

Listing 1: Simplex algorithm

```

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 "matrix_ops.hpp"
30 #include "vector_ops.hpp"
31 #include "linear_system.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     // TODO: Pass to functions 'vector_expression's and remove most of concept asserts
43         related to it.
44
45     // Types of linear programming solving results.
46     enum simplex_result_type
47     {
48         srt_min_found = 0,           // Function has minimum and it was founded.
49         srt_not_limited,             // Function is not limited from below.
50         srt_none,                   // Set of admissible points is empty.
51         srt_loop,                   // Loop in changing basis detected.
52     };
53
54     // Types of searching first basic vector results.
55     enum first_basic_vector_result_type
56     {
57         fbrt_found = 0,             // Found first basic vector.
58         fbrt_none,                 // Set of admissible points is empty.
59     };
60
61     // Types of searching next basic vector results.
62     enum next_basic_vector_result_type
63     {
64         nbrt_next_basic_vector_found = 0, // Found next basic vector.
65         nbrt_min_found,               // Current basic vector is solution of problem.
66         nbrt_not_limited,             // Function is not limited from below.
67         nbrt_none,                   // Set of admissible points is empty.
68         nbrt_loop,                   // Loop in changing basis detected.
69     };

```

```

70 namespace
71 {
72     template< class MatrixType, class VectorType >
73     bool assert_basic_vector( MatrixType const &A, VectorType const &b, VectorType const
74         &x )
75     {
76         // TODO: Assert that value types in all input is compatible, different types for
77         // different vectors.
78         BOOST_CONCEPT_ASSERT(( ublas::MatrixExpressionConcept<MatrixType>));
79         BOOST_CONCEPT_ASSERT(( ublas::VectorExpressionConcept<VectorType>));
80
81         typedef typename MatrixType::value_type value_type;
82         typedef vector<value_type> vector_type;
83         typedef matrix<value_type> matrix_type;
84         typedef basic_range<size_t, long> range_type;
85         typedef std::vector<size_t> range_container_type;
86         typedef linear_independent_vectors<vector_type> li_vectors_type;
87
88         range_type const N(0, A.size2()), M(0, A.size1());
89
90         // TODO
91         BOOST_ASSERT(N.size() > 0);
92         BOOST_ASSERT(M.size() > 0);
93
94         // TODO:
95         //BOOST_ASSERT(M.size() < N.size());
96         //BOOST_ASSERT(is_linear_independent(matrix_rows_begin(A), matrix_rows_end(A)));
97
98         BOOST_ASSERT(x.size() == N.size());
99         BOOST_ASSERT(b.size() == M.size());
100
101         BOOST_ASSERT(std::find_if(x.begin(), x.end(), boost::bind<bool>(std::less<
102             value_type>(), _1, 0.)) == x.end());
103
104         range_container_type Nkp;
105         copy_if(N.begin(), N.end(), std::back_inserter(Nkp),
106             boost::bind<bool>(std::logical_not<bool>(), boost::bind<bool>(eq_zero_functor<
107                 value_type>(0.), boost::bind<value_type>(x, _1))));
108         BOOST_ASSERT(Nkp.size() > 0);
109         BOOST_ASSERT(Nkp.size() <= M.size());
110
111         li_vectors_type basicVectorLICols;
112         BOOST_ASSERT(is_linear_independent(matrix_columns_begin(submatrix(A, M.begin(), M.
113             end(), Nkp.begin(), Nkp.end()))),
114             matrix_columns_end (submatrix(A, M.begin(), M.
115                 end(), Nkp.begin(), Nkp.end()))));
116
117         // Asserting that basic vector lies in set of admissible points.
118         for (size_t r = 0; r < M.size(); ++r)
119         {
120             value_type const result = std::inner_product(row(A, r).begin(), row(A, r).end(),
121                 x.begin(), 0.);
122             BOOST_ASSERT(eq_zero(result - b[r]));
123         }
124
125         return true;
126     }
127 } // End of anonymous namespace.
128
129 // Finds next basic vector, that closer to goal of linear programming problem.
130 template< class MatrixType, class VectorType >
131 inline
132 first_basic_vector_result_type
133 find_first_basic_vector( MatrixType const &A, VectorType const &b, VectorType const &
134     c,
135     VectorType &basicV )
136 {
137     // TODO: Assert that value types in all input is compatible, different types for
138     // different vectors.
139     BOOST_CONCEPT_ASSERT(( ublas::MatrixExpressionConcept<MatrixType>));
140     BOOST_CONCEPT_ASSERT(( ublas::VectorExpressionConcept<VectorType>));

```

```

132
133 typedef typename VectorType::value_type      value_type;
134 typedef ublas::vector<value_type>             vector_type;
135 typedef ublas::matrix<value_type>             matrix_type;
136 typedef ublas::scalar_vector<value_type>      scalar_vector_type;
137 typedef ublas::basic_range<size_t, long>      range_type;
138 typedef ublas::identity_matrix<value_type>    identity_matrix_type;
139 typedef ublas::matrix_row<matrix_type>        matrix_row_type;
140
141 range_type const N(0, A.size2()), M(0, A.size1());
142
143 // TODO
144 BOOST_ASSERT(N.size() > 0);
145 BOOST_ASSERT(M.size() > 0);
146
147 BOOST_ASSERT(M.size() < N.size());
148 BOOST_ASSERT(is_linear_independent(matrix_rows_begin(A), matrix_rows_end(A)));
149 BOOST_ASSERT(basicV.size() == N.size());
150 BOOST_ASSERT(c.size() == N.size());
151 BOOST_ASSERT(b.size() == M.size());
152
153 vector_type newC(N.size() + M.size()), newB(M.size()), newBasicV(N.size() + M.size())
154     , newResultV(N.size() + M.size());
155 matrix_type newA(M.size(), N.size() + M.size());
156
157 // Filling new 'c'.
158 ublas::project(newC, ublas::range(0, N.size())) = scalar_vector_type(N.size(), 0);
159 ublas::project(newC, ublas::range(N.size(), N.size() + M.size())) =
160     scalar_vector_type(M.size(), 1);
161
162 // Filling new 'A' and new 'b'.
163 for (size_t r = 0; r < M.size(); ++r)
164 {
165     value_type const factor = (b[r] >= 0 ? 1 : -1);
166
167     // TODO:
168     // ublas::project(matrix_row_type(ublas::row(newA, r)), ublas::range(0, N.size())) =
169     factor * ublas::row(A, r);
170     matrix_row_type row(newA, r);
171     ublas::vector_range<matrix_row_type>(row, ublas::range(0, N.size())) = factor *
172         ublas::row(A, r);
173
174     newB[r] = factor * b[r];
175 }
176 project(newA, ublas::range(0, M.size()), ublas::range(N.size(), N.size() + M.size()))
177     = identity_matrix_type(M.size());
178
179 // Filling new basic vector.
180 ublas::project(newBasicV, ublas::range(0, N.size())) = scalar_vector_type(N.size(),
181     0.);
182 ublas::project(newBasicV, ublas::range(N.size(), N.size() + M.size())) = newB;
183 BOOST_ASSERT(assert_basic_vector(newA, newB, newBasicV));
184
185 // Solving auxiliary problem.
186 simplex_result_type const result = solve_augment_with_basic_vector(newA, newB, newC,
187     newBasicV, newResultV);
188 BOOST_ASSERT(result == srt_min_found); // it always has solution
189
190 if (eq_zero(ublas::vector_norm_inf<vector_type>::apply(ublas::project(newResultV,
191     ublas::range(N.size(), N.size() + M.size())))))
192 {
193     // Found basic vector.
194     basicV = ublas::project(newResultV, ublas::range(0, N.size()));
195     assert_basic_vector(A, b, basicV);
196     return fbrt_found;
197 }
198 else
199 {
200     // Set of admissable points is empty.
201     return fbrt_none;
202 }

```

```

195 }
196
197 // Finds next basic vector, that closer to goal of linear programming problem.
198 template< class MatrixType, class VectorType >
199 inline
200 next_basic_vector_result_type
201 find_next_basic_vector( MatrixType const &A, VectorType const &b, VectorType const &c
202 ,
203 VectorType const &basicV, VectorType &nextBasicV )
204 {
205 // TODO: Assert that value types in all input is compatible, different types for
206 different vectors.
207 BOOST_CONCEPT_ASSERT(( ublas::MatrixExpressionConcept<MatrixType> ));
208 BOOST_CONCEPT_ASSERT(( ublas::VectorExpressionConcept<VectorType> ));
209
210 typedef typename MatrixType::value_type value_type;
211 typedef vector<value_type> vector_type;
212 typedef matrix<value_type> matrix_type;
213 typedef typename vector_type::size_type size_type;
214 typedef basic_range<size_t, long> range_type;
215 typedef std::vector<size_type> range_container_type;
216 typedef linear_independent_vectors<vector_type> li_vectors_type;
217 typedef identity_matrix<value_type> identity_matrix_type;
218
219 range_type const N(0, A.size2()), M(0, A.size1());
220
221 // TODO
222 BOOST_ASSERT(N.size() > 0);
223 BOOST_ASSERT(M.size() > 0);
224
225 BOOST_ASSERT(M.size() < N.size());
226 BOOST_ASSERT(is_linear_independent(matrix_rows_begin(A), matrix_rows_end(A)));
227 BOOST_ASSERT(basicV.size() == N.size());
228 BOOST_ASSERT(nextBasicV.size() == N.size());
229 BOOST_ASSERT(c.size() == N.size());
230 BOOST_ASSERT(b.size() == M.size());
231
232 BOOST_ASSERT(assert_basic_vector(A, b, basicV));
233
234 range_container_type Nkp, Nk;
235
236 // Filling 'Nkp'.
237 // Using strict check without precision. Not good.
238 copy_if(N.begin(), N.end(), std::back_inserter(Nkp),
239 boost::bind<bool>(std::logical_not<bool>(), boost::bind<bool>(eq_zero_functor<
240 value_type>(), boost::bind<value_type>(basicV, _1))));
241 BOOST_ASSERT(Nkp.size() > 0);
242 BOOST_ASSERT(Nkp.size() <= M.size());
243 BOOST_ASSERT(std::adjacent_find(Nkp.begin(), Nkp.end(), std::greater<size_type>()) ==
244 Nkp.end());
245 BOOST_ASSERT(is_linear_independent(matrix_columns_begin(submatrix(A, M.begin(), M.end()
246 (), Nkp.begin(), Nkp.end()))),
247 matrix_columns_end(submatrix(A, M.begin(), M.end()
248 (), Nkp.begin(), Nkp.end())));
249
250 // Iterating through bases till find suitable (Nk).
251 bool foundBasis(false);
252 combination::first_combination<size_type>(std::back_inserter(Nk), M.size());
253 do
254 {
255 BOOST_ASSERT(std::adjacent_find(Nk.begin(), Nk.end(), std::greater<size_type>()) ==
256 Nk.end());
257 BOOST_ASSERT(Nk.size() == M.size());
258 if (std::includes(Nk.begin(), Nk.end(), Nkp.begin(), Nkp.end()))
259 {
260 bool const isLI = is_linear_independent(
261 matrix_columns_begin(submatrix(A, M.begin(), M.end(), Nk.begin(), Nk.end())),
262 matrix_columns_end(submatrix(A, M.begin(), M.end(), Nk.begin(), Nk.end()))
263 );
264
265 if (isLI)

```

```

258 {
259     // Basis was found.
260     foundBasis = true;
261
262     range_container_type Nkz, Lk;
263
264     // Filling 'Nkz'.
265     std::set_difference(Nk.begin(), Nk.end(), Nkp.begin(), Nkp.end(), std::back_inserter(Nkz));
266     BOOST_ASSERT(std::adjacent_find(Nkz.begin(), Nkz.end(), std::greater<size_type>()) == Nkz.end());
267
268     // Filling 'Lk'.
269     std::set_difference(N.begin(), N.end(), Nk.begin(), Nk.end(), std::back_inserter(Lk));
270
271     BOOST_ASSERT(Nk.size() == M.size());
272     BOOST_ASSERT(Nkz.size() + Nkp.size() == M.size());
273     BOOST_ASSERT(Lk.size() == N.size() - M.size());
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(dLk.begin(), dLk.end(), boost::bind<bool>(sg_functor<value_type>(), _1, 0.)); // Check with
288     // precision. If vector satisfies this, than it will satisfy optimal point criteria.
289
290     if (jkIt == dLk.end())
291     {
292         // d[Lk] >= 0, current basic vector is optimal.
293         nextBasicV = basicV;
294         return nbrt_min_found;
295     }
296     else
297     {
298         // Searching next basic vector.
299
300         size_type const jk = Lk[jkIt.index()];
301         BOOST_ASSERT(sl(d(jk), 0.) && !eq_zero(d(jk)));
302
303         vector_type u(scalar_vector<value_type>(N.size(), 0.));
304         subvector(u, Nk.begin(), Nk.end()) = ublas::prod(BNk, ublas::column(A, jk));
305         u[jk] = -1;
306
307         vector_subvector<vector_type> uNk(subvector(u, Nk.begin(), Nk.end()));
308         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
309         // precision. Some errors may occur due to this.
310
311         if (iuIt == uNk.end())
312         {
313             // u <= 0, goal function is not limited from below.
314             return nbrt_not_limited;
315         }
316     }
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<
332             value_type>(), _1, 0.)) == uNkz.end())
333             canCalculateNextBasicV = true;
334     }
335
336     if (canCalculateNextBasicV)
337     {
338         // Basic vector is not singular or  $u[Nkz] \leq 0$ .
339         // Now we need to find 'theta' so that one coordinate of new basis vector
340         // will become zero,
341         // and one coordinate to 'theta'.
342
343         boost::optional<std::pair<size_t, value_type>> minTheta;
344         for (size_t ri = 0; ri < Nk.size(); ++ri)
345         {
346             size_t const r = Nk[ri];
347             if (sg(u[r], 0)) // not strict check
348             {
349                 static value_type const maxTheta = infinity<value_type>();
350
351                 value_type const theta = basicV(r) / u(r);
352
353                 if (theta < maxTheta && (!minTheta || theta < minTheta->second))
354                     minTheta = std::make_pair(r, theta);
355             }
356             else if (u[r] > 0 && eq_zero(u[r]))
357             {
358                 // Adjusting  $u[r]$  to zero, needed for cases when basic vector has
359                 // near zero components.
360                 u[r] = adjust(u[r]);
361             }
362         }
363
364         // Finally constructing next basic vector.
365         BOOST_VERIFY(minTheta);
366         nextBasicV = basicV - minTheta->second * u;
367         BOOST_ASSERT(eq_zero(nextBasicV[minTheta->first]));
368         // Adjusting new basic vector.
369         //nextBasicV = apply_to_all<functor::adjust>(nextBasicV);
370         //std::cout << "Before adjusting nextBasicV:\n " << nextBasicV << std::
371         //endl;
372         nextBasicV = apply_to_all<functor::adjust<value_type>>(nextBasicV);
373
374     {
375         // Debug: Checking new basis vector 'Nkp'.
376
377         range_container_type Nkp1;
378
379         copy_if(N.begin(), N.end(), std::back_inserter(Nkp1),
380             boost::bind<bool>(std::logical_not<bool>(), boost::bind<bool>(
381                 eq_zero_functor<value_type>(0.0), boost::bind<value_type>(
382                     nextBasicV, _1)))));
383
384         //  $Nkp1 = Nkp - \{minTheta \rightarrow first\} + \{jk\}$ 
385         BOOST_ASSERT(std::find(Nkp.begin(), Nkp.end(), jk) ==
386             Nkp.end());
387         BOOST_ASSERT(std::find(Nkp.begin(), Nkp.end(), minTheta->first) !=
388             Nkp.end());

```

```

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

```

```

443     case nbrt_min_found:
444         BOOST_ASSERT(curBasicV == nextBasicV);
445         resultV = curBasicV;
446         BOOST_ASSERT(assert_basic_vector(A, b, resultV));
447         return srt_min_found;
448         break;
449
450     case nbrt_not_limited:
451         return srt_not_limited;
452         break;
453
454     case nbrt_none:
455         return srt_none;
456         break;
457
458     case nbrt_loop:
459         return srt_loop;
460         break;
461     }
462 }
463
464 // Impossible case.
465 BOOST_ASSERT(0);
466 return srt_none;
467 }
468
469 // Returns true is system is consistent, false otherwise.
470 template< class M1, class E1, class M2, class E2 >
471 bool remove_dependent_constraints( matrix_expression<M1> const &A, vector_expression<
472     E1> const &b,
473                                     matrix_expression<M2>          &liA, vector_expression<
474     E2>          &lib )
475 {
476     typedef typename M1::value_type scalar_type; // TODO: Use type with
477         most precision.
478     typedef vector<scalar_type> vector_type;
479     typedef matrix<scalar_type> matrix_type;
480     typedef linear_independent_vectors<vector_type> li_vectors_type;
481
482     size_t const n = A().size2(), m = A().size1();
483
484     BOOST_ASSERT(b().size() == m);
485
486     if (n == 0 || m == 0)
487     {
488         // Empty set of constraints. It is consistent.
489         return true;
490     }
491
492     BOOST_ASSERT(n > 0);
493     BOOST_ASSERT(m > 0);
494
495     // Removing linear dependent constraints.
496     liA().resize(m, n);
497     lib().resize(m);
498     size_t nextAddingRow = 0;
499
500     li_vectors_type liARows, liARowsWithConstantTerm;
501
502     for (size_t r = 0; r < m; ++r)
503     {
504         matrix_row<matrix_type const> ARow(A(), r);
505         scalar_type const bval = b()(r);
506
507         if (eq_zero(norm_2(ARow)))
508         {
509             // Handling case when coefficient vector is zero.
510
511             if (!eq_zero(bval))
512             {
513                 // Constraints are inconsistent. Set of admissible points is empty.

```



```

511         return false;
512     }
513     else
514     {
515         // Omitting zero rows.
516         continue;
517     }
518 }
519
520 vector_type extendedRow = paste(ARow, bval);
521
522 if (liARows.is_independent(ARow))
523 {
524     // Adding linear independent constraint to result matrix.
525     row(liA(), nextAddingRow) = ARow;
526     lib()(nextAddingRow) = bval;
527
528     BOOST_VERIFY(liARows.insert(ARow));
529     BOOST_VERIFY(liARowsWithConstantTerm.insert(extendedRow));
530
531     ++nextAddingRow;
532 }
533 else
534 {
535     // Constraint coefficients vector is linearly dependent from previous
536     // coefficients rows.
537
538     if (liARowsWithConstantTerm.is_independent(extendedRow))
539     {
540         // Constraints are inconsistent. Set of admissible points is empty.
541         return false;
542     }
543     else
544     {
545         // Omitting linear dependent constraints.
546     }
547 }
548 BOOST_ASSERT(nextAddingRow <= A().size2());
549
550 liA().resize(nextAddingRow, n, true);
551 lib().resize(nextAddingRow, true);
552
553 return true;
554 }
555
556 // Solves linear programming problem described in augment form:
557 // min (c^T * x), where x: x >= 0, A * x = b
558 template< class MatrixType, class VectorType >
559 inline
560 simplex_result_type solve_augment( MatrixType const &A, VectorType const &b, VectorType
561                                     const &c,
562                                     VectorType &resultV )
563 {
564     // TODO: Assert that value types in all input is compatible, different types for
565     // different vectors.
566     BOOST_CONCEPT_ASSERT((ublas::MatrixExpressionConcept<MatrixType>));
567     BOOST_CONCEPT_ASSERT((ublas::VectorExpressionConcept<VectorType>));
568
569     typedef typename MatrixType::value_type value_type;
570     typedef ublas::vector<value_type> vector_type;
571     typedef ublas::matrix<value_type> matrix_type;
572     typedef ublas::basic_range<size_t, long> range_type;
573     typedef std::vector<size_t> range_container_type;
574     typedef linear_independent_vectors<vector_type> li_vectors_type;
575
576     BOOST_ASSERT(A.size1() == b.size());
577     BOOST_ASSERT(A.size2() == c.size());
578
579     size_t const n = A.size2(), m = A.size1();

```

```

579 // Removing linear dependent constraints.
580 matrix_type newA(m, n);
581 vector_type newb(m);
582 if (!remove_dependent_constraints<matrix_type, vector_type, matrix_type, vector_type>
    >(A, b, newA, newb))
583 {
584     // Constraints are incossistent. Set of admissible points is empty.
585     return srt_none;
586 }
587
588 BOOST_ASSERT(newA.size1() <= newA.size2());
589
590 if (newA.size1() == newA.size2())
591 {
592     // Linear program problem is well defined system of linear equations.
593
594     BOOST_VERIFY(linear_system::solve(newA, newb, resultV));
595     BOOST_ASSERT(eq_zero(norm_inf(prod(newA, resultV) - newb)));
596
597     BOOST_ASSERT(assert_basic_vector(newA, newb, resultV));
598     BOOST_ASSERT(assert_basic_vector(A, b, resultV));
599
600     return srt_min_found;
601 }
602 else
603 {
604     BOOST_ASSERT(newA.size1() < newA.size2());
605     return solve_li_augment(newA, newb, c, resultV);
606 }
607 }
608
609 // Solves linear programming problem described in augment form:
610 // min (c^T * x), where x: x >= 0, A * x = b and rank(A) is equal to number of
    columns.
611 template< class MatrixType, class VectorType >
612 inline
613 simplex_result_type solve_li_augment( MatrixType const &A, VectorType const &b,
    VectorType const &c,
614                                     VectorType &resultV )
615 {
616     // TODO: Assert that value types in all input is compatible, different types for
    different vectors.
617     BOOST_CONCEPT_ASSERT(( ublas::MatrixExpressionConcept<MatrixType> ));
618     BOOST_CONCEPT_ASSERT(( ublas::VectorExpressionConcept<VectorType> ));
619
620     typedef typename MatrixType::value_type value_type;
621     typedef ublas::vector<value_type> vector_type;
622     typedef ublas::matrix<value_type> matrix_type;
623     typedef ublas::basic_range<size_t, long> range_type;
624     typedef std::vector<size_t> range_container_type;
625     typedef linear_independent_vectors<vector_type> li_vectors_type;
626
627     range_type const N(0, A.size2()), M(0, A.size1());
628
629     // TODO
630     BOOST_ASSERT(N.size() > 0);
631     BOOST_ASSERT(M.size() > 0);
632
633     BOOST_ASSERT(M.size() < N.size());
634     BOOST_ASSERT(is_linear_independent(matrix_rows_begin(A), matrix_rows_end(A)));
635
636     BOOST_ASSERT(c.size() == N.size());
637     BOOST_ASSERT(b.size() == M.size());
638
639     // Searching first basic vector using artificial basis.
640     vector_type firstBasicV(N.size());
641     first_basic_vector_result_type const result = find_first_basic_vector(A, b, c,
    firstBasicV);
642
643     if (result == fbrt_found)
644     {

```

```

645     BOOST_ASSERT(assert_basic_vector(A, b, firstBasicV));
646     // Solving linear programming problem starting from founded basic vector.
647     return solve_augment_with_basic_vector(A, b, c, firstBasicV, resultV);
648 }
649 else
650 {
651     BOOST_ASSERT(result == fbrt_none);
652     // Set of admissible points is empty.
653     return srt_none;
654 }
655 }
656 } // End of namespace 'simplex'.
657 } // End of namespace 'numeric'.
658
659 #endif // NUMERIC_SIMPLEX_ALG_HPP

```