Listing 1: Simplex algorithm

```
/*
 * simplex_alg.hpp
 * Simplex algorithm.
 * Vladimir Rutsky <altsysrq@gmail.com>
 * 15.02.2009
 */

#ifndef NUMERIC_SIMPLEX_ALG_HPP
#define NUMERIC_SIMPLEX_ALG_HPP

#include <iterator>
#include <algorithm>
#include <numeric>
#include <functional>
#include <vector>

#include <boost/numeric/ublas/matrix.hpp>
#include <boost/numeric/ublas/vector.hpp>
#include <boost/numeric/ublas/storage.hpp>
#include <boost/numeric/ublas/matrix_proxy.hpp>
#include <boost/numeric/ublas/functional.hpp>
#include <boost/bind.hpp>
#include <boost/optional.hpp>

#include "numeric_common.hpp"

#include "li_vectors.hpp"
#include "iterator.hpp"
#include "matrix_ops.hpp"
#include "vector_ops.hpp"
#include "linear_system.hpp"
#include "combination.hpp"

namespace numeric
{
namespace simplex
{
  // TODO: Move implementation lower.
  // TODO: Code may be overgeneralized.
  // TODO: Rename `value_type' by `scalar_type'.
  // TODO: Replace `basis' by `basic'.
  // TODO: Pass to functions `vector_expression's and remove most of concept asserts related to it.

  // Types of linear programming solving results.
  enum simplex_result_type
  {
    srt_min_found = 0,                // Function has minimum and it was founded.
    srt_not_limited,                  // Function is not limited from below.
    srt_none,                         // Set of admissible points is empty.
    srt_loop,                         // Loop in changing basis detected.
  };

  // Types of searching first basic vector results.
  enum first_basic_vector_result_type
  {
    fbrt_found = 0,                   // Found first basic vector.
    fbrt_none,                        // Set of admissible points is empty.
  };

  // Types of searching next basic vector results.
  enum next_basic_vector_result_type
  {
    nbrt_next_basic_vector_found = 0, // Found next basic vector.
    nbrt_min_found,                   // Current basic vector is solution of problem.
    nbrt_not_limited,                 // Function is not limited from below.
    nbrt_none,                        // Set of admissible points is empty.
    nbrt_loop,                        // Loop in changing basis detected.
  };

```

```cpp
namespace
{
  template< class MatrixType, class VectorType >
  bool assert_basic_vector( MatrixType const &A, VectorType const &b, VectorType const &x )
  {
    // TODO: Assert that value types in all input is compatible, different types for different vectors.
    BOOST_CONCEPT_ASSERT((ublas::MatrixExpressionConcept<MatrixType>));
    BOOST_CONCEPT_ASSERT((ublas::VectorExpressionConcept<VectorType>));

    typedef typename MatrixType::value_type          value_type;
    typedef vector<value_type>                       vector_type;
    typedef matrix<value_type>                       matrix_type;
    typedef basic_range<size_t, long>                range_type;
    typedef std::vector<size_t>                      range_container_type;
    typedef linear_independent_vectors<vector_type> li_vectors_type;

    range_type const N(0, A.size2()), M(0, A.size1());

    // TODO
    BOOST_ASSERT(N.size() > 0);
    BOOST_ASSERT(M.size() > 0);

    // TODO:
    //BOOST_ASSERT(M.size() < N.size());
    //BOOST_ASSERT(is_linear_independent(matrix_rows_begin(A), matrix_rows_end(A)));

    BOOST_ASSERT(x.size() == N.size());
    BOOST_ASSERT(b.size() == M.size());

    BOOST_ASSERT(std::find_if(x.begin(), x.end(), boost::bind<bool>(std::less<
        value_type>(), _1, 0.)) == x.end());

    range_container_type Nkp;
    copy_if(N.begin(), N.end(), std::back_inserter(Nkp),
        boost::bind<bool>(std::logical_not<bool>(), boost::bind<bool>(eq_zero_functor<
            value_type>(0.), boost::bind<value_type>(x, _1))));
    BOOST_ASSERT(Nkp.size() > 0);
    BOOST_ASSERT(Nkp.size() <= M.size());

    li_vectors_type basicVectorLICols;
    BOOST_ASSERT(is_linear_independent(matrix_columns_begin(submatrix(A, M.begin(), M.
        end(), Nkp.begin(), Nkp.end())),
                                       matrix_columns_end  (submatrix(A, M.begin(), M.
                                           end(), Nkp.begin(), Nkp.end()))));

    // Asserting that basic vector lies in set of admissible points.
    for (size_t r = 0; r < M.size(); ++r)
    {
      value_type const result = std::inner_product(row(A, r).begin(), row(A, r).end(),
          x.begin(), 0.);
      BOOST_ASSERT(eq_zero(result - b[r]));
    }

    return true;
  }
} // End of anonymous namespace.

// Finds next basic vector, that closer to goal of linear programming problem.
template< class MatrixType, class VectorType >
inline
first_basic_vector_result_type
  find_first_basic_vector( MatrixType const &A, VectorType const &b, VectorType const &
      c,
                           VectorType &basicV )
{
  // TODO: Assert that value types in all input is compatible, different types for
      different vectors.
  BOOST_CONCEPT_ASSERT((ublas::MatrixExpressionConcept<MatrixType>));
  BOOST_CONCEPT_ASSERT((ublas::VectorExpressionConcept<VectorType>));
```

```

    typedef typename VectorType::value_type     value_type;
    typedef ublas::vector<value_type>           vector_type;
    typedef ublas::matrix<value_type>           matrix_type;
    typedef ublas::scalar_vector<value_type>    scalar_vector_type;
    typedef ublas::basic_range<size_t, long>    range_type;
    typedef ublas::identity_matrix<value_type> identity_matrix_type;
    typedef ublas::matrix_row<matrix_type>      matrix_row_type;

    range_type const N(0, A.size2()), M(0, A.size1());

    // TODO
    BOOST_ASSERT(N.size() > 0);
    BOOST_ASSERT(M.size() > 0);

    BOOST_ASSERT(M.size() < N.size());
    BOOST_ASSERT(is_linear_independent(matrix_rows_begin(A), matrix_rows_end(A)));
    BOOST_ASSERT(basicV.size()      == N.size());
    BOOST_ASSERT(c.size()           == N.size());
    BOOST_ASSERT(b.size()           == M.size());

    vector_type newC(N.size() + M.size()), newB(M.size()), newBasicV(N.size() + M.size()), newResultV(N.size() + M.size());
    matrix_type newA(M.size(), N.size() + M.size());

    // Filling new 'c'.
    ublas::project(newC, ublas::range(0, N.size())) = scalar_vector_type(N.size(), 0);
    ublas::project(newC, ublas::range(N.size(), N.size() + M.size())) = scalar_vector_type(M.size(), 1);

    // Filling new 'A' and new 'b'.
    for (size_t r = 0; r < M.size(); ++r)
    {
      value_type const factor = (b[r] >= 0 ? 1 : -1);

      // TODO:
      //ublas::project(matrix_row_type(ublas::row(newA, r)), ublas::range(0, N.size())) = factor * ublas::row(A, r);
      matrix_row_type row(newA, r);
      ublas::vector_range<matrix_row_type>(row, ublas::range(0, N.size())) = factor * ublas::row(A, r);

      newB[r] = factor * b[r];
    }
    project(newA, ublas::range(0, M.size()), ublas::range(N.size(), N.size() + M.size())) = identity_matrix_type(M.size());

    // Filling new basic vector.
    ublas::project(newBasicV, ublas::range(0, N.size())) = scalar_vector_type(N.size(), 0.);
    ublas::project(newBasicV, ublas::range(N.size(), N.size() + M.size())) = newB;
    BOOST_ASSERT(assert_basic_vector(newA, newB, newBasicV));

    // Solving auxiliary problem.
    simplex_result_type const result = solve_augment_with_basic_vector(newA, newB, newC, newBasicV, newResultV);
    BOOST_ASSERT(result == srt_min_found); // it always has solution

    if (eq_zero(ublas::vector_norm_inf<vector_type>::apply(ublas::project(newResultV, ublas::range(N.size(), N.size() + M.size())))))
    {
      // Found basic vector.
      basicV = ublas::project(newResultV, ublas::range(0, N.size()));
      assert_basic_vector(A, b, basicV);
      return fbrt_found;
    }
    else
    {
      // Set of admissable points is empty.
      return fbrt_none;
    }
```

```
  }

  // Finds next basic vector, that closer to goal of linear programming problem.
  template< class MatrixType, class VectorType >
  inline
  next_basic_vector_result_type
    find_next_basic_vector( MatrixType const &A, VectorType const &b, VectorType const &c,
                            VectorType const &basicV, VectorType &nextBasicV )
  {
    // TODO: Assert that value types in all input is compatible, different types for different vectors.
    BOOST_CONCEPT_ASSERT((ublas::MatrixExpressionConcept<MatrixType>));
    BOOST_CONCEPT_ASSERT((ublas::VectorExpressionConcept<VectorType>));

    typedef typename MatrixType::value_type          value_type;
    typedef vector<value_type>                       vector_type;
    typedef matrix<value_type>                       matrix_type;
    typedef typename vector_type::size_type          size_type;
    typedef basic_range<size_t, long>                range_type;
    typedef std::vector<size_type>                   range_container_type;
    typedef linear_independent_vectors<vector_type> li_vectors_type;
    typedef identity_matrix<value_type>              identity_matrix_type;

    range_type const N(0, A.size2()), M(0, A.size1());

    // TODO
    BOOST_ASSERT(N.size() > 0);
    BOOST_ASSERT(M.size() > 0);

    BOOST_ASSERT(M.size() < N.size());
    BOOST_ASSERT(is_linear_independent(matrix_rows_begin(A), matrix_rows_end(A)));
    BOOST_ASSERT(basicV.size()     == N.size());
    BOOST_ASSERT(nextBasicV.size() == N.size());
    BOOST_ASSERT(c.size()          == N.size());
    BOOST_ASSERT(b.size()          == M.size());

    BOOST_ASSERT(assert_basic_vector(A, b, basicV));

    range_container_type Nkp, Nk;

    // Filling 'Nkp'.
    // Using strict check without precision. Not good.
    copy_if(N.begin(), N.end(), std::back_inserter(Nkp),
        boost::bind<bool>(std::logical_not<bool>(), boost::bind<bool>(eq_zero_functor<value_type>(), boost::bind<value_type>(basicV, _1))));
    BOOST_ASSERT(Nkp.size() > 0);
    BOOST_ASSERT(Nkp.size() <= M.size());
    BOOST_ASSERT(std::adjacent_find(Nkp.begin(), Nkp.end(), std::greater<size_type>()) == Nkp.end());
    BOOST_ASSERT(is_linear_independent(matrix_columns_begin(submatrix(A, M.begin(), M.end(), Nkp.begin(), Nkp.end())),
                                       matrix_columns_end  (submatrix(A, M.begin(), M.end(), Nkp.begin(), Nkp.end()))));

    // Iterating through basises till find suitable (Nk).
    bool foundBasis(false);
    combination::first_combination<size_type>(std::back_inserter(Nk), M.size());
    do
    {
      BOOST_ASSERT(std::adjacent_find(Nk.begin(), Nk.end(), std::greater<size_type>()) == Nk.end());
      BOOST_ASSERT(Nk.size() == M.size());
      if (std::includes(Nk.begin(), Nk.end(), Nkp.begin(), Nkp.end()))
      {
        bool const isLI = is_linear_independent(
            matrix_columns_begin(submatrix(A, M.begin(), M.end(), Nk.begin(), Nk.end())),
            matrix_columns_end  (submatrix(A, M.begin(), M.end(), Nk.begin(), Nk.end())));

        if (isLI)
```

```
      {
        // Basis was found.
        foundBasis = true;

        range_container_type Nkz, Lk;

        // Filling 'Nkz'.
        std::set_difference(Nk.begin(), Nk.end(), Nkp.begin(), Nkp.end(), std::back_inserter(Nkz));
        BOOST_ASSERT(std::adjacent_find(Nkz.begin(), Nkz.end(), std::greater<size_type>()) == Nkz.end());

        // Filling 'Lk'.
        std::set_difference(N.begin(), N.end(), Nk.begin(), Nk.end(), std::back_inserter(Lk));

        BOOST_ASSERT(Nk.size() == M.size());
        BOOST_ASSERT(Nkz.size() + Nkp.size() == M.size());
        BOOST_ASSERT(Lk.size() == N.size() - M.size());

        // Calculating 'A' submatrix inverse.
        matrix_type BNk(M.size(), M.size());
        BOOST_VERIFY(invert_matrix(submatrix(A, M.begin(), M.end(), Nk.begin(), Nk.end()), BNk));
        BOOST_ASSERT(eq_zero(ublas::matrix_norm_inf<matrix_type>::apply(ublas::prod(submatrix(A, M.begin(), M.end(), Nk.begin(), Nk.end()), BNk) - identity_matrix_type(M.size(), M.size())))));

        // Calculating 'd' vector.
        vector_type d(M.size());
        d = c - ublas::prod(ublas::trans(A), vector_type(ublas::prod(ublas::trans(BNk), subvector(c, Nk.begin(), Nk.end()))));

        BOOST_ASSERT(eq_zero(ublas::vector_norm_inf<matrix_type>::apply(subvector(d, Nk.begin(), Nk.end()))));

        vector_subvector<vector_type> dLk(subvector(d, Lk.begin(), Lk.end()));
        typename vector_subvector<vector_type>::const_iterator jkIt = std::find_if(
            dLk.begin(), dLk.end(),
            boost::bind<bool>(sl_functor<value_type>(), _1, 0.)); // Check with precision. If vector satisfies this, than it will satisfy optimal point criteria.

        if (jkIt == dLk.end())
        {
          // d[Lk] >= 0, current basic vector is optimal.
          nextBasicV = basicV;
          return nbrt_min_found;
        }
        else
        {
          // Searhcing next basic vector.

          size_type const jk = Lk[jkIt.index()];
          BOOST_ASSERT(sl(d(jk), 0.) && !eq_zero(d(jk)));

          vector_type u(scalar_vector<value_type>(N.size(), 0.));
          subvector(u, Nk.begin(), Nk.end()) = ublas::prod(BNk, ublas::column(A, jk));
          u[jk] = -1;

          vector_subvector<vector_type> uNk(subvector(u, Nk.begin(), Nk.end()));
          typename vector_subvector<vector_type>::const_iterator iuIt = std::find_if(
              uNk.begin(), uNk.end(),
              boost::bind<bool>(sg_functor<value_type>(), _1, 0.)); // Check with precision. Some errors may occur due to this.

          if (iuIt == uNk.end())
          {
            // u <= 0, goal function is not limited from below.
            return nbrt_not_limited;
          }
```

```
else
{
  // Found u[iu] > 0.
  BOOST_ASSERT((*iuIt > 0.) && sg(*iuIt, 0));

  bool canCalculateNextBasicV(false);

  if (Nkp.size() == Nk.size())
    canCalculateNextBasicV = true;

  if (!canCalculateNextBasicV)
  {
    vector_subvector<vector_type> uNkz(subvector(u, Nkz.begin(), Nkz.end()));
    if (std::find_if(uNkz.begin(), uNkz.end(), boost::bind<bool>(sg_functor<value_type>(), _1, 0.)) == uNkz.end())
      canCalculateNextBasicV = true;
  }

  if (canCalculateNextBasicV)
  {
    // Basic vector is not singular or u[Nkz] <= 0.
    // Now we need to find 'theta' so that one coordinate of new basis vector will become zero,
    // and one coordinate to 'theta'.

    boost::optional<std::pair<size_t, value_type> > minTheta;
    for (size_t ri = 0; ri < Nk.size(); ++ri)
    {
      size_t const r = Nk[ri];
      if (sg(u[r], 0)) // not strict check
      {
        static value_type const maxTheta = infinity<value_type>();

        value_type const theta = basicV(r) / u(r);

        if (theta < maxTheta && (!minTheta || theta < minTheta->second))
          minTheta = std::make_pair(r, theta);
      }
      else if (u[r] > 0 && eq_zero(u[r]))
      {
        // Adjusting u[r] to zero, needed for cases when basic vector has near zero components.
        u[r] = adjust(u[r]);
      }
    }

    // Finally constructing next basic vector.
    BOOST_VERIFY(minTheta);
    nextBasicV = basicV - minTheta->second * u;
    BOOST_ASSERT(eq_zero(nextBasicV[minTheta->first]));
    // Adjusting new basic vector.
    //nextBasicV = apply_to_all<functor::adjust>(nextBasicV);
    //std::cout << "Before adjusting nextBasicV:\n  " << nextBasicV << std::endl;
    nextBasicV = apply_to_all<functor::adjust<value_type> >(nextBasicV);

    {
      // Debug: Checking new basis vector 'Nkp'.

      range_container_type Nkp1;

      copy_if(N.begin(), N.end(), std::back_inserter(Nkp1),
           boost::bind<bool>(std::logical_not<bool>(), boost::bind<bool>(eq_zero_functor<value_type>(0.0), boost::bind<value_type>(nextBasicV, _1))));

      // Nkp1 = Nkp - {minTheta->first} + {jk}
      BOOST_ASSERT(std::find(Nkp.begin(), Nkp.end(), jk) == Nkp.end());
      BOOST_ASSERT(std::find(Nkp.begin(), Nkp.end(), minTheta->first) != Nkp.end());
```

```
                    BOOST_ASSERT(std::find(Nkp1.begin(), Nkp1.end(), jk) != Nkp1.end());
                    BOOST_ASSERT(std::find(Nkp1.begin(), Nkp1.end(), minTheta->first) == Nkp1.end());

                    range_container_type diff;
                    std::set_symmetric_difference(Nkp.begin(), Nkp.end(), Nkp1.begin(), Nkp1.end(), std::back_inserter(diff));

                    BOOST_ASSERT(diff.size() >= 2);
                    // End of debug.
                  }

                  BOOST_ASSERT(basicV.size() == nextBasicV.size() && basicV.size() == c.size()); // debug
                  // Asserting that next basic vector not increases goal function.
                  BOOST_ASSERT(std::inner_product(c.begin(), c.end(), basicV.begin(), 0.) >=
                               std::inner_product(c.begin(), c.end(), nextBasicV.begin(), 0.));
                  BOOST_ASSERT(assert_basic_vector(A, b, nextBasicV));

                  return nbrt_next_basic_vector_found;
                }
                else
                {
                  // Continuing and changing basis.
                }
              }
            }
          }
        }
      } while (combination::next_combination(Nk.begin(), N.size(), M.size()));

      // Basis not found: loop detected.
      return nbrt_loop;
    }

    // Solves linear programming problem described in augment form:
    //   min (c^T * x), where x: x >= 0, A * x = b,
    // using provided first basic vector.
    template< class MatrixType, class VectorType >
    inline
    simplex_result_type
      solve_augment_with_basic_vector( MatrixType const &A, VectorType const &b, VectorType const &c,
                                       VectorType const &basicV, VectorType &resultV )
    {
      // TODO: Assert that value types in all input is compatible, different types for different vectors.
      BOOST_CONCEPT_ASSERT((ublas::MatrixExpressionConcept<MatrixType>));
      BOOST_CONCEPT_ASSERT((ublas::VectorExpressionConcept<VectorType>));

      typedef typename MatrixType::value_type value_type;
      typedef ublas::vector<value_type>       vector_type;

      vector_type curBasicV = basicV;
      BOOST_ASSERT(assert_basic_vector(A, b, curBasicV));

      while (true)
      {
        vector_type nextBasicV(basicV.size());
        next_basic_vector_result_type const result = find_next_basic_vector(A, b, c, curBasicV, nextBasicV);
        switch (result)
        {
        case nbrt_next_basic_vector_found:
          BOOST_ASSERT(assert_basic_vector(A, b, nextBasicV));
          curBasicV = nextBasicV;
          break;

```

```
      case nbrt_min_found:
        BOOST_ASSERT(curBasicV == nextBasicV);
        resultV = curBasicV;
        BOOST_ASSERT(assert_basic_vector(A, b, resultV));
        return srt_min_found;
        break;

      case nbrt_not_limited:
        return srt_not_limited;
        break;

      case nbrt_none:
        return srt_none;
        break;

      case nbrt_loop:
        return srt_loop;
        break;
      }
    }

    // Impossible case.
    BOOST_ASSERT(0);
    return srt_none;
  }

  // Returns true is system is consistent, false otherwise.
  template< class M1, class E1, class M2, class E2 >
  bool remove_dependent_constraints( matrix_expression<M1> const &A,    vector_expression<E1> const &b,
                                     matrix_expression<M2>       &liA,  vector_expression<E2>       &lib )
  {
    typedef typename M1::value_type                   scalar_type; // TODO: Use type with most precision.
    typedef vector<scalar_type>                       vector_type;
    typedef matrix<scalar_type>                       matrix_type;
    typedef linear_independent_vectors<vector_type> li_vectors_type;

    size_t const n = A().size2(), m = A().size1();

    BOOST_ASSERT(b().size() == m);

    if (n == 0 || m == 0)
    {
      // Empty set of constraints. It is consistent.
      return true;
    }

    BOOST_ASSERT(n > 0);
    BOOST_ASSERT(m > 0);

    // Removing linear dependent constraints.
    liA().resize(m, n);
    lib().resize(m);
    size_t nextAddingRow = 0;

    li_vectors_type liARows, liARowsWithConstantTerm;

    for (size_t r = 0; r < m; ++r)
    {
      matrix_row<matrix_type const> ARow(A(), r);
      scalar_type const bval = b()(r);

      if (eq_zero(norm_2(ARow)))
      {
        // Handling case when coefficient vector is zero.

        if (!eq_zero(bval))
        {
          // Constraints are incosistent. Set of admissible points is empty.
```

```
        return false;
      }
      else
      {
        // Omitting zero rows.
        continue;
      }
    }

    vector_type extendedRow = paste(ARow, bval);

    if (liARows.is_independent(ARow))
    {
      // Adding linear independent constraint to result matrix.
      row(liA(), nextAddingRow) = ARow;
      lib()(nextAddingRow)      = bval;

      BOOST_VERIFY(liARows.insert(ARow));
      BOOST_VERIFY(liARowsWithConstantTerm.insert(extendedRow));

      ++nextAddingRow;
    }
    else
    {
      // Constraint coefficients vector is linearly dependent from previous coefficients rows.

      if (liARowsWithConstantTerm.is_independent(extendedRow))
      {
        // Constraints are incosistent. Set of admissible points is empty.
        return false;
      }
      else
      {
        // Omitting linear dependent constraints.
      }
    }
  }
  BOOST_ASSERT(nextAddingRow <= A().size2());

  liA().resize(nextAddingRow, n, true);
  lib().resize(nextAddingRow, true);

  return true;
}

// Solves linear programming problem described in augment form:
//   min (c^T * x), where x: x >= 0, A * x = b
template< class MatrixType, class VectorType >
inline
simplex_result_type solve_augment( MatrixType const &A, VectorType const &b, VectorType const &c,
                                   VectorType &resultV )
{
  // TODO: Assert that value types in all input is compatible, different types for different vectors.
  BOOST_CONCEPT_ASSERT((ublas::MatrixExpressionConcept<MatrixType>));
  BOOST_CONCEPT_ASSERT((ublas::VectorExpressionConcept<VectorType>));

  typedef typename MatrixType::value_type           value_type;
  typedef ublas::vector<value_type>                 vector_type;
  typedef ublas::matrix<value_type>                 matrix_type;
  typedef ublas::basic_range<size_t, long>          range_type;
  typedef std::vector<size_t>                       range_container_type;
  typedef linear_independent_vectors<vector_type> li_vectors_type;

  BOOST_ASSERT(A.size1() == b.size());
  BOOST_ASSERT(A.size2() == c.size());

  size_t const n = A.size2(), m = A.size1();

```

```
    // Removing linear dependent constraints.
    matrix_type newA(m, n);
    vector_type newb(m);
    if (!remove_dependent_constraints<matrix_type, vector_type, matrix_type, vector_type>(A, b, newA, newb))
    {
      // Constraints are incossistent. Set of admissible points is empty.
      return srt_none;
    }

    BOOST_ASSERT(newA.size1() <= newA.size2());

    if (newA.size1() == newA.size2())
    {
      // Linear program problem is well defined system of linear equations.

      BOOST_VERIFY(linear_system::solve(newA, newb, resultV));
      BOOST_ASSERT(eq_zero(norm_inf(prod(newA, resultV) - newb)));

      BOOST_ASSERT(assert_basic_vector(newA, newb, resultV));
      BOOST_ASSERT(assert_basic_vector(A, b, resultV));

      return srt_min_found;
    }
    else
    {
      BOOST_ASSERT(newA.size1() < newA.size2());
      return solve_li_augment(newA, newb, c, resultV);
    }
  }

  // Solves linear programming problem described in augment form:
  //   min (c^T * x), where x: x >= 0, A * x = b and rank(A) is equal to number of columns.
  template< class MatrixType, class VectorType >
  inline
  simplex_result_type solve_li_augment( MatrixType const &A, VectorType const &b, VectorType const &c,
                                        VectorType &resultV )
  {
    // TODO: Assert that value types in all input is compatible, different types for different vectors.
    BOOST_CONCEPT_ASSERT((ublas::MatrixExpressionConcept<MatrixType>));
    BOOST_CONCEPT_ASSERT((ublas::VectorExpressionConcept<VectorType>));

    typedef typename MatrixType::value_type          value_type;
    typedef ublas::vector<value_type>                vector_type;
    typedef ublas::matrix<value_type>                matrix_type;
    typedef ublas::basic_range<size_t, long>         range_type;
    typedef std::vector<size_t>                      range_container_type;
    typedef linear_independent_vectors<vector_type>  li_vectors_type;

    range_type const N(0, A.size2()), M(0, A.size1());

    // TODO
    BOOST_ASSERT(N.size() > 0);
    BOOST_ASSERT(M.size() > 0);

    BOOST_ASSERT(M.size() < N.size());
    BOOST_ASSERT(is_linear_independent(matrix_rows_begin(A), matrix_rows_end(A)));

    BOOST_ASSERT(c.size() == N.size());
    BOOST_ASSERT(b.size() == M.size());

    // Searching first basic vector using artificial basis.
    vector_type firstBasicV(N.size());
    first_basic_vector_result_type const result = find_first_basic_vector(A, b, c, firstBasicV);

    if (result == fbrt_found)
    {
```

```
        BOOST_ASSERT(assert_basic_vector(A, b, firstBasicV));
        // Solving linear programming problem starting from founded basic vector.
        return solve_augment_with_basic_vector(A, b, c, firstBasicV, resultV);
      }
      else
      {
        BOOST_ASSERT(result == fbrt_none);
        // Set of admissible points is empty.
        return srt_none;
      }
    }
} // End of namespace 'simplex'.
} // End of namespace 'numeric'.

#endif // NUMERIC_SIMPLEX_ALG_HPP
```