Код программы

Исходный код 1: Симплекс метод

```
1  /*
2   * simplex_alg.hpp
3   * Simplex algorithm.
4   * Vladimir Rutsky <altsysrq@gmail.com>
5   * 15.02.2009
6   */
7
8  #ifndef NUMERIC_SIMPLEX_ALG_HPP
9  #define NUMERIC_SIMPLEX_ALG_HPP
10
11 #include <iterator>
12 #include <algorithm>
13 #include <numeric>
14 #include <functional>
15 #include <vector>
16
17 #include <boost/numeric/ublas/matrix.hpp>
18 #include <boost/numeric/ublas/vector.hpp>
19 #include <boost/numeric/ublas/storage.hpp>
20 #include <boost/numeric/ublas/matrix_proxy.hpp>
21 #include <boost/numeric/ublas/functional.hpp>
22 #include <boost/bind.hpp>
23 #include <boost/optional.hpp>
24
25 #include "numeric_common.hpp"
26
27 #include "li_vectors.hpp"
28 #include "iterator.hpp"
29 #include "submatrix.hpp"
30 #include "subvector.hpp"
31 #include "invert_matrix.hpp"
32 #include "combination.hpp"
33
34 namespace numeric
35 {
36 namespace simplex
37 {
38   // TODO: Move implementation lower.
39   // TODO: Code may be overgeneralized.
40   // TODO: Rename 'value_type' by 'scalar_type'.
41   // TODO: Replace 'basis' by 'basic'.
42
43   // Types of linear programming solving results.
44   enum simplex_result_type
45   {
46     srt_min_found = 0,                // Function has minimum and it was founded.
47     srt_not_limited,                  // Function is not limited from below.
48     srt_none,                         // Set of admissible points is empty.
49     srt_loop,                         // Loop in changing basis detected.
50   };
51
52   // Types of searching first basic vector results.
53   enum first_basic_vector_result_type
54   {
55     fbrt_found = 0,                   // Found first basic vector.
56     fbrt_none,                        // Set of admissible points is empty.
57   };
58
59   // Types of searching next basic vector results.
60   enum next_basic_vector_result_type
61   {
62     nbrt_next_basic_vector_found = 0, // Found next basic vector.
63     nbrt_min_found,                   // Current basic vector is solution of problem.
64     nbrt_not_limited,                 // Function is not limited from below.
65     nbrt_none,                        // Set of admissible points is empty.
66     nbrt_loop,                        // Loop in changing basis detected.
67   };
```

```
68
69  namespace
70  {
71    template< class MatrixType, class VectorType >
72    bool assert_basic_vector( MatrixType const &A, VectorType const &b, VectorType const
          &x )
73    {
74      // TODO: Assert that value types in all input is compatible, different types for
            different vectors.
75      BOOST_CONCEPT_ASSERT((ublas::MatrixExpressionConcept<MatrixType>));
76      BOOST_CONCEPT_ASSERT((ublas::VectorExpressionConcept<VectorType>));
77
78      typedef typename MatrixType::value_type          value_type;
79      typedef vector<value_type>                vector_type;
80      typedef matrix<value_type>                matrix_type;
81      typedef basic_range<size_t, long>         range_type;
82      typedef std::vector<size_t>                      range_container_type;
83      typedef linear_independent_vectors<vector_type> li_vectors_type;
84
85      range_type const N(0, A.size2()), M(0, A.size1());
86
87      // TODO
88      BOOST_ASSERT(N.size() > 0);
89      BOOST_ASSERT(M.size() > 0);
90
91      // TODO:
92      //BOOST_ASSERT(M.size() < N.size());
93      //BOOST_ASSERT(is_linear_independent(matrix_rows_begin(A), matrix_rows_end(A)));
94
95      BOOST_ASSERT(x.size() == N.size());
96      BOOST_ASSERT(b.size() == M.size());
97
98      BOOST_ASSERT(std::find_if(x.begin(), x.end(), boost::bind<bool>(std::less<
            value_type>(), _1, 0.)) == x.end());
99
100     range_container_type Nkp;
101     copy_if(N.begin(), N.end(), std::back_inserter(Nkp),
102         boost::bind<bool>(std::logical_not<bool>(), boost::bind<bool>(eq_zero_functor<
                value_type>(0.), boost::bind<value_type>(x, _1))));
103     BOOST_ASSERT(Nkp.size() > 0);
104     BOOST_ASSERT(Nkp.size() <= M.size());
105
106     li_vectors_type basicVectorLICols;
107     BOOST_ASSERT(is_linear_independent(matrix_columns_begin(submatrix(A, M.begin(), M.
            end(), Nkp.begin(), Nkp.end())),
108                                        matrix_columns_end  (submatrix(A, M.begin(), M.
                                            end(), Nkp.begin(), Nkp.end()))));
109
110     // Asserting that basic vector lies in set of admissible points.
111     for (size_t r = 0; r < M.size(); ++r)
112     {
113       value_type const result = std::inner_product(row(A, r).begin(), row(A, r).end(),
              x.begin(), 0.);
114       BOOST_ASSERT(eq_zero(result - b[r]));
115     }
116
117     return true;
118   }
119 } // End of anonymous namespace.
120
121 // Finds next basic vector, that closer to goal of linear programming problem.
122 template< class MatrixType, class VectorType >
123 inline
124 first_basic_vector_result_type
125   find_first_basic_vector( MatrixType const &A, VectorType const &b, VectorType const &
        c,
126                            VectorType &basicV )
127 {
128   // TODO: Assert that value types in all input is compatible, different types for
        different vectors.
129   BOOST_CONCEPT_ASSERT((ublas::MatrixExpressionConcept<MatrixType>));
```

```
130     BOOST_CONCEPT_ASSERT((ublas::VectorExpressionConcept<VectorType>));
131
132     typedef typename VectorType::value_type     value_type;
133     typedef ublas::vector<value_type>           vector_type;
134     typedef ublas::matrix<value_type>           matrix_type;
135     typedef ublas::scalar_vector<value_type>    scalar_vector_type;
136     typedef ublas::basic_range<size_t, long>    range_type;
137     typedef ublas::identity_matrix<value_type> identity_matrix_type;
138     typedef ublas::matrix_row<matrix_type>      matrix_row_type;
139
140     range_type const N(0, A.size2()), M(0, A.size1());
141
142     // TODO
143     BOOST_ASSERT(N.size() > 0);
144     BOOST_ASSERT(M.size() > 0);
145
146     BOOST_ASSERT(M.size() < N.size());
147     BOOST_ASSERT(is_linear_independent(matrix_rows_begin(A), matrix_rows_end(A)));
148     BOOST_ASSERT(basicV.size()      == N.size());
149     BOOST_ASSERT(c.size()           == N.size());
150     BOOST_ASSERT(b.size()           == M.size());
151
152     vector_type newC(N.size() + M.size()), newB(M.size()), newBasicV(N.size() + M.size())
            , newResultV(N.size() + M.size());
153     matrix_type newA(M.size(), N.size() + M.size());
154
155     // Filling new 'c'.
156     ublas::project(newC, ublas::range(0, N.size())) = scalar_vector_type(N.size(), 0);
157     ublas::project(newC, ublas::range(N.size(), N.size() + M.size())) =
            scalar_vector_type(M.size(), 1);
158
159     // Filling new 'A' and new 'b'.
160     for (size_t r = 0; r < M.size(); ++r)
161     {
162       value_type const factor = (b[r] >= 0 ? 1 : -1);
163
164       // TODO:
165       //ublas::project(matrix_row_type(ublas::row(newA, r)), ublas::range(0, N.size())) =
              factor * ublas::row(A, r);
166       matrix_row_type row(newA, r);
167       ublas::vector_range<matrix_row_type>(row, ublas::range(0, N.size())) = factor *
            ublas::row(A, r);
168
169       newB[r] = factor * b[r];
170     }
171     project(newA, ublas::range(0, M.size()), ublas::range(N.size(), N.size() + M.size()))
            = identity_matrix_type(M.size());
172
173     // Filling new basic vector.
174     ublas::project(newBasicV, ublas::range(0, N.size())) = scalar_vector_type(N.size(),
            0.);
175     ublas::project(newBasicV, ublas::range(N.size(), N.size() + M.size())) = newB;
176     BOOST_ASSERT(assert_basic_vector(newA, newB, newBasicV));
177
178     // Solving auxiliary problem.
179     simplex_result_type const result = solve_augment_with_basic_vector(newA, newB, newC,
            newBasicV, newResultV);
180     BOOST_ASSERT(result == srt_min_found); // it always has solution
181
182     if (eq_zero(ublas::vector_norm_inf<vector_type>::apply(ublas::project(newResultV,
            ublas::range(N.size(), N.size() + M.size())))))
183     {
184       // Found basic vector.
185       basicV = ublas::project(newResultV, ublas::range(0, N.size()));
186       assert_basic_vector(A, b, basicV);
187       return fbrt_found;
188     }
189     else
190     {
191       // Set of admissable points is empty.
192       return fbrt_none;
```

```
193       }
194     }
195
196     // Finds next basic vector, that closer to goal of linear programming problem.
197     template< class MatrixType, class VectorType >
198     inline
199     next_basic_vector_result_type
200       find_next_basic_vector( MatrixType const &A, VectorType const &b, VectorType const &c
          ,
201                               VectorType const &basicV, VectorType &nextBasicV )
202     {
203       // TODO: Assert that value types in all input is compatible, different types for
            different vectors.
204       BOOST_CONCEPT_ASSERT((ublas::MatrixExpressionConcept<MatrixType>));
205       BOOST_CONCEPT_ASSERT((ublas::VectorExpressionConcept<VectorType>));
206
207       typedef typename MatrixType::value_type          value_type;
208       typedef vector<value_type>                       vector_type;
209       typedef matrix<value_type>                       matrix_type;
210       typedef typename vector_type::size_type          size_type;
211       typedef basic_range<size_t, long>                range_type;
212       typedef std::vector<size_type>                   range_container_type;
213       typedef linear_independent_vectors<vector_type> li_vectors_type;
214       typedef identity_matrix<value_type>              identity_matrix_type;
215
216       range_type const N(0, A.size2()), M(0, A.size1());
217
218       // TODO
219       BOOST_ASSERT(N.size() > 0);
220       BOOST_ASSERT(M.size() > 0);
221
222       BOOST_ASSERT(M.size() < N.size());
223       BOOST_ASSERT(is_linear_independent(matrix_rows_begin(A), matrix_rows_end(A)));
224       BOOST_ASSERT(basicV.size()     == N.size());
225       BOOST_ASSERT(nextBasicV.size() == N.size());
226       BOOST_ASSERT(c.size()          == N.size());
227       BOOST_ASSERT(b.size()          == M.size());
228
229       BOOST_ASSERT(assert_basic_vector(A, b, basicV));
230
231       range_container_type Nkp, Nk;
232
233       // Filling 'Nkp'.
234       // Using check with precision.
235       copy_if(N.begin(), N.end(), std::back_inserter(Nkp),
236           boost::bind<bool>(std::logical_not<bool>(), boost::bind<bool>(eq_zero_functor<
                value_type>(), boost::bind<value_type>(basicV, _1))));
237       BOOST_ASSERT(Nkp.size() > 0);
238       BOOST_ASSERT(Nkp.size() <= M.size());
239       BOOST_ASSERT(std::adjacent_find(Nkp.begin(), Nkp.end(), std::greater<size_type>()) ==
            Nkp.end());
240       BOOST_ASSERT(is_linear_independent(matrix_columns_begin(submatrix(A, M.begin(), M.end
            (), Nkp.begin(), Nkp.end())),
241                                          matrix_columns_end  (submatrix(A, M.begin(), M.end
                                               (), Nkp.begin(), Nkp.end()))));
242
243       // Iterating through basises till find suitable (Nk).
244       bool foundBasis(false);
245       combination::first_combination<size_type>(std::back_inserter(Nk), M.size());
246       do
247       {
248         BOOST_ASSERT(std::adjacent_find(Nk.begin(), Nk.end(), std::greater<size_type>()) ==
              Nk.end());
249         BOOST_ASSERT(Nk.size() == M.size());
250         if (std::includes(Nk.begin(), Nk.end(), Nkp.begin(), Nkp.end()))
251         {
252           bool const isLI = is_linear_independent(
253               matrix_columns_begin(submatrix(A, M.begin(), M.end(), Nk.begin(), Nk.end())),
254               matrix_columns_end  (submatrix(A, M.begin(), M.end(), Nk.begin(), Nk.end()))
                    ;
255
```

```
256         if (isLI)
257         {
258           // Basis was found.
259           foundBasis = true;
260
261           range_container_type Nkz, Lk;
262
263           // Filling 'Nkz'.
264           std::set_difference(Nk.begin(), Nk.end(), Nkp.begin(), Nkp.end(), std::
                back_inserter(Nkz));
265           BOOST_ASSERT(std::adjacent_find(Nkz.begin(), Nkz.end(), std::greater<size_type
                >()) == Nkz.end());
266
267           // Filling 'Lk'.
268           std::set_difference(N.begin(), N.end(), Nk.begin(), Nk.end(), std::
                back_inserter(Lk));
269
270           BOOST_ASSERT(Nk.size() == M.size());
271           BOOST_ASSERT(Nkz.size() + Nkp.size() == M.size());
272           BOOST_ASSERT(Lk.size() == N.size() - M.size());
273
274
275           // Calculating 'A' submatrix inverse.
276           matrix_type BNk(M.size(), M.size());
277           BOOST_VERIFY(invert_matrix(submatrix(A, M.begin(), M.end(), Nk.begin(), Nk.end
                ()), BNk));
278           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()))));
279
280           // Calculating 'd' vector.
281           vector_type d(M.size());
282           d = c - ublas::prod(ublas::trans(A), vector_type(ublas::prod(ublas::trans(BNk),
                 subvector(c, Nk.begin(), Nk.end()))));
283
284           BOOST_ASSERT(eq_zero(ublas::vector_norm_inf<matrix_type>::apply(subvector(d, Nk
                .begin(), Nk.end()))));
285
286           vector_subvector<vector_type> dLk(subvector(d, Lk.begin(), Lk.end()));
287           typename vector_subvector<vector_type>::const_iterator jkIt = std::find_if(
288               dLk.begin(), dLk.end(),
289               boost::bind<bool>(sl_functor<value_type>(), _1, 0.)); // Check with
                    precision. If vector satisfies this, than it will satisfy optimal point
                     criteria.
290
291           if (jkIt == dLk.end())
292           {
293             // d[Lk] >= 0, current basic vector is optimal.
294             nextBasicV = basicV;
295             return nbrt_min_found;
296           }
297           else
298           {
299             // Searhcing next basic vector.
300
301             size_type const jk = Lk[jkIt.index()];
302             BOOST_ASSERT(sl(d(jk), 0.) && !eq_zero(d(jk)));
303
304             vector_type u(scalar_vector<value_type>(N.size(), 0.));
305             subvector(u, Nk.begin(), Nk.end()) = ublas::prod(BNk, ublas::column(A, jk));
306             u[jk] = -1;
307
308             vector_subvector<vector_type> uNk(subvector(u, Nk.begin(), Nk.end()));
309             typename vector_subvector<vector_type>::const_iterator iuIt = std::find_if(
310                 uNk.begin(), uNk.end(),
311                 boost::bind<bool>(sg_functor<value_type>(), _1, 0.)); // Check with
                      precision. Some errors may occur due to this.
312
313             if (iuIt == uNk.end())
314             {
315               // u <= 0, goal function is not limited from below.
```

```
316               return nbrt_not_limited;
317             }
318             else
319             {
320               // Found u[iu] > 0.
321               BOOST_ASSERT((*iuIt > 0.) && sg(*iuIt, 0));
322
323               bool canCalculateNextBasicV(false);
324
325               if (Nkp.size() == Nk.size())
326                 canCalculateNextBasicV = true;
327
328               if (!canCalculateNextBasicV)
329               {
330                 vector_subvector<vector_type> uNkz(subvector(u, Nkz.begin(), Nkz.end()));
331                 if (std::find_if(uNkz.begin(), uNkz.end(), boost::bind<bool>(sg_functor<
                        value_type>(), _1, 0.)) == uNkz.end())
332                   canCalculateNextBasicV = true;
333               }
334
335               if (canCalculateNextBasicV)
336               {
337                 // Basic vector is not singular or u[Nkz] <= 0.
338                 // Now we need to find 'theta' so that one coordinate of new basis vector
                        will become zero,
339                 // and one coordinate to 'theta'.
340
341                 boost::optional<std::pair<size_t, value_type> > minTheta;
342                 for (size_t ri = 0; ri < Nk.size(); ++ri)
343                 {
344                   size_t const r = Nk[ri];
345                   if (sg(u[r], 0)) // not strict check
346                   {
347                     static value_type const maxTheta = infinity<value_type>();
348
349                     value_type const theta = basicV(r) / u(r);
350
351                     if (theta < maxTheta && (!minTheta || theta < minTheta->second))
352                       minTheta = std::make_pair(r, theta);
353                   }
354                   else if (u[r] > 0 && eq_zero(u[r]))
355                   {
356                     // Adjusting u[r] to zero, needed for cases when basic vector has
                            near zero components.
357                     u[r] = adjust(u[r]);
358                   }
359                 }
360
361                 // Finally constructing next basic vector.
362                 BOOST_VERIFY(minTheta);
363                 nextBasicV = basicV - minTheta->second * u;
364                 BOOST_ASSERT(eq_zero(nextBasicV[minTheta->first]));
365                 // Adjusting new basic vector.
366                 nextBasicV = apply_to_all<functor::adjust<value_type> >(nextBasicV);
367
368                 {
369                   // Debug: Checking new basis vector 'Nkp'.
370
371                   range_container_type Nkp1;
372
373                   copy_if(N.begin(), N.end(), std::back_inserter(Nkp1),
374                       boost::bind<bool>(std::logical_not<bool>(), boost::bind<bool>(
                              eq_zero_functor<value_type>(0.0), boost::bind<value_type>(
                              nextBasicV, _1))));
375
376                   // Nkp1 = Nkp - {minTheta->first} + {jk}
377                   BOOST_ASSERT(std::find(Nkp.begin(), Nkp.end(), jk)                ==
                          Nkp.end());
378                   BOOST_ASSERT(std::find(Nkp.begin(), Nkp.end(), minTheta->first) !=
                          Nkp.end());
```

```
379                     BOOST_ASSERT(std::find(Nkp1.begin(), Nkp1.end(), jk)                !=
                            Nkp1.end());
380                     BOOST_ASSERT(std::find(Nkp1.begin(), Nkp1.end(), minTheta->first) ==
                            Nkp1.end());
381
382                     range_container_type diff;
383                     std::set_symmetric_difference(Nkp.begin(), Nkp.end(), Nkp1.begin(),
                            Nkp1.end(), std::back_inserter(diff));
384
385                     BOOST_ASSERT(diff.size() >= 2);
386                     // End of debug.
387                   }
388
389                   BOOST_ASSERT(basicV.size() == nextBasicV.size() && basicV.size() == c.
                        size()); // debug
390                   // Asserting that next basic vector not increases goal function.
391                   BOOST_ASSERT(std::inner_product(c.begin(), c.end(), basicV.begin(), 0.)
                        >=
392                                std::inner_product(c.begin(), c.end(), nextBasicV.begin(),
                                    0.));
393                   BOOST_ASSERT(assert_basic_vector(A, b, nextBasicV));
394
395                   return nbrt_next_basic_vector_found;
396                 }
397                 else
398                 {
399                   // Continuing and changing basis.
400                 }
401               }
402             }
403           }
404         }
405       } while (combination::next_combination(Nk.begin(), N.size(), M.size()));
406
407       // Basis not found: loop detected.
408       return nbrt_loop;
409     }
410
411     // Solves linear programming problem described in augment form:
412     //    min (c^T * x), where x: x >= 0, A * x = b,
413     // using provided first basic vector.
414     template< class MatrixType, class VectorType >
415     inline
416     simplex_result_type
417       solve_augment_with_basic_vector( MatrixType const &A, VectorType const &b, VectorType
              const &c,
418                                        VectorType const &basicV, VectorType &resultV )
419     {
420       // TODO: Assert that value types in all input is compatible, different types for
              different vectors.
421       BOOST_CONCEPT_ASSERT((ublas::MatrixExpressionConcept<MatrixType>));
422       BOOST_CONCEPT_ASSERT((ublas::VectorExpressionConcept<VectorType>));
423
424       typedef typename MatrixType::value_type value_type;
425       typedef ublas::vector<value_type>       vector_type;
426
427       vector_type curBasicV = basicV;
428       BOOST_ASSERT(assert_basic_vector(A, b, curBasicV));
429
430       while (true)
431       {
432         vector_type nextBasicV(basicV.size());
433         next_basic_vector_result_type const result = find_next_basic_vector(A, b, c,
              curBasicV, nextBasicV);
434         switch (result)
435         {
436         case nbrt_next_basic_vector_found:
437           BOOST_ASSERT(assert_basic_vector(A, b, nextBasicV));
438           curBasicV = nextBasicV;
439           break;
440
```

```
441         case nbrt_min_found:
442           BOOST_ASSERT(curBasicV == nextBasicV);
443           resultV = curBasicV;
444           BOOST_ASSERT(assert_basic_vector(A, b, resultV));
445           return srt_min_found;
446           break;
447
448         case nbrt_not_limited:
449           return srt_not_limited;
450           break;
451
452         case nbrt_none:
453           return srt_none;
454           break;
455
456         case nbrt_loop:
457           return srt_loop;
458           break;
459         }
460       }
461
462       // Impossible case.
463       BOOST_ASSERT(0);
464       return srt_none;
465     }
466
467     // Solves linear programming problem described in augment form:
468     //   min (c^T * x), where x: x >= 0, A * x = b
469     template< class MatrixType, class VectorType >
470     inline
471     simplex_result_type solve_augment( MatrixType const &A, VectorType const &b, VectorType
          const &c,
472                                        VectorType &resultV )
473     {
474       // TODO: Assert that value types in all input is compatible, different types for
            different vectors.
475       BOOST_CONCEPT_ASSERT((ublas::MatrixExpressionConcept<MatrixType>));
476       BOOST_CONCEPT_ASSERT((ublas::VectorExpressionConcept<VectorType>));
477
478       typedef typename MatrixType::value_type          value_type;
479       typedef ublas::vector<value_type>                vector_type;
480       typedef ublas::matrix<value_type>                matrix_type;
481       typedef ublas::basic_range<size_t, long>         range_type;
482       typedef std::vector<size_t>                      range_container_type;
483       typedef linear_independent_vectors<vector_type> li_vectors_type;
484
485       range_type const N(0, A.size2()), M(0, A.size1());
486
487       // TODO
488       BOOST_ASSERT(N.size() > 0);
489       BOOST_ASSERT(M.size() > 0);
490
491       // Removing linear dependent constraints.
492       matrix_type newA(M.size(), N.size());
493       vector_type newb(M.size());
494       size_t nextAddingRow = 0;
495
496       li_vectors_type liARows;
497
498       for (size_t r = 0; r < M.size(); ++r)
499       {
500         matrix_row<MatrixType const> ARow(A, r);
501         value_type const bval = b(r);
502
503         if (eq_zero(norm_2(ARow)))
504         {
505           // Omitting zero rows.
506           BOOST_ASSERT(eq_zero(b(r))); // TODO: Handle as incorrect input return state.
507           continue;
508         }
509
```

```
510         if (liARows.is_independent(ARow))
511         {
512           // Adding linear independent constraint to result matrix.
513           row(newA, nextAddingRow) = ARow;
514           newb(nextAddingRow) = bval;
515
516           liARows.insert(ARow);
517
518           ++nextAddingRow;
519         }
520         else
521         {
522           // Omitting linear dependent constraints.
523           // FIXME: Must be checked is absolute terms is correspondent!
524         }
525       }
526       BOOST_ASSERT(nextAddingRow <= A.size2());
527
528       newA.resize(nextAddingRow, N.size(), true);
529       newb.resize(nextAddingRow, true);
530
531       if (newA.size1() == newA.size2())
532       {
533         // Linear program problem is well defined system of linear equations.
534         size_t const size = newA.size1();
535
536         matrix_type invNewA(size, size);
537         BOOST_VERIFY(invert_matrix(newA, invNewA)); // TODO: Handle zero determinant case.
538
539         resultV = prod(invNewA, newb);
540         BOOST_ASSERT(assert_basic_vector(newA, newb, resultV));
541         BOOST_ASSERT(assert_basic_vector(A, b, resultV));
542
543         return srt_min_found;
544       }
545       else
546       {
547         BOOST_ASSERT(newA.size1() < newA.size2());
548         return solve_li_augment(newA, newb, c, resultV);
549       }
550     }
551
552     // Solves linear programming problem described in augment form:
553     //   min (c^T * x), where x: x >= 0, A * x = b and rank(A) is equal to number of
           columns.
554     template< class MatrixType, class VectorType >
555     inline
556     simplex_result_type solve_li_augment( MatrixType const &A, VectorType const &b,
           VectorType const &c,
557                                           VectorType &resultV )
558     {
559       // TODO: Assert that value types in all input is compatible, different types for
             different vectors.
560       BOOST_CONCEPT_ASSERT((ublas::MatrixExpressionConcept<MatrixType>));
561       BOOST_CONCEPT_ASSERT((ublas::VectorExpressionConcept<VectorType>));
562
563       typedef typename MatrixType::value_type          value_type;
564       typedef ublas::vector<value_type>                vector_type;
565       typedef ublas::matrix<value_type>                matrix_type;
566       typedef ublas::basic_range<size_t, long>         range_type;
567       typedef std::vector<size_t>                      range_container_type;
568       typedef linear_independent_vectors<vector_type> li_vectors_type;
569
570       range_type const N(0, A.size2()), M(0, A.size1());
571
572       // TODO
573       BOOST_ASSERT(N.size() > 0);
574       BOOST_ASSERT(M.size() > 0);
575
576       BOOST_ASSERT(M.size() < N.size());
577       BOOST_ASSERT(is_linear_independent(matrix_rows_begin(A), matrix_rows_end(A)));
```

```
578
579     BOOST_ASSERT(c.size() == N.size());
580     BOOST_ASSERT(b.size() == M.size());
581
582     // Searching first basic vector using artificial basis.
583     vector_type firstBasicV(N.size());
584     first_basic_vector_result_type const result = find_first_basic_vector(A, b, c,
            firstBasicV);
585
586     if (result == fbrt_found)
587     {
588       BOOST_ASSERT(assert_basic_vector(A, b, firstBasicV));
589       // Solving linear programming problem starting from founded basic vector.
590       return solve_augment_with_basic_vector(A, b, c, firstBasicV, resultV);
591     }
592     else
593     {
594       BOOST_ASSERT(result == fbrt_none);
595       // Set of admissible points is empty.
596       return srt_none;
597     }
598   }
599 } // End of namespace 'simplex'.
600 } // End of namespace 'numeric'.
601
602 #endif // NUMERIC_SIMPLEX_ALG_HPP
```