ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ И ТЕЛЕКОММУНИКАЦИИ
INFORMATION TECHNOLOGY AND TELECOMMUNICATIONS

УДК 004. 421

DOI: 10.21822/2073-6185-2023-50-4-37-50 Оригинальная статья/Original article

Построение параллельных одномерных интерполяторов на C++

Ф.В. Абилова[1], М.В. Абилов[1], Э.В. Селимханов[2]
[1]Дагестанский государственный технический университет,
[1]367026, г. Махачкала, пр. И. Шамиля, 70, Россия,
[2]ООО «РН-БашНИПИНефть»,
[2]450006, г. Уфа, ул. Ленина, д. 86/1, Россия

Резюме. Цель. Целью работы является исследование и разработка интерполяторов на языке программирования C++, включая линейный, квадратичный и кубический интерполяторы, а также одномерный RBF-интерполятор. Основными задачами являются использование библиотеки GSL, интерполяционного многочлена Лагранжа, OpenMP, и сравнительный анализ с библиотекой SciPy. Эксперименты направлены на оценку эффективности и применимости различных методов интерполяции. **Методы**. Используются библиотеки GSL и Eigen для реализации интерполяторов и оптимизации вычислительных процессов. Для сравнения производительности применяются линейный, квадратичный и кубический интерполяторы, а также разрабатывается одномерный RBF-интерполятор. Метод интерполяции Лагранжа и параллельные вычисления с использованием OpenMP и SIMD-инструкций также используются для повышения эффективности. **Результат**. Результаты исследования включают в себя успешную разработку и реализацию различных методов интерполяции на языке C++. Особое внимание уделяется анализу производительности и точности каждого метода. Путем сравнительного анализа с библиотекой SciPy выявлены преимущества и недостатки различных интерполяторов. Основной результат - практическая применимость этих методов в контексте конкретных задач интерполяции. **Вывод.** Реализация интерполяторов на языке C++ обладает некоторыми значительными преимуществами по сравнению с использованием библиотеки SciPy. C++ позволяет более точное и быстрое управление вычислениями, что особенно важно в задачах, связанных с численными методами интерполяции. Библиотеки GSL и Eigen предоставляют мощные инструменты для оптимизации и высокопроизводительных вычислений, что позволяет добиваться высокой эффективности при реализации интерполяции на C++.

Ключевые слова: интерполяция, C++, GSL, радиально-базисные функции, Eigen, OpenMP, SIMD, интерполяционный многочлен Лагранжа, SciPy

Для цитирования. Ф.В. Абилова, М.В. Абилов, Э.В. Селимханов. Построение параллельных интерполяторов на C++. Вестник Дагестанского государственного технического университета. Технические науки. 2023; 50(4):37-50. DOI:10.21822/2073-6185-2023-50-4-37-50

Construction of parallel one-dimensional interpolators in C++

F.V. Abilova[1], M.V. Abilov[1], E.V. Selimkhanov[2]
[1]Daghestan State Technical University,
[1] 70 I. Shamil Ave., Makhachkala 367026, Russia,
[2] LLC “RN-BashNIPIneft”,
[2] 86/1 Lenina St., Ufa 450006, Russia

Abstract. **Objective**. The purpose of this article is to research and develop interpolators in the C++ programming language, including linear, quadratic and cubic interpolators, as well as a one-dimensional RBF interpolator. The main tasks are the use of the GSL library, the Lagrange interpolation polynomial, OpenMP, and comparative analysis with the SciPy

library. The experiments are aimed at evaluating the effectiveness and applicability of various interpolation methods. **Method**. The work uses the GSL and Eigen libraries to implement interpolators and optimize computational processes. Linear, quadratic, and cubic interpolators are used to compare performance, and a one-dimensional RBF interpolator is being developed. The Lagrange interpolation method and parallel computing using OpenMP and SIMD are also used to improve efficiency. **Result**. The results of the research include the successful development and implementation of various interpolation methods in C++. Particular attention is paid to the analysis of the performance and accuracy of each method. Through a comparative analysis with the SciPy library, the authors identify the advantages and disadvantages of various interpolators. The main result is the practical applicability of these methods in the context of specific interpolation problems. **Conclusion**. The study made it possible to make sure that the implementation of interpolators in the C ++ language has some significant advantages compared to using the SciPy library. In particular, C++ allows for more precise and faster control over calculations, which is especially important in tasks related to numerical interpolation methods. The GSL and Eigen libraries provide powerful tools for optimization and high performance computing, which allows you to achieve high efficiency when implementing interpolation in C ++.

Keywords: Interpolation, C++, GSL, Radial Basis Functions, Eigen, OpenMP, SIMD, Lagrange Interpolation Polynomial, SciPy

For citation. F.V. Abilova, M.V. Abilov, E.V. Selimkhanov. Construction of parallel one-dimensional interpolators in C++. Herald of Daghestan State Technical University. Technical Sciences. 2023; 50(4):37-50. DOI:10.21822/2073-6185-2023-50-4-37-50

Введение. Интерполяция играет ключевую роль в множестве приложений, начиная от визуализации данных до численных методов в науке и инженерии. Эффективная интерполяция имеет решающее значение для точности и производительности таких приложений. В данной статье мы исследуем создание одномерных интерполяторов на C++, включая линейные, квадратичные, кубические и радиально-базисные функции (RBF) интерполяторы. Чтобы достичь максимальной производительности, мы воспользуемся мощными библиотеками и технологиями, такими как OpenMP, включая SIMD (парадигма “single instruction, multiple data”) инструкции, и библиотека Eigen для работы с линейной алгеброй. Эти средства позволят нам оптимизировать интерполяторы и улучшить их многозадачность, что особенно актуально в современных многозадачных вычислительных средах. Особое внимание уделяется сравнению производительности наших C++ интерполяторов с аналогичными интерполяторами, реализованными на языке Python [1,9].

Постановка задачи. Наша цель – реализовать и продемонстрировать значительное преимущество в скорости, которое может быть достигнуто с помощью оптимизированных алгоритмов на C++ в сравнении с интерполяцией на Python (библиотека Scipy).

Задачи, поставленные в данной статье, включают в себя:

1. **Реализация интерполяторов**: Разработка и реализация интерполяторов, основанных на различных методах, включая линейную, квадратичную, кубическую интерполяцию и RBF-интерполяцию. Все интерполяторы будут созданы с использованием языка программирования C++.
2. **Оптимизация с OpenMP и SIMD**: Применение технологии OpenMP и SIMD для оптимизации вычислений в интерполяторах. Это включает в себя распараллеливание [10] и векторизацию вычислений [11] для увеличения производительности на многоядерных [12] системах.
3. **Использование библиотеки Eigen**: Внедрение библиотеки Eigen для выполнения операций линейной алгебры, что может значительно улучшить эффективность и устойчивость интерполяции.
4. **Сравнение с Python**: Сравнение производительности наших C++ интерполяторов с

аналогичными интерполяторами, реализованных в специализированных библиотеках, таких как SciPy на языке Python, с целью продемонстрировать преимущества высокооптимизированных алгоритмов на C++.

5. **Оценка результатов**: Проведение тестовых экспериментов для оценки точности и производительности различных интерполяторов на разных типах данных и объемах данных.

Методы исследования. Для построения линейных и кубических интерполяторов мы использовали библиотеку GSL (GNU Scientific Library) и её методы интерполяции.

Общая схема использования интерполятора выглядит следующим образом. Например, построение линейного интерполятора происходит командой

linear_interpolator interp(xp, yp);,

где «xp» и «yp» – исходные наборы иксов и игреков.

Далее, использование построенного интерполятора для искомого набора точек «x» выглядит так auto y = interp(x).

Рассмотрим шаблонный класс [1], реализующий линейную интерполяцию методами GSL, являющий по сути удобной C++-оберткой, инкапсулирующий в себе C-интерфейс библиотеки.

```
template<class T, class Value = T::value_type>
class linear_interpolator {
public:
   linear_interpolator(T xp, T yp) :
         xp_(std::move(xp)), yp_(std::move(yp)),
         interp(gsl_spline_alloc(gsl_interp_linear, xp_.size())) {
      gsl_spline_init(interp, xp_.data(), yp_.data(), xp_.size());
   }
   linear_interpolator(const linear_interpolator &) = delete;
   linear_interpolator &operator=(const linear_interpolator &) = delete;
   linear_interpolator(linear_interpolator &&) noexcept = delete;
   linear_interpolator &operator=(linear_interpolator &&) noexcept = delete;
   ~linear_interpolator() {
      gsl_spline_free(interp);
   }
   auto operator()(const T &x) const -> T {
      T y(x.size());
      bool omp_opt = x.size() >= 10000;
      auto num_t = max_threads / 2 > 8 ? 8 : max_threads / 2;
#pragma omp parallel for simd if(omp_opt) num_threads(num_t)
      for (size_t i = 0; i < x.size(); ++i) {
         y[i] = interp_(x[i]);
      }
      return y;
   }
private:
   auto interp_(Value xi) const -> Value {
      return gsl_spline_eval(interp, xi, nullptr);
   }
   T xp_;
   T yp_;
   gsl_spline *interp;
};
```

Шаблонный [15] тип T представляет собой тип стандартного контейнера [13] с поддержкой индексации (оператор «[]»), итерируемости и внутреннем «type-trait» [14] value_type. В качестве T могу выступать, например, std::vector [13], std::array [14] или Eigen::ArrayX. Тип Value представляет собой обобщенный тип элементов контейнера. Для данного метатипа достаточно поддержки следующего концепта [16] is_numeric:

```
template<class T>
concept is_numeric = std::is_integral_v<T> || std::is_floating_point_v<T>;
```

Конструктор [1] данного класса получает по значению два контейнера, после чего перемещает их в своё внутреннее состояние - поля T xp_ и T yp_ соответственно. Далее, в списке инициализации приватному полю interp типа gsl_spline * присваивается значение функции gsl_spline_alloc, что означает построения интерполяционного объекта с линейным методом интерполяции. После чего мы инициализируем и заполняем данными этот объект данным в теле конструктора с помощью функции gsl_spline_init, получая готовый интерполятор. Для данного класса из «rule of five» удалено всё, кроме деструктора во избежание неопределенного поведения после копирования или перемещения внутренних полей класса. В деструкторе вызывается функция gsl_spline_free, высвобождающая ресурсы переданного ей gsl-интерполятора. В приватном методе interp_ вызывается функция gsl_spline_eval непосредственно вычисляющая интерполированное значение в указанной точке xi для данного набора xp и yp. В перегруженном операторе «()» происходит создание результирующего набора точек «y» и заполнение его в цикле относительно входного набора точек «x».

Остановимся подробнее на цикле и директивах «pragma». Директива #pragma omp parallel for simd [5] – это директива препроцессора OpenMP(Open Multi-Processing), которая создает параллельную область выполнения, разбивая цикл на несколько одновременно выполняющихся во времени подциклов, что позволяет достичь существенного прироста в скорости исполнения алгоритма. Переменная max_threads это максимально количество потоков процессора (можно получить с помощью omp_get_max_threads() подключив заголовочный файл <omp.h> или воспользовавшись функцией стандартной библиотеки std::thread::hardware_concurrency [17] обе функции возвращают run-time значение). С помощью нее и условной части препроцессора num_threads() можно тонко настроить распараллеливание. В данном примере можно не опасаться гонки данных, т.к. запись идет в разные участки памяти независимо друг от друга. simd в данном случае обеспечивает параллелизм на уровне данных, т.е. процессор может выполнять несколько вычислений одновременно.

Шаблонный класс, реализующий кубическую интерполяцию строится аналогично, за исключение того, что вместо указателя gsl_interp_linear в функции gsl_spline_alloc будет использоваться gsl_interp_cspline. Алгоритмическая сложность [18] вычисления интерполяции в данной точке не превосходит O(n), хотя, конечно, в кубическом случае на практике вычисления занимают большее время. Алгоритмическую сложность вычисления интерполяции множества точек можно оценить, как сложность вычисления в одной точке помноженную на количество точек интерполяции. Для случая квадратической интерполяции построим собственный алгоритм. Для этого нам потребуется ряд дополнительный ряд шаблонных функций и переменных. Два концепта определят принадлежность данного типа к типу std::vector и Eigen::ArrayX соответственно

```
template<class Container>
concept is_vector = std::is_same_v<Container, std::vector<typename Container::value_type>>;
template<class Container>
concept is_eigen_array = std::is_same_v<Container, Eigen::ArrayX<typename Container::value_type>>;
```

Далее, положим

```
template<class T>
concept Floating = std::is_floating_point_v<T>;
template<Floating T = double>
constexpr auto nan = std::numeric_limits<T>::quiet_NaN();
constexpr int Precision = 8;
template<int Prec = 0, Floating T = double>
auto eps = (Prec == 0) ? (std::numeric_limits<T>::epsilon()) :

std::pow(10.0, Prec);
```

где Precision это количество значащих цифр после запятой у floating типа.

Напишем две обобщенные функции, которые будут делать реверс std::vector или Eigen::ArrayX соответственно, используя полиморфизм времени компиляции [1] на ранее определенных концептах

```
template<is_vector Container>
auto reverse(const Container &cont) {
 Container copy = cont;
  std::reverse(copy.begin(), copy.end());
   return copy;
}
template<is_eigen_array Container>
auto reverse(const Container &cont) {
   auto ref = cont.reverse();
   return ref;
}
```

Причем, функция для Eigen::ArrayX работает «на месте» [14], без внутренних копирований. Также определим ряд функций для корректного сравнения floating типов с учетом Precision

```
constexpr auto dbl_eq(double lhs, double rhs, double eps_ = eps<-Precision>) -> bool {
   return std::abs(lhs - rhs) < eps_;
}
constexpr auto dbl_less(double lhs, double rhs, double eps_ = eps<Precision>) -> bool {
   return rhs - lhs > eps_;
}
constexpr auto dbl_greater(double lhs, double rhs,
double eps_ = eps<-Precision>) -> bool {
   return lhs - rhs > eps_;
}
constexpr auto dbl_less_eq(double lhs, double rhs,
double eps_ = eps<-Precision>) -> bool {
   return std::abs(lhs - rhs) < eps_ || rhs - lhs > eps_;
}
constexpr auto dbl_greater_eq(double lhs, double rhs,
double eps_ = eps<-Precision>) -> bool {
   return std::abs(lhs - rhs) < eps_ || lhs - rhs > eps_;
}
```

Далее, определим функцию upper_bound_with_guess, которая работает как функция из стандартной библиотеки C++ std::upper_bound [13], т.е производит поиск в отсортированном массиве, но с улучшением скорости выполнения благодаря вычислению «предсказания» искомого элемента

```
template<class T, class V, class I>
auto upper_bound_with_guess(V key, const T &data, I guess) -> I {
   static I IN_CACHE = 8;
   auto size = data.size();
   I i_min = 0;
   I i_max = size;
   if (size <= 4) {
   auto it = std::upper_bound(data.begin(), data.end(), key);
   return it == data.end() ? data.size() - 1 : it - data.begin();
   }
   if (guess > size - 3) {
      guess = size - 3;
   }
   if (guess < 1) {
      guess = 1;
   }

   if (dbl_less(key, data[guess])) {
      if (dbl_less(key, data[guess - 1])) {
```

```
            i_max = guess - 1;
            if (guess > IN_CACHE && dbl_greater_eq(key, data[guess - IN_CACHE])) {
               i_min = guess - IN_CACHE;
            }
          } else {
            return guess - 1;
          }
        } else {
          if (dbl_less(key, data[guess + 1])) {
            return guess;
          } else {
            if (dbl_less(key, data[guess + 2])) {
              return guess + 1;
            } else {
              i_min = guess + 2;
              if (guess < size - IN_CACHE - 1 &&
                 dbl_less(key, data[guess + IN_CACHE])) {
                 i_max = guess + IN_CACHE;
              }
            }
          }
        }
        while (i_min < i_max) {
          const I i_mid = i_min + ((i_max - i_min) >> 1);
          if (dbl_greater_eq(key, data[i_mid])) {
            i_min = i_mid + 1;
          } else {
            i_max = i_mid;
          }
        }
        return i_min - 1;
      }
```

Тип T – это тип контейнера, V – тип значения, I – тип ключа. Данная функция хорошо себя ведет при поиске чисел из отсортированного набора чисел. Набор исходных чисел, очевидно, должен быть отсортирован, т.к. данная функция является усложненным вариантов бинарного поиска. Функция вычисления квадратичной интерполяции в точке будет основываться на интерполяционном многочлене Лагранжа второй степени [2]

$$L^2(x) = \sum_{i=1}^{n-1} P_i^2(x), \tag{1}$$

где

$$P_i^2(x) = a_i x^2 + b_i x + c_i \tag{2}$$

Для случая равноотстоящих узлов для набора точек $(x_0, y_0), (x_1, y_1), (x_2, y_2)$ можно явно получить выражение полинома

$$P(x) = \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} y_0 + \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)} y_1 + \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)} y_2, \tag{3}$$

где x – текущая точка.

Разобьём саму функцию интерполяции на вспомогательную и основную.

Основная функция с использованием reverse будет иметь вид

```
template<class T, class R, class V>
auto quadratic(const T &x, const T &xp, const T &yp, bool bounds_error, std::optional<std::pair<V, V>> fill_values,
bool extrapolate
) -> R {
  auto x_reversed = x.size() > 1 && x[1] < x[0];
```

```
    auto xp_reversed = xp.size() > 1 && xp[1] < xp[0];
    if (x_reversed && xp_reversed) {
        return reverse(quadratic_impl<R, V>(
                reverse(x),
                reverse(xp),
                reverse(yp),
                bounds_error,
                fill_values,
                extrapolate));
    } else if (x_reversed) {
        return reverse(detail::quadratic_impl<R, V>(reverse(x), xp, yp, bounds_error, fill_values, extrapolate));
    } else if (xp_reversed) {
      return detail::quadratic_impl<R, V>(x, reverse(xp), reverse(yp), bounds_error, fill_values, extrapolate);
    } else {
        return detail::quadratic_impl<R, V>(x, xp, yp, bounds_error, fill_values, extrapolate);
    }
}
```

Функция reverse дает возможность интерполяции набора точек, отсортированных по убыванию. Тогда реализация квадратичной интерполяции с использованием upper_bound_ with_guess и данных выше формул будет иметь следующий вид

```
template<class R, class V, class T1, class T2>
auto quadratic_impl(T1 x, T2 xp, T2 yp, bool bounds_error,
std::optional<std::pair<V, V>> fill_values, bool extrapolate
) -> R {
R r(x.size());
auto nan = nan<V>;
auto xp_sz = xp.size();
auto yp_sz = yp.size();
auto x_sz = x.size();
auto k = 0;
for (size_t i = 0; i < x_sz; ++i) {
    auto xi = x[i];
    if (dbl_less(xi, xp[0])) {
        if (extrapolate && xp_sz > 2) {
            double x1 = xp[0], x2 = xp[1], x3 = xp[2];
            double y1 = yp[0], y2 = yp[1], y3 = yp[2];
            r[i] = y1 + (xi - x1) * ((y1 - y2) / (x1 - x2)) +
                (xi - x1) * (xi - x2) * ((y1 - 2 * y2 + y3) / ((x1 - x2) * (x1 - x3)));
        } else if (!bounds_error) {
            r[i] = fill_values ? fill_values->first : nan;
        } else {
            throw std::range_error("A value in x is below the interpolation range.");
        }
    } else if (dbl_greater(xi, xp[xp_sz - 1])) {
        if (extrapolate && xp_sz > 2) {
            double x1 = xp[xp_sz - 3], x2 = xp[xp_sz - 2], x3 = xp[xp_sz - 1];
            double y1 = yp[yp_sz - 3], y2 = yp[yp_sz - 2], y3 = yp[yp_sz - 1];
            r[i] = y3 + (xi - x3) * ((y3 - y2) / (x3 - x2)) +
            (xi - x3) * (xi - x2) * ((y3 - 2 * y2 + y1) / ((x3 - x2) * (x3 - x1)));
        } else if (!bounds_error) {
                r[i] = fill_values ? fill_values->second : nan;
        } else {
            throw std::range_error("A value in x is above the interpolation range.");
        }
    } else {
        if (xp_sz == 1) {
            r[i] = yp[0];
        } else {
            k = upper_bound_with_guess(xi, xp, k);
```

```
                k = k == 0 ? 1 : k;
                double x1 = xp[k - 1], x2 = xp[k], x3 = xp[k + 1];
                double y1 = yp[k - 1], y2 = yp[k], y3 = yp[k + 1];
                auto val1 = y1 / ((x1 - x2) * (x1 - x3));
                auto val2 = y2 / ((x2 - x1) * (x2 - x3));
                auto val3 = y3 / ((x3 - x1) * (x3 - x2));
               r[i] = (xi - x2) * (xi - x3) * val1 + (xi - x1) * (xi -     x3) * val2 + (xi - x1) * (xi - x2) * val3;
              }
            }
          }
          return r;
          }
```

Данная функция имеет дополнительную возможность экстраполяции при включенном флаге extrapolate и отключенном флаге bounds_error. При отключенной экстраполяции значения за пределами входного диапазона заполняются значениями из fill_values. Теперь построим сам интерполятор, используя функции и определения данные выше

```
template<class T, class R = T, class Value = typename T::value_type>
class quadratic_interpolator {
public:
   quadratic_interpolator(T xp, T yp, bool bounds_error = true,
std::optional<std::pair<Value, Value>>          fill_values = std::nullopt,
                       bool extrapolate = false
                ) : xp_(std::move(xp)), yp_(std::move(yp)), bounds_error(bounds_error), fill_values(fill_values),
extrapolate(extrapolate) {
                }
   quadratic_interpolator(const quadratic_interpolator &) = delete;
   quadratic_interpolator(quadratic_interpolator &&) noexcept = delete;
   T &operator=(const quadratic_interpolator &) = delete;
   T &operator=(quadratic_interpolator &&) noexcept = delete;
   auto operator()(const T &x) const -> T {
      return quadratic(x, xp_, yp_, bounds_error, fill_values, extrapolate);
   }
private:
   T xp_;
   T yp_;
   bool bounds_error;
   std::optional<std::pair<Value, Value>> fill_values;
   bool extrapolate;
};
```

Принцип построения и использования квадратичного интерполятора схож со случаями линейной и кубической интерполяции, за исключением использования функции quadratic в перегруженном операторе «()». Ограничения на типе контейнера и внутреннем типа элементов остаются прежними. На все приведенные выше интерполяторы накладываются следующие ограничения. Наборы входных данных xp и yp должны быть одного размера, набор xp должен быть строго возрастающим (или также строго убывающим в случае квадратической интерполяции). Данные для интерполяции – набор чисел x в линейном случае должен содержать по крайней мере 2 различные точки, в квадратическом случае – 3, в кубическом – 4.

Далее рассмотрим построение RBF-интерполятора. RBF (Radial Basis Functions) - это мощное и универсальное средство для построения интерполяций и аппроксимаций для нерегулярных наборов исходных данных. Радиально-базисная функция – это действительная функция φ, значение которой зависит только от расстояния от точки фиксированного центра. Более точно, пусть

$$\varphi: [0, \infty) \to \mathbb{R} \quad (4)$$

Тогда радиально-базисная функция $\Phi_i : \mathbb{R}^d \to \mathbb{R}$ определяется так

$$\Phi_i(x) = \varphi(r_i) = \varphi(\|x - x_i\|) \quad (5)$$

где $\{x_i\}_{i=1}^N \subset \mathbb{R}^d$ это множество различных N точек-центров, и норма $\|\cdot\|$ – произвольная норма из $\mathbb{R}^d$. На практике, как и в нашей статье, целесообразно использовать Евклидову норму. Само название «радиально-базисная функция» основано на следующем ее свойстве. Значение Φ в любой точке некоторой окрестности ее центра константно, т.е.

$$\|x_1\| = \|x_2\| \Rightarrow \Phi(x_1) = \Phi(x_2), \quad x_1, x_2 \in \mathbb{R}^d \tag{6}$$

Таким образом, Φ радиально(сферически) симметрична относительно ее центра.

Приведем примеры классических радиально-базисных функций [19]

$$\varphi(r) = e^{-(\alpha r)^2} \text{ – функция Гаусса,} \tag{7}$$

$$\varphi(r) = \frac{1}{1+(\alpha r)^2} \text{ - обратно квадратичная функция,} \tag{8}$$

$$\varphi(r) = \frac{1}{\sqrt{1+(\alpha r)^2}} \text{ - обратно мультиквадратичная функция,} \tag{9}$$

$$\varphi(r) = \sqrt{1+(\alpha r)^2} \text{ - мультиквадратичная функция,} \tag{10}$$

где α – фиксированный параметр и r – значение «радиуса».

Применим данные функции к построению интерполятора.

Пусть мы имеем неотсортированный набор точек $\{x_i\}_1^N \in \mathbb{R}^1$. Отметим, что этот подход применим и для $\mathbb{R}^N$, что представляет особый интерес для дальнейших исследований[19]. Тогда интерполируемое значение функции f в точке x можно вычислить по формуле

$$f(x) = \sum_{j=1}^N c_j \varphi(r_j) = \sum_{j=1}^N c_j \varphi(\|x - x_j\|), \tag{11}$$

где значение функции $f(x)$ представлено в виде суммы N RBF функций, каждая из которых центрирована в различной точке x_j и помножена на соответствующий ей вес c_j.

Нетрудно заметить, что для решения интерполяционной задачи используется матрица расстояний и разложение по радиальному базису. Это приводит нас к решению системы линейных уравнений для данного набора точек

$$h_i = f(x) = \sum_{j=1}^N c_j \varphi(\|x - x_j\|) = \sum_{j=1}^N c_j \varphi_{i,j}, \qquad i = 1, \dots, N \tag{12}$$

Эта система линейных уравнений представима в виде матричного выражения

$$Ac = h, \tag{13}$$

Где матрица A это симметричная матрица. Выражение можно развернуть следующим образом

$$\begin{pmatrix} \varphi_{1,1} & \cdots & \varphi_{1,i} & \cdots & \varphi_{1,N} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \varphi_{i,1} & \cdots & \varphi_{i,i} & \cdots & \varphi_{i,N} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \varphi_{N,1} & \cdots & \varphi_{N,i} & \cdots & \varphi_{N,N} \end{pmatrix} \begin{pmatrix} c_1 \\ \vdots \\ c_i \\ \vdots \\ c_N \end{pmatrix} = \begin{pmatrix} h_1 \\ \vdots \\ h_i \\ \vdots \\ h_N \end{pmatrix} \tag{14}$$

Данную систему линейных уравнений можно решить методом Гаусса [20], LU-разложения [21] и т.д. Таким образом, задача состоит в решении системы линейных уравнений для нахождения весов c_i для использования их в (12).

Перейдем к реализации RBF-интерполятора на C++. Норма [22] в $\mathbb{R}^1$ вычисляется так

```
double dist(double a, double b) {
    return std::abs(a - b);
}
```

Тогда радиально базисные функции, указанные в (7), (8), (9), (10), можно записать так

```
double gauss(double x, double dx) {
    auto r = dist(x, dx);
    return std::exp(-std::pow(r, 2));
}
double multiquadric(double x, double dx) {
    auto r = dist(x, dx);
    return std::sqrt(1 + r * r);
```

```
}
double inverse_quadric(double x, double dx) {
   auto r = dist(x, dx);
   return 1.0 / (1 + r * r);
}
double inverse_multiquadric(double x, double dx) {
   auto r = dist(x, dx);
   return 1.0 / std::sqrt(1 + r * r);
}
```

В данной реализации α для простоты взят за 1.

Решение системы (14) для относительно большого набора входных точек является нетривиальной задачей, поэтому разумно воспользоваться готовым «решателем» системы линейных уравнений методом LU-декомпозиции из библиотеки Eigen. И по этой причине массивы данных для этой задачи будут представлены в виде массивов из Eigen, а именно Eigen::ArrayX и Eigen::VectorX. Для представления типа матрицы также воспользуемся встроенным типом Eigen::MatrixX. Код программы не трудно переделать так, чтобы входные данные представлялись в виде std::vector, выполнив преобразования к массивам из Eigen. Реализация интерполятора выглядит следующим образом

```
template<class T, class Value = T::value_type>
class rbf_interpolator {
public:
   rbf_interpolator(T &xp, T &yp, std::function<double(double, double)> rf) : xp_(xp), rf_(rf) {
      auto n = xp_.size();
      Eigen::MatrixX<Value> phi_m(n, n);
#pragma omp parallel for simd collapse(2)
      for (size_t i = 0; i < n; ++i) {
         for (size_t j = i; j < n; ++j) {
            auto val = rf_(xp_[i], xp_[j]);
            phi_m(i, j) = val;
            phi_m(j, i) = val;
         }
      }
            // no copy
      Eigen::Map<Eigen::MatrixX<Value>> yp_m(yp.data(), yp.size(), 1);
      weights = phi_m.partialPivLu().solve(yp_m);
   }
   auto operator()(const T &x) const {
      T y(x.size());
#pragma omp parallel for
      for (size_t i = 0; i < x.size(); ++i) {
         Value val = 0.0;
         for (size_t j = 0; j < weights.size(); ++j) {
            val += weights[j] * rf_(x[i], xp_[j]);
         }
         y[i] = val;
      }
      return y;
   }
private:
   T xp_;
   std::function<double(double, double)> rf_;
   Eigen::VectorX<Value> weights;
};
```

Конструктор данного класса принимает на вход наборы входных данных xp, yp и функтор rf, который является RBF-функцией. Далее в конструкторе происходит заполнение матрицы phi_m значений RBF-функции в входных точках. Тут применены две оптимизации: «#pragma omp parallel for simd collapse (2)» это специальная директива препроцессора

OpenMP, распараллеливающая вложенные циклы глубины 2 и учтена симметричность матрицы, поэтому внутренний цикл стартует сразу с «i» и заполняются симметричные позиции в матрице. Далее без копии создается объект матрицы yp_m из входного массива yp, для использования в решении СЛАУ. Метод solve() решает данную СЛАУ методом LU-декомпозиции, что является наиболее высоконагруженной частью алгоритма. Выбор такого способа неслучаен, т.к. только он имеет внутреннее распараллеливание среди прочих «решателей» из Eigen, что является очень существенным фактором, сказывающимся на времени работы программы.

Перегруженный оператор «()» принимает массив точек x, в которых нужно выполнить интерполяцию. Тут происходит дословное воспроизведение формулы [NUMBER] с применение распараллеливания только внешнего цикла, во избежание гонки данных. Использование распараллеливания вложенного цикла с использованием атомарных переменных или критических секций даст лишь падение производительности. На RBF интерполятор накладываются следующие ограничения. Наборы входных данных `xp` и `yp` должны быть одного размера. Требования упорядоченности наборов нет. Данный интерполятор сильно уступает в скорости приведенным ранее интерполяторам, но имеет преимущество в способности обрабатывать нерегулярные наборы входных данных, что особенно важно и полезно в двумерном или трехмерном случаях.

Обсуждение результатов. Проведем замеры производительности реализованных интерполяторов, сравним их с аналогами из SciPy и построим графики получившихся функций. Компиляция и запуск программ производилась в системе Ubuntu 23.04, с версией ядра 6.2, процессор Intel Core i5-12600K, компилятор GCC 13.1. Стандарт языка – C++-20. При компиляции указан флаг `-fopenmp` для включения OpenMP под GCC и флаг –lgsl для подключения статической библиотеки GSL. Желательно добавить флаг `-march=native` для максимальных оптимизаций под конкретную архитектуру процессора.

Тестовой функцией для проверки производительности являлась функция $f(x) = xsin(x)$ на отрезке $[-100, 100]$ с шагом 0.02. Итого, N = 10000, где N – количество точек. С указанным шагом заполнялись значения исходных массивов `xp` и `yp`. Далее в массив x размера M = 20000 записывались случайные числа, созданные с помощью генератора случайных чисел Мерсенна [23] из данного отрезка.

По полученному массиву x производилась интерполяция. Массивы и случайные числа в тестах на Python строились с помощью библиотеки NumPy [24]. Получены следующие результаты (табл.1). Как видно из табл.1, наилучший результат достигается при включенных оптимизациях OpenMP и SIMD.

Таблица 1. Сравнение времени выполнения интерполяторов
Table 1. Comparison of execution time of interpolators

	OpenMP + SIMD	OpenMP	Без OpenMP и SIMD	SciPy, interp1d
Linear, N = 10000, M = 20000	0.000545884 sec	0.000579729 sec	0.00142267 sec	0.002537 sec
Quadratic, N = 10000, M = 20000	-	-	0.00179693 sec	0.035637 sec
Cubic, N = 10000, M = 20000	0.000851297 sec	0.000861991 sec	0.00197976 sec	0.037861 sec
RBF, N = 10000, M = 20000	4.24728 sec	11.2738 sec	37.0253 sec	-

Эти результаты превосходят данные, полученные из SciPy в **4.6**, **19** и **44** раза в случаях линейной, квадратической и кубической интерполяциях соответственно. В случае RBF интерполяции виден существенный прирост скорости при использовании OpenMP и SIMD – в **8.8** раз. Тем не менее, этот тип интерполяции остается довольно медленным по сравнению с тремя первыми. В SciPy RBF интерполяции в одномерном случае нет.

Для наглядности приведем графики (рис.1-5) полученных интерполированных значений в сравнении с оригинальным графиком $f(x) = xsin(x)$, но уже на отрезке $[-10, 10]$. Построения производились в той же системе с помощью утилиты gnuplot

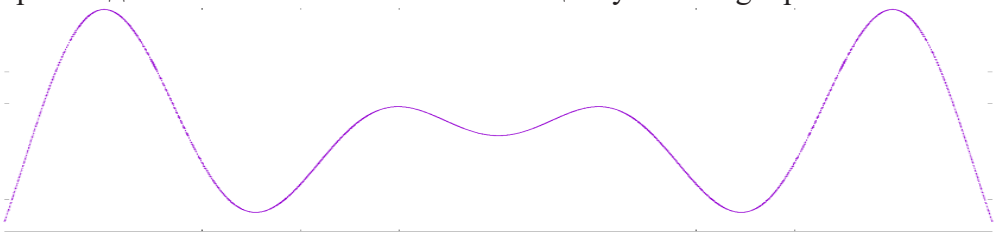

Рис. 1. Исходные точки
Fig. 1. Original points

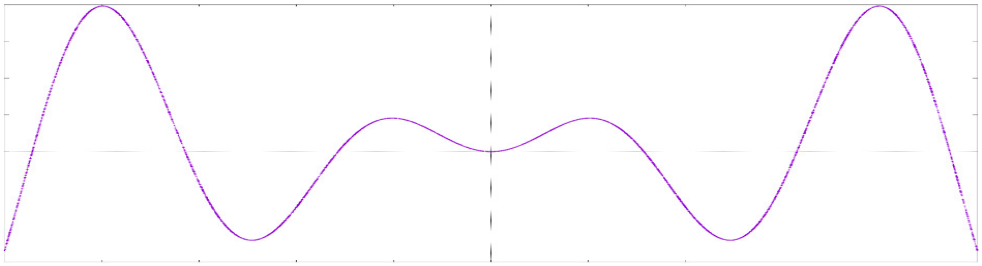

Рис. 2. Результат линейной интерполяции
Fig. 2. Result of linear interpolation

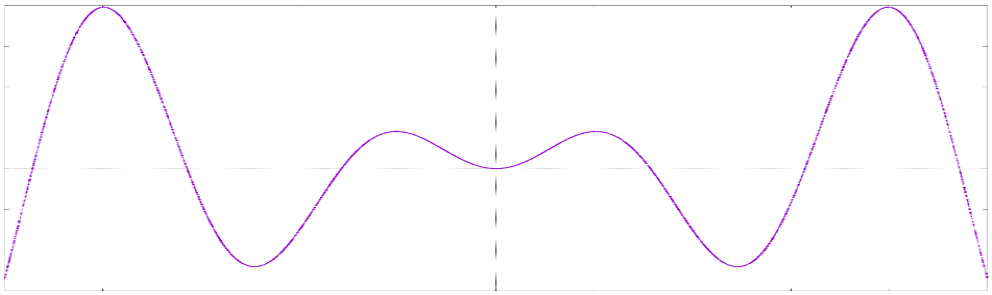

Рис 3. Результат квадратической интерполяции
Fig. 3. Result of quadratic interpolation

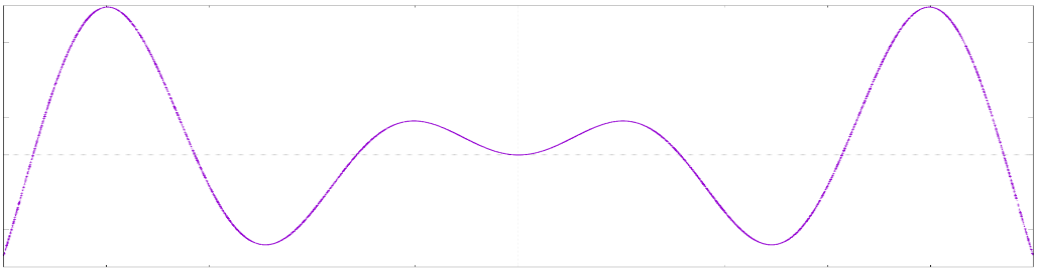

Рис. 4. Результат кубической интерполяции
Fig. 4. Result of cubic interpolation

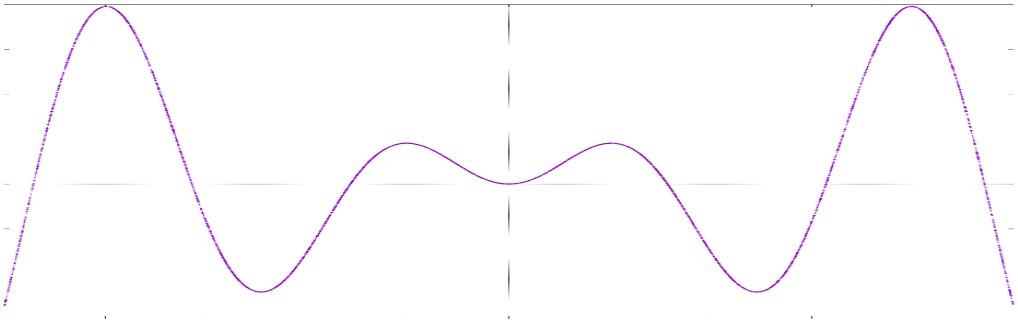

Рис. 5. Результат RBF-интерполяции
Fig. 5. Result of RBF-interpolation

Вывод. В статье мы рассмотрели различные виды одномерных интерполяторов, такие как линейные, квадратические, кубические, а также радиально-базисные функции(RBF), построение интерполяторов с их помощью, их реализацию на языке программирования C++ актуального стандарта C++-20 с использованием библиотек GSL и Eigen, а также применили ряд дополнительных оптимизаций с использованием OpenMP и SIMD. В результате, исследование приводит нас к следующим ключевым выводам:

1. Интерполяторы на C++ с использованием GSL и Eigen оказались более эффективными и быстрыми по сравнению с аналогичными готовыми интерполяторами из SciPy. В наших тестах мы использовали относительно скромные размеры входных данных. Если, к примеру, увеличить N до 500000 и M до 1000000, то время работы нашего кубического интерполятора будет равна 0.04741 с., в то время как время работы кубического интерполятора из SciPy будет равно 52.8163 с., что в **1114** медленнее. Это колоссальная разница.
2. Реализация интерполяторов, использующих разные методы интерполяции, позволяет выбирать наилучший метод в зависимости от конкретной задачи.
3. Применение технологии OpenMP и SIMD позволяет эффективно распараллеливать интерполяцию данных, ускоряя ее выполнение.

Библиографический список:

1. Stroustrup B. The C++ Programming Language, 4th Edition. Addison-Wesley. - 2013 – 1361 с.
2. Половко А.М., Бутусов П.Н. Интерполяция. Методы и компьютерные технологии их реализации. - СПб.: БХВ-Петербург, 2004 – 314 с.
3. Hardy R.L. Multiquadric equations of topography and other irregular surfaces. Journal of Geophysical Research. 76 (8): pp 1905–1915.
4. Galassi. M. GNU Scientific Library: Reference Manual, 3rd Edition. - Network Theory, 2009 – 573 с.
5. Антонов А.С. Параллельное программирование с использованием технологии OpenMP .— Москва: МГУ, 2009 .— 728 с.
6. Нуньес-Иглесиас Х., Уолт Ш., Дэншоу Х. Элегантный SciPy. - М.: ДМК Пресс, 2018. - 266с.
7. Документация библиотеки Eigen [дата обращения 29.08.2023]. Доступ по ссылке: https://eigen.tuxfamily.org
8. Paul C., Kenneth R. SIMD Programming Manual for Linux and Windows. - Springer, 2004 - 351с.
9. Лутц М. Изучаем Python, - Вильямс, 2019 - 832с.
10. Соснин В., Балакшин П., Шилко Д., Пушкарев, Д., Мишенёв А., Кустарев П., Тропченко А. Введение в параллельные вычисления. - Санкт-Петербург, ИТМО, 2023 - 130с.
11. Ермолицкий А.В., Нейман-заде М.И. Методы повышения эффективности автоматической векторизации вычислений. Доклады пятой международной конференции «Параллельные вычисления и задачи управления». - Москва. 2010 - с. 957-972.
12. Калачев А. Многоядерные процессоры. Учебное пособие. - Просвещение, 2014 - 247с.
13. Джосаттис Н. Стандартная библиотека C++, 2-е изд. : Пер с англ. – М. :ООО “И.Д. Вильямс”, 2014. – 1136 с.
14. Мейерс С. Эффективный и современный C++. Пер. С англ. М. : ООО “И.Д. Вильямс”, 2016. – 304 с.
15. Vandevoorde D., Josuttis N, Gregor D. C++ Templates. The Complete Guide, 2nd Edition. Addison-Wesley, 2018. – 849 с.
16. Josuttis N. C++20 – The Complete Guide. The LeanPub, 2022 – 762 с.
17. Уильямс Э. C++. Практика многопоточного программирования. – СПб.: Питер, 2020. – 640 с.
18. Пышкин Е.В. Структуры данных и алгоритмы: реализация на C++. СПб.:ФТК СПБГПУ, 2009. 200 с.
19. Majdisova Z. Interpolation and Approximation Methods for Large Geometric Datasets. - University of West Bohemia, 2016 – pp 18 – 24.
20. Авхадиев Ф.Г. Численные методы алгебры и анализа. Учебное пособие. Казань: Издательство Казанского университета, 2019. – 200 с.
21. Банников А.С., Ким И.Г., Латыпова Н.В. Численные методы. Учебное пособие. Ч 1. – 2-е изд., испр. и доп. – Ижевск: Издательский центр «Удмуртский университет», 2018. – 80 с.
22. Колмогоров А.Н., Фомин С.В. Элементы теории функций и функционального анализа. Главная редакция физико-математической литературы изд-ва «Наука», М., 1975. – 575 с.
23. Gentle J.E. «Random Number Generation and Monte Carlo Methods», 2nd Edition. Springer, 2003. 399 p.
24. Документация библиотеки NumPy [дата обращения 1.09.2023]. Доступ по ссылке: https://numpy.org/doc/stable/
25. Zhang N., Canini K., Silva S., Gupta M. Fast Linear Interpolation. ACM Journal on Emerging Technologies in Computer Systems, Volume 17, Article No.: 20, 2021, pp 1-15 .
26. Li C., Zhao R. Implementation of RBF Mesh Deformation with Topology Refinement in OpenFOAM. Proceedings of the 2020 4th International Conference on High Performance Compilation, Computing and Communications. 2020, pp 79-93.

References

1. Stroustrup B. The C++ Programming Language, 4th Edition. Addison-Wesley. 2013; 1361.
2. Polovko A.M., Butusov P.N. Interpolation. Methods and computer technologies for their implementation. - St. Petersburg: BHV-Petersburg, 2004;314. (In Russ)
3. Hardy R.L. Multiquadric equations of topography and other irregular surfaces. Journal of Geophysical Research. 76 (8): 1905–1915.
4. Galassi M. GNU Scientific Library: Reference Manual, 3rd Edition. Network Theory, 2009; 573.
5. Antonov A.S. Parallel programming using OpenMP technology . Moscow: MSU, 2009; 728. (In Russ)
6. Nunez-Iglesias J., Walt C., Danshaw H. Elegant SciPy. - M.: DMK Press, 2018; 266. (In Russ)
7. Eigen C++ Library Documentation [accessed 29.08.2023]. URL: https://eigen.tuxfamily.org
8. Paul C., Kenneth R. SIMD Programming Manual for Linux and Windows. Springer, 2004; 351.
9. Lutz M. Learning Python - Williams, 2019; 832. (In Russ)
10. Sosnin V., Balakshin P., Shilko D., Pushkarev D., Mishenyov A., Kustarev P., Tropchenko A. Introduction to parallel computing. St. Petersburg, ITMO, 2023; 130. (In Russ)
11. Ermolitsky A.V., Neumann-zade M.I. Methods for increasing the efficiency of automatic vectorization of calculations. Reports of the fifth international conference "Parallel Computing and Control Problems". - Moscow. 2010; 957-972. (In Russ)
12. Kalachev A. Multi-core processors. Tutorial. - Enlightenment, 2014;247. (In Russ)
13. Josattis N. C++ Standard Library, 2nd Ed. M. :OOO “I.D. Williams”, 2014; 1136. (In Russ)
14. Meyers S. Efficient and modern C++ – M. : OOO I.D. Williams”, 2016; 304. (In Russ)
15. Vandevoorde D., Josuttis N, Gregor D. C++ Templates. The Complete Guide, 2nd Edition. Addison-Wesley, 2018; 849.
16. Josuttis N. C++20 – The Complete Guide. The LeanPub, 2022; 762.
17. Williams E. C++. The practice of multithreaded programming. St. Petersburg: Peter, 2020; 640. (In Russ)
18. Pyshkin E.V. Data Structures and Algorithms: Implementation in C++. St. Petersburg: FTK SPbGPU, 2009; 200. (In Russ)
19. Majdisova Z. Interpolation and Approximation Methods for Large Geometric Datasets. University of West Bohemia, 2016;18 – 24.
20. Avkhadiev F.G. Numerical methods of algebra and analysis. Tutorial. Kazan: Kazan University Press, 2019; 200. (In Rus)
21. Bannikov A.S., Kim I.G., Latypova N.V. Numerical methods. Tutorial. Ch 1. - 2nd ed., Rev. and additional - Izhevsk: Publishing Center "Udmurt University", 2018; 80. (In Russ)
22. Kolmogorov A.N., Fomin S.V. Elements of the theory of functions and functional analysis. The main edition of the physical and mathematical literature of the publishing house "Nauka", M., 1975; 575. (In Russ)
23. Gentle J.E. «Random Number Generation and Monte Carlo Methods», 2nd Edition. Springer, 2003; 399.
24. NumPy Documentation [accessed 1.09.2023]. URL: https://numpy.org/doc/stable/
25. Zhang N., Canini K., Silva S., Gupta M. Fast Linear Interpolation. ACM Journal on Emerging Technologies in Computer Systems, Volume 17, Article No.: 20, 2021;1-15 .
26. Li C., Zhao R. Implementation of RBF Mesh Deformation with Topology Refinement in OpenFOAM. Proceedings of the 2020 4th International Conference on High Performance Compilation, Computing and Communications. 2020; 79-93.

Сведения об авторах:

Абилова Фарида Владимировна, кандидат физико-математических наук, доцент, заведующая кафедрой высшей математики; kvm@dstu.ru

Абилов Марат Владимирович, кандидат физико-математических наук, старший преподаватель кафедры высшей математики; abilov.marat70@mail.ru

Селимханов Эмирхан Валерьевич, ведущий специалист отдела разработки геологических проектов Управления развития информационных технологий; postpost42@yandex.ru

Information about authors:

Farida V. Abilova, Cand. Sci. (Physics and Mathematics), Assoc. Prof., Department of Higher Mathematics; kvm@dstu.ru

Marat V. Abilov, Cand. Sci. (Physics and Mathematics), Senior Lecturer, Department of Higher Mathematics; abilov.marat70@mail.ru

Emirkhan V. Selimkhanov, Leading Specialist of the Geological Projects Development Department of the Information Technology Development Department; postpost42@yandex.ru

Конфликт интересов/Conflict of interest.

Авторы заявляют об отсутствии конфликта интересов/The authors declare no conflict of interest.

Поступила в редакцию/Received 26.09.2023.

Одобрена после рецензирования/ Reviced 12.10.2023.

Принята в печать/Accepted for publication 12.10.2023.