

Решение задачи многомерной минимизации функции

Владимир Руцкий, 3057/2

7 апреля 2009 г.

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

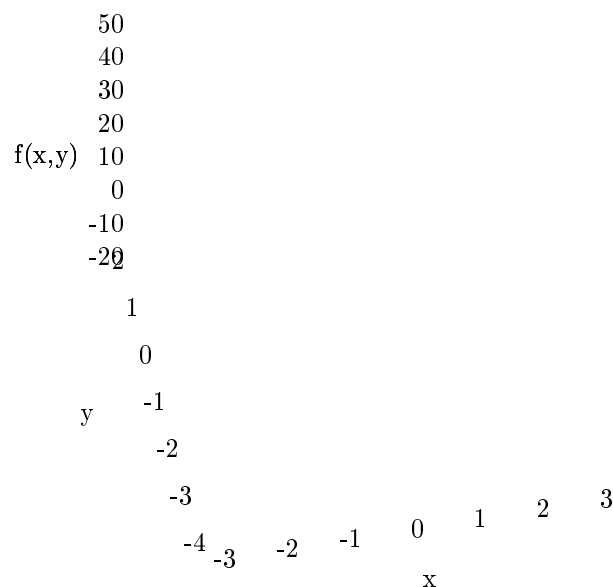
Исходная функция непрерывно дифференцируема:

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

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 особых точек, ограничена и гладка следует, что выполняется условие Липшица:

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

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 тремя 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 поколения перейдёт в следующее поколение.

- Размножение.

Особь 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
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,
76             vector_type>(0.0, 1.0, s0, s1), _1));
77     scalar_type const section = golden_section::find_min<function_bind_type,
78         scalar_type>(functionBind, 0.0, 1.0, precision);
79     BOOST_ASSERT(0 <= section && section <= 1);
80
81     // debug
82     /*
83     std::cout << "x=";
84     output_vector_coordinates(std::cout, x);
85     std::cout << "f(x0) = " << function(s0 + dir * step * 0) << std::endl;
86     std::cout << "f(x) = " << function(s0 + dir * step * section) << std::endl;
87     std::cout << "f(x1) = " << function(s0 + dir * step * 1) << std::endl;
88     std::cout << "section=" << section << std::endl; // debug
89     */
90     // end of debug
91
92     vector_type const nextX = s0 + dir * step * section;
93     if (ublas::norm_2(x - nextX) < precision)
94     {
95         // Next point is equal to current (with precision), seems found minimum.
96         return x;
97     }
98
99     // Moving to next point.
100     x = nextX;
101     *pointsOut++ = x;
102
103     ++iterations;
104
105     // debug
106     if (iterations >= 100)
107     {
108         std::cerr << "Too many iterations!\n";
109         break;
110     }
111     // end of debug
112 }
113
114 return x;
115 }
116 } // End of namespace 'gradient_descent'.
117 } // 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/variante_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::variante_generator<base_generator_type &, boost::uniform_real<> > uni(
63                         rndGenerator_, uni_dist);

```

```

63         v(r) = uni();
64
65         BOOST_ASSERT(a_(r) <= v(r) && v(r) <= b_(r));
66     }
67
68     return v;
69 }
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(
91             rndGenerator_, uni_dist);
92
93         double const lambda = uni();
94
95         return x * lambda + (1 - lambda) * y;
96     }
97
98 private:
99     mutable base_generator_type rndGenerator_;
100 };
101
102 template< class Scalar >
103 struct ParallelepipedonMutation
104 {
105     typedef Scalar scalar_type;
106
107     template< class OffsetFwdIt >
108     ParallelepipedonMutation( OffsetFwdIt first, OffsetFwdIt beyond )
109         : rndGenerator_(30u)
110     {
111         deviations_.assign(first, beyond);
112     }
113
114     template< class V, class S >
115     V operator()( V const &x, S const scale ) const
116     {
117         BOOST_ASSERT(deviations_.size() == x.size());
118
119         V result(deviations_.size());
120
121         // TODO: Optimize.
122         for (size_t r = 0; r < deviations_.size(); ++r)
123         {
124             boost::uniform_real<> uni_dist(0.0, 1.0);
125             boost::variate_generator<base_generator_type &, boost::uniform_real<> > uni(
126                 rndGenerator_, uni_dist);
127
128             double const lambda = uni();
129
130             result(r) = x(r) + deviations_[r] * lambda * scale;
131         }
132
133         return result;
134     }

```

```

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

```



```

197 BOOST_ASSERT(nPrecisionSelect > 0);
198 while (fittedIndividuals.size() > nPrecisionSelect)
199     fittedIndividuals.pop_back();
200
201 if (fittedIndividuals.size() == nPrecisionSelect)
202 {
203     // Checking is most fitted individual is changing in range of precision.
204
205     vector_type const lastMostFittedIndividual = fittedIndividuals.front();
206     bool satisfy(true);
207     for (typename fitted_individuals_deque_type::const_iterator it = boost::next(
208         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     if (satisfy)
219     {
220         // Evolved to population which meets precision requirements.
221         return lastMostFittedIndividual;
222     }
223 }
224
225 {
226     // Generating next population.
227
228     nextPopulation.resize(0);
229
230     // Copying good individuals.
231     std::copy(population.begin(), boost::next(population.begin(), nSelected),
232         std::back_inserter(nextPopulation));
233     BOOST_ASSERT(nextPopulation.size() == nSelected);
234
235     // Crossover and mutation.
236     for (size_t i = nSelected; i < nIndividuals; ++i)
237     {
238         // TODO: Optimize.
239         boost::uniform_int<> uni_dist(0, nIndividuals - 1);
240         boost::variate_generator<base_generator_type &, boost::uniform_int<> > uni(
241             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:
254             Process may be unstable.
255
256         nextPopulation.push_back(mutant);
257     }
258 }
259
260 // Replacing old population.
261 population.swap(nextPopulation);
262
263 // debug, TODO
264 ++iterations;
265 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*	Количество шагов	x	$f(x)$	$f_i(x) - f_{i-1}(x)$
1e-03	12	(4.61855e-06, -0.81648024)	4.89897949	
1e-04	12	(-8.49295e-07, -0.81646860)	4.89897949	4.488383e-10
1e-05	13	(1.20244e-07, -0.81649645)	4.89897949	-7.200684e-10
1e-06	13	(4.11864e-07, -0.81649654)	4.89897949	1.758593e-13
1e-07	14	(1.41643e-08, -0.81649657)	4.89897949	-2.104983e-13
1e-08	17	(3.07062e-09, -0.81649657)	4.89897949	0.000000e+00

5.2 Генетический алгоритм

Таблица 2: Результаты работы генетического алгоритма

Точность	Количество шагов	x	$f(x)$	$f_i(x) - f_{i-1}(x)$
1e-03	51	(-8.89596e-05, -0.81641967)	4.89897950	
1e-04	73	(1.91949e-06, -0.81650223)	4.89897949	-1.509204e-08
1e-05	89	(1.91949e-06, -0.81650223)	4.89897949	0.000000e+00
1e-06	120	(2.42424e-07, -0.81649652)	4.89897949	-3.372858e-11
1e-07	120	(2.42424e-07, -0.81649652)	4.89897949	0.000000e+00
1e-08	120	(2.42424e-07, -0.81649652)	4.89897949	0.000000e+00

6 Возможные дополнительные исследования

6.1 Метод градиентного спуска

6.2 Генетический алгоритм

7 Обоснование достоверности полученного результата

7.1 Метод градиентного спуска

7.2 Генетический алгоритм