Код программы

Исходный код 1: Барьерный метод

```
1  /*
2   * barrier_method.hpp
3   * Constrained minimization using barrier method.
4   * Vladimir Rutsky <altsysrq@gmail.com>
5   * 29.03.2009
6   */
7
8  #ifndef NUMERIC_BARRIER_METHOD_HPP
9  #define NUMERIC_BARRIER_METHOD_HPP
10
11 #include "numeric_common.hpp"
12
13 #include <vector>
14
15 #include <boost/assert.hpp>
16 #include <boost/concept/assert.hpp>
17 #include <boost/concept_check.hpp>
18 #include <boost/bind.hpp>
19 #include <boost/lambda/lambda.hpp>
20 #include <boost/function.hpp>
21
22 #include "gradient_descent.hpp"
23
24 namespace numeric
25 {
26 namespace barrier_method
27 {
28   namespace
29   {
30     // TODO: Use boost::lambda instead.
31     // f(x) + mu * Summ(-1 / g_i(x))
32     template< class S >
33     struct AdditionalFunction
34     {
35     public:
36       typedef S                                                    scalar_type;
37       typedef vector<scalar_type>                                  vector_type;
38
39     private:
40       typedef boost::function<scalar_type( vector_type )>          function_type;
41       typedef std::vector<function_type>
              limit_functions_vec_type;
42
43     public:
44       template< class Func, class LimitFuncIterator >
45       AdditionalFunction( Func func,
46                           LimitFuncIterator limitFuncBegin, LimitFuncIterator
                              limitFuncEnd )
47         : function_       (func)
48         , limitFunctions_(limitFuncBegin, limitFuncEnd)
49       {
50         // TODO: Assertions on input types.
51       }
52
53       scalar_type operator()( scalar_type mu, vector_type const &x )
54       {
55         scalar_type result(0.0);
56
57         result += function_(x);
58         for (size_t i = 0; i < limitFunctions_.size(); ++i)
59         {
60           scalar_type const denominator = limitFunctions_[i](x);
61
62           // TODO: Use normal constants.
63           scalar_type const eps = 1e-15; // FIXME
64           scalar_type const inf = 1e+8;
65           if (abs(denominator) < eps)
```

```
66            {
67              // Division by zero.
68              // TODO: Break loop and leave value infinite.
69              result = inf;
70            }
71            else
72            {
73              result += -mu / denominator;
74            }
75          }
76
77          return result;
78        }
79
80      private:
81        function_type            function_;
82        limit_functions_vec_type limitFunctions_;
83      };
84
85      // TODO: Use boost::lambda instead.
86      // f(x) + mu * Summ(-1 / g_i(x))
87      template< class S >
88      struct AdditionalFunctionGradient
89      {
90      public:
91        typedef S                                                    scalar_type;
92        typedef vector<scalar_type>                                  vector_type;
93
94      private:
95        typedef scalar_vector<scalar_type>                           scalar_vector_type
              ;
96        typedef boost::function<scalar_type( vector_type )>          function_type;
97        typedef boost::function<vector_type( vector_type )>          function_grad_type
              ;
98        typedef std::vector<function_type>
              limit_functions_vec_type;
99        typedef std::vector<function_grad_type>
              limit_functions_grads_vec_type;
100
101     public:
102       template< class FuncGrad, class LimitFuncIterator, class LimitFuncGradIterator >
103       AdditionalFunctionGradient( FuncGrad funcGrad,
104                                   LimitFuncIterator     limitFuncBegin,
                                        LimitFuncIterator     limitFuncEnd,
105                                   LimitFuncGradIterator limitFuncGradBegin,
                                        LimitFuncGradIterator limitFuncGradEnd )
106         : functionGrad_        (funcGrad)
107         , limitFunctions_      (limitFuncBegin,     limitFuncEnd)
108         , limitFunctionsGrads_(limitFuncGradBegin, limitFuncGradEnd)
109       {
110         // TODO: Assertions on input types.
111         BOOST_ASSERT(limitFunctions_.size() == limitFunctionsGrads_.size());
112       }
113
114       vector_type operator()( scalar_type mu, vector_type const &x )
115       {
116         vector_type result = functionGrad_(x);
117
118         for (size_t i = 0; i < limitFunctions_.size(); ++i)
119         {
120           scalar_type const gx     = limitFunctions_     [i](x);
121           vector_type const gGradx = limitFunctionsGrads_[i](x);
122
123           // TODO: Use normal constants.
124           scalar_type const eps = 1e-30; // FIXME!
125           scalar_type const inf = 1e+8;
126           if (abs(sqr(gx)) < eps)
127           {
128             // Division by zero.
129             // TODO: Break loop and leave value infinite.
130             return scalar_vector_type(x.size(), inf);
```

```
131           }
132           else
133           {
134             result = result + (mu / sqr(gx)) * gGradx;
135           }
136         }
137
138         return result;
139       }
140
141     private:
142       function_grad_type              functionGrad_;
143       limit_functions_vec_type        limitFunctions_;
144       limit_functions_grads_vec_type limitFunctionsGrads_;
145     };
146
147     // TODO: Use boost::lambda instead.
148     template< class S >
149     struct ConstraintPredicate
150     {
151     public:
152       typedef S                                                    scalar_type;
153       typedef vector<scalar_type>                                  vector_type;
154
155     private:
156       typedef boost::function<scalar_type( vector_type )>          function_type;
157       typedef std::vector<function_type>
              limit_functions_vec_type;
158
159     public:
160       template< class LimitFuncIterator >
161       ConstraintPredicate( LimitFuncIterator limitFuncBegin, LimitFuncIterator
              limitFuncEnd )
162         : limitFunctions_(limitFuncBegin, limitFuncEnd)
163       {
164         // TODO: Assertions on input types.
165       }
166
167       bool operator()( vector_type const &x )
168       {
169         for (size_t i = 0; i < limitFunctions_.size(); ++i)
170           if (limitFunctions_[i](x) > 0)
171             return false;
172
173         return true;
174       }
175
176     private:
177       limit_functions_vec_type limitFunctions_;
178     };
179   } // End of anonymous namespace
180
181   template< class S >
182   struct PointDebugInfo
183   {
184     typedef S                    scalar_type;
185     typedef vector<scalar_type> vector_type;
186
187     PointDebugInfo()
188     {}
189
190     PointDebugInfo( vector_type const &newx, scalar_type newmu, scalar_type newfx,
            scalar_type newBx )
191       : x (newx)
192       , mu(newmu)
193       , fx(newfx)
194       , Bx(newBx)
195     {}
196
197     vector_type x;
198     scalar_type mu;
```

```
199       scalar_type fx;
200       scalar_type Bx;
201     };
202
203     // TODO: Habdle more end cases, not all problems input have solutions.
204     template< class Func, class FuncGrad,
205               class S,
206               class LimitFuncIterator, class LimitFuncGradIterator,
207               class PointsOut >
208     inline
209     vector<S>
210       find_min( Func function, FuncGrad functionGrad,
211                 LimitFuncIterator     gBegin,     LimitFuncIterator     gEnd,
212                 LimitFuncGradIterator gGradBegin, LimitFuncGradIterator gGradEnd,
213                 vector<S> const &startPoint,
214                 S startMu, S beta,
215                 S epsilon,
216                 S gradientDescentPrecision, S gradientDescentStep,
217                 PointsOut pointsOut )
218     {
219       typedef S                                   scalar_type;
220       typedef ublas::vector       <scalar_type> vector_type;
221       typedef ublas::scalar_traits<scalar_type> scalar_traits_type;
222
223       // TODO: Check for iterators concept assert.
224       // Note: Input should be accurate so, that start point must be admissible not only
              for input function but for
225       //   additional function too.
226
227       typedef typename LimitFuncIterator::value_type     limit_func_type;
228       typedef typename LimitFuncGradIterator::value_type limit_func_grad_type;
229
230       BOOST_CONCEPT_ASSERT((boost::UnaryFunction<Func,                 scalar_type,
              vector_type>));
231       BOOST_CONCEPT_ASSERT((boost::UnaryFunction<FuncGrad,             vector_type,
              vector_type>));
232       BOOST_CONCEPT_ASSERT((boost::UnaryFunction<limit_func_type,      scalar_type,
              vector_type>));
233       BOOST_CONCEPT_ASSERT((boost::UnaryFunction<limit_func_grad_type, vector_type,
              vector_type>));
234
235       BOOST_ASSERT(epsilon > 0);
236
237       std::vector<limit_func_type>      g    (gBegin,     gEnd);
238       std::vector<limit_func_grad_type> gGrad(gGradBegin, gGradEnd);
239
240       BOOST_ASSERT(g.size() == gGrad.size());
241       BOOST_ASSERT(beta > 0 && beta < 1);
242
243       // Building additional function and it's gradient.
244       typedef boost::function<scalar_type( vector_type )> function_type;
245       typedef boost::function<vector_type( vector_type )> function_gradient_type;
246       typedef AdditionalFunction        <scalar_type>     additional_function_type;
247       typedef AdditionalFunctionGradient<scalar_type>     additional_function_gradient_type
              ;
248       typedef ConstraintPredicate<scalar_type>            constraint_predicate_type;
249       typedef PointDebugInfo<scalar_type>                 points_debug_info_type;
250
251       additional_function_type          additionalFunc    (function,     gBegin, gEnd);
252       additional_function_gradient_type additionalFuncGrad(functionGrad, gBegin, gEnd,
              gGradBegin, gGradEnd);
253       constraint_predicate_type         constraintPred    (gBegin, gEnd);
254
255       // Initializing
256       vector_type x  = startPoint;
257       scalar_type mu = startMu;
258
259       BOOST_ASSERT(constraintPred(x));
260
261       mu /= beta;
```

```
262     points_debug_info_type pdi(x, mu, function(x), (additionalFunc(mu, x) - function(x))
            / mu);
263     mu *= beta;
264     *pointsOut++ = pdi;
265
266     size_t iterations = 0;
267     while (true)
268     {
269       // Additional function: f(x) + mu * Summ(-1 / g_i(x))
270
271       function_type           currFunc     = boost::bind<scalar_type>(additionalFunc,
              mu, _1);
272       function_gradient_type currFuncGrad = boost::bind<vector_type>(additionalFuncGrad,
              mu, _1);
273
274       // Solving additional unconstrained problem.
275       vector_type newx;
276       gradient_descent::gradient_descent_result const result =
277           gradient_descent::find_min
278             <function_type, function_gradient_type, vector_type>
279               (currFunc, currFuncGrad,
280                x,
281                gradientDescentPrecision, gradientDescentStep,
282                newx,
283                constraintPred, DummyOutputIterator());
284       BOOST_ASSERT(result == result); // TODO: Handle result states.
285
286       // debug
287       std::cout << iterations << ": " << newx << std::endl;
288       // end of debug
289
290       scalar_type const muBx = currFunc(newx) - function(newx);
291       scalar_type const Bx = muBx / mu;
292       points_debug_info_type pdi(newx, mu, function(newx), Bx);
293       *pointsOut++ = pdi;
294
295       // mu_k * B(x_k+1) < epsilon
296       BOOST_ASSERT(muBx >= 0);
297       if (muBx < epsilon)
298       {
299         // Required precision reached.
300         return newx;
301       }
302       else
303       {
304         // Moving to next point.
305         x = newx;
306         mu *= beta;
307       }
308
309       ++iterations;
310
311       // debug
312       if (iterations >= 100)
313       {
314         std::cerr << "barrier_method::find_min(): Too many iterations!\n";
315         break;
316       }
317       // end of debug
318     }
319
320     return x;
321   }
322 } // End of namespace 'barrier_method'.
323 } // End of namespace 'numeric'.
324
325 #endif // NUMERIC_BARRIER_METHOD_HPP
```

Таблица 1: Детальная работа алгоритма при точности 10^{-3}

k	x_k	$f(x_k)$	μ_k	$B(x_k)$	$\mu_k B(x_k)$	$\theta(x_k)$
1	(-20.00000000, -20.00000000)	1160.00000000	10000000.00000000	0.03225806	322580.64516129	323740.64516129
2	(-52.71337242, -53.52108827)	6598.50895222	1000000.00000000	0.01239536	12395.36144230	18993.87039452
3	(-23.26405214, -24.05945195)	1545.18948678	100000.00000000	0.02740361	2740.36085107	4285.55033784
4	(-9.65735618, -10.42556406)	381.93498853	10000.00000000	0.06226691	622.66912161	1004.60411014
5	(-3.46094033, -4.16915096)	97.32253865	1000.00000000	0.14885592	148.85592121	246.17845986
6	(-0.77877483, -1.36351004)	21.16147848	100.00000000	0.38483508	38.48350805	59.64498653
7	(0.23912614, -0.14916639)	-1.11849837	10.00000000	1.08409873	10.84098728	9.72248891
8	(0.54561393, 0.33171155)	-7.70210458	1.00000000	2.99720197	2.99720197	-4.70490262
9	(0.54561393, 0.33171155)	-7.70210458	0.10000000	2.99720197	0.29972020	-7.40238439
10	(0.66177407, 0.57473345)	-10.44734479	0.01000000	15.12882066	0.15128821	-10.29605659
11	(0.66389643, 0.65542767)	-11.01204170	0.00100000	99.20331113	0.09920331	-10.91283838
12	(0.66579957, 0.66309668)	-11.07978282	0.00010000	313.41967897	0.03134197	-11.04844085
13	(0.66639338, 0.66553611)	-11.10120421	0.00001000	990.82504340	0.00990825	-11.09129596
14	(0.66658034, 0.66630897)	-11.10797821	0.00000100	3132.90451393	0.00313290	-11.10484530
15	(0.66663938, 0.66655353)	-11.11012034	0.00000010	9906.20374429	0.00099062	-11.10912972

Таблица 2: Результаты работы барьерного метода

Точность	Шаги	x	$f(x)$	$f_i(x) - f_{i-1}(x)$	$\nabla f(x)$	$g_1(x)$	$g_2(x)$
1e-01	11	(0.66389643, 0.65542767)	-11.01204170		(-8.672207e+00, -6.689145e+00)	-0.0252482	-0.0167795
1e-02	13	(0.66639338, 0.66553611)	-11.10120421	-8.916251e-02	(-8.667213e+00, -6.668928e+00)	-0.0025344	-0.00167714
1e-03	15	(0.66663938, 0.66655353)	-11.11012034	-8.916132e-03	(-8.666721e+00, -6.666893e+00)	-0.000253568	-0.000167715
1e-04	17	(0.66666395, 0.66665534)	-11.11101202	-8.916856e-04	(-8.666672e+00, -6.666689e+00)	-2.53763e-05	-1.67663e-05
1e-05	19	(0.66666640, 0.66666553)	-11.11110120	-8.917952e-05	(-8.666667e+00, -6.666669e+00)	-2.54186e-06	-1.67465e-06
1e-06	21	(0.66666666, 0.66666651)	-11.11111000	-8.801393e-06	(-8.666667e+00, -6.666667e+00)	-3.15646e-07	-1.73226e-07