Listing 1: Genetic algorithm

```
/*
 * genetic.hpp
 * Genetics algorithms.
 * Vladimir Rutsky <altsysrq@gmail.com>
 * 31.03.2009
 */

#ifndef NUMERIC_GENETIC_HPP
#define NUMERIC_GENETIC_HPP

#include "numeric_common.hpp"

#include <vector>
#include <deque>

#include <boost/assert.hpp>
#include <boost/concept/assert.hpp>
#include <boost/concept_check.hpp>
#include <boost/bind.hpp>
#include <boost/random/linear_congruential.hpp>
#include <boost/random/uniform_real.hpp>
#include <boost/random/uniform_int.hpp>
#include <boost/random/variate_generator.hpp>
#include <boost/optional.hpp>
#include <boost/next_prior.hpp>

namespace numeric
{
namespace genetic
{
  typedef boost::minstd_rand base_generator_type; // TODO

  template< class V >
  struct ParallelepipedonUniformGenerator
  {
  private:
    BOOST_CONCEPT_ASSERT((ublas::VectorExpressionConcept<V>));

  public:
    typedef V vector_type;

  public:
    ParallelepipedonUniformGenerator( vector_type const &a, vector_type const &b )
      : a_(a)
      , b_(b)
      , rndGenerator_(42u)
    {
      BOOST_ASSERT(a_.size() == b_.size());
      BOOST_ASSERT(a_.size() > 0);
    }

    vector_type operator()() const
    {
      vector_type v(a_.size());

      for (size_t r = 0; r < v.size(); ++r)
      {
        BOOST_ASSERT(a_(r) <= b_(r));

        // TODO: Optimize.
        boost::uniform_real<> uni_dist(a_(r), b_(r));
        boost::variate_generator<base_generator_type &, boost::uniform_real<> > uni(
            rndGenerator_, uni_dist);

        v(r) = uni();

        BOOST_ASSERT(a_(r) <= v(r) && v(r) <= b_(r));
      }

      return v;
```

```
70      }
71
72    private:
73      vector_type const a_, b_;
74
75      mutable base_generator_type rndGenerator_;
76    };
77
78    struct LCCrossOver
79    {
80      LCCrossOver()
81        : rndGenerator_(30u)
82      {
83      }
84
85      template< class V >
86      V operator()( V const &x, V const &y ) const
87      {
88        // TODO: Optimize.
89        boost::uniform_real<> uni_dist(0.0, 1.0);
90        boost::variate_generator<base_generator_type &, boost::uniform_real<> > uni(
              rndGenerator_, uni_dist);
91
92        double const lambda = uni();
93
94        return x * lambda + (1 - lambda) * y;
95      }
96
97    private:
98      mutable base_generator_type rndGenerator_;
99    };
100
101   template< class Scalar >
102   struct ParallelepipedonMutation
103   {
104     typedef Scalar scalar_type;
105
106     template< class OffsetFwdIt >
107     ParallelepipedonMutation( OffsetFwdIt first, OffsetFwdIt beyond )
108       : rndGenerator_(30u)
109     {
110       deviations_.assign(first, beyond);
111     }
112
113     template< class V, class S >
114     V operator()( V const &x, S const scale ) const
115     {
116       BOOST_ASSERT(deviations_.size() == x.size());
117
118       V result(deviations_.size());
119
120       // TODO: Optimize.
121       for (size_t r = 0; r < deviations_.size(); ++r)
122       {
123         boost::uniform_real<> uni_dist(0.0, 1.0);
124         boost::variate_generator<base_generator_type &, boost::uniform_real<> > uni(
              rndGenerator_, uni_dist);
125
126         double const lambda = uni();
127
128         result(r) = x(r) + deviations_[r] * lambda * scale;
129       }
130
131       return result;
132     }
133
134   private:
135     std::vector<scalar_type>    deviations_;
136     mutable base_generator_type rndGenerator_;
137   };
138
```

```
139   // TODO: Documentation.
140   template< class Generator, class Crossover, class Mutation, class V, class Func, class
          FuncScalar, class PointsVecsOut >
141   V vectorSpaceGeneticSearch( Generator generator, Crossover crossover, Mutation mutation
          , Func fitness,
142                               size_t nIndividuals, double liveRate,
143                               typename V::value_type precision, size_t nPrecisionSelect,
144                               PointsVecsOut selectedPointsVecsOut, PointsVecsOut
                                      notSelectedPointsVecsOut )
145   {
146     BOOST_CONCEPT_ASSERT((ublas::VectorExpressionConcept<V>));
147     BOOST_CONCEPT_ASSERT((boost::UnaryFunction<Func, FuncScalar, V>));
148     // TODO: Concept asserts for Generator and Crossover.
149
150     typedef FuncScalar                 function_scalar_type;
151     typedef V                          vector_type;
152     typedef typename V::value_type     value_type;
153     typedef std::vector<vector_type> individuals_vector_type;
154
155     BOOST_ASSERT(0 <= liveRate && liveRate <= 1);
156     BOOST_ASSERT(nPrecisionSelect > 0);
157
158     individuals_vector_type population;
159     population.reserve(nIndividuals);
160     individuals_vector_type nextPopulation;
161     nextPopulation.reserve(nIndividuals);
162
163     base_generator_type rndGenerator(57u);
164
165     typedef std::deque<vector_type> fitted_individuals_deque_type;
166     fitted_individuals_deque_type fittedIndividuals;
167
168     // Spawning initial population.
169     for (size_t i = 0; i < nIndividuals; ++i)
170       population.push_back(generator());
171
172     size_t iterations = 0;
173     while (true)
174     {
175       // Sorting current population.
176       std::sort(population.begin(), population.end(),
177                 boost::bind(std::less<function_scalar_type>(), boost::bind(fitness, _1),
                        boost::bind(fitness, _2)));
178       size_t const nSelected = liveRate * nIndividuals;
179
180       BOOST_ASSERT(nSelected != 0 && nSelected != nIndividuals);
181
182       {
183         // Outputting current population.
184         individuals_vector_type selected;
185         selected.reserve(nSelected);
186         std::copy(population.begin(), boost::next(population.begin(), nSelected), std::
              back_inserter(selected));
187         *selectedPointsVecsOut++ = selected;
188
189         individuals_vector_type notSelected;
190         notSelected.reserve(nIndividuals - nSelected);
191         std::copy(boost::next(population.begin(), nSelected), boost::next(population.
              begin(), nIndividuals),
192                   std::back_inserter(notSelected));
193         *notSelectedPointsVecsOut++ = notSelected;
194       }
195
196       fittedIndividuals.push_front(population[0]);
197
198       BOOST_ASSERT(nPrecisionSelect > 0);
199       while (fittedIndividuals.size() > nPrecisionSelect)
200         fittedIndividuals.pop_back();
201
202       if (fittedIndividuals.size() == nPrecisionSelect)
203       {
```

```
204         // Checking is most fitted individual is changing in range of precision.
205
206         vector_type const lastMostFittedIndividual = fittedIndividuals.front();
207         bool satisfy(true);
208         for (typename fitted_individuals_deque_type::const_iterator it = boost::next(
                fittedIndividuals.begin()); it != fittedIndividuals.end(); ++it)
209         {
210           value_type const dist = ublas::norm_2(lastMostFittedIndividual - *it);
211           if (dist >= precision)
212           {
213             satisfy = false;
214             break;
215           }
216         }
217
218
219         if (satisfy)
220         {
221           // Evolved to population which meets precision requirements.
222           return lastMostFittedIndividual;
223         }
224       }
225
226       {
227         // Generating next population.
228
229         nextPopulation.resize(0);
230
231         // Copying good individuals.
232         std::copy(population.begin(), boost::next(population.begin(), nSelected),
233                   std::back_inserter(nextPopulation));
234         BOOST_ASSERT(nextPopulation.size() == nSelected);
235
236         // Crossover and mutation.
237         for (size_t i = nSelected; i < nIndividuals; ++i)
238         {
239           // TODO: Optimize.
240           boost::uniform_int<> uni_dist(0, nIndividuals - 1);
241           boost::variate_generator<base_generator_type &, boost::uniform_int<> > uni(
                rndGenerator, uni_dist);
242
243           size_t const xIdx = uni();
244           size_t const yIdx = uni();
245           BOOST_ASSERT(xIdx < population.size());
246           BOOST_ASSERT(yIdx < population.size());
247
248           // Crossover.
249           vector_type const x = population[xIdx], y = population[yIdx];
250           vector_type const child = crossover(x, y);
251
252           // Mutation.
253           vector_type const mutant = mutation(child, ublas::norm_2(x - y)); // TODO:
                Process may be unstable.
254
255           nextPopulation.push_back(mutant);
256         }
257       }
258
259       // Replacing old population.
260       population.swap(nextPopulation);
261
262       // debug, TODO
263       ++iterations;
264       if (iterations >= 1000)
265       {
266         std::cerr << "Too much iterations!\n";
267         break;
268       }
269       // end of debug
270     }
271
```

```
272         return population[0];
273       }
274   } // End of namespace 'genetic'.
275   } // End of namespace 'numeric'.
276
277   #endif // NUMERIC_GENETIC_HPP
```