# Bit-Cost Analysis, Implementation and Empirical Evaluation of the Fast Multipole Method for Trummer's Problem

Master-Thesis von Bastian Dörig

Tag der Einreichung:

1. Gutachten: Prof. Dr. Marc Pfetsch
2. Gutachten: Prof. Dr. Martin Ziegler

TECHNISCHE UNIVERSITÄT DARMSTADT

Discrete Optimization

# Bit-Cost Analysis, Implementation and Empirical Evaluation of the Fast Multipole Method for Trummer's Problem

Vorgelegte Master-Thesis von Bastian Dörig

1. Gutachten: Prof. Dr. Marc Pfetsch
2. Gutachten: Prof. Dr. Martin Ziegler

Tag der Einreichung:

# Statement of authorship

I, Bastian Dörig, hereby certify that this master thesis has been composed by myself, and describes my own work, unless otherwise acknowledged in the text. All references and verbatim extracts have been quoted, and all sources of information have been specifically acknowledged. It has not been accepted in any previous application for a degree.

# Acknowledgment

First and foremost I would particularly like to thank my supervisor Prof. Dr. Martin Ziegler for introducing me to this very interesting subject of my master thesis and the supervision of it, as well as the invitation to KAIST in the Republic of Korea. During my stay I have learned a lot and was able to actively research this topic. The detailed discussions with him were always very helpful and expedient. Giving me access to his benchmark computer helped me a lot during my extensive benchmarks.
Moreover I would like to thank Prof. Dr. Marc Pfetsch for his immediate willingness for reviewing my master thesis and for lecturing me the basics of Complexity Theory.
Most importantly, I am thanking my family, my friends, and my girlfriend, Fabiana Lang, who supported me during my studies and while researching this topic.

# Abstract

We will introduce a quasilinear, in terms of bit-cost, and implementation friendly version of the *Fast Multipole Method* (FMM) in order to solve the real version of Trummer's Problem, which is closely related to the N-Body Simulation Problem with pairwise Coulomb forces, whereas our implementation does *not* use floating point values, i.e. doubles, which would cause rounding errors during arithmetic operations, but uses only long integers which allow exact addition, subtraction and multiplication. While solving the differential equation for the previously mentioned problem numerically one has to solve Trummer's Problem in each iteration of the Euler Method. Trummer's Problem is also a common subproblem in many other applications ranging from nuclear physics, computational biology, computation with structured matrices, etc.

**Definition 1 (Trummer's Problem).** Let $q := (q_1, \ldots, q_N) \in \mathbb{R}^N$ be a vector of $N$ real values, let $z := (z_1, \ldots, z_N) \in \mathbb{R}^N$ be a vector of $N$ real values and let $C(q, z)$ be the *Cauchy-Matrix* defined by $C(q, z)_{i,j} := \frac{q_i}{z_i - z_j}$ for $i \neq j$ and $C(q, z)_{i,i} := 0$. The real version of *Trummer's Problem* is the computation of the matrix vector product $C(q, z) \cdot q$ up to a given error $2^{-\mathbb{E}}$.

The naive algorithm, multiplying each row of the matrix with the vector, has a quadratic unit-cost complexity. The first quasilinear, in terms of unit cost, solution to Trummer's Problem was given by Gerasoulis et al. [8] by using fast polynomial interpolation and evaluation techniques. Further investigations [19] indicate that this approach has a quadratic bit-complexity. We have shown that, independent of the approach, Trummer's Problem has a quadratic lower bound indeed. When working with real numbers and real complexity theory [17, 18] the Fast Multipole Method [26] applied on a Well-Separated Pair Decomposition [5] turns out to have quasilinear bit-complexity for solving Trummer's Problem.
The main idea of the Fast Multipole Method is to approximate the problem with Laurent and Taylor series. These series get manipulated throughout the algorithm and evaluated at the end. The manipulation includes: truncation of Laurent and Taylor series to a reasonably order, which depends on the given error tolerance $2^{-\mathbb{E}}$, combination of multiple Laurent and Taylor series, and conversion between Laurent and Taylor series.
The analysis shows that these manipulations require a fast multiplication of polynomials which uses a Fast Fourier Transform (FFT) internally. There are libraries which allow a fast multiplication of polynomials with floating point coefficients, but they all lack a rigorous error analysis.
Since we only deal with integers we need a multiplication of integer polynomials. Therefore we need an integer FFT, which is far more complicated than the continuous FFT. In order to adapt the continuous FFT to an integer FFT we need an algebraic field extension of the integers by adjunction of unit roots. Schönhage and Strassen [29] have accomplished this adaption in order to develop the fast integer multiplication, which is available in some libraries as well. This multiplication is crucial for a fast integer polynomial multiplication.
But to the best of our knowledge there exist no implementation combining both, i.e. multiplication of integer polynomials.
As stated by Eric Darve [6] the

> [...] implementation of the FMM proved to be a rather difficult task in part because of its complexity (many lines in a complex and long code) and because of the need to optimize all the steps of the FMM.

Therefore we try to reduce the implementation complexity on all costs without losing performance. For the ease of implementation we do not implement the complicated multiplication of integer polynomials, but we show that we can transform this problem to

Parts of this thesis have already been presented at the fifteenth International Conference on Computability and Complexity in Analysis in 2018.

a simple multiplication of long integers which has quasilinear bit-complexity. The idea to multiply two polynomials with integer coefficients is by embedding the coefficients into long integers and multiply them. During the multiplication one has to prevent overflows, hence the embedded coefficients need a safety distance. By choosing a safety distance of 3, i.e. each coefficient gets embedded into a block with 3 digits, we get

$$(\underbrace{5}_{005} + \underbrace{2x}_{002}) \cdot (\underbrace{6}_{006} + \underbrace{4x}_{004}) = \underbrace{30}_{030} + \underbrace{32x}_{032} + \underbrace{8x^2}_{008}$$
$$\overbrace{5}^{5}\overbrace{002}^{2x} \cdot \overbrace{6}^{6}\overbrace{004}^{4x} = \overbrace{30}^{30}\overbrace{032}^{32x}\overbrace{008}^{8x^2},$$

which both evaluate the same. Therefore the need to implement a FFT to multiply polynomials and switch between different data structures is not necessary any more. N.B. The FFT is still crucial to obtain quasilinear complexity, but the implementation is moved to the long integer library, e.g. the GNU Multiple Precision Arithmetic Library [12] contains a hybrid algorithm which uses the FFT for very long integers. These integers will be used throughout the complete algorithm, hence the data management becomes clearer. We keep track of all error bounds on our own and use long integers to encode the real values, instead of relying on other frameworks for exact real arithmetic [22]. This should result in faster runtime and tighter error bounds. Analysis of the complete algorithm using the above representation in fact turns out to have quasilinear bit-cost complexity.

# Contents

Chapter 1

# Introduction

The problem we are going to solve in this thesis was stated 1985 by Prof. Manfred Trummer and published by Gene Golub [10] in the same year.

> The other day we had a seminar by Manfred Trummer of UBC (soon MIT). The following problem arose: We are given $2n$ distinct elements $c_i, i = 1, \ldots, 2n$. Let $b_{i,j} = (c_i - c_j)^{-1}$ when $i \neq j$ else $b_{i,j} = 0$. Let $Y$ be a vector with $2n$ components.
>
> *TRUMMER'S PROBLEM*
>
> Give an algorithm which depends on the form of $B$ for computing $B \times Y$ in less than $\mathcal{O}(n^2)$ multiplications. If this is impossible, show that it can't be done.
> Reward: \$100

One and a half year later Gerasoulis, Grigoriadis and Liping Sun published an algorithm fulfilling this task in $\mathcal{O}_1(N \log n)$ [8], measured in unit cost, using fast polynomial interpolation and evaluation techniques. But the big problem of fast polynomial arithmetic — it is numerically unstable [19], therefore should not be used in practice. This result conjectures, that using fast polynomial arithmetic to solve Trummer's Problem is quadratic in $N$, measured in bit cost. Last year Tsigaridas and Pan [24] have shown that this approach is quadratically in $N$ indeed.
We have shown that solving Trummer's Problem exactly, i.e. outputting two integers $a$ and $b$, such that $\frac{a}{b}$ is the exact solution, has a lower bound of $\Omega_2(N^2)$, measured in bit cost, cf. section 2.1. In order to achieve subquadratic algorithms we have to drop the requirement of an exact solution. Therefore we will strive for an algorithm, which computes the solution up to an arbitrary error. By using this approach we are also able to allow real and complex numbers as input, instead of only integers and rationals. Therefore we will state the following task

> We are given $2n$ distinct elements $c_i \in \mathbb{C}, i = 1, \ldots, 2n$. Let $b_{i,j} = (c_i - c_j)^{-1}$ when $i \neq j$ else $b_{i,j} = 0$. Let $Y$ be a vector with $2n$ components.
>
> *real version of TRUMMER'S PROBLEM*
>
> Give an algorithm which depends on the form of $B$ for computing $B \times Y$ in less than $\mathcal{O}_2(n^2)$ runtime, measured in bit cost. If this is impossible, show that it can't be done.

In the next sections we will restate Trummer's Problem, in section 1.1, which is needed throughout this thesis, give a short recap about (real) complexity theory in section 1.2 and some motivations in section 1.3 why Trummer's Problem is not only academically very interesting, but also very interesting in practice.
Chapter 2 is dedicated to compute the lower bound for both versions of Trummer's Problem, namely the exact discrete version and the real version. We develop the main ideas of the Fast Multipole Method in chapter 3 and give an in depth analysis of this algorithm in chapter 4. In chapter 5 we will develop a nearly optimal algorithm to solve Trummer's Problem. The next two chapters deal with the implementation, cf. chapter 6, and the empirical evaluation of the implementation, cf. chapter 7. Chapter 8 rounds out the thesis with a summary and an outlook.

## 1.1 Trummer's Problem

We restate the previous problem in a formulation we will need later on.

**Definition 2.**

- Let $q := (q_1, \dots, q_N) \in \mathbb{C}^N$ be a vector of $N$ complex values.
- Let $z := (z_1, \dots, z_N) \in \mathbb{C}^N$ be a vector of $N$ complex values.
- Let $C(q, z)$ be the *Cauchy-Matrix* defined by $C(q, z)_{i,j} := \frac{q_i}{z_i - z_j}$ for $i \neq j$ and $C(q, z)_{i,i} := 0$.

$$C(q,z) = \begin{pmatrix} 0 & \frac{q_1}{z_1-z_2} & \cdots & \frac{q_1}{z_1-z_N} \\ \frac{q_2}{z_2-z_1} & 0 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \frac{q_{N-1}}{z_{N-1}-z_N} \\ \frac{q_N}{z_N-z_1} & \cdots & \frac{q_N}{z_N-z_{N-1}} & 0 \end{pmatrix}$$

*Trummer's Problem* is the matrix vector product

$$C(q, z) \cdot q.$$

Therefore the $i$-th element of $C(q, z) \cdot q$ is defined by

$$(C(q,z) \cdot q)_i = \sum_{\substack{j=1 \\ j \neq i}}^{N} \frac{q_i q_j}{z_i - z_j}.$$

## 1.2 Complexity Theory

This section should be a short recap about discrete and real complexity theory. We will omit too technical notations and details, e.g. we won't introduce Turing Machines or computability in general. A good introduction to discrete complexity theory can be found in [1, 9, 25, 31] and for the real case [17, 18, 33] should be emphasized.
In complexity theory we want to estimate the runtime of an algorithm. We are interested in worst case estimations, i.e. over estimating the runtime, such that it holds for all possible inputs. In general we want to know the asymptotic runtime, i.e. the behavior of the runtime, when the size of the problem grows to infinity. In the next sections we will formalize the asymptotic behavior of a function and define the discrete and real complexity.

### 1.2.1 Asymptotic Behavior of Functions — The Big $\mathcal{O}$ Notation

Let $f, g : \mathbb{N} \to \mathbb{R}$ be functions, mapping integers to reals. We say $f \in \mathcal{O}(g)$, iff for a fixed value $M \in \mathbb{R}$, $M \cdot g$ dominates $f$ almost everywhere. More formally that means, there exists a real number $M$ and an integer $n_0$, such that $f(n) < M \cdot g(n)$ for all $n > n_0$. Informally speaking $f$ does not grow faster than $g$. The following facts can be proven very easily with the above definition and are only stated to give an better intuition of the big $\mathcal{O}$ Notation.

- Scalar Multiplication: If $f \in \mathcal{O}(g)$, then for every $\lambda \in \mathbb{R}$, $\lambda \cdot f \in \mathcal{O}(g)$ follows.
- Addition: If $f_1, f_2 \in \mathcal{O}(g)$, then $f_1 + f_2 \in \mathcal{O}(g)$ follows.
- Linear Combination: For $f_1, \dots, f_N \in \mathcal{O}(g)$ and $\lambda_1, \dots, \lambda_N \in \mathbb{R}$, then $\sum_{i=1}^{N} \lambda_i \cdot f_i \in \mathcal{O}(g)$ follows.
- Transitivity: If $h \in \mathcal{O}(f)$ and $f \in \mathcal{O}(g)$, then $h \in \mathcal{O}(g)$ follows.

For example $0.000001x^2 + 99999x \in \mathcal{O}(x^2)$ — a polynomial is dominated by its leading coefficient. Later in the thesis we will omit polylogarithmic terms, which could distract from the interesting aspects and to simplify the expressions. Therefore we will sometimes use $\tilde{\mathcal{O}}$ to emphasize that we have omitted polylogarithmic terms. We will also use $g \in \Omega(f)$ to denote that $g$ does not grow slower than $f$, therefore we can define $\Omega$ by using the equivalence

$$g \in \Omega(f) \iff f \in \mathcal{O}(g).$$

In order to say that two functions grow asymptotically with the same speed we use $f \in \Theta(g)$, which is equivalent to $g \in \Theta(f)$. We define $\Theta$ by using both previous definitions

$$f \in \Theta(g) \iff f \in \mathcal{O}(g) \text{ and } f \in \Omega(g).$$

### 1.2.2 Discrete Complexity Theory

In discrete Complexity Theory we want to estimate the runtime of an algorithm, which input and output can be encoded finitely, e.g. the input could be integers and the output could be rationals. This is the case for Trummer's Problem, when we restrict the input to be integers, the output will still be rational. The next question is how to compute the runtime of an algorithm. Since the time of an implementation of an algorithm to terminate will depend on the specific computer and hardware it uses, we should abstract from the machine. A first abstraction is called unit cost, where the measure is defined by counting the number of arithmetic operations, e.g. addition and multiplication of numbers. When dealing with unit cost we will use a 1 as a subscript in the big $\mathcal{O}$ Notation, i.e. $\mathcal{O}_1$. This abstraction is good when dealing with floating point arithmetic of fixed size. But when dealing with numbers of arbitrary size, in reality the time of an arithmetic operation depends on the size of the numbers. Therefore we should not count the number of arithmetic operations, but the number of bit operations. This leads to the definition of bit complexity. In the big $\mathcal{O}$ Notation we will use 2 as a subscript to denote bit complexity, i.e. $\mathcal{O}_2$. But since we will only use bit complexity later on, we will omit this subscript in most cases. The bit complexity is a more realistic measure of time, because it describes the mechanics of a real computer better. To give some examples, we are looking at the arithmetic operations addition and multiplication: Let $N_1, N_2 \in \mathbb{N}$ be two integers. Both integers can be encoded using $n_1 = \log N_1$, resp. $n_2 = \log N_2$ bits which is called the size of the encoding and let $n = \max(n_1, n_2)$ be the maximum of both encodings.

- The addition of $N_1, N_2$ is exactly one arithmetic operation, therefore the runtime is constant w.r.t unit cost, i.e. $+ \in \mathcal{O}_1(1)$.
- The addition of $N_1, N_2$ needs linearly many bit operations, therefore the runtime is linear, w.r.t. bit cost, i.e. $+ \in \mathcal{O}_2(n)$.
- The multiplication of $N_1, N_2$ is exactly one arithmetic operation, therefore the runtime is constant w.r.t unit cost, i.e. $* \in \mathcal{O}_1(1)$.
- In bit cost, the multiplication of $N_1, N_2$ is more interesting. Using the written multiplication method, we know since primary school, we need quadratic many bit operations, hence $* \in \mathcal{O}_2(n^2)$. Using the sophisticated Schönhage–Strassen algorithm we can multiply in subquadratic runtime $* \in \mathcal{O}_2(n \cdot \log n \cdot \log \log n)$, w.r.t. bit cost. By omitting polylogarithmic terms, addition and multiplication are both linear.

### 1.2.3 Real Complexity Theory

Real Complexity Theory provides a computer-scientific foundation to Numerics bridging from Recursive Analysis to practice. [17]

When dealing with real numbers we can simply carry over the unit cost measurements, by defining a machine, the Blum–Shub–Smale machine (BSS machine) [3], which is capable to do arithmetic over the reals. But it is quite obvious that this machine is far from reality, because it is not even possible to encode a real number with finitely many bits. We choose real numbers, which can be approximated by a fast converging sequence of dyadic rationals. Therefore we will encode a real number $r \in \mathbb{R}$ by a sequence of integers $z_n \in \mathbb{Z}$, such that for all $n \in \mathbb{N}$ the $n$-th number divided by $2^n$ is a approximation up to error $2^{-n}$, which means each additional number gives an additional correct bit in the dyadic encoding. Formally this means $|\frac{z_n}{2^n} - r| < 2^{-n}$ for all $n \in \mathbb{N}$. Using this idea we are able to define the bit complexity for real numbers, by doing the same as in the discrete case, but we use an additional parameter — the error exponent $\mathbb{E} \in \mathbb{N}$, which bounds the error of the approximated result of the algorithm $x$ to the exact result $r \in \mathbb{R}$, i.e. $|x - r| < 2^{-\mathbb{E}}$. Therefore this algorithm will return a sequence of numbers $x_i$ which converge fast to the exact result $r$ and therefore encode this number. Throughout the thesis we will use this definition to estimate the runtime of our algorithms. To give some examples, we are looking at the arithmetic operations addition and multiplication again: Let $x, y \in \mathbb{R}$ be two reals with the corresponding series $(x_i)_i, (y_i)_i \in \mathbb{N}^{\mathbb{N}}$, such that $|\frac{x_i}{2^n} - x| < 2^{-n}, |\frac{y_i}{2^n} - y| < 2^{-n}$. Let $\mathbb{E} \in \mathbb{N}$ be the error exponent for the desired accuracy.

- The addition of $x, y$ is exactly one arithmetic operation, therefore the runtime is constant w.r.t unit cost (using the BSS machine), i.e. $+ \in \mathcal{O}_1(1)$.
- For the addition of $x, y$, we choose the numbers $x_{\mathbb{E}+1}, y_{\mathbb{E}+1}$ and add both, the result has the desired accuracy. The encoding length of both numbers is linear in $\mathbb{E}$, therefore we see the runtime is linear, w.r.t. bit cost, i.e. $+ \in \mathcal{O}_2(\mathbb{E})$.
- The multiplication of $x, y$ is exactly one arithmetic operation, therefore the runtime is constant w.r.t unit cost (using the BSS machine), i.e. $* \in \mathcal{O}_1(1)$.
- Analogous computations show that the multiplication of $x, y$ is subquadratic in $\mathbb{E}$, i.e. $* \in \mathcal{O}_2(\mathbb{E} \cdot \log \mathbb{E} \cdot \log\log \mathbb{E})$, w.r.t. bit cost. By omitting polylogarithmic terms, addition and multiplication are both linear.

## 1.3 Motivation

In this section we will give further motivation, why Trummer's Problem is interesting. Therefore we have chosen two applications — the N-Body Simulation in section 1.3.1 and computations with structured matrices in section 1.3.2. In order to show that our rigorous approach is important for practice we have implemented another algorithm which uses only the CPP standard implementation for floating points **double** in section 1.3.3. In this section we show the numerically instabilities, i.e. huge errors resulting from the floating point operations and that our approach does not have these obstacles.

### 1.3.1 Physical View: N-Body Simulation

Consider the problem — you got 2 charged particles on a plane, i.e. in 2 real dimensions. The charges will be denoted by $q_1, q_2 \in \mathbb{R}$ and the positions by $\begin{pmatrix} x_1 \\ y_1 \end{pmatrix}, \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} \in \mathbb{R}^2$. The induced force onto particle 2 is given by the Coulomb force (in 2 dimensions)

$$\begin{aligned}\begin{pmatrix} F_x \\ F_y \end{pmatrix} &= \overbrace{\frac{1}{2\pi\varepsilon}}^{\text{prefactor}} \cdot \overbrace{\frac{q_1 \cdot q_2}{\left| \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} - \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} \right|}}^{\text{coupling of both particles}} \cdot \overbrace{\frac{\begin{pmatrix} x_1 \\ y_1 \end{pmatrix} - \begin{pmatrix} x_2 \\ y_2 \end{pmatrix}}{\left| \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} - \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} \right|}}^{\substack{\text{unit vector denoting} \\ \text{the direction of the force}}} \\ &= c \cdot \frac{q_1 \cdot q_2}{(x_1 - x_2)^2 + (y_1 - y_2)^2} \cdot \begin{pmatrix} x_1 - x_2 \\ y_1 - y_2 \end{pmatrix},\end{aligned}$$

Instead of encoding the positions in the real plane by $\begin{pmatrix} x \\ y \end{pmatrix} \in \mathbb{R}^2$, we will encode them into the complex plane by $x + Iy \in \mathbb{C}$. Then we see the following instead

$$\begin{aligned} c \cdot I \cdot \frac{q_1 \cdot q_2}{(x_1 + Iy_1) - (x_2 + Iy_2)} &= c \cdot I \cdot \frac{q_1 \cdot q_2}{(x_1 - x_2) + I(y_1 - y_2)} \\ &= c \cdot I \cdot \frac{q_1 \cdot q_2}{((x_1 - x_2) + I(y_1 - y_2)) \cdot ((x_1 - x_2) - I(y_1 - y_2))}((x_1 - x_2) - I(y_1 - y_2)) \\ &= c \cdot I \cdot \frac{q_1 \cdot q_2}{(x_1 - x_2)^2 + (y_1 - y_2)^2}((x_1 - x_2) - I(y_1 - y_2)) \\ &= c \cdot \frac{q_1 \cdot q_2}{(x_1 - x_2)^2 + (y_1 - y_2)^2}((y_1 - y_2) + I(x_1 - x_2)), \end{aligned}$$

which is equivalent to $\begin{pmatrix} F_y \\ F_x \end{pmatrix}$.
Summation over multiple charged particles shows that computing all pairwise Coulomb forces of particles in two real dimensions is equivalent to Trummer's Problem in the complex case. When simulating multiple charged particles one has to solve the differential equation $F(x) = m\frac{dx}{dt}$, whereas $F$ is the Coulomb forces introduced before. Solving this equation numerically, e.g. via the Euler Method, one has to compute all pairwise Coulomb forces in each iteration. Therefore Trummer's Problem is the key subproblem for N-Body simulations and faster algorithms for Trummer's Problem would result in faster algorithms for N-Body simulation.
As stated by Jack Dongarra and Francis Sullivan [32]

> (the Fast Multipole Method belongs to) the 10 algorithms with the greatest influence on the development and practice of science and engineering in the 20th century.

Which shows that the Fast Multipole Method is very important for practice. Four years earlier Reif and Tate [26] cited a study which shows that this problem is relevant in practice and needs a lot of computing resources

> a study [...] showed that over 30 percent of all compute time on their CRAY-YMP was used for n-body simulation by molecular chemists.

### 1.3.2 Computation with Structured Matrices

Another interesting property occurs when computing with Structured Matrices.

**Definition 3 (Structured Matrices).** The four structured matrices, Toeplitz, Hankel, Vandermonde and Cauchy matrices are defined in table table 1.1. Pan and Tsigaridas have shown [24] that the computation with three of these four of these matrices is subquadratic in $n$, namely computation with Toeplitz, Hankel and Vandermonde matrices, whereas computation involving Cauchy matrices is still quadratic in $N$. In this case computation means

- Matrix Vector Multiplication, cf. Trummer's Problem
- Inversion of a Matrix
- Solution of nonsingular linear systems of equations with these matrices

These operations should be possible in subquadratic runtime using our algorithm. Therefore these families of structured matrices all behave similar and the Cauchy matrices do not have a special role, which increases their complexity somehow. But these

Table 1.1.: Definition of Toeplitz, Hankel, Vandermonde and Cauchy matrices [24].

| | |
|---|---|
| Toeplitz matrices $T = (t_{i-j})_{i,j=0}^{n-1}$ $\begin{pmatrix} t_0 & t_{-1} & \dots & t_{1-n} \\ t_1 & t_0 & \ddots & \vdots \\ \vdots & \ddots & \ddots & t_{-1} \\ t_{n-1} & \dots & t_1 & t_0 \end{pmatrix}$ | Hankel matrices $H = (h_{i+j})_{i,j=0}^{n-1}$ $\begin{pmatrix} h_0 & h_1 & \dots & h_{n-1} \\ h_1 & h_2 & \iddots & \vdots \\ \vdots & \iddots & \iddots & t_{-1} \\ h_{n-1} & h_n & \dots & h_{2n-2} \end{pmatrix}$ |
| Vandermonde matrices $V = V_s = (s_i^{j-1})_{i,j=1}^{n}$ $\begin{pmatrix} 1 & s_1 & \dots & s_1^{n-1} \\ 1 & s_2 & \dots & s_2^{n-1} \\ \vdots & \vdots & & \vdots \\ 1 & s_n & \dots & s_n^{n-1} \end{pmatrix}$ | Cauchy matrices $C = C_{s,t} = \left(\frac{1}{s_i - t_j}\right)_{i,j=1}^{n}$ $\begin{pmatrix} \frac{1}{s_1-t_1} & \frac{1}{s_1-t_2} & \dots & \frac{1}{s_1-t_n} \\ \frac{1}{s_2-t_1} & \frac{1}{s_2-t_2} & \dots & \vdots \\ \vdots & \vdots & & \vdots \\ \frac{1}{s_n-t_1} & \frac{1}{s_n-t_2} & \dots & \frac{1}{s_n-t_n} \end{pmatrix}$ |

results can be generalized to an even larger family of structured matrices, e.g. to transposed Vandermonde matrices by the following equality [24]

$$V_s^T = -\frac{f^{1-n}}{v}\mathrm{diag}(f^j)_{j=0}^{n-1}\Omega\mathrm{diag}(\omega_n^j)_{j=0}^{n-1}C_{f,s}\mathrm{diag}(s_i^n - f^n)_{i=0}^{n-1},$$

whereas $f$ is a scalar, $s$ is a vector which defines the Vandermonde matrix $V_s$, $t$ is a vector defining the function $t(x) = \prod_{j=0}^{n-1}(x - t_j)$ and $w$ is a vector defining the coefficients of $w(x) = t(x) - x^n$. This shows, that computation with the transposed of a Vandermonde matrix is equivalent to computations with Cauchy matrices. Therefore we have shown that the computations with transposed Vandermonde matrices are possible in subquadratic runtime, instead of the previous known quadratic runtime. Using similar equalities by Pan and Tsigaridas one can generalize these results to the more general classes of structured matrices with small displacement rank, e.g. Toeplitz, Hankel, Vandermonde and Cauchy like matrices, which are defined by using the associated Sylvester displacement operators of the form $M \mapsto AM - MB$ where the pair of operator matrices $A$ and $B$ represents the matrix structure.

### 1.3.3 Comparison with a naive Implementation using only Doubles

In this section we solve Trummer's Problem by simply evaluating

$$\sum_{\substack{j=1 \\ j \neq i}}^{N} \frac{q_i q_j}{z_i - z_j},$$

whereas only the CPP standard implementation for floating points **double** is used for all values.

An alternative implementation to solve Trummer's Problem using only double values.

```
std::map<DoubleParticle*, double> solution;
for (auto const& particle : particles) {
  double result = 0;
  double q1 = particle.charge();
  double z1 = particle.position();
  for (auto const& other : particles) {
    if (particle == other) {
      continue;
    }
    double q2 = other.charge();
    double z2 = other.position();
    result += (q1 * q2) / (z1 - z2);
  }
  solution[&particle] = result;
}
```

We choose the following 7 particles $p_{-3} = (-3, 10^{20}), p_{-2} = (-2, 1), p_{-1} = (-1, -10^{20}), p_0 = (0, 1), p_1 = (1, -10^{20}), p_2 = (2, 1), p_3 = (3, 10^{20})$. Because the problem is symmetric to the particle in the middle $p_0$, i.e. $q_i = q_{-i}, z_i = -z_{-i}$, the resulting force $F_0$ at this particle vanishes

$$\begin{aligned}
F_0 &= \sum_{\substack{j=-3\\ j\neq 0}}^{3} \frac{q_0 q_j}{z_0 - z_j} \\
&= \frac{q_0 q_{-3}}{z_0 - z_{-3}} + \frac{q_0 q_{-2}}{z_0 - z_{-2}} + \frac{q_0 q_{-1}}{z_0 - z_{-1}} + \frac{q_0 q_1}{z_0 - z_1} + \frac{q_0 q_2}{z_0 - z_2} + \frac{q_0 q_3}{z_0 - z_3} \\
&= \frac{q_3}{z_3} + \frac{q_2}{z_2} + \frac{q_1}{z_1} + \frac{q_1}{-z_1} + \frac{q_2}{-z_2} + \frac{q_3}{-z_3} \\
&= \frac{q_3}{z_3} - \frac{q_3}{z_3} + \frac{q_2}{z_2} - \frac{q_2}{z_2} + \frac{q_1}{z_1} - \frac{q_1}{z_1} \\
&= 0.
\end{aligned}$$

By applying the above implementation we get $F_0 = -4096 \neq 0$. We have implemented this example in different online cpp shells which can be found under `https://www.onlinegdb.com/BkezQaF8m`, `http://tpcg.io/ZPxiGl`, `https://wandbox.org/permlink/qb8N58h6DzU4UF3W` and `http://cpp.sh/7l12b` to test different systems and compilers and that everyone is able to reproduce this value very easy. We hope that some of these links live long enough, otherwise simply copy&paste the below code and build it with `g++ prog.cc -Wall -Wextra -std=c++11` to run locally

Fully functional implementation, using only double values, to solve a special instance of Trummer's Problem, with 7 specific particles, to show the numerically instability.

```cpp
#include <iostream>
#include <tuple>
#include <vector>

int main() {
  std::vector<std::pair<double, double>> particles({{-3, 1e20}, {-2, 1}, {-1, -1e20},
                                                    {0, 1}, {1, -1e20}, {2, 1}, {3, 1e20}});
  std::vector<std::pair<double, double>> solution;
  for (auto const& particle : particles) {
    double result = 0;
    double z1 = particle.first;
    double q1 = particle.second;
    for (auto const& other : particles) {
      if (particle == other) {
        continue;
      }
      double z2 = other.first;
      double q2 = other.second;
      result += (q1 * q2) / (z1 - z2);
    }
    solution.emplace_back(z1, result);
  }
  std::cout << "at z=" << solution.at(3).first
            << ", we got the force = " << solution.at(3).second << std::endl;
}
```

When we use our implementation, with the error exponent $\mathbb{E} = 10$, i.e. an absolute error of at most $10^{-3}$, we get $F_0 = 6.33822 \cdot 10^{-29}$. We have only used 7 particles for this example, when we deal with thousands of particles the error could be even worse. This indicates again that Trummer's Problem is numerically unstable. This could yield to enormous problems in practice when there occur large forces which are not existent. This problem shows why our implementation is very interesting for practice — we can avoid such obstacles.

Chapter 2

# Lower Bounds for Trummer's Problem

In this chapter we compute the lower bounds for both versions of Trummer's problem. All results are with respect to bit-cost. In section 2.1 we show that the discrete version of Trummer's Problem has a quadratic lower bound in $N$ and section 2.2 proofs the lower bound to solve the real version of Trummer's Problem is linear in $N$. Therefore all algorithms solving one of these problems can not be faster than the corresponding bounds.

## 2.1 Lower Bound for the Discrete Version of Trummer's Problem

In this section we proof the lower bound to solve the discrete version of Trummer's Problem is quadratic in $N$. To proof this we need some preliminary lemmas and proof our claimed lower bound in theorem 8. In the following we use $p_n$ to denote the $n$th prime.

**Lemma 4.** *Let $N > 2$ and $0 \leq i, j \leq N$ and $k \geq \frac{N}{2}$ and $x_{ij} = (i - j)$, then $k^2$ does not divide $x_{ij}$.*

*Proof.* Let $N > 2$ and $0 \leq i, j \leq N$ and $k \geq \frac{N}{2}$ and $x_{ij} = (i - j)$. By estimating $k^2 \geq \frac{N^2}{4} > N \geq |x_{ij}|$, we get $k^2 > |x_{ij}|$, hence $k^2$ does not divide $x_{ij}$. ☐

**Lemma 5.** *Let $N > 2$, $0 \leq i \leq N$, $x_{ij} = (i - j)$ and define $a_{ij} = \prod_{\substack{k=1\\k\neq j\\k\neq i}}^{N} p_k x_{ik}$.*

*Let $l \geq \frac{N}{2}$, with $l \neq i$. If $p_l^n \mid \sum_{\substack{j=1\\j\neq i}}^{N} a_{ij}$ then there exists a set $K_n^l = \{\, k_o,\ \text{for } o = 1, \ldots, n,\ \text{with } k_o \neq l, k_o \neq k_{o'} \,\}$, such that $p_l \mid x_{ik_o}$ for all $k_o \in K_n^l$.*

*Proof.* In this proof we have to restrict all indices in all sums and and in all products not being $i$. Because $i$ is fixed and too enhance the readability we we omit the restrictions of the indices not being $i$ in the sums and products, e.g. $a_{ij} = \prod_{\substack{k=1\\k\neq j\\k\neq i}}^{N} p_k x_{ik} \longrightarrow \prod_{\substack{k=1\\k\neq j}}^{N} p_k x_{ik}$. Because we require $i \neq l$, we do not divide by the $i$th prime $p_i$, therefore the "index not equal $i$" constraint in all sums and products, is not necessary in order to follow the proof. We also omit $i$ in the subscripts, e.g. $x_j = (i - j)$ and $a_j = \prod_{\substack{k=1\\k\neq j}}^{N} p_k x_k$. Therefore we get lesser indices and the constraints on the indices of the sums and products are shorter, which should simplify following the proof.
Let $N > 2$, and let $0 \leq i \leq N$ be fixed. This lemma will be shown by induction over $n$ for a fixed $l \neq i$.
Induction start: Choose $n = 1$ and let $p_l^1 \mid \sum_{j=1}^{N} a_j$ for an $l \neq i$, then we get

$$\begin{aligned}
\sum_{j=1}^{N} a_j &= \sum_{j=1}^{N} \prod_{\substack{k=1\\k\neq j}}^{N} p_k x_k \\
&= \sum_{\substack{j=1\\j\neq l}}^{N} \prod_{\substack{k=1\\k\neq j}}^{N} p_k x_k + \prod_{\substack{k=1\\k\neq l}}^{N} p_k x_k \\
&= \sum_{\substack{j=1\\j\neq l}}^{N} p_l x_l \prod_{\substack{k=1\\k\neq j\\k\neq l}}^{N} p_k x_k + \prod_{\substack{k=1\\k\neq l}}^{N} p_k x_k.
\end{aligned}$$

Therefore $p_l$ divides the first summand $\sum_{\substack{j=1\\j\neq l}}^{N} p_l x_l \prod_{\substack{k=1\\k\neq j\\k\neq l}}^{N} p_k x_k$. Since $p_l$ divides the sum, it has to divide the second summand $\prod_{\substack{k=1\\k\neq l}}^{N} p_k x_k$ as well. The prime $p_l$ can not divide any of the $p_k$s, because they are all primes and pairwise distinct, hence there exists a $k \neq l$ (and $k \neq i$), such that $p_l \mid x_k$.
Induction step: Choose $n+1$ and let $p_l^{n+1} \mid \sum_{j=1}^{N} a_j$. Since $p_l^{n+1}$ is a factor, $p_l^n$ is a factor as well and we can use the induction hypothesis. Therefore there exists a set $K_n = \{\, k_o, \text{ for } o = 1, \ldots, n, \text{ with } k_o \neq l, k_o \neq k_{o'} \,\}$, such that $p_l \mid x_{k_o}$. Using the set $K_n$ we get

$$\begin{aligned}\sum_{j=1}^{N} a_j &= \sum_{\substack{j=1\\j\neq l}}^{N} p_l x_l \prod_{\substack{k=1\\k\neq j\\k\neq l}}^{N} p_k x_k + \prod_{\substack{k=1\\k\neq l}}^{N} p_k x_k \text{ cf. the induction start}\\ &= \sum_{\substack{j=1\\j\neq l}}^{N} p_l x_l \prod_{\substack{k=1\\k\neq j\\k\neq l\\k\notin K_n}}^{N} p_k x_k \prod_{k_o\in K_n} p_{k_o} x_{k_o} + \prod_{\substack{k=1\\k\neq l\\k\notin K_n}}^{N} p_k x_k \prod_{k_o\in K_n} p_{k_o} x_{k_o}\end{aligned}$$

Therefore we see $p_l^{n+1}$ divides the first summand, e.g. by choosing $p_l, x_{k_o}$ for $k_o \in K_n$. Hence $p_l^{n+1}$ has to divide the second summand $\prod_{\substack{k=1\\k\neq l\\k\notin K_n}}^{N} p_k x_k \prod_{k_o\in K_n} p_{k_o} x_{k_o}$ as well. We see that $p_l^n$ divides the last part, i.e. the $x_{k_o}$. Because $p_{k_o} \neq p_l$ we can not use them for the division, also $p_l^2 \nmid x_{ik_o} = x_{k_o}$, by lemma 4. Hence $p_l$ has to divide $\prod_{\substack{k=1\\k\neq l\\k\notin K_n}}^{N} p_k x_k$. Therefore there exists a $k_{n+1} \neq l, k_{n+1} \notin K_n$, such that $p_l \mid x_{k_{n+1}}$. Then we get $K_{n+1}^l = K_n^l \cup \{\, k_{n+1} \,\}$ which finishes the induction step and our proof. □

**Lemma 6.** *Let* $a_{ij} = \prod_{\substack{k=1\\k\neq j\\k\neq i}}^{N} p_k x_{ik}$. *For each* $i = 1, \ldots, N$ *and for each* $\frac{N}{2} \leq l \leq N$, *with* $l \neq i$ *and after repeated cancellation of* $p_l$ *from* $\frac{\sum_{\substack{j=1\\j\neq i}}^{N} a_{ij}}{\prod_{\substack{k=1\\k\neq i}}^{N} p_k x_{ik}}$ *till* $p_l$ *can not be canceled anymore, the prime* $p_l$ *still divides the denominator.*

*Proof.* We use the same notations from lemma 5 and omit the $i$ subscripts and constraints of the sums and products.
Since $\sum_{j=1}^{N} a_j$ is bounded, there exists a number $M$ such that $p_l^M \nmid \sum_{j=1}^{N} a_j$, e.g. by choosing $M = \sum_{j=1}^{N} a_j$. Therefore there exists a number $n_0$, such that $p_l^{n_0} \mid \sum_{j=1}^{N} a_j$ and $p_l^{n_0+1} \nmid \sum_{j=1}^{N} a_j$.
Since $p_l^{n_0} \mid \sum_{j=1}^{N} a_j$ then, by lemma 5, there exists the set $K_{n_0}^l = \{\, k_o, \text{ for } o = 1, \ldots, n_0, \text{ with } k_o \neq l, k_o \neq k_{o'} \,\}$, such that $p_l \mid x_{k_o}$ for all $k_o \in K_{n_0}^l$. Then we see

$$\prod_{k=1}^{N} p_k x_k = p_l x_l \prod_{\substack{k=1\\k\neq l\\k\notin K_{n_0}^l}}^{N} p_k x_k \prod_{k\in K_n^l} p_k x_k,$$

hence $p_l^{n_0+1} \mid \prod_{k=1}^{N} p_k x_k$ follows directly.
Hence $p_l^{n_0}$ can be canceled out and $p_l^{n_0+1}$ can not be canceled out. After cancellation of $p_l^{n_0}$, the prime $p_l$ does not divide the remaining numerator, but divides the remaining denominator. □

**Lemma 7.** *Let* $a_{ij} = \prod_{\substack{k=1\\k\neq j\\k\neq i}}^{N} p_k x_{ik}$. *For each* $i = 1, \dots, N$ *and after complete cancellation of* $\frac{\sum_{\substack{j=1\\j\neq i}}^{N} a_{ij}}{\prod_{\substack{k=1\\k\neq i}}^{N} p_k x_{ik}}$, *then for the remaining denominator* $D$

$$|D| \geq \prod_{\substack{k=\frac{N}{2}\\k\neq i}}^{N} p_k$$

*holds.*

*Proof.* We use the same notations from lemma 5 and omit the $i$ subscripts and constraints of the sums and products. By using lemma 6 we get for each $\frac{N}{2} \leq l \leq N$ with $l \neq i$, that the prime $p_l$ still divides the denominator $D$ after full cancellation. Therefore $\prod_{\substack{k=\frac{N}{2}\\k\neq i}}^{N} p_k \mid D$, which implies our claim. □

By using the previous lemmas we are finally able to proof the lower bounds of Trummer's Problem.

**Theorem 8.** *Let* $N$ *denote the number of particles and let* $M$ *denote the overall encoding size of the input for the discrete version of Trummer's Problem, i.e. where we choose rationals for the input and the output, which are encoded using two integers, one for the numerator and one for the denominator. N.B.* $N$ *and* $M$ *can be independent. For the runtime* $T(N), T(M)$ *we get the following lower bounds*

$$T(N) \in \Omega(N^2(\ln N + \ln\ln N))$$
$$T(M) \in \Omega(M^2(\ln M + \ln\ln M))$$

*for all algorithms which solve the problem.*

*Proof.* Let $N > 10^{11}$ denote the number of particles. We choose the following particles $(q_k, z_k)_k$ for $k = 1, \dots, N$, with $q_k = \frac{1}{p_k}$ and $z_k = k$. For a fixed $i \in \mathbb{N}$, Trummer's Problem is given by

$$\begin{aligned}
\sum_{\substack{j=1\\j\neq i}}^{N} \frac{q_i q_j}{z_i - z_j} &= \frac{1}{p_i} \sum_{\substack{j=1\\j\neq i}}^{N} \frac{1}{p_j(i-j)} \\
&= \frac{1}{p_i} \frac{\sum_{\substack{j=1\\j\neq i}}^{N} \overbrace{\prod_{\substack{k=1\\k\neq i\\k\neq j}}^{N} p_k \overbrace{(i-k)}^{x_{ik}}}^{a_{ij}}}{\prod_{\substack{j=1\\j\neq i}}^{N} p_j \underbrace{(i-j)}_{x_{ij}}} \\
&= \frac{1}{p_i} \frac{\sum_{\substack{j=1\\j\neq i}}^{N} a_{ij}}{\prod_{\substack{j=1\\j\neq i}}^{N} p_j x_{ij}}.
\end{aligned}$$

Using lemma 7 we see that after complete cancellation of $\sum_{\substack{j=1\\j\neq i}}^{N} \frac{q_i q_j}{z_i - z_j} = \frac{1}{p_i} \frac{\sum_{\substack{j=1\\j\neq i}}^{N} a_{ij}}{\prod_{\substack{j=1\\j\neq i}}^{N} p_j x_{ij}}$, then for the denominator $D$

$$\begin{aligned} |D| &\geq p_i \prod_{\substack{k=\frac{N}{2}\\k\neq i}}^{N} p_k \\ &\geq \prod_{k=\frac{N}{2}}^{N} p_k \end{aligned}$$

holds. Therefore we can estimate the encoding length $L$ of the output $O = \frac{n}{d}$, by using $D$ being the denominator after complete cancellation we get

$$\begin{aligned} L(O) &= \ln|n| + \ln|d| \\ &\geq \ln|D| \\ &\geq \ln \prod_{k=\frac{N}{2}}^{N} p_k \\ &\geq \ln \prod_{k=1}^{\frac{N}{2}} p_k \\ &= \sum_{k=1}^{\frac{N}{2}} \ln p_k \\ &= \theta\left(\frac{N}{2}\right) \\ [7] &\geq \frac{N}{2}\left(\ln\frac{N}{2} + \ln\ln\frac{N}{2}\right), \end{aligned}$$

with $\theta(\cdot)$ being the Chebyshev function [7]. Therefore the encoding for each $i = 1, \ldots, N$ has a lower bound of $\frac{N}{2}\left(\ln\frac{N}{2} + \ln\ln\frac{N}{2}\right)$. Iteration over $i$ shows $T(N) \in \Omega(N^2(\ln N + \ln\ln N))$.
The next step is to compute the lower bound w.r.t. to the encoding size of the input $M$. We will estimate $M$ depending on the number of particles $N$.

$$\begin{aligned} M = {} & \overbrace{1 + \ldots + 1}^{\substack{\text{the size to encode the}\\ \text{numerators of } q_1,\ldots,q_N}} + \overbrace{1 + \ldots + 1}^{\substack{\text{the size to encode the}\\ \text{numerators of } z_1,\ldots,z_N}} + \overbrace{\sum_{i=1}^{N} \ln p_i}^{\substack{\text{the size to encode the}\\ \text{denominators of } q_1,\ldots,q_N}} + \overbrace{\sum_{i=1}^{N} \ln i}^{\substack{\text{the size to encode the}\\ \text{denominators of } z_1,\ldots,z_N}} \\ &\leq N + N + \sum_{i=1}^{N} \ln p_i + \sum_{i=1}^{N} \ln p_i \\ &= 2N + 2\theta(N) \\ [7] &< 2N + 2 \cdot 1.000081N \\ &\leq 5N. \end{aligned}$$

By using the estimation for $T(N)$ and using $M < 5N$ we get $T(M) \in \Omega(M^2(\ln M + \ln\ln M))$, which finishes the proof. □

## 2.2 Lower Bound for the Real Version of Trummer's Problem

In this section we proof the lower bound to solve the real version of Trummer's Problem is linear in $N$. This is interesting in order to rate our developed algorithm, i.e. how good is the algorithm compared to the best possible.

**Theorem 9.** *Let $(\tilde{q}_i, \tilde{z}_i)$, for $i = 1, \dots, N$, denote $N$ particles with $\tilde{q}_i \in [0,1] \subset \mathbb{R}$ and $\tilde{z}_i \in [0,1] \subset \mathbb{R}$. Let $m_s$ be the separation of the $z_i$, i.e. $|z_i - z_j| \geq 2^{-m_s}$.*
*Each algorithm, solving Trummer's Problem, up to an absolute error of $2^{-\mathbb{E}}$ has at least*

$$N\mathbb{E} + Nm_s$$

*runtime.*

*Proof.* We will construct an input, such that we need at least $Nm_s$ bits for the input and at least $N\mathbb{E}$ bits for the output. The claim follows directly.
Let $N \in \mathbb{N}, 1 \leq \mathbb{E} \in \mathbb{N}, 1 \leq m_s \in \mathbb{N}$ be arbitrary but fixed. Define $\tilde{q}_i = 1$, for each $i = 1, \dots, N$ and define $\tilde{z}_i = i \cdot 2^{-m_s}$, for $i = 1, \dots, N$, then the precondition $m_s$ being the separation of the $z_i$, i.e. $|z_i - z_j| \geq 2^{-m_s}$, is full filled. If we would use less than $m_s$ bits to encode the positions, there are two cases:

1. Two positions have the same encoding. Without loss of generality, let $\tilde{z}_1, \tilde{z}_2$ be these two positions, then the encoded positions $z_1, z_2$ are equal. Therefore, $\frac{1}{z_1 - z_2}$ is undefined and we can not solve Trummer's Problem.
2. One encoded position differs more than $2^{-m_s}$ to the original value. Then there exist another position, such that the encoded distance is at least $2 \cdot 2^{-m_s}$, whereas the real distance is only $2^{-m_s}$. The resulting error will be at least $2^{m_s - 1} \geq 2^{-\mathbb{E}}$. We will show this by assuming $z_1$ gets encoded to 0 instead of the real value $2^{-m}$, then we get

$$\Big|\sum_{i=2}^{N} \frac{1}{z_1 - z_i} - \sum_{i=2}^{N} \frac{1}{\tilde{z}_1 - \tilde{z}_i}\Big| \geq \Big|\frac{1}{0 - \frac{2}{2^{-m_s}}} - \frac{1}{\frac{1}{2^{-m_s}} - \frac{2}{2^{-m_s}}}\Big| \qquad = |2^{m_s - 1} - 2^{m_s}| = 2^{m_s - 1} \qquad \geq 2^{-\mathbb{E}}.$$

In both cases we have seen that the error would be too large, hence the input encoding must have at least the length $Nm_s$.
The output encoding must have length $N\mathbb{E}$ at least, otherwise the result can not approximate the exact solution up to error $2^{-\mathbb{E}}$. □

Chapter 3

# Main Idea of the Fast Multipole Method

In this chapter we develop the main ideas of the Fast Multipole Method. We start with different approaches in sections 3.1 to 3.3 by using Taylor series, Laurent series or sums of Laurent series to approximate the problem and finalize the main ideas in section 3.4.

## 3.1 Ansatz: Taylor Series

Trummer's Problem can be rewritten into the problem of evaluating $N$ functions at 1 point each which is shown by using the next reinterpretation.

$$(C(q,z)\cdot q)_i = \sum_{\substack{j=1\\ j\neq i}}^{N} \frac{q_i q_j}{z_i - z_j}$$

Reinterpretation of the $i$-th element of $C(q,z)\cdot q$, given above, yields to evaluation of the function

$${}^{i}\Phi : \mathbb{C} \to \mathbb{C} : z \mapsto \sum_{\substack{j=1\\ j\neq i}}^{N} \frac{q_i q_j}{z - z_j}$$

in the point $z = z_i$. We see that the complexity of this problem is $\mathcal{O}_1(N^2)$, because we evaluate $N$ functions and each function has $N$ summations. The next idea is to approximate these functions with polynomials, e.g. Taylor polynomials with expansion point $z_L$, i.e.

$${}^{i}T : \mathbb{C} \to \mathbb{C} : z \mapsto \sum_{k=0}^{q} a_k (z - z_{L_i})^k.$$

This yields to evaluating $N$ polynomials at 1 point each. The complexity will be $\mathcal{O}_1(N \cdot q)$, because we evaluate $N$ functions and each function has $q$ summations. Our hope is

- the approximation error is small,
- $q \ll N$, and
- the computation of the coefficients is subquadratic in $N$.

If we can accomplish this, the resulting algorithm will have the desired subquadratic runtime. Lemma 26 shows that there exists a small radius of convergence $R$ and a constant $c$, such that

$$\Big|\sum_{\substack{j=1\\ j\neq i}}^{N} \frac{q_i q_j}{z_i - z_j} - \sum_{k=0}^{q} a_k (z - z_{L_i})^k\Big| \le c \cdot 2^{-q}.$$

Therefore the approximation error is small and the truncation coefficient $q$ can be chosen linear in the error exponent $\mathbb{E}$, such that the overall error is smaller than $2^{-\mathbb{E}}$. Hence the first 2 points are fulfilled. Now we will look at the third point, how to compute the coefficients. Referring to lemma 26 again, we see that the coefficients are given by

$$a_k = \sum_{i=0}^{N} \frac{q_i}{(z_L - z_i)^{k+1}},$$

therefore the direct computation of these coefficients would need $N$ operations for a single coefficient. Since we compute $N \cdot q$ coefficients, this will be quadratic in $N$. Therefore the third point is *not* fulfilled and we need another approach. Nevertheless we see, that Taylor series are a good approximation and could be handy later.

## 3.2 Ansatz: Laurent Series

Consider the following observation

$$\frac{a}{z-z_a} + \frac{b}{z-z_b} = \frac{a}{(z - \frac{z_a+z_b}{2}) + \frac{z_b-z_a}{2}} + \frac{b}{(z - \frac{z_a+z_b}{2}) + \frac{z_a-z_b}{2}}$$

For $|z - \frac{z_a+z_b}{2}| \gg |\frac{z_b-z_a}{2}|$, this yields to

$$\approx \frac{a+b}{z - \frac{z_a+z_b}{2}}.$$

This means, that if $z$ is far away to the mean of $z_a$ and $z_b$ compared to their distance, then $\frac{a+b}{z-\frac{z_a+z_b}{2}}$ is a good approximation to $\frac{a}{z-z_a} + \frac{b}{z-z_b}$. By generalizing this observation we will approximate Trummer's Problem with Laurent series instead of Taylor series.

$${}^{i}l : \mathbb{C} \to \mathbb{C} : z \mapsto \sum_{k=0}^{p} \frac{b_k}{(z - z_{L_i})^k}.$$

This yields to evaluating $N$ polynomials (in $\frac{1}{z}$) at 1 point each. The complexity will be $\mathcal{O}_1(N \cdot p)$, because we evaluate $N$ functions and each function has $p$ summations. We got a similar hope as in section 3.1, namely

- the approximation error is small,
- $p \ll N$, and
- the computation of the coefficients is subquadratic in $N$.

If we can accomplish this, the resulting algorithm will have the desired subquadratic runtime.
In the next example we'll show that a simple approximation with a single Laurent series is *not* sufficient.

**Example 10 (The Convergence of Laurent Series).** Let $N = 4, q = (1,1,1,1), z = (-1,0,1,2)$. Then ${}^{2}\Phi : z \mapsto \frac{1}{z+1} + \frac{1}{z-1} + \frac{1}{z-2}$ has singularities at $z_1 = -1, z_3 = 1$ and $z_4 = 2$. *Each* circle containing all three singularities also contains $z_2 = 0$ (a circle is convex and $z_2$ lies on the line between $z_1, z_3$). Therefore expanding ${}^{2}\Phi$ into a Laurent series would yield to a divergent function in $z = 0$.
But, expanding $z \mapsto \frac{1}{z+1}$ and $z \mapsto \frac{1}{z-1} + \frac{1}{z-2}$ into Laurent series, evaluating them at $z = 0$ and adding these values would be possible. This shows that we cannot expand the whole expression into a Laurent series, but we can expand it into a sum of Laurent series.
In the general case we want to decompose $Z = \{z_1, \dots, z_N\}$ into *clusters* $U \subset Z$, such that

- each cluster $U$ contains one or several points $z_i$ (the clusters won't be distinct),
- there exist spheres $\mathfrak{S}_1, \mathfrak{S}_2, \ldots$, such that $\mathfrak{S}_j$ covers all points in $U_j$,
- the union of all clusters is $Z$, and
- for each $z_i \in Z$ there exist some clusters $U_1(z_i), \ldots, U_k(z_i)$, which form a partition of $Z \setminus \{z_i\}$.

Using this decomposition, the function

$$z \mapsto q_i \sum_{l=1}^{k} \underbrace{\sum_{j \in U_l} \frac{q_j}{z - z_j}}_{\text{expand to Laurent series } {}_{U_l}\mathfrak{L}(z)} = q_i \sum_{l=1}^{k} {}_{U_l}\mathfrak{L}(z)$$

is convergent in $z = z_i$ and the sum runs over all particles exactly once, due to the fourth property of the decomposition.

Nevertheless this examples motivates the next approach, a sum of Laurent series.

## 3.3 Ansatz: Sum of Laurent Series

Approximation of Trummer's Problem into a sum of Laurent series yields to

$${}^{i}L : \mathbb{C} \to \mathbb{C} : z \mapsto \sum_{m=0}^{M} \sum_{k=0}^{p} \frac{b_k}{(z - z_{m_i})^k}.$$

Hence we evaluate $N$ functions, which are the addition of $M$ polynomials (in $\frac{1}{z}$) at 1 point each. The complexity will be $\mathcal{O}_1(N \cdot p \cdot M)$. We got a similar hope as in section 3.2

- the approximation error is small,
- $p \ll N$ and $M \ll N$, and
- the computation of the coefficients is subquadratic in $N$.

If we can accomplish this, the resulting algorithm will have the desired subquadratic runtime.
Now the important question arises

*How to choose the partitioning of the Laurent series?*

We will describe the idea behind the construction in the next section using a more geometric approach.

## 3.4 Main Idea of the Fast Multipole Method

In this section we will omit the technical aspects and only give the idea of the algorithm which will be analysed in detail later.

- STEP 1: Let $N = 8$ and choose some $q, z \in \mathbb{C}^8$. The first step is to reinterpret the 8 values of $z$ as 8 different points in the complex plane with the corresponding values of $q$, cf. fig. 3.1. Now, each point represents a function $\Phi : \mathbb{C} \to \mathbb{C} : z \mapsto \frac{q_i}{z - z_i}$, which is defined in the complete complex plane. From a physical point of view, these 8 points in the complex plane with the value $q_i$, can be seen as particles with a charge $q_i$. The above function $\Phi$ is the Coulomb field of a charged particle in two dimensions up to a prefactor. Using this analogue one sees that Trummer's Problem is equivalent to the N-body problem, where one has to compute the induced forces by evaluating the Coulomb field for all particles.

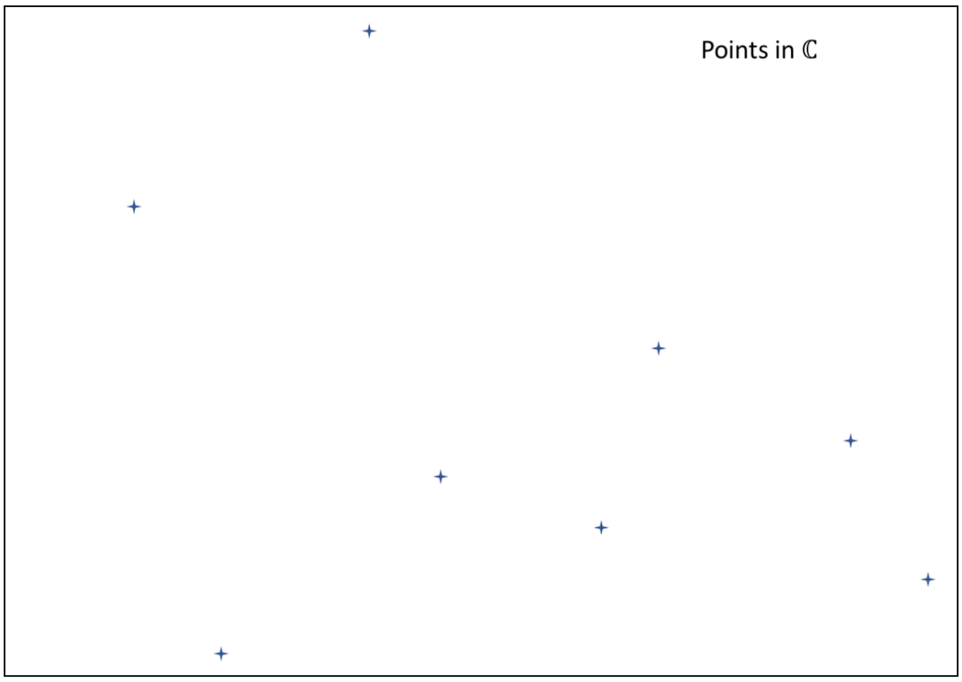


Figure 3.1.: STEP 1: $N = 8$ different particles in the complex plane at position $z_i$ with their corresponding charges $q_i$, for $i = 1 \dots 8$. Each particle induces the Coulomb field $\Phi_i : z \mapsto c\frac{q_i}{z - z_i}$ in two dimensions.

- STEP 2: In the next step, we search relatively near pairs of points, cf. fig. 3.2. Each circle, contains 2 neighbors. We say these neighbors form a cluster of points, which is represented by the circle. Now we combine the functions of two neighbored points and expand this combination into a Laurent series in the center of the corresponding circle. By doing this for all neighbored pairs, each circle gets a corresponding Laurent series with the center of the circle as the expansion point.
- STEP 3: After each neighbored pair of points forms a cluster with corresponding Laurent series, the next step is to merge neighbored cluster and combine the corresponding Laurent series again, cf. fig. 3.3. This step should be iterated for the new generated cluster until there is exactly one cluster containing all points. We see that the Laurent series diverge in the interior of the corresponding circle and converge in the exterior.
- STEP 4: Now we fix one point and choose the left most and call it $z_1$. Our ansatz says we should choose circles, which do not contain $z_1$ and evaluate the corresponding Laurent series at $z_1$, cf. fig. 3.4. Since these 3 Laurent series converge at $z_1$, the sum converges as well and the operation is safe. There exist counterexamples, cf. section 4.2.3, where the number of Laurent series $M$, we have to consider, is linear in $N$, therefore overall complexity would be quadratic in $N$ again. Hence the direct evaluation of the Laurent series should be avoided. As seen in section 3.1 Taylor series could be suitable, if we are able to compute the coefficients in subquadratic runtime.
- STEP 5: The next step is to transform the 3 Laurent series from above into a Taylor series of the two point cluster in the upper left corner (this cluster contains $z_1$), cf. fig. 3.5. Later on we will show that this can be done sufficiently fast in order to get a subquadratic runtime. The next step which leaps into ones eye is to evaluate the constructed Taylor series at $z_1$, but $z_1$ is on the boundary of the circle of convergence of the Taylor series which will result in problems when evaluating at $z_1$. Therefore direct evaluation of the Taylor series is not applicable. One can show that direct evaluation of the Taylor series would indeed result in a quadratic runtime.

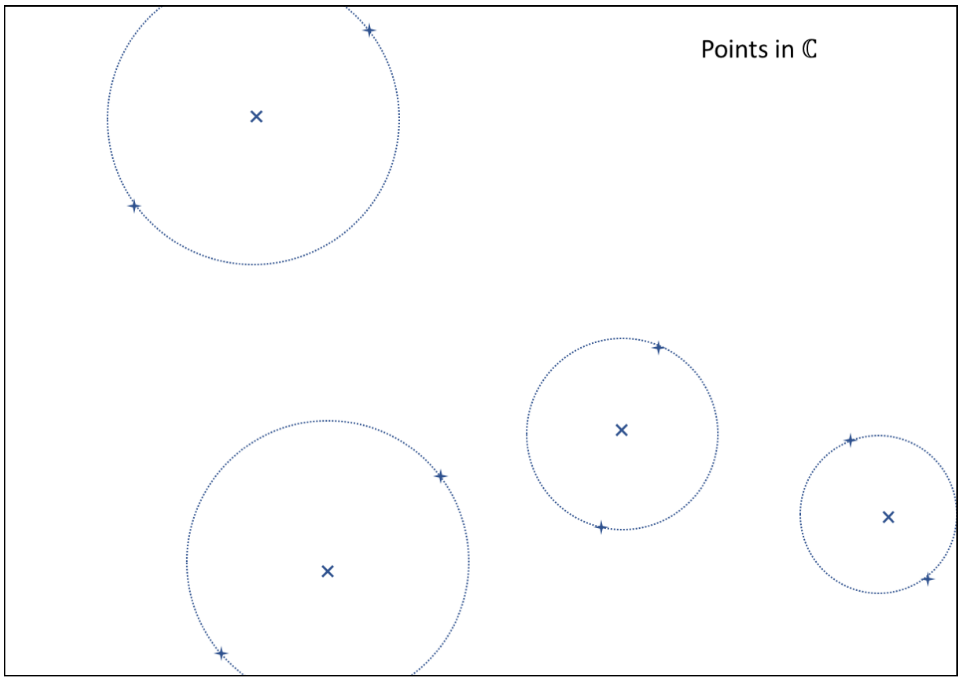


Figure 3.2.: STEP 2: Merging neighbored particles to a cluster, which is represented by a circle. Additionally calculating the corresponding Laurent series for each circle $C$ with the center point $z_C$ being the expansion point. Each cluster induces the Coulomb field $\Phi_C : z \mapsto \sum_{k=0}^{\infty} \frac{b_k}{(z-z_C)^k}$.

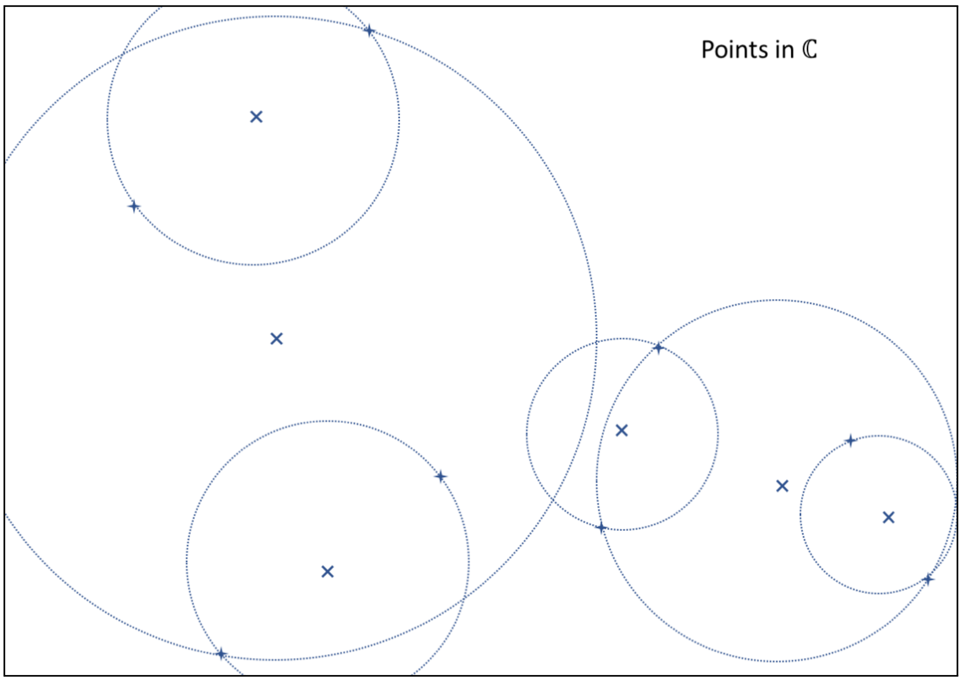


Figure 3.3.: STEP 3: Iteratively building up the hierarchy of clusters by merging two clusters to a new cluster in each iteration. Additionally calculating the corresponding Laurent series for each circle $C$ with the center point $z_C$ being the expansion point. Each cluster induces the Coulomb field $\Phi_C : z \mapsto \sum_{k=0}^{\infty} \frac{b_k}{(z-z_C)^k}$.

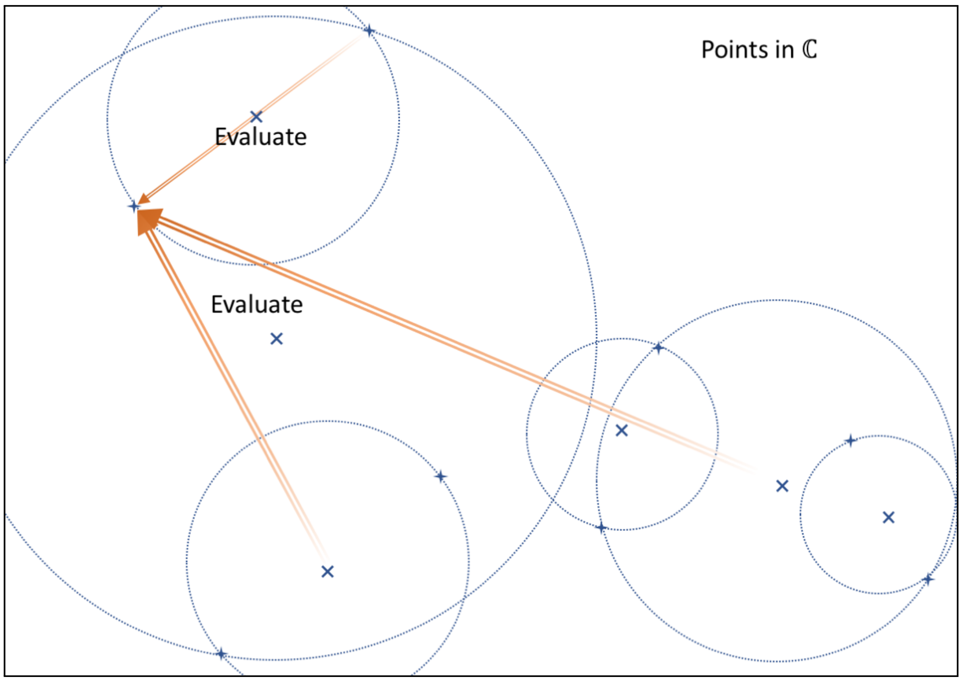


Figure 3.4.: STEP 4: Choose the left most particle at point $z_1$ and evaluate the $3$ Laurent series, which clusters form a partition of all other particles, at $z_1$. These Laurent series converge at $z_1$ and the truncation error would be small. But direct evaluation would yield to a quadratic algorithm.

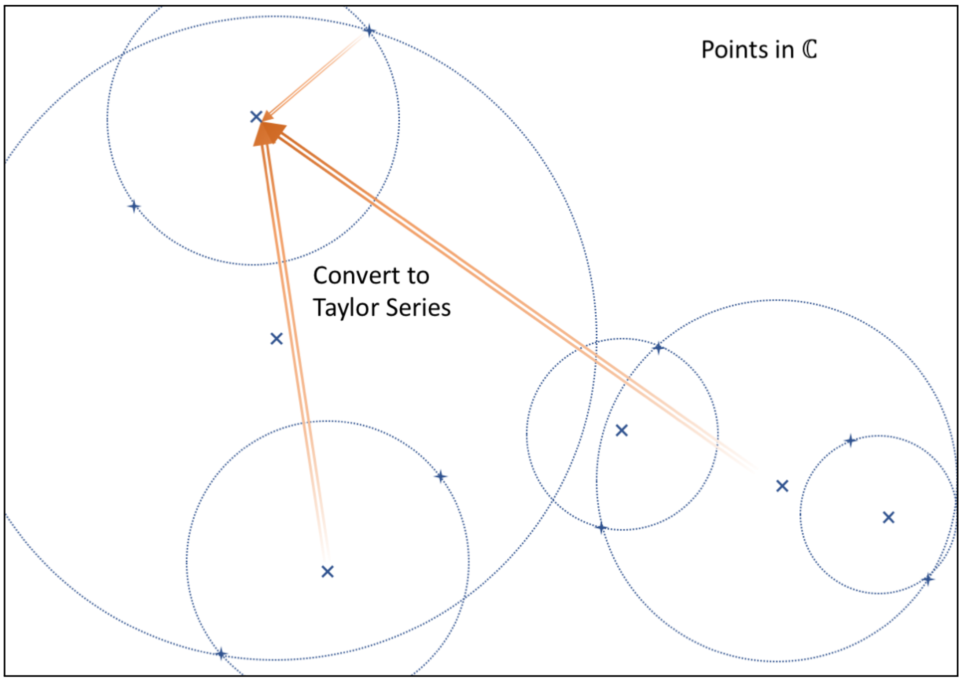


Figure 3.5.: STEP 5: Transform the before mentioned $3$ Laurent series $\Phi_C : \mathbb{C} \to \mathbb{C} : z \mapsto \sum_{k=0}^{\infty} \frac{b_k}{(z-z_C)^k}$ into a Taylor series ${}^{C_0}T : \mathbb{C} \to \mathbb{C} : z \mapsto \sum_{k=0}^{\infty} a_k(z - z_{C_0})^k$ of the two point cluster $C_0$ in the upper left corner (this cluster contains $z_1$).

- STEP 6: The next step is to "move" the Taylor series to $z_1$, i.e. changing the expansion point from the center of one circle to the center of another circle (the circle at $z_1$ with radius 0), cf. fig. 3.6. Now the evaluation of this Taylor series at $z_1$ is trivial, because all non constant terms vanish. $\sum_{k=0}^{q} a_k(z_i - z_i)^k) = \sum_{k=0}^{q} a_k 0^k) = a_0$.

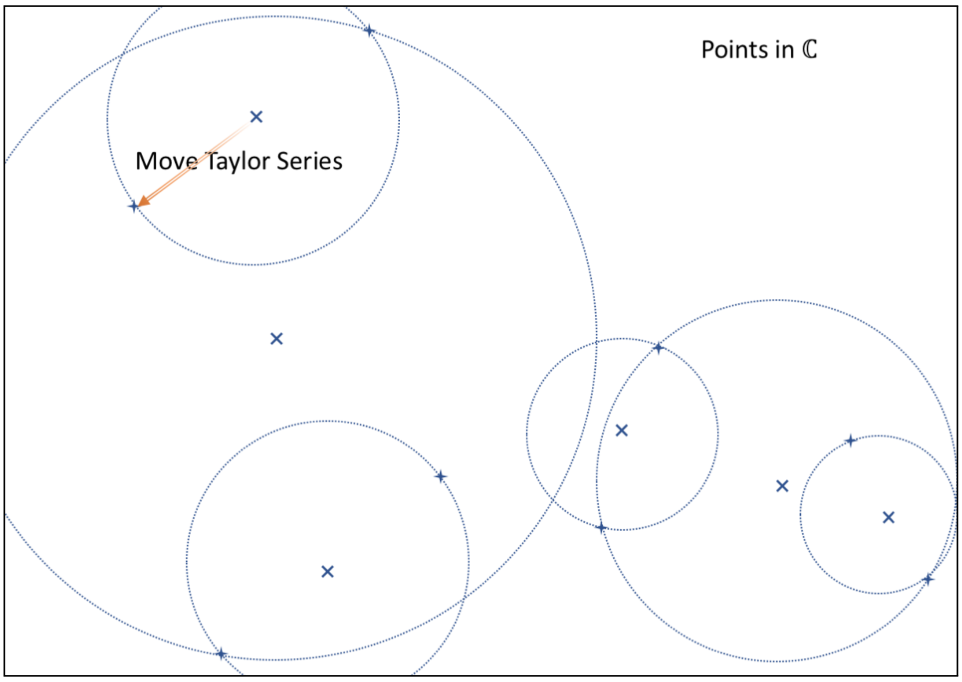


Figure 3.6.: STEP 6: Moving the Taylor series ${}^{C_0}T : \mathbb{C} \to \mathbb{C} : z \mapsto \sum_{k=0}^{\infty} a_k(z - z_{C_0})^k$ of the two point cluster $C_0$ in the upper left corner to a Taylor series ${}^{1}T : \mathbb{C} \to \mathbb{C} : z \mapsto \sum_{k=0}^{\infty} a_k(z - z_1)^k$ at $z_1$, i.e. changing the expansion point from $z_{C_0}$ to $z_1$. The evaluation at $z_1$ is trivial, because all non constant terms vanish.

When carefully reading the above six steps, one notices, that STEP 2 and STEP 3 induce a partial ordering of the clusters. We say a cluster is smaller than another cluster, if the all of the represented points of a cluster are contained in the other cluster. When merging two cluster, we see that the merged cluster is larger than both clusters, w.r.t the partial ordering. This defines a binary tree, where the last cluster from iterating step 3, is the root and represents all points. The leaves of this tree represent a single point each. Now we can formalize the procedure from above in algorithm algorithm 1. Now we have shown the main idea of the Fast Multipole Method. In the next sections of this thesis we will answer the open questions.

COMPUTE CLUSTER HIERARCHY:

- Which particles and clusters should be merged?
- Is this approach possible for arbitrary distributions of particles?
- What is the runtime?

To answer these questions we will introduce the Fair Split Tree in section 4.1 and show we can construct this hierarchy of clusters for arbitrary distributions of particles in subquadratic runtime. This Fair Split Tree got all desired properties we will need later on.

INITIALIZE:

- How to initialize the Laurent series for each particle?
- How large is the approximation error when the input of the positions and charges of the particles gets approximated?
- What is the runtime?

The construction of the Laurent series is the aim in section 4.3.1. The approximation of the input with the corresponding error will be addressed in section 5.4. The subquadratic runtime will be a shown in section 4.4.

UPCAST:

- How to merge two Laurent series into a new one?
- How large is the approximation error by using truncated Laurent series up to order $q$?
- What is the runtime?

The method to merge two Laurent series into a new one is described in section 4.3.2. The approximation error by using truncated Laurent series up to order $q$ will be addressed in section 5.1. The subquadratic runtime will be stated in section 4.4.

CONVERSION:

- What is $\mathscr{W}$?

  Which pairs of clusters do we need?

  How many of these pairs do we have to consider?

  Does $\mathscr{W}$ always exist for arbitrary distributions of particles?

  How to compute $\mathscr{W}$ and what is the runtime?
- How to convert one Laurent series into a Taylor series?
- How large is the approximation error by using truncated Taylor series up to order $q$?
- What is the runtime?

To answer the questions about $\mathscr{W}$ we will introduce the Well-Separated Pair Decomposition in section 4.1.
The method to convert one Laurent series into a Taylor series is described in section 4.3.3. The approximation error by using truncated Taylor series up to order $p$ will be addressed in section 5.1. The subquadratic runtime will be stated in section 4.4.

DOWNCAST:

- How to change the expansion point of a Taylor series?
- How to merge multiple Taylor series?
- What is the runtime?

The method to change the expansion point of a Taylor series and to merge multiple Taylor series is described in section 4.3.4. The subquadratic runtime will be stated in section 4.4.

EVALUATION:

- How to evaluate the truncated Taylor series at a given point?
- What is the runtime?

As seen before, the questions about EVALUATION are easy to answer: The evaluation of a Taylor series $\sum_{k=0}^{\infty} a_k(z-z_0)^k)$ at the expansion point $z_0$ is trivial, because all non constant terms vanish. $\sum_{k=0}^{\infty} a_k(z_0-z_0)^k = \sum_{k=0}^{\infty} a_k 0^k = a_0$. Therefore the evaluation only has to output $a_0$ which is linear in the encoding length of $a_0$.
Let $N$ denote the number of particles and let $\mathbb{E}$ be the error exponent, such that the result is approximated up to an error $2^{-\mathbb{E}}$. In section 4.4 we will introduce an algorithm with runtime $\mathscr{O}_2(N \cdot \mathbb{E}^4)$. By further enhancements of the computation process we are able to save one factor of $\mathbb{E}$, which leads to an algorithm with runtime $\mathscr{O}_2(N \cdot \mathbb{E}^3)$, cf. section 4.5.5. By introducing a new Well-Separated Pair Decomposition with the corresponding Fair Split Tree adding new main steps to the algorithm and using results from last year by Tsigaridas and Pan [24] we are able to save an additional factor of $\mathbb{E}$, which leads to an algorithm with

runtime $\mathcal{O}_2(N \cdot \mathbb{E}^2)$, cf. section 5.7. In section 2.2 we have shown the lower bound of Trummer's Problem is $\mathcal{O}_2(N \cdot \mathbb{E})$, therefore our developed algorithm is nearly optimal.

**Algorithm 1:** A Tree Algorithm

```
Input  : $N$ particles with charges $q_i$ and positions $z_i$ for $i = 1, \dots, N$.
Output : $(C(q,z) \cdot q)_i = \sum_{\substack{j=1 \\ j \neq i}}^{N} \frac{q_i q_j}{z_i - z_j} = {}^i\Phi(z_i)$, for all $i = 1, \dots, N$.
// COMPUTE CLUSTER HIERARCHY, cf. STEP 2 and STEP 3
Compute the hierarchy of the clusters for the particles. This forms a tree which will be denoted by $\mathscr{T}$.
// INITIALIZE, cf. STEP 1
foreach leaf $U \in \mathscr{T}$ do // i.e. each particle
    Initialize the Laurent series ${}_U\mathfrak{L}(z)$ for each particle
end
// UPCAST, cf. STEP 2 and STEP 3
for $l := depth(\mathscr{T})$ to 1 do // Iterate from the leaves to the root.
    foreach inner node $U \in \mathscr{T}$ with depth $l$ do
        Compute the Laurent series ${}_U\mathfrak{L}(z)$ by combining the Laurent series ${}_V\mathfrak{L}(z), {}_W\mathfrak{L}(z)$ of the children $V, W$.
    end
end
// CONVERSION, cf. STEP 5
for all appropriate pairs of clusters $U, V \in \mathscr{T}$. We will call the set of all appropriate pairs $\mathscr{W}$. do
    Convert the Laurent series ${}_V\mathfrak{L}(z)$ of the cluster $V$ to the Taylor series ${}_U\mathfrak{T}(z)$ of the cluster $U$
end
// DOWNCAST, cf. STEP 6
for $l := 1$ to $depth(\mathscr{T})$ do // Iterate from the root to the leaves.
    foreach node $U \in \mathscr{T}$ with depth $l$ do
        Compute the new Taylor series ${}_U\mathfrak{T}(z)$ by combining the Taylor polynomial ${}_V\mathfrak{L}(z)$ of the parent $V$ with the old Taylor
        polynomial ${}_U\mathfrak{T}(z)$.
    end
end
// EVALUATION, cf. STEP 6
foreach leaf $U \in \mathscr{T}$ do // i.e. each particle
    Evaluate the Taylor polynomial ${}_U\mathfrak{T}(z)$ at $z = z_U$., i.e. return ${}_U\mathrm{T}_0$;
end
```

Chapter 4

# Bit-Cost Analysis of the Fast Multipole Method

In section 3.4 we have already stated the main ideas of the *Fast Multipole Method* (FMM) and formalize them within this chapter. Therefore we will develop the FMM step by step, starting by a method with quadratic runtime in $N$ and improving this method continually. By highlighting possible pitfalls and lacks of efficiency, we will see that one cannot simplify or omit the main steps of this version of the FMM. The promised subquadratic runtime will be proven later; we will concentrate on developing the algorithm at first. Since we want to use truncated Laurent and Taylor series, we have to be able to control the truncation error. Later on we will see that if the clusters are relatively far away, in comparison to the radius of the enclosing sphere, the error can be exponentially suppressed in terms of the truncation order. This *relatively far away* will be defined in section 4.1 by *s-well separated pairs*. In the same section we define the *Well-Separated Pair Decomposition* (WSPD) which is needed throughout the complete thesis. By using the properties of the WSPD we are able to proof the runtime of our algorithm, reduce the truncation errors and also proof some possible pitfalls.

## 4.1 Clustering of Points — The Well-Separated Pair Decomposition

In this section we define the *Well-Separated Pair Decomposition* and state the most important properties.

**Definition 11 (Well-Separated Pair).** This definition is based on [23]. Let $U, V \subset \mathbb{R}^d$ be two finite sets of points. The bounding box $R(U)$ is the smallest axes-parallel $d$ dimensional hyper-rectangle that contains $U$.
Let $\mathbb{R} \ni s \geq 0$. The two sets $U, V$ are called *$s$-well-separated* iff

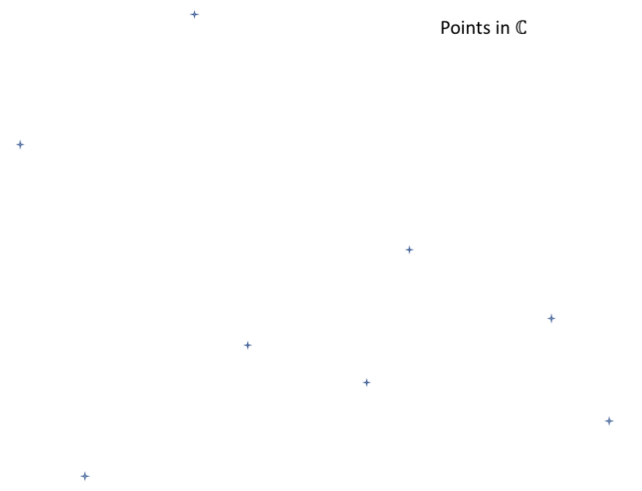
- there exist two $d$-dimensional spheres $\mathfrak{S}_U, \mathfrak{S}_V$ with the same radius $R$,
- $\mathfrak{S}_U$ contains the bounding box $R(U)$ of $U$,
- $\mathfrak{S}_V$ contains the bounding box $R(V)$ of $V$, and
- the distance between $\mathfrak{S}_U, \mathfrak{S}_V$ is at least $sR$.

The real number $s$ is called the *separation ratio.*

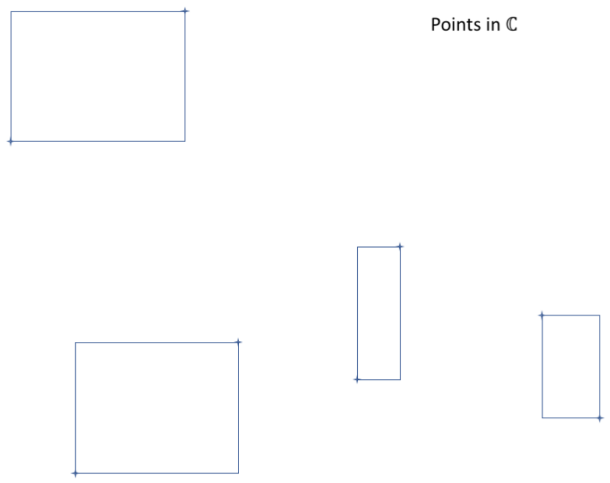
**Example 12 (Well-Separated Pair).** In this example we want to show what *$s$-well-separated* means geometrically in fig. 4.1. Therefore we use always the same points, cf. fig. 4.1(a). Figure 4.1(b) shows the minimal bounding boxes $R(\cdot)$ for two particles each and fig. 4.1(c) shows the corresponding minimal spheres containing these bounding boxes. These spheres got different radii, but in order to say that two sets are well-separated we need spheres with the same radii. In fig. 4.1(d) we want to see if the two pairs of points in the middle are well-separated, wherefore we have enlarged the smaller sphere, such that both radii of the spheres are equal. In the next step we move this sphere to the right in order to maximize the distance, without moving the points in the exterior. Now we see that both sets are 0.8-*well-separated.* In fig. 4.1(e) we consider the two points in the upper left corner and one of the previous pairs and do a similar construction as in fig. 4.1(d). Then we see that both sets are 2-*well-separated.*

**Definition 13 (Well-Separated Pair Decomposition).** This definition is based on [23]. Let $Z \subset \mathbb{R}^d$ be a set of $N$ points. Let $\mathbb{R} \ni s \geq 0$ be the *separation ratio.*
A *$s$-Well-Separated Pair Decomposition* (WSPD) for $Z$ is a list

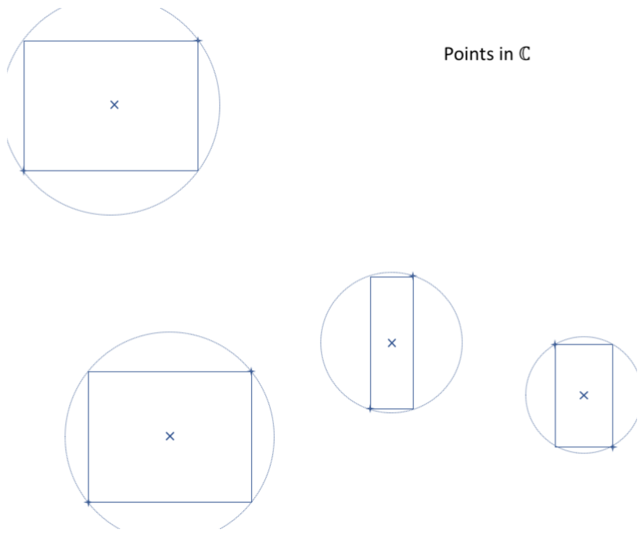
$$\big[(U_1, V_1), (U_2, V_2), \dots, (U_l, V_l)\big]$$



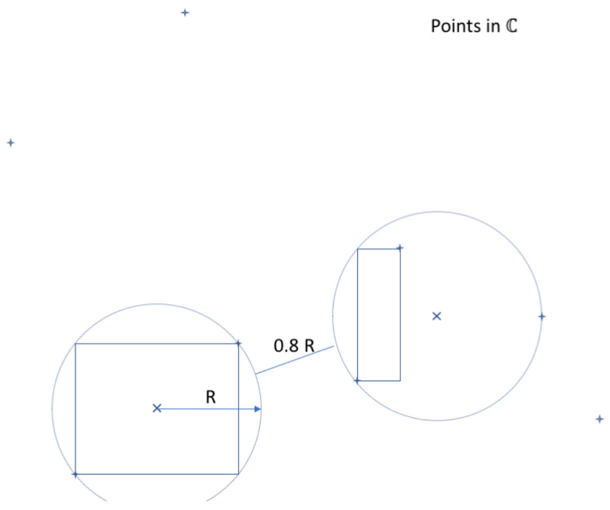
(a) $8$ points in the complex plane used as a basis to show the $s$-*well-separated* geometrically.



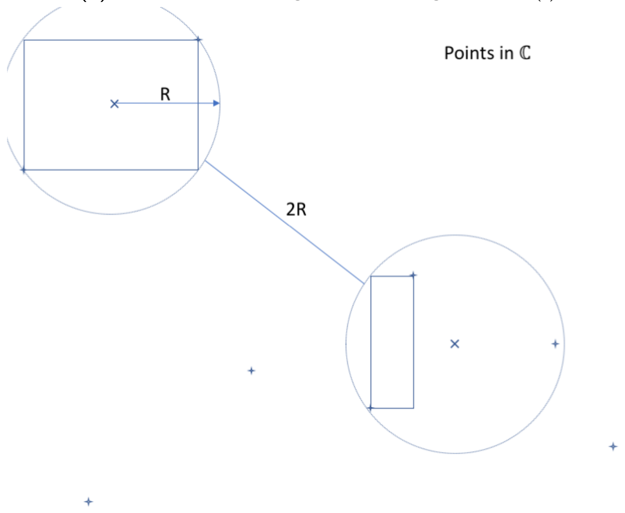
(b) Bounding boxes $R(\cdot)$ for two particles each.



(c) Spheres containing the bounding boxes $R(\cdot)$.



(d) Two pairs of points which are $0.8$-*well-separated*.



(e) Two pairs of points which are $2$-*well-separated*.

Figure 4.1.: Examples for well-separated points.

of pairs of non-empty subsets of $Z$, such that

- for each $i = 1, \dots, l$, the sets $U_i$ and $V_i$ are $s$-well-separated, and
- for any two distinct points $x, y \in Z$, there exists exactly one index $i$ with $1 \leq i \leq l$, such that $x \in U, y \in V$ or vice versa.

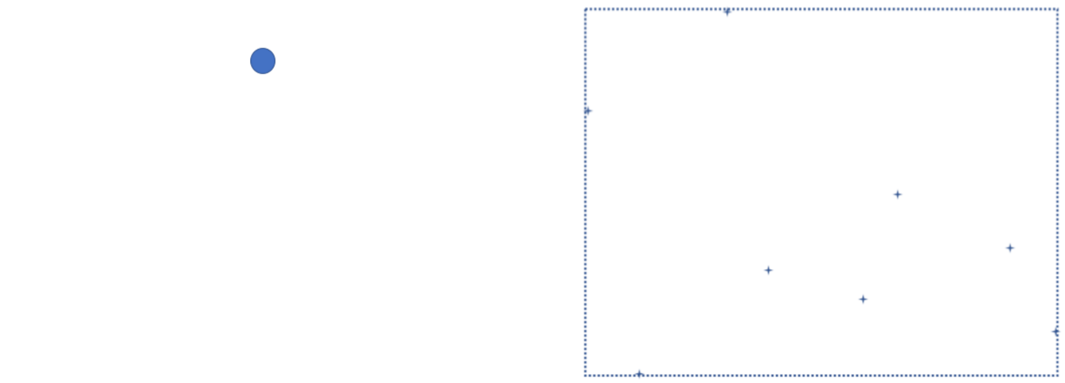
The integer $l$ is called the *size* of the WSPD.

**Lemma 14 (Properties of the Well-Separated Pair Decomposition).** *Let $Z \subset \mathbb{R}^d$ be a set of $N$ points. Let $\mathbb{R} \ni s \geq 0$ be the separation ratio.*

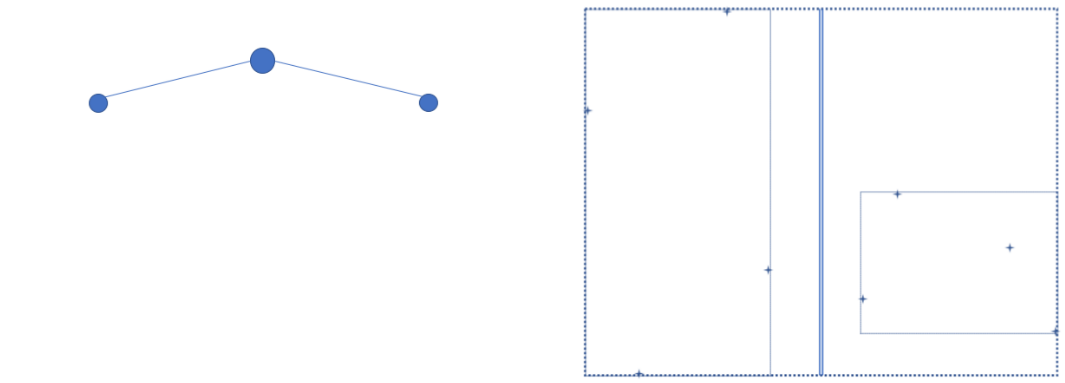
- *There exists a WSPD $\mathscr{W}$ of size $\mathscr{O}(s^d N)$.*
- *This WSPD can be computed in $\mathscr{O}(N \log N)$.*
- $\sum_{(U,V) \in \mathscr{W}} |U| + |V| \in \Theta(N^2)$.
- *The clusters $U$ in the WSPD $\mathscr{W}$, i.e. $(U, *) \in \mathscr{W}$, or $(*, U) \in \mathscr{W}$ form a binary tree $\mathscr{T}$ by adding $Z$ as the root node. This tree is called the Fair Split Tree.*
  - *Let $U \in \mathscr{W}$. If $|U| \geq 2$ then $U$ has exactly two children $V, W$, which are a partition of $U$.*
  - *For each two nodes $V, W$ in the tree, either the points of $V$ are included in $U$ (or vice versa) or they are completely distinct, i.e. do not share any points.*
  - *By iterating from a leaf, which represents exactly one point, to the root only more points get added and no points get subtracted.*
- *The size $|\mathscr{T}|$ of the split tree $\mathscr{T}$, i.e. the number of nodes, is given by $|\mathscr{T}| \in \Theta(N)$ nodes.*
- *The depth depth$(\mathscr{T})$ of the split tree $\mathscr{T}$ is given by depth$(\mathscr{T}) \in \Theta(N)$ in worst case.*

*Proof.* These properties were proven by Callahan in his PhD thesis [5]. □

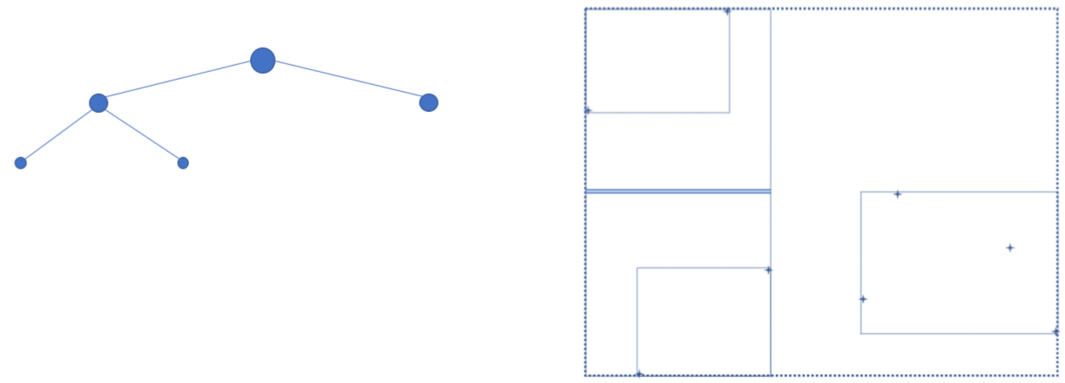
**Example 15 (Fair Split Tree).** In this example we want to show how the *Fair Split Tree* gets constructed and the correspondence between the tree, the clusters and the points with the help of figs. 4.2 and 4.3. We use always the same points, cf. fig. 4.1(a). On the left of each figure there is the *Fair Split Tree* and on the right the corresponding clusters or bounding boxes of points. Figure 4.2(a) shows all points with the bounding box of all together. This forms the root of our split tree. By splitting this bounding box in the middle, cf. the two lines in the middle representing this split in fig. 4.2(b), we get two new boxes. Each box contains 4 points and represents a node in the tree. By splitting the left box in fig. 4.2(c) again we get two new bounding boxes with their corresponding nodes. In fig. 4.3(a) we see that both small boxes on the left get split into single points. These points get added to the tree and form its leaves. Figure 4.3(b) is similar to fig. 4.2(b), but we have split the right box now. And finally fig. 4.3(c) finishes the tree by splitting the last two boxes on the right into single points, similar to fig. 4.3(a), and appending the corresponding leaves to the tree.



(a) The root of the Fair Split Tree (on the left) represents all points with their bounding box (on the right).

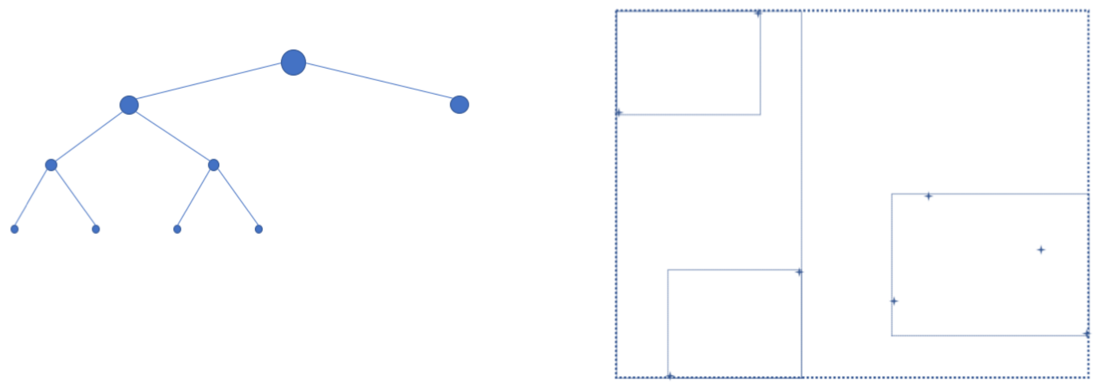


(b) By splitting (represented by the two lines in the middle) the bounding box of all points into two parts, we get two new clusters of points with their bounding box on each side. Both clusters create a new node in the tree each.

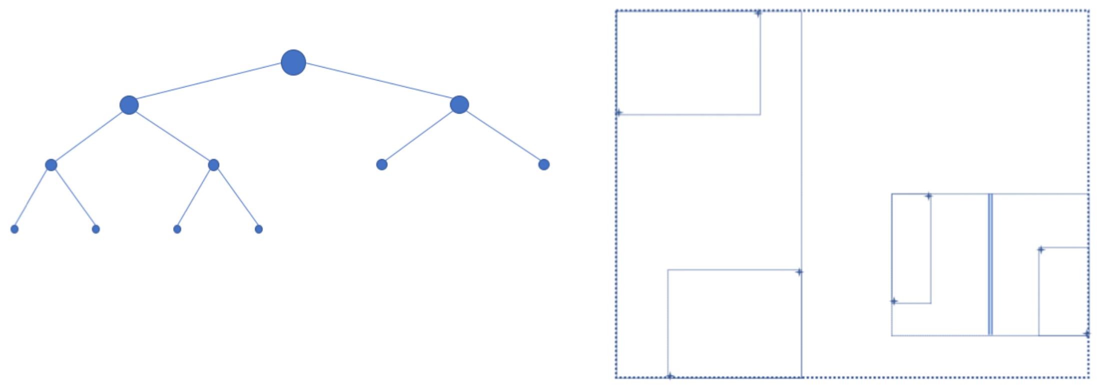


(c) Splitting the left most bounding box creates two new clusters and nodes in the tree.

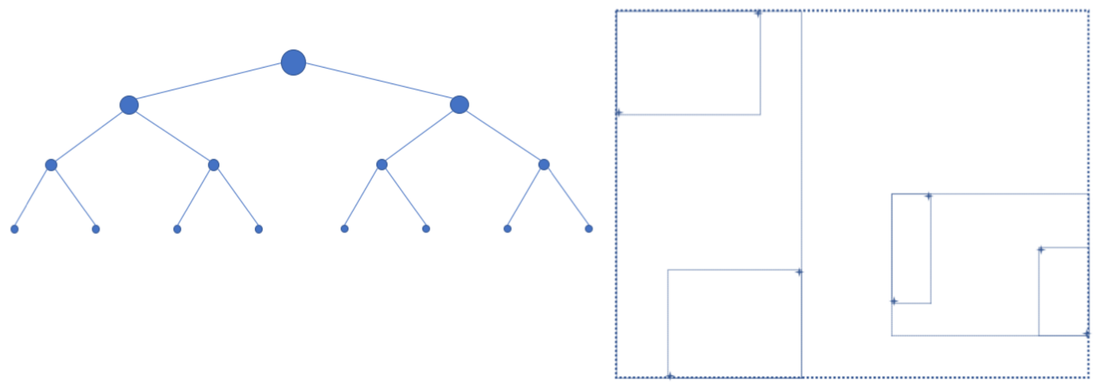
Figure 4.2.: Example of a Fair Split Tree — part one, cf. fig. 4.2 for the second part. On the left the tree is presented and on the right the corresponding cluster of points with their bounding boxes.



(a) Both small boxes on the left get split into single points. These points get added to the tree and form its leaves.



(b) Splitting the right most bounding box creates two new clusters and nodes in the tree.



(c) Both small boxes on the right get split into single points. These points get added to the tree and form its leaves.

Figure 4.3.: Example of a Fair Split Tree — part two, cf. fig. 4.3 for the first part. On the left the tree is presented and on the right the corresponding cluster of points with their bounding boxes.

## 4.2 Pitfalls, Lacks of Efficiency and their Solutions

In this section we will present some pitfalls and lacks of efficiency which could occur during the FMM and develop an algorithm which bypasses each of these problems, afterwards. Without knowing these problems the algorithm could seem to be a pedestrian method.
In section 4.2.1 we state a very easy algorithm using the idea of Laurent series combined with the WSPD to solve Trummer's Problem. In section 4.2.2 we proof that it takes too much time to construct all Laurent series using the properties of the WSPD. By using these properties again, we show in section 4.2.3 that simple evaluation of the Laurent series is too slow as well. Therefore our later algorithm has to bypass these problems.

### 4.2.1 A First Algorithm

With the notation of a Well-Separated Pair Decomposition we can now formalize the idea of building clusters, computing the Laurent series and evaluating them in algorithm 2. The runtime of this naive algorithm will be analyzed in sections 4.2.2 and 4.2.3. Afterwards, we will give improved versions of the construction and evaluation step.

**Algorithm 2:** A First Algorithm for the Fast Multipole Method.

```
1 Compute the WSPD $\mathscr{W}$ with the corresponding split tree $\mathscr{T}$;
2 Compute the Laurent series of the function $z \mapsto \underbrace{\sum_{j \in U} \frac{q_j}{z - z_j}}_{\text{expand to Laurent series } {}_U\mathfrak{L}(z)} = {}_U\mathfrak{L}(z)$ for each cluster $U \in \mathscr{T}$;
3 Evaluate the function ${}^i\Phi(z)$, by using the previously computed Laurent series, at the point $z = z_i$, i.e. compute
  ${}^i\Phi(z_i) = \sum_{\substack{(U,V) \in \mathscr{W} \\ z_i \in U}} {}_V\mathfrak{L}(z_i) + \sum_{\substack{(U,V) \in \mathscr{W} \\ z_i \in V}} {}_U\mathfrak{L}(z_i)$ for each $z_i \in Z$;
```

### 4.2.2 Construction of the Laurent Series

**Lemma 16.** *In algorithm 2, the construction of the Laurent series (line 2) is in $\Omega(N^2)$, in worst case.*

*Proof.* The construction of the Laurent series ${}_U\mathfrak{L}(z)$ of $U$, which represents $|U|$ points, takes at least $|U|$ steps, otherwise we could exchange one point and the computed Laurent series would not be changed, which cannot be possible.
Then the number of steps in line 2 are at least $\sum_{U \in \mathscr{T}} |U|$. By choosing $Z = (2^{-1}, 2^{-2}, 2^{-3}, \ldots, 2^{-N})$, the split tree $\mathscr{T}$ will have depth $N$ and on each level $n$ there are exactly two nodes, one leaf with size 1 and a node $U$ with size $N - n$. Then we get

$$\sum_{U \in \mathscr{T}} |U| = \sum_{n=1}^{N} (1 + N - n) = \sum_{n=1}^{N} n \in \Omega(N^2). \qquad \square$$

In order to obtain a subquadratic algorithm we improve the construction of the Laurent series, by exploiting the structure of the split tree $\mathscr{T}$. The construction of all Laurent series will be a bottom-up approach. We start by computing all Laurent series for the leaves. Afterwards we will compute the Laurent series of an inner node by combining the two Laurent series of its two children. In fact, this approach tuns out to be subquadratic in $N$, which will be shown later.

### 4.2.3 Evaluation of the Laurent Series

**Lemma 17.** *In algorithm 2, the evaluation of all Laurent series (line 3) is in $\Theta(N^2)$, in worst case.*

*Proof.* Line 3 in algorithm 2 is equivalent to algorithm 3, which needs at least $\sum_{(U,V)\in\mathscr{W}}|U| + |V|$ number of steps. Using the properties of the Well-Separated Pair Decomposition, cf. lemma 14, we get

$$\sum_{(U,V)\in\mathscr{W}} |U| + |V| \in \Theta(N^2),$$

which finishes the proof. □

```
1 foreach $(U,V) \in \mathscr{W}$ do
2     Evaluate ${}_V\mathfrak{L}(z)$ for each $z \in U$;
3     Evaluate ${}_U\mathfrak{L}(z)$ for each $z \in V$;
4 end
5 Combine all previous values appropriately, i.e. add all values which belong to $z_i$ for each $i = 1, \dots, N$;
```
**Algorithm 3:** Evaluation of all Laurent series

In order to obtain a subquadratic algorithm we improve the evaluation of the Laurent series. The idea of the fast evaluation is a top-down approach. For each pair $(U,V) \in \mathscr{W}$ we convert the Laurent series ${}_V\mathfrak{L}$, which is convergent in the outside of a sphere containing $V$, into a truncated Taylor series ${}_U\mathfrak{T}$, which is convergent in the bounding box $R(U)$ of $U$ (if there already existed a Taylor series, we simply add both) and we do the same with exchanged roles of $U$ and $V$. This conversion turns out to be subquadratic in $N$, which will be shown later.
Direct evaluation of these Taylor series turns out to be quadratic in $N$, with the same argument as used for the direct construction of the Laurent series. Therefore we exploit the tree structure of $\mathscr{T}$ again and go from a node to its children and combine the Taylor series of the parent and the child. This step is subquadratic in $N$, which will be shown later.
At the end we evaluate a Taylor series for each point $z_i \in Z$ which is linear in $N$.

## 4.3 Main Operators for the Fast Multipole Method

Combining the ideas from the previous sections, we can synthesize the subquadratic algorithm 4, with the main steps `INITIALIZE`, `UPCAST`, `CONVERSION`, `DOWNCAST` and `EVALUATION`. To analyze this algorithm we derive the formulas for each step in sections 4.3.1 to 4.3.5.

### 4.3.1 `INITIALIZE`

Let $U \in \mathscr{T}$ be a leaf, then $U$ represents exactly one particle. Let $z_U$ be the position of this particle and let $q_U$ be the charge of this particle, then the corresponding function $z \mapsto \frac{q_U}{z - z_U}$ is already a Laurent series with the expansion point $z_U$.
Therefore, the coefficients of the Laurent series are ${}_U\mathrm{L}_k := \begin{cases} q_U, & \text{for } k = 1 \\ 0, & \text{else.} \end{cases}$

**Algorithm 4:** Fast Multipole Method — abstract

```
Input  : A tree $\mathscr{T}$ with associated WSPD $\mathscr{W}$ for $N$ particles with charges $q_i$ and positions $z_i$ for $i = 1, \dots, N$.
Output : $(C(q,z) \cdot q)_i = \sum_{\substack{j=1 \\ j \neq i}}^{N} \frac{q_i q_j}{z_i - z_j} = {}^{i}\Phi(z_i)$ for $i = 1, \dots, N$.
// INITIALIZE
Compute the Laurent series ${}_U\mathfrak{L}(z)$ for each leaf $U \in \mathscr{T}$;
// UPCAST
for $l := depth(\mathscr{T})$ to 1 do
    foreach inner node $U \in \mathscr{T}$ with depth $l$ do
        // Let $V, W$ be the children of $U$.
        Compute the Laurent series ${}_U\mathfrak{L}(z)$ of $U$ by combining the Laurent series ${}_V\mathfrak{L}(z)$ and ${}_W\mathfrak{L}(z)$;
    end
end
// CONVERSION
foreach pair $(U, V) \in \mathscr{W}$ do
    Convert the Laurent series ${}_V\mathfrak{L}(z)$ of $V$ into a Taylor series ${}_U\mathfrak{T}(z)$ of the paired node $U$;
    Convert the Laurent series ${}_U\mathfrak{L}(z)$ of $U$ into a Taylor series ${}_V\mathfrak{T}(z)$ of the paired node $V$;
    // If there already existed Taylor series before, add both.
end
// DOWNCAST
for $l := 1$ to $depth(\mathscr{T})$ do
    foreach node $U \in \mathscr{T}$ with depth $l$ do
        // Let $V$ be the parent of $U$.
        Compute the new Taylor series ${}_U\mathfrak{T}(z)$ of $U$ by combining the old Taylor series ${}_U\mathfrak{T}(z)$ of $U$ with the Taylor series ${}_V\mathfrak{T}(z)$ of the parent node $V$;
    end
end
// EVALUATION
foreach leaf $U \in \mathscr{T}$ do
    // Let $z_U$ be the position of the point corresponding to the leaf $U$.
    Evaluate the Taylor series ${}_U\mathfrak{T}(z)$ of $U$ at $z = z_U$; // which represents evaluating ${}^{i}\Phi(z)$ at $z = z_U$.
end
```

### 4.3.2 UPCAST

The idea to combine Laurent series is to shift each series to the same expansion point and add the resulting coefficients afterwards. This shifting of Laurent series will be treated in lemma 18. Afterwards we present the formulas for the coefficients of the combined Laurent series.

**Lemma 18 (Shifting of Laurent Series).** *Let the Laurent series* ${}_0\mathfrak{L}(z) := \sum_{k=0}^{\infty} \frac{{}_0\mathrm{L}_k}{(z-z_0)^k}$, *with expansion point* $z_0$, *be holomorphic on* $\mathfrak{S}_R(z_0)^\complement$. *The shifted Laurent series* ${}_1\mathfrak{L}(z) := \sum_{k=0}^{\infty} \frac{{}_1\mathrm{L}_k}{(z-z_1)^k}$, *with expansion point* $z_1$, *is holomorphic on* $\mathfrak{S}_\rho(z_1)^\complement$, *with* $\rho = R + |z_0 - z_1|$, *and the coefficients are given by*

$$\begin{aligned} {}_1\mathrm{L}_0 &= {}_0\mathrm{L}_0, \\ {}_1\mathrm{L}_k &= \sum_{n=1}^{k} (z_0 - z_1)^{k-n}\, {}_0\mathrm{L}_n \binom{k-1}{n-1}, \qquad \text{for } k \geq 1. \end{aligned}$$

*Proof.* Let $\rho = R + |z_0 - z_1|$, let $z \in \mathfrak{S}_\rho(z_1)^\complement$ and let $1 \le n \in \mathbb{N}$, then we get

$$\begin{aligned}
\frac{{}_0\mathrm{L}_n}{(z-z_0)^n} &= \frac{{}_0\mathrm{L}_n}{(z-z_1-(z_0-z_1))^n} \\
&= (z_0-z_1)^{-n}\,{}_0\mathrm{L}_n \frac{\left(\frac{z_0-z_1}{z-z_1}\right)^n}{\left(1-\frac{z_0-z_1}{z-z_1}\right)^n} \\
\text{lemma 39} \quad &= (z_0-z_1)^{-n}\,{}_0\mathrm{L}_n \sum_{k=n-1}^{\infty} \binom{k}{n-1} \Big(\frac{z_0-z_1}{z-z_1}\Big)^{k+1} \\
&= \sum_{k=n}^{\infty} (z_0-z_1)^{-n}\,{}_0\mathrm{L}_n \binom{k-1}{n-1} \Big(\frac{z_0-z_1}{z-z_1}\Big)^{k}.
\end{aligned}$$

Hence we get for the shifted function

$$\begin{aligned}
\sum_{n=0}^{\infty} \frac{{}_0\mathrm{L}_n}{(z-z_0)^n} &= {}_0\mathrm{L}_0 + \sum_{n=1}^{\infty}\sum_{k=n}^{\infty} (z_0-z_1)^{-n}\,{}_0\mathrm{L}_n \binom{k-1}{n-1}\Big(\frac{z_0-z_1}{z-z_1}\Big)^{k} \\
\text{changing the summation order and the "path" in which the indices are traversed} \quad &= {}_0\mathrm{L}_0 + \sum_{k=1}^{\infty}\sum_{n=1}^{k} (z_0-z_1)^{-n}\,{}_0\mathrm{L}_n \binom{k-1}{n-1}\Big(\frac{z_0-z_1}{z-z_1}\Big)^{k} \\
&= {}_0\mathrm{L}_0 + \sum_{k=1}^{\infty} (z-z_1)^{-k} \underbrace{\sum_{n=1}^{k} (z_0-z_1)^{k-n}\,{}_0\mathrm{L}_n \binom{k-1}{n-1}}_{{}_1\mathrm{L}_k}.
\end{aligned}$$

Since ${}_0\mathfrak{L}(z)$ is holomorphic on $\mathfrak{S}_R(z_0)^\complement \supseteq \mathfrak{S}_\rho(z_1)^\complement$ and ${}_0\mathfrak{L}(z)$ and ${}_1\mathfrak{L}(z)$ are identical on $\mathfrak{S}_\rho(z_1)^\complement$, the claim follows. ☐

Using the previous lemma the combined Laurent series can be easily calculated. Let ${}_V\mathfrak{L}(z) := \sum_{k=0}^{\infty} \frac{{}_V\mathrm{L}_k}{(z-z_V)^k}$ and ${}_W\mathfrak{L}(z) := \sum_{k=0}^{\infty} \frac{{}_W\mathrm{L}_k}{(z-z_W)^k}$ be two Laurent series, with the expansion points $z_V$ and $z_W$, respectively, then the coefficients of the combined Laurent series ${}_U\mathfrak{L}(z) := \sum_{k=0}^{\infty} \frac{{}_U\mathrm{L}_k}{(z-z_U)^k}$, with expansion point $z_U$, are given by

$$\begin{aligned}
{}_U\mathrm{L}_0 &= {}_V\mathrm{L}_0 + {}_W\mathrm{L}_0, \\
{}_U\mathrm{L}_k &= \sum_{n=1}^{k} \Big( (z_U-z_V)^{k-n}\,{}_V\mathrm{L}_n + (z_U-z_W)^{k-n}\,{}_W\mathrm{L}_n \Big) \binom{k-1}{n-1}, \qquad \text{for } k \ge 1.
\end{aligned}$$

### 4.3.3 CONVERSION

In this section we will show how to convert a Laurent polynomial into a Taylor polynomial. Algorithmically we are only able to handle finitely many coefficients and we only need finitely many coefficients for a good approximation, hence we will restrict the analysis to truncated series, i.e. polynomials.

**Lemma 19 (Conversion from Laurent Polynomials to Taylor Series).** *Let $\mathfrak{L}(z) := \sum_{k=0}^{p} \frac{\mathrm{L}_k}{(z-z_L)^k}$, with expansion point $z_L$, be holomorphic on $\mathfrak{S}_R(z_L)^\complement$ and let $z_T \in \mathfrak{S}_R(z_L)^\complement$, then the corresponding Taylor series $\mathfrak{T}(z) := \sum_{k=0}^{\infty} \mathrm{T}_k (z-z_T)^k$, with expansion point $z_T$, is holomorphic on $\mathfrak{S}_\rho(z_T)$, with $\rho = |z_L - z_T| - R$, and the coefficients are given by*

$$\mathrm{T}_0 = \sum_{n=0}^{p} \mathrm{L}_n (z_T - z_L)^{-n},$$

$$\mathrm{T}_k = \sum_{n=1}^{p} \mathrm{L}_n \binom{k+n-1}{n-1} (z_T - z_L)^{-k-n}, \qquad \textit{for } k \geq 1.$$

*Proof.* Let $\rho = |z_L - z_T| - R$, let $z \in \mathfrak{S}_\rho(z_T)$ and let $1 \leq n \in \mathbb{N}$, then we get

$$\begin{aligned}
\frac{\mathrm{L}_n}{(z-z_L)^n} &= \frac{\mathrm{L}_n}{(z - z_T + z_T - z_L)^n} \\
&= (z_T - z)^{-n} \mathrm{L}_n \frac{\left(\frac{z_T - z}{z_T - z_L}\right)^n}{\left(1 - \frac{z_T - z}{z_T - z_L}\right)^n} \\
\text{lemma 39} \quad &= (z_T - z)^{-n} \mathrm{L}_n \sum_{k=n-1}^{\infty} \binom{k}{n-1} \left(\frac{z_T - z}{z_T - z_L}\right)^{k+1} \\
&= \sum_{k=n}^{\infty} (z_T - z)^{k-n} \mathrm{L}_n \binom{k-1}{n-1} (z_T - z_L)^{-k} \\
&= \sum_{k=0}^{\infty} (z_T - z)^{k} \mathrm{L}_n \binom{k+n-1}{n-1} (z_T - z_L)^{-k-n}.
\end{aligned}$$

Hence we get for the local expansion

$$\begin{aligned}
\sum_{n=0}^{p} \frac{\mathrm{L}_n}{(z-z_L)^n} &= \mathrm{L}_0 + \sum_{n=1}^{p} \sum_{k=0}^{\infty} (z_T - z)^k \mathrm{L}_n \binom{k+n-1}{n-1} (z_T - z_L)^{-k-n} \\
\substack{\text{changing}\\ \text{the summation order}} \quad &= \mathrm{L}_0 + \sum_{k=0}^{\infty} \sum_{n=1}^{p} (z_T - z)^k \mathrm{L}_n \binom{k+n-1}{n-1} (z_T - z_L)^{-k-n} \\
&= \mathrm{L}_0 + \sum_{n=1}^{p} \mathrm{L}_n \binom{n-1}{n-1} (z_T - z_L)^{-n} + \sum_{k=1}^{\infty} (z_T - z)^k \sum_{n=1}^{p} \mathrm{L}_n \binom{k+n-1}{n-1} (z_T - z_L)^{-k-n} \\
&= \underbrace{\mathrm{L}_0 + \sum_{n=1}^{p} \mathrm{L}_n (z_T - z_L)^{-n}}_{\mathrm{T}_0} + \sum_{k=1}^{\infty} (z_T - z)^k \underbrace{\sum_{n=1}^{p} \mathrm{L}_n \binom{k+n-1}{n-1} (z_T - z_L)^{-k-n}}_{\mathrm{T}_k}.
\end{aligned}$$

Since $\mathfrak{L}(z)$ is holomorphic on $\mathfrak{S}_R(z_L)^\complement \supseteq \mathfrak{S}_\rho(z_T)$ and $\mathfrak{L}(z)$ and $\mathfrak{T}(z)$ are identical on $\mathfrak{S}_\rho(z_T)$, the claim follows. ☐

Suppose there already existed a Taylor polynomial ${}_U\mathfrak{T}(z) := \sum_{k=0}^{q} {}_U\mathrm{T}_k (z - z_U)^k$ with the expansion point $z_U$ and want to convert another Laurent polynomial ${}_V\mathfrak{L}(z) := \sum_{k=0}^{p} {}_V\mathrm{L}_k (z - z_V)^k$ to a Taylor polynomial in $z_U$, it remains to add the coefficients in order to combine the old one with the converted one.

## 4.3.4 DOWNCAST

The idea to combine Taylor series, is equivalent to the idea for `UPCAST`, i.e. shift each series to the same expansion point and add the resulting coefficients afterwards. As in section 4.3.3, we will only consider truncated Taylor series, i.e. Taylor polynomials.

**Lemma 20 (Shifting of Polynomials).** *Let ${}_0\mathfrak{T}(z) := \sum_{k=0}^{q} {}_0\mathrm{T}_k (z - z_0)^k$ be a Taylor polynomial with expansion point $z_0$. The coefficients of the shifted polynomial ${}_1\mathfrak{T}(z) := \sum_{k=0}^{q} {}_1\mathrm{T}_k (z - z_1)^k$, with expansion point $z_1$, are given by*

$${}_1\mathrm{T}_k = \sum_{n=k}^{q} \binom{n}{k} {}_0\mathrm{T}_n (z_1 - z_0)^{n-k}, \qquad \textit{for } k \geq 0.$$

*Proof.* Let $z, z_0, z_1 \in \mathbb{C}$ and let $1 \leq n \in \mathbb{N}$, then we get

$$\begin{aligned}
{}_0\mathrm{T}_n(z-z_0)^n &= {}_0\mathrm{T}_n(z-z_1-(z_0-z_1))^n \\
&= {}_0\mathrm{T}_n(z_1-z_0)^n\big(\frac{z-z_1}{z_1-z_0}+1\big)^n \\
\text{lemma 39} \quad &= {}_0\mathrm{T}_n(z_1-z_0)^n\sum_{k=0}^{n}\binom{n}{k}\big(\frac{z-z_1}{z_1-z_0}\big)^k \\
&= \sum_{k=0}^{n}\binom{n}{k}\,{}_0\mathrm{T}_n(z_1-z_0)^n\big(\frac{z-z_1}{z_1-z_0}\big)^k.
\end{aligned}$$

Hence we get

$$\begin{aligned}
\sum_{n=0}^{q}{}_0\mathrm{T}_n(z-z_0)^n &= \sum_{n=0}^{q}\sum_{k=0}^{n}\binom{n}{k}\,{}_0\mathrm{T}_n(z_1-z_0)^n\big(\frac{z-z_1}{z_1-z_0}\big)^k \\
\substack{\text{changing the summation order}\\ \text{and the "path" in which the indices are traversed}} \quad &= \sum_{k=0}^{q}\sum_{n=k}^{q}\binom{n}{k}\,{}_0\mathrm{T}_n(z_1-z_0)^n\big(\frac{z-z_1}{z_1-z_0}\big)^k \\
&= \sum_{k=0}^{q}(z-z_1)^k\underbrace{\sum_{n=k}^{q}\binom{n}{k}\,{}_0\mathrm{T}_n(z_1-z_0)^{n-k}}_{{}_1\mathrm{T}_k}.
\end{aligned}$$

□

Using the previous lemma the combined Taylor polynomials can be easily calculated. Let ${}_U\mathfrak{T}(z) := \sum_{k=0}^{q}{}_U\mathrm{T}_k(z-z_U)^k$ and ${}_V\mathfrak{T}(z) := \sum_{k=0}^{q}{}_V\mathrm{T}_k(z-z_V)^k$ be two Taylor polynomials with the expansion points $z_U, z_V$, respectively. Then the coefficients of the combined Taylor polynomial ${}_U\mathfrak{T}(z) := \sum_{k=0}^{q}{}_U\tilde{\mathrm{T}}_k(z-z_U)^k$, with the *same* expansion point $z_U$, are given by

$${}_U\tilde{\mathrm{T}} = {}_U\mathrm{T} + \sum_{n=k}^{q}\binom{n}{k}\,{}_V\mathrm{T}_n(z_U-z_V)^{n-k}.$$

### 4.3.5 EVALUATION

In the EVALUATION step, we evaluate for each leaf $U \in \mathscr{T}$, which represents a particle with charge $q_U$ and position $z_U$, the Taylor polynomial ${}_U\mathfrak{T}(z) := \sum_{k=0}^{q}{}_U\mathrm{T}_k(z-z_U)^k$, with the expansion point $z_U$, at the point $z = z_U$. Therefore we get

$$\begin{aligned}
{}_U\mathfrak{T}(z_U) &= \sum_{k=0}^{q}{}_U\mathrm{T}_k(z_U-z_U)^k \\
&= {}_U\mathrm{T}_0, \ \text{because all other summands vanish.}
\end{aligned}$$

Hence we only return ${}_U\mathrm{T}_0$ instead of evaluating anything.

## 4.4 Fast Multipole Method — a quartic Version

By using the operators derived in section 4.3 we get the explicit version of the Fast Multipole Method in algorithm 5. We use the notation, that $z_U$ is the middle point of the bounding box $R(U)$ of a node $U \in \mathscr{T}$.

**Algorithm 5:** Fast Multipole Method

```
Input  : A tree $\mathscr{T}$ with associated WSPD $\mathscr{W}$ for $N$ particles with charges $q_i$ and positions $z_i$ for $i = 1, \dots, N$.
Output : $(C(q,z) \cdot q)_i = \sum_{\substack{j=1 \\ j \neq i}}^{N} \frac{q_i q_j}{z_i - z_j} = {}^{i}\Phi(z_i)$, for all $i = 1, \dots, N$.
// INITIALIZE, cf. section 4.3.1
foreach leaf $U \in \mathscr{T}$ do // Compute the Laurent polynomial ${}_U\mathfrak{L}(z)$
    $\mathrm{L}_1 := q_U$,  $\mathrm{L}_k := 0$, for all $k = 2, \dots, p$;
end
foreach node $U \in \mathscr{T}$ do // Initialize a Taylor polynomial, which is constant 0.
    $\mathrm{T}_k := 0$, for all $k = 1, \dots, q$;
end
// UPCAST, cf. section 4.3.2
for $l := depth(\mathscr{T})$ to 1 do // Iterate from the leaves to the root.
    foreach inner node $U \in \mathscr{T}$ with depth $l$ do // Compute the coefficients of the Laurent polynomial ${}_U\mathfrak{L}(z)$ by combining the Laurent polynomials ${}_V\mathfrak{L}(z), {}_W\mathfrak{L}(z)$ of the children $V, W$.
        // Let $V, W$ be the two children of $U$.
        ${}_U\mathrm{L}_k = \sum_{n=1}^{k} \left( (z_U - z_V)^{k-n}\, {}_V\mathrm{L}_n + (z_U - z_W)^{k-n}\, {}_W\mathrm{L}_n \right) \binom{k-1}{n-1}$, for all $k = 1, \dots, p$;
    end
end
// CONVERSION, cf. section 4.3.3
foreach $(U, V) \in \mathscr{W}$ do // Compute the coefficients of the Taylor polynomial ${}_U\mathfrak{L}(z)$ by converting the Laurent polynomial ${}_V\mathfrak{L}(z)$ and repeat with exchanged roles of $U$ and $V$.
    ${}_U\mathrm{T}_0 := {}_U\mathrm{T}_0 + \sum_{n=1}^{p} {}_V\mathrm{L}_n (z_U - z_V)^{-n}$;
    ${}_U\mathrm{T}_k := {}_U\mathrm{T}_k + \sum_{n=1}^{p} {}_V\mathrm{L}_n \binom{k+n-1}{n-1} (z_U - z_V)^{-k-n}$, for all $k = 1, \dots, q$;
    Repeat with exchanged roles of $U$ and $V$;
end
// DOWNCAST, cf. section 4.3.4
for $l := 1$ to $depth(\mathscr{T})$ do // Iterate from the root to the leaves.
    foreach node $U \in \mathscr{T}$ with depth $l$ do // Compute the coefficients of the new Taylor polynomial ${}_U\mathfrak{T}(z)$ by combining the Taylor polynomial ${}_V\mathfrak{L}(z)$ of the parent $V$ with the old Taylor polynomial ${}_U\mathfrak{T}(z)$.
        // Let $V$ be the parent of $U$.
        ${}_U\mathrm{T}_k := {}_U\mathrm{T}_k + \sum_{n=k}^{q} \binom{n}{k}\, {}_V\mathrm{T}_n (z_U - z_V)^{n-k}$, for all $k = 0, \dots, q$;
    end
end
// EVALUATION, cf. section 4.3.5
foreach leaf $U \in \mathscr{T}$ do // Evaluate the Taylor polynomial ${}_U\mathfrak{T}(z)$ at $z = z_U$.
    Return ${}_U\mathrm{T}_0$;
end
```

**Lemma 21.** *If computing* `UPCAST`*,* `CONVERSION` *and* `DOWNCAST` *straight forward, i.e. evaluating the expressions directly, this yields to the following runtime (in unit cost).*

- `INITIALIZE` $\in \mathcal{O}(Np + |\mathscr{T}|q) = \mathcal{O}(N(p+q))$.
- `UPCAST` $\in \mathcal{O}((|\mathscr{T}| - N)p^2) = \mathcal{O}(Np^2)$.
- `CONVERSION` $\in 2\mathcal{O}(|\mathscr{W}|pq) = \mathcal{O}(s^d Npq)$.
- `DOWNCAST` $\in \mathcal{O}(|\mathscr{T}|q^2) = \mathcal{O}(Nq^2)$.
- `EVALUATION` $\in \mathcal{O}(N)$.

*Whereas $s$ denotes the separation ratio of the WSPD and $d$ denotes the dimension. Therefore, the overall runtime, measured in unit cost, is linear in $N$ and quadratic in $p$ and $q$, while omitting logarithmic terms.*
*But the overall runtime, measured in bit cost, in is linear in $N$ and quartic in $p$ and $q$, while omitting logarithmic terms.*

*Proof.* The runtime, measured in *unit cost*, of `INITIALIZE` and `EVALUATE` is obvious. By using Horner's Scheme and lemma 14 we get the claimed *unit cost* runtime for `UPCAST`, `CONVERSION` and `DOWNCAST`.
Analysis of the formulas for `UPCAST`, `CONVERSION` and `DOWNCAST` shows that the intermediate results become too large, which results in additional factors of $p$ or $q$, respectively, in terms of *bit cost* runtime. Since we will derive a better algorithm later on, we will omit the rigorous proof of this claim. □

## 4.5 Towards a Fast Multipole Method linear in $N$ and cubic in $p, q$

In this section we will improve the runtime of `UPCAST`, `CONVERSION` and `DOWNCAST` to be linear in $N$ and cubic in $p, q$, with respect to *bit complexity*. In section 4.2 we have already seen, that we cannot omit these steps otherwise the runtime would be quadratic in $N$.
In sections 4.5.1 to 4.5.3 we will rewrite the expressions for `UPCAST`, `CONVERSION` and `DOWNCAST` to reveal an interesting structure of these expressions which can be exploited to get the promised runtime. The structure was first examined by Greengard and Rokhlin [13] to speed up the continuous Fast Multipole Method using a Fast Fourier Transform. This approach is not possible in our case, but we will exploit this structure in a different way to reduce the overall runtime by a factor of $p, q$. In the following we will use the convention that the *first bit* of an integer is the *least significant bit.*

### 4.5.1 `UPCAST`

The expression for `UPCAST` is equivalent to

$$\begin{aligned}\frac{{}_U\mathrm{L}_k}{(k-1)!} &= \frac{1}{(k-1)!}\sum_{n=1}^{k}\left((z_V - z_U)^{k-n}\,{}_V\mathrm{L}_n + (z_W - z_U)^{k-n}\,{}_W\mathrm{L}_n\right)\binom{k-1}{n-1} \\ &= \sum_{n=1}^{k}\frac{(z_V - z_U)^{k-n}}{(k-n)!}\frac{{}_V\mathrm{L}_n}{(n-1)!} + \sum_{n=1}^{k}\frac{(z_W - z_U)^{k-n}}{(k-n)!}\frac{{}_W\mathrm{L}_n}{(n-1)!}.\end{aligned}$$

The idea to compute all $\frac{{}_U\mathrm{L}_k}{(k-1)!}$, for $k = 1, \ldots, p$, simultaneously, is to encode all information in two large integers and multiply them afterwards — this approach is called Kronecker Embedding, cf. examples 22 to 24 and the short paragraph in the abstract. The procedure is split into eight steps:

1. Define an integer ${}^{(p,P)}_{V}\mathscr{L}$ consisting of $p$ blocks of size $P$. This integer is called the *Laurent coefficient integer* of $V$.
2. Encode the scaled Laurent coefficients $\frac{{}_V\mathrm{L}_j}{(j-1)!}$, for $j = 1, \ldots, p$, in the $j$-th block of ${}^{(p,P)}_{V}\mathscr{L}$.
3. Define an integer ${}^{(p,P)}_{(U,V)}\mathscr{D}$ consisting of $p$ blocks of size $P$. This integer is called the *distance integer* of $(U, V)$.
4. Encode the scaled distances $\frac{(z_V - z_U)^{i-1}}{(i-1)!}$, for $i = 1, \ldots, p$, in the $i$-th block of ${}^{(p,P)}_{(U,V)}\mathscr{D}$.
5. Multiply both integers and save the result.
6. Repeat steps $1 - 5$ with $W$ and $V$ exchanged.
7. Add the two saved integers from step 5.
8. The coefficients $\frac{{}_U\mathrm{L}_k}{(k-1)!}$, for $k = 1, \ldots, p$, are now in the first $p$ blocks of the result of step 7. (The integer of the first $p$ blocks will be called ${}^{(p,P)}_{U}\mathscr{L}$ later.)

Example 22 shows what is happening during the multiplication.

Here we have assumed that all numbers are integers, otherwise this approach would not be possible. Later on we will modify the expressions to assure that all numbers will be integers.

**Example 22 (The Kronecker Embedding for `UPCAST`).** Let $p = 3, x = z_V - z_U$. We will show steps $1 - 5$; the remaining steps are obvious. The bars $|$ represent the borders of the blocks and we will use the "normal school method" to compute the product. $*$ denotes a value which will not be used, hence we will omit this value.

$$
\begin{array}{c|c|c|c|c|c|c|c|}
 & \frac{x^2}{2!} & \frac{x^1}{1!} & \frac{x^0}{0!} & \times & \frac{{}_V\mathrm{L}_3}{2!} & \frac{{}_V\mathrm{L}_2}{1!} & \frac{{}_V\mathrm{L}_1}{0!} \\
 & & & & & & & \\
= & & & & & \frac{x^0}{0!}\frac{{}_V\mathrm{L}_3}{2!} & \frac{x^0}{0!}\frac{{}_V\mathrm{L}_2}{1!} & \frac{x^0}{0!}\frac{{}_V\mathrm{L}_1}{0!} \\
+ & & & & \frac{x^1}{1!}\frac{{}_V\mathrm{L}_3}{2!} & \frac{x^1}{1!}\frac{{}_V\mathrm{L}_2}{1!} & \frac{x^1}{1!}\frac{{}_V\mathrm{L}_1}{0!} & 0 \\
+ & & & \frac{x^2}{2!}\frac{{}_V\mathrm{L}_3}{2!} & \frac{x^2}{2!}\frac{{}_V\mathrm{L}_2}{1!} & \frac{x^2}{2!}\frac{{}_V\mathrm{L}_1}{0!} & 0 & 0 \\
 & & & & & & & \\
= & & & & & \frac{x^{3-3}}{(3-3)!}\frac{{}_V\mathrm{L}_3}{(3-1)!} & \frac{x^{2-2}}{(2-2)!}\frac{{}_V\mathrm{L}_2}{(2-1)!} & \frac{x^{1-1}}{(1-1)!}\frac{{}_V\mathrm{L}_1}{(1-1)!} \\
+ & & & & \frac{x^{3-2}}{(3-2)!}\frac{{}_V\mathrm{L}_3}{(3-1)!} & \frac{x^{3-2}}{(3-2)!}\frac{{}_V\mathrm{L}_2}{(2-1)!} & \frac{x^{2-1}}{(2-1)!}\frac{{}_V\mathrm{L}_1}{(1-1)!} & 0 \\
+ & & & \frac{x^{3-1}}{(3-1)!}\frac{{}_V\mathrm{L}_3}{(3-1)!} & \frac{x^{3-1}}{(3-1)!}\frac{{}_V\mathrm{L}_2}{(2-1)!} & \frac{x^{3-1}}{(3-1)!}\frac{{}_V\mathrm{L}_1}{(1-1)!} & 0 & 0 \\
 & & & & & & & \\
= & & & * & * & \sum_{n=1}^{3}\frac{x^{3-n}}{(3-n)!}\frac{{}_V\mathrm{L}_n}{(n-1)!} & \sum_{n=1}^{2}\frac{x^{2-n}}{(2-n)!}\frac{{}_V\mathrm{L}_n}{(n-1)!} & \sum_{n=1}^{1}\frac{x^{1-n}}{(1-n)!}\frac{{}_V\mathrm{L}_n}{(n-1)!} \\
 & & & & & & & \\
= & & & * & * & \text{belongs to } \frac{{}_U\mathrm{L}_3}{(3-1)!} & \text{belongs to } \frac{{}_U\mathrm{L}_2}{(2-1)!} & \text{belongs to } \frac{{}_U\mathrm{L}_1}{(1-1)!}
\end{array}
$$

The result encodes the "$(v)$-part" of the coefficients $\frac{{}_U\mathrm{L}_1}{(1-1)!}, \frac{{}_U\mathrm{L}_2}{(2-1)!}, \frac{{}_U\mathrm{L}_3}{(3-1)!}$ in the first three blocks. By doing the same for $W$ we can compute $\frac{{}_U\mathrm{L}_k}{(k-1)!}$. We see that each block has to have the same size, otherwise the alignment would be destroyed. The size of each block has to be as large as the size of the largest output coefficient, otherwise there could be an carryover between two blocks.

## 4.5.2 CONVERSION

The expression for `CONVERSION` is equivalent to

$$
\begin{aligned}
k!\,{}_U\mathrm{T}_k = k! \sum_{n=1}^{p} {}_V\mathrm{L}_n \binom{k+n-1}{n-1} (z_u - z_v)^{-k-n} \\
= \sum_{n=1}^{p} \frac{{}_V\mathrm{L}_n}{(n-1)!} \frac{(k+n-1)!}{(z_u - z_v)^{k+n}}
\end{aligned}
$$

The idea to compute all $k!\,{}_U\mathrm{T}_k$, for $k = 0, \ldots, q$, simultaneously, is the Kronecker Embedding, again.

The procedure is split into six steps:

1. Define an integer ${}^{(p,Q)}_{\phantom{(}V}\mathscr{L}$ consisting of $p$ blocks of size $Q$.

2. Encode the scaled Laurent coefficients $\frac{{}_V\mathrm{L}_j}{(j-1)!}$ in the $j$-th block of ${}^{(p,Q)}_{\phantom{(}V}\mathscr{L}$. (This is exactly the same integer as ${}^{(p,Q)}_{\phantom{(}V}\mathscr{L}$ in `UPCAST`, up to the block size $Q$.)
3. Define an integer ${}^{(p+q,Q)}_{(V,U)}\mathscr{R}$ consisting of $p+q$ blocks of size $Q$. This integer is called the *reciprocal integer* of $(V,U)$.
4. Encode the scaled reciprocal distances $\frac{(i-1)!}{(z_U-z_V)^{i-1}}$, for $i=1,\ldots,p+q$, in the $i$-th block of ${}^{(p+q,Q)}_{(V,U)}\mathscr{R}$.
5. Multiply both integers.
6. The coefficients $k!\,{}_U\mathrm{T}_k$, for $k=0,\ldots,q$, are now in the blocks $p$ to $p+q$ of the result of step 5. (The integer of the blocks blocks $p$ to $p+q$ will be called ${}^{(q+1,Q)}_{\phantom{(}U}\mathscr{T}$ later.)

Example 23 shows what is happening during the multiplication.

Here we have assumed that all numbers are integers, otherwise this approach would not be possible. Later on we will modify the expressions to assure that all numbers are integers.

**Example 23 (The Kronecker Embedding for `CONVERSION`).** Let $p=2, q=3, x=z_U-z_V$. We will show steps $1-6$. As in example 22, the bars | represent the borders of the blocks and we will use the "normal school method" to compute the product and $*$ denotes a value which will not be used, hence we will omit the value.

$$\begin{array}{r|c|c|c|c|c|c|c|c|}
 & \frac{{}_V\mathrm{L}_2}{1!} & \frac{{}_V\mathrm{L}_1}{0!} & \times & \frac{0!}{x^1} & \frac{1!}{x^2} & \frac{2!}{x^3} & \frac{3!}{x^4} & \frac{4!}{x^5} \\
 & & & & & & & & \\
= & & & & \frac{{}_V\mathrm{L}_1}{0!}\frac{0!}{x^1} & \frac{{}_V\mathrm{L}_1}{0!}\frac{1!}{x^2} & \frac{{}_V\mathrm{L}_1}{0!}\frac{2!}{x^3} & \frac{{}_V\mathrm{L}_1}{0!}\frac{3!}{x^4} & \frac{{}_V\mathrm{L}_1}{0!}\frac{4!}{x^5} \\
+ & & \frac{{}_V\mathrm{L}_2}{1!}\frac{0!}{x^1} & & \frac{{}_V\mathrm{L}_2}{1!}\frac{1!}{x^2} & \frac{{}_V\mathrm{L}_2}{1!}\frac{2!}{x^3} & \frac{{}_V\mathrm{L}_2}{1!}\frac{3!}{x^4} & \frac{{}_V\mathrm{L}_2}{1!}\frac{4!}{x^5} & 0 \\
 & & & & & & & & \\
= & & * & & \sum_{n=1}^{2}\frac{{}_V\mathrm{L}_n}{(n-1)!}\frac{(0+n-1)!}{x^{0+n}} & \sum_{n=1}^{2}\frac{{}_V\mathrm{L}_n}{(n-1)!}\frac{(1+n-1)!}{x^{1+n}} & \sum_{n=1}^{2}\frac{{}_V\mathrm{L}_n}{(n-1)!}\frac{(2+n-1)!}{x^{2+n}} & \sum_{n=1}^{2}\frac{{}_V\mathrm{L}_n}{(n-1)!}\frac{(3+n-1)!}{x^{3+n}} & * \\
 & & & & & & & & \\
= & & * & & =0!\,{}_U\mathrm{T}_0 & =1!\,{}_U\mathrm{T}_1 & =2!\,{}_U\mathrm{T}_2 & =3!\,{}_U\mathrm{T}_3 & *
\end{array}$$

### 4.5.3 `DOWNCAST`

The expression for `DOWNCAST` is equivalent to

$$\begin{aligned}
k!\,{}_U\mathrm{T}_k &= k!\sum_{n=k}^{q}\binom{n}{k}\,{}_V\mathrm{T}_n(z_U-z_V)^{n-k} \\
&= \sum_{n=k}^{q} n!\,{}_V\mathrm{T}_n\frac{(z_U-z_V)^{n-k}}{(n-k)!}.
\end{aligned}$$

The idea to compute all $k!\,{}_U\mathrm{T}_k$, for $k=0,\ldots,q$, simultaneously, is the Kronecker Embedding, again.

The procedure is split into six steps:

1. Define an integer ${}^{(q+1,Q)}_{\phantom{(}V}\mathscr{T}$ consisting of $q+1$ blocks of size $Q$. This integer is called the *Taylor coefficient integer* of $V$.
2. Encode $j!\,{}_V\mathrm{T}_j$ in the $(q-j)$-th block of ${}^{(q+1,Q)}_{\phantom{(}V}\mathscr{T}$, for $j=0,\ldots,q$. (This is exactly the same integer returned by `CONVERSION`.)
3. Define an integer ${}^{(q+1,Q)}_{(V,U)}\mathscr{D}$ consisting of $q+1$ blocks of size $Q$.

4. Encode $\frac{(z_U - z_V)^{i-1}}{(i-1)!}$ in the $i$-th block of ${}^{(q+1,Q)}_{(V,U)}\mathscr{D}$, for $i = 1, \dots, q+1$. (The first $p$ blocks are exactly the same blocks of ${}^{(p,P)}_{(V,U)}\mathscr{D}$ in `UPCAST`, up to the size $Q$.)
5. Multiply both integers.
6. The coefficients $k!\,{}_U\mathrm{T}_k$, for $k = 0, \dots, q$, are now in the first $q+1$ blocks of the result of step 5. (The integer of the blocks first $q+1$ blocks will be called ${}^{(q+1,Q)}_{U}\mathscr{T}$ later.)

Example 24 shows what is happening during the multiplication.

Here we have assumed that all numbers are integers, otherwise this approach would not be possible. Later on we will modify the expressions to assure that all numbers are integers.

**Example 24 (The Kronecker Embedding for `DOWNCAST`).** Let $q = 3, x = z_U - z_V$. We will show steps $1 - 5$. As in examples 22 and 23, the bars $|$ represent the borders of the blocks and we will use the "normal school method" to compute the product and $*$ denotes a value which will not be used, hence we will omit the value.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | $\frac{x^2}{2!}$ | $\frac{x^1}{1!}$ | $\frac{x^0}{0!}$ | $\times$ | $0!\,{}_V\mathrm{T}_0$ | $1!\,{}_V\mathrm{T}_1$ | $2!\,{}_V\mathrm{T}_2$ |
| $=$ | | | | | $\frac{x^0}{0!}0!\,{}_V\mathrm{T}_0$ | $\frac{x^0}{0!}1!\,{}_V\mathrm{T}_1$ | $\frac{x^0}{0!}2!\,{}_V\mathrm{T}_2$ |
| $+$ | | | | $\frac{x^1}{1!}0!\,{}_V\mathrm{T}_0$ | $\frac{x^1}{1!}1!\,{}_V\mathrm{T}_1$ | $\frac{x^1}{1!}2!\,{}_V\mathrm{T}_2$ | $0$ |
| $+$ | | | $\frac{x^2}{2!}0!\,{}_V\mathrm{T}_0$ | $\frac{x^2}{2!}1!\,{}_V\mathrm{T}_1$ | $\frac{x^2}{2!}2!\,{}_V\mathrm{T}_2$ | $0$ | $0$ |
| $=$ | | | | | $\frac{x^{0-0}}{(0-0)!}0!\,{}_V\mathrm{T}_0$ | $\frac{x^{1-1}}{(1-1)!}1!\,{}_V\mathrm{T}_1$ | $\frac{x^{2-2}}{(2-2)!}2!\,{}_V\mathrm{T}_2$ |
| $+$ | | | | $*$ | $\frac{x^{1-0}}{(1-0)!}1!\,{}_V\mathrm{T}_1$ | $\frac{x^{2-1}}{(2-1)!}2!\,{}_V\mathrm{T}_2$ | $0$ |
| $+$ | | | $*$ | $*$ | $\frac{x^{2-0}}{(2-0)!}2!\,{}_V\mathrm{T}_2$ | $0$ | $0$ |
| $=$ | | | $*$ | $*$ | $\sum_{n=0}^{2}\frac{x^{n-0}}{(n-0)!}n!\,{}_V\mathrm{T}_n$ | $\sum_{n=1}^{2}\frac{x^{n-1}}{(n-1)!}n!\,{}_V\mathrm{T}_n$ | $\sum_{n=2}^{2}\frac{x^{n-2}}{(n-2)!}n!\,{}_V\mathrm{T}_n$ |
| $=$ | | | $*$ | $*$ | $= 0!\,{}_U\mathrm{T}_0$ | $= 0!\,{}_U\mathrm{T}_1$ | $= 0!\,{}_U\mathrm{T}_2$ |

### 4.5.4 `INITIALIZE` and `EVALUATION`

We have seen that `UPCAST` and `CONVERSION` use the Laurent coefficient integer ${}^{(p,Q)}_{V}\mathscr{L}$ which is exactly the return value of `UPCAST` (up to a different size of blocks, but this can be repaired easily) and `DOWNCAST` uses the Taylor coefficient integer ${}^{(q+1,Q)}_{V}\mathscr{T}$ which is exactly the return value of `DOWNCAST` and `CONVERSION`. The distance integer ${}^{(q+1,Q)}_{(V,U)}\mathscr{D}$ used by `DOWNCAST` has the same first $p$ blocks of the distance integer ${}^{(p,P)}_{(V,U)}\mathscr{D}$ used by `UPCAST` (up to a different size of blocks, but this can be repaired easily). Therefore it would be convenient if `INITIALIZE` and `EVALUATION` would use exactly the same integers. This task will be solved within this small section.

While `INITIALIZE` we should define the Laurent coefficient integer ${}^{(p,P)}_{U}\mathscr{L}$, with $p$ blocks of size $P$, of each leaf $U \in \mathscr{T}$. For

the leaf $U \in \mathscr{T}$ the first block of the integer gets initialized by the value $q_U$ and the remaining blocks get initialized with 0. Afterwards we should define the Taylor coefficient integer ${}^{(q+1,Q)}_{U}\mathscr{T}$, with $q+1$ blocks of size $Q$, for each node $U \in \mathscr{T}$ and initialize it with 0 for each block. This is equivalent to our first version of `INITIALIZE`.
In the `EVALUATION` step, we return for each leaf $U \in \mathscr{T}$ the last block of the Taylor coefficient integer ${}^{(q+1,Q)}_{U}\mathscr{T}$. This block contains the coefficient $0!\,{}_U\mathrm{T}_0 = {}_U\mathrm{T}_0 = {}^U\Phi(z_U)$.

### 4.5.5 Fast Multipole Method linear in $N$ and cubic in $p, q$

Combining the ideas from sections 4.5.1 to 4.5.4, we can synthesize algorithm 6, which is linear in $N$ and cubic in $p, q$ with respect to *bit cost*. We will use the notation ${}^{(x,y)}_{U}\mathscr{L}$, ${}^{(x,y)}_{U}\mathscr{T}$, ${}^{(x,y)}_{(V,U)}\mathscr{D}$, ${}^{(x,y)}_{(U,V)}\mathscr{R}$ are the Laurent coefficient integer, the Taylor coefficient integer, the distance integer and the reciprocal integer, with $x$ blocks of size $y$ and $z_U$ denotes the middle point of the bounding box $R(U)$ of $U$. The claimed runtime can be seen by combining sections 5.6.2, 5.6.3 and 5.6.5 and is given by $\mathscr{O}_2((|\mathscr{T}| + |\mathscr{W}|)(p+q)^3) = \mathscr{O}_2(N(p+q)^3)$, by using the properties of the WSPD. Since we will derive an even faster algorithm in the next chapter we omit a detailed discussion of the claimed runtime.

**Algorithm 6:** Fast Multipole Method

```
Input  : A tree $\mathscr{T}$ with associated WSPD $\mathscr{W}$ for $N$ particles with charges $q_i$ and positions $z_i$ for $i = 1, \dots, N$.
Output : $(C(q,z) \cdot q)_i = \sum_{\substack{j=1 \\ j \neq i}}^{N} \frac{q_i q_j}{z_i - z_j} = {}^{i}\Phi(z_i)$,  for all $i = 1, \dots, N$.
// NOTATION
// ${}^{(x,y)}_{U}\mathscr{L}$, ${}^{(x,y)}_{U}\mathscr{T}$, ${}^{(x,y)}_{(V,U)}\mathscr{D}$, ${}^{(x,y)}_{(U,V)}\mathscr{R}$ are the Laurent coefficient integer, the Taylor coefficient integer, the
   distance integer and the reciprocal integer, with $x$ blocks of size $y$.
// INITIALIZE, cf. section 4.5.4
1  Define the Laurent coefficient integer ${}^{(p,P)}_{U}\mathscr{L}$, with $p$ blocks of size $P$, whereas the first block gets initialized by the value $q_U$
   and the remaining blocks get initialized with 0, for each leaf $U \in \mathscr{T}$;
2  Define the Taylor coefficient integer ${}^{(q+1,Q)}_{U}\mathscr{T}$, with $q+1$ blocks of size $Q$, whereas each block gets initialized with 0, for
   each node $U \in \mathscr{T}$;
// UPCAST, cf. section 4.5.1
3  for $l := depth(\mathscr{T})$ to 1 do
4      foreach inner node $U \in \mathscr{T}$ with depth $l$ do
           // Let $V, W$ be the two children of $U$.
5          Encode the scaled distances $\frac{(z_V - z_U)^{i-1}}{(i-1)!}$, for $i = 1, \dots, p$, in the $i$-th block of ${}^{(p,P)}_{(U,V)}\mathscr{D}$;
6          Encode the scaled distances $\frac{(z_W - z_U)^{i-1}}{(i-1)!}$, for $i = 1, \dots, p$, in the $i$-th block of ${}^{(p,P)}_{(U,W)}\mathscr{D}$;
7          ${}^{(p,P)}_{U}\mathscr{L} := {}^{(p,P)}_{(U,V)}\mathscr{D} \cdot {}^{(p,P)}_{V}\mathscr{L} + {}^{(p,P)}_{(U,W)}\mathscr{D} \cdot {}^{(p,P)}_{W}\mathscr{L}$ truncated to the last $p$ blocks. // now, $\frac{{}_{U}\mathrm{L}_i}{(i-1)!}$ is in the $i$-th block.
8      end
9  end
// CONVERSION, cf. section 4.5.2
10 foreach $(U, V) \in \mathscr{W}$ do
11     Encode the scaled reciprocal distances $\frac{(i-1)!}{(z_U - z_V)^{i-1}}$, for $i = 1, \dots, p+q$, in the $i$-th block of ${}^{(p+q,Q)}_{(V,U)}\mathscr{R}$;
12     Expand each block of ${}^{(p,P)}_{V}\mathscr{L}$ to size $Q$ and rename it to ${}^{(p,Q)}_{V}\mathscr{L}$;
13     ${}^{(q+1,Q)}_{U}\mathscr{T}^{\text{conv}} := {}^{(p+q,Q)}_{(U,V)}\mathscr{R} \cdot {}^{(p,Q)}_{V}\mathscr{L}$ truncated to the blocks from $p$ to $p+q$;
14     ${}^{(q+1,Q)}_{U}\mathscr{T} := {}^{(q+1,Q)}_{U}\mathscr{T} + {}^{(q+1,Q)}_{U}\mathscr{T}^{\text{conv}}$ // now, $i!\,{}_{U}\mathrm{T}_i$ is in the $(q-i)$-th block.
15     Redo these steps with interchanged roles of $U$ and $V$;
16 end
// DOWNCAST, cf. section 4.5.3
17 for $l := 1$ to $depth(\mathscr{T})$ do
18     foreach node $U \in \mathscr{T}$ with depth $l$ do
           // Let $V$ be the parent of $U$.
19         Encode $\frac{(z_U - z_V)^{i-1}}{(i-1)!}$ in the $i$-th block of ${}^{(q+1,Q)}_{(V,U)}\mathscr{D}$, for $i = 1, \dots, q+1$ ;
20         ${}^{(q+1,Q)}_{U}\mathscr{T}^{\text{down}} := {}^{(q+1,Q)}_{V}\mathscr{T} \cdot {}^{(q+1,Q)}_{(V,U)}\mathscr{D}$ truncated to the last $p$ blocks;
21         ${}^{(q+1,Q)}_{U}\mathscr{T} := {}^{(q+1,Q)}_{U}\mathscr{T} + {}^{(q+1,Q)}_{U}\mathscr{T}^{\text{down}}$ // now, $i!\,{}_{U}\mathrm{T}_i$ is in the $(q-i)$-th block.
22     end
23 end
// EVALUATION, cf. section 4.5.4
24 Return the last block of the Taylor coefficient integer ${}^{(q+1,Q)}_{U}\mathscr{T}$, for each leaf $U \in \mathscr{T}$;
```

Chapter 5

# Truncated Tree Fast Multipole Method

Remember, the runtime of the "$p, q$-cubic" algorithm from the previous section was given by $\mathcal{O}_2((|\mathcal{T}| + |\mathcal{W}|)(p+q)^3)$. And using the "normal" WSPD we get $|\mathcal{T}| + |\mathcal{W}| \in \mathcal{O}(N)$. In order to develop an even faster FMM we try to shrink the tree, such that $|\mathcal{T}| + |\mathcal{W}| \in \mathcal{O}(N/(p+q))$. If this would be possible the corresponding algorithm would have runtime $\mathcal{O}(N(p+q)^2)$.
In the former WSPD the tree $\mathcal{T}$ had 1 particle per leaf. If we would be able to construct a tree with $\mathcal{O}(M)$ particles per leaf, the remaining tree would have size $\mathcal{O}(N/M)$ and $|\mathcal{W}| \in \mathcal{O}(N/M)$. Therefore we will now aim for trees with $\mathcal{O}(M)$ particles per leaf. When dealing with leaves of size $M$, the steps `INITIALIZE` and `EVALUATE` have to be changed. During `INITIALIZE` we construct a Laurent series using multiple particles, which is done in lemma 25. In the `EVALUATE` step we evaluate the Taylor series now at $M$ different points, instead of simply returning the constant coefficient. Section 5.1 is the groundwork for this algorithm, in this section we consider leaves which represent multiple particles and the influence on the corresponding Laurent and Taylor series. In order to work with leaves with more than one particle we need a new Well-Separated Pair Decomposition which will be introduced in section 5.2. Using these new techniques we are able to define new operators for our algorithm which will be used afterwards to develop the new algorithm in section 5.3.
In section 5.4 we take care of the sensitivity of the problem, i.e. the induced errors by rounding the input, and in section 5.5 we consider the other errors which occur, e.g. the truncation errors. These two sections ensure that the overall error gets bounded by $2^{-\mathbb{E}}$. In section 5.6 we determine the runtime of each step and combine all results in section 5.7 to proof the claimed runtime being linear in the number of particles $N$ and quadratic in the error exponent $\mathbb{E}$, which is nearly optimal.

## 5.1 Truncated Laurent and Taylor Series for multiple Particles

Lemmas 25 and 26 are essential for

- An rigorous error analysis of the algorithm. (Since we use truncated Laurent and Taylor series within our algorithm we need to control the truncation error.)
- Accelerating the algorithm. (We will get new operators, which allow us to skip several steps of `UPCAST`, `CONVERSION` and `DOWNCAST`.)

**Lemma 25 (Truncated Laurent Series for multiple particles).** *For $i = 1, \dots, M$, let $(q_i, z_i)$ represent $M$ particles with the associated Laurent series ${}_i\mathfrak{L}(z) := \frac{q_i}{z-z_i}$. The Laurent series for all $M$ particles ${}_{[M]}\mathfrak{L}(z) := \sum_{i=1}^{M} {}_i\mathfrak{L}(z) = \sum_{k=0}^{\infty} \frac{\mathrm{L}_k}{(z-z_L)^k}$, with expansion point $Z_L$, is holomorphic on $\mathfrak{S}_R(z_L)^{\complement}$, with $R = \max_i |z_L - z_i|$ and the coefficients are given by*

$$\mathrm{L}_0 = 0,$$
$$\mathrm{L}_k = \sum_{i=1}^{M} q_i (z_i - z_L)^{k-1}, \qquad \textit{for } k \geq 1.$$

*Moreover, let $\mathcal{Q} = \sum_{i=1}^{M} |q_i|$, then the values of the coefficients can be estimated by*

$$|\mathrm{L}_k| \leq \mathcal{Q} R^{k-1}.$$

*Let $z \in \mathfrak{S}_{cR}(z_L)^{\complement}$, with $\mathbb{R} \ni c \geq 1$. When truncating the Laurent series up to $p$-th order, the error will be at most*

$$\Big|\, {}_{[M]}\mathfrak{L}(z) - \sum_{k=0}^{p} \frac{\mathrm{L}_k}{(z-z_L)^k} \Big| \leq \frac{\mathcal{Q}}{c^p(c-1)R}.$$

*Proof.* At first we consider the coefficients. The idea is to shift each function to $z_L$ and add the coefficients afterwards.

$$\begin{aligned}
{}_{[M]}\mathfrak{L}(z) &= \sum_{i=1}^{M} \frac{q_i}{z-z_i} \\
&= \sum_{i=1}^{M} {}_i b_0 + \sum_{k=1}^{\infty} \frac{{}_i b_k}{(z-z_L)^k} \quad \text{with } {}_i b_0 = {}_i\mathrm{L}_0 = 0 \text{ and } {}_i b_k = \sum_{n=0}^{k} (z_i - z_L)^{k-n} \underbrace{{}_i\mathrm{L}_n}_{{}_i\mathrm{L}_n = q_i,\ \text{for } k=1;\quad {}_i\mathrm{L}_n = 0,\ \text{else}} \binom{k-1}{n-1} \text{ for } k \geq 1 \\
&= \sum_{i=1}^{M} \sum_{k=1}^{\infty} (z_i - z_L)^{k-1} q_i \binom{k-1}{1-1} (z-z_L)^{-k} \\
&= \sum_{k=1}^{\infty} (z-z_L)^{-k} \underbrace{\sum_{i=1}^{M} q_i (z_i - z_L)^{k-1}}_{\mathrm{L}_k}.
\end{aligned}$$

The next step is to prove the estimate for the coefficients. Let $\mathcal{Q} = \sum_{i=1}^{M} |q_i|$, then we get

$$\begin{aligned}
|\mathrm{L}_k| &= \Big| \sum_{i=1}^{M} q_i (z_i - z_L)^{k-1} \Big| \\
&\leq \sum_{i=1}^{M} |q_i| |z_i - z_L|^{k-1} \\
&\leq \mathcal{Q} R^{k-1}.
\end{aligned}$$

In the last step we will prove the error estimate. Let $z \in \mathfrak{S}_{cR}(z_L)^{\complement}$, with $\mathbb{R} \ni c \geq 1$.

$$\begin{aligned}
\Big|\, {}_{[M]}\mathfrak{L}(z) - \sum_{k=0}^{p} (z-z_L)^{-k} \mathrm{L}_k \Big| &\leq \sum_{k=p+1}^{\infty} |z-z_L|^{-k} |\mathrm{L}_k| \\
&\leq \sum_{k=p+1}^{\infty} (cR)^{-k} \mathcal{Q} R^{k-1} \\
&= \frac{\mathcal{Q}}{cR} \sum_{k=p}^{\infty} \Big(\frac{1}{c}\Big)^k \\
\text{lemma 39} \quad &= \frac{\mathcal{Q}}{cR} \frac{\left(\frac{1}{c}\right)^p}{1 - \frac{1}{c}} \\
&= \frac{\mathcal{Q}}{c^p (c-1) R}.
\end{aligned}$$

□

Using this lemma we can now generalize the step `INITIALIZE` for leaves with multiple particles.

**Lemma 26 (The Truncated Taylor Series for multiple particles).** *For $i = 1, \dots, M$, let $(q_i, z_i)$ represent $M$ particles with the associated Laurent series ${}_i\mathfrak{L}(z) := \frac{q_i}{z-z_i}$. Let $\mathfrak{S}_{R_L}(z_L)$ be a sphere with radius $R_L$ and middle point $z_L$ enclosing all $z_i$ for $i = 1, \dots, M$. Let $z_T \in \mathfrak{S}_{R_L}(z_L)^\complement$ and let $\mathfrak{S}_{R_T}(z_T)$ be a sphere with radius $R_T$ and middle point $z_T$. We consider two cases*

*(1) The two spheres $\mathfrak{S}_{R_L}(z_L)$ and $\mathfrak{S}_{R_T}(z_T)$ are $(c-2)$-well-separated.*

*(2) The sphere $\mathfrak{S}_{R_T}(z_T)$ is $(c-2)$-well-separated to each point $z_i$ for $i = 1, \dots, M$.*

*N.B. Case (1) and (2) are not disjoint. But neither (1) is implies (2), nor vice versa.*
*In the first case, we construct an inexact Taylor series ${}_1\mathfrak{T} := \sum_{k=0}^{\infty} {}_1\mathrm{T}_k (z-z_T)^k$, with expansion point $z_T$, by using the truncated Laurent series $\sum_{k=0}^{p} \frac{\mathrm{L}_k}{(z-z_L)^k}$, from lemma 25.*
*In the second case, we construct an exact Taylor series ${}_2\mathfrak{T} := \sum_{k=0}^{\infty} {}_2\mathrm{T}_k (z-z_T)^k = \sum_{i=1}^{M} {}_i\mathfrak{L}(z) := \frac{q_i}{z-z_i}$, with expansion point $z_T$.*
*Both Taylor series are holomorphic in $\mathfrak{S}_{R_T}(z_T)$ and the coefficients are given by*

$$(1) \qquad {}_1\mathrm{T}_k = \sum_{n=1}^{p} \sum_{i=1}^{M} q_i (z_i - z_L)^{n-1} \binom{k+n-1}{n-1} (z_T - z_L)^{-k-n}, \qquad \textit{for } k \geq 0$$

$$(2) \qquad {}_2\mathrm{T}_k = \sum_{i=1}^{M} \frac{q_i}{(z_T - z_i)^{k+1}}, \qquad \textit{for } k \geq 0,$$

*and obey the following inequality*

$$(1) \qquad |{}_1\mathrm{T}_k| \leq \mathfrak{Q} \frac{1}{R^{k+1}(c-1)^{k+1}},$$

$$(2) \qquad |{}_2\mathrm{T}_k| \leq \mathfrak{Q} \frac{1}{R^{k+1}(c-1)^{k+1}},$$

*with $\mathfrak{Q} = \sum_{i=1}^{M} |q_i|$. Let $z \in \mathfrak{S}_R(z_T)$. When truncating the Taylor series up to $q$-th order, the error will be at most*

$$(1) \qquad \Big| {}_1\mathfrak{T}(z) - \sum_{k=0}^{q} {}_1\mathrm{T}_k (z - z_L)^k \Big| \leq \frac{\mathfrak{Q}}{R(c-2)} \Big(\frac{1}{c-1}\Big)^{q+1},$$

$$(2) \qquad \Big| {}_2\mathfrak{T}(z) - \sum_{k=0}^{q} {}_2\mathrm{T}_k (z - z_L)^k \Big| \leq \frac{\mathfrak{Q}}{R(c-2)} \Big(\frac{1}{c-1}\Big)^{q+1}.$$

*Proof.* We start with case (1) and conclude case (2) by taking the limit $p \to \infty$.
The coefficients and the holomorphic area follow directly from lemma 25 in combination with lemma 19.
Let $R = \max_i |z_L - z_i|$, let $|z_T - z_L| \geq cR$ with $\mathbb{R} \ni c > 2$, then

$$\begin{aligned}
|\mathrm{T}_k| &= \Big| \sum_{n=1}^{p} \sum_{i=1}^{M} q_i (z_i - z_L)^{n-1} \binom{k+n-1}{n-1} (z_T - z_L)^{-k-n} \Big| \\
&\leq \sum_{n=1}^{p} \sum_{i=1}^{M} |q_i| |z_i - z_L|^{n-1} \binom{k+n-1}{n-1} |z_T - z_L|^{-k-n} \\
&\leq \mathfrak{Q} \sum_{n=1}^{p} R^{n-1} \binom{k+n-1}{n-1} (cR)^{-k-n} \\
&\leq \mathfrak{Q} \frac{1}{(cR)^{k+1}} \sum_{n=0}^{\infty} \binom{k+n}{n} \Big(\frac{1}{c}\Big)^n \\
\text{lemma 39} \quad &= \mathfrak{Q} \frac{1}{(cR)^{k+1}} \frac{1}{\big(1 - \frac{1}{c}\big)^{k+1}}
\end{aligned}$$

$$= \mathscr{Q} \frac{1}{R^{k+1}} \frac{1}{(c-1)^{k+1}}.$$

Now we prove the error estimate for the truncated local expansion. Let $\mathscr{Q} = \sum_{i=1}^{M} |q_i|$, let $|z_T - z_L| \geq cR$ with $\mathbb{R} \ni c \geq 2$, and let $z \in \mathfrak{S}_R(z_T)$, then we get

$$\begin{aligned}
\Big|\sum_{k=0}^{\infty} \mathrm{T}_k (z-z_T)^k - \sum_{k=0}^{q} \mathrm{T}_k (z-z_T)^k\Big| &\leq \sum_{k=q+1}^{\infty} |\mathrm{T}_k||z-z_T|^k \\
&\leq \sum_{k=q+1}^{\infty} \frac{\mathscr{Q}}{R^{k+1}(c-1)^{k+1}} R^k \\
&= \frac{\mathscr{Q}}{(c-1)R} \sum_{k=q+1}^{\infty} \Big(\frac{1}{c-1}\Big)^k \\
\text{lemma 39} \qquad &= \frac{\mathscr{Q}}{(c-1)R} \frac{\left(\frac{1}{c-1}\right)^{q+1}}{1-\frac{1}{c-1}} \\
&= \frac{\mathscr{Q}}{R(c-2)} \Big(\frac{1}{c-1}\Big)^{q+1}.
\end{aligned}$$

For the second case, we get the coefficients in case (1) by taking the limit $p \to \infty$, i.e.

$$\begin{aligned}
\mathrm{T}_k &= \lim_{p\to\infty} \sum_{n=1}^{p} \sum_{i=1}^{M} q_i (z_i - z_L)^{n-1} \binom{k+n-1}{n-1} (z_T - z_L)^{-k-n} \\
&= \sum_{i=1}^{M} q_i \frac{1}{(z_T - z_L)^{k+1}} \lim_{p\to\infty} \sum_{n=}^{p-1} \binom{k+n}{n} (z_T - z_L)^{-n} (z_i - z_L)^n \\
\text{lemma 39} \qquad &= \sum_{i=1}^{M} q_i \frac{1}{(z_T - z_L)^{k+1}} \frac{1}{\left(1 - \frac{z_i - z_L}{z_T - z_L}\right)^{k+1}} \\
&= \sum_{i=1}^{M} \frac{q_i}{(z_T - z_i)^{k+1}}.
\end{aligned}$$

For the estimate of the coefficients, we already know $|z_T - z_i| \geq cR$, with $\mathbb{R} \ni c > 2$, therefore

$$\begin{aligned}
|\mathrm{T}_k| &\leq \sum_{i=1}^{M} |q_i||z_T - z_i|^{-k-1} \\
&\leq \mathscr{Q}(cR)^{-k-1}.
\end{aligned}$$

The proof for the error estimate for the truncated local expansion is exactly the same as in case (1). □

We see that in case (1), we can use the previous FMM. Using case (2) of this lemma we will define the new step, `TAYLOR`, in the FMM, which directly computes the truncated Taylor series for $M$ particles, without using `UPCAST`, `CONVERSION` and `DOWNCAST`. Let $U, V$ denote two nodes in the tree, e.g. the tree from the previous FMM. Let $V$ represent $|V|$ particles $(q_1, z_1), \ldots, (q_{|V|}, z_{|V|})$. The Taylor series for the particles in $U$, ${}_U\mathfrak{T}(z) := \sum_{k=0}^{\infty} \mathrm{T}_k (z - z_U)^k$, with expansion point $z_U$, converges for all particles in $U$

and the error of the truncated series is exponentially suppressed by the truncation order $q$, if the requirements of case (2) are fulfilled. The coefficients are given by

$$ {}_U\mathrm{T}_k = \sum_{i=1}^{|V|} \frac{q_i}{(z_U - z_i)^{k+1}}, \qquad \text{for } k \geq 0. $$

Later on we will define a new WSPD, such that the requirements of case (2) will be fulfilled.

## 5.2 Truncated Tree Well-Separated Pair Decomposition

Since our new idea uses a tree which has $\Theta(M)$ particles in each leaf, there will not exist a WSPD in general. By redefining a well-separated pair we are able to construct the Well-Separated Pair Decomposition again. Using this approach we get are finally able to develop the Truncated Tree Fast Multipole Method.

**Definition 27 (Internal Fair Split Tree and Well-Separated Pairs).** This definition is based on [26]. Let $Z = z_1, \dots, z_N \in \mathbb{R}^n$ be $N$ points. Let $Z$ be the root node of a binary tree $\mathscr{T}_M$. The children of a node $U \in \mathscr{T}_M$ are obtained by splitting the bounding box $R(U)$ of $U$ at the longest side in half. The particles corresponding to the "left" part belong to one children, and the particles on the "right" side belong to the other children of $U$. We do not split if the size of the node is less than $2M$. The size of each leaf $U \in \mathscr{T}_M$ is bounded by $M$ and $2M$, i.e. $M \leq |U| < 2M$. A tree obeying these properties is called an *internal fair split tree*. Let $\mathscr{T}_M$ be an internal fair split tree of $Z$. A pair of nodes $(U, V)$ is called truncated tree $s$-well-separated iff

- *Case 1*: $U$ and $V$ are internal nodes and they are $s$-well-separated in the old definition (the requirements of case (1) in lemma 26 are fulfilled), i.e.
  - there exist two $d$-dimensional spheres $\mathfrak{S}_U, \mathfrak{S}_V$ with the same radius $R$,
  - $\mathfrak{S}_U$ contains the bounding box $R(U)$ of $U$,
  - $\mathfrak{S}_V$ contains the bounding box $R(V)$ of $V$, and
  - the distance between $\mathfrak{S}_U, \mathfrak{S}_V$ is at least $sR$.
- *Case 2*: $U$ is an internal node, $V$ is a leaf and $U$ is $s$-well-separated to each point $z_i$ in $V$ (the requirements of case (2) in lemma 26 are fulfilled), i.e.
  - there exist a $d$-dimensional sphere $\mathfrak{S}_U$ with the radius $R$,
  - $\mathfrak{S}_U$ contains the bounding box $R(U)$ of $U$, and
  - the distance between $\mathfrak{S}_U$ and each point in $V$ is at least $sR$.
- *Case 3*: $U$ and $V$ are leaves.

The real number $s$ is called the *separation ratio*.
N.B. This definition is *not* symmetric, i.e. $(U, V)$ could be well-separated, if $(V, U)$ is not well-separated, due to case 2. Case 3 assures that there will always exist a WSPD for an internal fair split tree $\mathscr{T}_M$. We see that this definition is dependent of the underlying tree and it is a generalization of the old definition. For the tree $\mathscr{T} = \mathscr{T}_1$, where each leaf contains exactly one particle, both definitions are equivalent.

The old definition of the WSPD does not have to be changed as long as we use the new definition of well-separated pairs.
Let $\mathscr{T}_M$ be a internal fair split tree with leaf size $M$ and associated WSPD $\mathscr{W}$.
We will denote $\mathscr{W}^{(1)}$ to contain all pairs of nodes which are well-separated, w.r.t. case 1, i.e. $\mathscr{W}^{(1)}$ contains all well-separated inner node pairs. We will denote $\mathscr{W}^{(2)}$ to contain all pairs of nodes which are well-separated, w.r.t. case 2, i.e. $\mathscr{W}^{(2)}$ contains all well-separated (leaf, inner node) pairs. We will denote $\mathscr{W}^{(3)}$ to contain all pairs of nodes which are well-separated, w.r.t. case

3, i.e. $\mathscr{W}^{(3)}$ contains all well-separated (leaf, leaf) pairs. By using similar techniques like Callahan to estimate the number of well-separated pairs, Reif and Tate [26] have shown $|\mathscr{W}^{(1)}| + |\mathscr{W}^{(2)}| + |\mathscr{W}^{(3)}| \in \mathscr{O}(|\mathscr{T}_M|) \in \mathscr{O}\left(\frac{N}{M}\right)$ and this truncated tree with the corresponding WSPD can be computed in subquadratic time. In the section 5.3 we will discuss the consequences of this new definition of well-separated pairs and develop a new FMM algorithm.

## 5.3 Truncated Tree Fast Multipole Method

In this section we will track what is happening for an arbitrary but fixed particle $(z_1, q_1)$ in the different cases in the WSPD. During this observation we will develop an algorithm using the new WSPD. The proof of correctness follows by the construction. Let $Z = z_1, \ldots, z_N \in \mathbb{R}^n$ be the positions of $N$ particles with the charges $q_1, \ldots, q_N$. Let $\mathscr{T}_M$ be an internal fair split tree of $Z$ with the corresponding WSPD $\mathscr{W}$. Due to the properties of the internal fair split tree $\mathscr{T}_M$, there exists for each point $z_i$ exactly one leaf, which contains $z_i$. This leaf will be denoted by $\mathfrak{l}(z_i)$. Due to the properties of the WSPD, there exists for each pair of leaves $(\mathfrak{l}(z_1), \mathfrak{l})$, with $\mathfrak{l}(z_1)$ as the first entry, exactly one pair $(U, V) \in \mathscr{W}$, such that $\mathfrak{l}(z_1) \in U$ and $\mathfrak{l} \in V$, which represents a pairing of $z_1$ to each particle in $\mathfrak{l}$. Therefore, there exists exactly one pairing of $z_1$ to each other particle. This ensures that each pair of particles gets regarded exactly once in the algorithm. N.B. $z_1$ will be paired to itself as well. Now we compute $\sum_{\substack{j \in V \\ j \neq i}} \frac{q_1 q_j}{z_1 - z_j}$ for each $V$ from the pairing. Therefore we consider the three cases in the definition of the well-separated pairs.

- *Case 1*: $U$ and $V$ are internal nodes and they are $s$-well-separated definition. Then we can use `INITIALIZE` and `UPCAST` to compute the Laurent series for $V$. Afterwards we use `CONVERSION` to get the Taylor series and evaluate it at $z = z_1$, or use `DOWNCAST` and postpone the evaluation.
- *Case 2*: $U$ is an internal node, $V$ is a leaf and $U$ is $s$-well-separated to each point $z_i$ in $V$ (the requirements of case (2) in lemma 26 are fulfilled). Therefore we can directly construct the Taylor series for $U$ induced by the particles in $V$. Afterwards evaluate at $z = z_1$, or use `DOWNCAST` and postpone the evaluation.
- *Case 3*: $U$ and $V$ are leaves. Then we will directly compute $\sum_{\substack{j \in V \\ j \neq i}} \frac{q_1 q_j}{z_1 - z_j}$.

These ideas are formalized in algorithm 7.

**Algorithm 7:** Truncated Tree Fast Multipole Method — abstract

```
Input  : An internal fair split tree $\mathscr{T}_M$ with leaf size $M$ and associated WSPD $\mathscr{W}$ for $N$ particles with charges $q_i$ and positions $z_i$
         for $i = 1, \dots, N$.
Output : $(C(q,z) \cdot q)_i = \sum_{\substack{j=1\\ j\neq i}}^{N} \frac{q_i q_j}{z_i - z_j} = {}^{i}\Phi(z_i)$ for $i = 1, \dots, N$.
// INITIALIZE
foreach leaf $U \in \mathscr{T}$ do // Remember that $U$ represents multiple particles.
    Compute the Laurent series ${}_U\mathfrak{L}(z)$, by using lemma 25;
end
// UPCAST
for $l := depth(\mathscr{T})$ to 1 do
    foreach inner node $U \in \mathscr{T}$ with depth $l$ do
        // Let $V, W$ be the children of $U$.
        Compute the Laurent series ${}_U\mathfrak{L}(z)$ of $U$ by combining the Laurent series ${}_V\mathfrak{L}(z)$ and ${}_W\mathfrak{L}(z)$;
    end
end
// CONVERSION
foreach pair $(U,V) \in \mathscr{W}_M^{(1)}$ do
    Convert the Laurent series ${}_V\mathfrak{L}(z)$ of $V$ into a Taylor series ${}_U\mathfrak{T}(z)$ of the paired node $U$;
    Convert the Laurent series ${}_U\mathfrak{L}(z)$ of $U$ into a Taylor series ${}_V\mathfrak{T}(z)$ of the paired node $V$;
    // If there already existed Taylor series before, add both.
end
// TAYLOR
foreach pair $(U,V) \in \mathscr{W}_M^{(2)}$ do
    Directly compute the Taylor series ${}_U\mathfrak{T}(z)$ for $U$ induced by the particles in $V$, by using lemma 26;
    // If there already existed Taylor series before, add both.
end
// DOWNCAST
for $l := 1$ to $depth(\mathscr{T})$ do
    foreach node $U \in \mathscr{T}$ with depth $l$ do
        // Let $V$ be the parent of $U$.
        Compute the new Taylor series ${}_U\mathfrak{T}(z)$ of $U$ by combining the old Taylor series ${}_U\mathfrak{T}(z)$ of $U$ with the Taylor series ${}_V\mathfrak{T}(z)$ of
        the parent node $V$;
    end
end
// EVALUATION1
foreach leaf $U \in \mathscr{T}$ do // Remember that $U$ represents multiple particles.
    // Let $z_i$, for $i = 1, \dots, |U|$, be the positions of the particles in $U$.
    Evaluate the Taylor series ${}_U\mathfrak{T}(z)$ of $U$ at $z = z_i$, for $i = 1, \dots, |U|$ ;
end
// EVALUATION2
foreach pair $(U,V) \in \mathscr{W}_M^{(3)}$ do
    // Let ${}_U z_i, {}_U q_i$, for $i = 1, \dots, |U|$, be the positions and charges of the particles in $U$ and let ${}_V z_j, {}_V q_j$, for
        $j = 1, \dots, |V|$, be the positions and charges of the particles in $V$.
    Directly compute $\sum_{\substack{j\in V\\ j\neq i}} \frac{{}_U q_i \, {}_V q_j}{{}_U z_i - {}_V z_j}$, for $i = 1, \dots, |U|$ ;
end
Add the results from EVALUATION1 and EVALUATION2 and return them;
```

## 5.4 Sensitivity of the Problem

We will consider Trummer's Problem in the real case and, without loss of generality, the absolute value of all coefficients is bounded by 1, which can be achieved by scaling the problem appropriately. Let $(\tilde{q}_i, \tilde{z}_i)$, for $i = 1, \dots, N$, denote $N$ particles with $\tilde{q}_i \in [0,1] \subset \mathbb{R}$ and $\tilde{z}_i \in [0,1] \subset \mathbb{R}$. The first task is to encode the real numbers $\tilde{q}_i$ and $\tilde{z}_i$. Since real numbers have an infinite encoding, we will consider $m$-approximations of the real numbers, i.e. numbers $q_i, z_i$, such that

$$\begin{aligned}|\tilde{q}_i - q_i| &\le 2^{-m_q}\\ |\tilde{z}_i - z_i| &\le 2^{-m_z},\end{aligned}$$

holds. An $m$-approximations is equivalent to using a lattice with distance $2^{-m}$ and rounding each point to the nearest lattice point.
Lemmas 28 and 29 regards the errors which occur by using $m$-approximations and corollary 30 summarizes how to choose $m_q$ and $m_z$ in order to obtain an error of at most $2^{-\mathbb{E}}$.

**Lemma 28 (A Lattice for the Positions).** *Let $(\tilde{q}_i, \tilde{z}_i)$, for $i = 1, \dots, N$, denote $N$ particles with $\tilde{q}_i \in [0,1] \subset \mathbb{R}$ and $\tilde{z}_i \in [0,1] \subset \mathbb{R}$. Let $m_s$ be the separation of the $z_i$, i.e. $|z_i - z_j| \ge 2^{-m_s}$. Let $z_i$ be the $m_z$-approximation, with $m_z \ge m_s + 1$, of $\tilde{z}_i$, then the approximation error will be at most*

$$\sum_{i=1}^{N} \left| \sum_{\substack{j=1\\ j\neq i}}^{N} \frac{\tilde{q}_i \tilde{q}_j}{\tilde{z}_i - \tilde{z}_j} - \sum_{\substack{j=1\\ j\neq i}}^{N} \frac{\tilde{q}_i \tilde{q}_j}{z_i - z_j} \right| \le 2^{2m_s - m_z + 1} N^2.$$

*Proof.* Let $(\tilde{q}_i, \tilde{z}_i)$, for $i = 1, \dots, N$, denote $N$ particles with $\tilde{q}_i \in [0,1] \subset \mathbb{R}$ and $\tilde{z}_i \in [0,1] \subset \mathbb{R}$. Let $m_s$ be the separation of the $z_i$, i.e. $|z_i - z_j| \ge 2^{-m_s}$. Let $z_i$ be the $m_z$-approximation, with $m_z \ge m_s + 1$, of $\tilde{z}_i$, then we get

$$\begin{aligned}
\left| \frac{\tilde{q}_i \tilde{q}_j}{\tilde{z}_i - \tilde{z}_j} - \frac{\tilde{q}_i \tilde{q}_j}{z_i - z_j} \right| &= \left| \frac{\tilde{q}_i \tilde{q}_j}{\tilde{z}_i - \tilde{z}_j} - \frac{\tilde{q}_i \tilde{q}_j}{2^{-m_z}[2^{m_z}\tilde{z}_i] - 2^{-m_z}[2^{m_z}\tilde{z}_j]} \right| \\
&\le |\tilde{q}_i||\tilde{q}_j| \left| \frac{2^{-m_z}[2^{m_z}\tilde{z}_i] - \tilde{z}_i + \tilde{z}_j - 2^{-m_z}[2^{m_z}\tilde{z}]_j}{(\tilde{z}_i - \tilde{z}_j)(2^{-m_z}[2^{m_z}\tilde{z}_i] - 2^{-m_z}[2^{m_z}\tilde{z}_j])} \right| \\
&\le 2^{-m_z} \frac{|[2^{m_z}\tilde{z}_i] - 2^{m_z}\tilde{z}_i| + |2^{m_z}\tilde{z}_j - [2^{m_z}\tilde{z}]_j|}{|\tilde{z}_i - \tilde{z}_j| \cdot |2^{-m_z}[2^{m_z}\tilde{z}_i] - 2^{-m_z}[2^{m_z}\tilde{z}_j]|} \\
\text{lemma 40} \qquad &\le 2^{-m_z} \frac{1}{|\tilde{z}_i - \tilde{z}_j| \cdot (|\tilde{z}_i - \tilde{z}_j| - 2^{-m_z})} \\
&\le 2^{-m_z} \frac{1}{2^{-m_s} \cdot \underbrace{(2^{-m_s} - 2^{-m_z})}_{\ge 2^{-m_s - 1}}} \\
&\le 2^{-m_z + 2m_s + 1}.
\end{aligned}$$

Summation over $i, j = 1, \dots, N$, with $i \neq j$, finishes the proof. □

**Lemma 29 (A Lattice for the Charges).** *Let $(\tilde{q}_i, \tilde{z}_i)$, for $i = 1, \dots, N$, denote $N$ particles with $\tilde{q}_i \in [0,1] \subset \mathbb{R}$ and $\tilde{z}_i \in [0,1] \subset \mathbb{R}$. Let $m_s$ be the separation of the $z_i$, i.e. $|z_i - z_j| \ge 2^{-m_s}$. Let $q_i$ be the $m_q$-approximation, with $m_q \ge m_s + 1$, of $\tilde{q}_i$, then the approximation error will be at most*

$$\sum_{i=1}^{N} \left| \sum_{\substack{j=1\\ j\neq i}}^{N} \frac{\tilde{q}_i \tilde{q}_j}{z_i - z_j} - \sum_{\substack{j=1\\ j\neq i}}^{N} \frac{q_i q_j}{z_i - z_j} \right| \le 2^{m_s - m_q + 3} N^2.$$

*Proof.* Let $(\tilde{q}_i, \tilde{z}_i)$, for $i = 1, \dots, N$, denote $N$ particles with $\tilde{q}_i \in [0,1] \subset \mathbb{R}$ and $\tilde{z}_i \in [0,1] \subset \mathbb{R}$. Let $m_s$ be the separation of the $z_i$, i.e. $|z_i - z_j| \geq 2^{-m_s}$. Let $q_i$ be the $m_q$-approximation, with $m_q \geq m_s + 1$, of $\tilde{q}_i$, then we get

$$\begin{aligned}
& \left|\frac{\tilde{q}_i\tilde{q}_j}{z_i - z_j} - \frac{q_i q_j}{z_i - z_j}\right| = \left|\frac{\tilde{q}_i\tilde{q}_j}{z_i - z_j} - \frac{2^{-m_q}[2^{m_q}\tilde{q}_i]2^{-m_q}[2^{m_q}\tilde{q}_j]}{z_i - z_j}\right| \\
& \qquad = \frac{|\tilde{q}_i\tilde{q}_j - (\tilde{q}_i + 2^{-m_q}[2^{m_q}\tilde{q}_i] - \tilde{q}_i)(\tilde{q}_j + 2^{-m_q}[2^{m_q}\tilde{q}_j] - \tilde{q}_j)|}{|z_i - z_j|} \\
\begin{matrix}\text{lemma 40}\\ \text{expanding}\\ \text{triangle inequality}\end{matrix} & \qquad \leq 2^{m_s+1}(|\tilde{q}_i(2^{-m_q}[2^{m_q}\tilde{q}_j] - \tilde{q}_j)| + |(2^{-m_q}[2^{m_q}\tilde{q}_i] - \tilde{q}_i)\tilde{q}_j| + |(2^{-m_q}[2^{m_q}\tilde{q}_i] - \tilde{q}_i)(2^{-m_q}[2^{m_q}\tilde{q}_j] - \tilde{q}_j)|) \\
& \qquad \leq 2^{m_s+1}(2^{-m_q} + 2^{-m_q} + 2^{-m_q} + 2^{-m_q}) \\
& \qquad \leq 2^{m_s+3-m_q}.
\end{aligned}$$

Summation over $i, j = 1, \dots, N$, with $i \neq j$, finishes the proof. ☐

**Corollary 30.** *Let $(\tilde{q}_i, \tilde{z}_i)$, for $i = 1, \dots, N$, denote $N$ particles with $\tilde{q}_i \in [0,1] \subset \mathbb{R}$ and $\tilde{z}_i \in [0,1] \subset \mathbb{R}$. Let $m_s$ be the separation of the $z_i$, i.e. $|z_i - z_j| \geq 2^{-m_s}$.*
*Let $\tilde{r} \in \mathbb{R}^N$ be the vector of the exact solution of Trummer's Problem, by using the exact values of $\tilde{q}_i$ and $\tilde{z}_i$, for $i = 1, \dots, N$, as input. Let ${}_{m_q,m_z}r \in \mathbb{R}^N$ be the vector of the approximative solution of Trummer's Problem, by using the $m_q$-approximation of $\tilde{q}_i$ and the $m_z$-approximation of $\tilde{z}_i$, for $i = 1, \dots, N$, as input. By choosing $m_q = \mathbb{E} + m_s + 4 + 2\lg N$ and $m_z = \mathbb{E} + 2m_s + 2 + 2\lg N$, the vector ${}_{m_q,m_z}r \in \mathbb{R}^N$ of the approximative solution is an $\mathbb{E}$-approximation of the vector $\tilde{r} \in \mathbb{R}^N$ of the exact solution, w.r.t. the $L_1$ norm, i.e.*

$$\sum_{i=1}^{N} |{}_{m_q,m_z}r_i - \tilde{r}_i| \leq 2^{-\mathbb{E}}.$$

*Therefore $m_q \in \tilde{\Theta}(\mathbb{E} + m_s)$ and $m_z \in \tilde{\Theta}(\mathbb{E} + m_s)$.*

*Proof.* Let $(\tilde{q}_i, \tilde{z}_i)$, for $i = 1, \dots, N$, denote $N$ particles with $\tilde{q}_i \in [0,1] \subset \mathbb{R}$ and $\tilde{z}_i \in [0,1] \subset \mathbb{R}$. Let $m_s$ be the separation of the $z_i$, i.e. $|z_i - z_j| \geq 2^{-m_s}$. Let $\tilde{r} \in \mathbb{R}^N$ be the vector of the exact solution of Trummer's Problem, by using the exact values of $\tilde{q}_i$ and $\tilde{z}_i$, for $i = 1, \dots, N$, as input. Let ${}_{m_q,m_z}r \in \mathbb{R}^N$ be the vector of the approximative solution of Trummer's Problem, by using the $m_q$-approximation of $\tilde{q}_i$ and the $m_z$-approximation of $\tilde{z}_i$, for $i = 1, \dots, N$, as input. Let $m_q = \mathbb{E} + m_s + 4 + 2\lg N$ and $m_z = \mathbb{E} + 2m_s + 2 + 2\lg N$, then we get

$$\begin{aligned}
& \sum_{i=1}^{N} |{}_{m_q,m_z}r_i - \tilde{r}_i| = \sum_{i=1}^{N} \left| \sum_{\substack{j=1\\ j\neq i}}^{N} \frac{q_i q_j}{z_i - z_j} - \sum_{\substack{j=1\\ j\neq i}}^{N} \frac{\tilde{q}_i\tilde{q}_j}{\tilde{z}_i - \tilde{z}_j} \right| \\
& \qquad \leq \sum_{i=1}^{N} \left| \sum_{\substack{j=1\\ j\neq i}}^{N} \frac{q_i q_j}{z_i - z_j} - \sum_{\substack{j=1\\ j\neq i}}^{N} \frac{\tilde{q}_i\tilde{q}_j}{z_i - z_j} \right| + \sum_{i=1}^{N} \left| \sum_{\substack{j=1\\ j\neq i}}^{N} \frac{\tilde{q}_i\tilde{q}_j}{z_i - z_j} - \sum_{\substack{j=1\\ j\neq i}}^{N} \frac{\tilde{q}_i\tilde{q}_j}{\tilde{z}_i - \tilde{z}_j} \right| \\
\text{lemmas 28 and 29} & \qquad \leq 2^{2m_s - m_z + 1}N^2 + 2^{m_s - m_q + 3}N^2 \\
& \qquad \leq 2^{2m_s - (\mathbb{E} + 2m_s + 2 + 2\lg N) + 1 + 2\lg N} + 2^{m_s - (\mathbb{E} + m_s + 4 + 2\lg N) + 3 + 2\lg N} \\
& \qquad = 2^{-\mathbb{E}}.
\end{aligned}$$

☐

Within this section we analyze where errors, e.g. errors due to truncated series, occur and how to suppress them such that the overall error of the result will be at most $2^{-\mathbb{E}}$, for an arbitrary error exponent $\mathbb{E}$.
In the algorithm we will use integer arithmetic and not floating point arithmetic. Therefore addition, subtraction and multiplication are exact and division is inexact. This error can be suppressed by multiplying with a large number before. For generality we will allow real numbers and an input, instead of rational numbers. The output will be rational numbers, which approximate the exact output arbitrarily precise. Let $r_1, \dots, r_N$ be the exact results and let $\hat{r}_1, \dots, \hat{r}_N$ be the approximate output, then we require $|r_1 - \hat{r}_1| + \dots + |r_N - \hat{r}_N| \le 2^{-\mathbb{E}}$ for an arbitrary error exponent $\mathbb{E}$.

### 5.5.1 Truncation Error for Laurent Series

Lemma 31 states the error when using truncated Laurent series instead of Laurent series up to infinity order. Using this lemma we are able to compute an appropriate truncation parameter $p$ for the Laurent series, such that the overall error is bounded by $2^{-\mathbb{E}}$.

**Lemma 31 (The Truncation Error for Laurent Series).**

$$\left| \sum_{\substack{(U,V)\in\mathscr{W}^1\\ z_i\in U}} \sum_{k=0}^{\infty} \frac{{}_V\mathrm{L}_k}{(z_i - z_V)^k} - \sum_{\substack{(U,V)\in\mathscr{W}^1\\ z_i\in U}} \sum_{k=0}^{p} \frac{{}_V\mathrm{L}_k}{(z_i - z_V)^k} \right| \le \sum_{\substack{(U,V)\in\mathscr{W}^1\\ z_i\in U}} \frac{\mathscr{Q}_V}{(c-1)R_V} \big(\frac{1}{c}\big)^p .$$

*Proof.*

$$\begin{aligned} \left| \sum_{\substack{(U,V)\in\mathscr{W}^1\\ z_i\in U}} \sum_{k=0}^{\infty} \frac{{}_V\mathrm{L}_k}{(z_i - z_V)^k} - \sum_{\substack{(U,V)\in\mathscr{W}^1\\ z_i\in U}} \sum_{k=0}^{p} \frac{{}_V\mathrm{L}_k}{(z_i - z_V)^k} \right| &\le \sum_{\substack{(U,V)\in\mathscr{W}^1\\ z_i\in U}} \left| \sum_{k=0}^{\infty} \frac{{}_V\mathrm{L}_k}{(z_i - z_V)^k} - \sum_{k=0}^{p} \frac{{}_V\mathrm{L}_k}{(z_i - z_V)^k} \right| \\ \text{lemma 25} \quad &\le \sum_{\substack{(U,V)\in\mathscr{W}^1\\ z_i\in U}} \frac{\mathscr{Q}_V}{c^p(c-1)R_V} . \end{aligned}$$

□

Therefore we see $p \in \tilde{\Theta}(\mathbb{E})$.

### 5.5.2 Truncation Error for Taylor Series

This section is similar to the previous, here we consider truncated Taylor series instead of Laurent series. Lemma 32 states the error when using truncated Taylor series instead of Taylor series up to infinity order. Using this lemma we are able to compute an appropriate truncation parameter $q$ for the Taylor series, such that the overall error is bounded by $2^{-\mathbb{E}}$.

**Lemma 32 (The Truncation Error for Taylor Series).**

$$\left| \sum_{\substack{(U,V)\in\mathscr{W}^1\\ z_i\in U}} \sum_{k=0}^{\infty} {}_U\mathrm{T}_k (z_i - z_U)^k - \sum_{\substack{(U,V)\in\mathscr{W}^1\\ z_i\in U}} \sum_{k=0}^{q} {}_U\mathrm{T}_k (z_i - z_U)^k \right| \le \sum_{\substack{(U,V)\in\mathscr{W}^1\\ z_i\in U}} \frac{\mathscr{Q}_U}{R_U(c-2)} \big(\frac{1}{c-1}\big)^{q+1} .$$

*Proof.*

$$\left|\sum_{\substack{(U,V)\in\mathscr{W}^1\\ z_i\in U}}\sum_{k=0}^{\infty}{}_U\mathrm{T}_k(z_i-z_U)^k-\sum_{\substack{(U,V)\in\mathscr{W}^1\\ z_i\in U}}\sum_{k=0}^{q}{}_U\mathrm{T}_k(z_i-z_U)^k\right|\leq\sum_{\substack{(U,V)\in\mathscr{W}^1\\ z_i\in U}}\left|\sum_{k=0}^{\infty}{}_U\mathrm{T}_k(z_i-z_U)^k-\sum_{k=0}^{q}{}_U\mathrm{T}_k(z_i-z_U)^k\right|$$

$$\text{lemma 26}\quad\leq\sum_{\substack{(U,V)\in\mathscr{W}^1\\ z_i\in U}}\frac{\mathscr{Q}_U}{R_U(c-2)}\Big(\frac{1}{c-1}\Big)^{q+1}. \qquad\square$$

Therefore we see $q\in\tilde{\Theta}(\mathbb{E})$.

## 5.5.3 Rounding Error for CONVERSION

During the `CONVERSION` step, cf. algorithm 6 line 9ff., we need a division, which could result in rounding errors. These rounding errors will be estimated in lemma 34.

**Lemma 33.**

$$|{}_U\mathrm{T}_k-{}^{\mathit{CONV}}_{U}\mathrm{T}_k|\leq 2^{-m_{\mathit{CONV}}}\mathscr{Q}_V R_V^p 6^p,$$

*whereas* ${}^{\mathit{CONV}}_{U}\mathrm{T}_k$ *is obtained by* CONVERSION*, cf. algorithm 6 line 9ff., including an inexact division.*

*Proof.*

$$\begin{aligned}
&|{}_U\mathrm{T}_k-{}^{\mathtt{CONV}}_{U}\mathrm{T}_k|\\
&=\left|\frac{1}{k!(p-1)!}\sum_{n=1}^{p}\frac{(p-1)!\,{}_V\mathrm{L}_n}{(n-1)!}\frac{(k+n-1)!}{(z_u-z_v)^{k+n}}-\frac{2^{-m_{\mathtt{CONV}}}}{k!(p-1)!}\sum_{n=1}^{p}\frac{(p-1)!\,{}_V\mathrm{L}_n}{(n-1)!}\left[\frac{2^{m_{\mathtt{CONV}}}}{(z_u-z_v)^{k+n}}\right](k+n-1)!\right|\\
&\leq 2^{-m_{\mathtt{CONV}}}\frac{1}{k!}\sum_{n=1}^{p}\frac{(k+n-1)!|{}_V\mathrm{L}_n|}{(n-1)!}\left|\frac{2^{m_{\mathtt{CONV}}}}{(z_u-z_v)^{k+n}}-\left[\frac{2^{m_{\mathtt{CONV}}}}{(z_u-z_v)^{k+n}}\right]\right|\\
&\leq 2^{-m_{\mathtt{CONV}}}\sum_{n=1}^{p}\mathscr{Q}_V R_V^{n-1}\binom{k+n-1}{n-1}\\
&\leq 2^{-m_{\mathtt{CONV}}}\mathscr{Q}_V\frac{1-R_V^p}{1-R_V}\binom{2p}{p}\\
&\leq 2^{-m_{\mathtt{CONV}}}\mathscr{Q}_V R_V^p 6^p. \qquad\square
\end{aligned}$$

**Lemma 34 (The Rounding Error for `CONVERSION`).**

$$\left|\sum_{\substack{(U,V)\in\mathscr{W}^1\\ z_i\in U}}\sum_{k=0}^{q}{}_U\mathrm{T}_k(z_i-z_U)^k-\sum_{\substack{(U,V)\in\mathscr{W}^1\\ z_i\in U}}\sum_{k=0}^{q}{}^{\mathit{CONV}}_{U}\mathrm{T}_k(z_i-z_U)^k\right|\leq 2^{-m_{\mathit{CONV}}}\sum_{\substack{(U,V)\in\mathscr{W}^1\\ z_i\in U}}\mathscr{Q}_V R_U^{q+1}R_V^p 6^p.$$

*Proof.*

$$\left|\sum_{\substack{(U,V)\in\mathscr{W}^1\\ z_i\in U}}\sum_{k=0}^{q}{}_U\mathrm{T}_k(z_i-z_U)^k-\sum_{\substack{(U,V)\in\mathscr{W}^1\\ z_i\in U}}\sum_{k=0}^{q}{}^{\mathtt{CONV}}_{U}\mathrm{T}_k(z_i-z_U)^k\right|\le\sum_{\substack{(U,V)\in\mathscr{W}^1\\ z_i\in U}}\sum_{k=0}^{q}|z_i-z_U|^k|{}_U\mathrm{T}_k-{}^{\mathtt{CONV}}_{U}\mathrm{T}_k|$$

$$\text{lemma 33}\quad\le\sum_{\substack{(U,V)\in\mathscr{W}^1\\ z_i\in U}}\sum_{k=0}^{q}R_U^k2^{-m_{\mathtt{CONV}}}\mathscr{Q}_VR_V^p6^p$$

$$\text{lemma 39}\quad\le 2^{-m_{\mathtt{CONV}}}\sum_{\substack{(U,V)\in\mathscr{W}^1\\ z_i\in U}}\mathscr{Q}_VR_U^{q+1}R_V^p6^p.\qquad\square$$

Therefore $m_{\mathtt{CONV}}\in\tilde{\Theta}(\mathbb{E}m_z)\in\tilde{\Theta}(\mathbb{E}^2+\mathbb{E}m_s)$.

## 5.6 Runtime of the Main Steps

In this section we analyze each step of the Truncated Tree Fast Multipole Method sections 5.6.1 to 5.6.6. In section 5.7 we combine all previous results in order to give an estimation of the overall runtime, which is one of our main results.
In the following analysis let $\mu(k)$ denote the runtime of the multiplication of two integers with $k$ bits encoding. Also we scale all charges and positions by $m_q$, respectively $m_z$, and round them afterwards to integers. These values will be the input of our algorithm. Then $2^{m_z-m_s}\le|z_i|\le 2^{m_z}$, for all $i=1,\dots,N$ and $|q_i|\le 2^{m_q}$ holds.

### 5.6.1 INITIALIZE

```
  // INITIALIZE
1 foreach leaf $U\in\mathscr{T}$ do // Remember that $U$ represents multiple particles.
      // Compute the Laurent series ${}_U\mathfrak{L}(z)$, by using lemma 25
2     ${}_U\hat{\mathrm{L}}_k := (p-1)!\frac{{}_U\mathrm{L}_k}{(k-1)!} = (p-1)!\sum_{i:z_i\in U}q_i(z_i-z_U)^{k-1}$,    for $k\ge 1$;
3     Define the Laurent coefficient integer ${}^{(p,P)}_{U}\mathscr{L}$, with $p$ blocks of size $P$, whereas the $k$-th block gets initialized by the value
      $\hat{\mathrm{L}}_k$;
4 end
```

We see that for each leaf, this problem is equivalent to the multiplication of a transposed Vandermonde Matrix, whereas the generating vector of the Vandermonde Matrix is given by $s_i=z_i-Z_U$, with the vector $q$. As shown by Tsigaridas and Pan [24] this problem can solved in $\tilde{\mathscr{O}}(\mathbb{E}^3)$, because $p\in\mathscr{O}(\mathbb{E})$. Therefore the overall runtime is given by $\tilde{\mathscr{O}}(|\mathscr{T}_M|\mathbb{E}^3)$.

### 5.6.2 UPCAST

**Lemma 35 (Runtime of UPCAST).** *The runtime of UPCAST is $\tilde{\mathscr{O}}(|\mathscr{T}_M|(\mathbb{E}^2m_s+\mathbb{E}^3))$ and all divisions are exact.*

*Proof.* Since $p\ge i$ the divisions in line 5 is exact.
For the division in line 10 we recall section 4.5.1 with the equation

$$\frac{{}_U\mathrm{L}_k}{(k-1)!}=\sum_{n=1}^{k}\frac{(z_V-z_U)^{k-n}}{(k-n)!}\frac{{}_V\mathrm{L}_n}{(n-1)!}+\sum_{n=1}^{k}\frac{(z_W-z_U)^{k-n}}{(k-n)!}\frac{{}_W\mathrm{L}_n}{(n-1)!}.$$

```
   // UPCAST
1  for $l := depth(\mathscr{T})$ to 1 do
2      foreach inner node $U \in \mathscr{T}$ with depth $l$ do
           // Let $V, W$ be the two children of $U$.
           // Let ${}^{(p,P)}_{(U,V)}\mathscr{D}, {}^{(p,P)}_{(U,W)}\mathscr{D}$ be the distance integers with $p$ blocks of size $P$.
3          $(z_V - z_U)^{1-1}\frac{(p-1)!}{(1-1)!} = (p-1)!$;
4          for $i := 1$ to $p-1$ do
5              $(z_V - z_U)^{i}\frac{(p-1)!}{i!} = \underbrace{(z_V - z_U)^{i-1}\frac{(p-1)!}{(i-1)!}}_{\text{known from previous iteration}} / i \cdot (z_V - z_U)$;
6          end
7          Encode the scaled distances $(z_V - z_U)^{i-1}\frac{(p-1)!}{(i-1)!}$, for $i = 1, \ldots, p$, in the $i$-th block of ${}^{(p,P)}_{(U,V)}\mathscr{D}$;
8          Redo the above steps with interchanged roles of $V$ and $W$;
           // Let ${}^{(p,P)}_{\phantom{(p,}U}\mathscr{L}$ be the Laurent coefficient integer with $p$ blocks of size $P$ of $U$.
9          ${}^{(p,P)}_{\phantom{(p,}U}\mathscr{L}' := {}^{(p,P)}_{(U,V)}\mathscr{D} \cdot {}^{(p,P)}_{\phantom{(p,}V}\mathscr{L} + {}^{(p,P)}_{(U,W)}\mathscr{D} \cdot {}^{(p,P)}_{\phantom{(p,}W}\mathscr{L}$ truncated to the last $p$ blocks;
10         ${}^{(p,P)}_{\phantom{(p,}U}\mathscr{L} := {}^{(p,P)}_{\phantom{(p,}U}\mathscr{L}'/(p-1)!$ // now, ${}_U\hat{\mathrm{L}}_i$ is in the $i$-th block.
11     end
12 end
```

By multiplying with $\big((p-1)!\big)^2$ on both sides, we see that $\big((p-1)!\big)^2 \frac{{}_U\mathrm{L}_k}{(k-1)!}$ is in the $k$-th block of ${}^{(p,P)}_{\phantom{(p,}U}\mathscr{L}'$, cf. line 5. Therefore we can divide by $(p-1)!$ exactly.

For line 3 we assume the integer $(p-1)!$ is precomputed. In line 5, the runtime for each multiplication and division is $\mathscr{O}(\mu(p(\lg p + m_z)))$, cf. lemma 41. Hence the overall runtime for lines 3 to 8 is $\mathscr{O}(p\mu(p(\lg p + m_z)))$.

In line 9 we can neglect the runtime of the addition and the truncation, because the time for the multiplication is larger and is given by $\mathscr{O}(\mu(pP))$. The division in line 10 has the same runtime.

Iteration over each node yields the complete runtime $\mathscr{O}(|\mathscr{T}_M|\mu(pP + p^2(\lg p + m_z)))$.

By estimating the size of the blocks, we get

$$\begin{aligned} \lg\left|\frac{((p-1)!)^2\, {}_U\mathrm{L}_k}{(k-1)!}\right| &\leq \lg \mathscr{Q}_U R_U^{k-1} p^{2p} && \text{lemma 25} \\ &\leq m_q \lg N + pm_z + 2p\lg p. \end{aligned}$$

Therefore we get

$$\begin{aligned} P &\leq m_q \lg N + pm_z + 2p\lg p \\ &\in \tilde{\Theta}(\mathbb{E}m_s + \mathbb{E}^2). \end{aligned}$$

Therefore the long integers, into which the coefficients get embedded, have $p$ blocks of size $\lceil m_q \lg N + pm_z + 2p\lg p\rceil$. Hence the integers have size $p\lceil m_q \lg N + pm_z + 2p\lg p\rceil \in \tilde{\Theta}(\mathbb{E}^2 m_s + \mathbb{E}^3)$. Therefore the overall runtime is $\tilde{\mathscr{O}}(|\mathscr{T}_M|(\mathbb{E}^2 m_s + \mathbb{E}^3))$.□

### 5.6.3 CONVERSION

**Lemma 36 (Runtime of CONVERSION).** *The runtime of CONVERSION is $\tilde{\mathscr{O}}(|\mathscr{W}_M^1|(\mathbb{E}^2 m_s + \mathbb{E}^3))$ and the division in line 11 is exact.*

```
   // CONVERSION
1  foreach $(U,V) \in \mathscr{W}^{(1)}$ do
       // Let ${}^{(q+p,Q)}_{(V,U)}\mathscr{R}$ be reciprocal integers with $q+p$ blocks of size $Q$.
2      $(z_U - z_V)^{1-1} := 1$;
3      Compute $\left[\frac{2^{m_{\mathtt{CONV}}}}{(z_U-z_V)^{1-1}}\right](1-1)!$;
4      for $i := 1$ to $p+q-1$ do
5          $(z_U - z_V)^i := \underbrace{(z_U - z_V)^{i-1}}_{\text{known from previous iteration}} \cdot (z_U - z_V)$ ;
6          Compute $\left[\frac{2^{m_{\mathtt{CONV}}}}{(z_U-z_V)^{i}}\right] i!$
7      end
8      Encode the scaled reciprocal distances $\left[\frac{2^{m_{\mathtt{CONV}}}}{(z_U-z_V)^{i-1}}\right](i-1)!$, for $i = 1,\dots,p+q$, in the $i$-th block of ${}^{(p+q,Q)}_{(V,U)}\mathscr{R}$.;
9      Expand each block of ${}^{(p,P)}_{V}\mathscr{L}$ to size $Q$ and rename it to ${}^{(p,Q)}_{V}\mathscr{L}$;
10     ${}^{(q+1,Q)}_{U}\mathscr{T}^{\text{conv}} := {}^{(p+q,Q)}_{(U,V)}\mathscr{R} \cdot {}^{(p,Q)}_{V}\mathscr{L}$ truncated to the blocks from $p$ to $p+q$;
11     ${}^{(q+1,Q)}_{U}\mathscr{T}^{\text{conv}} := {}^{(q+1,Q)}_{U}\mathscr{T}^{\text{conv}}/(p-1)!$;
12     ${}^{(q+1,Q)}_{U}\mathscr{T} := {}^{(q+1,Q)}_{U}\mathscr{T} + {}^{(q+1,Q)}_{U}\mathscr{T}^{\text{conv}}$ // now, $2^{m_{\mathtt{CONV}}} i!\,{}_U\mathrm{T}_i$ is in the $(q-i)$-th block.
13     Redo these steps with interchanged roles of $U$ and $V$;
14 end
```

*Proof.* For the division in line 11 we recall section 4.5.2 with the equation

$$k!\,{}_U\mathrm{T}_k = \sum_{n=1}^{p} \frac{{}_V\mathrm{L}_n}{(n-1)!}\frac{(k+n-1)!}{(z_u - z_v)^{k+n}}.$$

By multiplying with $(p-1)!2^{m_{\mathtt{CONV}}}$ on both sides, we see that $(p-1)!2^{m_{\mathtt{CONV}}}k!\,{}_U\mathrm{T}_k$ is in the $(q-k)$-th block of ${}^{(q+1,Q)}_{U}\mathscr{T}^{\text{conv}}$, cf. line 10. Therefore we can divide by $(p-1)!$ exactly.
In line 5, the runtime for each multiplication is $\mathcal{O}(\mu(pm_z))$, cf. lemma 41. In line 6, the runtime for the division is $\mathcal{O}(\mu(m_{\mathtt{CONV}} + pm_z))$ and for the multiplication we get $\mathcal{O}(\mu(m_{\mathtt{CONV}} + p\lg p))$, cf. lemma 41. Hence the overall runtime for lines 2 to 9 is $\mathcal{O}((p+q)\mu(m_{\mathtt{CONV}} + p(\lg p + m_z)))$.
The runtime of line 12 is dominated by the runtime of line 10, which is given by $\mathcal{O}(\mu((p+q)Q))$. The division in line 11 has the runtime $\mathcal{O}(\mu(qQ + p\lg p))$.
Iteration over each type-1 well-separated pair yields the complete runtime $\mathcal{O}(|\mathscr{W}_M^1|\mu((p+q)(Q + m_{\mathtt{CONV}} + p(\lg p + m_z))))$.
By estimating the size of the blocks, we get

$$\begin{aligned}\lg|2^{m_{\mathtt{CONV}}}k!\,{}_U\mathrm{T}_k| &\le \lg 2^{m_{\mathtt{CONV}}}\mathscr{Q}_U p^p \quad \text{lemma 26}\\ &\le m_{\mathtt{CONV}} + m_q \lg N + p\lg p.\end{aligned}$$

Therefore we get

$$\begin{aligned}Q &\le m_{\mathtt{CONV}} + m_q \lg N + p\lg p\\ &\in \tilde{\Theta}(\mathbb{E}m_s + \mathbb{E}^2).\end{aligned}$$

Therefore the long integers, into which the coefficients get embedded, have $p+q$ blocks of size $\lceil m_{\mathtt{CONV}} + m_q \lg N + p\lg p\rceil$. Hence the integers have size $(p+q)\lceil m_{\mathtt{CONV}} + m_q \lg N + p\lg p\rceil \in \tilde{\Theta}(\mathbb{E}^2 m_s + \mathbb{E}^3)$. Therefore the overall runtime is $\mathcal{O}(|\mathscr{W}_M^1|(\mathbb{E}^2 m_s + \mathbb{E}^3))$.□

### 5.6.4 TAYLOR

Lemma 26 shows how to compute the coefficients of the Taylor series. This can be done similar as in section 5.6.1 — one sees that the second case can be solved by multiplying a transposed Vandermonde matrix with a vector and for the first case we need to multiply a transposed Vandermonde matrix with a vector and afterwards a transposed Vandermonde matrix with the solution. Therefore this problem can be solved again in $\tilde{\mathcal{O}}(\mathbb{E}^3)$, cf. [24], for each $\mathscr{W}^{(2)}$-pair. Therefore the overall runtime is given by $\tilde{\mathcal{O}}(|\mathscr{W}^{(2)}|\mathbb{E}^3)$.

### 5.6.5 DOWNCAST

```
   // DOWNCAST
1  for $l := 1$ to $depth(\mathscr{T})$ do
2      foreach node $U \in \mathscr{T}$ with depth $l$ do
           // Let $V$ be the parent of $U$.
           // Let ${}^{(q+1,Q)}_{(V,U)}\mathscr{D}$ be the distance integer with $q+1$ blocks of size $Q$. One could reuse the
              distance integers from UPCAST.
3          $(z_U - z_V)^{1-1}\frac{(q-1)!}{(1-1)!} = (q-1)!$;
4          for $i := 1$ to $q$ do
5              $(z_U - z_V)^{i}\frac{(q-1)!}{i!} = \underbrace{(z_U - z_V)^{i-1}\frac{(q-1)!}{(i-1)!}}_{\text{known from previous iteration}} / i \cdot (z_U - z_V)$;
6          end
7          Encode $\frac{(z_U-z_V)^{i-1}}{(i-1)!}$ in the $i$-th block of ${}^{(q+1,Q)}_{(V,U)}\mathscr{D}$, for $i = 1, \dots, q+1$ ;
8          ${}^{(q+1,Q)}_{U}\mathscr{T}^{\text{down}} := {}^{(q+1,Q)}_{V}\mathscr{T} \cdot {}^{(q+1,Q)}_{(V,U)}\mathscr{D}$ truncated to the last $p$ blocks;
9          ${}^{(q+1,Q)}_{U}\mathscr{T}^{\text{down}} := {}^{(q+1,Q)}_{U}\mathscr{T}^{\text{down}}/(q-1)!$;
10         ${}^{(q+1,Q)}_{U}\mathscr{T} := {}^{(q+1,Q)}_{U}\mathscr{T} + {}^{(q+1,Q)}_{U}\mathscr{T}^{\text{down}}$ // now, $2^{m_{\text{CONV}}} i!\, {}_U\mathrm{T}_i$ is in the $(q-i)$-th block.
11     end
12 end
```

**Lemma 37 (Runtime of DOWNCAST).** *The runtime of DOWNCAST is $\tilde{\mathcal{O}}(|\mathscr{T}_M|(\mathbb{E}^2 m_s + \mathbb{E}^3))$ and all division are exact.*

*Proof.* Since $q \geq i$ the divisions in line 5 is exact.
For the division in line 9 we recall section 4.5.3 with the equation

$$k!\,{}_U\mathrm{T}_k = \sum_{n=k}^{q} n!\,{}_V\mathrm{T}_n \frac{(z_U - z_V)^{n-k}}{(n-k)!}.$$

By multiplying with $(q-1)!2^{m_{\text{CONV}}}$ on both sides, we see that $(q-1)!2^{m_{\text{CONV}}}k!\,{}_U\mathrm{T}_k$ is in the $(q-k)$-th block of ${}^{(q+1,Q)}_{U}\mathscr{T}^{\text{down}}$, cf. line 8. Therefore we can divide by $(q-1)!$ exactly.
For line 3 we assume the integer $(q-1)!$ is precomputed. In line 5, the runtime for each multiplication and division is $\mathcal{O}(\mu(q(\lg q + m_z)))$, cf. lemma 41. Hence the overall runtime for lines 3 to 7 is $\mathcal{O}(q\mu(q(\lg q + m_z)))$.
In line 10 we can neglect the runtime of the addition and the truncation in line 8, because the time for the multiplication is larger and is given by $\mathcal{O}(\mu((p+q)Q))$. The division in line 11 has the runtime $\mathcal{O}(\mu(qQ + q\lg q))$.
Iteration over each node yields the complete runtime $\mathcal{O}(|\mathscr{T}_M|\mu((p+q)(Q + q(\lg q + m_z)))) \in \mathcal{O}(|\mathscr{T}_M|(\mathbb{E}^2 m_s + \mathbb{E}^3))$, cf. the proof of lemma 36. ☐

### 5.6.6 `EVALUATION1` and `EVALUATION2`

`EVALUATION2` is Trummer's Problem again. In order to implement this, we suggest an recursive implementation. In order to show the runtime, we simply use Tsigaridas and Pan's results [24] again. The cubic runtime in $\mathbb{E}$ is sufficient for our required overall runtime.
`EVALUATION1` is the multipoint evaluation, i.e. multiplication of a Vandermonde matrix with a vector again. Hence the runtime is cubic in $\mathbb{E}$ again. By iterating over all leaves we get the combined runtime of `EVALUATION1` and `EVALUATION2`, which is given by $\tilde{\mathscr{O}}(|\mathscr{T}_M|\mathbb{E}^3)$.

## 5.7 Fast Multipole Method linear in $N$ and quadratic in $\mathbb{E}$

Combining the sections about the runtime analysis sections 5.6.1 to 5.6.6 with the sections about the errors analysis sections 5.4 and 5.5.1 to 5.5.3, we see that algorithm 7 is linear in the number of nodes in the Tree, i.e. $\frac{N}{\mathbb{E}}$ and cubic in the error exponent $\mathbb{E}$, resp. in $\mathbb{E}^3 + \mathbb{E}^2 \cdot m_s$, with respect to *bit cost*, therefore complete problem can be solved in $\tilde{\mathscr{O}}_2(N \cdot (\mathbb{E}^2 + \mathbb{E} \cdot m_s)$. Compared to the lower bound $\mathscr{O}_2(N \cdot (\mathbb{E} + m_s)$, cf. section 2.2, this algorithm is nearly optimal up to a factor of $\mathbb{E}$ and polylogarithmic terms. What makes it labor-intensive to implement this algorithm is the need to implement the fast computation with structured matrices, i.e. Vandermonde and transposed Vandermonde matrices, whereas the latter requires computations with a Cauchy matrices. Therefore a recursive approach would be very interesting and promising.

Chapter 6

# Implementation

The big problem while implementing the Fast Multipole Method was stated by Eric Darve [6]

> However, implementation of the FMM proved to be a rather difficult task in part because of its complexity (many lines in a complex and long code) and because of the need to optimize all the steps of the FMM.

Therefore the big challenge for the implementation is to reduce the implementation complexity on all costs without losing performance.
In section 6.1 we present the challenges to implement the Fast Multipole Method and in section 6.2 we describe how to implement the two modules, the Fast Multipole Method and the Well-Separated Pair Decomposition, as well as some helper functionalities which are needed during the implementation. Section 6.3 states some more interesting properties of our implementation, e.g. we have solved an even larger family of problems and not only Trummer's Problem and section 6.4 completes this chapter with some implementation statistics about the effort which is needed for the implementation and the resulting implementation complexity, measured using the cyclomatic complexity.

## 6.1 Challenges to implement the Fast Multipole Method

To estimate the time we need to implement the software we use the COCOMO model developed by Barry W. Boehm [4]. The needed effort $E$, i.e. the number of month one person needs to implement the algorithm, is given by

$$E = a \cdot \left(\frac{\text{LOC}}{1000}\right)^b,$$

whereas LOC is the number of lines of delivered code, i.e. number of lines of code, without comments, without blank lines, without tests, without benchmarks, without makefiles, etc. The parameters $a$ and $b$ are fitted by Boehm from historical projects. By classifying the projects into 'easy', 'intermediate' and 'complex' he got the following values

- $a = 2.4, b = 1.05$ for an 'easy' project,
- $a = 3.0, b = 1.12$ for an 'intermediate' project, and
- $a = 3.6, b = 1.20$ for a 'complex' project.

The model function is more interesting than the explicit values of $a$ and $b$, namely the non linear behavior $\left(\frac{\text{LOC}}{1000}\right)^b$. This means that we should try to split the algorithm in multiple independent modules. The steps of the Fast Multipole Method are not independent of each other, therefore the whole FMM should be contained in one module. But the Well-Separated Pair Decomposition is completely independent of the FMM and solves an interesting problem on its own. Therefore the Well-Separated Pair Decomposition forms the second module. This separation into two modules should reduce the overall effort to implement everything.
The biggest "problem" of all existing approaches are the two different views of the series, namely the series in position space, the "normal" view, and the coefficients in the momentum space, the view after the Fourier transform. This duality is called Pontryagin duality, cf. [28]. Using these two views one is able to speed up the algorithm, because the multiplication of polynomials is way

faster by using the latter view., after the Fourier transform. These two views induce a lot of additional complexity; but on the other hand, we should use the Fast Fourier Transform in order to speed up the computations. The big challenge is now how to unify both views such that the implementation complexity is reduced, but the FFT is still applicable to speed up the multiplication of polynomials. This view and the corresponding data structure should be used throughout the complete algorithm to further simplify the implementation.
In order to support more software and other libraries, the next challenge is dedicated the user-friendliness. Here we want that on the one hand the library should be easy to build and to integrate in other software and on the other hand the usage should be easy.
Summarized we got the following challenges for our implementation

- The Library has to be fast and satisfy the claimed runtime complexity.
- User-friendliness: easy building, integration and usage of the library.
- Minimized implementation complexity
  - Split the solution into two modules the *Well-Separated Pair Decomposition* and the *Fast Multipole Method*.
  - Unify the two views onto the series and the corresponding unified data structure should be used throughout the complete algorithm to simplify the implementation.
  - The source code should be easily maintainable, i.e. a good coding style and a clean code.

The requirement for maintainability and readability of the code, is obvious and should not be emphasized here. To fulfill this task, we comply with 'Google C++ Style Guide' [11], remove redundant code by refactoring, do not nest "if", "for", "switch" control structures too deeply, etc.
The other requirements from above will be discussed in the next sections. The runtime of the implementation will be discussed in section 7.2.

### 6.1.1 User-friendliness: easy Building, Integration and Usage of the Library

In order to build the library we have chosen CMake (version >= 3.5) as our build tool, which simplifies the build process for multiple platforms. We have build and tested the library with macOS and ubuntu. The library needs the C++11 standard, but no C++14 or C++17 features. These features sometimes result in problems when compiling with windows, therefore we have omitted them.
The only requirement is GMP [12]. This library supports computations with arbitrary long integers. The integer multiplication in GMP uses a hybrid ansatz, i.e. the called algorithm depends on the length of the integers. For small integers the school book method with the bit-complexity $\mathcal{O}_2(n^2)$, is used. For medium sized inputs the Karatsuba method with the bit complexity $\mathcal{O}_2(n^{\log_2 3})$ is used, for larger sized inputs one of the Toom-Cook methods, which run in $\mathcal{O}_2(n^{1+\varepsilon})$, are used, namely in increasing order: Toom-3, Toom-4, Toom-6.5 and Toom-8.5. For very large inputs GMP uses the Schönhage–Strassen algorithm, with $\mathcal{O}_2(n \cdot \log n \cdot \log\log n$ runtime. Therefore we've found a library which implements the required fast integer multiplication, such that our estimated runtime can be achieved. Since GMP is the only requirement and available on all systems, e.g. by using their packet managers: apt-get on ubuntu, brew on macOS, choco or vcpkg on Windows, etc. this simplifies the building and integration of the library drastically.
To simplify the integration the complete library is header only, therefore one gets no problems while linking.
Due to heavy use of templates one is able to use a lot of different types as input, e.g. integers and doubles can be used for the positions and charges. Using GMP values, e.g. mpz_t, mpq_t and mpf_t, is also possible. Therefore it should be quit easy to integrate this library into an existing application, because there is no conversion of the data types needed.

### 6.1.2 Unified Data Structure throughout the complete Algorithm

As described before the biggest "problem" of all existing approaches are the two different views of the series. Therefore we have chosen to unify both views using the Kronecker Embedding, cf. example 22. Throughout the whole implementation we are able to use only this view — the Coefficient Integers. The problem of the embedding are the different signs of the coefficients, because we are only able to encode them properly when all coefficients have the same sign. Without considering the signs the multiplication would produce wrong results.
Using the following observation we able to work with different signs of the coefficients.

*Observation 38.* Let $x, y$ be arbitrary real valued functions and $x_+, x_-, y_+, y_-$ positive real valued functions, such that $x = x_+ - x_-, y = y_+ - y_-$. Then we get the following

$$\begin{aligned} x \cdot y &= (x_+ - x_-) \cdot (y_+ - y_-) \\ &= x_+ \cdot y_+ + x_- \cdot y_- - x_+ \cdot y_- - x_- \cdot y_+ \\ &= (x_+ \cdot y_+ + x_- \cdot y_-) - (x_+ \cdot y_- + x_- \cdot y_+) \\ &= xy_+ - xy_-. \end{aligned}$$

Therefore we can simply multiply real valued functions just by multiplication and addition of positive functions. We say two positive real valued functions $x_+, x_- \geq 0$ represent the function $x$ iff $x = x_+ - x_-$. We say $x_+, x_- \geq 0$ are a normalized representation of $x$ iff for all values $k$ in the domain not both functions $x_+$ and $x_-$ are strictly greater than zero at $k$, i.e. $x_+(k) > 0 \implies x_-(k) = 0$ and $x_-(k) > 0 \implies x_+(k) = 0$, which is equivalent to $x_+ \cdot x_- = 0$, whereas $\cdot$ is the pointwise multiplication. By choosing a finite domain $x, y$ become vectors which correspond to our large integers. Hence we distinguish between the positive and negative coefficients and are able to safely multiply them using the previous method. To implement this view we use the four large integers for each node in the fair split tree

- positive Taylor coefficients,
- negative Taylor coefficients,
- positive Laurent coefficients,
- negative Laurent coefficients.

We do not need any more additional data structures. This simplifies the implementation and reduces its complexity. But the biggest advantage by using this approach is we do not have to implement an integer FFT. For small instances the FFT could not be the best choice, because of the constants, and a naive method is faster. Therefore a hybrid algorithm would be better, which induces even more implementation complexity. Using our approach we simply multiply the long integers and GMP takes care of the hybridisation. Therefore the cumbersome part is outsourced to GMP, which is a highly optimized library [12].

> The speed of GMP is achieved by using fullwords as the basic arithmetic type, by using sophisticated algorithms, by including carefully optimized assembly code for the most common inner loops for many different CPUs, and by a general emphasis on speed (as opposed to simplicity or elegance).

By caching $2^m, (p-1)!, (q-1)!$ we get further speedups.
The classes of the two modules of our algorithm are described in sections 6.2.1 and 6.2.4. We omit some functions and internals which do not help the reader to understand the algorithms.

## 6.2 Implementation of the Algorithms

In sections 6.2.1 and 6.2.4 we describe how to implement the Well-Separated Pair Decomposition and the Fast Multipole Method, whereas section 6.2.2 describes the implementation for some helper functionality, which is used for refactoring the code. The implementation to compute all necessary parameters needed during the WSPD and the FMM can be found in section 6.2.3.

### 6.2.1 Well-Separated Pair Decomposition

In this section we describe all classes which are involved to compute the Well-Separated Pair Decomposition.
The **Particle** class defines the input of our algorithm. In order to support multiple different value types for the charge and the position, both are templatized. This class has no further functionality besides storing the input. The template parameters **X** and **Q** are forwarded public to other classes by using **Particle**::POSITION and **Particle**::CHARGE.

Particle class storing the templatized input

```
template <class X, class Q>
class Particle {
 public:
  using POSITION = X;
  using CHARGE = Q;

 public:
  Particle(POSITION position, CHARGE charge);
  POSITION position() const;
  CHARGE charge() const;

 private:
  POSITION _x;
  CHARGE _q;
};
```

The **Particles** class represents multiple particles. The very important feature of this container is the very small and constant size, i.e. the size is independent of the number of particles it contains. The class only contains two iterators, otherwise we would have huge memory problems. In order to achieve this very small memory footprint all particles will be stored in a std::list<Particle>. Similar to the particle class it forwards all necessary internal types. The function **diameter**() returns the diameter of the smallest ball containing all particles, i.e. the maximal pairwise distance of all contained particles in the one dimensional case. The function **center**() returns the center point of this ball. These geometric properties are not cached to save memory but will be computed in constant time. The function **begin**(), **end**(), **empty**() and **size**() are the standard functions needed, such that the particles class is a nice C++ container, i.e. the possibility of range-based for loops and the usage of the standard C++ algorithms which are defined in the <algorithm> header [16].

Particles class representing multiple particles including their geometric properties of the smallest surrounding ball

```
template <class Particle>
class Particles {
 public:
  using POSITION = typename Particle::POSITION;
  using CHARGE = typename Particle::CHARGE;
  using ITERATOR = typename std::list<Particle>::iterator;

 public:
  Particles(ITERATOR begin, ITERATOR end);

  ITERATOR begin();
  ITERATOR end();
  bool empty() const;
  int size() const;

  POSITION diameter() const;
  POSITION center() const;

 private:
  typename std::list<Particle>::iterator _begin;
  typename std::list<Particle>::iterator _end;
};
```

The **Node** class represents multiple particles and is an internal class only, i.e. the user does not need it. It got these three main tasks

- Storing the children of this node of the corresponding tree and giving access to the tree structure, e.g. iterating over the children or retrieving the parent.
- Giving access to all other well-separated nodes.
- Storing the data which is needed in the Fast Multipole Method, i.e. the four large integers mentioned in section 6.1.2.

The functions **add_child**(**const** Particles particles) and **add_WS_partner**(Node& partner) are needed while constructing the Fair Split Tree and the corresponding Well-Separated Pair Decomposition. The functions **parent**(), **children**() and **WS_partners**() are used to iterate over the tree and the corresponding WSPD. **diameter**() and **center**() simply forward the same functions of the represented particles. By using **signed_distance**(Node **const**& second_node) one gets the distance between the centers of two nodes $U, W$, whereas the center of a node is the center of the corresponding particles. The distance is signed, i.e.

$$U.\texttt{signed\_distance}(V) = -V.\texttt{signed\_distance}(U).$$

**abs_distance**(Node **const**& second_node) is the absolute value of the signed distance. Similar to the other classes it forwards all necessary internal types.

Node class representing particles and the ordering of the corresponding clusters. It also stores the data structures needed for the Fast Multipole Method. Using this class one is able to iterate over the Fair Split Tree and the corresponding Well-Separated Pair Decomposition.

```
template <class Particles>
class Node {
  using POSITION = typename Particles::POSITION;
  using CHARGE = typename Particles::CHARGE;

 public:
  Node(const Node* parent, const Particles particles);

  Node& add_child(const Particles particles);
  void add_WS_partner(Node& partner);

  Node* parent();
  std::vector<Node>& children();
  std::list<Node*>& WS_partners();

  POSITION diameter() const;
  POSITION center() const;

  POSITION abs_distance(Node const& second_node) const;
  POSITION signed_distance(Node const& second_node) const;

 private:
  const sorted_particles _particles;
  mpz_class _Laurent_coefficient_pos;
  mpz_class _Laurent_coefficient_neg;
  mpz_class _Taylor_coefficient_pos;
  mpz_class _Taylor_coefficient_neg;
};
```

The **Tree** class is constructed using all particles and initializes the root node with all given particles during the constructor. By calling **compute_split_tree**(**double** ratio, **int** leaf_size) the Fair Split Tree gets computed, where **split**(Node<Particles>& parent, **double** ratio, **int** leaf_size) is used internally. The `double ratio` is the ratio $r$ between the volumes of the two resulting clusters after a split, cf. [5] for a detailed description of this algorithm. The possible values for the ratio $r$ are given by $\frac{1}{3} \leq r \leq \frac{2}{3}$. The parameter **int** leaf_size corresponds to the size of the leaves of the Fair Split Tree. This parameter is important when one wants to implement the Truncated Tree Fast Multipole Method, cf. chapter 5. When using the normal Fast Multipole Method we choose leaf_size $= 1$ and for the Truncated Tree Fast Multipole Method we choose leaf_size $\in \mathcal{O}(\mathbb{E})$. Afterwards one can iterate over all nodes by calling **nodes**(). The nodes get traversed in hierarchical order, which are defined by a breath first search over the fair Split Tree. This function is needed for the UPCAST. The function **reverted_nodes**() returns the same nodes, but in reverted order. This function is needed for the DOWNCAST.
**compute_WSPD**(**double** ratio, **int** leaf_size, **double** separation) is used to compute the Well-Separated Pair Decomposition with the separation constant **double** separation, where **find_pairs**(Node<Particles>& left, Node<Particles>& right, **double**

separation) is used internally. The runtime of this algorithm is linearithmic in the number of particles, cf. [5] for a detailed description and analysis of this algorithm. After the WSPD is computed one can iterate over all well-separated pairs by calling the **pairs**() function.
The to_plantuml(**bool** with_hierarchy, **bool** with_WSPD) function can be used to export the complete tree into a plantuml file [27]. Afterwards this file can be transformed via plantuml into a diagram of the tree. Using the argument with_hierarchy one can decide whether the hierarchy should be shown or not and by using the argument with_WSPD one can decide whether he wants to see the well-separated pairs as well. The corresponding Laurent and Taylor series of each node shown as well. This functionality was very helpful during implementation and debugging.

The Tree class is used to compute the Fair Split Tree and the Well-Separated Pair Decomposition. After initializing these structures one can iterate over all nodes in hierarchical order and over all pairs of well-separated clusters.

```
template <class Particles>
class Tree {
 public:
  Tree(Particles particles);

  std::string to_plantuml(bool with_hierarchy, bool with_WSPD) const;

  std::vector<Node<Particles>*>& nodes();
  std::vector<Node<Particles>*>& reverted_nodes();

  std::vector<std::pair<Node<Particles>*, Node<Particles>*>>& pairs();

 protected:
  int compute_split_tree(double ratio, int leaf_size);
  int split(Node<Particles>& parent, double ratio, int leaf_size);

  int compute_WSPD(double ratio, int leaf_size, double separation);
  void find_pairs(Node<Particles>& left, Node<Particles>& right, double separation);

 private:
  Node<sorted_particles> _root;
};
```

These four classes are sufficient for the Well-Separated Pair Decomposition and form the foundations of the Fast Multipole Method. The most interesting part is the algorithm to compute the Well-Separated Pair Decomposition in linearithmic time which is far from obvious.

### 6.2.2 Helper Functionality

In this section we describe some helper functionality which is widely used throughout the Fast Multipole Method. The functionality is independent of the Fast Multipole Method, therefore it is outsourced in this class. Some more Fast Multipole

Method specific helper functionality is included in the **FMM** class. The **Helper** does not store any data, it only consolidates different functionalities which are described here. Most of the functions should return multiple return values, instead we have chosen to always return **void** and only change the first $N$ arguments with non constant references which are mostly called r, r_pos or r_neg. This unifies and simplifies our API.
By using **set_zero**(mpz_class& integer, **uint64_t** size) one sets size limbs of the GMP integer to zero. This functionality is needed to initialize an GMP integer during the INITIALIZE step and during computations to reuse allocated memory. When one wants to add two integers and one of both summands is not necessary anymore, **in_place_add**(mpz_class& r, mpz_class **const**& s, **uint64_t** size) should be used which could be seen as r += s. The **uint64_t** size is needed for internal purposes.
The next functions are meant to used for our coefficient vectors $x$, i.e. we should encode them into two integers, the positive part x_pos and the negative part x_neg. The result is always r_pos and r_neg. We only want to consider the case, where the block_size of all integers are the same. This assumption is sufficient for our algorithms and simplifies the implementation.
To multiply two coefficient vectors $x$ and $y$, cf. section 6.1.2, we split up both into the positive and negative part. The analysis of our algorithm shows that there are a lot of cases, where the negative part is zero, e.g. the scaled (reciprocal) distance integers, cf. sections 4.3.2 to 4.3.4, in UPCAST, CONVERSION and DOWNCAST have an alternating sign when computed from the node $U$ to the node $V$ and are strictly positive in the other direction, i.e. from the node $V$ to the node $U$. Therefore we got a specialization for the case, where one of the coefficient vectors are strictly positive. But we generalize the multiplication functions such that the number of coefficients in the vectors are different, i.e. the number of blocks (**uint64_t** blocks1 $\neq$ **uint64_t** blocks2). The two cases of the **product** function run in $\mathcal{O}_2(n \cdot \log n \cdot \log\log n)$ for large inputs. For smaller inputs a hybrid method is used by the underlying GMP library.
The next two functions are block by block, resp. vector, functions as well. **blockwise_subtract**(...) is used subtract two vectors $s_1$ and $s_2$ and returns the resulting vector split into the positive and negative part. The **blockwise_division**(...) function is used to divide each coefficient of a vector which is encoded into an integer by the given divisor.
The last function is the normalization of a coefficient vector, cf. section 6.1.2, i.e. for all $k = 1, \dots, n$ not both $k$th coefficient of $x_+$ and $x_-$ are strictly greater than zero. This minimizes the encoding of the coefficient vector and accelerates the algorithm. By calling **normalize**(mpz_class& pos, mpz_class& neg, ...) the corresponding vector $r = \text{pos} + \text{neg}$ gets normalized to achieve the minimal encoding.

Helper class containing several methods needed in the Fast Multipole Method

```
class Helper {
 public:
  static void set_zero(mpz_class& integer, uint64_t size);

  static void in_place_add(mpz_class& r, mpz_class const& s, uint64_t size);

  static void product(mpz_class& r_pos, mpz_class& r_neg,
                      mpz_class const& s1_pos, uint64_t blocks1,
                      mpz_class const& s2_pos, mpz_class const& s2_neg, uint64_t blocks2,
                      uint64_t block_size);

  static void product(mpz_class& r_pos, mpz_class& r_neg,
                      mpz_class const& s1_pos, mpz_class const& s1_neg, uint64_t blocks1,
                      mpz_class const& s2_pos, mpz_class const& s2_neg, uint64_t blocks2,
                      uint64_t block_size);

  static void blockwise_subtract(mpz_class& r_pos, mpz_class& r_neg,
                                 mpz_class const& s1, mpz_class const& s2,
                                 uint64_t nbr_blocks, uint64_t block_size);

  static void blockwise_division(mpz_class& r, mpz_class const& block,
                                 uint64_t nbr_blocks, uint64_t block_size,
                                 mpz_class const& divisor);

  static void normalize(mpz_class& pos, mpz_class& neg,
                        uint64_t nbr_blocks, uint64_t block_size);
}
```

### 6.2.3 Parameters

In this section we describe the **Parameters** class which computes and stores all needed parameters for our algorithms. The class gets constructed using all particles a factor xDen which rescales all positions by $\frac{1}{\texttt{xDen}}$ and a factor qDen which rescales all charges by $\frac{1}{\texttt{qDen}}$ and the desired accuracy E, such that the result is an approximation to the exact result up to an error of at most $2^{-\texttt{E}}$. The parameters are described below

- The parameter **uint64_t** ms corresponds to the minimal pairwise distance of all particles (**uint64_t** ms is computed in linear time).
- The parameter **uint64_t** mq is a scaling factor for all charges, such that they can be encoded by integers and the resulting error gets bounded by $2^{-\texttt{E}}$.
- The parameter **uint64_t** mz is a scaling factor for all positions, such that they can be encoded by integers and the resulting error gets bounded by $2^{-\texttt{E}}$.
- The parameter **uint64_t** p is the truncation parameter of the Laurent series, i.e. it denotes the number of coefficients of the Laurent series.

- The parameter **uint64_t** P is the size of the blocks in the integer representing the Laurent series.
- The parameter **uint64_t** P_limbs is the number of limbs, cf. [12], which are needed that the size of the blocks are great enough, i.e. $\text{P\_limbs} \cdot \text{mp\_bits\_per\_limb} \geq \text{P}$.
- The parameter **uint64_t** q is the truncation parameter of the Taylor series, i.e. it denotes the number of coefficients of the Taylor series.
- The parameter **uint64_t** Q is the size of the blocks in the integer representing the Taylor series.
- The parameter **uint64_t** Q_limbs is the number of limbs, cf. [12], which are needed that the size of the blocks are great enough, i.e. $\text{Q\_limbs} \cdot \text{mp\_bits\_per\_limb} \geq \text{Q}$.
- The parameter **uint64_t** m_conv is an additional factor needed in the CONVERSION step, such that the resulting error gets bounded by $2^{-\text{E}}$ despite the divisions are not exact.
- The parameter **double** c is the separation constant needed for the Well-Separated Pair Decomposition.

The size of the integers representing a Laurent series are given by $\text{P\_limbs} \cdot \text{mp\_bits\_per\_limb} \cdot \text{p}$ and size of the integers representing a Taylor series are given by $\text{Q\_limbs} \cdot \text{mp\_bits\_per\_limb} \cdot \text{q}$. All parameters are computed accordingly to the analysis of the Fast Multipole Method in order to achieve our estimated runtime and error bounds.

Parameters class used to compute and store all necessary parameters for our algorithms

```
class Parameters {
 public:
  template <class Particle>
  Parameters(std::vector<Particle> const& particles,
             typename Particle::POSITION const& xDen,
             typename Particle::POSITION const& qDen,
             uint64_t E);

 private:
  uint64_t ms;
  uint64_t mq;
  uint64_t mz;

  uint64_t p;
  uint64_t P;
  uint64_t P_limbs;

  uint64_t q;
  uint64_t Q;
  uint64_t Q_limbs;

  uint64_t m_conv;

  double c;
};
```

### 6.2.4 Fast Multipole Method

In this section we describe the **FMM** class which computes the Fast Multipole Method.
The two protected functions **distance_integer**(mpz_class& r_pos, mpz_class **const**& dist_pos) and **distance_integer**(mpz_class& r_pos, mpz_class& r_neg, mpz_class **const**& dist_neg) are used to compute the scaled (reciprocal) distance integers, cf. sections 4.3.2 to 4.3.4. If the distance between two nodes is positive, the scaled (reciprocal) distance integer only contains positive entries, therefore we only need one output argument. In this case we will use the first method. If the distance between two nodes is negative, the signs of the entries of the scaled (reciprocal) distance integer are alternating, therefore the output has to be split into the positive and the negative part. In this case we will use the second method.
The functions **INIT**(Tree<Particles>& tree), **UPCAST**(Tree<Particles>& tree), **CONVERSION**(Tree<Particles>& tree) and **DOWNCAST**(Tree<Particles>& tree) are implemented as described in sections 4.3.1 to 4.3.4. The implementation for the function std::map<Particles*, mpf_class> **EVALUATE**(Tree<Particles>& tree) is straight forward by simply returning the scaled constant coefficient for each leaf, i.e. particle. std::map<Particles*, mpf_class> was chosen as the return type, because the mpf_class holds arbitrarily accurate floating point values and the map is a convenient container for other applications requiring the Fast Multipole Method.

FMM class computing the Fast Multipole Method

```
template <class Particles>
class FMM {
 public:
  FMM(Parameters const& parameters);

  void INIT(Tree<Particles>& tree);
  void UPCAST(Tree<Particles>& tree);
  void CONVERSION(Tree<Particles>& tree);
  void DOWNCAST(Tree<Particles>& tree);
  std::map<Particles*, mpf_class> EVALUATE(Tree<Particles>& tree);

 protected:
  void distance_integer(mpz_class& r_pos, mpz_class const& dist_pos);
  void distance_integer(mpz_class& r_pos, mpz_class& r_neg, mpz_class const& dist_neg);
};
```

## 6.3 Additional Properties of the Implementation

It turns out, that this implementation does not only solve Trummer's Problem

$$(C(q,z)\cdot q)_i = \sum_{\substack{j=1\\j\neq i}}^{N} \frac{q_i q_j}{z_i - z_j},$$

it solves a variety of similar problems, namely

$$\sum_{\substack{j=1\\j\neq i}}^{N} \frac{q_i q_j}{(z_i - z_j)^k},$$

for a fixed $k \in \mathbb{N}$. Therefore we are able to solve the N-body problem for more forces, e.g. using intermolecular forces [20], like the Debye force which scales with $r^{-6}$ by choosing $k = 6$. By choosing $k = 2$ we can solve the three dimensional Coulomb force. By choosing different $k$'s running the corresponding algorithms and adding the results we can easily solve the following problem

$$\sum_{\substack{j=1\\j\neq i}}^{N} \sum_{k=0}^{K} \frac{q_{i,k}q_{j,k}}{(z_{i,k} - z_{j,k})^k}.$$

Therefore we can solve the N-body problem, where multiple forces occur, e.g. the Coulomb and Debye forces. In molecular chemistry in nearly all problems multiple forces occur, therefore our algorithm can be easily adapted to these wider range of problems. But to be fair, at the moment the algorithm only supports the one dimensional real case, the extension to two dimensions is straight forward using complex values. The extension to three dimension should be possible using quaternions. The trick to handle these numbers is equivalent to the trick for the negative numbers, i.e. splitting them up into the positive real, positive complex, negative real and negative complex part and for quaternions analogously. By using quaternions one has to take a bit more care, because of their non commutativity. With this algorithm we are now able to extend the problem to an even wider range of problems, namely problems where the forces are not a Laurent Polynomial, like before, to problems which forces can be approximated by Laurent Polynomial. These forces occur in nuclear physics.
Only with a slight modification of the INITIALIZE step one can solve the following problem

$$\sum_{\substack{j=1\\j\neq i}}^{N} \sum_{k=0}^{K} \frac{q_{i,k}q_{j,k}}{(z_i - z_j)^k}.$$

Hence we do not need to run the algorithm multiple times, which saves an additional factor of $k$ for the runtime. Also the overall implementation complexity is reduced.

## 6.4 Statistics of our Implementation

As stated in section 6.1 the effort, in person month, to implement software is given by

$$E = a \cdot \left(\frac{\text{LOC}}{1000}\right)^b,$$

whereas LOC is the number of lines of delivered code and the parameters $a$ and $b$ are given by

- $a = 2.4, b = 1.05$ for an 'easy' project,
- $a = 3.0, b = 1.12$ for an 'intermediate' project, and
- $a = 3.6, b = 1.20$ for a 'complex' project.

Our Implementation needs 998 lines of delivered code for the implementation of the Fast Multipole Method and 498 lines of delivered code for the implementation of the Well-Separated Pair Decomposition. By using the COCOMO model we get the following estimated for the effort in person months

- 3.6 person month if the algorithm is considered 'easy',
- 4.4 person month if the algorithm is considered 'intermediate',
- 5.2 person month if the algorithm is considered 'complex'.

The order of magnitude of the effort is not that bad estimated.
Using the COCOMO model we have estimated the overall effort needed for implementation. In order to estimate the complexity of each function separately we need a different approach, the *cyclomatic complexity* which was developed by Thomas J. McCabe [21] in order to estimate the complexity of source code. Informally speaking the cyclomatic complexity of an implementation of a procedure is the number of independent paths. If there are no control flow statements, like "if", there is exactly one path, hence the cyclomatic complexity is 1. If there is one "if", the path splits up, and the cyclomatic complexity is 2 and if there is a "switch" statement with 4 possible cases, then the path gets split into 4 independent paths, therefore the cyclomatic complexity is 4 for this part. The cyclomatic complexity can be used to compute the number of test cases we need to achieve a 100% test coverage of our implementation.
Thomas J. McCabe has suggested that the cyclomatic complexity should be at most 10 for each function, if the complexity is higher, the function should be rewritten to reduce it. We have used the `pmccabe` [2] tool to analyze the cyclomatic complexity during our implementation in order to fulfill McCabe's suggestion. The final version of our implementation[1] got the following cyclomatic complexities $cc$

- Well-Separated Pair Decomposition from section 6.2.1: $cc \leq 8$.
- Fast Multipole Method from section 6.2.4: $cc \leq 7$.
- Computation of the parameters from section 6.2.3: $cc \leq 4$.
- Helper functionalities from section 6.2.2: $cc \leq 4$.

All cyclomatic complexities are lower than 10, hence we have accomplished McCabe's suggestion.

[1] We have ignored the complexity of output functionality, e.g. functions which produce plantuml output, because these functions could be deleted without influencing the Fast Multipole Method.

Chapter 7

# Empirical Evaluation

In this chapter we empirically evaluate our implementation of the Fast Multipole Method in order to show the correctness of our implementation in section 7.1 and the claimed subquadratic runtime in section 7.2. In section 7.3 we estimate the memory usage of our algorithm by using a fit to measured values. All benchmarks have been executed on the same computer (Ubuntu 16.04.4 LTS, 64 GB RAM, Intel® Core™ i7-6950X CPU @ 3.00GHz) and using only a single CPU core. More detailed information on the machine can be found in the footnote[1].

## 7.1 Correctness of the Implementation

To test the correctness of the implementation we have run several instances of Trummer's Problem with different particles, i.e. different positions and charges. By using Mathematica [15] we are able to solve the same problem up to arbitrary accuracy. Our tests have shown that both produce the same results, which indicates that our implementation contains no errors. These benchmarks were very helpful while debugging the algorithm.

## 7.2 Runtime of the Implementation

We have already estimated the asymptotic behavior $\mathcal{O}_2(N\mathbb{E}^3)$. In this section we will empirically evaluate the runtime, i.e. estimating the constants in the big-$\mathcal{O}$ notation. The idea to estimate the runtime is by fitting a model function to multiple benchmarks. For the benchmarks we use an equidistant distribution of particles in the unit interval. We've chosen the following grid for our parameters, the number of particles $N = 2000, \dots, 30000$ and the error exponent $\mathbb{E} = 2, \dots, 30$. Because of the huge memory consumption, cf. section 7.3, we can not expand the grid much further. The measured runtime are displayed in tables 7.1 to 7.3. In the next two section we operate with the measured values in order to fit the runtime function.

[1] MemTotal: 65886036 kB, Buffers: 39920 kB, SwapTotal: 67018748 kB, Writeback: 0 kB, AnonPages: 63927028 kB, Mapped: 46172 kB, Shmem: 4492 kB, Slab: 109496 kB, KernelStack: 7616 kB, PageTables: 219252 kB, NFS_Unstable: 0 kB, Bounce: 0 kB, WritebackTmp: 0 kB, CommitLimit: 99961764 kB, Committed_AS: 113900456 kB, VmallocTotal: 34359738367 kB, HardwareCorrupted: 0 kB, AnonHugePages: 403456 kB, CmaTotal: 0 kB, CmaFree: 0 kB, HugePages_Total: 0, HugePages_Rsvd: 0, HugePages_Surp: 0, Hugepagesize: 2048 kB, DirectMap4k: 135828 kB, DirectMap2M: 11309056 kB, DirectMap1G: 57671680 kB, vendor_id: GenuineIntel, cpu family: 6, model: 79, model name: Intel(R) Core(TM) i7-6950X CPU @ 3.00GHz, stepping: 1, microcode: 0xb00001c, cpu MHz: 2999.882, cache size: 25600 KB, physical id: 0, siblings: 20, core id: 0, cpu cores: 10, apicid: 0, initial apicid: 0, fpu: yes, fpu_exception: yes, cpuid level: 20, wp: yes, flags:, fpu, vme, de, pse, tsc, msr, pae, mce, cx8, apic, sep, mtrr, pge, mca, cmov, pat, pse36, clflush, dts, acpi, mmx, fxsr, sse, sse2, ss, ht, tm, pbe, syscall, nx, pdpe1gb, rdtscp, lm, constant_tsc, arch_perfmon, pebs, bts, rep_good, nopl, xtopology, nonstop_tsc, aperfmperf, eagerfpu, pni, pclmulqdq, dtes64, ds_cpl, vmx, est, tm2, ssse3, sdbg, fma, cx16, xtpr, pdcm, pcid, dca, sse4_1, sse4_2, x2apic, movbe, popcnt, tsc_deadline_timer, aes, xsave, avx, f16c, rdrand, lahf_lm, abm, 3dnowprefetch, epb, intel_pt, tpr_shadow, vnmi, flexpriority, ept, vpid, fsgsbase, tsc_adjust, bmi1, hle, avx2, smep, bmi2, erms, invpcid, rtm, cqm, rdseed, adx, smap, xsaveopt, cqm_llc, cqm_occup_llc, cqm_mbm_total, cqm_mbm_local, dtherm, ida, arat, pln, pts, bogomips: 5996.30, clflush size: 64, cache_alignment: 64, address sizes: 46 bits physical, 48 bits virtual

Table 7.1.: Measured Runtime for the implementation of the Fast Multipole Method for $N = 2500, 5000, 7500, 10000$.

| | 2500 | 5000 | 7500 | 10000 |
|---|---|---|---|---|
| 2 | 7, 7, 7, 7, 7 | 19, 19, 19, 18, 18 | 30, 30, 30, 30, 30 | 48, 48, 48, 48, 48 |
| 5 | 12, 12, 12, 12, 12 | 29, 29, 29, 29, 29 | 56, 56, 56, 56, 56 | 82, 82, 82, 82, 82 |
| 7 | 16, 16, 16, 16, 16 | 42, 41, 42, 41, 41 | 68, 68, 69, 68, 68 | 104, 103, 104, 103, 103 |
| 10 | 26, 26, 26, 26, 26 | 62, 62, 62, 62, 62 | 102, 103, 104, 103, 103 | 172, 171, 172, 169, 171 |
| 12 | 33, 34, 34, 34, 33 | 89, 89, 89, 89, 88 | 141, 141, 142, 141, 142 | 205, 207, 212, 210, 209 |
| 15 | 49, 49, 50, 49 | 127, 128, 127, 128, 127 | 183, 183, 183, 183 | 289, 288, 290, 288 |
| 17 | 65, 65, 65, 65 | 156, 156, 155, 155 | 230, 228, 229, 229 | 367, 369 |
| 20 | 84, 84, 84, 84 | 198, 198, 198, 198, 198 | 291, 292, 291 | 460, 462, 463 |
| 22 | 100, 100, 100 | 237, 237, 237 | 368, 368 | 601 |
| 25 | 134, 135, 135 | 324, 324, 324, 325 | 487, 486 | 696, 695 |
| 27 | 160, 160, 160 | 374, 373, 375 | 547, 547 | 819 |
| 30 | 197, 197, 197 | 444, 444, 443 | 665, 668 | 1101, 1101 |
| 32 | 240, 240, 239 | 520, 519 | 779, 777 | 1240 |

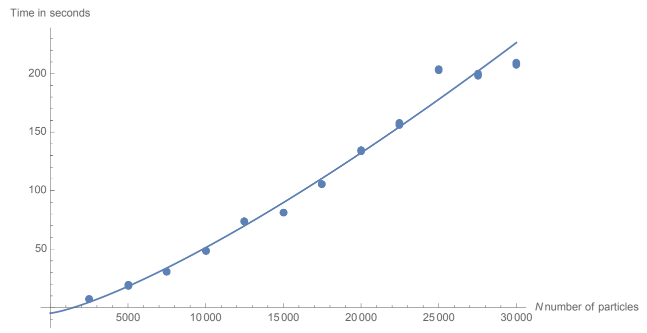
Table 7.2.: Measured Runtime for the implementation of the Fast Multipole Method for $N = 12500, 15000, 17500, 20000, 22500$.

| | 12500 | 15000 | 17500 | 20000 | 22500 |
|---|---|---|---|---|---|
| 2 | 74, 73, 73, 74 | 81, 81, 81 | 105, 105, 105 | 134, 134, 134 | 158, 155, 157 |
| 5 | 132, 132, 132, 132 | 129, 129, 128, 129 | 174, 174, 174 | 211, 212, 211, 210 | 253, 252, 253 |
| 7 | 165, 166, 166, 166 | 169, 170, 169 | 235, 234, 234 | 279, 278 | 340, 339 |
| 10 | 250, 248, 247, 249 | 267, 265, 267 | 340, 338 | 421, 423, 422 | 481, 488 |
| 12 | 320, 315 | 336, 333 | 436, 435 | 525, 519 | 614, 614 |
| 15 | 412, 411 | 417, 421, 420 | 605, 595 | 699, 684, 701 | 833, 835 |
| 17 | 528, 529 | 512, 516 | 707, 705 | 805, 803 | 994, 998 |
| 20 | 694, 685 | 690, 690, 690 | 915, 914 | 1099, 1089, 1108 | 1334 |
| 22 | 827 | 964 | 1108 | 1413 | 1549 |
| 25 | 1023 | 1131, 1130 | 1290 | 1667, 1682 | 2032 |
| 27 | 1226 | 1319 | 1633 | 1978 | 2405 |
| 30 | 1509 | 1649 | 2081 | 2475 | 3024 |
| 32 | 1782 | 1791 | 2266 | 2764 | 3395 |

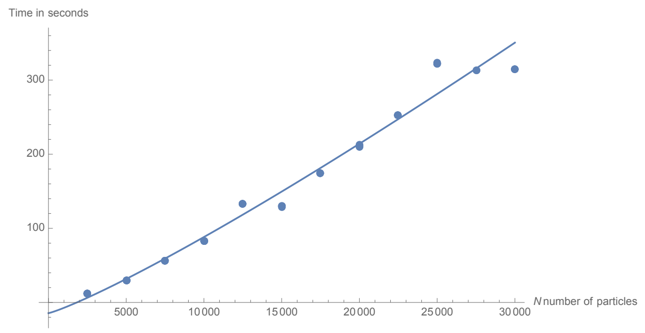
Table 7.3.: Measured Runtime for the implementation of the Fast Multipole Method for $N = 25000, 27500, 30000, 32500$.

| | 25000 | 27500 | 30000 | 32500 |
|---|---|---|---|---|
| 2 | 202, 203, 203 | 199, 199, 198 | 209, 208, 207 | 254, 255, 254 |
| 5 | 322, 322, 322, 324 | 313, 313 | 314, 315, 314 | 390, 389 |
| 7 | 423, 422 | 428, 432 | 457, 451 | 476, 473 |
| 10 | 613, 619, 607 | 606, 612 | 675, 674, 675 | 618, 622 |
| 12 | 784, 782 | 737, 736 | 774, 762 | 778, 791 |
| 15 | 1023, 1012, 1024 | 958, 957 | 1000, 985, 1000 | 1080, 1095 |
| 17 | 1240, 1242 | 1196, 1212 | 1241, 1238 | 1496, 1500 |
| 20 | 1653, 1651 | 1597 | 1712 | 1777 |
| 22 | 1760 | 1889 | 1892 | 2308 |
| 25 | 2532, 2542 | 2489 | 2685 | 2746, 2785 |
| 27 | 2786 | 2785 | 2862, 2872 | 3358, 3350 |
| 30 | 3637 | 3686, 3684 | 3551, 3567 | 3727, 3710 |
| 32 | 4053 | 3961, 3965 | 4511, 4506 | 4685, 4670 |

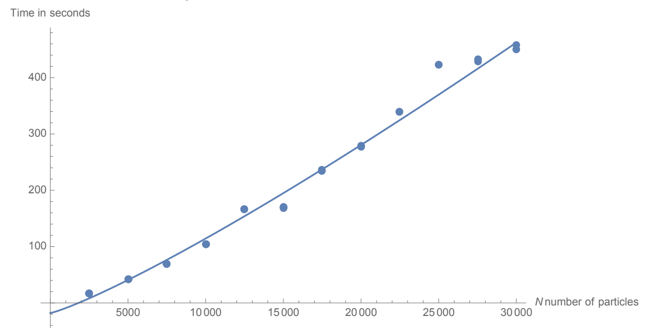
## 7.2.1 Runtime of the Implementation with a variable Number of Particles and fixed Error Exponent



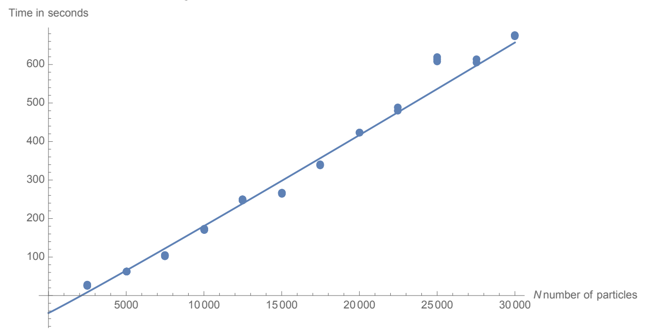
(a) Runtime of the implementation for a fixed error exponent $\mathbb{E} = 2$. The dots are the measured runtime and the line is the fitted function $T_{\mathbb{E}=2}(N) = 0.000384 \cdot N^{1.29} - 4.74$, which is nearly linear.



(b) Runtime of the implementation for a fixed error exponent $\mathbb{E} = 5$. The dots are the measured runtime and the line is the fitted function $T_{\mathbb{E}=5}(N) = 0.00248 \cdot N^{1.15} - 14.4$, which is nearly linear.



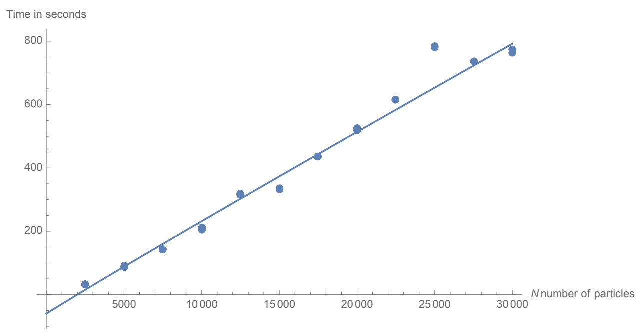
(c) Runtime of the implementation for a fixed error exponent $\mathbb{E} = 7$. The dots are the measured runtime and the line is the fitted function $T_{\mathbb{E}=7}(N) = 0.00266 \cdot N^{1.17} - 17.9$, which is nearly linear.



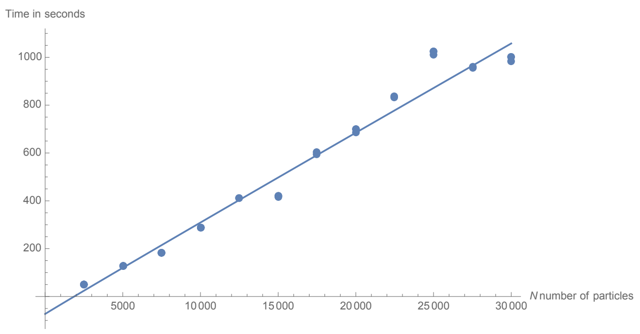
(d) Runtime of the implementation for a fixed error exponent $\mathbb{E} = 10$. The dots are the measured runtime and the line is the fitted function $T_{\mathbb{E}=10}(N) = 0.0167 \cdot N^{1.03} - 45.3$, which is nearly linear.

Figure 7.1.: Runtime of the implementation for a fixed error exponent $\mathbb{E} = 2, 5, 7, 10$ with the measured runtime and the fitted functions $T_{\mathbb{E}=*}(N) = a \cdot N^b - c$, with $b \approx 1$, which is nearly linear.

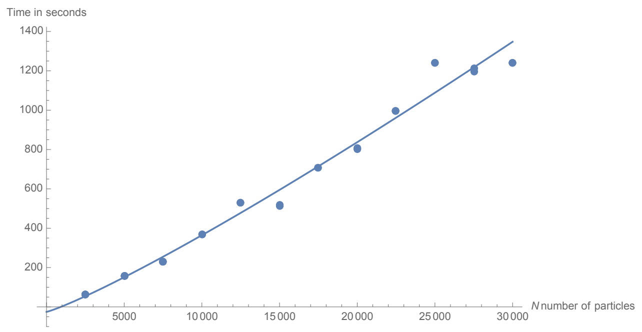
To fit the runtime for fixed values of the error exponent $\mathbb{E}$ we have chosen $T_{\mathbb{E}=*}(N) = a \cdot N^b + c$ as the model function. We get the following values for our fit parameters with the corresponding errors (all values are rounded up to three significant digits). Figures 7.1 to 7.3 shows the measured values and the corresponding fitted function which looks nearly linear in $N$ in all cases. The fitted exponent $b$ for all fixed values of $\mathbb{E}$ is approximately 1. For our further analysis we set the exponent to 1, such that the runtime is a polynomial.



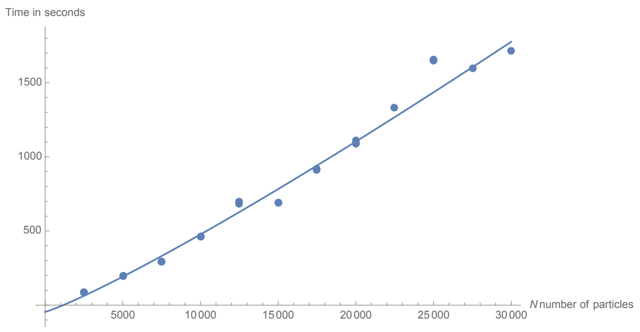
(a) Runtime of the implementation for a fixed error exponent $\mathbb{E} = 12$. The dots are the measured runtime and the line is the fitted function $T_{\mathbb{E}=12}(N) = 0.0368 \cdot N^{0.975} - 60.5$, which is nearly linear.



(b) Runtime of the implementation for a fixed error exponent $\mathbb{E} = 15$. The dots are the measured runtime and the line is the fitted function $T_{\mathbb{E}=15}(N) = 0.0430 \cdot N^{0.987} - 73.0$, which is nearly linear.



(c) Runtime of the implementation for a fixed error exponent $\mathbb{E} = 17$. The dots are the measured runtime and the line is the fitted function $T_{\mathbb{E}=17}(N) = 0.0101 \cdot N^{1.147} - 25.2$, which is nearly linear.

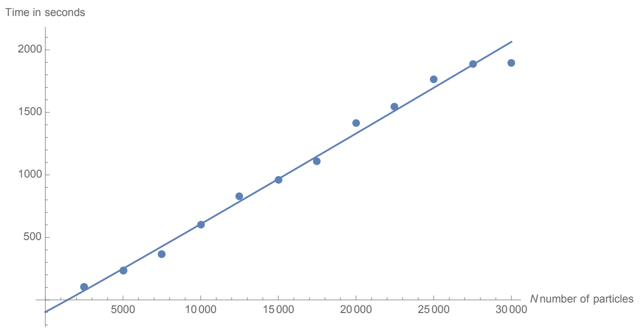


(d) Runtime of the implementation for a fixed error exponent $\mathbb{E} = 20$. The dots are the measured runtime and the line is the fitted function $T_{\mathbb{E}=20}(N) = 0.0148 \cdot N^{1.14} - 45.4$, which is nearly linear.

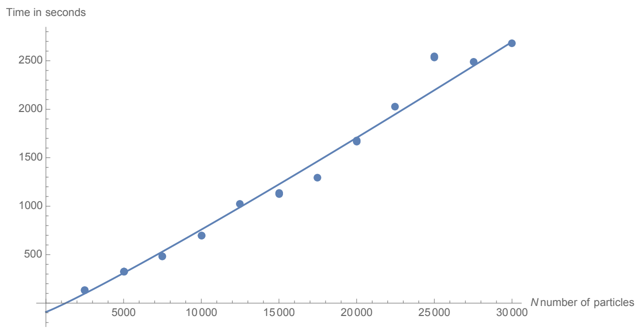
Figure 7.2.: Runtime of the implementation for a fixed error exponent $\mathbb{E} = 12, 15, 17, 20$ with the measured runtime and the fitted functions $T_{\mathbb{E}=*}(N) = a \cdot N^b - c$, with $b \approx 1$, which is nearly linear.

Table 7.4.: Fitted values with the corresponding error bounds for the runtime using the model function $T_{\mathbb{E}=*}(N) = a \cdot N^b + c$ for a fixed error exponents $\mathbb{E}$.

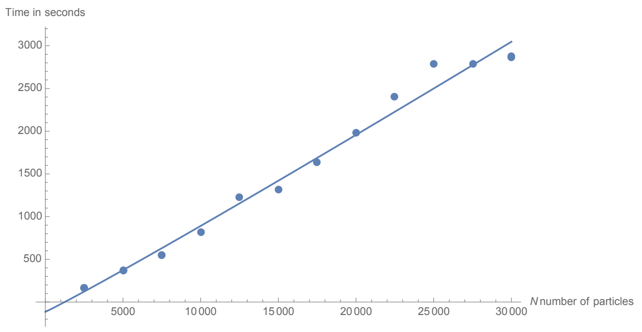
| fixed error exponent $\mathbb{E}$ | prefactor $a$ | exponent $b$ | constant offset $c$ |
|---|---|---|---|
| 2 | $0.000384 \pm 0.000264$ | $1.29 \pm 0.07$ | $-4.74 \pm 4.48$ |
| 5 | $0.00249 \pm 0.00210$ | $1.15 \pm 0.08$ | $-14.4 \pm 10.2$ |
| 7 | $0.00267 \pm 0.00188$ | $1.17 \pm 0.07$ | $-17.9 \pm 10.7$ |
| 10 | $0.0167 \pm 0.0145$ | $1.03 \pm 0.08$ | $-45.4 \pm 23.4$ |
| 12 | $0.0368 \pm 0.0345$ | $0.975 \pm 0.088$ | $-60.5 \pm 32.4$ |
| 15 | $0.0430 \pm 0.0432$ | $0.987 \pm 0.094$ | $-73.0 \pm 46.7$ |
| 17 | $0.0101 \pm 0.0091$ | $1.15 \pm 0.09$ | $-25.2 \pm 39.7$ |
| 20 | $0.0148 \pm 0.0144$ | $1.14 \pm 0.09$ | $-45.4 \pm 55.3$ |
| 22 | $0.0573 \pm 0.0480$ | $1.02 \pm 0.08$ | $-96.3 \pm 64.3$ |
| 25 | $0.0401 \pm 0.0472$ | $1.08 \pm 0.11$ | $-91.2 \pm 113.6$ |
| 27 | $0.0680 \pm 0.0662$ | $1.04 \pm 0.09$ | $-115 \pm 110$ |
| 30 | $0.431 \pm 0.585$ | $0.888 \pm 0.127$ | $-356 \pm 257$ |
| 32 | $0.0597 \pm 0.0568$ | $1.09 \pm 0.09$ | $-116 \pm 147$ |



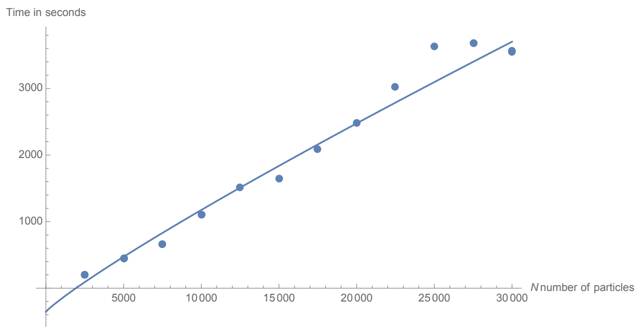
(a) Runtime of the implementation for a fixed error exponent $\mathbb{E} = 22$. The dots are the measured runtime and the line is the fitted function $T_{\mathbb{E}=22}(N) = 0.0573 \cdot N^{1.02} - 96.3$, which is nearly linear.



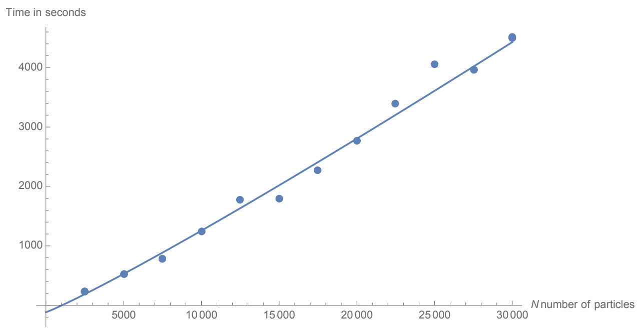
(b) Runtime of the implementation for a fixed error exponent $\mathbb{E} = 25$. The dots are the measured runtime and the line is the fitted function $T_{\mathbb{E}=25}(N) = 0.0401 \cdot N^{1.08} - 91.2$, which is nearly linear.



(c) Runtime of the implementation for a fixed error exponent $\mathbb{E} = 27$. The dots are the measured runtime and the line is the fitted function $T_{\mathbb{E}=27}(N) = 0.0680 \cdot N^{1.04} - 114$, which is nearly linear.



(d) Runtime of the implementation for a fixed error exponent $\mathbb{E} = 30$. The dots are the measured runtime and the line is the fitted function $T_{\mathbb{E}=30}(N) = 0.431 \cdot N^{0.888} - 356$, which is nearly linear.

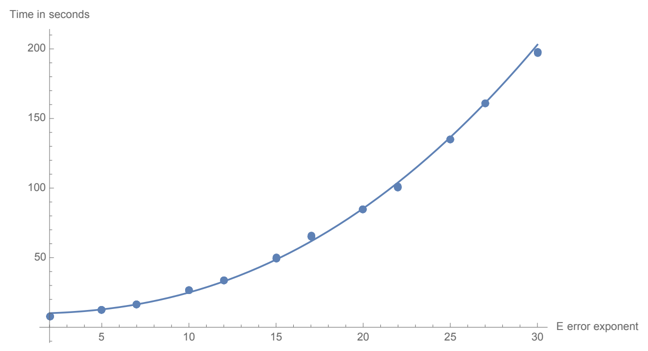


(e) Runtime of the implementation for a fixed error exponent $\mathbb{E} = 32$. The dots are the measured runtime and the line is the fitted function $T_{\mathbb{E}=32}(N) = 0.0597 \cdot N^{1.09} - 116$, which is nearly linear.

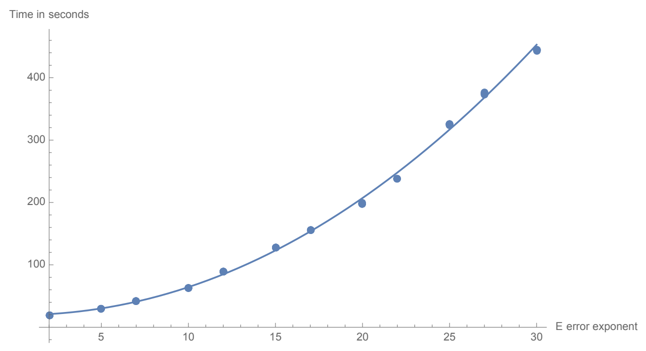
Figure 7.3.: Runtime of the implementation for a fixed error exponent $\mathbb{E} = 22, 25, 27, 30, 32$ with the measured runtime and the fitted functions $T_{\mathbb{E}=*}(N) = a \cdot N^b + c$, with $b \approx 1$, which is nearly linear.

### 7.2.2 Runtime of the Implementation with a fixed Number of Particles and variable Error Exponent

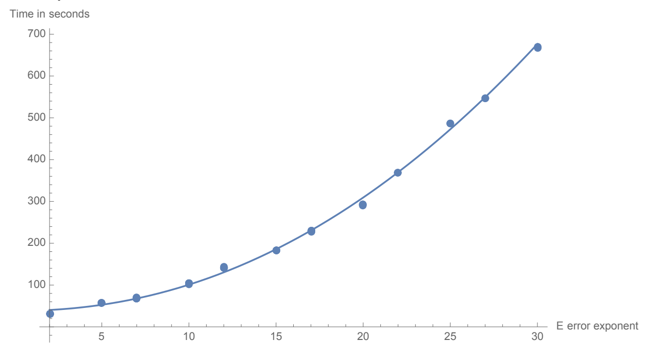
In this section we do the similar analysis as in section 7.2.1 but now the number of particles $N$ is fixed. To fit the runtime for



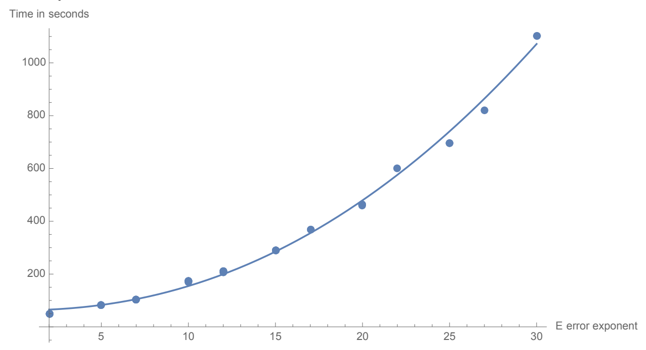
(a) Runtime of the implementation for a fixed number of particles $N = 2500$. The dots are the measured runtime and the line is the fitted function $T_{N=2500}(\mathbb{E}) = \cdot\mathbb{E}$, which is nearly quadratic.



(b) Runtime of the implementation for a fixed number of particles $N = 5000$. The dots are the measured runtime and the line is the fitted function $T_{N=5000}(\mathbb{E}) = \cdot\mathbb{E}$, which is nearly quadratic.



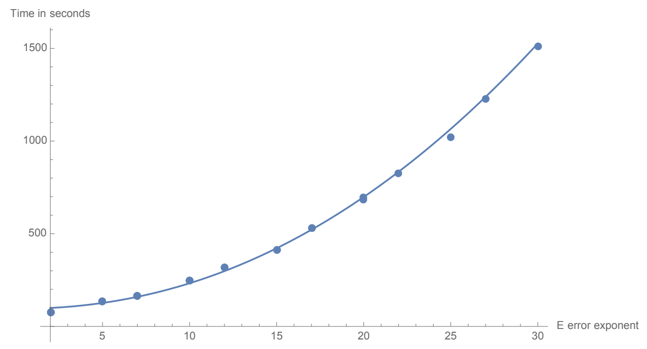
(c) Runtime of the implementation for a fixed number of particles $N = 7500$. The dots are the measured runtime and the line is the fitted function $T_{N=7500}(\mathbb{E}) = \cdot\mathbb{E}$, which is nearly quadratic.



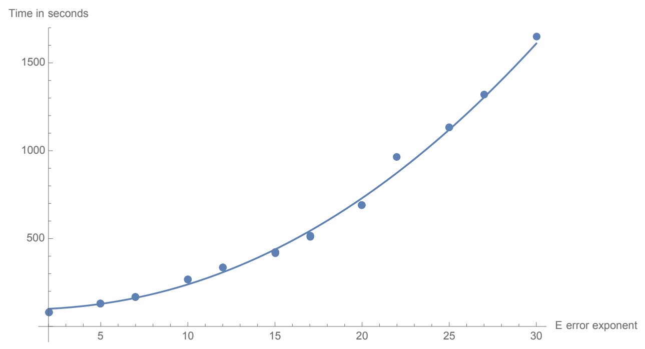
(d) Runtime of the implementation for a fixed number of particles $N = 10000$. The dots are the measured runtime and the line is the fitted function $T_{N=10000}(\mathbb{E}) = \cdot\mathbb{E}$, which is nearly quadratic.

Figure 7.4.: Runtime of the implementation for a fixed number of particles $N = 2500, 5000, 7500, 10000$ with the measured runtime and the fitted functions $T_{N=*}(\mathbb{E}) = a \cdot \mathbb{E}^b - c$, with $b \approx 2$, which is nearly quadratic.

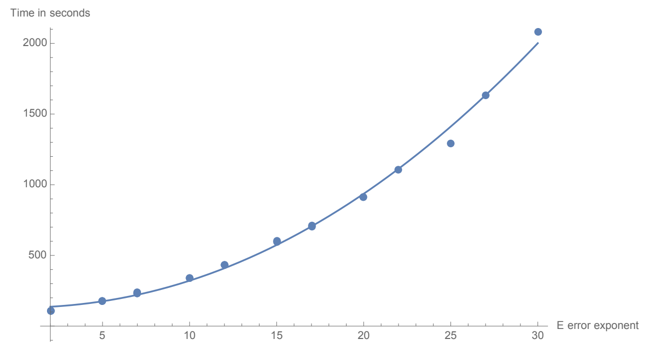
fixed values of the error exponent $\mathbb{E}$ we have chosen $T_{\mathbb{E}=*}(N) = a \cdot N^b + c$ as the model function. Figures 7.1 to 7.3 shows the measured values and the corresponding fitted function which looks nearly linear in $N$ in all cases. The fitted exponent $b$ for all fixed values of $\mathbb{E}$ is approximately 1. For our further analysis we set the exponent to 1, such that the runtime is a polynomial.



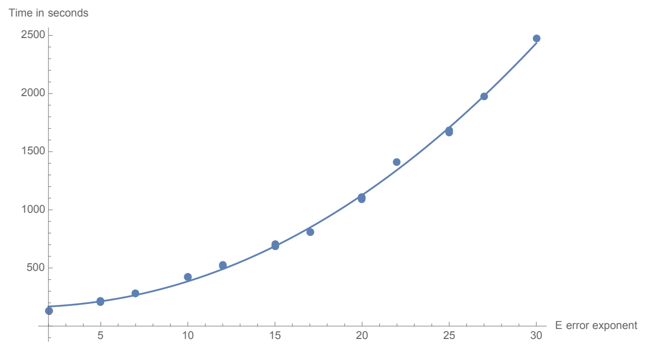
(a) Runtime of the implementation for a fixed number of particles $N = 12500$. The dots are the measured runtime and the line is the fitted function $T_{N=12500}(\mathbb{E}) = \cdot\mathbb{E}$, which is nearly quadratic.



(b) Runtime of the implementation for a fixed number of particles $N = 15000$. The dots are the measured runtime and the line is the fitted function $T_{N=15000}(\mathbb{E}) = \cdot\mathbb{E}$, which is nearly quadratic.



(c) Runtime of the implementation for a fixed number of particles $N = 17500$. The dots are the measured runtime and the line is the fitted function $T_{N=17500}(\mathbb{E}) = \cdot\mathbb{E}$, which is nearly quadratic.

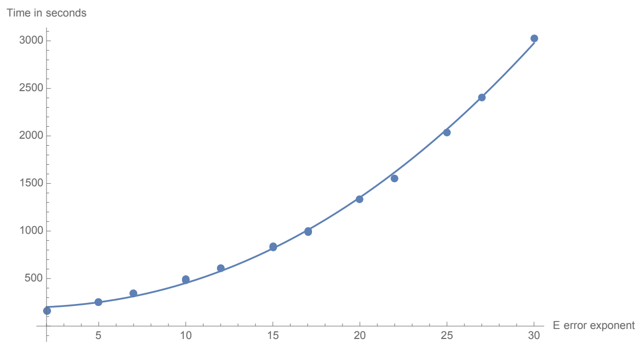


(d) Runtime of the implementation for a fixed number of particles $N = 20000$. The dots are the measured runtime and the line is the fitted function $T_{N=20000}(\mathbb{E}) = \cdot\mathbb{E}$, which is nearly quadratic.

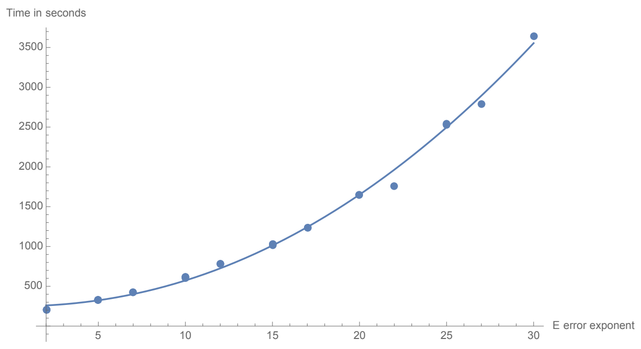
Figure 7.5.: Runtime of the implementation for a fixed number of particles $N = 12500, 15000, 17500, 20000$ with the measured runtime and the fitted functions $T_{N=*}(\mathbb{E}) = a \cdot \mathbb{E}^b - c$, with $b \approx 2$, which is nearly quadratic.

Table 7.5.: Fitted values with the corresponding error bounds for the runtime using the model function $T_{N=*}(\mathbb{E}) = a \cdot N^b + c$ for a fixed number of particles $N$.

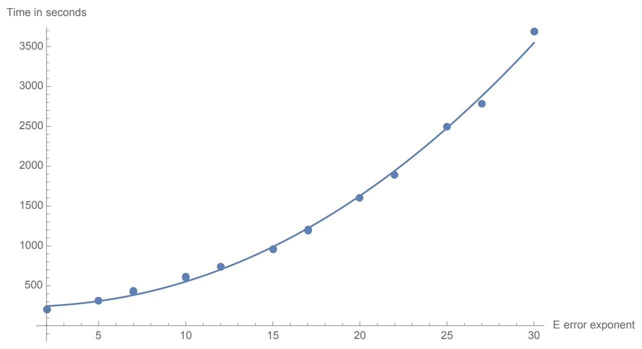
| fixed number of particles $N$ | prefactor $a$ | exponent $b$ | constant offset $c$ |
|---|---|---|---|
| 2500 | $0.0738 \pm 0.0079$ | $2.31 \pm 0.03$ | $9.74 \pm 0.76$ |
| 5000 | $0.386 \pm 0.036$ | $2.06 \pm 0.03$ | $19.6 \pm 1.7$ |
| 7500 | $0.482 \pm 0.050$ | $2.11 \pm 0.03$ | $38.3 \pm 2.6$ |
| 10000 | $0.605 \pm 0.101$ | $2.18 \pm 0.05$ | $62.7 \pm 5.8$ |
| 12500 | $1.02 \pm 0.14$ | $2.13 \pm 0.04$ | $95.2 \pm 6.8$ |
| 15000 | $1.02 \pm 0.21$ | $2.15 \pm 0.06$ | $95.8 \pm 11.7$ |
| 17500 | $1.60 \pm 0.34$ | $2.08 \pm 0.06$ | $130 \pm 16$ |
| 20000 | $1.74 \pm 0.25$ | $2.11 \pm 0.04$ | $161 \pm 13$ |
| 22500 | $1.81 \pm 0.26$ | $2.16 \pm 0.04$ | $192 \pm 14$ |
| 25000 | $2.49 \pm 0.47$ | $2.11 \pm 0.06$ | $249 \pm 24$ |
| 27500 | $2.31 \pm 0.50$ | $2.14 \pm 0.06$ | $237 \pm 29$ |
| 30000 | $1.36 \pm 0.49$ | $2.30 \pm 0.10$ | $295 \pm 44$ |
| 32500 | $3.00 \pm 0.94$ | $2.09 \pm 0.09$ | $268 \pm 49$ |



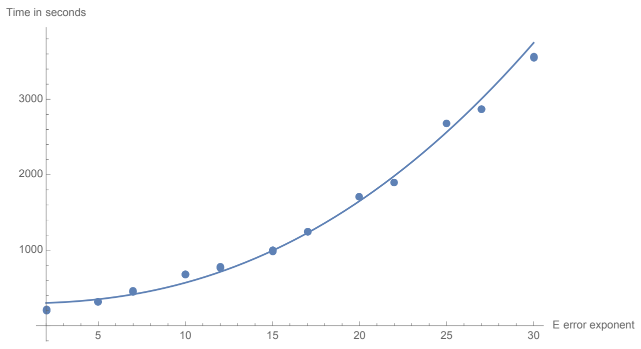
(a) Runtime of the implementation for a fixed number of particles $N = 22500$. The dots are the measured runtime and the line is the fitted function $T_{N=22500}(\mathbb{E}) = \cdot\mathbb{E}$, which is nearly quadratic.



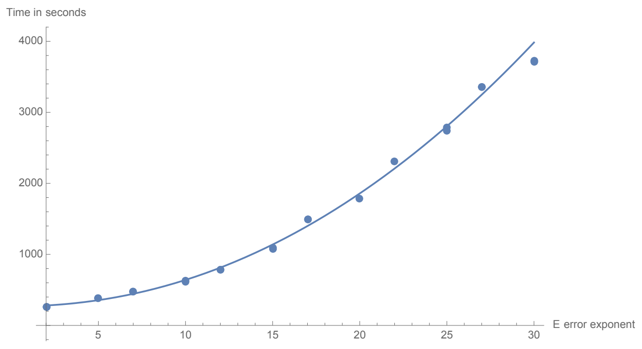
(b) Runtime of the implementation for a fixed number of particles $N = 25000$. The dots are the measured runtime and the line is the fitted function $T_{N=25000}(\mathbb{E}) = \cdot\mathbb{E}$, which is nearly quadratic.



(c) Runtime of the implementation for a fixed number of particles $N = 27500$. The dots are the measured runtime and the line is the fitted function $T_{N=27500}(\mathbb{E}) = \cdot\mathbb{E}$, which is nearly quadratic.



(d) Runtime of the implementation for a fixed number of particles $N = 30000$. The dots are the measured runtime and the line is the fitted function $T_{N=30000}(\mathbb{E}) = \cdot\mathbb{E}$, which is nearly quadratic.



(e) Runtime of the implementation for a fixed number of particles $N = 32500$. The dots are the measured runtime and the line is the fitted function $T_{N=32500}(\mathbb{E}) = \cdot\mathbb{E}$, which is nearly quadratic.

Figure 7.6.: Runtime of the implementation for a fixed number of particles $N = 22500, 25000, 27500, 30000, 32500$ with the measured runtime and the fitted functions $T_{N=*}(\mathbb{E}) = a \cdot \mathbb{E}^b - c$, with $b \approx 2$, which is nearly quadratic.

### 7.2.3 Runtime of the Implementation with a variable Number of Particles and a variable Error Exponent

As seen before the runtime for fixed values of $\mathbb{E}$ seems to be linear in $N$ and for fixed values of $N$ it seems to be quadratic in $\mathbb{E}$, although our estimates have shown that the runtime should be cubic in $\mathbb{E}$. Since our previous estimations only considered the worst case, this difference can be explained, because the equidistant distribution could not be the worst case. To get an overall estimation for the runtime we have chosen the superposition of both functions, i.e. a linear polynomial for fixed $\mathbb{E}$ and a quadratic polynomial for fixed $N$, which results in the following model function

$$T_{\text{seconds}}(N, \mathbb{E}) = a_{1,2} N \cdot \mathbb{E}^2 + a_{1,1} N \cdot \mathbb{E} + a_{1,0} N + a_{0,2} \mathbb{E}^2 + a_{0,1} \mathbb{E} + a_{0,0}.$$

When choosing $N$ in thousands we get the following coefficients

- $a_{1,2} = 0.147 \pm 0.005$,
- $a_{1,1} = -0.308 \pm 0.182$,
- $a_{1,0} = 9.996 \pm 1.267$,
- $a_{0,2} = -0.191 \pm 0.097$,
- $a_{0,1} = -0.357 \pm 3.285$,
- $a_{0,0} = -26.611 \pm 22.771$.

Therefore the runtime in seconds, with the number of particles in thousands $N$ and the error exponent $\mathbb{E}$, can be described by

$$T_{\text{seconds}}(N, \mathbb{E}) = 0.147 N \cdot \mathbb{E}^2 - 0.308 N \cdot \mathbb{E} + 9.996 N - 0.191 \mathbb{E}^2 - 0.357 \mathbb{E} - 26.611.$$

The relative error $\text{err}_{\text{rel}}$ between the fitted function $T_{\text{seconds}}(N, \mathbb{E})$ and the measured values $t_{\text{measured,seconds}}(N, \mathbb{E})$, i.e. $\text{err}_{\text{rel}} = \frac{t_{\text{measured,seconds}}(N,\mathbb{E}) - T_{\text{seconds}}(N,\mathbb{E})}{t_{\text{measured,seconds}}(N,\mathbb{E})}$, is bounded by $\pm 20\%$, i.e. $-20\% < \text{err}_{\text{rel}} < +20\%$. Figure 7.7 shows the fitted function $T_{\text{seconds}}(N, \mathbb{E})$ in orange and the measured values in blue and we see that the fit is quiet good. In order to further justify our fit, we have measured a point far from the measured grid, namely the point 60000 particles and an error exponent of 60. The measured runtime was $t_{\text{measured,seconds}}(60, 60) = 25835s$. The runtime forecasted by our fit is given by $T_{\text{seconds}}(60, 60) = 30575$, which is an overestimation with the relative error $\text{err}_{\text{rel}} = -18\%$, it is also bounded by $\pm 20\%$, like before, which shows the quality of our fit. In practice we will have a small error Exponent, but a lot of particles. Therefore we have also measured points with a lot of particles $N = 200000, 500000, 1000000$ and a small error exponent $\mathbb{E} = 8$. But we have seen that we ran out of memory for the large benchmark and got huge memory problems for the smaller ones, e.g. swapping memory to the harddrive. Therefore we need an algorithm with lesser memory consumption, cf. section 8.1.4. The runtime is given by $t_{\text{measured,seconds}}(200, 8) = 6432s$ and $t_{\text{measured,seconds}}(200, 8) = 18736s$. The discrepancy to the model function can be explained by the usage of the swap and the polylogarithmic terms we have omitted in the fit. But in order to fit the polylogarithmic terms we need way more points in our benchmark grid over a very long range of values.

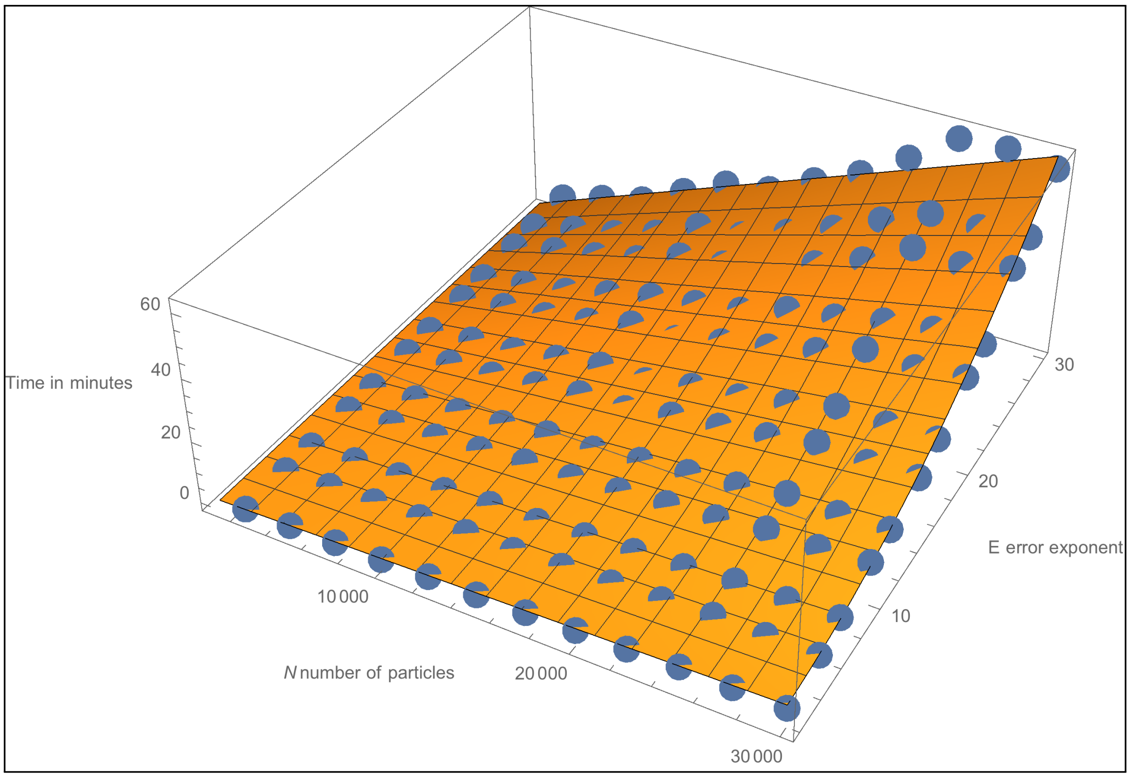


Figure 7.7.: Runtime of our implementation of the Fast Multipole Method.

## 7.3 Memory Consumption of the Implementation with a variable Number of Particles and a variable Error Exponent

The analysis for the memory consumption is similar to the analysis of the runtime. Therefore we omit the analysis here and simply state the results.

The memory consumption for fixed values of $\mathbb{E}$ seems to be linear in $N$ and for fixed values of $N$ it seems to be quadratic in $\mathbb{E}$. To get an overall estimation for the memory consumption we have chosen the superposition of both functions, i.e. a linear polynomial for fixed $\mathbb{E}$ and a quadratic polynomial for fixed $N$, which results in the following model function, which is the same as in section 7.2.3.

$$M_{\mathrm{MB}}(N, \mathbb{E}) = a_{1,2}N \cdot \mathbb{E}^2 + a_{1,1}N \cdot \mathbb{E} + a_{1,0}N + a_{0,2}\mathbb{E}^2 + a_{0,1}\mathbb{E} + a_{0,0}.$$

When choosing $N$ in thousands we get the following coefficients, rounded to three significant digits

- $a_{1,2} = 0.703 \pm 0.017$,
- $a_{1,1} = 9.98 \pm 0.57$,
- $a_{1,0} = 132 \pm 4$,
- $a_{0,2} = -0.257 \pm 0.291$,
- $a_{0,1} = -22.4 \pm 9.6$,
- $a_{0,0} = -276 \pm 65$.

Therefore the memory consumption in MB, with the number of particles in thousands $N$ and the error exponent $\mathbb{E}$, can be described by

$$M_{\mathrm{MB}}(N, \mathbb{E}) = 0.703N \cdot \mathbb{E}^2 9.98N \cdot \mathbb{E} + 132N - 0.257\mathbb{E}^2 - 22.4\mathbb{E} - 276.$$

The relative error $\mathrm{err}_{\mathrm{rel}}$ between the fitted function $M_{\mathrm{MB}}(N, \mathbb{E})$ and the measured values $m_{\mathrm{measured,MB}}(N, \mathbb{E})$, i.e. $\mathrm{err}_{\mathrm{rel}} = \frac{m_{\mathrm{measured,MB}}(N,\mathbb{E}) - M_{\mathrm{MB}}(N,\mathbb{E})}{m_{\mathrm{measured,MB}}(N,\mathbb{E})}$, is bounded by 5%, i.e. $-5\% < \mathrm{err}_{\mathrm{rel}} < +5\%$, which is even better as before. Figure 7.8 shows the fitted function $M_{\mathrm{GB}}(N, \mathbb{E})$ in orange and the measured values in blue and we see that the fit is very good. In order to further

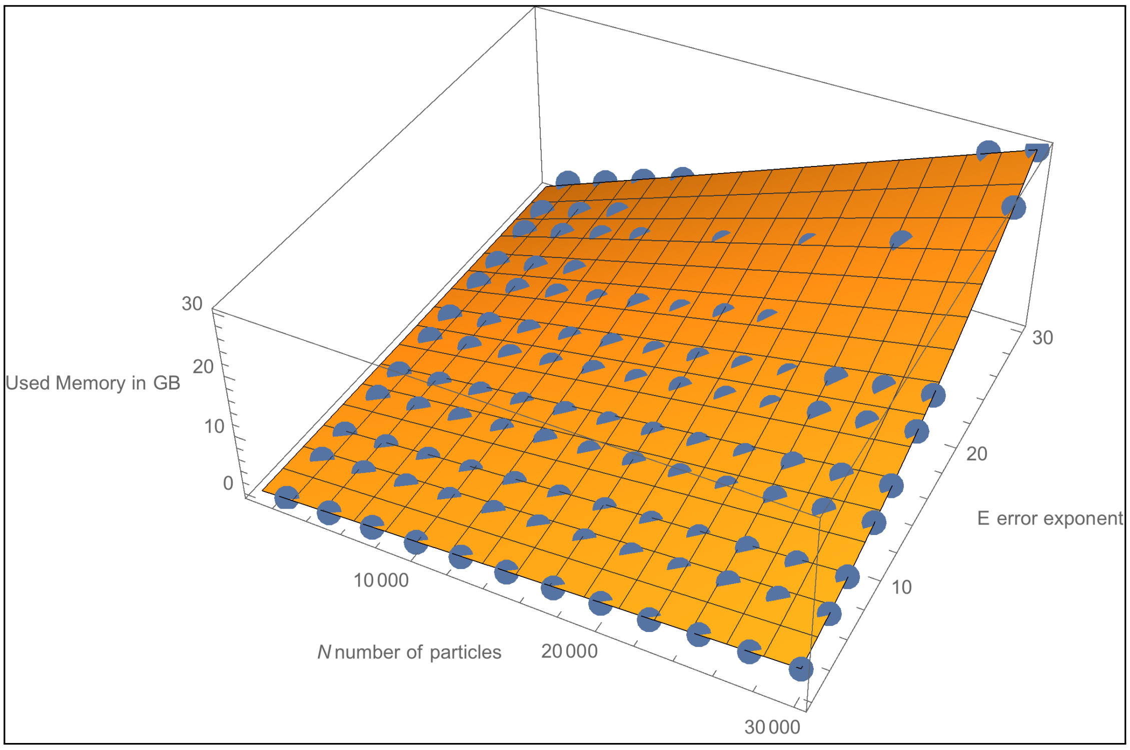


Figure 7.8.: Memory consumption of our implementation of the Fast Multipole Method.

justify our fit, we have measured a point far from the measured grid, namely the point 60000 particles and an error exponent of 60. The measured memory consumption was $m_{\mathrm{measured,MB}}(60, 60) = 167\mathrm{GB}$. The memory consumption forecasted by our fit is given by $M_{\mathrm{MB}}(60, 60) = 189\mathrm{GB}$, which is an overestimation with the relative error $\mathrm{err}_{\mathrm{rel}} = -18\%$, which shows the quality of our fit.

Chapter 8

# Conclusion

In this thesis we have achieved the following

- Showed the lower bound to solve the discrete version of Trummer's Problem is in $\Omega_2(N^2)$.
- Developed an algorithm to solve the real version of Trummer's Problem in $\mathcal{O}_2(N \cdot \mathbb{E}^2)$.
- Showed the lower bound to solve the real version of Trummer's Problem is in $\Omega_2(N \cdot \mathbb{E})$, therefore our algorithm is nearly optimal, besides polylogarithmic terms and an additional factor of $\mathbb{E}$.
- We have implemented the Well-Separated Pair Decomposition and the $\mathcal{O}_2(N \cdot \mathbb{E}^3)$ version of the Fast Multipole Method, with a low implementation complexity, measured using the cyclomatic complexity.
- Our algorithm can be used to solve the N-body problem, where multiple forces occur, e.g. the Coulomb and Debye forces.
- Our benchmarks have shown that the runtime of our implementation is in $\mathcal{O}_2(N \cdot \mathbb{E}^2)$ and the memory consumption is in $\mathcal{O}_2(N \cdot \mathbb{E}^2)$ as well.
- Shown with a minimal example that Trummer's Problem is numerically unstable, hence our exact algorithm is very interesting in practice.

## 8.1 Outlook

In this section we present four extensions to this thesis

- an efficient computation with structured matrices in section 8.1.1,
- the two dimensional real case in section 8.1.2,
- the three dimensional real case in section 8.1.3,
- a memory efficient algorithm to solve Trummer's Problem in section 8.1.4.

### 8.1.1 Efficient Computation with Structured Matrices with Small Displacement Rank

We can easily extend our results to the computation with structured matrices with small displacement rank, cf. section 1.3.2. This should be possible straight forward and unify Tsigaridas and Pan's results [24], i.e. all computations should be linear, independent whether a Cauchy matrix is included or not.

### 8.1.2 Two Dimensional Real Case

In order to extend our algorithm to the two dimensional real case we use complex values, cf. section 1.3.1. In order to encode these values we do the same trick as in section 6.1.2 but we split the complex coefficient vectors $x \in \mathbb{C}^n$ into 4 strictly positive real vectors $x_{+,r}, x_{-,r}, x_{+,i}, x_{-,i} \in \mathbb{R}^n_+$, such that $x = x_{+,r} - x_{-,r} + Ix_{+,i} - Ix_{-,i}$. The multiplication is done as follows

$$\begin{aligned}
x \cdot y =&(x_{+,r} - x_{-,r} + Ix_{+,i} - Ix_{-,i}) \cdot (y_{+,r} - y_{-,r} + Iy_{+,i} - Iy_{-,i})\\
= \quad & x_{+,r} \cdot y_{+,r} - x_{+,r} \cdot y_{-,r} + Ix_{+,r} \cdot y_{+,i} - Ix_{+,r} \cdot y_{-,i}\\
& - x_{-,r} \cdot y_{+,r} + x_{-,r} \cdot y_{-,r} - Ix_{-,r} \cdot y_{+,i} + Ix_{-,r} \cdot y_{-,i}\\
& Ix_{+,i} \cdot y_{+,r} - Ix_{+,i} \cdot y_{-,r} - x_{+,i} \cdot y_{+,i} + x_{+,i} \cdot y_{-,i}\\
& - Ix_{-,i} \cdot y_{+,r} + Ix_{-,i} \cdot y_{-,r} + x_{-,i} \cdot y_{+,i} - x_{-,i} \cdot y_{-,i}\\
= \quad & (x_{+,r} \cdot y_{+,r} + x_{-,r} \cdot y_{-,r} + x_{+,i} \cdot y_{-,i} + x_{-,i} \cdot y_{+,i})\\
- \quad & (x_{+,r} \cdot y_{-,r} + x_{-,r} \cdot y_{+,r} + x_{+,i} \cdot y_{+,i} + x_{-,i} \cdot y_{-,i})\\
& + I(x_{+,r} \cdot y_{+,i} + x_{-,r} \cdot y_{-,i} + x_{+,i} \cdot y_{+,r} + x_{-,i} \cdot y_{-,r})\\
& - I(x_{+,r} \cdot y_{-,i} + x_{-,r} \cdot y_{+,i} + x_{+,i} \cdot y_{-,r} + x_{-,i} \cdot y_{+,r})\\
=&xy_{+,r} - xy_{-,r} + Ixy_{+,i} - Ixy_{-,i}.
\end{aligned}$$

Hence we need 16 multiplications instead of 4 like before. Therefore the runtime will be approximately 4 times slower, but the analysis is the same and the error bounds are the same. This is extension is straightforward.

### 8.1.3 Three Dimensional Real Case

In order to solve the real three dimensional problem all existing approaches choose a expansion into spherical harmonics, cf. [14] for a detailed discussion, instead of our Laurent and Taylor series. The problem is, we would have to do a similar analysis again, we would need a new implementation, etc. Our idea is to use quaternions. Using this approach we can describe up to 4 dimensions. But the problem of the quaternions, is their non-commutativity, nevertheless Taylor and Laurent series are possible, cf. [30]. Using the same trick as before, we will split the coefficient vector into 8 strictly positive positive real vectors, with a similar multiplication scheme. This time we need 64 multiplications, hence the runtime will be approximately 4 times slower than in the complex case. For the analysis and error bounds we suppose they are the same or easily extendable, because in most proofs we only consider the absolute values. Therefore our algorithm is easily extendable up to four real dimensions.

### 8.1.4 Memory efficient algorithm

By choosing a different WSPD, where all well-separated pairs contain a leaf, cf. [5], the number of pairs only grows with an additional factor of $\log N$, which can be neglected. But using this WSPD we can use another algorithm algorithm 8, which should use substantial less memory. Therefore we could solve larger problems with this idea.

**Algorithm 8:** Memory Efficient Fast Multipole Method.

```
// INITIALIZE
Initialize the constant coefficient of the corresponding Laurent series for each leaf;
// DOWNCAST & CONVERSION
Let $U$ be the root;
Do the CONVERSION from all well-separated leaves to $U$;
DOWNCAST $U$ to both children;
Delete $U$ to save memory;
Set $U$ to the deepest children, i.e. use a depth first search over the tree and iterate again;
When we have reached a leaf only save the constant coefficient and delete the others;
// UPCAST & CONVERSION
Let $V$ be a leaf;
UPCAST $V$ and its sibling to their parent;
CONVERSION of this parent to all well-separated leaves (only save the constant coefficient);
Delete $V$ and its sibling again;
Iterate till the complete tree is traversed;
// EVALUATION
Simply return all saved values, i.e. the constant coefficients;
```

Appendix A

# Identities and Estimations

## A.1 Identities and Estimations

**Lemma 39 (Sums).** *Let $q \in \mathbb{C}$ with $|q| \neq 1$, then we get*

$$(1) \quad \sum_{n=a}^{b} q^n = \frac{q^a - q^{b+1}}{1-q} \underbrace{\leq}_{\text{for } |q|<1} \frac{q^a}{1-q},$$

$$(2) \quad \sum_{n=a}^{b} nq^{n-1} = \frac{aq^{a-1} + (1-a)q^a - (b+1)q^b + bq^{b+1}}{(1-q)^2} \underbrace{\leq}_{\text{for } |q|<1} \frac{aq^{a-1} + (1-a)q^a}{(1-q)^2}.$$

*Let $x, y \in \mathbb{C}$, with $|x|, |y| < 1$ and $\alpha \in \mathbb{Z}$, then we get*

$$(3) \quad \sum_{k=-\infty}^{\infty} \binom{\alpha}{k} x^k = (1+x)^\alpha, \quad \text{for } \alpha \in \mathbb{N}, \text{ this is also true for } |x| \geq 1,$$

$$(4) \quad \sum_{n=k}^{\infty} \binom{n}{k} y^n = \frac{y^k}{(1-y)^{k+1}},$$

$$(5) \quad \sum_{n=0}^{\infty} \binom{n+k}{n} x^n = \frac{1}{(1-x)^{k+1}},$$

$$(6) \quad \sum_{n,k} \binom{n+k}{n} x^n y^k = \frac{1}{1-x-y},$$

$$(7) \quad \sum_{n=0}^{\infty} \binom{n+k}{n} nx^{n-1} = \frac{(k+1)}{(1-x)^{k+2}}.$$

*Proof.* The first sum is the geometric series. The second equation is the derivative of the first equation. The third and fourth equation can be found in [34, p. 16]. The fifth and sixth equation can be found in [34, p. 53]. Equation seven is the derivative of the fifth equation. □

**Lemma 40 (Rounding estimation).** *Let $x, y \in \mathbb{R}$. Let $m \in \mathbb{N}$, such that $|x-y| \geq 2^{-m}$. Then*

$$|x-y| + 2^{-m} \geq 2^{-m}\big|[2^m x] - [2^m y]\big| \geq |x-y| - 2^{-m}.$$

*Proof.* Let $x, y \in \mathbb{R}$. Let $m \in \mathbb{N}$, such that $|x-y| \geq 2^{-m}$. Let $\varepsilon_x := [2^m x] - 2^m x, \varepsilon_y := [2^m y] - 2^m y$, then $-\frac{1}{2} \leq \varepsilon_x, \varepsilon_y \leq \frac{1}{2}$ follows.

$$\begin{aligned}
|x-y| + 2^{-m} &\geq 2^{-m}\big(|2^m x - 2^m y| + |\varepsilon_x| + |\varepsilon_y|\big) \\
&\geq 2^{-m}\big|2^m x + \varepsilon_x - 2^m y - \varepsilon_y\big| \\
&= 2^{-m}\big|[2^m x] - [2^m y]\big| \\
&= 2^{-m}|2^m x + \varepsilon_x - 2^m y - \varepsilon_y|
\end{aligned}$$

$$\begin{aligned}
&\geq 2^{-m} \Big| \underbrace{2^m |x-y|}_{\geq 1} - \underbrace{(|\varepsilon_x| + |\varepsilon_y|)}_{\leq 1} \Big| \\
&= |x-y| - 2^{-m}(|\varepsilon_x| + |\varepsilon_y|) \\
&\geq |x-y| - 2^{-m}
\end{aligned}$$
□

**Lemma 41.** *Let* $1 \leq i \leq p \in \mathbb{N}$, *then*

$$\lg \frac{(p-1)!}{(i-1)!} \in \mathcal{O}(p \lg p).$$

*Let* $U, V \in \mathcal{T}_M$

$$\lg(|z_U - z_V|^i) \in \mathcal{O}(p(m_{INIT} + m_{\max(z)})).$$

*holds.*

# List of Algorithms

# List of Figures

# List of Tables

# Bibliography

[1] Arora, Sanjeev ; Barak, Boaz: *Computational Complexity — A Modern Approach*. Cambridge : Cambridge University Press, 2009. – ISBN 978-1-139-47736-9

[2] Bame, Paul: *pmccabe — McCabe-artige Funktionskomplexitäts- und Zeilenzählung für C und C++*. 2018. – URL `https://packages.debian.org/sid/pmccabe`. – Zugriffsdatum: 2018-08-17

[3] Blum, Lenore ; Shub, Mike ; Smale, Steve: On a theory of computation and complexity over the real numbers: $NP$-completeness, recursive functions and universal machines. In: *Bull. Amer. Math. Soc. (N.S.)* 21 (1989), 07, Nr. 1, S. 1–46. – URL `https://projecteuclid.org:443/euclid.bams/1183555121`

[4] Boehm, Barry W.: *Software Engineering Economics -*. New York : Prentice-Hall, 1981. – ISBN 978-0-138-22122-5

[5] Callahan, Paul B.: *Dealing with Higher Dimensions: The Well-Separated Pair Decomposition and Its Applications*. Baltimore, MD, USA, Dissertation, 1995. – UMI Order No. GAX95-33229

[6] Darve, Eric: The Fast Multipole Method: Numerical Implementation. In: *Journal of Computational Physics* 160 (2000), Nr. 1, S. 195 – 240. – URL `http://www.sciencedirect.com/science/article/pii/S0021999100964519`. – ISSN 0021-9991

[7] Dusart, Pierre: Sharper bounds for $\psi, \theta, \pi, p_k$. (1998)

[8] Gerasoulis, A. ; Grigoriadis, M. D. ; Sun, Liping: A Fast Algorithm for Trummer's Problem. In: *SIAM Journal on Scientific and Statistical Computing* 8 (1987), Nr. 1, S. s135–s138. – URL `http://dx.doi.org/10.1137/0908017`

[9] Goldreich, Oded: *Computational Complexity — A Conceptual Perspective*. Cambridge : Cambridge University Press, 2008. – ISBN 978-1-139-47274-6

[10] Golub, Gene: In: *SIGACT News* 17 (1985), Nr. 2. – ISSN 0163-5700

[11] Google: *Google C++ Style Guide*. 2018. – URL `https://google.github.io/styleguide/cppguide.html`. – Zugriffsdatum: 2018-08-17

[12] Granlund, Torbjrn ; Team, Gmp D.: *GNU MP 6.0 Multiple Precision Arithmetic Library*. United Kingdom : Samurai Media Limited, 2015. – ISBN 9789888381968, 9888381962

[13] Greengard, L. ; Rokhlin, V.: On the Efficient Implementation of the Fast Multipole Algorithm. In: *Research Report YALEU/DCS/RR-602* (1988)

[14] Greengard, Leslie: *The Rapid Evaluation of Potential Fields in Particle Systems -*. Cambridge : MIT Press, 1988. – ISBN 978-0-262-07110-9

[15] Inc., Wolfram R.: *Mathematica, Version 11.3*. – Champaign, IL, 2018

[16] Programming languages – C++ / International Organization for Standardization. Geneva, CH, Dezember 2017. – Standard

[17] Kawamura, A. ; Ziegler, M.: Invitation to Real Complexity Theory: Algorithmic Foundations to Reliable Numerics with Bit-Costs. In: *ArXiv e-prints* (2018), Januar

[18] Ko, Ker-I: *Computational Complexity of Real Functions.* S. 40–70. In: *Complexity Theory of Real Functions.* Boston, MA : Birkhäuser Boston, 1991. – URL `https://doi.org/10.1007/978-1-4684-6802-1`. – ISBN 978-1-4684-6802-1

[19] Köhler, Sven ; Ziegler, Martin: On the Stability of Fast Polynomial Arithmetic. In: *Proc. 8th Conference on Real Numbers and Computers*, 1 Juli 2008, S. 147–156

[20] Li, Shaofan ; Wang, Gang: *Introduction to Micromechanics and Nanomechanics.* Singapore : World Scientific Publishing Company, 2017. – ISBN 978-9-814-43678-6

[21] McCabe, T. J.: A Complexity Measure. In: *IEEE Transactions on Software Engineering* SE-2 (1976), Dec, Nr. 4, S. 308–320. – ISSN 0098-5589

[22] Müller, Norbert ; Ziegler, Martin: From Calculus to Algorithms without Errors. In: Hong, Hoon (Hrsg.) ; Yap, Chee (Hrsg.): *Mathematical Software — ICMS 2014*, Springer Berlin Heidelberg, 2014, S. 718–724. – ISBN 978-3-662-44199-2

[23] Narasimhan, Giri ; Smid, Michiel: *Geometric Spanner Networks* -. Cambridge : Cambridge University Press, 2007. – ISBN 978-1-139-46157-3

[24] Pan, Victor Y. ; Tsigaridas, Elias P.: Nearly optimal computations with structured matrices. In: *Theor. Comput. Sci.* 681 (2017), S. 117–137. – URL `https://doi.org/10.1016/j.tcs.2017.03.031`

[25] Papadimitriou, Christos H.: *Computational Complexity* -. Amsterdam : Addison-Wesley, 1994. – ISBN 978-0-020-15308-5

[26] Reif, John H. ; Tate, Stephen R.: N-body Simulation I: Fast Algorithms for Potential Field Evaluation and Trummer's Problem. 1996. – Forschungsbericht

[27] Roques, Arnaud: *Drawing UML with PlantUML — Language Reference Guide.* 2018. – URL `http://plantuml.com/PlantUML_Language_Reference_Guide.pdf`. – Zugriffsdatum: 2018-08-17

[28] Rudin, Walter: *Fourier Analysis on Groups* -. Mineola, New York : Courier Dover Publications, 2017. – ISBN 978-0-486-82101-6

[29] Schönhage, A. ; Strassen, V.: Schnelle Multiplikation großer Zahlen. In: *Computing* 7 (1971), Sep, Nr. 3, S. 281–292. – URL `https://doi.org/10.1007/BF02242355`. – ISSN 1436-5057

[30] Shpakivskyi, Vitalii ; Kuzmenko, Tetyana: Integral theorems for the quaternionic G-monogenic mappings. 24 (2014), 12

[31] Sipser, Michael: *Introduction to the Theory of Computation* -. Clifton Park, NY : Cengage Learning, 2012. – ISBN 128-5-401-069-

[32] Sullivan, F. ; Dongarra, J.: Guest Editors' Introduction: The Top 10 Algorithms. In: *Computing in Science and Engineering* 2 (2000), 01, S. 22–23. – URL `doi.ieeecomputersociety.org/10.1109/MCISE.2000.814652`. – ISSN 1521-9615

[33] Weihrauch, Klaus: *Computable Analysis — An Introduction.* Berlin Heidelberg : Springer Science and Business Media, 2012. – ISBN 978-3-642-56999-9

[34] Wilf, Herbert S.: *Generatingfunctionology.* Academic Press, 1994

Each link in the bibliography was checked at August 26, 2018 for being up-to-date and validity.