**Министерство науки и высшего образования Российской Федерации**
**Федеральное государственное бюджетное образовательное учреждение высшего образования**
**«Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)»**
**(МГТУ им. Н.Э. Баумана)**

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Теоретическая информатика и компьютерные технологии»

# Домашняя работа № 3

## по курсу «Теория искусственных нейронных сетей»

«Методы многомерного поиска»

Студент группы ИУ9-71Б Афанасьев И.

Преподаватель Каганов Ю.Т.

*Москва 2024*

# 1 Цель работы

1. Изучение алгоритмов многомерного поиска 1-го и 2-го порядка.
2. Разработка программ реализации алгоритмов многомерного поиска 1-го и 2-го порядка.
3. Вычисление экстремумов функции.

# 2 Постановка задачи

Требуется найти минимум тестовой функции Розенброка

$$f(x) = \sum_{i=1}^{n-1} [a(x_i^2 - x_{i+1})^2 + b(x_i - 1)^2] + f_0$$

методами сопряжённых градиентов (Флетчера-Ривза и Полака-Рибьера), квазиньютоновским методом (Девидона-Флетчера-Пауэлла), методом Левенберга-Марквардта.

Вариант № 4: $a = 250$, $b = 2$, $f_0 = 50$, $n = 2$.

# 3 Реализация

Программа написана на языке C++ с использованием библиотеки Eigen. В листинге 1 приводится исходный код программы.

Листинг 1: Файл main.cc

```
#include <spdlog/spdlog.h>

#include <Eigen/Dense>
#include <cmath>

namespace {

class IMultivariateFunction {
 public:
```

```
  virtual ~IMultivariateFunction() = default;

 public:
  virtual std::size_t Size() const = 0;

  virtual double At(const Eigen::VectorXd& u) const = 0;

  virtual Eigen::VectorXd Gradient(const Eigen::VectorXd& u) const = 0;

  virtual Eigen::MatrixXd Hessian(const Eigen::VectorXd& u) const = 0;
};

class RosenbrockFunction final : public IMultivariateFunction {
 public:
  static constexpr std::size_t kInputSize = 2;

  std::size_t Size() const override { return kInputSize; }

  double At(const Eigen::VectorXd& u) const override {
    return 250 * std::pow(std::pow(u.x(), 2) - u.y(), 2) +
           2 * std::pow(u.x() - 1, 2) + 50;
  }

  Eigen::VectorXd Gradient(const Eigen::VectorXd& u) const override {
    return Eigen::Vector<double, kInputSize>{
        1000 * std::pow(u.x(), 3) - 1000 * u.x() * u.y() + 4 * u.x() - 4,
        -500 * std::pow(u.x(), 2) + 500 * u.y()};
  }

  Eigen::MatrixXd Hessian(const Eigen::VectorXd& u) const override {
    return Eigen::Matrix<double, kInputSize, kInputSize>{
        {3000 * std::pow(u.x(), 2) - 1000 * u.y() + 4, -1000 * u.x()},
        {-1000 * u.x(), 500},
    };
  }
};

template <std::size_t N>
constexpr std::array<std::size_t, N + 1> GetFibonacciNumbers() {
  auto numbers = std::array<std::size_t, N + 1>{};
```

```
  numbers[0] = 0;
  numbers[1] = 1;
  for (std::size_t i = 2; i <= N; ++i) {
    numbers[i] = numbers[i - 2] + numbers[i - 1];
  }
  return numbers;
}

double FibonacciSearch(const std::function<double(double)>& f, const double a,
                       const double b) {
  static constexpr std::size_t kN = 150;
  static constexpr auto kF = GetFibonacciNumbers<kN>();

  auto xl = a;
  auto xr = b;
  auto l0 = xr - xl;
  auto li = static_cast<double>(kF[kN - 2]) / static_cast<double>(kF[kN]) * l0;

  double x1 = 0, x2 = 0, f1 = 0, f2 = 0;
  for (std::size_t i = 2; i <= kN; ++i) {
    if (li > l0 / 2) {
      x1 = xr - li;
      x2 = xl + li;
    } else {
      x1 = xl + li;
      x2 = xr - li;
    }

    f1 = f(x1);
    f2 = f(x2);

    if (f1 < f2) {
      xr = x2;
      li = static_cast<double>(kF[kN - i]) / kF[kN - (i - 2)] * l0;
    } else if (f1 > f2) {
      xl = x1;
      li = static_cast<double>(kF[kN - i]) / kF[kN - (i - 2)] * l0;
    } else {
      xl = x1;
      xr = x2;
```

```
      li = static_cast<double>(kF[kN - i]) / kF[kN - (i - 2)] * (xr - xl);
    }

    l0 = xr - xl;
  }

  if (f1 <= f2) {
    return x1;
  }
  return x2;
}

Eigen::VectorXd GradientDescent(const IMultivariateFunction& f,
                                const Eigen::VectorXd& x0,
                                const std::size_t max_iterations,
                                const double grad_epsilon, const double delta,
                                const double epsilon, const double a,
                                const double b) {
  Eigen::VectorXd x = x0, x_next;
  Eigen::VectorXd grad;
  const auto phi = [&](const double alpha) {
    return f.At(x - alpha * grad);
  };
  for (std::size_t k = 0; k < max_iterations; ++k) {
    spdlog::debug("Iteration {}, x = ({}, {}), f(x)={}", k, x.x(), x.y(), f.At(x));
    grad = f.Gradient(x);
    if (grad.norm() < grad_epsilon) {
      spdlog::debug("||grad|| < epsilon");
      return x;
    }

    const auto alpha = FibonacciSearch(phi, a, b);
    x_next = x - alpha * grad;

    if ((x_next - x).norm() < delta &&
        std::abs(f.At(x_next) - f.At(x)) < epsilon) {
      spdlog::debug("||x_next - x|| < delta && |f(x_next) - f(x)| < epsilon");
      return x_next;
    }

```

```
    x = std::move(x_next);
  }

  return x;
}

Eigen::VectorXd FletcherReeves(const IMultivariateFunction& f,
                               const Eigen::VectorXd& x0,
                               const std::size_t max_iterations,
                               const double grad_epsilon, const double delta,
                               const double epsilon, const double a,
                               const double b) {
  Eigen::VectorXd x = x0, x_next;
  Eigen::VectorXd prev_grad, grad;
  Eigen::VectorXd prev_d, d;
  const auto phi = [&](const double alpha) {
    return f.At(x + alpha * d);
  };
  for (std::size_t k = 0; k < max_iterations; ++k) {
    spdlog::debug("Iteration {}, x = ({}, {}), f(x)={}", k, x.x(), x.y(), f.At(x));
    grad = f.Gradient(x);
    if (grad.norm() < grad_epsilon) {
      spdlog::debug("||grad|| < epsilon");
      return x;
    }

    d = -grad;
    if (k > 0) {
      const auto w_prev = grad.squaredNorm() / prev_grad.squaredNorm();
      d += w_prev * prev_grad;
    }

    const auto alpha = FibonacciSearch(phi, a, b);
    x_next = x + alpha * d;

    if ((x_next - x).norm() < delta &&
        std::abs(f.At(x_next) - f.At(x)) < epsilon) {
      spdlog::debug("||x_next - x|| < delta && |f(x_next) - f(x)| < epsilon");
      return x_next;
    }
```

```
    x = std::move(x_next);
    prev_grad = std::move(grad);
    prev_d = std::move(d);
  }

  return x;
}

Eigen::VectorXd PolakRibier(const IMultivariateFunction& f,
                            const Eigen::VectorXd& x0,
                            const std::size_t max_iterations,
                            const double grad_epsilon, const double delta,
                            const double epsilon, const double a,
                            const double b) {
  Eigen::VectorXd x = x0, x_next;
  Eigen::VectorXd prev_grad, grad;
  Eigen::VectorXd prev_d, d;
  double alpha;
  const auto phi = [&](const double alpha) {
    return f.At(x + alpha * d);
  };
  const auto n = f.Size();
  for (std::size_t k = 0; k < max_iterations; ++k) {
    spdlog::debug("Iteration {}, x = ({}, {}), f(x)={}", k, x.x(), x.y(), f.At(x));
    grad = f.Gradient(x);
    if (grad.norm() < grad_epsilon) {
      spdlog::debug("||grad|| < epsilon");
      return x;
    }

    d = -grad;
    if (k > 0) {
      const auto w_prev =
          (k % n == 0 ? 0
                      : grad.dot(grad - prev_grad) / prev_grad.squaredNorm());
      d += w_prev * prev_grad;
    }

    alpha = FibonacciSearch(phi, a, b);
```

```
    x_next = x + alpha * d;

    if ((x_next - x).norm() < delta &&
        std::abs(f.At(x_next) - f.At(x)) < epsilon) {
      spdlog::debug("||x_next - x|| < delta && |f(x_next) - f(x)| < epsilon");
      return x_next;
    }

    x = std::move(x_next);
    prev_grad = std::move(grad);
    prev_d = std::move(d);
  }

  return x;
}

Eigen::VectorXd DavidonFletcherPowell(const IMultivariateFunction& f,
                                      const Eigen::VectorXd& x0,
                                      const std::size_t max_iterations,
                                      const double grad_epsilon,
                                      const double delta, const double epsilon,
                                      const double a, const double b) {
  Eigen::VectorXd x = x0, x_next;
  Eigen::VectorXd grad = f.Gradient(x), grad_next;
  Eigen::VectorXd d;

  const auto phi = [&](const double alpha) {
    return f.At(x + alpha * d);
  };

  const auto n = f.Size();
  Eigen::MatrixXd g(n, n);
  g.setIdentity();

  for (std::size_t k = 0; k < max_iterations; ++k) {
    spdlog::debug("Iteration {}, x = ({}, {}), f(x)={}", k, x.x(), x.y(), f.At(x));
    if (grad.norm() < grad_epsilon) {
      spdlog::debug("||grad|| < epsilon");
      return x;
    }
```

```
    d = -g * grad;
    const auto alpha = FibonacciSearch(phi, a, b);
    x_next = x + alpha * d;

    if ((x_next - x).norm() < delta &&
        std::abs(f.At(x_next) - f.At(x)) < epsilon) {
      spdlog::debug("||x_next - x|| < delta && |f(x_next) - f(x)| < epsilon");
      return x_next;
    }

    const Eigen::VectorXd delta_x = x_next - x;
    grad_next = f.Gradient(x_next);
    const Eigen::VectorXd delta_grad = grad_next - grad;
    const Eigen::VectorXd w1 = delta_x;
    const Eigen::VectorXd w2 = g * delta_grad;
    const double sigma1 = 1 / w1.dot(delta_grad);
    const double sigma2 = -1 / w2.dot(delta_grad);
    g += sigma1 * w1 * w1.transpose() + sigma2 * w2 * w2.transpose();

    x = std::move(x_next);
    grad = std::move(grad_next);
  }

  return x;
}

Eigen::VectorXd LevenbergMarquardt(const IMultivariateFunction& f,
                                   const Eigen::VectorXd& x0,
                                   const std::size_t max_iterations,
                                   const double epsilon) {
  auto x = x0;
  auto mu = 1e+4;
  const auto n = f.Size();

  for (std::size_t k = 0; k < max_iterations; ++k) {
    spdlog::debug("Iteration {}, x = ({}, {}), f(x)={}", k, x.x(), x.y(), f.At(x));
    const auto grad = f.Gradient(x);
    if (grad.norm() < epsilon) {
      spdlog::debug("||grad|| < epsilon");
```

```
      return x;
    }

    const auto f_x = f.At(x);
    x -= (f.Hessian(x) + mu * Eigen::MatrixXd::Identity(n, n)).inverse() * grad;
    if (f.At(x) < f_x) {
      mu /= 2;
    } else {
      mu *= 2;
    }
  }

  return x;
}

}  // namespace

int main() {
  spdlog::set_level(spdlog::level::level_enum::debug);

  const auto f = RosenbrockFunction();
  const auto x0 = Eigen::Vector<double, 2>{100, 100};

  constexpr std::size_t kMaxIterations = 100;
  constexpr auto kDelta = 1e-10;
  constexpr auto kEpsilon = 1e-9;
  constexpr auto kGradientEpsilon = 1e-9;
  constexpr auto kA = -10.0;
  constexpr auto kB = 10.0;

  auto u = GradientDescent(f, x0, kMaxIterations, kGradientEpsilon, kDelta,
                           kEpsilon, kA, kB);
  spdlog::info("Gradient descent: x=({}, {}), f(x)={}", u.x(), u.y(),
               f.At(u));

  u = FletcherReeves(f, x0, kMaxIterations, kGradientEpsilon, kDelta, kEpsilon,
                     kA, kB);
  spdlog::info("Fletcher-Reeves: x=({}, {}), f(x)={}", u.x(), u.y(),
               f.At(u));

```

```
    u = PolakRibier(f, x0, kMaxIterations, kGradientEpsilon, kDelta, kEpsilon, kA,
                    kB);
    spdlog::info("Polak-Ribier: x=({}, {}), f(x)={}", u.x(), u.y(),
                 f.At(u));

    u = DavidonFletcherPowell(f, x0, kMaxIterations, kGradientEpsilon, kDelta,
                              kEpsilon, kA, kB);
    spdlog::info("Davidon-Fletcher-Powell: x=({}, {}), f(x)={}", u.x(), u.y(),
                 f.At(u));

    u = LevenbergMarquardt(f, x0, kMaxIterations, kGradientEpsilon);
    spdlog::info("Levenberg-Marquardt: x=({}, {}), f(x)={}", u.x(), u.y(),
                 f.At(u));
}
```

# 4 Результаты работы методов

Рассматривается функция $f(x) = 250(x_1^2 - x_2)^2 + 2(x_1 - 1)^2 + 50$. Глобальный минимум функции $f(x)$ достигается в точке $(1, 1)$, и равен $50$.

Стартовой точкой методов многомерного поиска выбрана точка $(100, 100)$. Построение последовательности $\{x^k\}$, $k = 0, 1, \ldots$, заканчивается в точке $x^k$, если $||\nabla f(x^k)|| \leq \varepsilon_1$, где $\varepsilon_1 = 10^{-9}$, или при одновременном выполнении неравенств $||x^{k+1} - x^k|| < \delta$, $|f(x^{k+1}) - f(x^k)| < \varepsilon_2$, где $\delta = 10^{-9}$ и $\varepsilon_2 = 10^{-9}$, или если достигается предельное число итераций $100$.

В качестве метода одномерного поиска используется поиск Фибоначчи на отрезке $[-10, 10]$.

## 4.1 Метод наискорейшего градиентного спуска

Метод завершает работу при достижении предельного числа итераций в точке $x \approx (10.029, 100.534)$, $f(x) \approx 213.065$. Результаты последних 10 итераций:

```
Iteration 89, x = (10.03148807122927, 100.63172017630485), f(x) = 213.1357874561481
Iteration 90, x = (-10.013013775454562, 100.26484032873051), f(x) = 292.5777748566868
Iteration 91, x = (-10.012813057598942, 100.25387819301962), f(x) = 292.56572485538834
Iteration 92, x = (10.030816294660914, 100.62088009295822), f(x) = 213.1145341013168
```

Iteration 93, x = (10.03085179084519, 100.61896304568614), f(x) = 213.11280598565742
Iteration 94, x = (-10.012165216528189, 100.24784778703183), f(x) = 292.54039553811526
Iteration 95, x = (-10.011961170187258, 100.23678868964839), f(x) = 292.5282388696106
Iteration 96, x = (10.030103267247357, 100.6065760093384), f(x) = 213.08877806311668
Iteration 97, x = (10.030138416760341, 100.60463698759254), f(x) = 213.08703020927487
Iteration 98, x = (-10.011839631207621, 100.2413283517329), f(x) = 292.526054343388
Iteration 99, x = (-10.011643535379804, 100.23050248101643), f(x) = 292.5141539522706

## 4.2 Метод Флетчера-Ривза

Метод завершает работу при достижении предельного числа итераций в точке $x \approx (-0.926, 0.864)$, $f(x) \approx 57.428$. Результаты последних 10 итераций:

Iteration 89, x = (-4.08412185948198, 16.68654607052422), f(x) = 101.70713547026128
Iteration 90, x = (3.7871653724492456, 14.345454132843685), f(x) = 65.53858749642693
Iteration 91, x = (3.770473569803468, 14.198729558724722), f(x) = 65.42973675955056
Iteration 92, x = (-0.1589940079574168, 0.027641936171113102), f(x) = 52.68792997607468
Iteration 93, x = (0.02932280121196934, -0.022889320348931187), f(x) = 52.025433944536545
Iteration 94, x = (0.0120311161498119, -0.010681364705851234), f(x) = 51.98146620867051
Iteration 95, x = (0.05106332672405491, 0.01578724182183233), f(x) = 51.8443882600091
Iteration 96, x = (0.050137085153883425, 0.003071311900725104), f(x) = 51.80455683914451
Iteration 97, x = (0.42951984809284094, 0.17336087010422682), f(x) = 50.68184456747179
Iteration 98, x = (-0.9281914600700055, 0.8730420131046611), f(x) = 57.46892221780569
Iteration 99, x = (-0.9283318868338613, 0.8619916121493209), f(x) = 57.4369369015416

## 4.3 Метод Полака-Рибьера

Метод завершает работу при достижении предельного числа итераций в точке $x \approx (1.005, 1.009)$, $f(x) \approx 50$. Результаты последних 10 итераций:

Iteration 89, x = (1.005633730653852, 1.011307936620449), f(x) = 50.000063496923296
Iteration 90, x = (1.0056302314589185, 1.0113098232318996), f(x) = 50.00006347698859
Iteration 91, x = (1.0043009072222773, 1.0088441122433192), f(x) = 50.0000495172154
Iteration 92, x = (1.0046386606693454, 1.009411457259613), f(x) = 50.000046205091216
Iteration 93, x = (1.0046763847195765, 1.009388999323969), f(x) = 50.0000437887099
Iteration 94, x = (1.0045945627761548, 1.009272728661987), f(x) = 50.000043196361176
Iteration 95, x = (1.0046125853667958, 1.009260045808578), f(x) = 50.000042598121624
Iteration 96, x = (1.0045875577058334, 1.0092340216735158), f(x) = 50.00004244972722
Iteration 97, x = (1.0045961353827297, 1.0092257724546962), f(x) = 50.00004228721986
Iteration 98, x = (1.0045874347221277, 1.0092210134289983), f(x) = 50.00004224660997
Iteration 99, x = (1.004591481057644, 1.009213616254047), f(x) = 50.00004218630451

## 4.4 Метод Девидона-Флетчера-Пауэлла

Метод завершает работу при достижении предельного числа итераций в точке $x \approx (1, 1)$, $f(x) = 50$. Результаты последних 10 итераций:

Iteration 89, x = (0.9999999867851308, 0.9999999735167926), f(x) = 50
Iteration 90, x = (1.000000041995718, 1.0000000084161353), f(x) = 50
Iteration 91, x = (0.9999999957367549, 0.9999999914562606), f(x) = 50
Iteration 92, x = (0.9999999772863342, 0.9999999544807684), f(x) = 50
Iteration 93, x = (1.0000000072181758, 1.0000000144655563), f(x) = 50
Iteration 94, x = (0.9999999983678916, 0.9999999967291797), f(x) = 50
Iteration 95, x = (0.9999999687412185, 0.9999999373559662), f(x) = 50
Iteration 96, x = (0.9999999992217837, 0.9999999984404188), f(x) = 50
Iteration 97, x = (0.9999999786008551, 0.9999999571151265), f(x) = 50
Iteration 98, x = (1.0000000068004409, 1.0000000136283973), f(x) = 50
Iteration 99, x = (1.0000000217681058, 1.0000000436242877), f(x) = 50

## 4.5 Метод Левенберга-Марквардта

Метод завершает работу в точке $x \approx (1, 1)$, $f(x) = 50$ с выполнением условия $||\nabla f(x)|| < \varepsilon_1$. Результаты последних 10 итераций:

Iteration 25, x = (19.171649934879973, 367.55120699842826), f(x) = 710.4179503490125
Iteration 26, x = (8.335900, -47.925231), f(x) = 3446579.7627404225
Iteration 27, x = (8.335611004239171, 69.48185088949523), f(x) = 157.62245598989048
Iteration 28, x = (2.3380938952157795, -30.503290027030715), f(x) = 323513.32201214595
Iteration 29, x = (2.33793396318421, 5.465763676818462), f(x) = 53.58014193612435
Iteration 30, x = (1.0633452377864907, -0.4938591153397933), f(x) = 709.8086188294966
Iteration 31, x = (1.0631845954627233, 1.1303537133729828), f(x) = 50.00798460130335
Iteration 32, x = (1.0002259021977198, 0.9964883775294373), f(x) = 50.003927391325206
Iteration 33, x = (1.0001118823222457, 1.0002237597086907), f(x) = 50.00000002503538
Iteration 34, x = (1.000000042156586, 1.0000000719382793), f(x) = 50.00000000000004
Iteration 35, x = (1.0000000000070761, 1.0000000000141718), f(x) = 50

# 5 Вывод

Метод наискорейшего градиентного спуска не обнаруживает точку глобального минимума функции $f(x)$ (идёт застревание между «оврагами» функции); остальные методы достаточно близко обнаруживают эту точку. Среди методов сопряжённых градиентов лучший результат показывает метод Полака-Рибьера.

Квазиньютоновский метод и метод Левенберга-Марквардта обнаруживают минимум со сколь угодно большой точностью. Метод Левенберга-Марквардта справляется с задачей за наименьшее число итераций, однако в общем случае требует вычисление обратной матрицы Гессе, что занимает больше времени.