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:

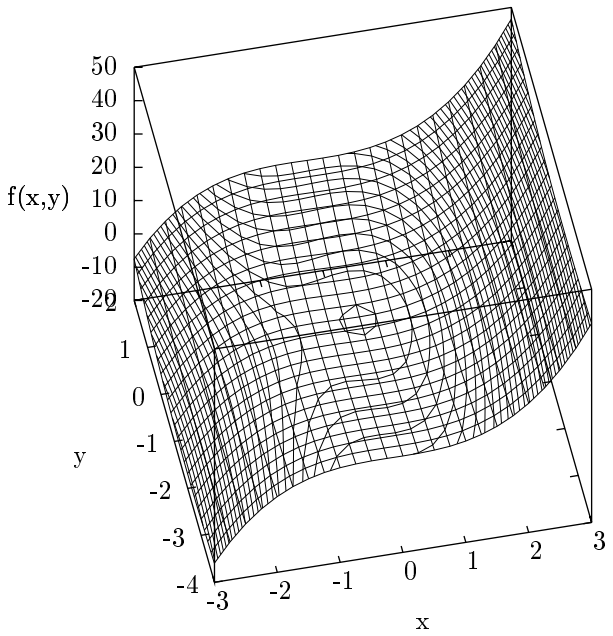
$$\frac{\partial f}{\partial x_1} = 3x_1^2 + \frac{4x_1}{\sqrt{2 + x_1^2 + x_2^2}},$$

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

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.

Из того, что функция в исследуемой области не имеет особенных точек, ограничена и гладка следует, что выполняется условие Липшица:

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

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   template< class Func, class FuncGrad, class V, class PointsOut >
27   inline
28   ublas::vector<typename V::value_type>
29     find_min( Func function, FuncGrad functionGrad,
30               V const &startPoint,
31               typename V::value_type precision,
32               typename V::value_type step,
33               PointsOut pointsOut )
34   {
35     // TODO: Now we assume that vector's coordinates and function values are same scalar
          types.
36     // TODO: Assert on correctness of 'ostr'.
37
38     BOOST_CONCEPT_ASSERT((ublas::VectorExpressionConcept<V>));
39
40     typedef typename V::value_type             scalar_type;
41     typedef ublas::vector<scalar_type>         vector_type;
42     typedef ublas::scalar_traits<scalar_type> scalar_traits_type;
43
44     BOOST_CONCEPT_ASSERT((boost::UnaryFunction<Func,     scalar_type, vector_type>));
45     BOOST_CONCEPT_ASSERT((boost::UnaryFunction<FuncGrad, vector_type, vector_type>));
46
```

```
47      BOOST_ASSERT(precision > 0);
48
49      // Setting current point to start point.
50      vector_type x = startPoint;
51
52      *pointsOut++ = x;
53
54      size_t iterations = 0;
55      while (true)
56      {
57        // Searching next point in direction opposite to gradient.
58        vector_type const grad = functionGrad(x);
59
60        scalar_type const gradNorm = ublas::norm_2(grad);
61        if (scalar_traits_type::equals(gradNorm, 0))
62        {
63          // Function gradient is almost zero, found minimum.
64          return x;
65        }
66
67        vector_type const dir = -grad / gradNorm;
68        BOOST_ASSERT(scalar_traits_type::equals(ublas::norm_2(dir), 1));
69
70        vector_type const s0 = x;
71        vector_type const s1 = s0 + dir * step;
72
73        typedef boost::function<scalar_type ( scalar_type )> function_bind_type;
74        function_bind_type functionBind =
75            boost::bind<scalar_type>(function, boost::bind<vector_type>(Lerp<scalar_type,
                vector_type>(0.0, 1.0, s0, s1), _1));
76        scalar_type const section = golden_section::find_min<function_bind_type,
              scalar_type>(functionBind, 0.0, 1.0, precision);
77        BOOST_ASSERT(0 <= section && section <= 1);
78
79        // debug
80        /*
81        std::cout << "x=";
82        output_vector_coordinates(std::cout, x);
83        std::cout << "f(x0) = " << function(s0 + dir * step * 0) << std::endl;
84        std::cout << "f(x)  = " << function(s0 + dir * step * section) << std::endl;
85        std::cout << "f(x1) = " << function(s0 + dir * step * 1) << std::endl;
86        std::cout << "section=" << section << std::endl; // debug
87        */
88        // end of debug
89
90        vector_type const nextX = s0 + dir * step * section;
91        if (ublas::norm_2(x - nextX) < precision)
92        {
93          // Next point is equal to current (with precision), seems found minimum.
94          return x;
95        }
96
97        // Moving to next point.
98        x = nextX;
99        *pointsOut++ = x;
100
101       ++iterations;
102
103       // debug
104       if (iterations >= 100)
105       {
106         std::cerr << "Too many iterations!\n";
107         break;
108       }
109       // end of debug
110     }
111
112     return x;
113   }
114 } // End of namespace 'gradient_descent'.
115 } // End of namespace 'numeric'.
```

```
116
117 #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	(4.61855e-06, -0.81648024)	4.89897949		(1.13132e-05, 3.001177e-05)
1e-04	12	(-8.49295e-07, -0.81646860)	4.89897949	4.488383e-10	(-2.08035e-06, 5.140646e-05)
1e-05	13	(1.20244e-07, -0.81649645)	4.89897949	-7.200684e-10	(2.94537e-07, 2.496002e-07)
1e-06	13	(4.11864e-07, -0.81649654)	4.89897949	1.758593e-13	(1.00886e-06, 7.665019e-08)
1e-07	14	(1.41643e-08, -0.81649657)	4.89897949	-2.104983e-13	(3.46953e-08, 1.945369e-08)
1e-08	17	(3.07062e-09, -0.81649657)	4.89897949	0.000000e+00	(7.52146e-09, 1.664443e-08)

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 Обоснование достоверности полученного результата

6.1 Метод градиентного спуска

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

6.2 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.