Университет ИТМО
Факультет программной инженерии и компьютерной техники

Вычислительная математика. Лабораторная работа №4. Аппроксимация функций методом наименьших квадратов

Группа: P32131
Студент: Смирнов Виктор Игоревич
Вариант: 17

2023

Ключевые слова

Аппроксимация, Метод Наименьших квадратов, МНК.

1 Цель работы

Цель работы - найти функцию, являющуюся наилучшим приближением заданной табличной функции по методу наименьших квадратов. А также был реализован программный модуль, предоставляющий API для аппроксимации вещественнозначных функций с одной переменной.

2 Вычислительная задача

Вычислительная задача была поставлена следующим образом: дана функция $f(x) = \frac{2x}{x^4+17}$, необходимо аппроксимировать ее линейной функцией и параболой на интервале $[0, 2]$, табулируя функцию с шагом 0.2.

Перед решением следует все же ввести читателя в курс дела и рассказать ему про МНК, не так ли? Поэтому отправим заинтересованного читателя ознакамливаться со статьей в Википедии.

На языке программирования C++ был реализован "Случай полиномиальной модели".

```
1
2  namespace Mathematica::Functional::Approx::LeastSqare {
3
4  using Mathematica::Collection::Array;
5
6  template <Abstract::Field F, Algebra::PolynomialDegree D, Count N>
7  Algebra::Linear::Matrix<F, D, D> buildMatrix(const Array<Point<F>, N>& points
8  ) noexcept {
9    const auto sums = Array<F, 2 * D - 1>([&points](auto i) {
10     auto sum = F::zero();
11     for (const auto point : points) {
12       sum += point.x.pow(i);
13     }
14     return sum;
15   });
16   return Algebra::Linear::Matrix<F, D, D>([&sums](auto i, auto j) {
17     return sums[i + j];
18   });
19 }
20
21 template <Abstract::Field F, Algebra::PolynomialDegree D, Count N>
22 Algebra::Linear::Vector<F, D> buildVector( //
23     const Array<Point<F>, N>& points
24 ) {
25   return Algebra::Linear::Vector<F, D>([&](auto i) {
26     auto value = F::zero();
27     for (const auto point : points) {
28       value += point.x.pow(i) * point.y;
29     }
30     return value;
31   });
32 }
33
34 template <Abstract::Field F, Algebra::PolynomialDegree D, Count N>
35 Algebra::Polynomial<F, D> optimalPolynomial(const Array<Point<F>, N>& points) {
36   const auto matrix = buildMatrix<F, D, N>(points);
37   const auto vector = buildVector<F, D, N>(points);
38   const auto gauss = Algebra::Linear::Eq::GaussSolver<F, D>();
39   const auto coefficients = gauss.solve({matrix, vector}).value;
40   return Algebra::Polynomial<F, D>(coefficients);
41 }
42
43 }
```

Листинг 1: Реализация метода наименьших квадратов на языке C++

С использованием данной функциональности напишем простенький тест для получения результатов вычисления:

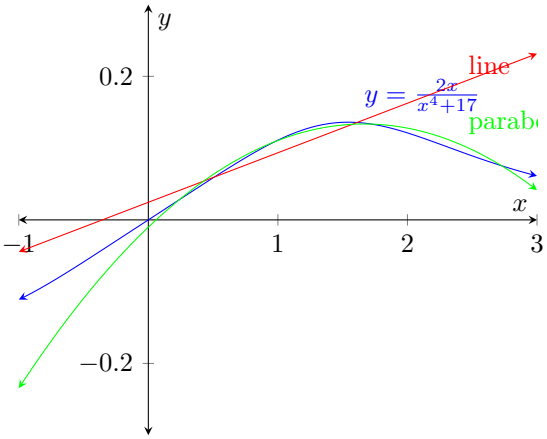
```cpp
#include "Mathematica/Functional/Approx/Model/Core.hpp"
#include "Mathematica/Functional/Approx/Model/Partition.hpp"
#include "Mathematica/Functional/Approx/Power.hpp"
#include "Mathematica/Functional/Exploration/Tabulate.hpp"
#include "Mathematica/Interop/Polynomial.hpp"
#include "Mathematica/Statistics/RSS.hpp"
#include "Mathematica/Statistics/Score.hpp"
#include "Symatica/Expression/DSL/Literals.hpp"
#include <gtest/gtest.h>

TEST(Approx, Playground) { // NOLINT
  using Mathematica::Interop::symbolic;
  using Symatica::Expression::DSL::var;

  using namespace Mathematica; // NOLINT
  using R = Mathematica::Abstract::Float<double>;
  using namespace Mathematica::Functional; // NOLINT
  constexpr auto MIN = 0;
  constexpr auto MAX = 2;
  constexpr auto FINENESS = 0.2;
  constexpr Mathematica::Count N = (MAX - MIN) / FINENESS;

  const auto scope = Interval<R>(MIN, MAX);
  const auto f = [](R x) -> R {              // NOLINT
    return (R(2) * x) / (x.pow(4) + R(17)); // NOLINT
  };

  const auto points = Exploration::Tabulate::trivial<R, N>(f, scope);
  for (const auto& point : points) {
    std::cout << "( " << point.x.asString() //
              << ", " << point.y.asString() //
              << " )" << std::endl;
  }

  const auto line = Approx::LeastSqare::optimalPolynomial<R, 2>(points);
  const auto sline = symbolic(line, var(1));
  const auto parabola = Approx::LeastSqare::optimalPolynomial<R, 3>(points);
  const auto sparabola = symbolic(parabola, var(1));

  const auto lineS = Statistics::standartDeviation(line, points);
  const auto parabolaS = Statistics::standartDeviation(parabola, points);

  const auto lineR2 = Statistics::Score::R2(line, points);
  const auto parabolaR2 = Statistics::Score::R2(parabola, points);
```

Листинг 2: Тест для решения вычислительной задачи

```
( 0.100000, 0.011765 )
( 0.300000, 0.035277 )
( 0.500000, 0.058608 )
( 0.700000, 0.081206 )
( 0.900000, 0.101948 )
( 1.100000, 0.119150 )
( 1.300000, 0.130942 )
( 1.500000, 0.135977 )
( 1.700000, 0.134111 )
( 1.900000, 0.126531 )
Line: (0.024521 + (0.069030 * $1))
Line: STD = 0.015459
Line: R2 = 0.868067
Parabola: ((-0.010176 + (0.172604 * $1)) + (-0.051787 * ($1 ^ 2)))
Parabola: STD = 0.003526
Parabola: R2 = 0.993138
```

Листинг 3: Результаты вывода программы для решения вычислительной задачи

Итак, мы получили прямую линейного тренда $y = 0.069030 \cdot x + 0.024521$ и параболу $y = -0.051787 \cdot x^2 + 0.172604 \cdot x + -0.010176$. Видно, что парабола имеет меньшее среднеквадратичное отклонение, что толкает нас сделать вывод о том, что кдвадратическая зависимость лучше приближает заданную функцию, в чем нас окончательно убеждают показатели R^2.



3 Программная реализация задачи

Далее представлю реализации различных алгоритмов аппроксимации функций на языке программирования C++. Стоит заметить, что все они так или иначе реализованы через методе наименьших квадратов.

3.1 Нахождение линии линейного тренда

```cpp
namespace Mathematica::Functional::Approx::Linear {

using Mathematica::Collection::Array;

template <Abstract::Field F>
class TrendLine {
public:
  explicit TrendLine(F slope, F bias) : _slope(slope), _bias(bias) {} // NOLINT
  TrendLine(const TrendLine& other) : TrendLine(other._slope, other._bias) {}

  F operator()(F x) const noexcept { return _slope * x + _bias; }

  [[nodiscard]] String asString() const noexcept {
    const auto a = slope().asString();
    const auto b = bias().asString();
    return a + " * x + " + b;
  }

  F slope() const noexcept { return _slope; }
  F bias() const noexcept { return _bias; }

private:
  // y = kx + b
  F _slope;
  F _bias;
};

template <Abstract::Field F, Count N>
TrendLine<F> trendLine(const Array<Point<F>, N>& points) noexcept {
  const auto poly = LeastSqare::optimalPolynomial<F, 2>(points);
  return TrendLine(poly[1], poly[0]);
}
```

Листинг 4: Нахождение линии линейного тренда

3.2 Нахождение линии логарифмического тренда

```cpp
namespace Mathematica::Functional::Approx::Logarithmic {

using Mathematica::Collection::Array;

template <Abstract::Real F>
struct Trend {
  F slope; // NOLINT
```

```
  F bias;  // NOLINT

  F operator()(F x) const noexcept { return slope * Abstract::ln(x) + bias; }

  [[nodiscard]] String asString() const noexcept {
    const auto a = slope.asString();
    const auto b = bias.asString();
    return a + " * ln(x) + " + b;
  }
};

template <Abstract::Real F, Count N>
Trend<F> train(const Array<Point<F>, N>& points) noexcept {
  // TODO: X = Ln X, Y = Y
  const auto line = Linear::trendLine(points.map([](auto point) {
    return Point(Abstract::ln(point.x), point.y);
  }));
  return {
      .slope = line.slope(),
      .bias = line.bias(),
  };
}

}
```

Листинг 5: Нахождение линии логарифмического тренда

3.3 Нахождение линии степенного тренда

```
namespace Mathematica::Functional::Approx::Power {

using Mathematica::Collection::Array;

template <Abstract::Real F>
struct Trend {
  F multiplier; // NOLINT
  F power;      // NOLINT

  F operator()(F x) const noexcept { return multiplier * x.pow(power); }

  [[nodiscard]] String asString() const noexcept {
    const auto a = multiplier.asString();
    const auto b = power.asString();
    return a + " * x ^ " + b;
  }
};

template <Abstract::Real F, Count N>
Trend<F> train(const Array<Point<F>, N>& points) noexcept {
  const auto line = Linear::trendLine(points.map([](auto point) {
    return Point(Abstract::ln(point.x), Abstract::ln(point.y));
  }));
  return {
      .multiplier = F::exp().pow(line.bias()),
      .power = line.slope(),
  };
}

}
```

Листинг 6: Нахождение линии степенного тренда

3.4 Нахождение линии экспоненциального тренда

```
namespace Mathematica::Functional::Approx::Exponential {

using Mathematica::Collection::Array;

template <Abstract::Real F>
struct Trend {
  F multiplier; // NOLINT
  F powerSlope; // NOLINT

```

```
  F operator()(F x) const noexcept {
    return multiplier * F::exp().pow(powerSlope * x);
  }

  [[nodiscard]] String asString() const noexcept {
    const auto a = multiplier.asString();
    const auto b = powerSlope.asString();
    return a + " * e ^ (" + b + " * x)";
  }
};


template <Abstract::Real F, Count N>
Trend<F> train(const Array<Point<F>, N>& points) noexcept {
  const auto line = Linear::trendLine(points.map([](auto point) {
    return Point(point.x, Abstract::ln(point.y));
  }));
  return {
      .multiplier = F::exp().pow(line.bias()),
      .powerSlope = line.slope(),
  };
}

}
```

Листинг 7: Нахождение линии экспоненциального тренда

3.5 Ядро моделей аппроксимации

```
namespace Mathematica::Functional::Approx::Model {

using Mathematica::Collection::Array;

template <Abstract::Real R, Count N>
class Model {
public:
  virtual Function<R> trainedOn(const Array<Point<R>, N>& points) = 0;
};

template <Abstract::Real R, Count N, typename Predictor>
class Algorithm : public Model<R, N> {
public:
  explicit Algorithm(const Mapping<Predictor(const Array<Point<R>, N>&)>& method)
      : method(method) {}

  Function<R> trainedOn(const Array<Point<R>, N>& points) override {
    const auto trend = method(points);
    return [trend](R x) { return trend(x); };
  }

private:
  Mapping<Predictor(const Array<Point<R>, N>&)> method;
};

}
```

Листинг 8: Ядро моделей аппроксимации

3.6 Модель, выбирающая лучшую из данных

```
namespace Mathematica::Functional::Approx::Model {

using Mathematica::Collection::Array;

template <Abstract::Real R, Count N>
using Score
    = Mapping<R(const Function<R>& predict, const Array<Point<R>, N>& points)>;

template <Abstract::Real R, Count N, Count M>
class ChoiceOptimal : public Model<R, N> {
public:
  explicit ChoiceOptimal(
      const Array<Model<R, N>*, M>& models, //
```

```
      const Score<R, N>& score
  )
      : models(models), score(score) {}

  Function<R> trainedOn(const Array<Point<R>, N>& points) override {
    const auto trends = Array<Function<R>, M>([*this, &points](auto i) {
      return models[i]->trainedOn(points);
    });

    const auto r2scores = Array<R, M>([*this, &trends, &points](auto i) {
      return score(trends[i], points);
    });

    const auto best = std::max_element(r2scores.begin(), r2scores.end());
    const auto bestIndex = best - r2scores.begin();
    return trends[bestIndex];
  }

private:
  Array<Model<R, N>*, M> models;
  Score<R, N> score;
};

}
```

Листинг 9: Модель, выбирающая лучшую из данных

3.7 Модель, приблищая функцию кусочно

```
namespace Mathematica::Functional::Approx::Model {

using Mathematica::Collection::Array;

template <Abstract::Real R, Count N, Count M>
class PartitionModel : public Model<R, N> {
public:
  explicit PartitionModel(Model<R, N / M>& model) : model(model) {}

  // points must be sorted by x
  // TODO: create class for such partial functions
  Function<R> trainedOn(const Array<Point<R>, N>& points) override {
    std::vector<std::pair<Interval<R>, Function<R>>> fragments;
    for (auto i = 0; i < M; i++) {
      const auto chosen = Array<Point<R>, N / M>([i, &points](auto j) {
        return points[N / M * i + j];
      });
      const auto fragment = model.trainedOn(chosen);
      fragments.emplace_back(Interval(chosen[0].x, chosen[N / M - 1].x), fragment);
    }
    return [fragments](R x) {
      for (const auto& [interval, fragment] : fragments) {
        if (interval.contains(x)) {
          return fragment(x);
        }
      }
      if (x <= fragments[0].first.left()) {
        return fragments[0].second(x);
      }
      return fragments[fragments.size() - 1].second(x);
    };
  }

private:
  Model<R, N / M>& model;
};

}
```

Листинг 10: Модель, приблищая функцию кусочно

4 Вывод

Выполнив данную лабораторную работу я познакомился с базовыми методами аппроксимации функций. По сути все рассмотренные методы являются формой метода наименьших квадратов (МНК). МНК позволяет достаточно точно приближать функции разной природы, кажется, что эффективность построенного полинома растет пропорционально его степени. Также хорошо показывает себя метод кусочной аппроксимации функции.

Список литературы

[1] Б.П. Демидович, И.А. Марон Основы вычислительной математики: учебное пособие — 1966 год.

[2] Лекции Татьяны Алексеевны Малышевой