Решение задачи многомерной минимизации функции
Метод Ньютона

Владимир Руцкий, 3057/2

1 Постановка задачи

Требуется найти с наперёд заданной точностью точку, в которой достигается минимум (локальный) многомерной функции $f(x)$ в некоторой области:

$$\min f(x), \quad x \in \mathbb{R}^n,$$

используя *метод Ньютона*.

Исходная функция: $f(x) = x_1^3 + 2x_2 + 4\sqrt{2 + x_1^2 + x_2^2}$, заданная на $\mathbb{R}^2$.

2 Исследование применимости метода Ньютона

Исходная функция непрерывно дифференцируема:

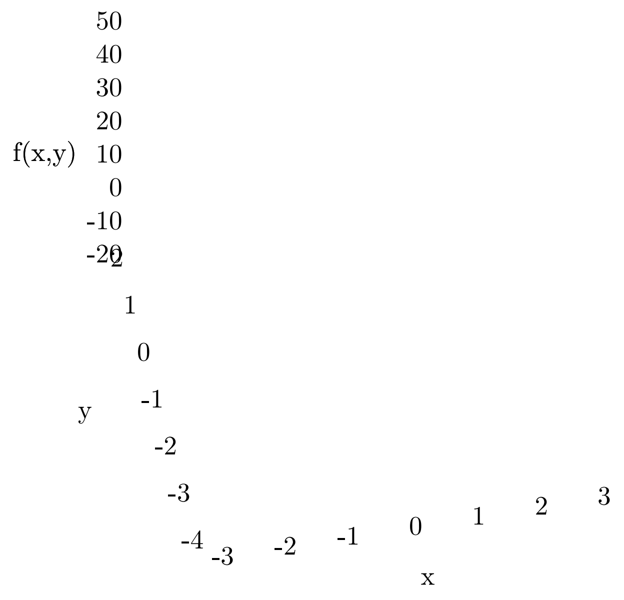
$$\frac{\partial f}{\partial x_1} = 3x_1^2 + \frac{4x_1}{\sqrt{2 + x_1^2 + x_2^2}},$$

$$\frac{\partial f}{\partial x_2} = 2 + \frac{4x_2}{\sqrt{2 + x_1^2 + x_2^2}}.$$

Функция не является ни выпуклой, ни ограниченной на $\mathbb{R}^2$, значит следует искать локальный минимум в некоторой области.

Построив график функции, можно попытаться найти достаточно близкую к локальному минимуму область, на которой функция будет выпуклой.

Рис. 1: График функции $f(x)$



Из графика видно, что минимум достигается близко к точке $(0, -1)$, будем исследовать функцию в окрестности этой точки.

Функция в исследуемой области не имеет особенных точек, ограничена и гладка, что предрасполагает к выполнению условия условия Липшица:

$$\exists R \in \mathbb{R} : \quad ||\nabla f(x) - \nabla f(y)|| \leqslant R||x - y||, \forall x, y \in \mathbb{R}^n,$$

Константа Липшица R была вычислена численно для дискретного набора точек из сетки $[-0.9; 2] \times [-3; 1]$ с шагом 0.01, и оказалась равной примерно 15.8. Следовательно итерационный процесс градиентного спуска будет сходиться.

2.1 Генетический алгоритм

Генетический алгоритм применим для поиска минимума выпуклой функции, а значит его можно использовать на области близкой к локальному минимуму функции, там где функция выпукла.

3 Описание алгоритма

3.1 Метод градиентного спуска

Метод градиентного спуска основывается на том, что для гладкой выпуклой функции градиент функции в точке направлен в сторону увеличения функции (в некоторой окрестности). Используя этот факт строится итерационный процесс приближения рассматриваемых точек области определения к точке минимума.

Выбирается начальное приближение минимума, далее строится последовательность точек, в которой каждая следующая точка выбирается на антиградиенте (луче, противоположном градиенту) в текущей точке:

$$x_{k+1} = x_k - \lambda_k \nabla f(x_k), \quad \lambda_k > 0$$

Шаг, на который "двигается" текущая точка за одну итерацию, выбирается следующим образом:

$$\lambda_k \in (0, q) \colon \quad f(x_k - \lambda_k \nabla f(x_k)) = \min_{0<\lambda<q} f(x_k - \lambda \nabla f(x_k)),$$

значение λ_k ищется *методом золотого сечения.*

Константа q задаёт интервал поиска минимума на антиградиенте.

Условием остановки итерационного процесса является событие, когда следующая точка находится от предыдущей на расстоянии меньшим ε:

$$||x_{k+1} - x_k|| < \varepsilon.$$

3.2 Генетический алгоритм

Суть *генетического алгоритма* для поиска минимума состоит в моделировании процесса биологической эволюции таким образом, что в качестве наиболее приспособленных особей выступают объекты, соответствующие минимуму функции.

Точки области определения функции f выступают в роли особей. Первоначальная популяция выбирается как набор произвольных точек в исследуемой области определения функции.

Каждая итерация работы алгоритма — это смена поколения. Смена поколения определяется трёмя процессами:

- Отбор.

  Из текущей популяции выбираются наиболее приспособленные. В качестве функции приспособленности выступает f: особь (точка) x более приспособлена чем y, если $f(x) < f(y)$.

  Отобранная, более приспособленная часть текущего поколения, перейдёт в следующее поколение.

- Размножение.

  Особи популяции в произвольном порядке скрещиваются друг с другом. Скрещивание особей (точек) x_1, x_2 порождает третью точку $y = \lambda x_1 + (1 - \lambda) x_2$, где λ выбирается произвольным образом из отрезка $[0, 1]$.

  Такое скрещивание обеспечивает в некоторой степени передачу потомству признаков родителей: положения в пространстве.

- Мутация.

  В свойства потомков текущей популяции вносятся хаотические изменения, это обеспечивает стабильное разнообразие каждой новой популяции.

  Мутация реализована как смещение особи (точки) на некоторый произвольный вектор: $y_{\text{mutated}} = y + \text{RandomVector}(||x_1 - x_2||)$. Модуль произвольного вектора линейно связан с расстоянием между родителями особи.

В результате новое поколение будет составлено из отобранных особей и мутировавших детей текущего поколения. Количество особей в поколении постоянно, недостающие в результате отбора особи выбираются из потомства.

Условием выхода из алгоритма является событие, что наиболее приспособленная особь (точка) на протяжении нескольких последних поколений не меняется больше чем на ε: $||x_i - x_{i-1}|| < \varepsilon$.

4 Код программы

4.1 Метод градиентного спуска

Исходный код 1: Градиентный спуск

```
/*
 * gradient_descent.hpp
 * Searching multidimensional function minimum with gradient descent algorithm.
 * Vladimir Rutsky <altsysrq@gmail.com>
 * 29.03.2009
 */

#ifndef NUMERIC_GRADIENT_DESCENT_HPP
#define NUMERIC_GRADIENT_DESCENT_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"

namespace numeric
{
namespace gradient_descent
{
  template< class Func, class FuncGrad, class V, class ConstrainPredicate, class PointsOut >
  inline
  ublas::vector<typename V::value_type>
    find_min( Func function, FuncGrad functionGrad,
              V const &startPoint,
              typename V::value_type precision,
              typename V::value_type step,
              ConstrainPredicate constrainPred,
              PointsOut          pointsOut )
  {
    // TODO: Now we assume that vector's coordinates and function values are same scalar types.
    // TODO: Assert on correctness of 'pointsOut'.

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

    typedef typename V::value_type            scalar_type;
    typedef ublas::vector<scalar_type>        vector_type;
    typedef ublas::scalar_traits<scalar_type> scalar_traits_type;

    BOOST_CONCEPT_ASSERT((boost::UnaryFunction<Func,     scalar_type, vector_type>));
```

```cpp
  BOOST_CONCEPT_ASSERT((boost::UnaryFunction<FuncGrad, vector_type, vector_type>));

  BOOST_ASSERT(precision > 0);

  // Setting current point to start point.
  vector_type x = startPoint;
  BOOST_ASSERT(constrainPred(x));

  *pointsOut++ = x;

  size_t iterations = 0;
  while (true)
  {
    // Searching next point in direction opposite to gradient.
    vector_type const grad = functionGrad(x);

    scalar_type const gradNorm = ublas::norm_2(grad);
    if (scalar_traits_type::equals(gradNorm, 0))
    {
      // Function gradient is almost zero, found minimum.
      return x;
    }

    vector_type const dir = -grad / gradNorm;
    BOOST_ASSERT(scalar_traits_type::equals(ublas::norm_2(dir), 1));

    scalar_type currStep = step;
    vector_type nextX;
    do
    {
      vector_type const s0 = x;
      vector_type const s1 = s0 + dir * currStep;

      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);

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

      // Decreasing search step.
      currStep /= 2.;
    } while (!constrainPred(nextX));

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

    ++iterations;

    // debug
    if (iterations >= 1000)
    {
      std::cerr << "gradient_descent::find_min(): Too many iterations!\n";
      break;
    }
    // end of debug
  }

  return x;
}
```

```
} // End of namespace 'gradient_descent'.
} // End of namespace 'numeric'.

#endif // NUMERIC_GRADIENT_DESCENT_HPP
```

4.2 Генетический алгоритм

Исходный код 2: Генетический алгоритм

```
/*
 * 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;
  }

private:
  vector_type const a_, b_;

  mutable base_generator_type rndGenerator_;
};

struct LCCrossOver
{
  LCCrossOver()
    : rndGenerator_(30u)
  {
  }

  template< class V >
  V operator()( V const &x, V const &y ) const
  {
    // TODO: Optimize.
    boost::uniform_real<> uni_dist(0.0, 1.0);
    boost::variate_generator<base_generator_type &, boost::uniform_real<> > uni(
        rndGenerator_, uni_dist);

    double const lambda = uni();

    return x * lambda + (1 - lambda) * y;
  }

private:
  mutable base_generator_type rndGenerator_;
};

template< class Scalar >
struct ParallelepipedonMutation
{
  typedef Scalar scalar_type;

  template< class OffsetFwdIt >
  ParallelepipedonMutation( OffsetFwdIt first, OffsetFwdIt beyond )
    : rndGenerator_(30u)
  {
    deviations_.assign(first, beyond);
  }

  template< class V, class S >
  V operator()( V const &x, S const scale ) const
  {
    BOOST_ASSERT(deviations_.size() == x.size());

    V result(deviations_.size());

    // TODO: Optimize.
    for (size_t r = 0; r < deviations_.size(); ++r)
    {
      boost::uniform_real<> uni_dist(0.0, 1.0);
      boost::variate_generator<base_generator_type &, boost::uniform_real<> > uni(
          rndGenerator_, uni_dist);

      double const lambda = uni();

      result(r) = x(r) + deviations_[r] * lambda * scale;
```

```
    }

    return result;
  }

private:
  std::vector<scalar_type>     deviations_;
  mutable base_generator_type rndGenerator_;
};

// TODO: Documentation.
template< class Generator, class Crossover, class Mutation, class V, class Func, class FuncScalar, class PointsVecsOut >
V vectorSpaceGeneticSearch( Generator generator, Crossover crossover, Mutation mutation, Func fitness,
                            size_t nIndividuals, double liveRate,
                            typename V::value_type precision, size_t nPrecisionSelect,
                            PointsVecsOut selectedPointsVecsOut, PointsVecsOut notSelectedPointsVecsOut )
{
  BOOST_CONCEPT_ASSERT((ublas::VectorExpressionConcept<V>));
  BOOST_CONCEPT_ASSERT((boost::UnaryFunction<Func, FuncScalar, V>));
  // TODO: Concept asserts for Generator and Crossover.

  typedef FuncScalar                function_scalar_type;
  typedef V                         vector_type;
  typedef typename V::value_type    value_type;
  typedef std::vector<vector_type> individuals_vector_type;

  BOOST_ASSERT(0 <= liveRate && liveRate <= 1);
  BOOST_ASSERT(nPrecisionSelect > 0);

  individuals_vector_type population;
  population.reserve(nIndividuals);
  individuals_vector_type nextPopulation;
  nextPopulation.reserve(nIndividuals);

  base_generator_type rndGenerator(57u);

  typedef std::deque<vector_type> fitted_individuals_deque_type;
  fitted_individuals_deque_type fittedIndividuals;

  // Spawning initial population.
  for (size_t i = 0; i < nIndividuals; ++i)
    population.push_back(generator());

  size_t iterations = 0;
  while (true)
  {
    // Sorting current population.
    std::sort(population.begin(), population.end(),
              boost::bind(std::less<function_scalar_type>(), boost::bind(fitness, _1), boost::bind(fitness, _2)));
    size_t const nSelected = liveRate * nIndividuals;

    BOOST_ASSERT(nSelected != 0 && nSelected != nIndividuals);

    {
      // Outputting current population.
      individuals_vector_type selected;
      selected.reserve(nSelected);
      std::copy(population.begin(), boost::next(population.begin(), nSelected), std::back_inserter(selected));
      *selectedPointsVecsOut++ = selected;

      individuals_vector_type notSelected;
      notSelected.reserve(nIndividuals - nSelected);
      std::copy(boost::next(population.begin(), nSelected), boost::next(population.begin(), nIndividuals),
                std::back_inserter(notSelected));
      *notSelectedPointsVecsOut++ = notSelected;
```

```
      }

      fittedIndividuals.push_front(population[0]);

      BOOST_ASSERT(nPrecisionSelect > 0);
      while (fittedIndividuals.size() > nPrecisionSelect)
        fittedIndividuals.pop_back();

      if (fittedIndividuals.size() == nPrecisionSelect)
      {
        // Checking is most fitted individual is changing in range of precision.

        vector_type const lastMostFittedIndividual = fittedIndividuals.front();
        bool satisfy(true);
        for (typename fitted_individuals_deque_type::const_iterator it = boost::next(
            fittedIndividuals.begin()); it != fittedIndividuals.end(); ++it)
        {
          value_type const dist = ublas::norm_2(lastMostFittedIndividual - *it);
          if (dist >= precision)
          {
            satisfy = false;
            break;
          }
        }


        if (satisfy)
        {
          // Evolved to population which meets precision requirements.
          return lastMostFittedIndividual;
        }
      }

      {
        // Generating next population.

        nextPopulation.resize(0);

        // Copying good individuals.
        std::copy(population.begin(), boost::next(population.begin(), nSelected),
                  std::back_inserter(nextPopulation));
        BOOST_ASSERT(nextPopulation.size() == nSelected);

        // Crossover and mutation.
        for (size_t i = nSelected; i < nIndividuals; ++i)
        {
          // TODO: Optimize.
          boost::uniform_int<> uni_dist(0, nIndividuals - 1);
          boost::variate_generator<base_generator_type &, boost::uniform_int<> > uni(
              rndGenerator, uni_dist);

          size_t const xIdx = uni();
          size_t const yIdx = uni();
          BOOST_ASSERT(xIdx < population.size());
          BOOST_ASSERT(yIdx < population.size());

          // Crossover.
          vector_type const x = population[xIdx], y = population[yIdx];
          vector_type const child = crossover(x, y);

          // Mutation.
          vector_type const mutant = mutation(child, ublas::norm_2(x - y)); // TODO:
              Process may be unstable.

          nextPopulation.push_back(mutant);
        }
      }

      // Replacing old population.
      population.swap(nextPopulation);

```

```
      // debug, TODO
      ++iterations;
      if (iterations >= 1000)
      {
        std::cerr << "Too much iterations!\n";
        break;
      }
      // end of debug
    }

    return population[0];
  }
} // End of namespace 'genetic'.
} // End of namespace 'numeric'.

#endif // NUMERIC_GENETIC_HPP
```

5 Результаты решения

5.1 Метод градиентного спуска

Результаты решения приведены в таблице **??**.

Начальной точкой была выбрана точка (2.5, 2.5) , шаг для поиска минимума методом золотого сечения был равен 0.5 .

Таблица 1: Результаты работы алгоритма градиентного спуска

Точность	Шаги	x	$f(x)$	$f_i(x) - f_{i-1}(x)$	$\nabla f(x)$
1e-03	12	(1.67515e-05, -0.81647076)	4.89897949		(4.10337e-05, 4.744064e-05)
1e-04	12	(5.32455e-06, -0.81647068)	4.89897949	-3.053593e-10	(1.30426e-05, 4.757985e-05)
1e-05	13	(2.39964e-07, -0.81649641)	4.89897949	-6.507568e-10	(5.8779e-07, 3.093111e-07)
1e-06	13	(4.23671e-07, -0.81649656)	4.89897949	1.234568e-13	(1.03778e-06, 3.759286e-08)
1e-07	14	(8.27773e-09, -0.81649657)	4.89897949	-2.202682e-13	(2.02762e-08, 2.704985e-08)
1e-08	15	(2.61515e-09, -0.81649658)	4.89897949	0.000000e+00	(6.40578e-09, 2.714819e-09)

5.2 Генетический алгоритм

Результаты решения приведены в таблице **??**.

Популяция состояла из 1000 особей, первоначально расположенных в прямоугольнике $[-0.9; 2] \times [-3; 1]$. 80% особей отбирались и оставались в популяции.

Таблица 2: Результаты работы генетического алгоритма

Точность*	Шаги	x	$f(x)$	$f_i(x) - f_{i-1}(x)$	$\nabla f(x)$
1e-03	51	(-8.89596e-05, -0.81641967)	4.89897950		(-0.000217887, 1.413057e-04)
1e-04	73	(1.91949e-06, -0.81650223)	4.89897949	-1.509204e-08	(4.70178e-06, -1.037411e-05)
1e-05	89	(1.91949e-06, -0.81650223)	4.89897949	0.000000e+00	(4.70178e-06, -1.037411e-05)
1e-06	120	(2.42424e-07, -0.81649652)	4.89897949	-3.372858e-11	(5.93816e-07, 1.107199e-07)
1e-07	120	(2.42424e-07, -0.81649652)	4.89897949	0.000000e+00	(5.93816e-07, 1.107199e-07)
1e-08	120	(2.42424e-07, -0.81649652)	4.89897949	0.000000e+00	(5.93816e-07, 1.107199e-07)

6 Возможные дополнительные исследования

6.1 Метод градиентного спуска

Найденная методом золотого сечения следующая точка на антиградиете x_{k+1}:

$$\lambda_k \in (0, q): \quad f(x_k - \lambda_k \nabla f(x_k)) = \min_{0<\lambda<q} f(x_k - \lambda \nabla f(x_k)),$$

$$x_{k+1} = x_k - \lambda_k \nabla f(x_k),$$

является минимумом $\psi(v) = f(x_k - v\nabla f(x_k)), v \in (0, q)$. Если λ_k лежит сильно внутри $[0, q]$ (возможен случай, когда λ_k стремиться к q, если итерационный шаг недостаточно велик), то из условия минимальности следует, что $0 = f'(\lambda_k) = \frac{(\nabla f(x_{k+1}), \nabla f(x_k))}{||\nabla f(x_k)||}$, т. е. величина проекции градиента f в x_{k+1} на линию антиградиента равна нулю. Из этого следует, что, при достаточной длине шага, у каждого отрезка $[x_k, x_{k+1}]$ начало x_k будет перпендикулярно силовой линии, проходящей через x_k, а конец будет лежать на касательной к силовой линии, проходящей через x_{k+1}.

Данная особенность слабо проявляется на исследуемой функции, т.к. был выбран достаточно малый шаг и большая часть x_{k+1} соответствует краям отрезков антиградиента.

7 Обоснование достоверности полученного результата

7.1 Метод градиентного спуска

Градиент исходной функции (его норма), в полученном с точностью ε решении, обращается в ноль с некоторой точностью, пропорциональной ε, что является достаточным условием для минимума выпуклой функции.

7.2 Генетический алгоритм

Полученные результаты работы генетического алгоритма близки к результатам, полученным методом градиентного спуска, но менее точны, ввиду большого числа случайных факторов, использовавшихся в алгоритме.