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);
      //std::cout << "hessian: " << hessian << "\n"; // debug
      scalar_type const hessianDet = matrix_determinant(hessian);
      //std::cout << "hessianDet: " << hessianDet << "\n"; // debug

      if (eq_zero(hessianDet))
      {
```

```
      // Hessian determinant zero, it's means something. // TODO
      return x;
    }

    matrix_type invHessian;
    VERIFY(invert_matrix(hessian, invHessian));
    //std::cout << "invHessian: " << invHessian << "\n"; // debug

    vector_type const grad      = functionGrad(x);
    //std::cout << "grad: " << grad << "\n"; // debug
    vector_type const dirLong   = -ublas::prod(invHessian, grad);
    //std::cout << "dirLong: " << dirLong << "\n"; // debug

    scalar_type const dirLen = ublas::norm_2(dirLong);
    //std::cout << "dirLen: " << dirLen << "\n"; // debug
    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));
    //std::cout << "dir: " << dir << "\n"; // debug

    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);

    // debug
    /*
    std::cout << "x=";
    output_vector_coordinates(std::cout, x);
    std::cout << "dir=" << dir << "\n";
    std::cout << "grad=" << grad << "\n";
    std::cout << "invH=" << invHessian << "\n";
    std::cout << "f(x0) = " << function(s0 + dir * step * 0) << std::endl;
    std::cout << "f(x)  = " << function(s0 + dir * step * section) << std::endl;
    std::cout << "f(x1) = " << function(s0 + dir * step * 1) << std::endl;
    std::cout << "section=" << section << std::endl; // debug
    */
    // end of debug

    vector_type const nextX = s0 + dir * step * section;
    //std::cout << "dist= " << ublas::norm_2(x - nextX) << std::endl; // debug
    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
```

Таблица 1: Метод Ньютона

Точность	Шаги	x	$f(x)$	$f_i(x) - f_{i-1}(x)$	$\nabla f(x)$
1e-03	11	(0.00001671, -0.81627145)	4.89897953		(4.093416e-05, 4.136302e-04)
1e-04	12	(-0.00000069, -0.81649038)	4.89897949	-4.686444e-08	(-1.694262e-06, 1.139126e-05)
1e-05	12	(-0.00000003, -0.81649574)	4.89897949	-3.525447e-11	(-8.537893e-08, 1.541512e-06)
1e-06	12	(0.00000000, -0.81649664)	4.89897949	-6.457057e-13	(6.699014e-09, -1.043777e-07)
1e-07	12	(-0.00000000, -0.81649657)	4.89897949	-3.552714e-15	(-5.828198e-10, 2.497871e-08)
1e-08	15	(0.00000000, -0.81649659)	4.89897949	0.000000e+00	(8.046178e-10, -2.021831e-08)