Fast Summation of Multiquadric Kernels

Jerry Lingjie Mei

December 13, 2017

1 Background

The *radial basis functions* are a set of real-valued functions whose value only depends on the variable's distance from some other point (i.e. $\phi(\mathbf{x}) = \phi(\|(\mathbf{x} - \mathbf{c}\|))$. Here are some of the most commonly-used types of radial basis (let r denote $\|\mathbf{x} - \mathbf{c}\|$ for simplicity):

$$\begin{array}{cc} \phi(r) = e^{-(\varepsilon r)^2} & \phi(r) = \sqrt{1 + (\varepsilon r)^2} \\ \phi(r) = \dfrac{1}{1 + (\varepsilon r)^2} & \phi(r) = \dfrac{1}{\sqrt{1 + (\varepsilon r)^2}} \\ \phi(r) = r^k & \phi(r) = r^k \ln r \end{array}$$

The radial basis functions are widely used in mathematical physics. The multiquadric function $\phi(r) = \sqrt{1 + (\varepsilon r)^2}$, for example, is used to represent multidimensional high energy physics data[1]. In this paper, we will discuss and implement a fast numerical method to evaluate the sum of multiquadric function in $\mathbb{R}$:

$$s(x) = \sum_{j=1}^{N} w_j \phi(x - x_j)$$

where $\phi(x) = \sqrt{x^2 + h^2}$ (a small variation from the original representation of the multiquadric function). Here w_j denotes the weight of the jth charge, and x_j denotes the place of the jth charge. In particular, we set $x_j = jh$ and $h = 1/N$ so that all weights would be equidistributed on the interval $[0, 1]$.

2 Brute Force Algorithm

The simplest way to evaluate the function s is to evaluate it at all possible $x \in \mathbb{R}$. We would calculate each multiquadric terms, and then compute its weighted sum and we will get $s(x)$.

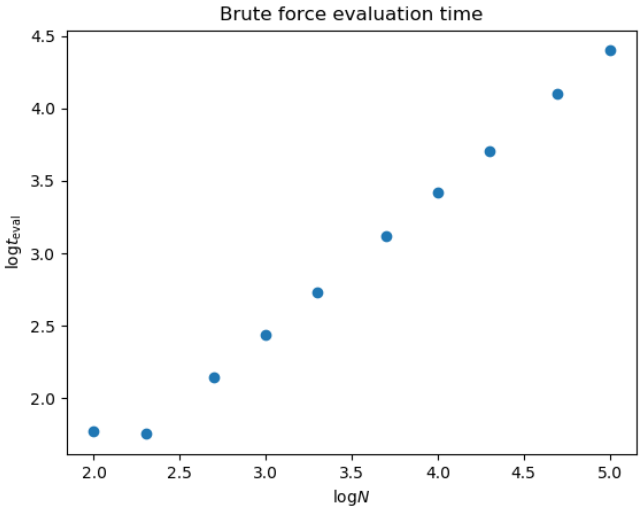
2.1 Implementation

The brute force algorithm is implemented by `BruteForce` class, which considers all charges places at `[begin, end)`, with weights in `[weightBegin, weightEnd)` respectively. The parameter in the multiquadric function would be `h`.

To compute $s(x)$ at an evaluation point x, we implement the method `BruteForce::evaluate`, which computes the sum of multiquadric functions at `x`.

2.2 Time Complexity

The build time complexity of the brute force algorithm is $O(1)$, as no actual calculations are implemented in the building process. The evaluation time complexity of brute force algorithm is $O(N)$, where N is the number of charges. Each evaluation needs to compute all charges' influence on the point x, which makes an iteration of length N.



We run the brute force algorithm in our test code, and record the time it takes to build and evaluate, as shown in the table A in the appendix. The linear regression test on $\log N$ vs. $\log t_{\text{eval}}$ gives a slope of 0.929 and p-value of 1.02×10^{-9}, which proves our estimation of time complexity.

As we can see, since the building time complexity is much smaller than the evaluate time complexity, the brute force method is better used when the total number of queries is small. If there is going to be as much as M queries, then the total time complexity would be $O(MN)$, which would be large when M is as large as N. Therefore an algorithm with smaller evaluate time complexity is needed.

3 FMM method

The `FMM` method is an acronym for Fast Multiple Method, or a comprehensible solution to the summation of radial basis function. The first idea behind the FMM method is that we should not evaluate the sum of multiquadric function precisely, but with an acceptable error less than ε. This trade-off allows a faster computation scheme.

The essence is that if the charge is distant from the evaluation point x, then the multiquadric function can be approximated by a Laurent series at infinity:

$$\begin{aligned}\sqrt{(x-t)^2+h^2} &= |x-c|\sqrt{1+\frac{h^2}{(x-t)^2}} \\ &= |x-t|\sum_{i=0}^{\infty}\binom{1/2}{i}\frac{h^{2i}}{(x-t)^{2i}} \\ &= \operatorname{sgn}(x-t)\left(x-t+\sum_{i=1}^{\infty}\binom{1/2}{i}\frac{h^{2i}x^{1-2i}}{(1-tx^{-1})^{2i-1}}\right) \\ &= \operatorname{sgn}(x-t)\left(x-t+\sum_{i=1}^{\infty}\binom{1/2}{i}\sum_{j=0}^{\infty}\binom{2i-2+j}{2i-2}h^{2i}t^j x^{-2i+1-j}\right)\end{aligned} \tag{1}$$

with $|x|$ sufficiently large. To be exact, we use the truncation of the Laurent series to approximate the multiquadric function:

$$\left|\sqrt{(x-t)^2+h^2}-\sum_{i=0}^{p+1}q_i x^{-i+1}\right| \le 2(|t|+h)\left(\frac{\sqrt{t^2+h^2}}{|x|}\right)^{p+1}\frac{1}{1-\frac{\sqrt{t^2+h^2}}{|x|}}$$

given $|x|>\sqrt{t^2+h^2}$. This implies that if the evaluation point is farther away from the source, the approximation by Laurent series converges to the multiquadric function as $p\to\infty$.

Now let us consider a cluster of charges that are placed in the interval $I=[x_m-d,x_m+d]$. We also make an translation if the Laurent series so that it terms can be denoted by $(x-m)^{-1}$ rather than x^{-1}. Then the approximation above looks like this:

$$\sqrt{(x-t)^2+h^2}=\sqrt{((x-m)-(t-m))^2+h^2},$$

$$\begin{aligned}&\left|\sqrt{(x-t)^2+h^2}-\sum_{i=0}^{p+1}q_i(x-m)^{-i+1}\right| \\ &\le 2(|t-m|+h)\left(\frac{\sqrt{(t-m)^2+h^2}}{|x-m|}\right)^{p+1}\frac{1}{1-\frac{\sqrt{(t-m)^2+h^2}}{|x-m|}}\end{aligned}$$

If we assume the evaluation point x is placed outside $[m-3d,m+3d]$, we say that x is *well-separated* from I. Then we may deduce an upper bound for the approximation:

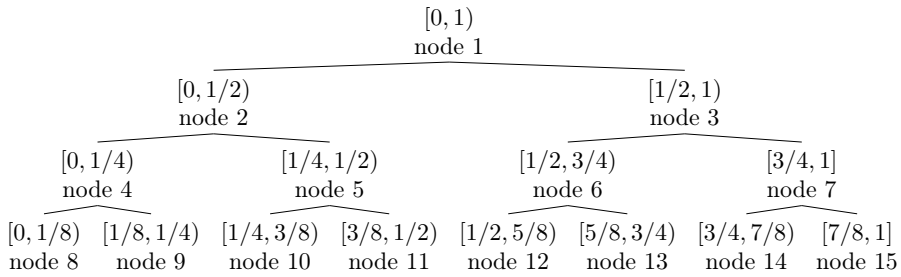
$$\left|\frac{\sqrt{(t-m)^2+h^2}}{|x-m|}\right| \le \frac{\sqrt{d^2+h^2}}{3d} \le \frac{\sqrt{2}}{3}$$

$$\left|s_I(x)-\sum_{i=0}^{p+1}q_{I,i}(x-m)^{-i+1}\right| \le 8dW_I(\frac{\sqrt{2}}{3})^{p+1} \tag{2}$$

where W_I denotes the sum of the absolute value of the weight of charges in I, s_I denotes the influence of charges in I on the evaluation point x. Here we assume that h is smaller than the minimum radius of I.

If the evaluation point x is placed inside I, then we cannot use the approximation formula. But as d is so small, the charges that I may contain is rather limited and the brute force formula will do the work.

Moreover, we may observe if two adjacent intervals are both well-separated from the evaluation point x, there is a great chance that the union of two intervals is still well-separated from x. Based on this intuition, we may build a binary segment tree of M layers on the entire interval of all possible charges (here we assume it is as simple as $[0, 1]$).



Segment tree when $M = 3$

We may choose $M \approx \log_2 n_{\text{charges}}$ such that every bottom node may contain roughly $O(1)$ charges with high probability. With the segment tree being built, we can store the coefficients of Laurent series (whose terms are expressed in $(x - m)^{-1}$, where m is the midpoint of the interval) in each node, which is updated from the bottom to above.

The major benefit of building the segment tree is that evaluation comes faster. If we naively use all bottom nodes, the evaluation is still $O(N)$ in time complexity. But if we use the highest possible set of nodes that still partition $[0, 1]$, the time complexity will be reduced to $O(\log N)$. We only need to start from the top node $[0, 1]$, and bisect the nodes that are not well-separated from the evaluation point x. For the purpose of illustration, let $x = 0.3$, then the evaluation process would be like

$$\begin{aligned} s_1(x) &= s_2(x) + s_3(x) \\ &= s_4(x) + s_5(x) + s_6(x) + s_7(x) \\ &= s_8(x) + s_9^*(x) + s_{10}^*(x) + s_{11}^*(x) + s_{12}(x) + s_{13}(x) + s_{15}(x) \end{aligned}$$

The terms with $*$ will be evaluated by brute force, while the other terms will be evaluated by using the coefficients of the Laurent series. In practice, we will rather use recursion to evaluate the sum.

The aggregated error of evaluation at point x is bounded by

$$\sum_I 8dW_I(\frac{\sqrt{2}}{3})^{p+1} \leq 8W_{\text{all}}(\frac{\sqrt{2}}{3})^{p+1}$$

To keep the error below ε, we should choose $p \geq \log_{3/\sqrt{2}} 8W_{\text{all}}/\varepsilon - 1$.

Algorithm 1: FFM method: Build

1 Let $M \sim \log_2 N$, $p \geq \log_{3/\sqrt{2}} 8W_{\text{all}}/\varepsilon - 1$
2 Build segment tree of $[0, 1]$ of $M + 1$ layers.
3 For *node* from 1 to $2^{M+1} - 1$,
(I) If *node* has no children, compute its coefficients by the formula (1).
(II) If *node* has (two) children, compute its coefficients by summing the coefficients of its two children, but using the distance from the midpoint of interval.

Algorithm 2: FFM method: Evaluate at a node

1 If I well-separated from x
Return the result using current nodes' coefficients.
2 Else if I has no children node
Return the result by brute force method.
3 Else
Call the evaluate method on its two children children, and sum the two evaluation results up.

To summarize the algorithm, we give a step-by-step procedure to FMM algorithm:

To evaluate the entire $[0, 1]$'s influence on the evaluation point x, just evaluate it at the top node (node 1).

3.1 Implementation

The FMM method is implemented by the `FFM1d` class, which considers all charges places at `[begin, end)`, with weights in `[weightBegin, weightEnd)` respectively. The parameter in the multiquadric function is set to be `h`. The maximal accepted error is `eps`. The class also has an attribute `nodes` that contains all nodes in the segment tree.

To compute $s(x)$ at an evaluation point x, we implement the method of `FFM1d::evaluate`, which computes the sum of multiquadric functions at `x`.

Closely related to the `FMM1d` class is the class of its nodes, `FFM1dNode`. Each node represents charges places at `[begin, end)` (or in`[lower,upper)`), with weights of `[weightBegin, weightEnd)` respectively, and the parameter in the multiquadric function being `h`. The coefficients of this node is stored in `coeff` with length `p+2`. It may also has a left child `lChild`, and a right Child `rChild`.

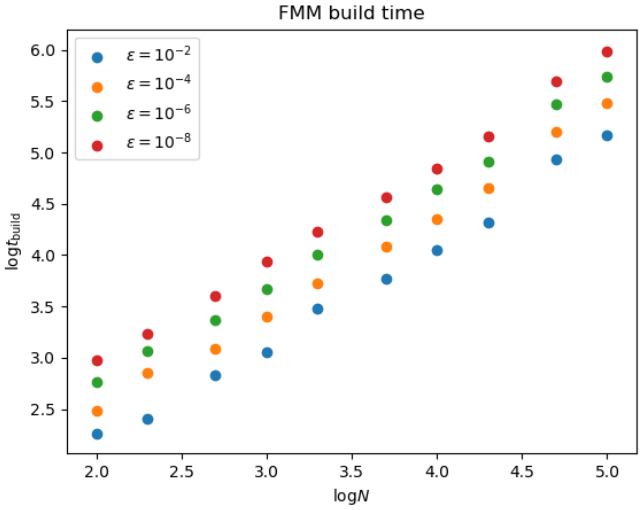
The `FMM1d` has method `FMM1d::buildCoeff` to calculate all coefficients at this node. It also has method `FMM1d::evaluate` to compute the influence of charges in the interval on the evaluation position `x`.

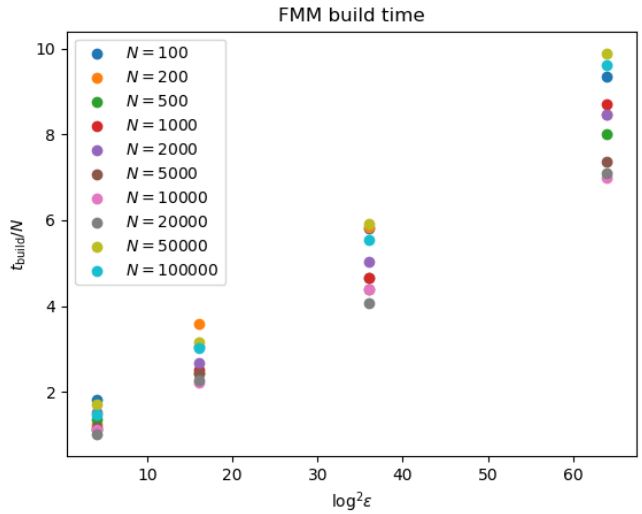
3.2 Time complexity

The building process has time complexity $O(Np^2)$, or $O(N \log_2(W_{all}/\varepsilon))$. Every `FMM1d` has $O(N)$ nodes, as the number of layers $M + 1 \approx \log_2 N$. The coefficients of bottom node use the formula (1), with two loops of length

$O(p)$, and the loops run at $O(1)$ times for all charges in the interval, marking a $O(p^2)$ complexity. The coefficients of non-bottom nodes are a combination of the coefficient of its two children, which also has time complexity $O(p^2)$. Therefore the building process requires a $O(Np^2) = O(N \log^2(W_{all}/\varepsilon)$ time complexity.

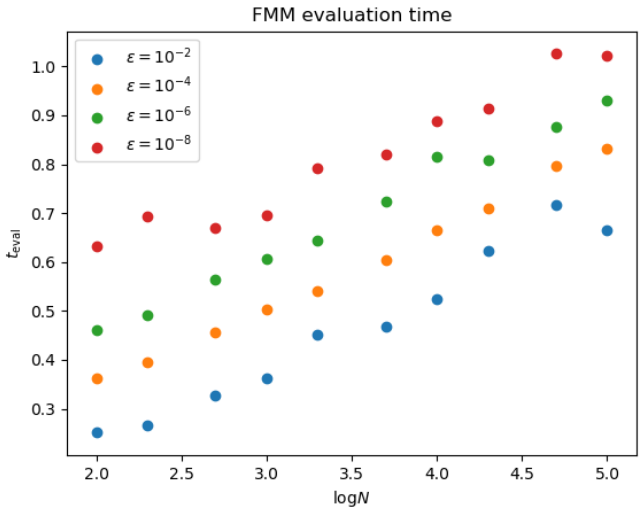
The evaluation process has time complexity $O(Mp) = O(N \log_2(W_{all}/\varepsilon) \log_N$. The evaluation of $s_I(x)$ at each node requires $O(p)$ time if it is evaluated using the Laurent series coefficients, or $O(1)$ if it is evaluated using brute force. The total number of nodes calling its `evaluate` method is $O(M)$, as there cannot be more than three same-layer nodes that are evaluated and reside on the same side of x. The total time complexity is $O(Mp) = O(\log_2(W_{all}/\varepsilon) \log_N)$.

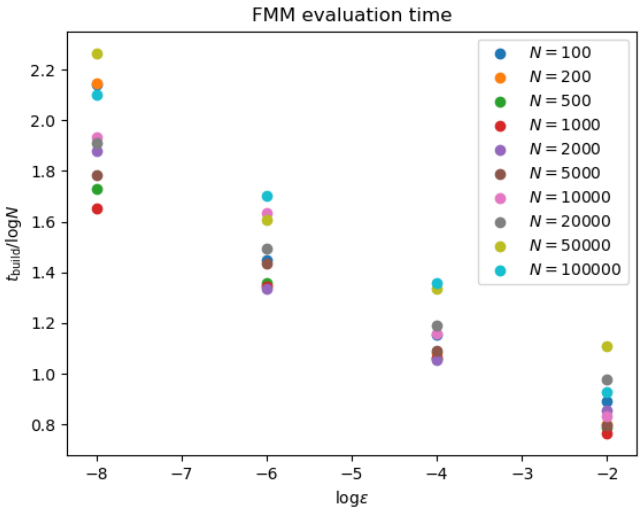




We run the FMM with our test code and record the time it takes for the building and evaluation process, as shown in table B in the appendix.

The linear regression test on $\log N$ vs. t_{build} gives a p-value of 1.08×10^{-10}; the linear regression test on $\log^2 \varepsilon$ vs. t_{build}/N gives a p-value of 4.50×10^{-4}. The two linear regression indicates that $t_{\text{build}} \propto N \log^2(1/\varepsilon)$, which proves our estimation of time complexity[1].





The linear regression test on $\log N$ vs. $\log t_{\text{eval}}$ gives a slope of 0.986 and p-value of 1.00×10^{-10}; the linear regression test on $\log \varepsilon$ vs. $t_{\text{eval}}/\log N$ gives a p-value of 3.9×10^{-3}. The two linear regression indicates that $t_{\text{eval}} \propto \log N \log(1/\varepsilon)$, which proves our previous estimation of time complexity.

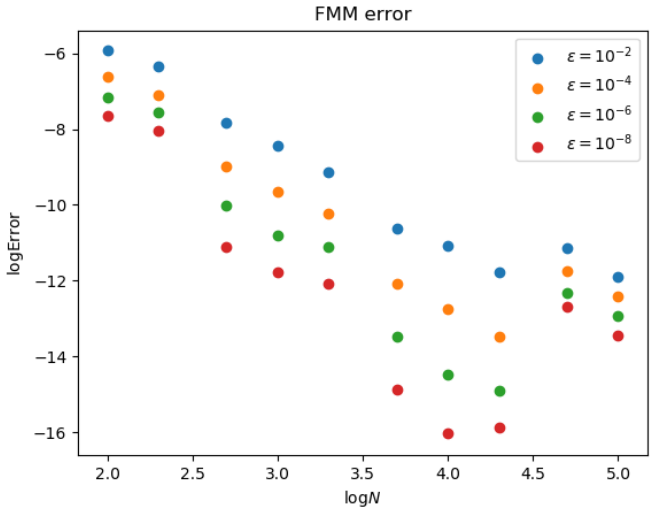
The time complexity of the FMM method is considerably shorter and will be suitable for large numbers of evaluations.

[1]In the test code, we assume that $W_{all} = O(1)$ so only ε matters with the time complexity.

4 Comparison between Brute Force and FMM

4.1 Error

As stated before, the FMM method is based on approximations of error level ε. Here we need to test if our `FMM1d:evaluate` really gives the $s(x)$ at the evaluation point x with error less than ε. We record the $\mathcal{L}^\infty$ error of all FMM methods with various N and ε, as shown in Table C in the appendix.



The plot shows that the $\mathcal{L}^\infty$ error is far less than the ε that we require. The actual $\mathcal{L}^\infty$ error decreases as ε decreases[2].

4.2 Time Efficiency

We will also conclude the situations whether the brute force algorithm or the FMM method should be used to achieve the highest time-efficiency. As shown in table D in the appendix, the typical threshold for switching between brute force algorithm and FMM method is less than 50 for all of our test cases, or $O(Nlog_2^2(W_{all}/\varepsilon))$[3] in a broader sense. If the number of evaluations is greater than this threshold, the FMM method is preferred for less evaluation time; otherwise, the brute force method is preferred for less build time. In real practice, the FMM method is more useful as evaluations are typically required on a large scale, or at least over 100 times.

[2] As N grows too large, our current implementation is affected by double arithmetic

[3] The constant of which is around 1.5

N	ε=10^{-2}	ε=10^{-4}	ε=10^{-6}	ε=10^{-8}
1×10^2	3	5	10	17
2×10^2	4	13	21	32
5×10^2	4	8	17	29
1×10^3	4	9	17	32
2×10^3	5	10	18	31
5×10^3	4	9	16	27
1×10^4	4	8	16	26
2×10^4	4	8	16	27
5×10^4	6	12	23	38
1×10^5	5	12	22	38

Number of evaluations with same time taken in both methods

5 Conclusion

In this paper, we discussed two ways to compute the summation of multiquadric functions. The first way is through brute force, which produces the accurate result within the time complexity of $O(N)$. The other method is through FMM, which is based on Laurent series approximation of well-separated intervals and the construction of segment tree. The FMM method produces the summation result with error less than ε, takes $O(N \log_2^2(W_{all}/\varepsilon))$ time to build a problem and $O(\log_2 N \log_2(W_{all}/\varepsilon))$ to evaluate at a point. We implemented codes for both methods and used them to prove the proposed time complexity and error bound. We also give a strategy to decide which solution to choose so that total time cost is minimized, depending on the desired error bound. We are looking forward to generalizing this method to other 1-dimensional radial basis functions, as well as other kernels in the second or higher dimension.

Appendices

A Running Time: Brute Force vs. FMM

N	BruteForte build (μs)	BruteForte evaluate (μs)	FMM, $\varepsilon = 10^{-4}$ build (μs)	FMM, $\varepsilon = 10^{-4}$ evaluate (μs)
1×10^2	<1	58.5	305	2.30
2×10^2	<1	56.5	717	2.49
5×10^2	<1	139.1	1.21×10^3	2.85
1×10^3	<1	282.5	2.50×10^3	3.18
2×10^3	<1	536.5	5.37×10^4	3.47
5×10^3	<1	1.346×10^3	1.22×10^4	4.03
1×10^4	<1	2.582×10^3	4.55×10^4	5.12
2×10^4	<1	5.102×10^3	2.23×10^5	4.62
5×10^4	<1	1.267×10^4	1.57×10^6	6.27
1×10^5	<1	2.503×10^4	3.02×10^6	6.78

B Running Time of FMM

N	$\varepsilon = 10^{-2}$ build (μs)	$\varepsilon = 10^{-2}$ eval (μs)	$\varepsilon = 10^{-4}$ build (μs)	$\varepsilon = 10^{-4}$ eval (μs)	$\varepsilon = 10^{-6}$ build (μs)	$\varepsilon = 10^{-6}$ eval (μs)	$\varepsilon = 10^{-8}$ build (μs)	$\varepsilon = 10^{-8}$ eval (μs)
1×10^2	183	1.78	305	2.30	582	2.89	935	4.28
2×10^2	254	1.84	717	2.49	1.16×10^3	3.10	1.69×10^3	4.94
5×10^2	680	2.12	1.21	2.85	2.33×10^3	3.66	3.99×10^3	4.66
1×10^3	1.13×10^3	2.29	2.50×10^3	3.18	4.65×10^3	4.03	8.71×10^3	4.96
2×10^3	3.04×10^3	2.82	5.37×10^3	3.47	1.00×10^4	4.40	1.69×10^4	6.20
5×10^3	5.90×10^3	2.94	1.22×10^4	4.03	2.19×10^4	5.30	3.68×10^4	6.59
1×10^4	2.06×10^4	4.20	4.55×10^4	5.12	8.15×10^4	6.42	1.42×10^5	8.20
2×10^4	1.12×10^5	3.33	2.23×10^5	4.62	4.38×10^5	6.53	7.00×10^5	7.73
5×10^4	8.58×10^5	5.20	1.57×10^6	6.27	2.95×10^6	7.54	4.93×10^6	10.6
1×10^5	1.47×10^6	4.63	3.02×10^6	6.78	5.55×10^6	8.50	9.62×10^6	10.4

C Error of FMM Algorithms

N	ε=10^{-2}	ε=10^{-4}	ε=10^{-6}	ε=10^{-8}
1×10^2	1.23×10^{-6}	2.49×10^{-7}	6.84×10^{-8}	2.30×10^{-8}
2×10^2	4.45×10^{-7}	8.10×10^{-8}	2.71×10^{-8}	9.15×10^{-9}
5×10^2	1.47×10^{-8}	1.02×10^{-9}	9.92×10^{-11}	7.92×10^{-12}
1×10^3	3.52×10^{-9}	2.27×10^{-10}	1.54×10^{-11}	1.62×10^{-12}
2×10^3	7.32×10^{-10}	5.86×10^{-11}	7.96×10^{-12}	8.31×10^{-13}
5×10^3	2.35×10^{-11}	8.32×10^{-13}	3.36×10^{-14}	1.30×10^{-15}
1×10^4	8.51×10^{-12}	1.77×10^{-13}	3.38×10^{-15}	9.62×10^{-17}
2×10^4	1.71×10^{-12}	3.29×10^{-14}	1.25×10^{-15}	1.31×10^{-16}
5×10^4	7.11×10^{-12}	1.85×10^{-12}	4.78×10^{-13}	1.99×10^{-13}
1×10^5	1.29×10^{-12}	3.94×10^{-13}	1.18×10^{-13}	3.71×10^{-14}

D Number of Evaluations with Same Time Cost: Brute Force vs. FMM

N	ε=10^{-2}	ε=10^{-4}	ε=10^{-6}	ε=10^{-8}
1×10^2	3	5	10	17
2×10^2	4	13	21	32
5×10^2	4	8	17	29
1×10^3	4	9	17	32
2×10^3	5	10	18	31
5×10^3	4	9	16	27
1×10^4	4	8	16	26
2×10^4	4	8	16	27
5×10^4	6	12	23	38
1×10^5	5	12	22	38

E Implementation Code

The code to implement radial basis summation is listed below:

```
#include <iostream>
#include <cmath>
#include <random>

using namespace std;

class FMM1dNode {
public:
    double lower, upper;
    double *begin, *end;
    double *weightBegin, *weightEnd;
    double *coeff, h;
    int p;
    FMM1dNode *lChild = nullptr, *rChild = nullptr;

    FMM1dNode(double lower, double upper, double *begin,
       double *end, double *weightBegin, double *
       weightEnd,
              double h, int p) {
        this->lower = lower, this->upper = upper, this->
           begin = begin, this->end = end,
        this->weightBegin = weightBegin, this->weightEnd
           = weightEnd, this->h = h, this->p = p, coeff =
            new double[p+2]();
    }

    void buildCoeff() {
        if (lChild == nullptr)
            for (double *it = begin, *weightIt =
               weightBegin; it != end; it++, weightIt++)
               {
                coeff[0] += *weightIt;
                coeff[1] -= *weightIt * (*it - (lower +
                   upper) / 2);
                int *binom = new int[p]();
                for (int i = 0; i < p; i++)
                    binom[i] = 1;
                double alpha = *weightIt, t = *it - (
                   lower + upper) / 2;
                for (int i = 1; i <= p; i += 2) {
                    alpha *= (.5 - (i >> 1)) / ((i >> 1)
                       + 1) * h * h;
                    double temp = alpha;
                    for (int j = 0; i + j <= p; j++)
                        coeff[i + j + 1] += binom[j] *
                           temp, temp *= t;
```

```
                for (int j = 0; j < 2; j++)
                    for (int k = 1; k < p; k++)
                        binom[k] += binom[k - 1];
            }
        }
    else {
        double t1 = (lower - upper) / 4, t2 = (upper
            - lower) / 4;
        coeff[0] = lChild->coeff[0] + rChild->coeff
            [0];
        coeff[1] = lChild->coeff[1] + rChild->coeff
            [1] - t1 * lChild->coeff[0] - t2 * rChild
            ->coeff[0];
        int *binom = new int[p]();
        for (int i = 0; i < p; i++)
            binom[i] = 1;
        for (int i = 1; i <= p; i++) {
            double temp1 = lChild->coeff[i + 1],
                temp2 = rChild->coeff[i + 1];
            for (int j = 0; i + j <= p; j++)
                coeff[i + j + 1] += binom[j] * (temp1
                    + temp2), temp1 *= t1, temp2 *=
                    t2;
            for (int k = 1; k < p; k++)
                binom[k] += binom[k - 1];
        }
    }

}

double evaluate(double x) {
    if (upper * 2 - lower <= x || lower * 2 - upper
        >= x) {
        double middle = (upper + lower) / 2;
        double result = 0, x0 = 1. / (x - middle);
        for (int i = p + 1; i >= 1; i--)
            result *= x0, result += coeff[i];
        result += coeff[0] * (x - middle);
        if (x > middle)
            return result;
        else
            return -result;
    } else if (lChild == nullptr) {
        double result = 0;
        for (double *it = begin, *weightIt =
            weightBegin; it != end; it++, weightIt++)
            result += *weightIt * sqrt((x - *it) * (x
                - *it) + h * h);
        return result;
    } else
```

```
            return lChild->evaluate(x) + rChild->evaluate
                (x);
    }

    ~FMM1dNode() {
        delete coeff;
    }
};

class FMM1d {
public:
    int M, p;
    double eps;
    FMM1dNode *nodes;

    FMM1d(double *begin, double *end, double *weightBegin
        , double *weightEnd, double h, double eps) :
            M((int) round(log2(end - begin))), eps(eps) {
        double weightAll = 0;
        for (auto iter = weightBegin; iter != weightEnd;
            iter++)
            weightAll += abs(*iter);
        p = int(log2(8 * weightAll / eps) / log2(3 / sqrt
            (2)));
        nodes = (FMM1dNode *) operator new(sizeof(
            FMM1dNode) * (1 << (M + 1)));
        for (int i = 1; i < 1 << (M + 1); i++) {
            FMM1dNode *node = &nodes[i];
            if (i == 1)
                new(node) FMM1dNode(0, 1, begin, end,
                    weightBegin, weightEnd, h, p);
            else {
                FMM1dNode *parent = &nodes[i / 2];
                if (i & 1)
                    new(node) FMM1dNode((parent->lower +
                        parent->upper) / 2, parent->upper,
                         (node - 1)->end, parent->end, (
                        node - 1)->weightEnd, parent->
                        weightEnd, h, p), parent->rChild =
                         node;
                else {
                    double *mid = lower_bound(parent->
                        begin, parent->end, (parent->lower
                         + parent->upper) / 2);
                    new(node) FMM1dNode(parent->lower, (
                        parent->lower + parent->upper) /
                        2, parent->begin, mid, parent->
                        weightBegin, mid - parent->begin +
                         parent->weightBegin, h, p),
                        parent->lChild = node;
```

```
                }
            }
        }
        for (int i = (1 << (M + 1)) - 1; i >= 1; i--)
            nodes[i].buildCoeff();
    }

    double evaluate(double x) {
        return nodes[1].evaluate(x);
    }

    ~FMM1d() {
        delete nodes;
    }

};


class BruteForce {
public:
    double *begin, *end;
    double *weightBegin, *weightEnd;
    double h;

    BruteForce(double *begin, double *end, double *
       weightBegin, double *weightEnd, double h) :
            begin(begin), end(end), weightBegin(
               weightBegin), weightEnd(weightEnd), h(h)
               {}

    double evaluate(double x) {
        double result = 0;
        for (double *it = begin, *weightIt = weightBegin;
            it != end; it++, weightIt++)
            result += *weightIt * sqrt((x - *it) * (x - *
               it) + h * h);
        return result;
    }
};

int main() {
    int NList[10] = {100, 200, 500, 1000, 2000, 5000,
       10000, 20000, 50000, 100000};
    for (int N:NList) {
        double *place = new double[N], *weight = new
           double[N];
        for (int i = 0; i < N; i++)
            place[i] = (double) i / N;
        double h = 1. / N;
        default_random_engine generator;
```

```
normal_distribution<double> distribution(0, 1. /
    N);
for (int i = 0; i < N; i++)
    weight[i] = distribution(generator);
uniform_real_distribution<double> uniform(0, 1);
double *test = new double[1000];
for (int i = 0; i < 1000; i++)
    test[i] = uniform(generator);

auto time = chrono::high_resolution_clock::now();
BruteForce MyBruteForce(place, place + N, weight,
     weight + N, h);
cout << (chrono::high_resolution_clock::now() -
    time).count() << endl;
double *bruteForceResult = new double[1000];
time = chrono::high_resolution_clock::now();
for (int i = 0; i < 1000; i++)
    bruteForceResult[i] = MyBruteForce.evaluate(
        test[i]);
cout << (chrono::high_resolution_clock::now() -
    time).count() / 1000
     << endl, time = chrono::
         high_resolution_clock::now();

FMM1d MyFMM1(place, place + N, weight, weight + N
    , h, 0.01);
cout << (chrono::high_resolution_clock::now() -
    time).count() << endl;
double *FMMResult1 = new double[1000];
time = chrono::high_resolution_clock::now();
for (int i = 0; i < 1000; i++)
    FMMResult1[i] = MyFMM1.evaluate(test[i]);
cout << (chrono::high_resolution_clock::now() -
    time).count() / 1000
     << endl, time = chrono::
         high_resolution_clock::now();

FMM1d MyFMM2(place, place + N, weight, weight + N
    , h, 0.0001);
cout << (chrono::high_resolution_clock::now() -
    time).count() << endl;
double *FMMResult2 = new double[1000];
time = chrono::high_resolution_clock::now();
for (int i = 0; i < 1000; i++)
    FMMResult2[i] = MyFMM2.evaluate(test[i]);
cout << (chrono::high_resolution_clock::now() -
    time).count() / 1000
     << endl, time = chrono::
         high_resolution_clock::now();
```

```
        FMM1d MyFMM3(place, place + N, weight, weight + N
            , h, 0.000001);
        cout << (chrono::high_resolution_clock::now() -
            time).count() << endl;
        double *FMMResult3 = new double[1000];
        time = chrono::high_resolution_clock::now();
        for (int i = 0; i < 1000; i++)
            FMMResult3[i] = MyFMM3.evaluate(test[i]);
        cout << (chrono::high_resolution_clock::now() -
            time).count() / 1000
             << endl, time = chrono::
                high_resolution_clock::now();

        FMM1d MyFMM4(place, place + N, weight, weight + N
            , h, 0.00000001);
        cout << (chrono::high_resolution_clock::now() -
            time).count() << endl;
        double *FMMResult4 = new double[1000];
        time = chrono::high_resolution_clock::now();
        for (int i = 0; i < 1000; i++)
            FMMResult4[i] = MyFMM4.evaluate(test[i]);
        cout << (chrono::high_resolution_clock::now() -
            time).count() / 1000
             << endl, time = chrono::
                high_resolution_clock::now();

        double error1 = 0, error2 = 0, error3 = 0, error4
            = 0;
        for (int i = 0; i < 1000; i++)
            error1 = max(error1, abs(bruteForceResult[i]
                - FMMResult1[i])),
            error2 = max(error2, abs(bruteForceResult[i]
                - FMMResult2[i])),
            error3 = max(error3, abs(bruteForceResult[i]
                - FMMResult3[i])),
            error4 = max(error4, abs(bruteForceResult[i]
                - FMMResult4[i]));
        cout << error1 << endl << error2 << endl <<error3
            << endl << error4 << endl;
    }
    return 0;
}
```

References

[1] John Allison. "Multiquadric radial basis functions for representing multidimensional high energy physics data". In: *Computer Physics Communications* 77.3 (1993), pp. 377 –395. ISSN: 0010-4655. DOI: https://doi.org/10.1016/0010-4655(93)90184-E. URL: http://www.sciencedirect.com/science/article/pii/001046559390184E.