Решение задачи многомерной минимизации функции с ограничениями методом барьеров

Владимир Руцкий, 3057/2

Код программы

Исходный код 1: Барьерный метод

```
/*
 * barrier_method.hpp
 * Constrained minimization using barrier method.
 * Vladimir Rutsky <altsysrq@gmail.com>
 * 29.03.2009
 */

#ifndef NUMERIC_BARRIER_METHOD_HPP
#define NUMERIC_BARRIER_METHOD_HPP

#include "numeric_common.hpp"

#include <vector>

#include <boost/assert.hpp>
#include <boost/concept/assert.hpp>
#include <boost/concept_check.hpp>
#include <boost/bind.hpp>
#include <boost/lambda/lambda.hpp>
#include <boost/function.hpp>

#include "gradient_descent.hpp"

namespace numeric
{
namespace barrier_method
{
  namespace
  {
    // TODO: Use boost::lambda instead.
    // f(x) + mu * Summ(-1 / g_i(x))
    template< class S >
    struct AdditionalFunction
    {
    public:
      typedef S                                                  scalar_type;
      typedef vector<scalar_type>                                vector_type;

    private:
      typedef boost::function<scalar_type( vector_type )>        function_type;
      typedef std::vector<function_type>
          limit_functions_vec_type;

    public:
      template< class Func, class LimitFuncIterator >
      AdditionalFunction( Func func,
                          LimitFuncIterator limitFuncBegin, LimitFuncIterator
                              limitFuncEnd )
        : function_       (func)
        , limitFunctions_(limitFuncBegin, limitFuncEnd)
      {
        // TODO: Assertions on input types.
      }

      scalar_type operator()( scalar_type mu, vector_type const &x )
      {
        scalar_type result(0.0);

        result += function_(x);
        //std::cout << "  F (" << x << ") = " << result << " + ";// debug
        for (size_t i = 0; i < limitFunctions_.size(); ++i)
        {
          scalar_type const denominator = limitFunctions_[i](x);

          // TODO: Use normal constants.
          scalar_type const eps = 1e-15; // FIXME
          scalar_type const inf = 1e+8;
```

```
        if (abs(denominator) < eps)
        {
          // Division by zero.
          // TODO: Break loop and leave value infinite.
          result = inf;
        }
        else
        {
          result += -mu / denominator;
        }
        //std::cout << -mu / denominator << " + ";// debug
      }
      //std::cout << " == " << result << std::endl;// debug

      return result;
    }

  private:
    function_type              function_;
    limit_functions_vec_type limitFunctions_;
  };

  // TODO: Use boost::lambda instead.
  // f(x) + mu * Summ(-1 / g_i(x))
  template< class S >
  struct AdditionalFunctionGradient
  {
  public:
    typedef S                                              scalar_type;
    typedef vector<scalar_type>                            vector_type;

  private:
    typedef scalar_vector<scalar_type>                     scalar_vector_type;
    typedef boost::function<scalar_type( vector_type )>    function_type;
    typedef boost::function<vector_type( vector_type )>    function_grad_type;
    typedef std::vector<function_type> limit_functions_vec_type;
    typedef std::vector<function_grad_type> limit_functions_grads_vec_type;

  public:
    template< class FuncGrad, class LimitFuncIterator, class LimitFuncGradIterator >
    AdditionalFunctionGradient( FuncGrad funcGrad,
                                LimitFuncIterator     limitFuncBegin, LimitFuncIterator     limitFuncEnd,
                                LimitFuncGradIterator limitFuncGradBegin, LimitFuncGradIterator limitFuncGradEnd )
      : functionGrad_        (funcGrad)
      , limitFunctions_      (limitFuncBegin,     limitFuncEnd)
      , limitFunctionsGrads_ (limitFuncGradBegin, limitFuncGradEnd)
    {
      // TODO: Assertions on input types.
      BOOST_ASSERT(limitFunctions_.size() == limitFunctionsGrads_.size());
    }

    vector_type operator()( scalar_type mu, vector_type const &x )
    {
      vector_type result = functionGrad_(x);

      for (size_t i = 0; i < limitFunctions_.size(); ++i)
      {
        scalar_type const gx     = limitFunctions_      [i](x);
        vector_type const gGradx = limitFunctionsGrads_[i](x);

        // TODO: Use normal constants.
        scalar_type const eps = 1e-30; // FIXME!
        scalar_type const inf = 1e+8;
        if (abs(sqr(gx)) < eps)
        {
```

```
            // Division by zero.
            // TODO: Break loop and leave value infinite.
            return scalar_vector_type(x.size(), inf);
          }
          else
          {
            result = result + (mu / sqr(gx)) * gGradx;
          }
        }

        return result;
      }

    private:
      function_grad_type              functionGrad_;
      limit_functions_vec_type        limitFunctions_;
      limit_functions_grads_vec_type limitFunctionsGrads_;
    };

    // TODO: Use boost::lambda instead.
    template< class S >
    struct ConstrainPredicate
    {
    public:
      typedef S                                                  scalar_type;
      typedef vector<scalar_type>                                vector_type;

    private:
      typedef boost::function<scalar_type( vector_type )>        function_type;
      typedef std::vector<function_type>
          limit_functions_vec_type;

    public:
      template< class LimitFuncIterator >
      ConstrainPredicate( LimitFuncIterator limitFuncBegin, LimitFuncIterator
          limitFuncEnd )
        : limitFunctions_(limitFuncBegin, limitFuncEnd)
      {
        // TODO: Assertions on input types.
      }

      bool operator()( vector_type const &x )
      {
        for (size_t i = 0; i < limitFunctions_.size(); ++i)
          if (limitFunctions_[i](x) > 0)
            return false;

        return true;
      }

    private:
      limit_functions_vec_type limitFunctions_;
    };
  } // End of anonymous namespace

  template< class S >
  struct PointDebugInfo
  {
    typedef S                   scalar_type;
    typedef vector<scalar_type> vector_type;

    PointDebugInfo()
    {}

    PointDebugInfo( vector_type const &newx, scalar_type newmu, scalar_type newfx,
        scalar_type newBx )
      : x (newx)
      , mu(newmu)
      , fx(newfx)
      , Bx(newBx)
    {}
```

```
    vector_type x;
    scalar_type mu;
    scalar_type fx;
    scalar_type Bx;
  };

  // TODO: Habdle more end cases, not all problems input have solutions.
  template< class Func, class FuncGrad,
            class S,
            class LimitFuncIterator, class LimitFuncGradIterator,
            class PointsOut >
  inline
  vector<S>
    find_min( Func function, FuncGrad functionGrad,
              LimitFuncIterator     gBegin,     LimitFuncIterator     gEnd,
              LimitFuncGradIterator gGradBegin, LimitFuncGradIterator gGradEnd,
              vector<S> const &startPoint,
              S startMu, S beta,
              S epsilon,
              S gradientDescentPrecision, S gradientDescentStep,
              PointsOut pointsOut )
  {
    typedef S                                    scalar_type;
    typedef ublas::vector       <scalar_type> vector_type;
    typedef ublas::scalar_traits<scalar_type> scalar_traits_type;

    // TODO: Check for iterators concept assert.
    // Note: Input should be accurate so, that start point must be admissible not only for input function but for
    //   additional function too.

    typedef typename LimitFuncIterator::value_type     limit_func_type;
    typedef typename LimitFuncGradIterator::value_type limit_func_grad_type;

    BOOST_CONCEPT_ASSERT((boost::UnaryFunction<Func,                 scalar_type, vector_type>));
    BOOST_CONCEPT_ASSERT((boost::UnaryFunction<FuncGrad,             vector_type, vector_type>));
    BOOST_CONCEPT_ASSERT((boost::UnaryFunction<limit_func_type,      scalar_type, vector_type>));
    BOOST_CONCEPT_ASSERT((boost::UnaryFunction<limit_func_grad_type, vector_type, vector_type>));

    BOOST_ASSERT(epsilon > 0);

    std::vector<limit_func_type>      g    (gBegin,     gEnd);
    std::vector<limit_func_grad_type> gGrad(gGradBegin, gGradEnd);

    BOOST_ASSERT(g.size() == gGrad.size());
    BOOST_ASSERT(beta > 0 && beta < 1);

    // Building additional function and it's gradient.
    typedef boost::function<scalar_type( vector_type )> function_type;
    typedef boost::function<vector_type( vector_type )> function_gradient_type;
    typedef AdditionalFunction        <scalar_type>     additional_function_type;
    typedef AdditionalFunctionGradient<scalar_type>     additional_function_gradient_type;
    typedef ConstrainPredicate<scalar_type>             constrain_predicate_type;
    typedef PointDebugInfo<scalar_type>                 points_debug_info_type;

    additional_function_type          additionalFunc    (function,     gBegin, gEnd);
    additional_function_gradient_type additionalFuncGrad(functionGrad, gBegin, gEnd, gGradBegin, gGradEnd);
    constrain_predicate_type          constrainPred     (gBegin, gEnd);

    // Initializing
    vector_type x  = startPoint;
    scalar_type mu = startMu;

    BOOST_ASSERT(constrainPred(x)); // TODO: Rename 'constrain' by 'constraint'.
```

```
    mu /= beta;
    points_debug_info_type pdi(x, mu, function(x), (additionalFunc(mu, x) - function(x))
        / mu);
    mu *= beta;
    *pointsOut++ = pdi;

    size_t iterations = 0;
    while (true)
    {
      // Additional function: f(x) + mu * Summ(-1 / g_i(x))

      function_type          currFunc     = boost::bind<scalar_type>(additionalFunc,
          mu, _1);
      function_gradient_type currFuncGrad = boost::bind<vector_type>(additionalFuncGrad,
          mu, _1);

      // Solving additional unconstrained problem.
      vector_type newx;
      gradient_descent::gradient_descent_result const result =
          gradient_descent::find_min
            <function_type, function_gradient_type, vector_type>
              (currFunc, currFuncGrad,
               x,
               gradientDescentPrecision, gradientDescentStep,
               newx,
               constrainPred, DummyOutputIterator());
      BOOST_ASSERT(result == result); // TODO: Handle result states.

      // debug
      std::cout << iterations << ": " << newx << std::endl;
      // end of debug

      scalar_type const muBx = currFunc(newx) - function(newx);
      scalar_type const Bx = muBx / mu;
      points_debug_info_type pdi(newx, mu, function(newx), Bx);
      *pointsOut++ = pdi;

      // mu_k * B(x_k+1) < epsilon
      BOOST_ASSERT(muBx >= 0);
      if (muBx < epsilon)
      {
        // Required precision reached.
        return newx;
      }
      else
      {
        // Moving to next point.
        x = newx;
        mu *= beta;
      }

      ++iterations;

      // debug
      if (iterations >= 100)
      {
        std::cerr << "barrier_method::find_min(): Too many iterations!\n";
        break;
      }
      // end of debug
    }

    return x;
  }
} // End of namespace 'barrier_method'.
} // End of namespace 'numeric'.

#endif // NUMERIC_BARRIER_METHOD_HPP
```

Результаты решения

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