Исходный код 1: Метод Ньютона

```cpp
/*
 * newton.hpp
 * Searching multidimensional function minimum with Newton algorithm.
 * Vladimir Rutsky <altsysrq@gmail.com>
 * 07.04.2009
 */

#ifndef NUMERIC_NEWTON_HPP
#define NUMERIC_NEWTON_HPP

#include "numeric_common.hpp"

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

#include "golden_section_search.hpp"
#include "lerp.hpp"
#include "determinant.hpp"
#include "invert_matrix.hpp"

namespace numeric
{
namespace newton
{
  // TODO: Inverse Hessian is a bad thing.
  template< class Func, class FuncGrad, class FuncHessian, class V, class PointsOut >
  inline
  ublas::vector<typename V::value_type>
    find_min( Func function, FuncGrad functionGrad, FuncHessian functionHessian,
              V const &startPoint,
              typename V::value_type precision,
              typename V::value_type step,
              PointsOut pointsOut )
  {
    // TODO: Now we assume that vector's coordinates and function values are same scalar types.
    // TODO: Assert on correctness of `ostr'.

    BOOST_CONCEPT_ASSERT((ublas::VectorExpressionConcept<V>));

    typedef typename V::value_type            scalar_type;
    typedef ublas::vector<scalar_type>        vector_type;
    typedef ublas::matrix<scalar_type>        matrix_type;

    BOOST_CONCEPT_ASSERT((boost::UnaryFunction<Func,        scalar_type, vector_type>));
    BOOST_CONCEPT_ASSERT((boost::UnaryFunction<FuncGrad,    vector_type, vector_type>));
    BOOST_CONCEPT_ASSERT((boost::UnaryFunction<FuncHessian, matrix_type, vector_type>));

    BOOST_ASSERT(precision > 0);

    // Setting current point to start point.
    vector_type x = startPoint;

    *pointsOut++ = x;

    size_t iterations = 0;
    while (true)
    {
      // Searching next point in specific direction based on antigradient.

      matrix_type const hessian    = functionHessian(x);
      scalar_type const hessianDet = matrix_determinant(hessian);

      if (eq_zero(hessianDet))
      {
        // Hessian determinant zero, it's means something. // TODO
        return x;
```

```
      }

      matrix_type invHessian;
      VERIFY(invert_matrix(hessian, invHessian));

      vector_type const grad     = functionGrad(x);
      vector_type const dirLong  = -ublas::prod(invHessian, grad);

      scalar_type const dirLen = ublas::norm_2(dirLong);
      if (eq_zero(dirLen))
      {
        // Function gradient is almost zero, found minimum.
        return x;
      }

      // Obtaining normalized direction of moving.
      vector_type const dir = dirLong / dirLen;
      BOOST_ASSERT(eq(ublas::norm_2(dir), 1));

      vector_type const s0 = x;
      vector_type const s1 = s0 + dir * step;

      typedef boost::function<scalar_type ( scalar_type )> function_bind_type;
      function_bind_type functionBind =
          boost::bind<scalar_type>(function, boost::bind<vector_type>(Lerp<scalar_type,
              vector_type>(0.0, 1.0, s0, s1), _1));
      scalar_type const section =
          golden_section::find_min<function_bind_type, scalar_type>(functionBind, 0.0,
              1.0, precision / step);
      BOOST_ASSERT(0 <= section && section <= 1);

      vector_type const nextX = s0 + dir * step * section;
      if (ublas::norm_2(x - nextX) < precision)
      {
        // Next point is equal to current (with precision), seems found minimum.
        return x;
      }

      // Moving to next point.
      x = nextX;
      *pointsOut++ = x;

      ++iterations;

      // debug
      if (iterations >= 100)
      {
        std::cerr << "Too many iterations!\n";
        break;
      }
      // end of debug
    }

    return x;
  }
} // End of namespace 'newton'.
} // End of namespace 'numeric'.

#endif // NUMERIC_NEWTON_HPP
```