Код программы

Исходный код 1: Метод отсекающей гиперплоскости

```cpp
/*
 * kelley_cutting_plane.hpp
 * Kelley's convex cutting plane algorithm.
 * Vladimir Rutsky <altsysrq@gmail.com>
 * 27.04.2009
 */

#ifndef NUMERIC_KELLEY_CUTTING_PLANE_HPP
#define NUMERIC_KELLEY_CUTTING_PLANE_HPP

#include "numeric_common.hpp"

#include "linear_problem.hpp"
#include "linear_problem_algs.hpp"
#include "simplex_alg.hpp"

#include <vector>

#include <boost/assert.hpp>
#include <boost/concept/assert.hpp>
#include <boost/concept_check.hpp>

namespace numeric
{
namespace kelley_cutting_plane
{
  // For details see
  //   David G. Luenberger, Yinyu Ye
  //   Linear and Nonlinear Programming, Third Edition
  //   section 14.8 Kelley's convex cutting plane algorithm (p. 463).

  // Finds linear function minimum prefer to convex differentiable constraints.
  //   min c^T * x,   g(x) <= 0
  // g(x) and grad g(x) are defined by coordinates through iterators.
  // Initial constraints and problem formalization is stored in common linear problem structure,
  // which is expanded by new constraints along algorithm run.
  // TODO: Handle more cases, return value should be enumeration of different exit statuses.
  template< class S, class CLPTraits, class FuncIterator, class GradFuncIterator >
  inline
  vector<S>
    find_min( FuncIterator     funcBegin,     FuncIterator     funcEnd,
              GradFuncIterator gradFuncBegin, GradFuncIterator gradFuncEnd,
              linear_problem::common_linear_problem<S, CLPTraits> &commonLP )
  {
    typedef CLPTraits                                   clp_traits;
    typedef S                                           scalar_type;
    typedef vector<scalar_type>                         vector_type;
    typedef matrix<scalar_type>                         matrix_type;
    typedef zero_matrix<scalar_type>                    zero_matrix;
    typedef scalar_traits<scalar_type>                  scalar_traits_type;

    typedef typename FuncIterator::value_type           function_type;
    typedef typename GradFuncIterator::value_type       gradient_function_type;

    typedef linear_problem::common_linear_problem       <scalar_type> common_linear_problem_type;
    typedef linear_problem::canonical_linear_problem    <scalar_type> canonical_linear_problem_type;
    typedef typename linear_problem::converter_template_type<scalar_type >::type converter_type;

    // TODO: Using same type in much places now (like scalar_type).
    BOOST_CONCEPT_ASSERT((boost::UnaryFunction<function_type,         scalar_type, vector_type>));
```

```
  BOOST_CONCEPT_ASSERT((boost::UnaryFunction<gradient_function_type, vector_type, vector_type>));

  // TODO: Assert that input constraints are valid (they rise correct LP).
  BOOST_ASSERT(linear_problem::is_valid(commonLP));

  size_t const n = linear_problem::variables_count(commonLP);

  BOOST_ASSERT(n > 0);

  // Storing constrain function and its gradient.
  std::vector<function_type>          g    (funcBegin,     funcEnd);
  std::vector<gradient_function_type> gGrad(gradFuncBegin, gradFuncEnd);

  size_t nIterations(0);
  size_t const nMaxIterations(1000); // debug
  while (nIterations < nMaxIterations)
  {
    // Solving linear problem.
    vector_type commonResult;
    simplex::simplex_result_type const result = solve_by_simplex(commonLP, commonResult);
    BOOST_ASSERT(result == simplex::srt_min_found); // FIXME: Handle other cases.
    BOOST_ASSERT(linear_problem::check_linear_problem_solving_correctness(commonLP));

    // Adding new limits to common linear problem according to elements that satisfies g_i(x) > 0.
    bool isInside(true);
    for (size_t r = 0; r < n; ++r)
    {
      scalar_type const gr = g[r](commonResult);
      if (gr > 0)
      {
        isInside = false;

        // Adding new constraint:
        // g[r](commonResult) + grad g[r](commonResult) * (x - commonResult) <= 0.
        // or
        // grad g[r](commonResult) * x <= grad g[r](commonResult) * commonResult - g[r](commonResult).

        size_t const newRows = commonLP.b().size() + 1;
        BOOST_ASSERT(commonLP.ASign().size() == newRows - 1);
        BOOST_ASSERT(commonLP.A().size1()    == newRows - 1);
        BOOST_ASSERT(commonLP.A().size2()    == n);

        commonLP.b().resize(newRows, true);
        commonLP.A().resize(newRows, n, true);
        commonLP.ASign().resize(newRows, true);

        commonLP.ASign()(newRows - 1) = linear_problem::inequality_leq;

        vector_type const grGrad = gGrad[r](commonResult);
        BOOST_ASSERT(!eq_zero(norm_2(grGrad))); // FIXME: I think this is possible case.
        row(commonLP.A(), newRows - 1) = grGrad;

        commonLP.b()(newRows - 1) = inner_prod(grGrad, commonResult) - gr;

        // Assert that builded constraint cuts previosly founded minimum point.
        BOOST_ASSERT(inner_prod(row(commonLP.A(), newRows - 1), commonResult) > commonLP.b()(newRows - 1));

        BOOST_ASSERT(linear_problem::assert_valid(commonLP));
      }
    }

    if (isInside)
    {
      // Founded minimum of linear problem lies inside convex limits, so this is the answer.
```

```
        return commonResult;
      }
    }

    BOOST_ASSERT(0); // FIXME: Handle different cases.
    return vector_type(0);
  }
} // End of namespace 'kelley_cutting_plane'.
} // End of namespace 'numeric'.

#endif // NUMERIC_KELLEY_CUTTING_PLANE_HPP
```