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);
        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-8;
          scalar_type const inf = 1e+8;
          if (abs(denominator) < eps)
```

```
        {
          // Division by zero.
          result = inf;
        }
        else
        {
          result += -mu / denominator;
        }
      }

      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 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 fx     = limitFunctions_      [i](x);
        vector_type const fgradx = limitFunctionsGrads_[i](x);

        for (size_t r = 0; r < x.size(); ++r)
        {
          // TODO: Use normal constants.
          scalar_type const eps = 1e-8;
          scalar_type const inf = 1e+8;
          if (abs(sqr(fx)) < eps)
          {
            // Division by zero.
            result[r] = inf;
          }
          else
```

```
            {
              result[r] += mu / sqr(fx) * fgradx[r];
            }
          }
        }

        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.

  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'.

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

  size_t iterations = 0;
  while (true)
  {
```

```
        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 const &newx =
            gradient_descent::find_min
              <function_type, function_gradient_type, vector_type>
                (currFunc, currFuncGrad,
                 x,
                 gradientDescentPrecision, gradientDescentStep,
                 constrainPred, DummyOutputIterator());

        points_debug_info_type pdi(newx, mu, function(newx), currFunc(newx) - function(newx));
        *pointsOut++ = pdi;

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

        ++iterations;

        // debug
        if (iterations >= 1000)
        {
          std::cerr << "barrier_method::find_min():ΓTooΓmanyΓiterations!\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: 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	(0.657644, 0.63160858)	-10.79788183		(-8.68471, -6.736783e+00)	-0.079139	-0.0531038
1e-02	12	(0.657644, 0.63160858)	-10.79788183	0.000000e+00	(-8.68471, -6.736783e+00)	-0.079139	-0.0531038
1e-03	13	(0.657644, 0.63160858)	-10.79788183	0.000000e+00	(-8.68471, -6.736783e+00)	-0.079139	-0.0531038
1e-04	14	(0.657644, 0.63160858)	-10.79788183	0.000000e+00	(-8.68471, -6.736783e+00)	-0.079139	-0.0531038
1e-05	15	(0.657644, 0.63160858)	-10.79788183	0.000000e+00	(-8.68471, -6.736783e+00)	-0.079139	-0.0531038
1e-06	16	(0.657644, 0.63160858)	-10.79788183	0.000000e+00	(-8.68471, -6.736783e+00)	-0.079139	-0.0531038
1e-07	17	(0.657644, 0.63160858)	-10.79788183	0.000000e+00	(-8.68471, -6.736783e+00)	-0.079139	-0.0531038
1e-08	18	(0.657644, 0.63160858)	-10.79788183	0.000000e+00	(-8.68471, -6.736783e+00)	-0.079139	-0.0531038
1e-09	19	(0.657644, 0.63160858)	-10.79788183	0.000000e+00	(-8.68471, -6.736783e+00)	-0.079139	-0.0531038