из отобранных особей и мутировавших детей текущего поколения. Количество особей в поколении постоянно, недостающие в результате отбора особи выбираются из потомства.

Условием выхода из алгоритма является событие, что наиболее приспособленная особь (точка) на протяжении нескольких последних поколений не меняется больше чем на ε: $||x_i - x_{i-1}|| < \varepsilon$.

4 Код программы

4.1 Метод градиентного спуска

Исходный код 1: Градиентный спуск

```
1  /*
2   * gradient_descent.hpp
3   * Searching multidimensional function minimum with gradient descent algorithm.
4   * Vladimir Rutsky <altsysrq@gmail.com>
5   * 29.03.2009
6   */
7
8  #ifndef NUMERIC_GRADIENT_DESCENT_HPP
9  #define NUMERIC_GRADIENT_DESCENT_HPP
10
11 #include "numeric_common.hpp"
12
13 #include <boost/assert.hpp>
14 #include <boost/concept/assert.hpp>
15 #include <boost/concept_check.hpp>
16 #include <boost/bind.hpp>
17 #include <boost/function.hpp>
18
19 #include "golden_section_search.hpp"
20 #include "lerp.hpp"
21
22 namespace numeric
23 {
24 namespace gradient_descent
25 {
26   // TODO: Implement all possible states handling.
27   enum gradient_descent_result
28   {
29     gd_close_point = 0,      // Ok, next point founded by one dimension minimisation is
          almost old point.
30     gd_zero_gradient,        // Ok, found point with almost zero gradient.
31     gd_step_too_small,       // Ok, step decreased to value smaller than precision.
32     gd_too_many_iterations, // Debug failure, iterations numer excided predefined number.
33     gd_inf_gradient,         // Failure, in some point function gradient is infinite,
          algorithm interrupted.
34     gd_inf_function,         // Failure, function value is infimum.
35   };
36
37   template< class Func, class FuncGrad, class V, class ConstraintPredicate, class
       PointsOut >
38   inline
39   gradient_descent_result
40     find_min( Func function, FuncGrad functionGrad,
41               V const &startPoint,
```

```
42                typename V::value_type precision,
43                typename V::value_type step,
44                V &result,
45                ConstraintPredicate constraintPred,
46                PointsOut           pointsOut )
47  {
48    // TODO: Now we assume that vector's coordinates and function values are same scalar
          types.
49    // TODO: Assert on correctness of 'pointsOut'.
50    // TODO: On each iteration something is stored in some container.
51
52    BOOST_CONCEPT_ASSERT((ublas::VectorExpressionConcept<V>));
53
54    typedef typename V::value_type             scalar_type;
55    typedef ublas::vector<scalar_type>         vector_type;
56    typedef ublas::scalar_traits<scalar_type> scalar_traits_type;
57
58    BOOST_CONCEPT_ASSERT((boost::UnaryFunction<Func,     scalar_type, vector_type>));
59    BOOST_CONCEPT_ASSERT((boost::UnaryFunction<FuncGrad, vector_type, vector_type>));
60
61    BOOST_ASSERT(precision > 0);
62
63    // Setting current point to start point.
64    vector_type x = startPoint;
65    BOOST_ASSERT(constraintPred(x));
66
67    *pointsOut++ = x;
68
69    size_t iterations = 0;
70    while (true)
71    {
72      // Searching next point in direction opposite to gradient.
73      vector_type const grad = functionGrad(x);
74
75      scalar_type const gradNorm = ublas::norm_2(grad);
76      if (scalar_traits_type::equals(gradNorm, 0))
77      {
78        // Function gradient is almost zero, found minimum.
79        result = x;
80        return gd_zero_gradient;
81      }
82
83      // TODO: Use normal constants.
84      scalar_type const inf = 1e8;
85      // FIXME: Very tricky!
86      scalar_type const gradInfNorm = ublas::norm_inf(grad);
87      //std::cout << "grad=" << grad << std::endl;
88      //std::cout << "infNorm(grad)=" << gradInfNorm << std::endl;
89      if (gradInfNorm >= inf / 2)
90      {
91        // Infinite gradient. Thats bad. Really bad.
92        result = x;
93        std::cout << "GD exit by inf gradient: " << x << "; " << grad << std::endl; //
              debug
94        return gd_inf_gradient;
95      }
96
97      vector_type const dir = -grad / gradNorm;
98      BOOST_ASSERT(scalar_traits_type::equals(ublas::norm_2(dir), 1));
99
100     vector_type nextX;
101     do
102     {
103       vector_type const s0 = x;
104       vector_type const s1 = s0 + dir * step;
105
106       //std::cout << "golden start" << std::endl; // debug
107       typedef boost::function<scalar_type ( scalar_type )> function_bind_type;
108       function_bind_type functionBind =
109           boost::bind<scalar_type>(function, boost::bind<vector_type>(Lerp<scalar_type,
                  vector_type>(0.0, 1.0, s0, s1), _1));
```

```
110         scalar_type const section =
111             golden_section::find_min<function_bind_type, scalar_type>(functionBind, 0.0,
                    1.0, precision / step);
112         BOOST_ASSERT(0 <= section && section <= 1);
113         //std::cout << "golden end" << std::endl; // debug
114
115         nextX = s0 + dir * step * section;
116         if (ublas::norm_2(x - nextX) < precision)
117         {
118           // Next point is equal to current (with precision and constraint), seems found
                minimum.
119           std::cout << "GD exit by constraint: " << s0 << "; " << (s1 - s0) << " ; " <<
                grad << std::endl; // debug
120           result = x;
121           return gd_close_point;
122         }
123
124         //std::cout << "nextX=" << nextX << std::endl; // debug
125
126         // TODO: Use normal constants.
127         // FIXME: Tricky place!
128         if (abs(function(nextX)) >= inf / 2)
129         {
130           result = x;
131           return gd_inf_function;
132         }
133
134         // Decreasing search step if point is not admissible.
135         if (!constraintPred(nextX)) // TODO: Do not rerund constraint predicate for same
                point.
136           step /= 2.;
137
138         if (step <= precision)
139         {
140           result = x;
141           std::cout << "gd_step_too_small: " << x << ", step=" << step << ", prec=" <<
                precision << std::endl; // debug
142           return gd_step_too_small;
143         }
144       } while (!constraintPred(nextX));
145
146       // Moving to next point.
147       x = nextX;
148       *pointsOut++ = x;
149
150       ++iterations;
151
152       // debug
153       if (iterations >= 1000)
154       {
155         std::cerr << "gradient_descent::find_min(): Too many iterations!\n";
156         break;
157       }
158       // end of debug
159     }
160
161     result = x;
162     return gd_too_many_iterations;
163   }
164 } // End of namespace 'gradient_descent'.
165 } // End of namespace 'numeric'.
166
167 #endif // NUMERIC_GRADIENT_DESCENT_HPP
```

4.2 Генетический алгоритм

Исходный код 2: Генетический алгоритм

```
1 /*
```

```
2  * genetic.hpp
3  * Genetics algorithms.
4  * Vladimir Rutsky <altsysrq@gmail.com>
5  * 31.03.2009
6  */
7
8 #ifndef NUMERIC_GENETIC_HPP
9 #define NUMERIC_GENETIC_HPP
10
11 #include "numeric_common.hpp"
12
13 #include <vector>
14 #include <deque>
15
16 #include <boost/assert.hpp>
17 #include <boost/concept/assert.hpp>
18 #include <boost/concept_check.hpp>
19 #include <boost/bind.hpp>
20 #include <boost/random/linear_congruential.hpp>
21 #include <boost/random/uniform_real.hpp>
22 #include <boost/random/uniform_int.hpp>
23 #include <boost/random/variate_generator.hpp>
24 #include <boost/optional.hpp>
25 #include <boost/next_prior.hpp>
26
27 namespace numeric
28 {
29 namespace genetic
30 {
31   typedef boost::minstd_rand base_generator_type; // TODO
32
33   template< class V >
34   struct ParallelepipedonUniformGenerator
35   {
36   private:
37     BOOST_CONCEPT_ASSERT((ublas::VectorExpressionConcept<V>));
38
39   public:
40     typedef V vector_type;
41
42   public:
43     ParallelepipedonUniformGenerator( vector_type const &a, vector_type const &b )
44       : a_(a)
45       , b_(b)
46       , rndGenerator_(42u)
47     {
48       BOOST_ASSERT(a_.size() == b_.size());
49       BOOST_ASSERT(a_.size() > 0);
50     }
51
52     vector_type operator()() const
53     {
54       vector_type v(a_.size());
55
56       for (size_t r = 0; r < v.size(); ++r)
57       {
58         BOOST_ASSERT(a_(r) <= b_(r));
59
60         // TODO: Optimize.
61         boost::uniform_real<> uni_dist(a_(r), b_(r));
62         boost::variate_generator<base_generator_type &, boost::uniform_real<> > uni(
               rndGenerator_, uni_dist);
63
64         v(r) = uni();
65
66         BOOST_ASSERT(a_(r) <= v(r) && v(r) <= b_(r));
67       }
68
69       return v;
70     }
71
```

```
72    private:
73      vector_type const a_, b_;
74
75      mutable base_generator_type rndGenerator_;
76    };
77
78    struct LCCrossOver
79    {
80      LCCrossOver()
81        : rndGenerator_(30u)
82      {
83      }
84
85      template< class V >
86      V operator()( V const &x, V const &y ) const
87      {
88        // TODO: Optimize.
89        boost::uniform_real<> uni_dist(0.0, 1.0);
90        boost::variate_generator<base_generator_type &, boost::uniform_real<> > uni(
              rndGenerator_, uni_dist);
91
92        double const lambda = uni();
93
94        return x * lambda + (1 - lambda) * y;
95      }
96
97    private:
98      mutable base_generator_type rndGenerator_;
99    };
100
101   template< class Scalar >
102   struct ParallelepipedonMutation
103   {
104     typedef Scalar scalar_type;
105
106     template< class OffsetFwdIt >
107     ParallelepipedonMutation( OffsetFwdIt first, OffsetFwdIt beyond )
108       : rndGenerator_(30u)
109     {
110       deviations_.assign(first, beyond);
111     }
112
113     template< class V, class S >
114     V operator()( V const &x, S const scale ) const
115     {
116       BOOST_ASSERT(deviations_.size() == x.size());
117
118       V result(deviations_.size());
119
120       // TODO: Optimize.
121       for (size_t r = 0; r < deviations_.size(); ++r)
122       {
123         boost::uniform_real<> uni_dist(0.0, 1.0);
124         boost::variate_generator<base_generator_type &, boost::uniform_real<> > uni(
              rndGenerator_, uni_dist);
125
126         double const lambda = uni();
127
128         result(r) = x(r) + deviations_[r] * lambda * scale;
129       }
130
131       return result;
132     }
133
134   private:
135     std::vector<scalar_type>    deviations_;
136     mutable base_generator_type rndGenerator_;
137   };
138
139   // TODO: Documentation.
```

```
140  template< class Generator, class Crossover, class Mutation, class V, class Func, class
         FuncScalar, class PointsVecsOut >
141  V vectorSpaceGeneticSearch( Generator generator, Crossover crossover, Mutation mutation
         , Func fitness,
142                              size_t nIndividuals, double liveRate,
143                              typename V::value_type precision, size_t nPrecisionSelect,
144                              PointsVecsOut selectedPointsVecsOut, PointsVecsOut
                                     notSelectedPointsVecsOut )
145  {
146    BOOST_CONCEPT_ASSERT((ublas::VectorExpressionConcept<V>));
147    BOOST_CONCEPT_ASSERT((boost::UnaryFunction<Func, FuncScalar, V>));
148    // TODO: Concept asserts for Generator and Crossover.
149
150    typedef FuncScalar                 function_scalar_type;
151    typedef V                          vector_type;
152    typedef typename V::value_type     value_type;
153    typedef std::vector<vector_type> individuals_vector_type;
154
155    BOOST_ASSERT(0 <= liveRate && liveRate <= 1);
156    BOOST_ASSERT(nPrecisionSelect > 0);
157
158    individuals_vector_type population;
159    population.reserve(nIndividuals);
160    individuals_vector_type nextPopulation;
161    nextPopulation.reserve(nIndividuals);
162
163    base_generator_type rndGenerator(57u);
164
165    typedef std::deque<vector_type> fitted_individuals_deque_type;
166    fitted_individuals_deque_type fittedIndividuals;
167
168    // Spawning initial population.
169    for (size_t i = 0; i < nIndividuals; ++i)
170      population.push_back(generator());
171
172    size_t iterations = 0;
173    while (true)
174    {
175      // Sorting current population.
176      std::sort(population.begin(), population.end(),
177                boost::bind(std::less<function_scalar_type>(), boost::bind(fitness, _1),
                       boost::bind(fitness, _2)));
178      size_t const nSelected = liveRate * nIndividuals;
179
180      BOOST_ASSERT(nSelected != 0 && nSelected != nIndividuals);
181
182      {
183        // Outputting current population.
184        individuals_vector_type selected;
185        selected.reserve(nSelected);
186        std::copy(population.begin(), boost::next(population.begin(), nSelected), std::
               back_inserter(selected));
187        *selectedPointsVecsOut++ = selected;
188
189        individuals_vector_type notSelected;
190        notSelected.reserve(nIndividuals - nSelected);
191        std::copy(boost::next(population.begin(), nSelected), boost::next(population.
               begin(), nIndividuals),
192                  std::back_inserter(notSelected));
193        *notSelectedPointsVecsOut++ = notSelected;
194      }
195
196      fittedIndividuals.push_front(population[0]);
197
198      BOOST_ASSERT(nPrecisionSelect > 0);
199      while (fittedIndividuals.size() > nPrecisionSelect)
200        fittedIndividuals.pop_back();
201
202      if (fittedIndividuals.size() == nPrecisionSelect)
203      {
204        // Checking is most fitted individual is changing in range of precision.
```

```
205
206         vector_type const lastMostFittedIndividual = fittedIndividuals.front();
207         bool satisfy(true);
208         for (typename fitted_individuals_deque_type::const_iterator it = boost::next(
                fittedIndividuals.begin()); it != fittedIndividuals.end(); ++it)
209         {
210           value_type const dist = ublas::norm_2(lastMostFittedIndividual - *it);
211           if (dist >= precision)
212           {
213             satisfy = false;
214             break;
215           }
216         }
217
218
219         if (satisfy)
220         {
221           // Evolved to population which meets precision requirements.
222           return lastMostFittedIndividual;
223         }
224       }
225
226       {
227         // Generating next population.
228
229         nextPopulation.resize(0);
230
231         // Copying good individuals.
232         std::copy(population.begin(), boost::next(population.begin(), nSelected),
233                   std::back_inserter(nextPopulation));
234         BOOST_ASSERT(nextPopulation.size() == nSelected);
235
236         // Crossover and mutation.
237         for (size_t i = nSelected; i < nIndividuals; ++i)
238         {
239           // TODO: Optimize.
240           boost::uniform_int<> uni_dist(0, nIndividuals - 1);
241           boost::variate_generator<base_generator_type &, boost::uniform_int<> > uni(
                rndGenerator, uni_dist);
242
243           size_t const xIdx = uni();
244           size_t const yIdx = uni();
245           BOOST_ASSERT(xIdx < population.size());
246           BOOST_ASSERT(yIdx < population.size());
247
248           // Crossover.
249           vector_type const x = population[xIdx], y = population[yIdx];
250           vector_type const child = crossover(x, y);
251
252           // Mutation.
253           vector_type const mutant = mutation(child, ublas::norm_2(x - y)); // TODO:
                Process may be unstable.
254
255           nextPopulation.push_back(mutant);
256         }
257       }
258
259       // Replacing old population.
260       population.swap(nextPopulation);
261
262       // debug, TODO
263       ++iterations;
264       if (iterations >= 1000)
265       {
266         std::cerr << "Too much iterations!\n";
267         break;
268       }
269       // end of debug
270     }
271
272     return population[0];
```

```
273     }
274 } // End of namespace 'genetic'.
275 } // End of namespace 'numeric'.
276
277 #endif // NUMERIC_GENETIC_HPP
```

5 Результаты решения

5.1 Метод градиентного спуска

Результаты решения приведены в таблице 1.

Начальной точкой была выбрана точка (2.5, 2.5) , шаг для поиска минимума методом золотого сечения был равен 0.5 .

Таблица 1: Результаты работы алгоритма градиентного спуска

Точность	Шаги	x	$f(x)$	$f_i(x) - f_{i-1}(x)$	$\nabla f(x)$
1e-03	12	(1.67515e-05, -0.81647076)	4.89897949		(4.10337e-05, 4.744064e-05)
1e-04	12	(5.32455e-06, -0.81647068)	4.89897949	-3.053593e-10	(1.30426e-05, 4.757985e-05)
1e-05	13	(2.39964e-07, -0.81649641)	4.89897949	-6.507568e-10	(5.8779e-07, 3.093111e-07)
1e-06	13	(4.23671e-07, -0.81649656)	4.89897949	1.234568e-13	(1.03778e-06, 3.759286e-08)
1e-07	14	(8.27773e-09, -0.81649657)	4.89897949	-2.202682e-13	(2.02762e-08, 2.704985e-08)
1e-08	15	(2.61515e-09, -0.81649658)	4.89897949	0.000000e+00	(6.40578e-09, 2.714819e-09)

5.2 Генетический алгоритм

Результаты решения приведены в таблице 2.

Популяция состояла из 1000 особей, первоначально расположенных в прямоугольнике $[-0.9; 2] \times [-3; 1]$. 80% особей отбирались и оставались в популяции.

Таблица 2: Результаты работы генетического алгоритма

Точность*	Шаги	x	$f(x)$	$f_i(x) - f_{i-1}(x)$	$\nabla f(x)$
1e-03	51	(-8.89596e-05, -0.81641967)	4.89897950		(-0.000217887, 1.413057e-04)
1e-04	73	(1.91949e-06, -0.81650223)	4.89897949	-1.509204e-08	(4.70178e-06, -1.037411e-05)
1e-05	89	(1.91949e-06, -0.81650223)	4.89897949	0.000000e+00	(4.70178e-06, -1.037411e-05)
1e-06	120	(2.42424e-07, -0.81649652)	4.89897949	-3.372858e-11	(5.93816e-07, 1.107199e-07)
1e-07	120	(2.42424e-07, -0.81649652)	4.89897949	0.000000e+00	(5.93816e-07, 1.107199e-07)
1e-08	120	(2.42424e-07, -0.81649652)	4.89897949	0.000000e+00	(5.93816e-07, 1.107199e-07)

6 Возможные дополнительные исследования

6.1 Метод градиентного спуска

Найденная методом золотого сечения, следующая точка на антиградиете x_{k+1}:

$$\lambda_k \in (0, q): \quad f(x_k - \lambda_k \nabla f(x_k)) = \min_{0<\lambda<q} f(x_k - \lambda \nabla f(x_k)),$$

$$x_{k+1} = x_k - \lambda_k \nabla f(x_k),$$

является минимумом $\psi(v) = f(x_k - v\nabla f(x_k)), v \in (0, q)$. Если λ_k лежит сильно внутри $[0, q]$ (возможен случай, когда λ_k стремится к q, если итерационный шаг недостаточно велик), то из условия

минимальности следует, что $0 = \psi'(\lambda_k) = \frac{(\nabla f(x_{k+1}), \nabla f(x_k))}{||\nabla f(x_k)||}$, т. е. величина проекции градиента f в точке x_{k+1} на линию антиградиента равна нулю. Из этого следует, что, при достаточной длине шага, у каждого отрезка $[x_k, x_{k+1}]$ начало x_k будет перпендикулярно силовой линии, проходящей через x_k, а конец x_{k+1} будет лежать на касательной к силовой линии, проходящей через x_{k+1}.

Данная особенность слабо проявляется на исследуемой функции, т. к. был выбран достаточно малый шаг и большая часть x_{k+1} соответствует краям отрезков антиградиента.

7 Обоснование достоверности полученного результата

7.1 Метод градиентного спуска

Градиент исходной функции (его норма), в полученном с точностью ε решении, обращается в ноль с некоторой точностью, пропорциональной ε, что является достаточным условием для минимума выпуклой функции.

7.2 Генетический алгоритм

Полученные результаты работы генетического алгоритма близки к результатам, полученным методом градиентного спуска, но менее точны, ввиду большого числа случайных факторов, использовавшихся в алгоритме.