An $\mathcal{H}$-Matrix-accelerated Boundary Element Method for Acoustics

Institut für Mathematik und Informatik
der Universität Greifswald

Masterarbeit

zur Erlangung des akademischen Grades
Master of Science

vorgelegt von

Karl Garcke
geboren am 16.01.1993 in Bonn
(Matrikel Nr.: 143464)

am 11.1.2022

Betreuer und Erstgutachter:	Prof. Dr. rer. nat. habil. Roland Pulch
Zweitgutachter:	Prof. Dr. rer. nat. habil. Bernd Kugelmann

Contents

1 Introduction

1.1 Outline

In this paper we study $\mathcal{H}$-matrices as a means to drastically improve the memory and runtime efficiency of the boundary element method (BEM) for the Helmholtz equation. We aim to provide the readers with all the neccessary information to implement an accelerated boundary element method themselves. The text starts out with some general background information on the Helmholtz equation and its BEM. We then build the subject of $\mathcal{H}$-matrices from the ground up. Implementation and performance aspects are discussed using our own implementation as an example.

1.2 Motivation

The boundary element method involves generating a dense matrix and solving the corresponding linear system. The matrix assembly alone introduces quadratic memory and runtime costs. Solving the linear system raises the time complexity to cubic with direct solvers like LU-factorization. These memory and runtime requirements make the conventional BEM infeasible for larger geometries (with more than a few thousand elements). By substituting the ordinary matrix arithmetic with $\mathcal{H}$-matrix arithmetic, we can reduce the memory and time complexity to quasi-linear $O(n * \log(n))$ for low to medium frequencies.

2 Preliminaries

The space that the acoustic medium occupies, will be referred to mathematically as the domain. Throughout this text, the domain is denoted U and is three-dimensional. U is always assumed to be simply connected and closed. ∂U refers to the boundary of the acoustic medium. If we were to simulate the sound field in a closed room for example; ∂U would refer to the floor, walls, ceiling and the surfaces of objects in that room. Vectors are bold, for example, $\mathbf{r} \in U$, except in directional derivatives, e.g. $\frac{\partial}{\partial r}$ and rows and columns of matrices. The vector $\mathbf{p} \in U$ will be referred to as the observation point. The vectors $\mathbf{n}_p$ and $\mathbf{n}_q$ are the unit normals to the boundary at $\mathbf{p}$, $\mathbf{q} \in \partial U$ respectively. The normals point from ∂U into U. Note that in some references, the normals are assumed to point into the opposite direction.

We use a Matlab style notation for the rows and columns of a matrix. The i-th row of a matrix is $A_{i:}$. The j-th column of a matrix is $A_{:j}$. A_{ij} signifies the element of matrix A with row index i and column index j.

3 The conventional BEM for the Helmholtz equation

This chapter is a short summary of the Helmholtz equation and its BEM. For a more detailed introduction see [Kir98] or [Gar17].

3.1 The Helmholtz equation

The Helmholtz equation is the governing equation for frequency domain acoustics.

$$\nabla^2\phi(x) + \kappa^2\phi(x) = 0$$

The constant κ is called the wavenumber. It relates to frequency f and wave speed c as follows

$$\kappa = \frac{2\pi f}{c} = \frac{2\pi}{\lambda}.$$

The function $\phi(x)$ is a scalar velocity potential. We can easily convert $\phi(x)$ to the physical quantities sound pressure $p(x)$ and particle velocity $\mathbf{u}(x)$ as follows:

$$\mathbf{u} = \nabla\phi(x)$$

and

$$p(x) = i\rho\kappa c\phi(x),$$

where ρ is the density of the medium.

3.2 Impedance boundary conditions

For a Helmholtz problem to be *well posed*, we need to supply boundary conditions that relate ϕ and its normal derivative (relative to the boundary) $\frac{\partial\phi(\mathbf{q})}{\partial n_q}$ to each other. The boundary conditions take the general form:

$$a(\mathbf{q})\phi(\mathbf{q}) + b(\mathbf{q})\frac{\partial\phi(\mathbf{q})}{\partial n_q} = f(\mathbf{q}), \quad \mathbf{q} \in \partial U,$$

with a, b and f being complex-valued functions, defined on the boundary. Clearly $a(\mathbf{q}) = b(\mathbf{q}) = 0$ for any $\mathbf{q} \in \partial U$ is not admissible. The most relevant boundary condition for physical modelling is the impedance boundary condition:

$$\frac{\partial\phi(\mathbf{q})}{\partial n_q} + i\kappa\beta\phi(\mathbf{q}) = f(\mathbf{q})$$

β is the relative *surface admittance* of the boundary at the specified frequency (or wavenumber). f is zero for pure scattering problems, and non-zero, where there is a radiating boundary. For $\beta = 0$ and $f = 0$ we have *Neumann* boundary conditions:

$$\frac{\partial \phi(\mathbf{q})}{\partial n_q} = 0 \tag{3.1}$$

Surfaces with *Neumann* boundary conditions are called *sound hard* or fully reflective. As $\frac{\partial \phi(\mathbf{q})}{\partial n_q}$ can be interpreted as the surface particle velocity of the boundary at $\mathbf{q}$, (3.1) states that the particles at $\mathbf{q}$ travel into the boundary with the same velocity as they travel into the opposite direction.

3.3 The fundamental solution

The fundamental solution of the Helmholtz equation is

$$G_\kappa(\mathbf{p}, \mathbf{q}) = \frac{1}{4\pi} \frac{e^{i\kappa \|\mathbf{p}-\mathbf{q}\|_2}}{\| \mathbf{p} - \mathbf{q} \|_2}. \tag{3.2}$$

It is also called the *outgoing wave Green's function.*

Equation (3.2) describes an acoustic point source. The $e^{i\kappa\|\mathbf{p}-\mathbf{q}\|_2}$-term captures the oscillatory part; the $\frac{1}{\|\mathbf{p}-\mathbf{q}\|_2}$ term describes the spreading of acoustic energy over a growing spherical surface. Note that $G_\kappa(\mathbf{p}, \mathbf{q})$ is singular. The singularity will need special attention during numerical integration.

As a fundamental solution $G(\mathbf{p}, \mathbf{q})$ satisfies

$$(\nabla^2 + \kappa^2) G(\mathbf{p}, \mathbf{q}) = -\delta(\mathbf{p} - \mathbf{q}), \tag{3.3}$$

where $\delta(\mathbf{p} - \mathbf{q})$ is the delta distribution.

3.4 The boundary integral formulation of the Helmholtz equation

We very briefly outline the derivation of the boundary integral formulation of the Helmholtz equation. The details can be found in [Gar17]. It holds the following identity:

$$\int_U G(\mathbf{p}, \mathbf{q}) \underbrace{(\nabla^2 + \kappa^2)\, \phi(\mathbf{q})}_{=0} - \underbrace{(\nabla^2 + \kappa^2) G(\mathbf{p}, \mathbf{q})}_{=-\delta(\mathbf{p}-\mathbf{q})} \phi(\mathbf{q})\, \mathrm{d}q = \int_U G(\mathbf{p}, \mathbf{q})\, \nabla^2 \phi(\mathbf{q})\, \mathrm{d}q - \int_U \nabla^2 G(\mathbf{p}, \mathbf{q})\, \phi(\mathbf{q})\, \mathrm{d}q$$

By applying the sampling property of the delta distribution to the left-hand side, and Green's second identity non-rigorously to the right-hand side, it follows directly

$$\phi(\mathbf{p}) = \int_{\partial U} \left(\frac{\partial G(\mathbf{p}, \mathbf{q})}{\partial n_q} \phi(\mathbf{q}) - G(\mathbf{p}, \mathbf{q}) \frac{\partial \phi(\mathbf{q})}{\partial n_q} \right) \mathrm{d}q$$

for $\mathbf{p}$ in the interior of the domain. The vectors $\mathbf{n}_p$ and $\mathbf{n}_q$ are the unit normals to the boundary at $\mathbf{p}$, $\mathbf{q} \in \partial U$, respectively. The normals point from ∂U into U. We refer to $\mathbf{p}$ an observation point from now on. In the context of integral equations $G(\mathbf{p}, \mathbf{q})$ and $\frac{\partial G(\mathbf{p},\mathbf{q})}{\partial n_q}$ are called kernel functions. If we assume $\mathbf{p}$ to be on the boundary of the domain the above boundary integral equation changes to

$$\frac{1}{2}\phi(\mathbf{p}) = \int_{\partial U} \left(\frac{\partial G(\mathbf{p}, \mathbf{q})}{\partial n_q} \phi(\mathbf{q}) - G(\mathbf{p}, \mathbf{q}) \frac{\partial \phi(\mathbf{q})}{\partial n_q} \right) \mathrm{d}q$$

If we want to simulate an incident sound field, like in sound scattering problems, the incident field term is simply added to the right-hand side.

$$\frac{1}{2}\phi(\mathbf{p}) = \phi^{in}(\mathbf{p}) + \int_{\partial U} \left(\frac{\partial G(\mathbf{p}, \mathbf{q})}{\partial n_q} \phi(\mathbf{q}) - G(\mathbf{p}, \mathbf{q}) \frac{\partial \phi(\mathbf{q})}{\partial n_q} \right) \mathrm{d}q \tag{3.4}$$

Equation (3.4) forms the basis for the boundary element method.

3.4.1 The collocation BEM

The boundary integral equation (3.4) is generally not analytically solvable for any but very simple domains. Therefore one resorts to numerical schemes to get to a solution. We follow the simplest ansatz to solve the boundary equation; the so called collocation method. We describe the method constructively.

We discretize the boundary of the domain ∂U into a set of n non-intersecting triangular panels; a triangle mesh. One has to strike a balance between an accurate representation of the boundary geometry and an increased computational effort.

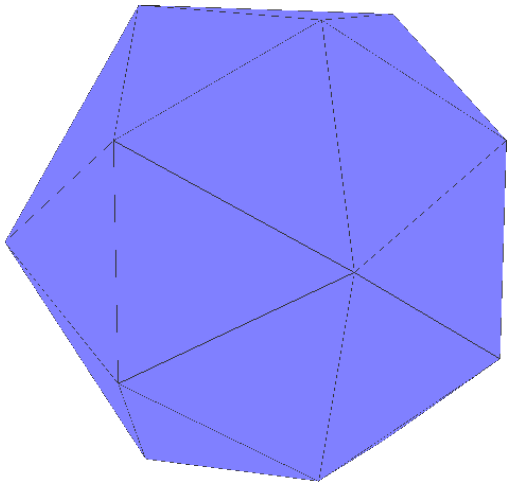


Figure 3.1: Rough approximation of a sphere with 36 triangles

We shall denote the triangle mesh domain $\triangle U$. Also let $I = \{1, \ldots, n\}$ be the index set of the triangular panels. We identify the triangle i by $\triangle_i$. Therefore

$$\partial U \approx \triangle U = \dot{\bigcup_{i=1\ldots n}} \triangle_i$$

The discretization of the domain translates to the boundary integral equation (3.4):

$$\frac{1}{2}\phi(\mathbf{p}) = \phi^{in}(\mathbf{p}) + \sum_{j=1}^{n} \int_{\triangle_j} \left(\frac{\partial G(\mathbf{p}, \mathbf{q})}{\partial n_q} \phi(\mathbf{q}) - G(\mathbf{p}, \mathbf{q}) \frac{\partial \phi(\mathbf{q})}{\partial n_q} \right) \mathrm{d}q \tag{3.5}$$

Physically speaking $\frac{\partial \phi}{\partial n}$ is the surface particle velocity. For the simplicity of following notations we will write $\frac{\partial \phi}{\partial n}$ as υ.

We further simplify equation 3.5 by assuming/enforcing ϕ and υ to be constant over each panel. We define $\boldsymbol{\phi}, \boldsymbol{\upsilon} \in \mathbb{C}^n$ to hold the values of ϕ and υ over the different triangles:

$$\boldsymbol{\phi}_i \coloneqq \phi(\mathbf{q}) \quad \text{and} \quad \boldsymbol{\upsilon}_i \coloneqq \upsilon(\mathbf{q}), \tag{3.6}$$

where $\mathbf{q} \in \triangle_i,\ i \in I$. We define the map $\mathrm{midPoint}(\triangle_j)$ to return the geometric midpoint of $\triangle_j$ for $j \in I$. Let $\mathbf{p}_i = \mathrm{midPoint}(\triangle_i)$. We call $\mathbf{p}_i$ a collocation point. Now we can write the discretised boundary integral equation for $\mathbf{p}_i$ with $\boldsymbol{\phi}$ and $\boldsymbol{\upsilon}$ factorized out of the integral:

$$\frac{1}{2}\phi(\mathbf{p}_i) = \frac{1}{2}\boldsymbol{\phi}_i = \phi^{in}(\mathbf{p}_i) + \sum_{j=1}^{n} \int_{\triangle_j} \frac{\partial G(\mathbf{p}_i, \mathbf{q})}{\partial n_q} \,\mathrm{d}q\, \boldsymbol{\phi}_j - \sum_{j=1}^{n} \int_{\triangle_j} G(\mathbf{p}_i, \mathbf{q}) \,\mathrm{d}q\, \boldsymbol{\upsilon}_j$$

We rearrange the equation by use of the Kronecker delta:

$$\sum_{j=1}^{n} \int_{\triangle_j} G(\mathbf{p}_i, \mathbf{q}) \,\mathrm{d}q\, \boldsymbol{\upsilon}_j = \phi^{in}(\mathbf{p}_i) + \sum_{j=1}^{n} \left(\int_{\triangle_j} \frac{\partial G(\mathbf{p}_i, \mathbf{q})}{\partial n_q} \,\mathrm{d}q - \delta_{ij} \frac{1}{2} \right) \boldsymbol{\phi}_j,$$

Now we have one summation equation with $2n$ unknowns. We can set up the boundary equation for each triangle midpoint. This leads directly to a system of n equations for the n collocation points.

$$\begin{pmatrix} \int_{\triangle_1} G(\mathbf{p}_1, \mathbf{q}) \,\mathrm{d}q & \cdots & \int_{\triangle_n} G(\mathbf{p}_1, \mathbf{q}) \,\mathrm{d}q \\ \vdots & \ddots & \vdots \\ \int_{\triangle_1} G(\mathbf{p}_n, \mathbf{q}) \,\mathrm{d}q & \cdots & \int_{\triangle_n} G(\mathbf{p}_n, \mathbf{q}) \,\mathrm{d}q \end{pmatrix} \begin{pmatrix} \boldsymbol{\upsilon}_1 \\ \vdots \\ \boldsymbol{\upsilon}_n \end{pmatrix}$$

$$= \begin{pmatrix} \boldsymbol{\phi}^{in}(\mathbf{p}_1) \\ \vdots \\ \boldsymbol{\phi}^{in}(\mathbf{p}_n) \end{pmatrix} + \left(\begin{pmatrix} \int_{\triangle_1} \frac{\partial G(\mathbf{p}_1, \mathbf{q})}{\partial n_q} \,\mathrm{d}q & \cdots & \int_{\triangle_n} \frac{\partial G(\mathbf{p}_1, \mathbf{q})}{\partial n_q} \,\mathrm{d}q \\ \vdots & \ddots & \vdots \\ \int_{\triangle_1} \frac{\partial G(\mathbf{p}_n, \mathbf{q})}{\partial n_q} \,\mathrm{d}q & \cdots & \int_{\triangle_n} \frac{\partial G(\mathbf{p}_n, \mathbf{q})}{\partial n_q} \,\mathrm{d}q \end{pmatrix} - \frac{1}{2} I \right) \begin{pmatrix} \boldsymbol{\phi}_1 \\ \vdots \\ \boldsymbol{\phi}_n \end{pmatrix},$$

where $p_i = \text{midPoint}(\triangle_i)$ and I is the $n \times n$ identity matrix. We can describe the matrices as a discretization of the kernel integral equations. To bring the system into a solvable form, we need to exploit the discretized boundary conditions:

$$\boldsymbol{a}_i \boldsymbol{\phi}_i + \boldsymbol{b}_i \boldsymbol{v}_i = \boldsymbol{f}_i \quad \text{for } i \in I$$

3.4.2 The substituted system

Depending on the boundary condition, we substitute. If $\boldsymbol{a}_i \geq \boldsymbol{b}_i$:

$$\boldsymbol{\phi}_i = \frac{\boldsymbol{f}_i - \boldsymbol{b}_i \boldsymbol{v}_i}{\boldsymbol{a}_i}.$$

Else if $\boldsymbol{b}_i > \boldsymbol{a}_i$:

$$\boldsymbol{v}_i = \frac{\boldsymbol{f}_i - \boldsymbol{a}_i \boldsymbol{\phi}_i}{\boldsymbol{b}_i}.$$

Unfortunately an explicit representation of the linear system with substitutions would be too congested. We motivate the substitutions with a simple example. Let the unsubstituted system be

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} \begin{pmatrix} \boldsymbol{v}_1 \\ \boldsymbol{v}_2 \\ \boldsymbol{v}_3 \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} & C_{13} \\ C_{21} & C_{22} & C_{23} \\ C_{31} & C_{32} & C_{33} \end{pmatrix} \begin{pmatrix} \boldsymbol{\phi}_1 \\ \boldsymbol{\phi}_2 \\ \boldsymbol{\phi}_3 \end{pmatrix} + \begin{pmatrix} \boldsymbol{\phi}_1^{in} \\ \boldsymbol{\phi}_2^{in} \\ \boldsymbol{\phi}_3^{in} \end{pmatrix},$$

with boundary conditions, where $\boldsymbol{a}_1 \geq \boldsymbol{b}_1$, $\boldsymbol{a}_2 < \boldsymbol{b}_2$ and $\boldsymbol{a}_3 \geq \boldsymbol{b}_3$. Now we use the substitutions:

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} \begin{pmatrix} \boldsymbol{v}_1 \\ \frac{\boldsymbol{f}_2 - \boldsymbol{a}_2 \boldsymbol{\phi}_2}{\boldsymbol{b}_2} \\ \boldsymbol{v}_3 \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} & C_{13} \\ C_{21} & C_{22} & C_{23} \\ C_{31} & C_{32} & C_{33} \end{pmatrix} \begin{pmatrix} \frac{\boldsymbol{f}_1 - \boldsymbol{b}_1 \boldsymbol{v}_1}{\boldsymbol{a}_1} \\ \boldsymbol{\phi}_2 \\ \frac{\boldsymbol{f}_3 - \boldsymbol{b}_3 \boldsymbol{v}_3}{\boldsymbol{a}_3} \end{pmatrix} + \begin{pmatrix} \boldsymbol{\phi}_1^{in} \\ \boldsymbol{\phi}_2^{in} \\ \boldsymbol{\phi}_3^{in} \end{pmatrix}$$

The system is now reduced to n unknowns. We rearrange the system, so that all the substitution terms are on the right-hand side:

$$\begin{pmatrix} A_{11} & -C_{12} & A_{13} \\ A_{21} & -C_{22} & A_{23} \\ A_{31} & -C_{32} & A_{33} \end{pmatrix} \begin{pmatrix} \boldsymbol{v}_1 \\ \boldsymbol{\phi}_2 \\ \boldsymbol{v}_3 \end{pmatrix} = \begin{pmatrix} C_{11} & -A_{12} & C_{13} \\ C_{21} & -A_{22} & C_{23} \\ C_{31} & -A_{32} & C_{33} \end{pmatrix} \begin{pmatrix} \frac{\boldsymbol{f}_1 - \boldsymbol{b}_1 \boldsymbol{v}_1}{\boldsymbol{a}_1} \\ \frac{\boldsymbol{f}_2 - \boldsymbol{a}_2 \boldsymbol{\phi}_2}{\boldsymbol{b}_2} \\ \frac{\boldsymbol{f}_3 - \boldsymbol{b}_3 \boldsymbol{v}_3}{\boldsymbol{a}_3} \end{pmatrix} + \begin{pmatrix} \boldsymbol{\phi}_1^{in} \\ \boldsymbol{\phi}_2^{in} \\ \boldsymbol{\phi}_3^{in} \end{pmatrix}$$

Finally we transform the system, so that the right-hand side is only dependent on the vector $\boldsymbol{f}$ and not on $\boldsymbol{\phi}$ nor $\boldsymbol{v}$:

$$\begin{aligned} &\begin{pmatrix} A_{11} + \frac{\boldsymbol{b}_1}{\boldsymbol{a}_1} C_{11} & -C_{12} - \frac{\boldsymbol{a}_2}{\boldsymbol{b}_2} A_{12} & A_{13} + \frac{\boldsymbol{b}_3}{\boldsymbol{a}_3} C_{13} \\ A_{21} + \frac{\boldsymbol{b}_1}{\boldsymbol{a}_1} C_{21} & -C_{22} - \frac{\boldsymbol{a}_2}{\boldsymbol{b}_2} A_{22} & A_{23} + \frac{\boldsymbol{b}_3}{\boldsymbol{a}_3} C_{23} \\ A_{31} + \frac{\boldsymbol{b}_1}{\boldsymbol{a}_1} C_{31} & -C_{32} - \frac{\boldsymbol{a}_2}{\boldsymbol{b}_2} A_{32} & A_{33} + \frac{\boldsymbol{b}_3}{\boldsymbol{a}_3} C_{33} \end{pmatrix} \begin{pmatrix} \boldsymbol{v}_1 \\ \boldsymbol{\phi}_2 \\ \boldsymbol{v}_3 \end{pmatrix} \\ &= \begin{pmatrix} C_{11}/\boldsymbol{a}_1 & -A_{12}/\boldsymbol{b}_2 & C_{13}/\boldsymbol{a}_3 \\ C_{21}/\boldsymbol{a}_1 & -A_{22}/\boldsymbol{b}_2 & C_{23}/\boldsymbol{a}_3 \\ C_{31}/\boldsymbol{a}_1 & -A_{32}/\boldsymbol{b}_2 & C_{33}/\boldsymbol{a}_3 \end{pmatrix} \begin{pmatrix} \boldsymbol{f}_1 \\ \boldsymbol{f}_2 \\ \boldsymbol{f}_3 \end{pmatrix} + \begin{pmatrix} \boldsymbol{\phi}_1^{in} \\ \boldsymbol{\phi}_2^{in} \\ \boldsymbol{\phi}_3^{in} \end{pmatrix} \end{aligned} \tag{3.7}$$

The vector $\boldsymbol{f}$ is known. The right-hand side of equation (3.7) is therefore evaluated as a standard matrix-vector product. Then we have a linear system in the form $A\mathbf{x} = \mathbf{b}$. The eventual solution is then extracted from $\mathbf{x}$ by substituting back. For simplicity of notation we assume that the substitutions are handled implicitly throughout the remainder of this text. When we speak of discretizations of kernel function integrals in later part of this text, we implicitly mean linear combinations in the fashion of equation (3.7), that conform to the substitutions via the boundary conditions.

3.4.3 The Burton Miller formulation

Though we have a solvable system, we are not yet at the final formulation for the boundary element method. Unfortunately the original Helmholtz boundary equation formulation (equation 3.4) suffers from the so-called *non-uniqueness problem* for exterior domain problems. Regardless of the imposed boundary conditions, the Helmholtz integral equation (3.4) does not yield a unique solution at the eigenfrequencies of the corresponding interior Dirichlet problem [MW08, page 412]. Though physically unconnected, a resonance in the interior problem pollutes the solution of the exterior problem. These characteristic frequencies of the interior Dirichlet problem are therefore also termed *fictitious* eigenfrequencies. The non-uniqueness problem is not due to a specific numerical implementation, but inherent to the integral formulation. The interior problem itself is not affected by this issue as the eigenfrequencies of the interior problem are the 'real' eigenfrequencies of the interior domain [Juh94, page 106].

The Burton and Miller[BM71] approach to the non-uniqueness problem is to find a solution, that solves a linear combination of equation (3.4) and the directional derivative of (3.4) with regards to the normal to the boundary at $\mathbf{p}$. The argument is, that both formulations have fictitious eigenfrequencies, but that their fictitious eigenfrequencies do not overlap.

Note that (3.4) is a function of $\mathbf{p}$. Therefore the directional derivative of the boundary equation is the following:

$$\frac{1}{2}\frac{\partial\phi(\mathbf{p})}{\partial n_p} = \frac{1}{2}\upsilon(\mathbf{p}) = \frac{\partial\phi^{in}(\mathbf{p})}{\partial n_p} + \int_{\partial U}\left(\frac{\partial^2 G(\mathbf{p},\mathbf{q})}{\partial n_q \partial n_p}\phi(\mathbf{q}) - \frac{\partial G(\mathbf{p},\mathbf{q})}{\partial n_p}\upsilon(\mathbf{q})\right)\mathrm{d}q \tag{3.8}$$

The linear combination of (3.4) and (3.8) is the Burton-Miller combined boundary equation [CCH09, page 165]:

$$\begin{aligned}\frac{1}{2}\phi(\mathbf{p}) + \alpha\frac{1}{2}\upsilon(\mathbf{p}) =& \phi^{in}(\mathbf{p}) + \alpha\frac{\partial\phi^{in}(\mathbf{p})}{\partial n_p} \\ &+ \int_{\partial U}\left(\frac{\partial G(\mathbf{p},\mathbf{q})}{\partial n_q}\phi(\mathbf{q}) - G(\mathbf{p},\mathbf{q})\frac{\partial\phi(\mathbf{q})}{\partial n_q}\right)\mathrm{d}q \\ &+ \alpha\int_{\partial U}\left(\frac{\partial^2 G(\mathbf{p},\mathbf{q})}{\partial n_q \partial n_p}\phi(\mathbf{q}) - \frac{\partial G(\mathbf{p},\mathbf{q})}{\partial n_p}\upsilon(\mathbf{q})\right)\mathrm{d}q\end{aligned} \tag{3.9}$$

with coupling parameter $\alpha \in \mathbb{C}$.
Most of the encountered literature gives $\alpha = \frac{i}{\kappa}$ as an optimal choice (See [ZCGD15, page 50] for reference).

3.4.4 The combined linear system

As a preliminary to this section, we note that the kernel functions, from which the BEM system matrices arise, are singular at $\mathbf{p} = \mathbf{q}$, which occurs on the diagonal elements of the matrices. The integrals in the diagonal elements are transformed in [Gar17] to make them amenable to standard numerical integration techniques. We have included the results directly into the matrix formulations. On that account we define helper functions $R_i(\theta) : [0, 2\pi] \to \mathbb{R}^+$ for $i \in I$. The mapping $R_i(\theta)$ returns the distance from the collocation point $\mathbf{p}_i$ to the border of the triangle $\triangle_i$ as a function of the angle θ.

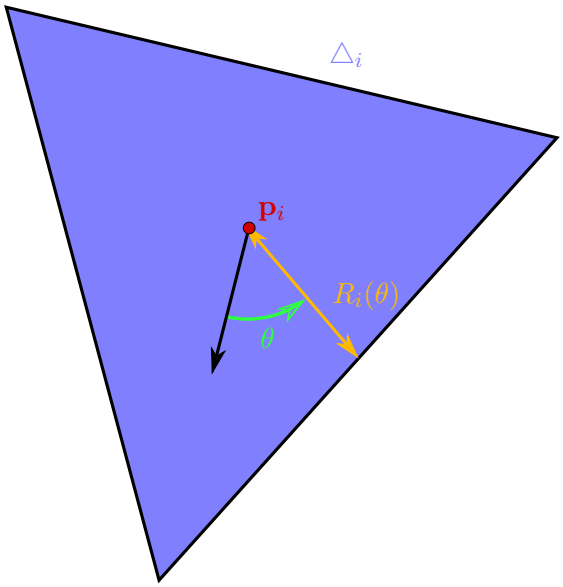


Figure 3.2: $R_i(\theta)$ on $\triangle_i$

Following the same discretization method as in section (3.4.1), we arrive at the Burton-Miller combined boundary equation system:

$$(A + \alpha B)\,\boldsymbol{v} = (C + \alpha D)\,\boldsymbol{\phi} + \left(\boldsymbol{\phi}^{in} + \alpha \boldsymbol{v}^{in}\right) \tag{3.10}$$

with A, B, C and $D \in \mathbb{C}^{n \times n}$ and $\boldsymbol{\phi}$, $\boldsymbol{v}$, $\boldsymbol{\phi}^{in}$ and $\boldsymbol{v}^{in} \in \mathbb{C}^n$.

$$\boldsymbol{\phi}^{in}_{vec} \coloneqq \begin{pmatrix} \phi^{in}(\mathbf{p}_1) \\ \vdots \\ \phi^{in}(\mathbf{p}_n) \end{pmatrix} \qquad \boldsymbol{v}^{in}_{vec} \coloneqq \begin{pmatrix} \frac{\partial \phi^{in}(\mathbf{p}_1)}{\partial n_{p_1}} \\ \vdots \\ \frac{\partial \phi^{in}(\mathbf{p}_n)}{\partial n_{p_n}} \end{pmatrix}$$

The matrices are explicitly given in the following.

$$A \coloneqq \begin{pmatrix} \frac{i}{2\kappa} - \frac{i}{4\pi\kappa}\int_0^{2\pi} e^{i\kappa R_1(\theta)}\,\mathrm{d}\theta & \int_{\triangle_2} G(\mathbf{p}_1,\mathbf{q})\,\mathrm{d}q & \dots & \int_{\triangle_{n-1}} G(\mathbf{p}_1,\mathbf{q})\,\mathrm{d}q & \int_{\triangle_n} G(\mathbf{p}_1,\mathbf{q})\,\mathrm{d}q \\ \int_{\triangle_1} G(\mathbf{p}_2,\mathbf{q})\,\mathrm{d}q & \frac{i}{2\kappa} - \frac{i}{4\pi\kappa}\int_0^{2\pi} e^{i\kappa R_2(\theta)}\,\mathrm{d}\theta & \ddots & \int_{\triangle_{n-1}} G(\mathbf{p}_2,\mathbf{q})\,\mathrm{d}q & \int_{\triangle_n} G(\mathbf{p}_2,\mathbf{q})\,\mathrm{d}q \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \int_{\triangle_1} G(\mathbf{p}_{n-1},\mathbf{q})\,\mathrm{d}q & \int_{\triangle_2} G(\mathbf{p}_{n-1},\mathbf{q})\,\mathrm{d}q & \ddots & \frac{i}{2\kappa} - \frac{i}{4\pi\kappa}\int_0^{2\pi} e^{i\kappa R_{n-1}(\theta)}\,\mathrm{d}\theta & \int_{\triangle_n} G(\mathbf{p}_{n-1},\mathbf{q})\,\mathrm{d}q \\ \int_{\triangle_1} G(\mathbf{p}_n,\mathbf{q})\,\mathrm{d}q & \int_{\triangle_2} G(\mathbf{p}_n,\mathbf{q})\,\mathrm{d}q & \dots & \int_{\triangle_{n-1}} G(\mathbf{p}_n,\mathbf{q})\,\mathrm{d}q & \frac{i}{2\kappa} - \frac{i}{4\pi\kappa}\int_0^{2\pi} e^{i\kappa R_n(\theta)}\,\mathrm{d}\theta \end{pmatrix} \tag{3.11}$$

$$B \coloneqq \begin{pmatrix} \frac{1}{2} & \int_{\triangle_2} \frac{\partial G(\mathbf{p}_1,\mathbf{q})}{\partial n_{p_1}}\,\mathrm{d}q & \dots & \int_{\triangle_{n-1}} \frac{\partial G(\mathbf{p}_1,\mathbf{q})}{\partial n_{p_1}}\,\mathrm{d}q & \int_{\triangle_n} \frac{\partial G(\mathbf{p}_1,\mathbf{q})}{\partial n_{p_1}}\,\mathrm{d}q \\ \int_{\triangle_1} \frac{\partial G(\mathbf{p}_2,\mathbf{q})}{\partial n_{p_2}}\,\mathrm{d}q & \frac{1}{2} & \ddots & \int_{\triangle_{n-1}} \frac{\partial G(\mathbf{p}_2,\mathbf{q})}{\partial n_{p_2}}\,\mathrm{d}q & \int_{\triangle_n} \frac{\partial G(\mathbf{p}_2,\mathbf{q})}{\partial n_{p_2}}\,\mathrm{d}q \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \int_{\triangle_1} \frac{\partial G(\mathbf{p}_{n-1},\mathbf{q})}{\partial n_{p_{n-1}}}\,\mathrm{d}q & \int_{\triangle_2} \frac{\partial G(\mathbf{p}_{n-1},\mathbf{q})}{\partial n_{p_{n-1}}}\,\mathrm{d}q & \ddots & \frac{1}{2} & \int_{\triangle_n} \frac{\partial G(\mathbf{p}_{n-1},\mathbf{q})}{\partial n_{p_{n-1}}}\,\mathrm{d}q \\ \int_{\triangle_1} \frac{\partial G(\mathbf{p}_n,\mathbf{q})}{\partial n_{p_n}}\,\mathrm{d}q & \int_{\triangle_2} \frac{\partial G(\mathbf{p}_n,\mathbf{q})}{\partial n_{p_n}}\,\mathrm{d}q & \dots & \int_{\triangle_{n-1}} \frac{\partial G(\mathbf{p}_n,\mathbf{q})}{\partial n_{p_n}}\,\mathrm{d}q & \frac{1}{2} \end{pmatrix}$$

$$C \coloneqq \begin{pmatrix} -\frac{1}{2} & \int_{\triangle_2} \frac{\partial G(\mathbf{p}_1,\mathbf{q})}{\partial n_q}\,\mathrm{d}q & \dots & \int_{\triangle_{n-1}} \frac{\partial G(\mathbf{p}_1,\mathbf{q})}{\partial n_q}\,\mathrm{d}q & \int_{\triangle_n} \frac{\partial G(\mathbf{p}_1,\mathbf{q})}{\partial n_q}\,\mathrm{d}q \\ \int_{\triangle_1} \frac{\partial G(\mathbf{p}_2,\mathbf{q})}{\partial n_q}\,\mathrm{d}q & -\frac{1}{2} & \ddots & \int_{\triangle_{n-1}} \frac{\partial G(\mathbf{p}_2,\mathbf{q})}{\partial n_q}\,\mathrm{d}q & \int_{\triangle_n} \frac{\partial G(\mathbf{p}_2,\mathbf{q})}{\partial n_q}\,\mathrm{d}q \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \int_{\triangle_1} \frac{\partial G(\mathbf{p}_{n-1},\mathbf{q})}{\partial n_q}\,\mathrm{d}q & \int_{\triangle_2} \frac{\partial G(\mathbf{p}_{n-1},\mathbf{q})}{\partial n_q}\,\mathrm{d}q & \ddots & -\frac{1}{2} & \int_{\triangle_n} \frac{\partial G(\mathbf{p}_{n-1},\mathbf{q})}{\partial n_q}\,\mathrm{d}q \\ \int_{\triangle_1} \frac{\partial G(\mathbf{p}_n,\mathbf{q})}{\partial n_q}\,\mathrm{d}q & \int_{\triangle_2} \frac{\partial G(\mathbf{p}_n,\mathbf{q})}{\partial n_q}\,\mathrm{d}q & \dots & \int_{\triangle_{n-1}} \frac{\partial G(\mathbf{p}_n,\mathbf{q})}{\partial n_q}\,\mathrm{d}q & -\frac{1}{2} \end{pmatrix}$$

$$D \coloneqq \begin{pmatrix} \frac{i\kappa}{2} - \int_0^{2\pi} \frac{e^{i\kappa R_1(\theta)}}{4\pi R_1(\theta)}\,\mathrm{d}\theta & \int_{\triangle_2} \frac{\partial^2 G(\mathbf{p}_1,\mathbf{q})}{\partial n_q \partial n_{p_1}}\,\mathrm{d}q & \dots & \int_{\triangle_{n-1}} \frac{\partial^2 G(\mathbf{p}_1,\mathbf{q})}{\partial n_q \partial n_{p_1}}\,\mathrm{d}q & \int_{\triangle_n} \frac{\partial^2 G(\mathbf{p}_1,\mathbf{q})}{\partial n_q \partial n_{p_1}}\,\mathrm{d}q \\ \int_{\triangle_1} \frac{\partial^2 G(\mathbf{p}_2,\mathbf{q})}{\partial n_q \partial n_{p_2}}\,\mathrm{d}q & \frac{i\kappa}{2} - \int_0^{2\pi} \frac{e^{i\kappa R_2(\theta)}}{4\pi R_2(\theta)}\,\mathrm{d}\theta & \ddots & \int_{\triangle_{n-1}} \frac{\partial^2 G(\mathbf{p}_2,\mathbf{q})}{\partial n_q \partial n_{p_2}}\,\mathrm{d}q & \int_{\triangle_n} \frac{\partial^2 G(\mathbf{p}_2,\mathbf{q})}{\partial n_q \partial n_{p_2}}\,\mathrm{d}q \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \int_{\triangle_1} \frac{\partial^2 G(\mathbf{p}_{n-1},\mathbf{q})}{\partial n_q \partial n_{p_{n-1}}}\,\mathrm{d}q & \int_{\triangle_2} \frac{\partial^2 G(\mathbf{p}_{n-1},\mathbf{q})}{\partial n_q \partial n_{p_{n-1}}}\,\mathrm{d}q & \ddots & \frac{i\kappa}{2} - \int_0^{2\pi} \frac{e^{i\kappa R_{n-1}(\theta)}}{4\pi R_{n-1}(\theta)}\,\mathrm{d}\theta & \int_{\triangle_n} \frac{\partial^2 G(\mathbf{p}_{n-1},\mathbf{q})}{\partial n_q \partial n_{p_{n-1}}}\,\mathrm{d}q \\ \int_{\triangle_1} \frac{\partial^2 G(\mathbf{p}_n,\mathbf{q})}{\partial n_q \partial n_{p_n}}\,\mathrm{d}q & \int_{\triangle_2} \frac{\partial^2 G(\mathbf{p}_n,\mathbf{q})}{\partial n_q \partial n_{p_n}}\,\mathrm{d}q & \dots & \int_{\triangle_{n-1}} \frac{\partial^2 G(\mathbf{p}_n,\mathbf{q})}{\partial n_q \partial n_{p_n}}\,\mathrm{d}q & \frac{i\kappa}{2} - \int_0^{2\pi} \frac{e^{i\kappa R_n(\theta)}}{4\pi R_n(\theta)}\,\mathrm{d}\theta \end{pmatrix}$$

The above system is our complete BEM formulation. Again we need to use substitutions via the boundary conditions (as described in section 3.4.2) to solve our boundary element system with standard methods.

Once the boundary states have been solved for, the solution for a point $\mathbf{p}$ in the interior of the domain can be acquired via:

$$\phi(\mathbf{p}) = \phi^{in}(\mathbf{p}) + \sum_{j=1}^{n} \int_{\triangle_j} \left(\frac{\partial G(\mathbf{p},\mathbf{q})}{\partial n_q} \phi_j - G(\mathbf{p},\mathbf{q}) \upsilon_j \right) \mathrm{d}q \tag{3.12}$$

Note that $p \notin \triangle U$ for (3.12), therefore all integrals can be solved directly via standard Gaussian quadrature methods for planar triangles.

We finish the chapter by giving all relevant directional derivatives of $G(\mathbf{p},\mathbf{q})$ to actually compute the BEM system.

3.4.5 The derivatives of the fundamental solution

We introduce the following definitions to enable us to break down the derivatives of G:

$$\mathbf{r}(\mathbf{p},\mathbf{q}) \coloneqq \mathbf{p} - \mathbf{q}$$

$$r(\mathbf{p},\mathbf{q}) \coloneqq \| \mathbf{r}(\mathbf{p},\mathbf{q}) \|_2$$

Now G with regard to $r(\mathbf{p},\mathbf{q})$ is:

$$G_\kappa(r(\mathbf{p},\mathbf{q})) = \frac{1}{4\pi} \frac{e^{i\kappa r(\mathbf{p},\mathbf{q})}}{r(\mathbf{p},\mathbf{q})}$$

Normal derivatives of r

As stated earlier, the vectors $\mathbf{n}_p$ and $\mathbf{n}_q$ are the unit normals to the boundary at $\mathbf{p}$ and $\mathbf{q} \in \partial U$ respectively. The normals always point from ∂U *into* U. Note that in all instances where $\frac{\partial r(\mathbf{p},\mathbf{q})}{\partial n_q}$ will be used, the variable of differentiation is $\mathbf{q}$. Therefore:

$$\frac{\partial r(\mathbf{p},\mathbf{q})}{\partial n_q} = -\frac{\mathbf{r}(\mathbf{p},\mathbf{q}) \cdot \mathbf{n}_q}{r(\mathbf{p},\mathbf{q})}$$

The situation is the other way around with $\frac{\partial r(\mathbf{p},\mathbf{q})}{\partial n_p}$.

$$\frac{\partial r(\mathbf{p},\mathbf{q})}{\partial n_p} = \frac{\mathbf{r}(\mathbf{p},\mathbf{q}) \cdot \mathbf{n}_p}{r(\mathbf{p},\mathbf{q})}$$

$$\frac{\partial^2 r(\mathbf{p},\mathbf{q})}{\partial n_p \partial n_q} = -\frac{1}{r(\mathbf{p},\mathbf{q})} \left(\mathbf{n}_p \cdot \mathbf{n}_q + \frac{\partial r(\mathbf{p},\mathbf{q})}{\partial n_q} \frac{\partial r(\mathbf{p},\mathbf{q})}{\partial n_p} \right)$$

Derivatives of G(r)

$$G_\kappa(r(\mathbf{p},\mathbf{q})) = \frac{1}{4\pi}\frac{e^{i\kappa r(\mathbf{p},\mathbf{q})}}{r(\mathbf{p},\mathbf{q})}$$

$$\frac{\partial G_\kappa(r(\mathbf{p},\mathbf{q}))}{\partial r} = \frac{G_\kappa(r(\mathbf{p},\mathbf{q}))}{r(\mathbf{p},\mathbf{q})}(i\kappa r(\mathbf{p},\mathbf{q}) - 1)$$

$$\frac{\partial^2 G_\kappa(r(\mathbf{p},\mathbf{q}))}{\partial r^2} = \frac{G_\kappa(r(\mathbf{p},\mathbf{q}))}{r(\mathbf{p},\mathbf{q})^2}(2 - 2i\kappa r(\mathbf{p},\mathbf{q}) - \kappa^2 r(\mathbf{p},\mathbf{q})^2)$$

Normal derivatives of the Greens function

Finally the normal derivatives of $G(\mathbf{p},\mathbf{q})$, expressed through the terms of the previous two subsections:

$$\frac{\partial G_\kappa(r(\mathbf{p},\mathbf{q}))}{\partial n_q} = \frac{\partial G_\kappa(r(\mathbf{p},\mathbf{q}))}{\partial r}\frac{\partial r(\mathbf{p},\mathbf{q})}{\partial n_q}$$

$$\frac{\partial G_\kappa(r(\mathbf{p},\mathbf{q}))}{\partial n_p} = \frac{\partial G_\kappa(r(\mathbf{p},\mathbf{q}))}{\partial r}\frac{\partial r(\mathbf{p},\mathbf{q})}{\partial n_p}$$

$$\frac{\partial^2 G_\kappa(r(\mathbf{p},\mathbf{q}))}{\partial n_p \partial n_q} = \frac{\partial G_\kappa(r(\mathbf{p},\mathbf{q}))}{\partial r}\frac{\partial^2 r(\mathbf{p},\mathbf{q})}{\partial n_p \partial n_q} + \frac{\partial^2 G_\kappa(r(\mathbf{p},\mathbf{q}))}{\partial r^2}\frac{\partial r(\mathbf{p},\mathbf{q})}{\partial n_p}\frac{\partial r(\mathbf{p},\mathbf{q})}{\partial n_q}$$

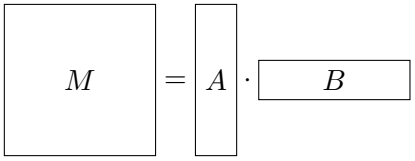
4 $\mathcal{R}(k)$-matrices

So far we have shown the build-up of the BEM system. Just the assembly of the full system already causes quadratic costs in memory and computation time. Therefore we need to find structures to approximate the linear system in a compressed way. We start by introducing low-rank matrix factorizations as a central concept to this goal. In the following we will give a definition of low rank matrices. Then we will look at their arithmetic to motivate how they economise memory and runtime. Finally we will treat the optimal truncation of given matrices to lower ranks.

4.1 $\mathcal{R}(k)$-matrix factorization

Any matrix $M \in \mathbb{C}^{m\times n}$ with $\operatorname{rank}(M) \leq k$ can be factorized:

$$M = AB, \quad \text{with } A \in \mathbb{C}^{m\times k}, B \in \mathbb{C}^{k\times n}$$



(4.1)

The existence and calculation of such a factorization will become clear in a later section. The memory consumption of the factorized representation of M is $k(n+m)$. This is in contrast to the full matrix representation with an mn memory requirement.

An equivalent notion to the factorized form is to see the rank-k matrix M as a sum of k rank-1 matrices:

$$M = \sum_{i=1}^{k} A_{:i} B_{i:} \tag{4.2}$$

We recreate the following definitions in the style of the standard works of the $\mathcal{H}$-matrix literature [BGH03] [Hac15].

Definition 4.1 ($\mathcal{R}(k,m,n)$). *Let m, n, $k \in \mathbb{N}$. We define the set of $m \times n$-matrices over $\mathbb{C}$, where there is a factorization in the format of (4.1) explicitly given as $\mathcal{R}(k,m,n)$.*

$$\mathcal{R}(k,m,n) \coloneqq \left\{ M \in \mathbb{C}^{m\times n} \,\middle|\, \begin{array}{r} M \text{ is explicitly given in the format } (A,B), \\ \text{with } M = AB,\ A \in \mathbb{C}^{m\times k},\ B \in \mathbb{C}^{k\times n} \end{array} \right\} \tag{4.3}$$

Further we write

$$M \in \mathcal{R}(k) :\Leftrightarrow \exists m,n \in \mathbb{N}, \text{ so that } M \in \mathcal{R}(k,m,n) \tag{4.4}$$

and

$$M \in \mathcal{R} :\Leftrightarrow \exists k \in \mathbb{N}, \textit{ so that } M \in \mathcal{R}(k) \tag{4.5}$$

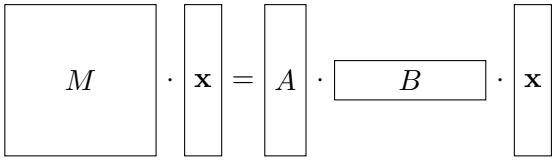
$$M \in \mathcal{R}(m,n) :\Leftrightarrow \exists k \in \mathbb{N}, \textit{ so that } M \in \mathcal{R}(k,m,n) \tag{4.6}$$

An $\mathcal{R}(k,m,n)$-matrix can be considered to be of low rank, if $k \leq \min\{m,n\}$. We will see later, that we will want at least $k^2 \lessapprox \min\{m,n\}$.

In the following sections we will give insight into the basic operations of $\mathcal{R}(k)$-matrix arithmetic.

4.2 $\mathcal{R}(k)$-matrix-vector product

We use the symbol $\coloneqq$ to signify that the left-hand side of the symbol is assigned a computational result. Let $M = AB \in \mathcal{R}(k,m,n)$ be a given low-rank matrix. The matrix-vector product with M is then efficiently evaluated as $\mathbf{z} \coloneqq A(B\mathbf{x})$.



operations count:

- $\mathbf{y} \coloneqq B\mathbf{x}$ contains kn multiplications and $k(n-1)$ additions
- $\mathbf{z} \coloneqq Ay$ contains mk multiplications and $m(k-1)$ additions

costs in the real case:

- $2k(m+n) - m - k$ flops

costs in the complex case:

- a complex addition costs 2 (real) flops
- a complex multiplication costs 6 flops
- overall costs: $8k(m+n) - 2(m+k)$ flops

The $\mathcal{R}(k)$-matrix-vector product is linear in $(m+n)$. This is again in contrast to the

matrix-vector product in full matrix representation with a complexity of $O(mn)$.

4.3 $\mathcal{R}(k)$-matrix multiplication

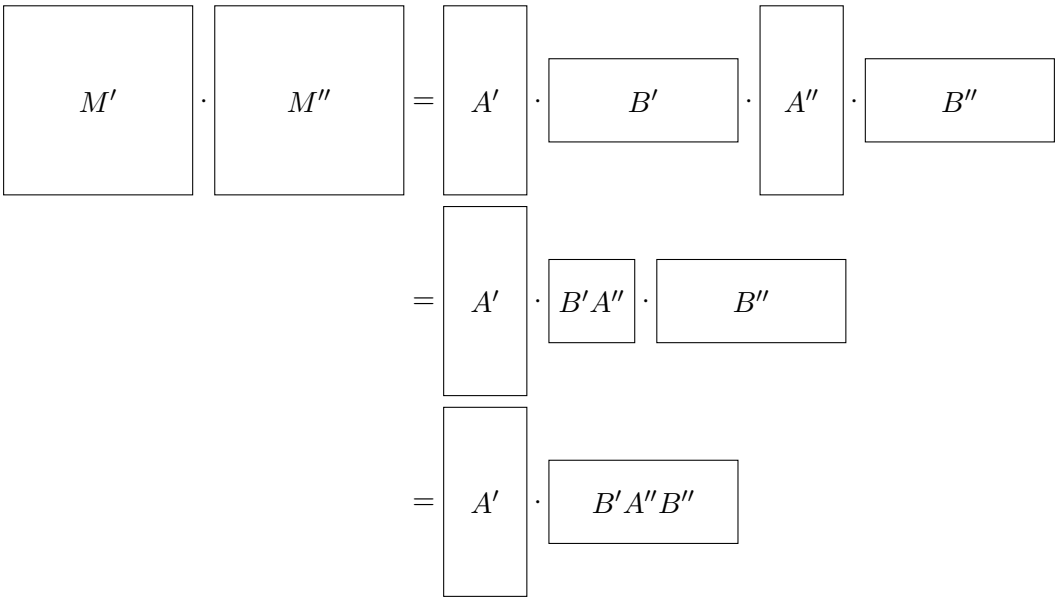
Now we look at the $\mathcal{R}(k)$-matrix multiplication. Let $M' \in \mathcal{R}(k', m, l)$ and $M'' \in \mathcal{R}(k'', l, n)$ be two low rank matrices. Their respective factorizations are $M' = A'B'$ and $M'' = A''B''$. The factorized Product $M'M''$ can be efficiently obtained via

$$M'M'' \coloneqq A'((B' \cdot A'') \cdot B'') \in \mathcal{R}(k', m, n) \tag{4.7}$$

or

$$M'M'' \coloneqq (A' \cdot (B' \cdot A''))B'' \in \mathcal{R}(k'', m, n). \tag{4.8}$$

We give a visualization of version (4.7):



operations count:

- $B' \cdot A''$ contains $k'k''l$ multiplications and $k'k''(l-1)$ additions
- version (4.7): $(B'A'') \cdot B''$ contains $k'k''n$ multiplications and $k'n(k''-1)$ additions
- version (4.8): $A' \cdot (B'A'')$ contains $k'k''m$ multiplications and $k''m(k'-1)$ additions

costs in the real case:

- version (4.7): $2k'k''(l+n) - k'(k''+n)$ flops
- version (4.8): $2k'k''(l+m) - k''(k'+m)$ flops

costs in the complex case:

- a complex addition costs 2 (real) flops
- a complex multiplication costs 6 flops
- overall costs version (4.7): $8k'k''(l+n) - 2k'(k''+n)$ flops
- overall costs version (4.8): $8k'k''(l+m) - 2k''(k'+m)$ flops

Just like the matrix-vector product, the low-rank matrix-matrix product has a linear operations count with regard to $m+l$ and $n+l$; whereas the multiplication in full representation has costs in the order of lmn.

4.4 $\mathcal{R}(k)$-matrix addition

Now we treat the $\mathcal{R}(k)$-matrix addition as the next fundamental arithmetical operation. Let us look at the structure of the sum first. Let $M' \in \mathcal{R}(k', m, n)$ and $M'' \in \mathcal{R}(k'', m, n)$. $M' = A'B'$, $M'' = A''B''$ are their respective factorizations. It holds $\operatorname{rank}(M' + M'') \leq \operatorname{rank}(M') + \operatorname{rank}(M'')$. Therefore $M' + M'' \in \mathcal{R}(k' + k'', m, n)$. We can sum the two matrices by writing them 'next' to each other:

$$M' + M'' = \begin{pmatrix} A' & A'' \end{pmatrix} \begin{pmatrix} B' \\ B'' \end{pmatrix}$$

If one has a data structure where one can link the matrices together, the addition of A', A'' and B', B'' can be implemented practically without computational cost. The increase in rank will increase the cost of all other further operations on the sum though.

4.5 Best $\mathcal{R}(k)$-matrix approximation

Now that we have dealt with the fundamentals of $\mathcal{R}(k)$-arithmetic, we will apply ourselves to the question, how to approximate a general matrix with a low-rank matrix. More specifically we will answer, how to find the best rank-k approximation to a rank-l matrix with $k < l$.

4.5.1 The singular value decomposition

For any matrix $A \in \mathbb{C}^{m \times n}$ there exists a factorization into two unitary matrices $U \in \mathbb{C}^{m \times m}$, $V \in \mathbb{C}^{n \times n}$ and a diagonal matrix $\Sigma = \operatorname{diag}(\sigma_1, \ldots, \sigma_{min(m,n)}) \in \mathbb{R}^{m \times n}$ such that

$$A = U \Sigma V^*, \tag{4.9}$$

where $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_{min(m,n)} \geq 0$. This factorization is called the singular value decomposition (SVD). For more detailed information on the SVD we reference the standard work on matrix computations [GvL13, page 80]. A good numerical implementation of the SVD is given in the same book [GvL13, page 492]. Just in case the readers intend to implement the SVD, they are advised to make the bidiagonalization step of [GvL13, page 492, algorithm 8.6.2] real-bidiagonalizing, to adapt the algorithm to the complex domain.

4.5.2 The cost of the SVD

Calculating the SVD is a rather expensive operation. Let $A \in \mathbb{C}^{m\times n}$ and without loss of generality $m \geq n$. The computational cost of a full-rank SVD implementation is $O(m^2 n)$. The specific cost of the real Golub-Reinsch SVD is $4m^2n + 8mn^2 + 9n^3$ flops [GvL13, page 493]. Multiplying this term by 4 gives a good approximation for the complex case [GvL13, page 256]. This is due to the fact, that the algorithm has about the same amount of complex additions and multiplications. The complex addition costs 2 (real) flops, whereas the complex multiplication costs 6 flops.

4.5.3 The Eckhart-Young theorem

The SVD is the key to finding the best rank-k approximation to any matrix $A \in \mathbb{C}^{m\times n}$.

Let $A \in \mathbb{C}^{m\times n}$ and $A = U\Sigma V^*$ is its SVD, where $\Sigma = \operatorname{diag}(\sigma_1, \ldots, \sigma_{\min(m,n)})$. For $k < \operatorname{rank}(A)$ let $\tilde{A}^k$ be the rank-k matrix, obtained through the truncated SVD of A:

$$\tilde{A}^k = U \cdot \operatorname{diag}(\sigma_1, \ldots, \sigma_k, 0_{k+1}, \ldots, 0_{\min(m,n)}) \cdot V^*. \tag{4.10}$$

Then

$$\left\| A - \tilde{A}^k \right\|_2 = \min_{\substack{B\in\mathbb{C}^{m\times n} \\ \operatorname{rank}(B)\leq k}} \|A - B\|_2 = \sigma_{k+1}. \tag{4.11}$$

The truncated SVD gives us the rank-k best approximation and also directly the approximation error in the spectral norm. We repeat statement (4.11) for the Frobenius norm:

$$\left\| A - \tilde{A}^k \right\|_F = \min_{\substack{B\in\mathbb{C}^{m\times n} \\ \operatorname{rank}(B)\leq k}} \|A - B\|_F = \sqrt{\sigma_{k+1}^2 + \cdots + \sigma_{\min(m,n)}^2}. \tag{4.12}$$

The approximation error in the Frobenius norm is the Euclidean norm of the truncated singular values. One can find a proof for above theorem in [GvL13, page 79].

Equation (4.10) gives us directly compressed factorized representations of $\tilde{A}^k$:

$$\begin{aligned} \tilde{A}^k &= \begin{pmatrix} U_{:1} & \cdots & U_{:k} \end{pmatrix} \cdot \operatorname{diag}(\sigma_1, \ldots, \sigma_k) \cdot \begin{pmatrix} V_{1:}^* \\ \vdots \\ V_{k:}^* \end{pmatrix} \\ &= \begin{pmatrix} U_{:1} & \cdots & U_{:k} \end{pmatrix} \cdot \left(\operatorname{diag}(\sigma_1, \ldots, \sigma_k) \cdot \begin{pmatrix} V_{1:}^* \\ \vdots \\ V_{k:}^* \end{pmatrix} \right) \end{aligned} \tag{4.13}$$

To avoid ambiguity; $V_{k:}^*$ signifies the k-th row of V^*.

It can be beneficial to store low rank matrices in the SVD format, at the cost of saving only one more vector. The Frobenius norm is easily available then. To keep the notation brief, we keep the low rank factorizations throughout this text mostly in the form AB.

4.5.4 The compressed SVD

We have described so far how to use the SVD to truncate a full matrix. Now we extend the SVD to $\mathcal{R}(k)$-matrices. Let $M = AB \in \mathcal{R}(k, m, n)$. We show an efficient SVD procedure in algorithmic form [BGH03, Implementation 5.6]:

Algorithm 4.1: compressedSVD(M)

input: $M = AB \in \mathcal{R}(k, m, n)$
output: $U \in \mathbb{C}^{m\times k}$, $\Sigma \in \mathbb{C}^{k\times k}$, $V^* \in \mathbb{C}^{k\times n}$ // U, V^* unitary; Σ diagonal
begin
$(Q_A, R_A) \coloneqq \text{thinQR}(A)$ // calculate the thin QR decomposition of A
$(Q_{B^\intercal}, R_{B^\intercal}) \coloneqq \text{thinQR}(B^\intercal)$ // calculate the thin QR decomposition of $B^\intercal$

// $R_A, R_{B^\intercal} \in \mathbb{C}^{k\times k}$
$(\tilde{U}, \Sigma, \tilde{V}^*) \coloneqq \text{SVD}(R_A R_{B^\intercal}^\intercal)$ // calculate the SVD of $R_A R_{B^\intercal}^\intercal \in \mathbb{C}^{k\times k}$
// now we could truncate
$U \coloneqq Q_A \tilde{U}$ // Q_A and $\tilde{U}$ are unitary $\implies$ U is unitary too
$V^* \coloneqq \tilde{V}^* Q_{B^\intercal}^\intercal$

return U, Σ, V^* // $U\Sigma V^* = M$
end

Information on the QR decomposition can also be found in [GvL13, page 248]. The authors give the cost of the real thinQR as $2k^2(m-k/3)$ for determining R and $2mk^2-k^2$ for evaluating the product of a factorized thin $(m \times k)$ Q and an $\mathbb{R}^{k\times k}$ matrix [GvL13, page 238] (Evaluating $U \coloneqq Q_A\tilde{U}$ for example). Multiplying those values by 4 gives a good approximation for the complex case [GvL13, page 256].

We assume these only approximate requirements for our further runtime estimates. The cost of the product of the triangular matrices $R_A R_{B^\intercal}^\intercal$ is

$$\begin{aligned} C_{R_A \cdot R_{B^\intercal}^\intercal} &= \sum_{i=1}^{k}\sum_{j=1}^{k} \underbrace{6}_{multiplication} \min\{i,j\} + \underbrace{2}_{addition} (\min\{i,j\} - 1) \\ &= \sum_{i=1}^{k} (2(k-(i-1))-1)(8i-2) \\ &= \sum_{i=1}^{k} (2k-2i+1)(8i-2) \\ &= \frac{4k}{3} + 2k^2 + \frac{8k^3}{3} \end{aligned}$$

We use an upper bound $C_{R_A \cdot R_{B^\intercal}^\intercal} = 6k^3$ for further estimates. The cost of the SVD of $R_A R_{B^\intercal}^\intercal$ is

$$C_{SVD(R_A R_{B^\intercal}^\intercal)} = 84k^3. \tag{4.14}$$

Again we use upper bounds to express the cost of the thin QR-factorizations and the products with Q:

$$C_{\text{thinQR}(A)} = 8k^2m$$

$$C_{\text{thinQR}(B^\intercal)} = 8k^2n$$

$$C_{Q_A\tilde{U}} = 8mk^2$$

$$C_{\tilde{V}^*Q_{B^\intercal}^\intercal} = 8nk^2$$

We then give an approximation/upper bound for the combined cost of the compressed SVD:

$$\begin{aligned} C_{CSVD} &= C_{\text{thinQR}(A)} + C_{\text{thinQR}(B^\intercal)} + C_{R_A \cdot R_{B^\intercal}^\intercal} \\ &\quad + C_{SVD(R_A R_{B^\intercal}^\intercal)} + C_{Q_A\tilde{U}} + C_{\tilde{V}^*Q_{B^\intercal}^\intercal} \\ &= 16k^2(n+m) + 90k^3 \end{aligned} \tag{4.15}$$

As with earlier operations, we can truncate low rank matrices in $O(n+m)$ runtime. In particular we can perform the truncated addition of $M' \in \mathcal{R}(k', m, n)$ and $M'' \in \mathcal{R}(k'', m, n)$ with costs of $16(k'+k'')^2(n+m) + 90(k'+k'')^3$.

5 A model $\mathcal{H}$-matrix

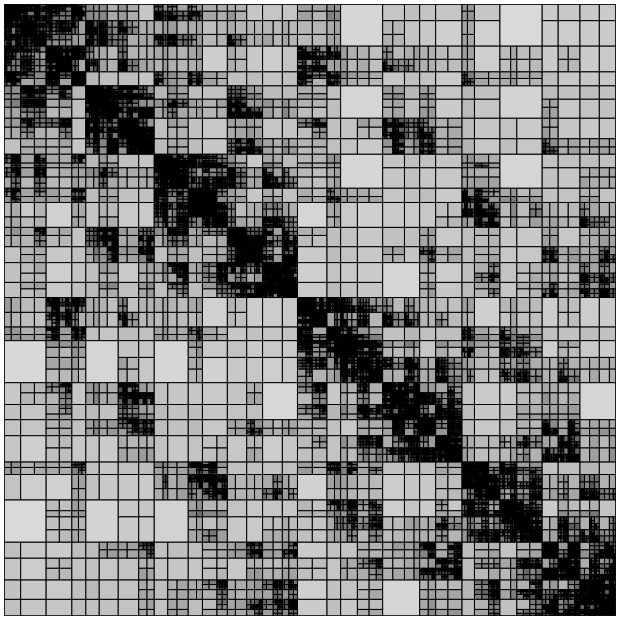


Figure 5.1: A 5468×5468 $\mathcal{H}$-Matrix

The BEM matrices are generally of full rank. We can not directly approximate them with a low rank matrix. In particular the kernel singularities lie on the diagonals of the BEM matrices. This diagonal structure cannot be efficiently approximated with a simple low-rank approach. We therefore have to refine our ansatz. Instead of approximating the entire system matrix with one low-rank matrix, we approximate submatrices (subblocks). We will first motivate a block based low rank scheme with an introductory example.

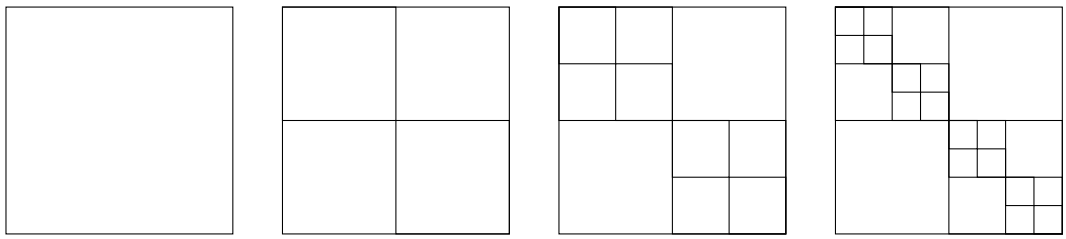
The following example is heavily inspired by [Hac09, Section 3]. For simplicity let $n \in \mathbb{N}$ be some power of two ($n = 2^p$). Also $k \in \mathbb{N}$. We can represent a matrix $M \in \mathbb{C}^{2^p \times 2^p}$ in block form:

$$M = \begin{pmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{pmatrix} \quad \text{with } M_{ij} \in \mathbb{C}^{2^{p-1} \times 2^{p-1}}$$

Now we define the recursive structure $\mathcal{H}_p$:

$$\mathcal{H}_p = \begin{pmatrix} \mathcal{H}_{p-1} & \mathcal{R}_{p-1} \\ \mathcal{R}_{p-1} & \mathcal{H}_{p-1} \end{pmatrix} \quad \text{with } \mathcal{R}_{p-1} \coloneqq \mathcal{R}(k, 2^{p-1}, 2^{p-1}). \tag{5.1}$$

The recursion goes down to $\mathcal{H}_0 = \mathbb{C}^{1\times 1}$. The off-diagonal blocks are compressed low-rank matrices.



$\mathcal{H}_0 = \mathbb{C}^{1\times 1}$ $\mathcal{H}_1 = \mathbb{C}^{2\times 2}$ $\mathcal{H}_2 \subset \mathbb{C}^{4\times 4}$ $\mathcal{H}_3 \subset \mathbb{C}^{8\times 8}$

We have for the number of blocks per recursion level $N_{blocks}(p) = 2N_{blocks}(p-1) + 2$. Therefore

$$\begin{aligned} N_{blocks}(p) &= 2N_{blocks}(p-1) + 2 \\ &= 2(2N_{blocks}(p-2) + 2) + 2 \\ &= 2(2(2N_{blocks}(p-3) + 2) + 2) + 2 \\ &= \cdots \\ &= 2 + 2^2 + \cdots + 2^i + 2^i N_{blocks}(p-i) \\ &= 2^p + \sum_{i=1}^{p} 2^i \\ &= 3n - 2 \quad \text{, with} \quad n = 2^p \end{aligned}$$

So the number of blocks is linearly dependent on n.

The memory recursion is $M_{\mathcal{H}}(p) = 2M_{\mathcal{H}}(p-1) + 2M_{\mathcal{R}}(p-1) = 2M_{\mathcal{H}}(p-1) + 2k(2^{p-1} + 2^{p-1})$. This can be resolved:

$$\begin{aligned} M_{\mathcal{H}}(p) &= 2M_{\mathcal{H}}(p-1) + 2k(2^{p-1} + 2^{p-1}) \\ &= 2M_{\mathcal{H}}(p-1) + k2^{p+1} \\ &= 2(2M_{\mathcal{H}}(p-2) + k2^p) + k2^{p+1} \\ &= 2(2(2M_{\mathcal{H}}(p-3) + k2^{p-1}) + k2^p) + k2^{p+1} \\ &= \cdots \\ &= 2^i M_{\mathcal{H}}(p-i) + ik2^{p+1} \\ &= 2^p \underbrace{M_{\mathcal{H}}(0)}_{\in \mathbb{C}^{1\times 1}} + pk2^{p+1} \\ &= n + 2kn\log_2(n) \end{aligned}$$

Thus the memory requirements of $\mathcal{H}_p \subset \mathbb{C}^{n\times n}$ are quasi-linear.

5.1 $\mathcal{H}_p$-matrix-vector product

Now we will examine the $\mathcal{H}_p$-matrix-vector product. Let again $n = 2^p$, $M \in \mathcal{H}_p$ and $\mathbf{x} \in \mathbb{C}^n$. M has a block partition like in (5.1):

$$M = \begin{pmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{pmatrix}$$

The product is $\mathbf{y} = M\mathbf{x}$, whereby $\mathbf{x}$ and $\mathbf{y}$ can also be suitably partitioned:

$$\mathbf{x} = \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{pmatrix} \text{ and } \mathbf{y} = \begin{pmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \end{pmatrix} \quad \text{with } \mathbf{x}_1, \mathbf{x}_2, \mathbf{y}_1, \mathbf{y}_2 \in \mathbb{C}^{2^{p-1}}.$$

The product in block form:

$$\boxed{\begin{matrix} \mathbf{y}_1 \\ \mathbf{y}_2 \end{matrix}} = \boxed{\begin{matrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{matrix}} \cdot \boxed{\begin{matrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{matrix}}$$

The product is composed of four sub-products and two additions. $\mathbf{y}_1 = M_{11}\mathbf{x}_1 + M_{12}\mathbf{x}_2$ and $\mathbf{y}_2 = M_{21}\mathbf{x}_1 + M_{22}\mathbf{x}_2$. As M_{11}, $M_{22} \in \mathcal{H}_{p-1}$, the $\mathcal{H}$-matrix-vector product is recursive. This leads to the following computational cost recursion:

$$\begin{aligned}
C_{MV_{\mathcal{H}}}(p) &= 2C_{MV_{\mathcal{H}}}(p-1) + 2C_{MV_{\mathcal{R}}}(p-1) + \underbrace{2^{p+1}}_{n \text{ additions}} \\
&= 2C_{MV_{\mathcal{H}}}(p-1) + 2(8k(2^{p-1} + 2^{p-1}) - 2(2^{p-1} + k)) + 2^{p+1} \\
&= 2C_{MV_{\mathcal{H}}}(p-1) + k2^{p+4} - 2^{p+1} - 4k + 2^{p+1} \\
&= 2C_{MV_{\mathcal{H}}}(p-1) + k(2^{p+4} - 4) \\
&= 2(2C_{MV_{\mathcal{H}}}(p-2) + k(2^{p+3} - 4)) + k(2^{p+4} - 4) \\
&= \cdots \\
&= 2^i C_{MV_{\mathcal{H}}}(p-i) + i2^{p+4}k - \sum_{j=1}^{i} 2^{j+1}k \\
&= 2^p \underbrace{C_{MV_{\mathcal{H}}}(0)}_{\overset{*}{=}6} + p2^{p+4}k - 4(n-1)k \\
&= n\log_2(n)16k + 6n - 4k(n-1)
\end{aligned}$$

The applicable unit is real flops. The computational costs of the $\mathcal{H}_p$-matrix-vector product are consequently quasi-linear.

*) We have set $C_{MV_{\mathcal{H}}}(0) = 6$, as the full (scalar) representation of $\mathcal{H}_0$ is more efficient than a redundant rank-k factorization. This is also true for any block, where k is of the same magnitude as its rank. For representational simplicity we have accounted for this only in the case of $\mathcal{H}_0$ in the recursion formulas. For further estimates, we simplify the cost of the $\mathcal{H}_p$-matrix-vector product to an upper bound of $C_{MV_{\mathcal{H}}}(p) = n\log_2(n)16k + 6n$.

5.2 $\mathcal{H}_p$-matrix addition

Let $n = 2^p$ again. We will now look into the $\mathcal{H}_p$-matrix addition. We distinguish three cases.

1. $M' \oplus_k M'' \in \mathcal{R}_p(k)$ with M', $M'' \in \mathcal{R}_p(k)$ and cost $C_{\mathcal{R}+\mathcal{R}}(p)$
2. $M' \oplus_k M'' \in \mathcal{H}_p(k)$ with M', $M'' \in \mathcal{H}_p(k)$ and cost $C_{\mathcal{H}+\mathcal{H}}(p)$
3. $M' \oplus_k M'' \in \mathcal{H}_p(k)$ with $M' \in \mathcal{H}_p(k)$, $M'' \in \mathcal{R}_p(k)$ and cost $C_{\mathcal{H}+\mathcal{R}}(p)$

The $\oplus_k$ symbol signifies, that the addition is not exact, but truncated block-wise to rank k.

Case 1.

The exact addition could be implemented practically without computational cost, though with an increased rank of the sum. The benefit of truncating the sum is to reduce the costs of further operations on the sum. The truncation can be implemented by calculating and truncating the compressed SVD (Algorithm 4.1) of $M' + M''$. The cost of $M' \oplus_k M''$ is therefore $C_{\mathcal{R}\oplus\mathcal{R}}(p) = 16(k+k)^2 2^{p+1} + 87(k+k)^3 = 128nk^2 + 720k^3$.

Case 2.

If both summands have the $\mathcal{H}_p$-structure, the sum will too. So for $\mathcal{H}_p = \mathcal{H}_p + \mathcal{H}_p$ we have:

$$\begin{pmatrix} \mathcal{H}_{p-1} & \mathcal{R}_{p-1} \\ \mathcal{R}_{p-1} & \mathcal{H}_{p-1} \end{pmatrix} + \begin{pmatrix} \mathcal{H}_{p-1} & \mathcal{R}_{p-1} \\ \mathcal{R}_{p-1} & \mathcal{H}_{p-1} \end{pmatrix} = \begin{pmatrix} \mathcal{H}_{p-1} + \mathcal{H}_{p-1} & \mathcal{R}_{p-1} + \mathcal{R}_{p-1} \\ \mathcal{R}_{p-1} + \mathcal{R}_{p-1} & \mathcal{H}_{p-1} + \mathcal{H}_{p-1} \end{pmatrix}.$$

Again we can derive a recursion formula for the computational cost:

$$\begin{aligned}
C_{\mathcal{H}\oplus\mathcal{H}}(p) &= 2\left(C_{\mathcal{H}\oplus\mathcal{H}}(p-1) + C_{\mathcal{R}\oplus\mathcal{R}}(p-1)\right) \\
&= 2\left(C_{\mathcal{H}\oplus\mathcal{H}}(p-1) + 16(k+k)^2 2^p + 87(k+k)^3\right) \\
&= 2C_{\mathcal{H}\oplus\mathcal{H}}(p-1) + 2^{7+p}k^2 + 2k^3 720 \\
&= 2^2 C_{\mathcal{H}\oplus\mathcal{H}}(p-2) + \sum_{i=1}^{2} 2^{7+p}k^2 + \sum_{i=1}^{2} 2^i 720k^3 \\
&= \cdots \\
&= 2^p C_{\mathcal{H}\oplus\mathcal{H}}(0) + \sum_{i=1}^{p} 2^{7+p}k^2 + \sum_{i=1}^{p} 2^i 720k^3 \\
&= n\underbrace{C_{\mathcal{H}\oplus\mathcal{H}}(0)}_{2} + 128k^2 n\log_2(n) + 1440k^3(n-1) \\
&= 128k^2 n\log_2(n) + 1440k^3(n-1) + 2n
\end{aligned}$$

Like the $\mathcal{H}_p$-matrix vector product, $\oplus_k : \mathcal{H}_p \times \mathcal{H}_p \mapsto \mathcal{H}_p$ is quasi linear in n. We simplify $C_{\mathcal{H}\oplus\mathcal{H}}(p)$ slightly to an upper bound $C_{\mathcal{H}\oplus\mathcal{H}}(p) = 128k^2 n\log_2(n) + 1440k^3 n + 2n$.

Case 3.

Now we examine the case, where one summand M' is in $\mathcal{H}_p(k)$ and the other summand M'' is a low-rank matrix. The low rank matrix $M'' \in \mathcal{R}_p(k) = \mathcal{R}(k, 2^p, 2^p)$ can also be viewed in a recursive block-structured way:

$$M'' = \begin{pmatrix} M''_{11} & M''_{12} \\ M''_{21} & M''_{22} \end{pmatrix} \quad \text{with } M''_{ij} \in \mathcal{R}_{p-1}(k)$$

So for $\mathcal{H}_p = \mathcal{H}_p + \mathcal{R}_p$ we have the recursive structure:

$$\mathcal{H}_p + \mathcal{R}_p = \begin{pmatrix} \mathcal{H}_{p-1} & \mathcal{R}_{p-1} \\ \mathcal{R}_{p-1} & \mathcal{H}_{p-1} \end{pmatrix} + \begin{pmatrix} \mathcal{R}_{p-1} & \mathcal{R}_{p-1} \\ \mathcal{R}_{p-1} & \mathcal{R}_{p-1} \end{pmatrix} = \begin{pmatrix} \mathcal{H}_{p-1} + \mathcal{R}_{p-1} & \mathcal{R}_{p-1} + \mathcal{R}_{p-1} \\ \mathcal{R}_{p-1} + \mathcal{R}_{p-1} & \mathcal{H}_{p-1} + \mathcal{R}_{p-1} \end{pmatrix} \tag{5.2}$$

The subblocks M''_{ij} of the $\mathcal{R}_p(k)$-summand M'' are accessible without computational cost in the following way. Let $A''B''$ be the low-rank factorization of M''. The matrices A'' and B'' can be viewed row-block and column-block-wise respectively:

$$A'' = \begin{pmatrix} A''_{11} \\ A''_{21} \end{pmatrix}, \; B'' = \begin{pmatrix} B''_{11} & B''_{12} \end{pmatrix} \quad \text{, with } A''_{11}, A''_{21} \in \mathbb{C}^{2^{p-1} \times k} \text{ and } B''_{11}, B''_{12} \in \mathbb{C}^{k \times 2^{p-1}}$$

The subblocks of M'' above are suitably factorized by the submatrices of A'' and B'':

$$M''_{11} = A''_{11}B''_{11}, \quad M''_{21} = A''_{21}B''_{11}, \quad M''_{12} = A''_{11}B''_{12}, \quad M''_{22} = A''_{21}B''_{22}$$

The recursive structure (5.2) leads to the following computational cost recursion for $\mathcal{H}_p \oplus_k \mathcal{R}_p$:

$$C_{\mathcal{H}\oplus\mathcal{R}}(p) = 2C_{\mathcal{H}\oplus_k\mathcal{H}}(p-1) + 2C_{\mathcal{R}\oplus\mathcal{R}}(p-1)$$

This is the same recursion as for $\mathcal{H}_p \oplus_k \mathcal{H}_p$ (case 2). Therefore:

$$\begin{aligned} C_{\mathcal{H}\oplus\mathcal{R}}(p) &= C_{\mathcal{H}\oplus\mathcal{H}}(p) \\ &= 128k^2 n \log_2(n) + 1440k^3 n + 2n \end{aligned}$$

The truncated $\mathcal{H}_p\mathcal{H}_p$-addition is hence also quasi-linear.

5.3 $\mathcal{H}_p$-matrix product

Let $n = 2^p$ again. We will now examine the $\mathcal{H}_p$-matrix product We distinguish four cases.

1. $M' \cdot M'' \in \mathcal{R}_p(k)$ with M', $M'' \in \mathcal{R}_p(k)$ and cost $C_{\mathcal{R}\cdot\mathcal{R}}(p)$
2. $M' \cdot M'' \in \mathcal{R}_p(k)$ with $M' \in \mathcal{H}_p(k)$, $M'' \in \mathcal{R}_p(k)$ and cost $C_{\mathcal{H}\cdot\mathcal{R}}(p)$
3. $M' \cdot M'' \in \mathcal{R}_p(k)$ with $M' \in \mathcal{R}_p(k)$, $M'' \in \mathcal{H}_p(k)$ and cost $C_{\mathcal{R}\cdot\mathcal{H}}(p)$
4. $M' \odot_k M'' \in \mathcal{H}_p(k)$ with M', $M'' \in \mathcal{H}_p(k)$ and cost $C_{\mathcal{H}\cdot\mathcal{H}}(p)$

The k subscript in $\odot_k$ signifies, that block-wise additions in the multiplication are truncated to rank k.

Case 1.

Case 1) is an exact operation and a special case of (4.7) or (4.8). Its cost is $C_{\mathcal{R}\cdot\mathcal{R}}(p) = 16nk^2 - 2nk - 2k^2$. We simplify to $C_{\mathcal{R}\cdot\mathcal{R}}(p) = 16nk^2$ for further estimates.

Case 2.

Let $M'' = AB$. The multiplication $M' \cdot M''$ is then realized by multiplying M' with A. This multiplication reduces to k $\mathcal{H}_p$-matrix-vector products. The overall cost of the operation is therefore $C_{\mathcal{H}\cdot\mathcal{R}}(p) = n\log_2(n)16k^2 + 6nk$

Case 3.

Case 3) is analogous to 2).

Case 4.

For case 4) we will illustrate the structure of the product first.

$$\begin{aligned}
\mathcal{H}_p &= \mathcal{H}_p \cdot \mathcal{H}_p \\
\begin{pmatrix} \mathcal{H}_{p-1} & \mathcal{R}_{p-1} \\ \mathcal{R}_{p-1} & \mathcal{H}_{p-1} \end{pmatrix} &= \begin{pmatrix} \mathcal{H}_{p-1} & \mathcal{R}_{p-1} \\ \mathcal{R}_{p-1} & \mathcal{H}_{p-1} \end{pmatrix} \cdot \begin{pmatrix} \mathcal{H}_{p-1} & \mathcal{R}_{p-1} \\ \mathcal{R}_{p-1} & \mathcal{H}_{p-1} \end{pmatrix} \\
&= \begin{pmatrix} \mathcal{H}_{p-1}\cdot\mathcal{H}_{p-1} + \mathcal{R}_{p-1}\cdot\mathcal{R}_{p-1} & \mathcal{H}_{p-1}\cdot\mathcal{R}_{p-1} + \mathcal{R}_{p-1}\cdot\mathcal{H}_{p-1} \\ \mathcal{R}_{p-1}\cdot\mathcal{H}_{p-1} + \mathcal{H}_{p-1}\cdot\mathcal{R}_{p-1} & \mathcal{R}_{p-1}\cdot\mathcal{R}_{p-1} + \mathcal{H}_{p-1}\cdot\mathcal{H}_{p-1} \end{pmatrix} \\
&= \begin{pmatrix} \mathcal{H}_{p-1} + \mathcal{R}_{p-1} & \mathcal{R}_{p-1} + \mathcal{R}_{p-1} \\ \mathcal{R}_{p-1} + \mathcal{R}_{p-1} & \mathcal{R}_{p-1} + \mathcal{H}_{p-1} \end{pmatrix}
\end{aligned} \tag{5.3}$$

Now let M', $M'' \in \mathcal{H}_p$. Their truncated product has the following form:

$$\begin{aligned}
X &= M' \odot_k M'' \\
\begin{pmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{pmatrix} &= \begin{pmatrix} M'_{11} & M'_{12} \\ M'_{21} & M'_{22} \end{pmatrix} \odot_k \begin{pmatrix} M''_{11} & M''_{12} \\ M''_{21} & M''_{22} \end{pmatrix} \\
\begin{pmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{pmatrix} &= \begin{pmatrix} M'_{11} \odot_k M''_{11} \oplus_k M'_{12}\cdot M''_{21} & M'_{11}\cdot M''_{12} \oplus_k M'_{12}\cdot M''_{22} \\ M'_{21}\cdot M''_{11} \oplus_k M'_{22}\cdot M''_{21} & M'_{21}\cdot M''_{12} \oplus_k M'_{22} \odot_k M''_{22} \end{pmatrix}
\end{aligned} \tag{5.4}$$

The cost recursion follows from equation (5.4):

$$
\begin{aligned}
C_{\mathcal{H}\cdot\mathcal{H}}(p) &= 2C_{\mathcal{H}\cdot\mathcal{H}}(p-1) + 2C_{\mathcal{R}\cdot\mathcal{R}}(p-1) + 4C_{\mathcal{H}\cdot\mathcal{R}}(p-1) \\
&\quad + 2C_{\mathcal{H}+\mathcal{R}}(p-1) + 2C_{\mathcal{R}+\mathcal{R}}(p-1) \\
&= 2C_{\mathcal{H}\cdot\mathcal{H}}(p-1) \\
&\quad + 2(16k^2 2^{p-1}) \\
&\quad + 4(2^{p-1}(p-1)16k^2 + 6k2^{p-1}) \\
&\quad + 2(128k^2 2^{p-1}(p-1) + 1440k^3 2^{p-1} + 2^p) \\
&\quad + 2(128k^2 2^{p-1} + 720k^3) \\
&\leq 2C_{\mathcal{H}\cdot\mathcal{H}}(p-1) \\
&\quad + 16k^2 2^p + 2^{p+1}\left(p16k^2 + 6k\right) \\
&\quad + 128k^2 2^p p + 1440k^3 2^p + 2^{p+1} \\
&\quad + 2(720k^3) \\
&= 2^p C_{\mathcal{H}\cdot\mathcal{H}}(0) \\
&\quad + \sum_{i=1}^{p} \Big[16k^2 n + 2n\left((p-i)16k^2 + 6k\right) \\
&\qquad\qquad + 128k^2 n\log_2(n) + 1440k^3 n + 2n + 2^i(720k^3) \Big] \\
&= 6n + 16k^2 n\log_2(n) + 16k^2 n\log_2^2(n) - 16k^2 n\log_2(n) + 12kn\log_2(n) \\
&\quad + 128k^2 n\log_2^2(n) + 1440k^3 n\log_2(n) + 2n\log_2^2(n) + 1440k^3(n-1) \\
&= 6n + 144k^2 n\log_2^2(n) + 12kn\log_2(n) \\
&\quad + 1440k^3 n\log_2(n) + 2n\log_2^2(n) + 1440k^3(n-1)
\end{aligned}
$$

The runtime of the $\mathcal{H}_p\mathcal{H}_p$-matrix product therefore $O(n\log_2^2(n))$ for fixed k.

5.4 $\mathcal{H}_p$-LU factorization

We end the $\mathcal{H}_p$-model chapter by giving the $\mathcal{H}_p$-LU factorization. The $\mathcal{H}_p$-LU factorization is a direct application of the usual recursive block LU factorization to the $\mathcal{H}_p$ structure. More information on the standard algorithm can be found at [GvL13, Algorithm 3.2.3, page 119]. Let $M \in \mathcal{H}_p$. The block LU factorization of M is:

$$
\begin{aligned}
M &= LU \quad \text{with } L, U \in \mathcal{H}_p \\
\begin{pmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{pmatrix} &= \begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{pmatrix} = \begin{pmatrix} L_{11}U_{11} & L_{11}U_{12} \\ L_{21}U_{11} & L_{21}U_{12} + L_{22}L_{22} \end{pmatrix},
\end{aligned} \tag{5.5}
$$

whereby L_{ii}, U_{ii} have the same structure as L and U respectively (an off-diagonal zero-block). The above equation directly prescribes the operations for the factorization algorithm:

1. $M_{11} = L_{11}U_{11} \rightarrow$ Call the factorization algorithm recursively on the M_{11} subblock of M. For $M_{11} \in \mathcal{H}_0$ one could use $L \coloneqq M$ and $U \coloneqq 1$. If we were to stop the recursion earlier, we would use the actual LU factorization of the subblock.

2. $M_{12} = L_{11}U_{12}$ and M_{12}, L_{11} are already determined $\rightarrow$ Solve for U_{12} via matrix-valued forward substitution.

3. $M_{21} = L_{21}U_{11}$ and M_{21}, U_{11} are already determined $\rightarrow$ Solve for L_{21} via transposed matrix-valued forward substitution.

4. $M_{22} - L_{21}U_{12} = L_{22}L_{22} \rightarrow$ Subtract $L_{21}U_{12}$ from M_{22} and call the factorization algorithm on the result.

We describe the procedure in algorithmic form.

Algorithm 5.1: BlockLU(M) // Recursive block LU-factorization

input: $M \in \mathcal{H}_p$
output: $L, U \in \mathcal{H}_p$ // $M = LU$
begin
$(L_{11}, U_{11}) \coloneqq BlockLU(M_{11})$ // recursive LU-factorization of subblock M_{11}
$U_{12} \coloneqq L_{11}^{-1}M_{12}$ // solve $L_{11}U_{12} = M_{12}$ for U_{12}
// via matrix-valued forward substitution
// formalized later as matrixForwardSubstitute(L_{11}, M_{12})
$L_{21} \coloneqq M_{21}U_{11}^{-1}$ // solve $L_{21}U_{11} = M_{21}$ for L_{21}
// via a transposed matrix-valued forward substitution

$L_{12} \coloneqq 0$ // set block to zero-block
$U_{21} \coloneqq 0$ // set block to zero-block

$M_{22} \coloneqq M_{22} \ominus_k L_{21}U_{12}$ // truncated subtraction of $L_{21}U_{12}$ from M_{22}
$(L_{22}, U_{22}) \coloneqq BlockLU(M_{22})$ // recursive LU-factorization of subblock M_{22}
return L, U
end

In the ensuing subsections we demonstrate the subprocedures of the $\mathcal{H}_p$-LU factorization; the matrix-valued forward substitution and the transposed matrix-valued forward substitution.

$\mathcal{H}_p$-forward substitution

Let $L \in \mathcal{H}_p$ be a lower triangular block matrix like in the above section. Let $\mathbf{y} \in \mathbb{C}^{2^p}$ be a given vector. Now we will investigate how to solve the system $L\mathbf{x} = \mathbf{y}$ for the unknown vector $\mathbf{x}$. Background information for the forward substitution on general block matrices can be found at [GvL13, section 3.1.4, page 108]. The system in block form:

$$\begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{pmatrix} = \begin{pmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \end{pmatrix}$$

This implies the recursion:

1. Solve $L_{11}\mathbf{x}_1 = \mathbf{y}_1$ for $\mathbf{x}_1$.
2. Calculate the $\mathcal{R}_p$-matrix-vector product and subtraction $\mathbf{z} \coloneqq \mathbf{y}_2 - L_{21}\mathbf{x}_1$.
3. Solve $L_{22}\mathbf{x}_2 = \mathbf{z}$ for $\mathbf{x}_2$.

We arrive at the following cost recursion:

$$\begin{aligned} C_{\mathrm{FSub}_{\mathcal{H}}}(p) &= 2C_{\mathrm{FSub}_{\mathcal{H}}}(p-1) + C_{MV_{\mathcal{R}}}(p-1) + \underbrace{2^p}_{2^{p-1} \text{ subtractions}} \\ &= 2C_{\mathrm{FSub}_{\mathcal{H}}}(p-1) + 8k2^p - 2^p + 2k + 2^p \\ &= 2C_{\mathrm{FSub}_{\mathcal{H}}}(p-1) + 8k2^p + 2k \\ &= 2^p \underbrace{C_{\mathrm{FSub}_{\mathcal{H}}}(0)}_{=6 \text{ (}\mathbb{C}\text{-division)}} + \sum_{i=1}^{p} 8k2^p + 2^i k \\ &= 8kn\log_2(n) + 2kn + 6n - 2k \end{aligned}$$

We simplify $C_{\mathrm{FSub}_{\mathcal{H}}}(p)$ for the use in further estimates $C_{\mathrm{FSub}_{\mathcal{H}}}(p) = 8k(n\log_2(n) + kn)$. We give the procedure as an algorithm.

Algorithm 5.2: forwardSubstitute(L, y)

input: $L \in \mathcal{H}_p$ and lower triangular, $\mathbf{y} \in \mathbb{C}^{2^p}$
output: $\mathbf{x} \in \mathbb{C}^{2^p}$ // $\mathbf{x}$ solves $L\mathbf{x} = \mathbf{y}$
begin
    $\mathbf{x}_1 \coloneqq$ forwardSubstitute$(L, \mathbf{y}_1)$ // call forwardSubstitute recursively
                                                // on the upper half of $\mathbf{y}$
    $\mathbf{y}_2 \coloneqq \mathbf{y}_2 - L_{21}\mathbf{x}_1$.
    $\mathbf{x}_2 \coloneqq$ forwardSubstitute$(L, \mathbf{y}_2)$ // call forwardSubstitute recursively

    return $\mathbf{x}$
end

Matrix-valued $\mathcal{H}_p$-forward substitution

In the $\mathcal{H}_p$-LU factorization algorithm (5.1), we have to solve the matrix system $LX = M$ for the matrix X with X, $M \in \mathcal{R}_p(k)$ and $L \in \mathcal{H}_p$ and triangular. Let $M = A^M B^M$ be the low-rank factorization of M. We then solve the system $LA^X = A^M$ column-wise for the factor matrix A^X. We then set the other factor of matrix X to $B^X \coloneqq B^M$.

Algorithm 5.3: matrixForwardSubstitute(L, M)

input: $L \in \mathcal{H}_p$ and lower triangular, $M \in \mathcal{R}_p(k)$ // $M = A^M B^M$
output: $X = A^X B^X \in \mathcal{R}_p(k)$ // X solves $LX = M$
begin
    for $l \coloneqq 1$ **to** k **do**
        $A^X_{:l} \coloneqq$ forwardSubstitute$(L, A^M_{:l})$ // compute all columns of A^X
                                                // via forward substitution
    endfor
    $B^X \coloneqq B^M$
    return A^X, B^X // $LA^X B^X = A^M B^M$
end

The cost of the matrix-valued forward substitution is k times the cost of the vector-valued forward substitution.

$$\begin{aligned} C_{\mathrm{MatFSub}_{\mathcal{H}}}(p) &= k\ C_{\mathrm{FSub}_{\mathcal{H}}}(p) \\ &= 8k^2(n\log_2(n) + kn) \end{aligned}$$

Transposed $\mathcal{H}_p$-forward substitution

Let $U \in \mathcal{H}_p$ be an upper triangular block matrix like in the above sections. Let $\mathbf{y}^\intercal \in \mathbb{C}^{2^p}$ be a given row vector. Now we will investigate how to solve the system $\mathbf{x}^\intercal U = \mathbf{y}^\intercal$ for the unknown vector $\mathbf{x}^\intercal$. The system in block form:

$$\begin{pmatrix} \mathbf{x}_1^\intercal & \mathbf{x}_2^\intercal \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{pmatrix} = \begin{pmatrix} \mathbf{y}_1^\intercal & \mathbf{y}_2^\intercal \end{pmatrix}$$

The system implies the recursion:

1. Solve $\mathbf{x}_1^\intercal U_{11} = \mathbf{y}_1^\intercal$ for $\mathbf{x}_1^\intercal$.
2. Calculate the $\mathcal{R}_p$-matrix-vector product and subtraction $\mathbf{z}^\intercal \coloneqq \mathbf{y}_2^\intercal - \mathbf{x}_1^\intercal U_{12}$.
3. Solve $\mathbf{x}_2^\intercal U_{22} = \mathbf{z}^\intercal$ for $\mathbf{x}_2^\intercal$.

The cost of the transposed forward substitution is the same as for the forward substitution.

$$\begin{aligned} C_{\mathrm{BSub}_{\mathcal{H}}}(p) &= C_{\mathrm{FSub}_{\mathcal{H}}}(p) \\ &= 8k(n\log_2(n) + kn) \end{aligned}$$

The matrix-valued transposed forward substitution is defined analogously to the matrix-valued forward substitution.

The cost of the $\mathcal{H}_p$-LU factorization

The $\mathcal{H}_p$-LU factorization has the following cost recursion:

$$\begin{aligned}
C_{\mathcal{H}_pLU}(p) &= 2C_{\mathcal{H}_pLU}(p-1) + C_{\mathrm{MatFSub}_{\mathcal{H}}}(p-1) + C_{\mathrm{MatTFSub}_{\mathcal{H}}}(p) + C_{\mathcal{R}\cdot\mathcal{R}}(p-1) \\
&\quad + C_{\mathcal{H}+\mathcal{R}}(p-1) \\
&= 2C_{\mathcal{H}_pLU}(p-1) + 2C_{\mathrm{MatFSub}_{\mathcal{H}}}(p-1) + C_{\mathcal{R}\cdot\mathcal{R}}(p-1) + C_{\mathcal{H}+\mathcal{R}}(p-1) \\
&= 2C_{\mathcal{H}_pLU}(p-1) + 2(8k^2 2^{p-1}((p-1)+k)) + 16(2^{p-1})k^2 \\
&\quad + 128k^2 2^{p-1}(p-1) + 1440k^3 2^{p-1} + 2^p \\
&= 2C_{\mathcal{H}_pLU}(p-1) + 8k^2 2^p(p+k) \\
&\quad + 64k^2 2^p(p-1) + 720k^3 2^p + 2^p \\
&= 2C_{\mathcal{H}_pLU}(p-1) + 72k^2 2^p p + 8k^3 2^p \\
&\quad - 64k^2 2^p + 720k^3 2^p + 2^p \\
&= 2^p \underbrace{C_{\mathcal{H}_pLU}(0)}_{=4} + \sum_{i=1}^{p} \left[72k^2 2^p(p-i+1) - 64k^2 2^p + 728k^3 2^p + 2^p \right] \\
&= 4n + \sum_{i=1}^{p} \left[72k^2 n(p-i+1) - 64k^2 n + 728k^3 n + n \right] \\
&= 4n + 36k^2 n(\log_2^2(n) + \log_2(n)) - 64k^2 n\log_2(n) + 728k^3 n\log_2(n) + n\log_2(n) \\
&= 36k^2 n\log_2^2(n) + 728k^3 n\log_2(n) - 28k^2 n\log_2(n) + n\log_2(n) + 4n
\end{aligned}$$

The $\mathcal{H}_p$-LU factorization is quasi linear in n. We can therefore solve $\mathcal{H}_p$ linear systems in quasi linear time.

6 Clustering

At this point we have constructed a matrix structure, with which we can do fast arithmetic. The question now is, how to make this applicable to our specific situation with Helmholtz BEM matrices? We'll now have another look at those matrices. Lets remember matrix A from (3.11):

$$A := \begin{pmatrix} \frac{i}{2\kappa} - \frac{i}{4\pi\kappa}\int_0^{2\pi} e^{ikR(\theta)}\,\mathrm{d}\theta_1 & \int_{\triangle_2} G(\mathbf{p}_1,\mathbf{q})\,\mathrm{d}q & \dots & \int_{\triangle_{n-1}} G(\mathbf{p}_1,\mathbf{q})\,\mathrm{d}q & \int_{\triangle_n} G(\mathbf{p}_1,\mathbf{q})\,\mathrm{d}q \\ \int_{\triangle_1} G(\mathbf{p}_2,\mathbf{q})\,\mathrm{d}q & \frac{i}{2k} - \frac{i}{4\pi\kappa}\int_0^{2\pi} e^{i\kappa R(\theta)}\,\mathrm{d}\theta_2 & \ddots & \int_{\triangle_{n-1}} G(\mathbf{p}_2,\mathbf{q})\,\mathrm{d}q & \int_{\triangle_n} G(\mathbf{p}_2,\mathbf{q})\,\mathrm{d}q \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \int_{\triangle_1} G(\mathbf{p}_{n-1},\mathbf{q})\,\mathrm{d}q & \int_{\triangle_2} G(\mathbf{p}_{n-1},\mathbf{q})\,\mathrm{d}q & \ddots & \frac{i}{2\kappa} - \frac{i}{4\pi\kappa}\int_0^{2\pi} e^{ikR(\theta)}\,\mathrm{d}\theta_{n-1} & \int_{\triangle_n} G(\mathbf{p}_{n-1},\mathbf{q})\,\mathrm{d}q \\ \int_{\triangle_1} G(\mathbf{p}_n,\mathbf{q})\,\mathrm{d}q & \int_{\triangle_2} G(\mathbf{p}_n,\mathbf{q})\,\mathrm{d}q & \dots & \int_{\triangle_{n-1}} G(\mathbf{p}_n,\mathbf{q})\,\mathrm{d}q & \frac{i}{2\kappa} - \frac{i}{4\pi\kappa}\int_0^{2\pi} e^{ikR(\theta)}\,\mathrm{d}\theta_n \end{pmatrix}$$

We create an informal example to illustrate an opportunity to save computational expenses. The image below shows the BEM domain $\triangle U$ consisting of six triangles.

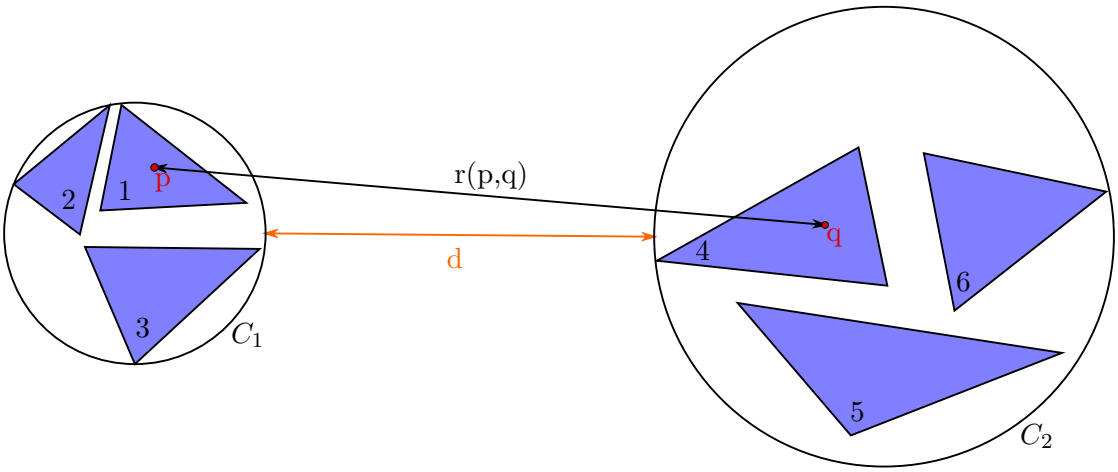


Figure 6.1: Clustering

The associated matrix A can be block partitioned in accordance with the bipartition of the triangle set:

$$A = \begin{matrix} \\ |C_1| \\ |C_2| \end{matrix} \begin{matrix} \begin{matrix} |C_1| & |C_2| \end{matrix} \\ \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \end{matrix}$$

The two circles partition the domain into two clusters based on geometric proximity. We informally assume the distance d of the clusters to be large in contrast to the diameters of the circles and let the circles also be small with regard to the wavelength (small κ in the Helmholtz equation). The off-diagonal blocks describe the interaction between the two different clusters C_1 and C_2. Let us look at block A_{12}. In this case the collocation points $\mathbf{p}_i$ are restricted to C_1 and the variable of integration $\mathbf{q}$ is restricted to C_2. Now we can reasonably assume that the *relative* variation of $r(\mathbf{p}_i, \mathbf{q})$ is only small for $\mathbf{p}_i \in C_1$ and $\mathbf{q} \in C_2$. Therefore $G(r(\mathbf{p}_i, \mathbf{q}))$ varies only little too. We can in this case compress the corresponding subblock A_{12} to a rank-1 matrix with some small error. For example:

$$A_{12} = \begin{pmatrix} \int\limits_{\triangle_4} G(\mathbf{p}_1, \mathbf{q})\,\mathrm{d}q & \int\limits_{\triangle_5} G(\mathbf{p}_1, \mathbf{q})\,\mathrm{d}q & \int\limits_{\triangle_6} G(\mathbf{p}_1, \mathbf{q})\,\mathrm{d}q \\ \int\limits_{\triangle_4} G(\mathbf{p}_2, \mathbf{q})\,\mathrm{d}q & \int\limits_{\triangle_5} G(\mathbf{p}_2, \mathbf{q})\,\mathrm{d}q & \int\limits_{\triangle_6} G(\mathbf{p}_2, \mathbf{q})\,\mathrm{d}q \\ \int\limits_{\triangle_4} G(\mathbf{p}_3, \mathbf{q})\,\mathrm{d}q & \int\limits_{\triangle_5} G(\mathbf{p}_3, \mathbf{q})\,\mathrm{d}q & \int\limits_{\triangle_6} G(\mathbf{p}_3, \mathbf{q})\,\mathrm{d}q \end{pmatrix} \approx \frac{1}{\int\limits_{\triangle_5} G(\mathbf{p}_1, \mathbf{q})\,\mathrm{d}q} \begin{pmatrix} \int\limits_{\triangle_5} G(\mathbf{p}_1, \mathbf{q})\,\mathrm{d}q \\ \int\limits_{\triangle_5} G(\mathbf{p}_2, \mathbf{q})\,\mathrm{d}q \\ \int\limits_{\triangle_5} G(\mathbf{p}_3, \mathbf{q})\,\mathrm{d}q \end{pmatrix} \begin{pmatrix} \int\limits_{\triangle_4} G(\mathbf{p}_1, \mathbf{q})\,\mathrm{d}q & \int\limits_{\triangle_5} G(\mathbf{p}_1, \mathbf{q})\,\mathrm{d}q & \int\limits_{\triangle_6} G(\mathbf{p}_1, \mathbf{q})\,\mathrm{d}q \end{pmatrix}$$

For the above rank-1 approximation we would have to compute only one block row and column. We could extend this argumentation to the other off-diagonal block and furthermore to the other BEM matrices as well.

So the basic approach that follows from this example is to partition the domain $\triangle U$ into clusters of neighbouring triangles. Note that in the example the triangles of a cluster are indexed contiguously. Clusters correspond therefore to contiguous matrix rows/columns. A cluster product then corresponds to one matrix block. This is generally not the case. After having found suitable clusters, one will have to reindex the triangles. The general strategy is to find a set of clusters and then build a set of cluster products that correspond to a block partition of our system matrices. A natural goal is to find the coarsest block partition with the lowest possible block ranks, that still conforms to a certain accuracy requirement.

We start at the first step by giving a strategy for the clustering of the domain. For that we need some fundamental constructs.

6.1 Minimal axially parallel cuboids

Let the domain $\triangle U$ consist of n triangles. Let I be the index set of the triangles. We define $\triangle I$ to express the part of the triangle domain that is contained in the index set I. Formally $\triangle I \coloneqq \bigcup_{i \in I} \triangle_i$. We also extend this notation to subsets of I.

Definition 6.1 (Minimal axially parallel cuboid). *Let $\tilde{Q}_I$ denote the minimal axially parallel cuboid that contains $\triangle I$.*

$$\tilde{Q}_I \coloneqq \prod_{d=1}^{3} [\min\{\triangle I|_d\}, \max\{\triangle I|_d\}] \tag{6.1}$$

In above equation $\triangle I|_d$ is the restriction of $\triangle I$ to the d-th components of $\triangle I$. So $\triangle I|_1$ is the set of the x-components of the elements of $\triangle I$.

The cuboid $\tilde{Q}_I$ is computed by iterating over all corners of all triangles of I.

For the construction of the clusters we will use a variation of (6.1). We construct the minimal cuboid to an Index set I, so that it contains the collocation points that correspond to I.

$$Q_I \coloneqq \prod_{d=1}^{3} [\min_{\{\mathbf{p}_i | i \in I\}} \{\mathbf{p}_i|_d\}, \max_{\{\mathbf{p}_i | i \in I\}} \{\mathbf{p}_i|_d\}] \tag{6.2}$$

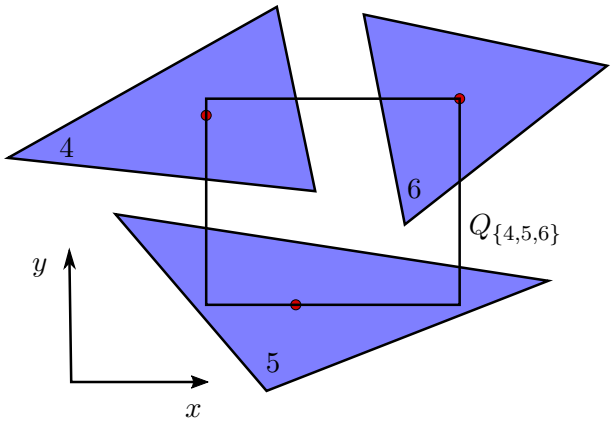
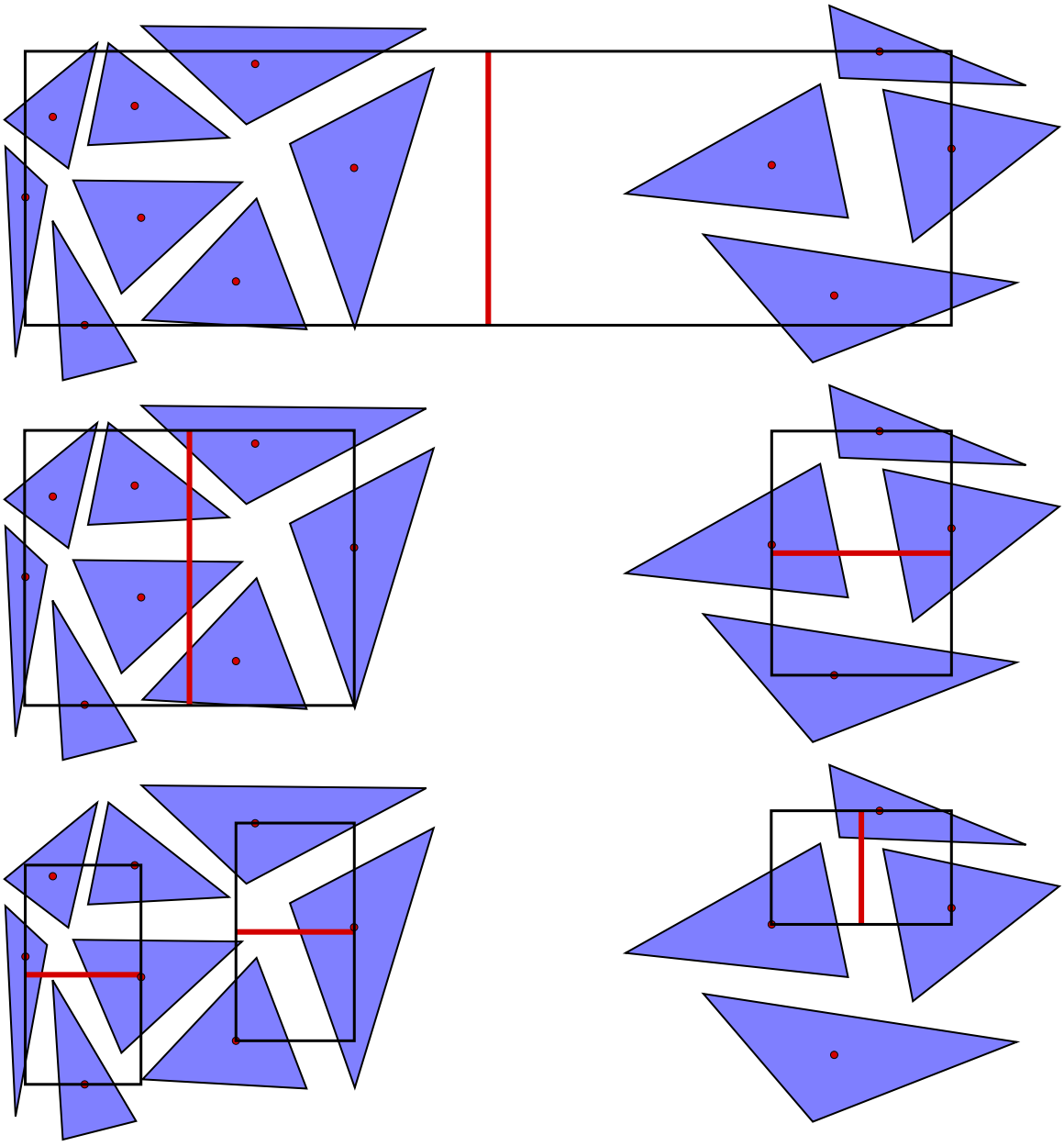


Figure 6.2: Minimal axially parallel cuboid $Q_{\{4,5,6\}}$

To calculate the minimal cuboid of an index set, one just has to find the maximum and minimum components of all the triangle nodes/collocation points of the set. The overall cost is linearly dependent on the set size.

$$C_{\tilde{Q}_I} = 18|I| \quad \text{and} \quad C_{Q_I} = 6|I|$$

We use the minimal cuboids to cluster/bipartition an Index set by halving its minimal cuboid along the cuboids longest dimension. The triangles are then partitioned, according to whether they lie in cuboid half one or cuboid half two. The following figures illustrate three iterations of successive nested clustering.



The following algorithm halves an axially parallel cuboid at its longest edge.

Algorithm 6.1: splitCuboid(Q)

input: Q // input cuboid
output: Q_1, Q_2 // two halves of the input cuboid, generated by splitting the input
// cuboid at its longest dimension
begin
// parametrisation of the 3D cuboid: $Q = \prod_{i=1}^{3}[a_i, b_i]$ with $b_i > a_i$
$d := \text{argmax}_{i \in \{1,2,3\}}\{b_i - a_i\}$ // find the longest dimension of Q

$l \coloneqq [b_d - a_d]$ // length of the longest edge of Q

$Q_1 \coloneqq \prod_{i<d}[a_i, b_i] \times [a_d, b_d - \frac{l}{2}] \times \prod_{i>d}[a_i, b_i]$ // Q_1 is half of Q
$Q_2 \coloneqq \prod_{i<d}[a_i, b_i] \times [a_d + \frac{l}{2}, b_d] \times \prod_{i>d}[a_i, b_i]$ // Q_2 is other half

return Q_1, Q_2
end

Now we formalize the bipartitioning of an Index set via its minimal cuboid.

Algorithm 6.2: splitCluster(I)

---

input: I // index set of triangles
output: I_1, I_2 // bipartition of the input set
begin
$I_1, I_2 \coloneqq \emptyset$ // initialize the subsets empty
$Q_I \coloneqq \text{minCuboid}(I)$ // calculate the minimal cuboid of the triangles of I
$(Q_1, Q_2) \coloneqq \text{splitCuboid}(Q_I)$ // halve the cuboid at its longest edge
for all $i \in I$ **do** // iterate over all triangle indices
if (midPoint($\triangle_i$) $\in Q_1$) // if the midpoint of triangle i lies in Q_1
$I_1 \coloneqq I_1 \cup \{i\}$ // add index to Set 1
else // if the midpoint of triangle i lies in Q_2
$I_2 \coloneqq I_2 \cup \{i\}$ // add index to Set 2
endelse
endfor

return I_1, I_2
end

The runtime dominant routines of algorithm (6.2) are minCuboid(I) and the iteration loop. Both have a complexity of $O(|I|)$.

6.2 The cluster tree T_I

There is a hardware dependent minimum cluster size. Partitioning beyond this minimum size just produces more computational overhead. We will denote this minimum size n_{min} throughout the remainder of this text.

Let I be an Index set. We define

$$\text{size}(I) \coloneqq \begin{cases} \text{true}, & \text{if } |I| \geq n_{min} \\ \text{false}, & \text{else} \end{cases}$$

Now we wrap the partitioning in a recursion. We also define a sons mapping from an index set I to its two subsets I_1 and I_2.

$$\text{sons}(I) \coloneqq (I_1, I_2) \tag{6.3}$$

The following algorithm implicitly creates a binary tree of nested, increasingly fine partitions.

Algorithm 6.3: recursiveSplit(I)

input: I // cluster
begin
    if size(I) // cluster I is large
        $(I_1, I_2) \coloneqq$ splitCluster(I) // bipartition the index set
        sons(I) $\coloneqq (I_1, I_2)$ // the sons-mapping connects a cluster
                                // to its bipartition
        recursiveSplit(I_1) // I_1 might need further splitting
        recursiveSplit(I_2)
    else // cluster I is small
        sons(I) $\coloneqq \emptyset$ // no further splitting $\rightarrow$ no sons
    endelse
end

The splitCluster(I) call in algorithm (6.3) has costs of $O(|I|)$. The splitCluster calls in the next recursion step cost $O(|I_1| + |I_2|) = O(|I|)$. Let I_{start} be the index set from the first call of the recursion. The overall cost is then the recursion depth times $|I_{start}|$.
The next algorithm only serves to bring structure to the tree creation, by setting an explicit root node and defining an unambiguous entry point for the splitting recursion. For that we define a map root, which returns the root cluster of a tree.

Algorithm 6.4: constructClusterTree(I)

input: I // set of triangle indices
output: T_I
begin
    T_I // initialize T_I
    root(T_I) $\coloneqq I$ // the entire index set is the root cluster
    recursiveSplit(root(T_I)) // bipartition the root cluster

    return T_I
end

Algorithm (6.4) constructively describes a binary tree structure. We call it the cluster tree T_I [Hac09, Section 5.3]. The root vertex of the cluster tree contains the index set of the entire geometry. The sons-mapping associates a larger cluster with its bipartition. The tree structure encodes the nesting hierarchy of the clusters; from the largest cluster, the root, to the smallest clusters, the tree leaves. We call the set of leaves of a tree T to $\mathcal{L}(T)$. The set of leaves of a cluster tree T_I yields a partition of the index set I. Furthermore the same is true for any subtree of T_I, which has the some root vertex as T_I. During the construction of T_I, every level has a memory cost of $O(n)$. For a "balanced" geometry, where a splitcluster call about halves a cluster, the tree depth is $O(\log_2(n))$.

This leads to an overall complexity of $O(n \log_2(n))$ for the construction of a cluster tree for appropriate geometries.

The number of leaves of a tree T is bound by $|I|$. It can be shown by induction that a binary tree T has $2|\mathcal{L}(T)| - 1$ vertices. The total number of clusters of a cluster tree is therefore also bound by n.

Let $\gamma \subset I$ be a cluster and T_I a cluster tree. We write $\gamma \in T_I$, to express that γ is a vertex in the cluster tree T_I and that there exists a sons mapping in accordance with algorithm 6.3, as long as $\gamma \notin \mathcal{L}(T)$. We also define the map depth(T_I) to return the tree depth of T_I. Furthermore the set of clusters at tree level l shall in the following refer to the clusters that were created at recursion depth l. We will describe this set by $T_I^{(l)}$.

Figure 6.3: Unbalanced geometry

Figure (6.3) suggests a worst case geometry where splitCluster(I) (algorithm 6.2) only splits off one triangle per execution. This would lead to an unbalanced tree with depth n and therefore an overall runtime and memory consumption of $O(n^2)$. One can amend the runtime by informing the cluster split more on the cardinality of the resulting sets. A cardinality based split enforces a runtime of $O(n \log_2(n))$ for any geometry. This encouraging worst-case behaviour is offset by a less efficient (for the construction of low rank blocks) clustering result on well formed geometries. In the end, the decision for a clustering strategy is geometry specific. We will now introduce the cardinality based clustering strategy.

6.3 Cardinality-based clustertree

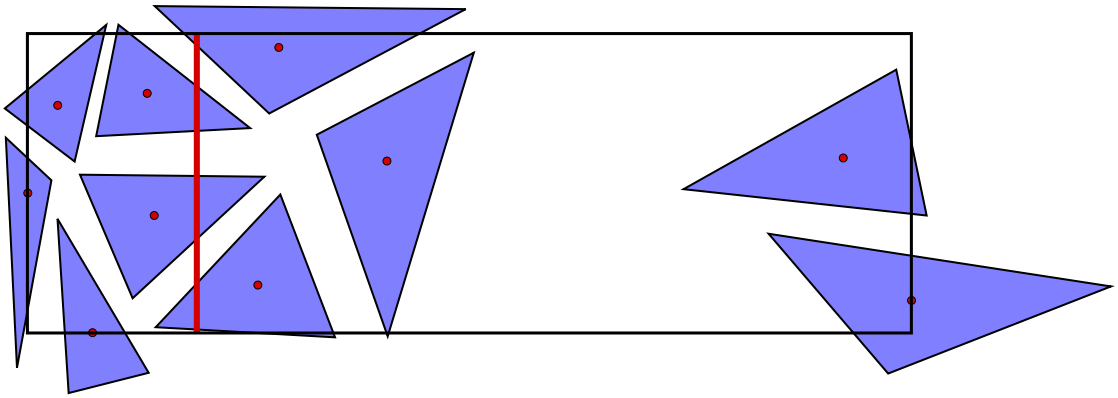


Figure 6.4: Cardinality-based cluster splitting

Figure (6.4) illustrates the main difference to the geometrically informed cluster split. The minimal cuboid is partitioned again at its longest dimension; but at a position, where the split balances the cardinality of the resulting subsets. Let us momentarily assume

the minimal cuboid is the longest in the x direction. For the decision in which subset a triangle ends up, we need a list of the triangle indices, ordered by the x-component of their midpoints. Sorting generally takes $O(n \log(n))$ time (for example with mergesort), so we want to sort only once for each geometric dimension at the beginning of the tree construction.

We give the tree construction algorithm for the cardinality based clustertree construction this time the other way around; we introduce the wrapper function first. Again; the main difference at this stage to the geometric clustertree construction is, that we need to compute three sorted tuples. Let I be a set of triangle indices. We define I_x to be the tuple, that results from sorting I by the x-component of the triangles midpoints. I_y and I_z are defined analogously. Such a tuple can be obtained from I with a sorting algorithm like mergesort in $O(n \log(n))$ time.

Algorithm 6.5: constructClusterTreeCardinality(I)

input: I // set of triangle indices
begin
    root($T(I)$) $\coloneqq I$ // the entire index set is the root node
    compute I_x^{root}, I_y^{root}, I_z^{root} // I as tuples, sorted by the x,y or y-component
                                                    // of the triangles midpoints; can be done in $O(n \log(n))$
                                                    // with mergesort
    recursiveSplitCardinality($I_x^{root}, I_y^{root}, I_z^{root}$) // bipartition the root cluster
end

Again the splitting routine is used recursively:

Algorithm 6.6: recursiveSplitCardinality(I_x, I_y, I_z)

input: I_x, I_y, I_z // tuples of triangle indices, sorted by the x,y or z-component
                          // of their midpoints; all tuples contain the same elements I
begin
    if size(I) // cluster I is large
        // bipartitioning the index set into I^1 and I^2
        $(I_x^1, I_y^1, I_z^1, I_x^2, I_y^2, I_z^2) \coloneqq$ splitClusterCardinality(I_x, I_y, I_z)
        sons$(I) \coloneqq (I^1, I^2)$ // the sons-mapping connects a cluster
                                    // to its bipartition
        recursiveSplitCardinality(I_x^1, I_y^1, I_z^1) // I^1 might need further splitting
        recursiveSplitCardinality(I_x^2, I_y^2, I_z^2) // ...
    else // cluster I is small
        sons$(I) \coloneqq \emptyset$ // no further splitting $\rightarrow$ no sons
    endelse
end

Algorithm 6.7: splitClusterCardinality(I_x, I_y, I_z)

input: I_x, I_y, I_z // tuples of triangle indices, sorted by the x,y or z-component
// of their midpoints; all tuples contain the same elements I
var: isInOne // a global boolean array, where we can record whether an input index
// goes into the output set I_1
output: I_x^1, I_y^1, I_z^1, I_x^2, I_y^2, I_z^2, // bipartitions of the input tuples
begin

$I_x^1, I_y^1, I_z^1, I_x^2, I_y^2, I_z^2 := \emptyset$ // initialize the output tuples empty

$Q_I :=$ minCuboid(I) // calculate the minimal cuboid of the triangles of I
// parametrisation of the 3D cuboid: $Q_I = \prod_{i\in(x,y,z)}[a_i, b_i]$ with $b_i > a_i$
$d := \text{argmax}_{i\in\{x,y,z\}}\{b_i - a_i\}$ // find the longest dimension of Q_I

$I_d^1 := I_d(1 : \frac{|I|}{2})$ // the elements of I_d^1 are the first $\frac{|I|}{2}$ elements of I_d
// I_d^1 is also a sorted tuple
$I_d^2 := I_d(\frac{|I|}{2} : |I|)$ // the elements of I_d^2 are the last $\frac{|I|}{2}$ elements of I_d
// I_d^2 is also a sorted tuple
// the last part of the routine only serves to partition the other two tuples
// in a way that their 'sons' are still ordered

// use the global array isInOne to record, whether an index is in output set one
// or two
for all $i \in I_d^1$
isInOne$_i$:= true
endfor
for all $i \in I_d^2$
isInOne$_i$:= false
endfor

// at this point we have bipartitioned the triangle set I into two sets I^1 and I^2
// according to the d-element of the triangles midpoints
for all $p \in \{x, y, z\} \setminus \{d\}$ // iterate over the other two dimensions
for all $i \in I_p$ // iterate over all indices in I_p according to their ordering
if isInOne$_i$ // true if $i \in I_d^1$
append i to I_p^1
else // if $i \in I_d^2$
append i to I_p^2
endelse
endfor
endfor

return I_x^1, I_y^1, I_z^1, I_x^2, I_y^2, I_z^2
end

Just for clarity the line '$I_d^1 := I_d(1 : \frac{|I|}{2})$' means, that the first half of the sorted tuple I_d is written to the 'son' tuple I_d^1.

As algorithm (6.7) halves an index set during each call, the tree depth is guaranteed to be $O(n \log_2(n))$.

6.4 Reindexing

The following reindexing algorithm traverses the clustertree in a depth-first manner (See depth-first search for reference). When the reindexing algorithm is called onto a cluster, that is a leaf in the clustertree, it reindexes the corresponding triangles to be contiguously increasing and starting at a global counter value. The global counter value tracks the number of all already reindexed triangles. If the algorithm is called onto an inner vertex of the clustertree, it is first recursively called onto the vertex' sons and afterwards updates the vertex' indices to the union of its sons' indices. After running the algorithm, every cluster corresponds to a continuous vector segment with regards to the linear system (3.10). A cluster×cluster product then corresponds analogously to a matrix blocks in the system (3.10). Let $\gamma \in T_I$ be a cluster index set.

Algorithm 6.8: reindexCluster(γ)

input: γ // cluster; $\gamma \subseteq I$
var: globalCounter // is global integer variable;
// it counts the number of already reindexed triangles
begin
if isLeaf(γ) = $false$ // cluster γ is split further; equivalently $\text{sons}_{T_I}(\gamma) \neq \emptyset$
$\{\gamma', \gamma''\} \coloneqq \text{sons}_{T_I}(\gamma)$ // get sons of cluster γ

// save the value of the reindexing counter
countAtEntry := globalCounter

// recursively call reindexCluster on the sons of γ
reindexCluster(γ')
reindexCluster(γ'')

// the value of globalCounter will have increased,
// due to the reindexCluster calls above

// update the cluster (triangle indices of) γ to the be
// the union of its sons (' triangle sets);
// formally: $\gamma \coloneqq \gamma' \cup \gamma''$; this can be done efficiently via:
$\gamma \coloneqq \{\text{countAtEntry}, ..., \text{globalCounter}\}$

else
// reindex the triangles that correspond to γ
// with the new index set $\{\text{globalCounter}, ..., \text{globalCounter} + |\gamma|\}$

// overwrite also the cluster indices with the new index set
$\gamma \coloneqq \{\text{globalCounter}, ..., \text{globalCounter} + |\gamma|\}$

globalCounter = globalCounter $+|\gamma|$ // increment the counter variable
endelse
end

Again we put the recursion in a wrapper to make clear, that a cluster tree is reindexed; also to give the 'globalCounter' variable an initial value.

Algorithm 6.9: reindexClusterTree(T_I)

input: T_I // clustertree
begin
    globalCounter := 1 // initialize the counter with one;
                       // the counter is referenced by reindexCluster
    reindexCluster(root(T_I)) // start reindexing recursion
end

The reindexing in the leaf nodes is done only once per triangle. This causes costs of $O(n)$. Changing the cluster index set in the inner vertices of the clustertree can be done in constant time. As the total number of vertices is $O(n)$, the overall cost of the reindexing algorithm is also linear in n.

7 $\mathcal{H}$-matrices

7.1 The blockclustertree $T_{I\times J}$

The clustertree T_I gives us hierarchized partitions to a vector $\mathbf{x} \in \mathbb{C}^I$. We are interested in block partitions for matrices over $\mathbb{C}^{I\times J}$. Similarly to the clustertree, we construct a hierarchical collection of block clusters. We call this structure the block cluster tree $T_{I\times J}$ [Hac09, Section 5.5]. The block cluster tree is built out of the clusters of T_I and T_J.

Algorithm 7.1: recursiveSplitBlock($\gamma \times \delta$)

input: $\gamma \times \delta$ // block cluster, $\gamma \in T_I$ and $\delta \in T_J$
begin
    if $\text{sons}_{T_I}(\gamma) = \emptyset$ or $\text{sons}_{T_J}(\delta) = \emptyset$ // if any cluster is leaf in clustertree
        $\text{sons}_{T(I\times J)}(\gamma \times \delta) := \emptyset$ // block cluster is leaf in blockclustertree
    else // both clusters are not leafs in their clustertree
        $\{\gamma', \gamma''\} := \text{sons}_{T_I}(\gamma)$ // get sons of cluster γ
        $\{\delta', \delta''\} := \text{sons}_{T_J}(\delta)$ // get sons of cluster δ

        // the block $\gamma \times \delta$ is split into four subblocks
        $\text{sons}_{T(I\times J)}(\gamma \times \delta) := \{\gamma' \times \delta', \gamma' \times \delta'', \gamma'' \times \delta', \gamma'' \times \delta''\}$
        recursiveSplitBlock($\gamma' \times \delta'$)
        recursiveSplitBlock($\gamma' \times \delta''$)
        recursiveSplitBlock($\gamma'' \times \delta'$)
        recursiveSplitBlock($\gamma'' \times \delta''$)
    endelse
end

Algorithm 7.2: constructBlockClusterTree(T_I, T_J)

input: T_I, T_J // cluster trees for both index sets
begin
    $\text{root}(T_{I\times J}) := I \times J$ // $\text{root}(T(I \times J)) := \text{root}(T_I) \times \text{root}(T_J)$
    // the root node corresponds to an unpartitioned matrix

    recursiveSplitBlock($I \times J$) // recursively split the block into four subblocks
end

The block cluster tree resulting from algorithm (7.2) is quaternary. If a block $b \in T_{I\times J}$

and $b \notin \mathcal{L}(T_{I\times J})$, then the block is cross split into four subblocks. It holds

$$\operatorname{depth}(T_{I\times J}) = \min\{\operatorname{depth}(T_I), \operatorname{depth}(T_J)\}. \tag{7.1}$$

Similarly to cluster trees, the leaves of a block cluster tree $T_{I\times J}$ constitute a partition of $I \times J$; the same holds for any subtree that is also rooted in $I \times J$. The constructed block cluster tree gives us hierarchically nested partitions. The fineness of the partitions increases with the tree depth. We basically want the coarsest possible partition, where all large blocks can still be approximated with low rank matrices. Whether a block cluster can be efficiently approximated is expressed with a so called admissibility condition.

7.2 Matrix partition

Definition 7.1 (Matrix partition). *Let $T_{I\times J}$ be a given block cluster tree. P is called a partition of $I \times J$ [Hac09, Definition 5.5.6], if*

- $P \subset T_{I\times J}$ *(P is consistent to $T(I \times J)$)*
- $b,\, b' \in P \implies (b = b' \text{ or } b \cap b' = \emptyset)$ *(disjointness)*
- $\cup_{b\in P} b = I \times J$ *(disjunct cover)*

We extend the definition of the size function for clusters to block clusters. Let $b = \gamma \times \delta \in T_{I\times J}$ be a block cluster. We define

$$\operatorname{size}_{T_{I\times J}}(\gamma \times \delta) :\Longleftrightarrow (\operatorname{size}_{T_I}(\gamma) \text{ and } \operatorname{size}_{T_J}(\delta))$$

Definition 7.2 (admissible matrix partition). *Let there be an admissibility condition* Adm *for block clusters given (specified in definition 8.5). We call a partition P admissible, if*

$$\begin{gathered} \textit{either } \operatorname{Adm}(b) = \text{true} \quad \textit{or} \quad \operatorname{size}_{T_{I\times J}}(b) = \text{false} \\ \textit{for all } b \in P. \end{gathered} \tag{7.2}$$

A partition is admissible if all its blocks are either low-rank approximable or small. For now the reader can use the intuition from our clustering example (page 54); a block is admissible if the clusters are relatively distant.

The following algorithm returns the coarsest (minimal) admissible partition in the blockclustertree.

Algorithm 7.3: minimalAdmissiblePartitionRecursion(b)

```
input: b // block cluster
var: P
begin
        if Adm(b) = true or size(b) = false
                P := P ∪ b
        else
                for all b' ∈ sons(b)
                        minimalAdmissiblePartitionRecursion(b')
                endfor
        endelse
end
```

Algorithm 7.4: minimalAdmissiblePartition($T(I \times J)$)

```
input: T(I × J) // blockclustertree
begin
        P := ∅ // initialize the partition
        // call the recursion on the root block
        minimalAdmissiblePartitionRecursion(root(T(I × J)))
end
```

We write Note that as the clustertree T_I has $O(n)$ leaves, the block cluster $T_{I\times I}$ tree has $O(n^2)$ leaves. It is therefore too costly to construct $T_{I\times I}$ fully and then to determine the (coarser) partition afterwards. A practical implementation would integrate the admissibility condition directly into the cluster tree recursion (algorithm 7.1) as an additional stopping criterion. We define the notation $T_{I\times J}(P)$ or $T(I \times J, P)$ for a block cluster tree $T_{I\times J}$, that has been trimmed beneath the partition blocks. So it holds for a tree $T_{I\times J}(P)$:

$$\mathcal{L}(T_{I\times J}(P)) = P$$

We further define a shorthand for the set of admissible/inadmissible partition blocks:

Definition 7.3 (Near- and far-field). *Let P be an admissible Partition. Then the near-field P^- and the far-field P^+ are defined as follows:*

$$P^- := \left\{ b \in P \middle| \operatorname{size}_{T(I\times J)}(b) = \text{false} \right\}, \quad P^+ := P \setminus P^-$$

The denotations near- and far-field directly relate to our argumentation on page 54. The near-field blocks are small and will be represented by full matrices. The admissible far-field blocks are treated with low-rank approximations. Finally we can give a tangible definition of $\mathcal{H}$-matrices.

Definition 7.4 ($\mathcal{H}$-matrix). *[Hac09, Definition 6.1.1] Let I and J be index sets, $T_{I\times J}$ a block cluster tree and P an admissible partition. Also let k be a local block rank distribution*

$$k : P \to \mathbb{N}.$$

Then the set $\mathcal{H}(k,P) \subset \mathbb{C}^{I\times J}$ of hierarchical matrices with Partition P and rank distribution k is composed of all $M \in \mathbb{C}^{I\times J}$ with

$$\operatorname{rank}(M|_b) \leq k(b) \quad \text{for all } b \in P^+.$$

More specifically it shall hold for all blocks $b \in P^+$, that $M|_b \in \mathcal{R}(k,b)$. The affiliation of M with the block cluster tree $T_{I\times J}$ is expressed through P.

8 Admissibility

In this chapter we will concern ourselves with the question, whether a matrix block is low-rank approximable. An approximable block will be referred to as ***admissible***. We will touch briefly on the standard argumentation for the efficiency of $\mathcal{H}$-matrix representations. The standard case is mainly concerned with the discretisation of asymptotically smooth kernel functions ([BGH03, section 3.2.2]). Unfortunately the Green's function (equation 3.2) is not asymptotically smooth and the $\mathcal{H}$-matrix technique loses efficiency as we go up in frequency. We describe the details in the following.

We start with some necessary definitions and observations for the standard case.

8.1 Admissibility for asymptotically smooth kernels

Definition 8.1 (Degenerate kernel). *We call a kernel function $g_k : U_\gamma \times U_\delta \to \mathbb{C}$ degenerate, if it can be expressed as a finite sum of functions with separated variables.*

$$g_k(\mathbf{p}, \mathbf{q}) = \sum_{l=1}^{k} u_l(\mathbf{p}) v_l(\mathbf{q}) \tag{8.1}$$

The number $k \in \mathbb{N}^+$ is called the separation rank.

Low-rank matrices are naturally suitable for discretization of degenerate kernels. Let the matrix $M_{\gamma\times\delta} \in \mathbb{C}^{\gamma\times\delta}$ be the discretisation of the degenerate kernel $g_k(\mathbf{p}, \mathbf{q})$:

$$(M_{ij})_{i\in\gamma,\ j\in\delta} = \int_{\triangle_j} g_k(\mathbf{p_i}, \mathbf{q})\,\mathrm{d}q = \sum_{l=1}^{k} u_l(\mathbf{p_i}) \int_{\triangle_j} v_l(\mathbf{q})\,\mathrm{d}q$$

Then $M_{\gamma\times\delta}$ has the rank-k factorization $M_{\gamma\times\delta} = \sum_{l=1}^{k} u^l v^{l\mathsf{T}}$ with

$$u^l = \begin{pmatrix} u_l(\mathbf{p}_{i_1}) \\ \vdots \\ u_l(\mathbf{p}_{i_{|\gamma|}}) \end{pmatrix}_{i_1,\ldots,i_{|\gamma|}\in\gamma} \quad \text{and } v^{l\mathsf{T}} = \begin{pmatrix} \int_{U_{j_1}} v_l(\mathbf{q})\,\mathrm{d}q & \cdots & \int_{U_{j_{|\delta|}}} v_l(\mathbf{q})\,\mathrm{d}q \end{pmatrix}_{j_1,\ldots,j_{|\delta|}\in\delta}.$$

Lemma 8.1 (Separable expansion via Taylor series). *Let $\mathbf{p}_0 \in U_I$ be and $g(\mathbf{p}, \mathbf{q}) \in C^\infty(U_I \times U_J)$ a kernel function. We arrive at a separable expansion of g via its Taylor series with center point $\mathbf{p}_0$ and residuum R_k:*

$$g(\mathbf{p}, \mathbf{q}) = \sum_{\alpha\in\mathbb{N}_0^3, |\alpha|\le l} (\mathbf{p} - \mathbf{p}_0)^\alpha \frac{1}{\alpha!} \partial_p^\alpha g(\mathbf{p}_0, \mathbf{q}) + R_k(\mathbf{p}, \mathbf{q}), \tag{8.2}$$

with $k = \binom{l+3}{3} = |\{\alpha \mid \alpha \in \mathbb{N}_0^3 \text{ and } |\alpha| \leq l\}|$ *as the separation rank of the Taylor polynomial.*

The Taylor polynomial is generally not convergent. We require additional conditions for g.

Definition 8.2 (Asymptotically smooth function). *We call a function* $g(\mathbf{p}, \mathbf{q}) \in C^\infty(U_\gamma \times U_\delta)$ *asymptotically smooth, if*

$$|\partial_p^\alpha \partial_q^\beta g(\mathbf{p}, \mathbf{q})| \leq \frac{C(\alpha + \beta)!}{(c_0 \|\mathbf{p} - \mathbf{q}\|)^{|\alpha|+|\beta|+\sigma}} \tag{8.3}$$

holds for constants $C, c_0 \in \mathbb{R}_{>0}$, $\sigma \in \mathbb{R}$ *[BGH03, page 46, equation 3.14]. The* $\alpha, \beta \in \mathbb{N}_0^3$ *are multi-indices for the notation of higher-order partial derivatives.*

An asymptotically smooth function $g(\mathbf{p}, \mathbf{q})$ decreases in the absolute with the distance between $\mathbf{p}$ and $\mathbf{q}$ (for $\sigma = 0$). The "order" of the decrease rises with the order of the derivatives.

If we approximate an asymptotically smooth kernel function $g(\mathbf{p}, \mathbf{q})$ with its Taylor polynomial $g_l(\mathbf{p}, \mathbf{q})$ of order l at a center point $\mathbf{p}_0$, we can estimate

$$\begin{aligned} |g(\mathbf{p}, \mathbf{q}) - g_l(\mathbf{p}, \mathbf{q})| &= |R_{k(l)}(\mathbf{p}, \mathbf{q})| \\ &\leq C \sum_{\alpha \in \mathbb{N}_0^3, |\alpha| > l} \frac{|(\mathbf{p} - \mathbf{p}_0)^\alpha|}{\alpha!} \frac{\alpha!}{(c_0 \|\mathbf{p}_0 - \mathbf{q}\|)^{|\alpha|+\sigma}} \\ &= C \sum_{\alpha \in \mathbb{N}_0^3, |\alpha| > l} \frac{|(\mathbf{p} - \mathbf{p}_0)^\alpha|}{(c_0 \|\mathbf{p}_0 - \mathbf{q}\|)^{|\alpha|+\sigma}} \\ &\leq C \sum_{\alpha \in \mathbb{N}_0^3, |\alpha| > l} \frac{\|\mathbf{p} - \mathbf{p}_0\|^{|\alpha|}}{(c_0 \|\mathbf{p}_0 - \mathbf{q}\|)^{|\alpha|+\sigma}} \end{aligned} \tag{8.4}$$

We can see from the above estimate that the Taylor approximation is convergent with $O(\exp(l))$ in its order l for $\frac{\|\mathbf{p}-\mathbf{p}_0\|}{(c_0\|\mathbf{p}_0-\mathbf{q}\|)} < 1$. The order l of the Taylor polynomial relates to the number of monomial k as $k(l) = \binom{l+3}{3}$. It follows $l(k) = O(\sqrt[3]{k})$.

Theorem 8.1 (Degenerate kernel expansion). *If a kernel function* $g : U_I \times U_J \to \mathbb{C}$ *is asymptotically smooth, then it can locally be approximated by a degenerate kernel [GH03, Section 3.2]:*

$$\begin{aligned} g(\mathbf{p}, \mathbf{q}) &= g_k(\mathbf{p}, \mathbf{q}) + R_k(\mathbf{p}, \mathbf{q}) \\ &= \sum_{i=0}^{k} u_i^{(k)}(\mathbf{p}) v_i^{(k)}(\mathbf{q}) + R_k(\mathbf{p}, \mathbf{q}) \end{aligned} \tag{8.5}$$

such that

$$\max_{(\mathbf{p},\mathbf{q}) \in U_\gamma \times U_\delta} |R_k(\mathbf{p}, \mathbf{q})| = O(C_{\gamma,\delta}^{\sqrt[3]{k}}) \tag{8.6}$$

for a block $\gamma \times \delta \in T_{I \times J}$, where the constant $C_{\gamma,\delta} < 1$ depends on the ratio of the distance between the clusters and the cluster size (specified in definition 8.5). We call the right-hand side of (8.5) a degenerate expansion of g in $U_\gamma \times U_\delta$ with rank k and remainder R_k.

Note that we used the Taylor approximation only as a theoretical argument. A practical approach to generating low-rank approximations based on interpolation can be found in [BGH03, Section 3.2]. In contrast we employ a purely algebraic method, which is described in section 10.3.

The following definitions will help us to clearly state the conditions under which there exist a degenerate kernel expansion for a block.

Definition 8.3 (Diameter of a domain). *We define the diameter of a domain U to be the maximal distance of two points of U.*

$$\operatorname{diam}(U) = \max_{\mathbf{p},\mathbf{q} \in U} \|\mathbf{p} - \mathbf{q}\|_2 \tag{8.7}$$

As $\operatorname{diam}(U)$ is generally difficult to compute, we resort to approximating a cluster domain with its minimal axially parallel cuboid (definition 6.1).

Lemma 8.2 (Diameter of cuboids). *Let γ be a cluster, $U\gamma$ its domain and $\tilde{Q}_\gamma$ be the containing minimal axis parallel cuboid. As $U\gamma \subseteq \tilde{Q}_\gamma$, the following holds:*

$$\operatorname{diam}(U\gamma) \leq \operatorname{diam}(\tilde{Q}_\gamma). \tag{8.8}$$

Let the parametrization of $\tilde{Q}_\gamma$ be $\tilde{Q}_\gamma = \prod_{d=1}^{3} [a_d, b_d]$, with $a_d \leq b_d$. The diameter of $\tilde{Q}_\gamma$ is then calculated as

$$\operatorname{diam}(\tilde{Q}_\gamma) = \sqrt{\sum_{d=1}^{3} (b_d - a_d)^2}.$$

Definition 8.4 (Distance between two domains). *We define the distance between two domains U and U' to be the minimum distance of two points, each from one domain.*

$$\operatorname{dist}(U, U') = \min_{\mathbf{p} \in U, \mathbf{q} \in U'} \|p - q\|_2$$

Again we resort to cuboids to make the last expression easily computable.

Lemma 8.3 (Distance between cuboids). *The distance between two cuboids Q and Q' is calculated as follows:*

$$\operatorname{dist}(Q, Q') = \sqrt{\sum_{d=1}^{3} (\operatorname{dist}([a_d, b_d], [a'_d, b'_d]))^2}$$

Furthermore for domains U and U', with $U \subseteq Q$ and $U' \subseteq Q'$, it holds

$$\operatorname{dist}(U, U') \geq \operatorname{dist}(Q, Q'). \tag{8.9}$$

Now that we have defined distance and diameter of a domain, we can formalize the condition for a block to have an exponentially convergent degenerate expansion for asymptotically smooth kernels. For $\mathbf{p}, \mathbf{p}_0 \in U\gamma$ and $\mathbf{q} \in U_\delta$ it was required $\frac{\|\mathbf{p}-\mathbf{p}_0\|}{(c_0\|\mathbf{p}_0-\mathbf{q}\|)} < 1$ (in equation 8.4).

Definition 8.5 (η-admissibility of a block). *Let $b = \gamma \times \delta \in T(I \times J)$ be a block cluster and $\eta > 0$. We denote the related (triangle mesh) domains U_γ and U_δ. The block b is called η-admissible, if*

$$\min\{\operatorname{diam}(U_\gamma), \operatorname{diam}(U_\delta)\} \leq \eta \operatorname{dist}(U_\gamma, U_\delta).$$

Theorem 8.2 (Convergence on η-admissible blocks). *With above definition we can say, a matrix block b of the discretisation of an asymptotically smooth kernel function has an exponentially convergent low rank approximation, if b is η-admissible with $\eta < c_0$ (constant from definition 8.3).*

As with $\operatorname{diam}(U_\gamma)$ and $\operatorname{dist}(U_\gamma, U_\delta)$ we carry over the concept of admissibility to cuboids. If we construct the minimal cuboids for the clusters γ and δ based on the triangle endpoints (construction 6.1), the admissibility condition for the cuboid block directly implies the admissibility condition for the cluster block.

Lemma 8.4 (η-admissibility with cuboids). *Let $b = \gamma \times \delta$ be a block, U_δ and U_γ the corresponding triangle mesh domains and $\tilde{Q}_\gamma$ and $\tilde{Q}_\delta$ the respective minimal cuboids in the style of (6.1). Then the following condition implies η-admissibility for the block b:*

$$\min\{\operatorname{diam}(\tilde{Q}_\gamma), \operatorname{diam}(\tilde{Q}_\delta)\} \leq \eta \operatorname{dist}(\tilde{Q}_\gamma, \tilde{Q}_\delta).$$

Above lemma results directly from the inequalities (8.8) and (8.9).

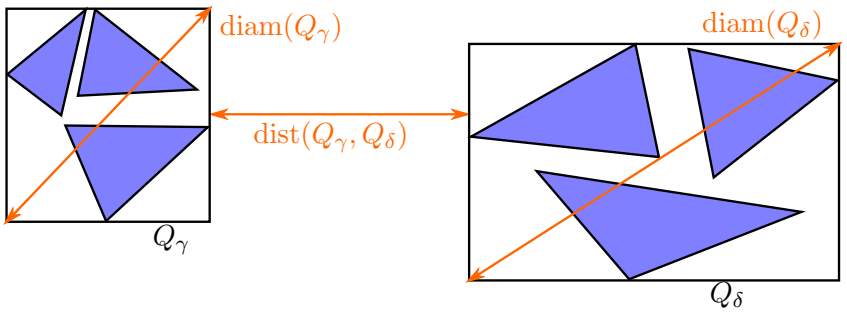


Figure 8.1: η-admissibility with cuboids

If we use the collocation point based minimal cuboids (equation 6.2) for the calculation of $\operatorname{diam}(U_\gamma)$ and $\operatorname{dist}(U_\gamma, U_\delta)$, lemma 8.4 does not hold. One could either employ the following inequalities:

$$\operatorname{diam}(\tilde{Q}_\gamma) \leq \operatorname{diam}(Q_\gamma) + \max_{i \in \gamma}(\operatorname{diam}(\tilde{Q}_{\triangle i}))$$

and

$$\operatorname{dist}(\tilde{Q}_\gamma, \tilde{Q}_\delta) \geq \operatorname{dist}(Q_\gamma, Q_\delta) - \max_{i \in \gamma \cup \delta}(\operatorname{diam}(\tilde{Q}_{\triangle i})).$$

The correctness of the admissibility condition is only critical for large blocks, where dist() and diam() differ relatively little for the two types of cuboids. We could therefore also just ignore the difference between $\operatorname{diam}(\tilde{Q})$ and $\operatorname{diam}(Q)$, et etcetera.

8.2 Admissibility for the Helmholtz kernel

If we were to deal with a BEM system with asymptotically smooth kernel functions, the η-admissibility condition as given above would be sufficient to ensure the existence of accurate low rank representations for admissible blocks. Unfortunately the Green's function (equation 3.2) and therefore its derivatives are only asymptotically smooth, when the Helmholtz equation reduces to the Laplace equation ($\kappa = 0$).

The fundamental solution has a separable expansion, its so called multipole expansion. A complete characterization of this expansion can be found in [Sto04, Satz 3.1.2] and [Des17, equation 2.32]. An accurate low rank approximation of this expansion is not within reach with standard methods like polynomial interpolation [BM17, section 2.2.1].

The upper bound we had for asymptotically smooth functions is weakened in the case of the Helmholtz kernel function $G_\kappa(\mathbf{p}, \mathbf{q})$ to

$$|\partial_p^\alpha \partial_q^\beta G_\kappa(\mathbf{p}, \mathbf{q})| \leq C(\alpha + \beta)! \frac{(1 + \kappa \|\mathbf{p} - \mathbf{q}\|)^{|\alpha|+|\beta|}}{(c_0 \|\mathbf{p} - \mathbf{q}\|)^{|\alpha|+|\beta|+\sigma}}, \tag{8.10}$$

with constants $C, c_0 \in \mathbb{R}_{>0}$, $\sigma \in \mathbb{R}$ and multi-indices $\alpha, \beta \in \mathbb{N}_0^3$ [BH07, equation 2.11].

We recreate the estimate for the residuum of the order-l Taylor polynomial approximation (equation 8.5) with the upper bound (8.10).

$$\begin{aligned}
|G_\kappa(\mathbf{p}, \mathbf{q}) - G_\kappa^l(\mathbf{p}, \mathbf{q})| &= |R_{k(l)}(\mathbf{p}, \mathbf{q})| \\
&\leq C \sum_{\alpha \in \mathbb{N}_0^3, |\alpha| > l} \frac{|(\mathbf{p} - \mathbf{p}_0)^\alpha|}{\alpha!} \frac{\alpha!(1 + \kappa \|\mathbf{p} - \mathbf{q}\|)^{|\alpha|}}{(c_0 \|\mathbf{p}_0 - \mathbf{q}\|)^{|\alpha|+\sigma}} \\
&\leq C(c_0 \|\mathbf{p}_0 - \mathbf{q}\|)^{-\sigma} \sum_{\alpha \in \mathbb{N}_0^3, |\alpha| > l} \left(\frac{\|\mathbf{p} - \mathbf{p}_0\| (1 + \kappa \|\mathbf{p} - \mathbf{q}\|)}{(c_0 \|\mathbf{p}_0 - \mathbf{q}\|)} \right)^{|\alpha|}
\end{aligned} \tag{8.11}$$

We see that the right-hand side tends to zero for $\|\mathbf{p} - \mathbf{p}_0\| (1 + \kappa \|\mathbf{p} - \mathbf{q}\|) < c_0 \|\mathbf{p}_0 - \mathbf{q}\|$. If we develop the Taylor polynomial for a point $\mathbf{q}_0$, we gather $\|\mathbf{q} - \mathbf{q}_0\| (1 + \kappa \|\mathbf{p} - \mathbf{q}\|) < c_0 \|\mathbf{q} - \mathbf{q}_0\|$ as another existence criterion for a local convergent degenerate expansion. We can derive a strict admissibility condition from the above:

$$\min\{\operatorname{diam}(Q_\gamma), \operatorname{diam}(Q_\delta)\} < \eta \frac{\operatorname{dist}(Q_\gamma, Q_\delta)}{1 + \kappa \operatorname{dist}(Q_\gamma, Q_\delta)} \quad \text{with } \eta < c_0$$

This condition is fulfilled by fewer blocks as we go up in frequency (growing κ). The $\mathcal{H}$-matrix method therefore loses efficiency in the high frequency domain. We follow

[CDC17, section 6.2] in that we resort in our implementation to the original admissibility condition for asymptotically smooth kernels (definition 8.5) with a fixed η. Although the authors of [CDC17] locate the constant c_0 below $\frac{1}{2}$, they achieve overall better compression with $\eta = 3$. Our implementation recreated this behaviour. Also, as our approximation scheme is purely algebraic, our implementation has not shown stability problems. We only collect higher ranks in the admissible blocks as we raise the frequency. The authors of [CDC17, section 8] observe a rank dependence of $O(\sqrt{n(\kappa)})$. n is a function of frequency, as one increases the mesh fineness to keep the number of mesh elements per wavelength constant.

9 $\mathcal{H}$-arithmetic

Now that we have introduced $\mathcal{H}$- matrices and discussed their applicability to the Helmholtz BEM; we concern ourselves with the arithmetic of $\mathcal{H}$-matrices, so that we can solve an $\mathcal{H}$-BEM system.

9.1 $\mathcal{H}$-matrix-vector product

Let $P \in T(I \times J)$. The following algorithm computes $\mathbf{y} \coloneqq \mathbf{y} + M\mathbf{x}$, with $M \in \mathcal{H}(k, P)$, $\mathbf{y} \in \mathbb{C}^I$ and $\mathbf{x} \in \mathbb{C}^J$. Furthermore let $b = \gamma \times \delta \in T(I \times J, P)$. The algorithm computes more generally $\mathbf{y}|_\gamma \coloneqq \mathbf{y}|_\gamma + M|_{\gamma\times\delta} \cdot \mathbf{x}|_\delta$.

Algorithm 9.1: MVM$(\mathbf{y}, M, \mathbf{x}, b)$

input: $y \in \mathbb{C}^I$ // product vector
$M \in \mathcal{H}(k, P)$ // factor matrix
$x \in \mathbb{C}^J$ // factor vector
$b = \gamma \times \delta \in T(I \times J, P)$ // block cluster
begin
    if $b \in P$ // b is in the partition $\rightarrow M|_b$ is explicitly represented
        if Adm(b) = true // b is a far-field block
            $\mathbf{y}|_\gamma = \mathbf{y}|_\gamma + AB\mathbf{x}|_\delta$ // $M|_b$ low-rank representation: $M|_b = AB$
        else // b is a near-field block
            $\mathbf{y}|_\gamma = \mathbf{y}|_\gamma + M|_b\mathbf{x}|_\delta$ // $M|_b$ has a full matrix representation
        endelse
    else // b is not in the partition
        for all $b' \in$ sons(b)
            MVM$(\mathbf{y}, M, \mathbf{x}, b')$ // recursively call MVM on all four sons of b
        endfor
    endelse
end

The matrix-vector product is the most important of the $\mathcal{H}$-arithmetic operations. One could use a matrix-free implementation of GMRES to implement an iterative solver with comparatively little effort. The computational complexity of this solver is in the order of $n \log_2(n)$ times the square of the number of iteration steps. We elaborate on this approach in section (9.8).

Interestingly one can massively parallelize the block·vector-segment multiplication over all blocks. Only the cheaper addition requires some programming effort to avoid write collisions.

9.2 vector-$\mathcal{H}$-matrix product

Now we treat the transposed case of the matrix vector product. Let $P \in T(I \times J)$ and $M \in \mathcal{H}(k, P)$. The following algorithm computes $\mathbf{y}^\intercal := \mathbf{y}^\intercal + \mathbf{x}^\intercal M$, with $\mathbf{y}^\intercal \in \mathbb{C}^J$ and $\mathbf{x}^\intercal \in \mathbb{C}^I$.

Algorithm 9.2: VMM($\mathbf{y}^\intercal, \mathbf{x}^\intercal, M, b$)

input: $\mathbf{y}^\intercal \in \mathbb{C}^J$ // product vector
$\mathbf{x}^\intercal \in \mathbb{C}^I$ // factor vector
$M \in \mathcal{H}(k, P)$ // factor matrix
$b = \gamma \times \delta \in T(I \times J)$ // block cluster
begin
    if $b \in P$ // b is in the partition $\rightarrow$ holds actual matrix information
        if Adm(b) = true // b is a far-field block
            $\mathbf{y}^\intercal|_\delta = \mathbf{y}^\intercal|_\delta + \mathbf{x}^\intercal|_\gamma AB$ // $M|_b \in \mathcal{R}(k, b)$
        else // b is a near-field block
            $\mathbf{y}^\intercal|_\delta = \mathbf{y}^\intercal|_\delta + \mathbf{x}^\intercal|_\gamma M|_b$ // $M|_b \in \mathbb{C}^b$
        endelse
    else // b is not in the partition
        for all $b' \in$ sons(b)
            VMM($\mathbf{y}^\intercal, \mathbf{x}^\intercal, M, b'$) // call recursion on all four sons of b
        endfor
    endelse
end

9.3 $\mathcal{H}$-matrix-addition

Let P, $P' \in T(I \times J)$. $M \in \mathcal{H}(k, P)$ and $M' \in \mathcal{H}(k, P')$. The following algorithm computes $M := M \oplus_k M' \in \mathcal{H}(k, P'')$, with $P'' \in T(I \times J)$. Note that the partitions of the summands can be different. Let us imagine for example that a block b is in the partition P, but an inner vertex in the block cluster tree $T(I \times J, P')$. Therefore b has a subtree in $T(I \times J, P')$. The addition algorithm then appends that subtree (with its block matrices in its leaves) to $T(I \times J, P)$. The block matrix $M|_b$ then has to be 'flushed down' the new subtree to the new leaves (the details are in algorithm 9.12). The notation 'restrict M to block b' signifies at that intermediary stage the actual block matrix, that is stored in the block cluster b in the tree. Before the superimposed matrix information is not united, $M|_b$ can not readily be evaluated in the classical sense as simple matrix. We deal with the same issue in the multiplication procedure; figure (9.1) might bring more clarity. Now the addition algorithm.

Algorithm 9.3: Add(M, M', b)

input: $M \in \mathcal{H}(k, P)$ // summand
$M' \in \mathcal{H}(k, P')$ // summand
$b = \gamma \times \delta \in T(I \times J)$ // block cluster
begin
  if $b \in P$ or $b \in P'$ // at least one partition block
    if $b \in P$ and $b \in P'$ // b is leaf in $T(I \times J, P)$ and $T(I \times J, P')$
    // $\implies$ $M|_b$ and $M'|_b$ have either full- or low-rank representation
      if Adm(b) = true
        $M|_b \coloneqq M|_b \oplus_k M'|_b$ // truncated $\mathcal{R}(k)$-addition
      else // size(b) = false
        $M|_b \coloneqq M|_b + M'|_b$ // ordinary matrix addition
      endelse
    else if $b \in P$ // b is only leaf in $T(I \times J, P)$
      $\text{sons}_{T(I\times J,P)}(b) \coloneqq \text{sons}_{T(I\times J,P')}(b)$ // append the sons of $M'|_b$
                                                  // to $M|_b$
      //move the information of $M|_b$ down to its new leaf nodes
      flushToSons(M, P'', b)
    else // b is only leaf in $T(I \times J, P')$
      $M|_b \coloneqq M'|_b$ // copy the matrix information of $M'|_b$ over to $M|_b$
      //move the information of $M|_b$ down to its old leaf nodes
      flushToSons(M, P'', b)
    endelse
  else // b is not a partition block
    for all $b' \in$ sons(b) //
      Add(M, M', b') // recursively call Add on all four sons of b
    endfor
  endelse
end

We will also use the symbol $\oplus_k$ for the addition of $\mathcal{H}(k)$-matrices for the remainder of this text.

9.4 $\mathcal{H}$-block-$\mathcal{R}$-block multiplication

When we will introduce the $\mathcal{H}$-matrix multiplication later, we will need the $\mathcal{H}$-matrix-$\mathcal{R}$-matrix product, or more generally the $\mathcal{H}$-block-$\mathcal{R}$-block product.

Let $M' \in \mathcal{H}(k, I, J)$ and $M'' \in \mathcal{H}(k, J, K)$ be $\mathcal{H}$-Matrices. Let also $\gamma \in T_I$, $\delta \in T_J$ and $\epsilon \in T_K$ be clusters, whereby the block $\gamma \times \delta$ is in the cluster tree $T(I \times J, P_{M'})$ and the block $\delta \times \epsilon$ is in the cluster tree $T(J \times K, P_{M''})$. We will also presume that the $\delta \times \epsilon$-block of M'' has a low-rank representation $(M''|_{\delta\times\epsilon} \in \mathcal{R}(\delta \times \epsilon))$. Let M be the product of M' and M''. The following algorithm computes $M|_{\gamma\times\epsilon} = M'|_{\gamma\times\delta} \cdot M''|_{\delta\times\epsilon}$.

Algorithm 9.4: HBlockRBlockProduct$(M, M', M'', \gamma, \delta, \epsilon)$

input: $M \in \mathcal{H}(k, I, K)$ // product matrix, with $M|_{\gamma\times\epsilon} \in \mathcal{R}(k, \gamma \times \epsilon)$
$M' \in \mathcal{H}(k, I, J)$ // factor matrix
$M'' \in \mathcal{H}(k, J, K)$ // factor matrix, with $M|_{\delta\times\epsilon} = A''B'' \in \mathcal{R}(k, \delta \times \epsilon)$
$\gamma \times \epsilon \in T(I \times K)$ // product block
$\gamma \times \delta \in T(I \times J)$ // first factor block
$\delta \times \epsilon \in T(J \times K)$ // second factor block
begin
    $A^{\text{product}} \in 0^{\gamma\times k}$ // initialize product block A-matrix as zero matrix
    // $A''B''$ is the low-rank form of the second factor block $M''|_{\delta\times\epsilon}$
    for $i \in k$ //iterate over all columns of the second factor
        // use the matrix vector product column-wise on the second
        // factors A'' matrix
        $\text{MVM}(A^{\text{product}}_{\gamma\times i}, M', A''_{\delta\times i}, \gamma \times \delta)$
    endfor
    $M|_{\gamma\times\epsilon} \coloneqq A^{\text{product}}B''$ // set the low-rank form of the product block
end

[Hac09, Section 7.4.2.3]
Now we treat the case of the first factor block being a low-rank block and the second block being an $\mathcal{H}$-block.

Algorithm 9.5: RBlockHBlockProduct$(M, M', M'', \gamma, \delta, \epsilon)$

input: $M \in \mathcal{H}(k, I, K)$ // product matrix, with $M|_{\gamma\times\epsilon} \in \mathcal{R}(k, \gamma \times \epsilon)$
$M' \in \mathcal{H}(k, I, J)$ // factor matrix, with $M'|_{\gamma\times\delta} = A'B' \in \mathcal{R}(k, \gamma \times \delta)$
$M'' \in \mathcal{H}(k, J, K)$ // factor matrix
$\gamma \times \epsilon \in T(I \times K)$ // product block
$\gamma \times \delta \in T(I \times J)$ // first factor block
$\delta \times \epsilon \in T(J \times K)$ // second factor block
begin
    $B^{\text{product}} \in 0^{k\times\delta}$ // initialize product block B-matrix as zero matrix
    // $A'B'$ is the low-rank form of $M'|_{\gamma\times\delta}$
    for $i \in k$ //iterate over all rows of the first factor
        // use the vector-matrix product row-wise on the first factor
        $\text{VMM}(B^{\text{product}}_{i\times\delta}, B'_{i\times\delta}, M'', \delta \times \epsilon)$
    endfor
    $M|_{\gamma\times\epsilon} \coloneqq A'B^{\text{product}}$ // set the low-rank form of the product block
end

9.5 $\mathcal{H}$-block-full-block multiplication

We focus our attention on the product of an $\mathcal{H}$-matrix block and a matrix block with full-rank representation. This case is related to HBlockRBlockProduct (Algorithm 9.4).We recreate similar conditions to (9.4):

Let $M' \in \mathcal{H}(k, I, J)$, $M'' \in \mathcal{H}(k, J, K)$, $\gamma \in T_I$, $\delta \in T_J$ and $\epsilon \in T_K$, whereby the block $\gamma \times \delta$ is in the cluster tree $T(I \times J, P_{M'})$ and the block $\delta \times \epsilon$ is in the cluster tree $T(J \times K, P_{M''})$. Furthermore the $\delta \times \epsilon$-block of M'' has a full-rank representation $(M''|_{\delta\times\epsilon} \in \mathbb{C}^{\delta\times\epsilon})$. Let again M be the product of M' and M''. The following algorithm computes $M|_{\gamma\times\epsilon} = M'|_{\gamma\times\delta} \cdot M''|_{\delta\times\epsilon}$. We have to deal with the complication, that in some cases the product block can be large and therefore unsuitable for a full-rank representation. This case arises, when we multiply a wide matrix with a tall matrix (size(γ) = true, size(δ) = false and size(ϵ) = true). We attend to this inefficiency by converting a factor to a low-rank representation.

Algorithm 9.6: HBlockFullBlockProduct($M, M', M'', \gamma, \delta, \epsilon$)

input: $M \in \mathcal{H}(k, I, K)$ // product matrix, with $M|_{\gamma\times\epsilon} \in \mathbb{C}^{\gamma\times\epsilon}$ or $M|_{\gamma\times\epsilon} \in R(k, \gamma \times \epsilon)$
$M' \in \mathcal{H}(k, I, J)$ // factor matrix
$M'' \in \mathcal{H}(k, J, K)$ // factor matrix, with $M''|_{\delta\times\epsilon} \in \mathbb{C}^{\delta\times\epsilon}$
$\gamma \times \epsilon \in T(I \times K)$ // product block
$\gamma \times \delta \in T(I \times J)$ // first factor block
$\delta \times \epsilon \in T(J \times K)$ // second factor block
begin
  if size($\gamma \times \epsilon$) // product block should have $\mathcal{R}(k)$-representation
    $A^{\text{product}} \in 0^{\gamma\times\delta}$ // initialize product block A-matrix as zero matrix
    $A'' :=$ Identity($\delta \times \delta$) //zero-padded identity matrix
    $B'' := M''|_{\delta\times\epsilon}$ // B'' is the full matrix
    // $A''B''$ is a low-rank form of $M''|_{\delta\times\epsilon}$
    for $i \in \delta$
    // A^{product} is $M'|_{\gamma\times\delta}$ converted to full matrix
      MVM($A^{\text{product}}_{\gamma\times i}, M', A''_{\delta\times i}, \gamma \times \delta$)
    endfor
    $M|_{\gamma\times\epsilon} := A^{\text{product}}B''$ // set the low-rank form of the product block
  else // product block has full-rank representation
    $M_{\gamma\times\epsilon} \in 0^{\gamma\times\epsilon}$ // initialize product block as zero matrix
    for $i \in \epsilon$ //iterate over all columns of the second factor
      // use the matrix vector product column-wise
      // on the second factor
      MVM($M_{\gamma\times i}, M', M''_{\gamma\times i}, \gamma \times \delta$)
    endfor
  endelse
end

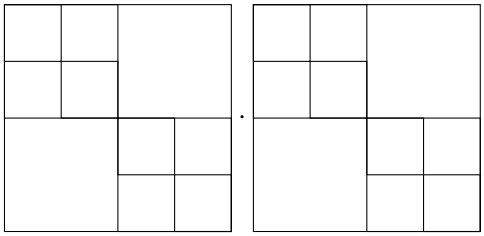
[Hac09, Section 7.4.2.4]

The transpose case fullBlockHBlockProduct is defined analogously.

9.6 $\mathcal{H}$-matrix-matrix multiplication

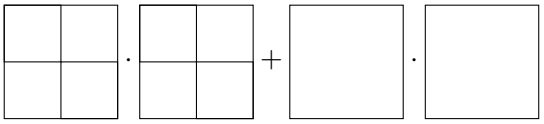
Now we study the $\mathcal{H}(k)$-matrix-matrix multiplication. We demonstrate the main difficulty of the algorithm with a simple example. Consider the following block matrix product:

$$M = \begin{pmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{pmatrix} =$$

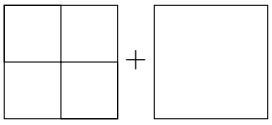


If we look at the block M_{11}, we see that the decompositions of the sub-products do not fit together.

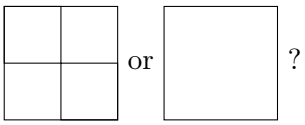
$$M_{11} =$$



$$=$$



$$=$$



or

?

A corresponding product $\mathcal{H}$-matrix would have overlapping sub products in its cluster tree. This would need to be amended by either merging the lower blocks (sub-products) together; upwards the block cluster tree. Or one could flush the 'higher' (in the tree) sub-products downwards. It is generally not obvious, how to determine the product partition.

The intermediate product

The following multiplication procedure calculates $M := M' \odot_k M''$, with the factors $M' \in \mathcal{H}(k, P')$, $M'' \in \mathcal{H}(k, P'')$ and their respective block cluster trees $T'(I \times J)$, $T''(J \times K)$. The product $M \in \mathcal{H}(k, P)$ has also a block cluster tree, $T^{prod}(I \times K)$. At the beginning of the multiplication, the product block cluster tree contains just the root node $I \times K$. The product tree is build up during the multiplication.

Algorithm 9.7: MM$(M, M', M'', \gamma, \delta, \epsilon)$

input: $M \in \mathcal{H}(k)$ // product
$M' \in \mathcal{H}(k, P')$ // factor
$M'' \in \mathcal{H}(k, P'')$ // factor
$\gamma \times \epsilon \in T^{prod}(I \times K)$ // product block cluster
$\gamma \times \delta \in T'(I \times J, P')$ // factor block cluster
$\delta \times \epsilon \in T''(J \times K, P'')$ // factor block cluster
var: $T^{prod}(I \times K)$ // MM builds up the product block cluster tree
begin
    if $\gamma \times \delta \in P'$ or $\delta \times \epsilon \in P''$ // at least one factor block is a partition block
        if ($\gamma \times \delta \in P'$ and $\mathrm{Adm}(\gamma \times \delta) = \mathrm{true}$) // first factor block is
                                                                // $\mathcal{R}(k)$-matrix
            $M|_{\gamma\times\epsilon} += \mathrm{RBlockHBlockProduct}(M, M', M'', \gamma, \delta, \epsilon)$
        else if ($\delta \times \epsilon \in P''$ and $\mathrm{Adm}(\delta \times \epsilon) = \mathrm{true}$) // $M''|_{\delta\times\epsilon} \in \mathcal{R}(k, \delta \times \epsilon)$
            $M|_{\gamma\times\epsilon} += \mathrm{HBlockRBlockProduct}(M, M', M'', \gamma, \delta, \epsilon)$
        else if $\gamma \times \delta \in P'$ // first factor block is a full matrix
            $M|_{\gamma\times\epsilon} += \mathrm{FullBlockHBlockProduct}(M, M', M'', \gamma, \delta, \epsilon)$
        else // second factor block is a full matrix
            $M|_{\gamma\times\epsilon} += \mathrm{HBlockFullBlockProduct}(M, M', M'', \gamma, \delta, \epsilon)$
        endelse
    else // no factor block is a partition block
        // we write for the cluster sons:
        $//\{\gamma_1, \gamma_2\} := \mathrm{sons}_{T_I}(\gamma)$
        $//\{\delta_1, \delta_2\} := \mathrm{sons}_{T_J}(\delta)$
        $//\{\epsilon_1, \epsilon_2\} := \mathrm{sons}_{T_K}(\epsilon)$

        // partition the product block; meaning, to append four sons to
        // $\gamma \times \epsilon$ in the block cluster tree $T(I \times K)$
        $\mathrm{sons}_{T^{prod}_{I\times I}}(\gamma \times \epsilon) := \{\gamma_1 \times \epsilon_1, \gamma_1 \times \epsilon_2, \gamma_2 \times \epsilon_1, \gamma_2 \times \epsilon_2\}$
        // recursively call the multiplication routine analogously to the ordinary
        // block-wise multiplication
        $\mathrm{MM}(M, M', M'', \gamma_1, \delta_1, \epsilon_1)$ // $M_{11} = M'_{11} \odot_k M''_{11}$
        $\mathrm{MM}(M, M', M'', \gamma_1, \delta_2, \epsilon_1)$ // $M_{11} = M'_{12} \odot_k M''_{21}$

        $\mathrm{MM}(M, M', M'', \gamma_1, \delta_1, \epsilon_2)$ // $M_{12} = M'_{11} \odot_k M''_{12}$
        $\mathrm{MM}(M, M', M'', \gamma_1, \delta_2, \epsilon_2)$ // $M_{12} = M'_{12} \odot_k M''_{22}$

        $\mathrm{MM}(M, M', M'', \gamma_2, \delta_1, \epsilon_1)$ // $M_{21} = M'_{21} \odot_k M''_{11}$
        $\mathrm{MM}(M, M', M'', \gamma_2, \delta_2, \epsilon_1)$ // $M_{21} = M'_{22} \odot_k M''_{21}$

        $\mathrm{MM}(M, M', M'', \gamma_2, \delta_1, \epsilon_2)$ // $M_{22} = M'_{21} \odot_k M''_{12}$
        $\mathrm{MM}(M, M', M'', \gamma_2, \delta_2, \epsilon_2)$ // $M_{22} = M'_{22} \odot_k M''_{22}$
    endelse
end

[Hac09, Section 7.4.3.3]

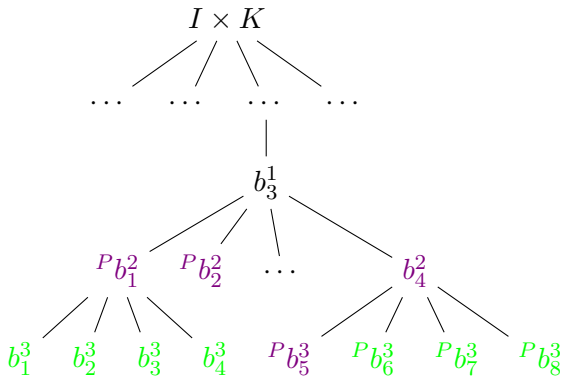


Figure 9.1: $T^{prod}(I \times K)$ after MM recursion

We use figure (9.1) to discuss several cases that can occur, after the MM recursion has run through. The green vertices represent blocks that accumulate full matrices. The violet vertices represent large blocks that receive low rank sub-products. The ^{P} prescript signifies that the block is in the product partition. Let us consider the block b_4^2. It stores a sum of low rank matrices and it is not in the product partition. The relevant partition blocks are further downward the tree. Therefore the information that is stored in b_4^2 has to be flushed downwards the tree and converted into the format of the respective information receiving blocks. This flushing and converting is done by the flushToSons algorithm (9.12).

Let us now consider the block ${}^P b_1^2$. The block is a partition block; by itself there is nothing to do. The block has sons. The product block cluster tree is build in a way, that guarantees that, if a block has sons, there is information below. This information has to be agglomerated upwards the tree (by the mergeSons algorithm 9.11).

Another important observation is, that it is memory-wise inefficient, to have superimposed information in the product tree. A very beneficial ansatz is to make the MM algorithm only organizational. It constructs the product block cluster tree. Instead of actually performing the sub-products, they are stored in lists in the product blocks. No actual multiplications are performed. In a later stage we can then traverse the product tree downwards, until we encounter blocks that hold information. If we encounter a block with pending sub-products, we then compute the list. If the block is not a partition block, we flush the computed information to its sons. Then we do the same (easily parallelizable) for all four sons of the block. Once we encounter partition blocks, we change to the mergeSons procedure. This style of multiplication algorithm is a variant of the algorithm described in [Bö19, page 13].

The obvious benefits of this approach are; no redundant memory usage and easy massive parallelization. Another organizational advantage is, that we can determine the product partition on the by MM constructed block cluster tree, before we do the actual sub-product computation. So far we had just assumed in the above text that the partition was already determined.

The product partition

The following algorithm returns the leaves of $T^{\text{product}}(I \times I) \cap T(I \times I, P') \cap T(I \times I, P'')$ as the product partition for square $\mathcal{H}$-matrices with $T_{I \times I}$ as block cluster tree. The algorithm attempts to be as close to idempotent as possible/sensible in this case. Choosing the leaves of the product block cluster tree as a product partition generally leads to an increasingly fine product partition [Hac09, page 184, section 7.8.3.4]. This can be costly if the product is used for further multiplications/operations. Agglomerating the product cluster tree to a high degree can be immediately costly in contrast. We chose to aim a priori for idempotency as a compromise. This compromise performed acceptably in our numerical tests. We did not find explicit information on optimal strategies for choosing product partitions for general products. We use the multiplication as a subroutine during the $\mathcal{H}$-LU factorization however. The resulting product is then subtracted from another $\mathcal{H}$-Matrix M. The resulting product partition should therefore be informed by the partition of M.

In the product partition/tree the admissibility is defined purely by the block cluster size. Let $b = \delta \times \epsilon$ be a block cluster from the product block cluster tree $T(I \times K)$. Then the admissibility for b to be low-rank represented is defined as follows:

$$\text{Adm}_{T(I\times K)}(b) := \text{size}_{T(I\times K)}(\gamma \times \epsilon) = (\text{size}_{T_I}(\gamma) \text{ and } \text{size}_{T_J}(\epsilon))$$

Now the algorithm:

Algorithm 9.8: productPartition(b, b', b'')

input: $b \in T^{prod}(I \times K)$ // product block cluster
$b' \in T(I \times J)$ // factor block cluster
$b'' \in T(J \times K)$ // factor block cluster
var: P // the product partition
begin
    if isLeaf$_{T^{prod}(I\times K)}(b)$ = true or $b' \in P'$ or $b'' \in P''$
        $P := P \cup \{b\}$ // add b to the product partition
        // determine whether $b \in P^+$ or $b \in P^-$
        $\text{Adm}_{T(I\times K)}(b) := \text{size}_{T(I\times K)}(b)$
    else // b is not a partition block
        productPartition$(b_{11}, b'_{11}, b''_{11})$
        productPartition$(b_{12}, b'_{12}, b''_{12})$
        productPartition$(b_{21}, b'_{21}, b''_{21})$
        productPartition$(b_{22}, b'_{22}, b''_{22})$
    endelse
end

We now describe a central sub-routine of the mergeSons algorithm. It merges a block row.

Algorithm 9.9: rowMerge(M', M'')

input: $M' = A'B' \in \mathcal{R}(k, \gamma \times \delta_1)$
$M'' = A''B'' \in \mathcal{R}(k, \gamma \times \delta_2)$ //low-rank matrices, whereby $\{\delta_1, \delta_2\} = \text{sons}(\delta)$
output: $M = AB \in \mathcal{R}(k, \gamma \times \delta)$ // returns the truncated agglomeration $M = (M', M'')$
begin

$$M'_{\gamma\times\delta} \coloneqq A' \begin{pmatrix} |\delta_1| & |\delta_2| \\ B' & 0 \end{pmatrix}$$

$$M''_{\gamma\times\delta} \coloneqq A'' \begin{pmatrix} |\delta_1| & |\delta_2| \\ 0 & B'' \end{pmatrix}$$

//return the best rank k approximation of $\begin{pmatrix} A' & A'' \end{pmatrix} \begin{pmatrix} B' & 0 \\ 0 & B'' \end{pmatrix}$

return $M'_{\gamma\times\delta} \oplus_k M''_{\gamma\times\delta}$ // truncated addition via compressed SVD

end

Now the related procedure for block columns.

Algorithm 9.10: columnMerge(M', M'')

input: $M' = A'B' \in \mathcal{R}(k, \gamma_1 \times \delta)$
$M'' = A''B'' \in \mathcal{R}(k, \gamma_2 \times \delta)$ //low-rank matrices, whereby $\{\gamma_1, \gamma_2\} = \text{sons}(\gamma)$
output: $M = AB \in \mathcal{R}(k, \gamma \times \delta)$ // returns the truncated agglomeration $M = \begin{pmatrix} M' \\ M'' \end{pmatrix}$
begin

$$M'_{\gamma\times\delta} \coloneqq \begin{matrix} |\gamma_1| \\ |\gamma_2| \end{matrix} \begin{pmatrix} A' \\ 0 \end{pmatrix} B'$$

$$M''_{\gamma\times\delta} \coloneqq \begin{matrix} |\gamma_1| \\ |\gamma_2| \end{matrix} \begin{pmatrix} 0 \\ A'' \end{pmatrix} B''$$

//return the best rank k approximation of $\begin{pmatrix} A' & 0 \\ 0 & A'' \end{pmatrix} \begin{pmatrix} B' & B'' \end{pmatrix}$

return $M'_{\gamma\times\delta} \oplus_k M''_{\gamma\times\delta}$ // truncated addition via compressed SVD

end

Now we give the mergeSons procedure.

Algorithm 9.11: mergeSons(M, b)

input: $M \in \mathcal{H}(k, T(I \times K))$
$b = \gamma \times \epsilon \in T(I \times K)$ // block cluster
begin
    do the work on the accumulated sub−products as described **in**
    if isLeaf$_{T(I\times K)}(b) =$ true
        if (size$(b) =$ false)
        //true, if block has a full matrix representation
            convert $M|_b$ **to** $\mathcal{R}$−matrix
        endif

    else // b is not a leaf vertex
        $\{\gamma_1, \gamma_2\} \coloneqq \text{sons}_{T_I}(\gamma)$
        $\{\epsilon_1, \epsilon_2\} \coloneqq \text{sons}_{T_K}(\epsilon)$
        declare matrices: $M_{\gamma_1\times\epsilon}, M_{\gamma_2\times\epsilon}$
        // iterate over all sons of b
        for all $\gamma_i \in \{\gamma_1, \gamma_2\}$
            for all $\epsilon_j \in \{\epsilon_1, \epsilon_2\}$
                // call mergeSons recursively on all sons
                mergeSons$(M, \gamma_i \times \epsilon_j)$
            endfor
            $M_{\gamma_i\times\epsilon} \coloneqq \text{rowMerge}(M|_{\gamma_i\times\epsilon_1}, M|_{\gamma_i\times\epsilon_2})$
        endfor
        // add the agglomerated sons to the block matrix
        $M_b \coloneqq M_b \oplus_k \text{columnMerge}(M_{\gamma_1\times\epsilon}, M_{\gamma_2\times\epsilon})$
        // b is not a leaf vertex in $T(I \times K) \implies M_b \in \mathcal{R}(k)$
    endelse
end

The following recursive algorithm traverses the product block cluster tree downwards. In each block the algorithm works on the accumulated sub-products. Afterwards the routine flushes the result downwards to the blocks sons, until it respectively reaches a partition vertex. If the reached partition vertex is not a leaf node in the block cluster tree, a merge routine is called, that agglomerates the sub-tree of the partition node into the partition node.

Algorithm 9.12: flushToSons(M, P, b)

input: $M \in \mathcal{H}(k, T(I \times K))$ // intermediate product matrix
$P \subseteq T(I \times K)$ // the product partition
$b = \gamma \times \epsilon \in T(I \times K)$ // block cluster
begin
    do the work on the accumulated sub−products
    if there is no work **to** be done/ block contains no information
     −> go directly **to** the recursion call
    if $b \in P$ // b is a partition block
        mergeSons(M, b) // agglomerate sub-tree of b
    else // b is not a partition block
        $\{\gamma_1, \gamma_2\} \coloneqq \text{sons}_{T_I}(\gamma)$
        $\{\epsilon_1, \epsilon_2\} \coloneqq \text{sons}_{T_K}(\epsilon)$
        // $M|_b = \begin{pmatrix} A_{\gamma_1} \\ A_{\gamma_2} \end{pmatrix} \begin{pmatrix} B_{\epsilon_1} & B_{\epsilon_2} \end{pmatrix}$; block low-rank form of $M|_b$
        // iterate over all sons of b
        for all $\gamma_i \in \{\gamma_1, \gamma_2\}$
            for all $\epsilon_j \in \{\epsilon_1, \epsilon_2\}$
                if(size($\gamma_i \times \epsilon_i$) = $true$) // son has low-rank form
                    // truncated addition via compressed SVD
                    $M|_{\gamma_i \times \epsilon_j} \coloneqq M|_{\gamma_i \times \epsilon_j} \oplus_k A_{\gamma_i} B_{\epsilon_j}$
                else // son has full-rank form
                    $M|_{\gamma_i \times \epsilon_j} \coloneqq M|_{\gamma_i \times \epsilon_j} + A_{\gamma_i} B_{\epsilon_j}$
                endelse
            endfor
        endfor
        // call recursion on all sons
        for all $\gamma_i \in \{\gamma_1, \gamma_2\}$
            for all $\epsilon_j \in \{\epsilon_1, \epsilon_2\}$
                flushToSons($M, P, \gamma_i \times \epsilon_j$)
            endfor
        endfor
    endelse
end

The following algorithm contains the entire $\mathcal{H}$-matrix multiplication.

Algorithm 9.13: HMultiply(M', M'')

input: $M' \in \mathcal{H}(k, P')$ with block cluster tree $T(I \times J)$ // factor
$M'' \in \mathcal{H}(k, P'')$ with block cluster tree $T(J \times K)$ // factor
// return the product matrix
output: $M \in \mathcal{H}(k, P)$ with block cluster tree $T^{prod}(I \times K)$
begin
$\gamma \coloneqq \text{root}(T_I)$
$\epsilon \coloneqq \text{root}(T_K)$
$\text{root}(T^{prod}(I \times K)) \coloneqq \gamma \times \epsilon$ // set the root block of the product
// block cluster tree
$M \coloneqq \text{root}(T(I \times K))$ // initialize product matrix
// build the product block cluster tree and
$\text{MM}(M, M', M'', I, J, K)$
$\text{productPartition}(I \times K, I \times J, J \times K)$
$\text{flushToSons}(M, P, I \times K)$
return M
end

9.7 $\mathcal{H}$-LU factorization

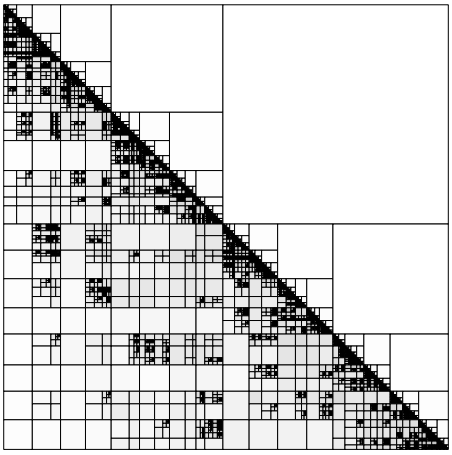


Figure 9.2: Lower triangular $\mathcal{H}$-matrix

Finally we treat the $\mathcal{H}$-LU factorization for square $\mathcal{H}$-matrices. Due to he way that our reindexing procedure (algorithm 6.8) indexed the clusters, for two distinct clusters γ and $\delta \in T_I$ at the same level in the tree, the indices of one cluster are either all larger or smaller than the indices of the other cluster. This defines an order on the clusters of a tree level. Formally we can write $\gamma < \delta$ or $\gamma > \delta$ for $\gamma, \delta \in T_I^{(l)}, \gamma \neq \delta$.

Let $b = \gamma \times \delta \in P$ be a block from the block cluster tree $T_{I\times I}(P)$. By construction, γ and δ are from the same tree level. We say L and U are lower and upper triangular matrices in the $\mathcal{H}$-format if they satisfy

$$L, U \in \mathcal{H}(k, P), \quad \text{with} \begin{cases} L_{\gamma\times\delta} = 0^{\gamma\times\delta} & \text{for } \gamma < \delta \\ L_{\gamma\times\gamma} \text{ is normalized lower triangular} & \\ U_{\gamma\times\gamma} \text{ is upper triangular} & \\ U_{\gamma\times\delta} = 0^{\gamma\times\delta} & \text{for } \gamma > \delta \end{cases} \tag{9.1}$$

[Hac09, 7.36]. The fundamental structure of the $\mathcal{H}$-LU factorization is the same as in algorithm (5.1).

Algorithm 9.14: LU(M, L, U, b)

input: $M \in \mathcal{H}(k, P)$ with block cluster tree $T(I \times I)$ and Partition P
$L, U \in \mathcal{H}(k)$
$b = \gamma \times \gamma \in T(I \times I)$ // b is a diagonal block
begin
  if $b \in P$
    // conventional LU factorization
    calculate $L|_b$,$U|_b$, so that $L|_b U|_b = M|_b$
  else
    $\{\gamma_1, \gamma_2\} \coloneqq \text{sons}_{T_I}(\gamma)$

    // calculate $L|_{\gamma_1\times\gamma_1}$ and $U|_{\gamma_1\times\gamma_1}$
    LU$(M, L, U, \gamma_1 \times \gamma_1)$
    // solve $L|_{\gamma_2\times\gamma_1} U|_{\gamma_1\times\gamma_1} = M|_{\gamma_2\times\gamma_1}$ for $L|_{\gamma_2\times\gamma_1}$
    matValTranHForwardSubstitution$(L, U, M, \gamma_2, \gamma_1)$
    // solve $L|_{\gamma_1\times\gamma_1} U|_{\gamma_1\times\gamma_2} = M|_{\gamma_1\times\gamma_2}$ for $U|_{\gamma_1\times\gamma_2}$
    matValHForwardSubstitution$(L, U, M, \gamma_1, \gamma_2)$
    // $M|_{\gamma_2\times\gamma_2} \coloneqq M|_{\gamma_2\times\gamma_2} - L|_{\gamma_2\times\gamma_1} U|_{\gamma_1\times\gamma_2}$
    $M|_{\gamma_2\times\gamma_2} \coloneqq M|_{\gamma_2\times\gamma_2} \ominus_k$ HMultiply$(L|_{\gamma_2\times\gamma_1}, U|_{\gamma_1\times\gamma_2})$
    // calculate $L|_{\gamma_2\times\gamma_2}$ and $U|_{\gamma_2\times\gamma_2}$
    LU$(M, L, U, \gamma_2 \times \gamma_2)$
  endelse
end

[Hac09, 7.40] We give the ingredients for above algorithm in the following.

Linear systems in $\mathcal{H}$-matrix form are solved as usual with forward and backward substitution.

Algorithm 9.15: luSolve($M, \mathbf{b}$)

input: $M \in \mathcal{H}(k, P)$ with block cluster tree $T(I \times I)$ and Partition P
$\mathbf{b} \in \mathbb{C}^I$ // right-hand side of the linear system
output: $\mathbf{x} \in \mathbb{C}^I$ // solution to $M\mathbf{x} = \mathbf{b}$
begin
    $L \coloneqq \text{root}(M)$
    $U \coloneqq \text{root}(M)$

    $\text{LU}(M, L, U, I \times I)$ // factorize M in lower- and upper-triangular block matrix

    $\mathbf{x} \coloneqq \mathbb{C}^I$ // declare output vector
    $\mathbf{y} \coloneqq \mathbb{C}^I$ // declare temporary vector

    $\text{hForwardSubstitute}(L, \mathbf{y}, \mathbf{b}, I)$ // forward substitution
    $\text{hBackwardSubstitute}(U, \mathbf{x}, \mathbf{y}, I)$ // backward substitution

    return x
end

$\mathcal{H}$-matrix substitutions

This subsection has the relevant substitutions to compute the $\mathcal{H}$-LU-factorization (algoritm 9.14). They are analogons to the procedures in section 5.4, just reformulated to account for the $\mathcal{H}$-tree structure.

Algorithm 9.16: hForwardSubstitute$(L, \mathbf{y}, \mathbf{b}, \gamma)$

input: $L \in \mathcal{H}(k)$ // lower triangular with block cluster tree $T(I \times I, P)$
$\mathbf{y} \in \mathbb{C}^I$ // $\mathbf{y}$ solves $L\mathbf{y} = \mathbf{b}$
$\mathbf{b} \in \mathbb{C}^I$ // right-hand side vector
$\gamma \in T_I$ // cluster
begin
    if $\gamma \times \gamma \in P$
        // ordinary forward substitution
        for $j = \gamma(1)$ **to** $\gamma(|\gamma|)$ // iterate over the ordered index set γ;
                                                // from smallest to largest
            $\mathbf{y}_j \coloneqq \mathbf{b}_j$
            for $i = j + 1$ **to** $\gamma(|\gamma|)$
                $\mathbf{b}_i \coloneqq \mathbf{b}_i - L_{ij}\mathbf{y}_j$
            endfor
        endfor
    else
    $\{\gamma_1, \gamma_2\} \coloneqq \text{sons}_{T_I}(\gamma)$
    hForwardSubstitute$(L, \mathbf{y}, \mathbf{b}, \gamma_1)$
    MVM$(\mathbf{b}, L, -\mathbf{y}, \gamma_2 \times \gamma_1)$ // $\mathbf{b}|_{\gamma_2} \coloneqq \mathbf{b}|_{\gamma_2} - L|_{\gamma_2 \times \gamma_1}\mathbf{y}|_{\gamma_1}$
    hForwardSubstitute$(L, \mathbf{y}, \mathbf{b}, \gamma_2)$
    endelse
end

[Hac09, 7.6.2]

Algorithm 9.17: hBackwardSubstitute$(U, \mathbf{x}, \mathbf{y}, \gamma)$

input: $U \in \mathcal{H}(k)$ // upper triangular with block cluster tree $T(I \times I, P)$
$\mathbf{x} \in \mathbb{C}^I$ // $\mathbf{x}$ solves $U\mathbf{x} = \mathbf{y}$
$\mathbf{y} \in \mathbb{C}^I$ // right-hand side vector
$\gamma \in T_I$ // cluster
begin
    if $\gamma \times \gamma \in P$
        // ordinary backward substitution
        for $j = \gamma(|\gamma|)$ **downto** $\gamma(1)$
            $\mathbf{x}_j \coloneqq \mathbf{y}_j / U_{jj}$
            for $i = \gamma(1)$ **to** $j - 1$
                $\mathbf{y}_i \coloneqq \mathbf{y}_i - U_{ij}\mathbf{x}_j$
            endfor
        endfor
    else
    $\{\gamma_1, \gamma_2\} \coloneqq \mathrm{sons}_{T_I}(\gamma)$
    hBackwardSubstitute$(U, \mathbf{x}, \mathbf{y}, \gamma_2)$
    MVM$(\mathbf{y}, U, -\mathbf{x}, \gamma_1 \times \gamma_2)$ // $\mathbf{y}|_{\gamma_1} \coloneqq \mathbf{y}|_{\gamma_1} - U|_{\gamma_1 \times \gamma_2}\mathbf{x}|_{\gamma_2}$
    hBackwardSubstitute$(U, \mathbf{x}, \mathbf{y}, \gamma_1)$
    endelse
end

[Hac09, 7.6.2]

Algorithm 9.18: matValHForwardSubstitution$(L, Y, Z, \gamma, \delta)$

input: $L \in \mathcal{H}(k)$ // lower triangular
$Y \in \mathcal{H}(k)$
$Z \in \mathcal{H}(k)$
$\gamma \times \delta \in T(I \times I)$
begin
    if $\gamma \times \delta \in P^+$
        for all $j \in \delta$
            hForwardSubstitute$(L, Y_{\gamma,j}, Z_{\gamma,j}, \gamma)$
        endfor
    else if $\gamma \times \delta \in P^-$
        // $Z_{\gamma\times\delta} = A^Z B^Z$
        for all $j \coloneqq 1$ **to** k
            hForwardSubstitute$(L, A^Y_{\gamma,j}, A^Z_{\gamma,j}, \gamma)$
        endfor
        $Y_{\gamma\times\delta} \coloneqq A^Y_{\gamma\times k} B^Z$
    else // $\gamma \times \delta$ is no partition block
        $\{\gamma_1, \gamma_2\} \coloneqq \text{sons}_{T_I}(\gamma)$
        $\{\delta_1, \delta_2\} \coloneqq \text{sons}_{T_I}(\delta)$

        // transposed forward substitution on the left
        // block column of $Y|_{\gamma\times\delta}$ and $Z|_{\gamma\times\delta}$
        matValHForwardSubstitution$(L, Y, Z, \gamma_1, \delta_1)$
        $Z_{\gamma_2,\delta_1} \coloneqq Z_{\gamma_2,\delta_1} \ominus_k \text{HMultiply}(L_{\gamma_2,\delta_1}, Y_{\gamma_1,\delta_1})$
        matValHForwardSubstitution$(L, Y, Z, \gamma_2, \delta_1)$

        // transposed forward substitution on the right
        // block column of $Y|_{\gamma\times\delta}$ and $Z|_{\gamma\times\delta}$
        matValHForwardSubstitution$(L, Y, Z, \gamma_1, \delta_2)$
        $Z_{\gamma_2,\delta_2} \coloneqq Z_{\gamma_2,\delta_2} \ominus_k \text{HMultiply}(L_{\gamma_2,\delta_2}, Y_{\gamma_1,\delta_2})$
        matValHForwardSubstitution$(L, Y, Z, \gamma_2, \delta_2)$
    endelse
end

[Hac09, 7.6.3]

Algorithm 9.19: hTransposedForwardSubstitute($\mathbf{x}^\intercal, U, \mathbf{y}^\intercal, \gamma$)

input: $\mathbf{x}^\intercal \in \mathbb{C}^I$ // $\mathbf{x}^\intercal$ solves $\mathbf{x}^\intercal U = \mathbf{y}^\intercal$
$U \in \mathcal{H}(k)$ // upper triangular with block cluster tree $T(I \times I, P)$
$\mathbf{y}^\intercal \in \mathbb{C}^I$ // right-hand side vector
$\gamma \in T_I$ // cluster
begin
    if $\gamma \times \gamma \in P$ // block cluster is in the partition
        for $i = \gamma(1)$ **to** $\gamma(|\gamma|)$
            $\mathbf{x}_i^\intercal \coloneqq \mathbf{y}_i^\intercal / U_{ii}$
            for $j = i + 1$ **to** $\gamma(|\gamma|)$
                $\mathbf{y}_j^\intercal \coloneqq \mathbf{y}_j^\intercal - \mathbf{x}_i^\intercal U_{ij}$
            endfor
        endfor
    else // block cluster is inner vertex in $T(I \times I, P)$
    $\{\gamma_1, \gamma_2\} \coloneqq \operatorname{sons}_{T_I}(\gamma)$
    hTransposedForwardSubstitute($\mathbf{x}^\intercal, U, \mathbf{y}^\intercal, \gamma_1$)
    VMM($\mathbf{y}^\intercal, -\mathbf{x}^\intercal, U, \gamma_1 \times \gamma_2$) // $\mathbf{y}^\intercal|_{\gamma_2} \coloneqq \mathbf{y}^\intercal|_{\gamma_2} - \mathbf{x}^\intercal|_{\gamma_1} U|_{\gamma_1 \times \gamma_2}$
    hTransposedForwardSubstitute($\mathbf{x}^\intercal, U, \mathbf{y}^\intercal, \gamma_2$)
    endelse
end

Algorithm 9.20: matValTranHForwardSubstitution$(X, U, Y, \gamma, \delta)$

input: $X \in \mathcal{H}(k)$ //X solves $XU = Y$
$U \in \mathcal{H}(k)$ // upper triangular
$Y \in \mathcal{H}(k)$
$\gamma \times \delta \in T(I \times I)$
begin
  if $\gamma \times \delta \in P^+$
    for all $i \in \gamma$
      hTransposedForwardSubstitute$(X_{i,\delta}, U, Y_{i,\delta}, \delta)$
    endfor
  else if $\gamma \times \delta \in P^-$
    // $Y_{\gamma\times\delta} = A^Y B^Y$
    for all $i \coloneqq 1$ **to** k
      hTransposedForwardSubstitute$(B^X_{i,\delta}, U, B^Y_{i,\delta}, \delta)$
    endfor
    $X_{\gamma\times\delta} \coloneqq A^Y B^X_{k\times\delta}$
  else // $\gamma \times \delta$ is no partition block
    $\{\gamma_1, \gamma_2\} \coloneqq \text{sons}_{T_I}(\gamma)$
    $\{\delta_1, \delta_2\} \coloneqq \text{sons}_{T_I}(\delta)$

    // transposed forward substitution on the higher
    // block row of $U|_{\gamma\times\delta}$ and $Y|_{\gamma\times\delta}$
    matValTranHForwardSubstitution$(X, U, Y, \gamma_1, \delta_1)$
    $Y_{\gamma_1,\delta_2} \coloneqq Y_{\gamma_1,\delta_2} \ominus_k \text{HMultiply}(X_{\gamma_1,\delta_1}, U_{\gamma_1,\delta_2})$
    matValTranHForwardSubstitution$(X, U, Y, \gamma_1, \delta_2)$

    // transposed forward substitution on the lower
    // block row of $U|_{\gamma\times\delta}$ and $Y|_{\gamma\times\delta}$
    matValTranHForwardSubstitution$(X, U, Y, \gamma_2, \delta_1)$
    $Y_{\gamma_2,\delta_2} \coloneqq Y_{\gamma_2,\delta_2} \ominus_k \text{HMultiply}(X_{\gamma_2,\delta_1}, U_{\gamma_1,\delta_2})$
    matValTranHForwardSubstitution$(X, U, Y, \gamma_2, \delta_2)$
  endelse
end

[Hac09, 7.6.3]

9.8 GMRES

GMRES stands for generalized minimal residual method. It is a general purpose method for the numerical solution of linear systems in the form $M\mathbf{x} = \mathbf{b}$. We treat the method throughout this text as a blackbox method. The GMRES interacts with the system matrix only in the form of a matrix-vector product during each iteration step. It is therefore almost trivial to interface $\mathcal{H}$-matrices with the method. One only has to

implement the $\mathcal{H}$-matrix-vector product. In addition to the cost of the matrix-vector product, the GMRES incurs additonal costs of $O(ln)$ during iteration step l. Therefore the total additional cost over these l steps is $O(l^2 n)$. It is therefore important to keep the number of steps low. An extension to address this problem is the so called GMRES with restarts. Another approach is preconditioning. We give [GvL13, Section 11.4.3, page 642] as a reference.

9.8.1 Preconditioned GMRES

Preconditioning is a tool to improve the convergence rate of the GMRES. Let $M\mathbf{x} = \mathbf{b}$, with $M \in \mathcal{H}(k)$ be a system that we want to solve. We use a (very) low-rank $\mathcal{H}$-LU-factorization of M:

$$LU \approx M, \quad \text{with } L, U \in \mathcal{H}(k') \text{ and } k' < k$$

We transform the system to be

$$(LU)^{-1} M\mathbf{x} = (LU)^{-1}\mathbf{b}.$$

The argument for preconditioning is $(LU)^{-1}M \sim I$ and therefore the combined matrix should be well conditioned. For reference and further information on preconditioned GMRES see [GvL13, Algorithm 11.5.2, page 653].

10 Cross approximation

So far we have introduced the $\mathcal{H}$-arithmetic and we have demonstrated the existence of low rank approximations of admissible blocks (at low to medium frequencies). Now we come to the actual assembly of the far-field blocks. There exist approaches based on direct interpolation of the kernel functions. See [BGH03, Section 3.2] for more information on the interpolation techniques. We employ a purely algebraic method; called cross approximation. The method iteratively samples one row and column (therefore cross approximation) of the matrix to be approximated. Then the information that was sampled in earlier steps is subtracted from the current samples. The method therefore effectively samples from the residuum of the target matrix and the approximant.

10.1 Cross approximation with full pivoting

We were introduced to the following algorithms in [BR03, page 16]. We start by giving the simplest form of the cross approximation; cross approximation with full pivoting. The so called pivot element describes the element at the center of the cross and therefore uniquely identifies the sampled row and column. Full pivoting is computationally too complex in practice, as we try to improve on the $O(n^2)$ matrix assembly time for full matrices. It serves us as an introduction and a benchmark for less informed pivoting schemes.

Let $A \in \mathbb{C}^{m \times n}$ be a given matrix. In our case A would be the system matrices in equation (3.10). We formulate the cross approximation to return an approximation to A in a similar shape as the compressed SVD:

$$UDV \approx A,$$

where D is diagonal.

In the following we will use a Matlab style notation for rows and columns of a matrix. The i-th row of a matrix is $A_{i:}$. The j-th column of a matrix is $A_{:j}$. A_{ij} signifies the element of matrix A with row index i and column index j. Let the constant $Tiny$ be a small multiple of the machine accuracy.

Algorithm 10.1: fullPivCA(A, k)

input: $A \in \mathbb{C}^{m\times n}$, $k \in \mathbb{N}$ // target matrix and maximum rank of output
output: $U \in \mathbb{C}^{m\times k}$, $D = \mathrm{diag}(D_{11}, \ldots, D_{kk}) \in \mathbb{C}^{k\times k}$, $V \in \mathbb{C}^{k\times n}$
begin
    $U := 0^{m\times k}$, $D := 0^{k\times k}$, $V := 0^{k\times n}$ // initialize the output matrices
    $A^1 := A$ // copy the input matrix.

    for $l := 1$ **to** k **do**
        // at this point, A^l is the residuum $A - UDV$
        $(i^l, j^l) := \mathrm{argmax}(|A^l|_{ij})$ // find the maximum absolute element of A

        $d^l := 1/A_{i^l j^l}$ // set the pivot element d^l to $1/A_{i^l j^l}$
        if $(1/|d^l| < Tiny)$ // the residuum is already close to zero
            return U, D, V // stop the sampling iteration
        endif

        $\mathbf{u}^l := A_{:j^l}$ // the current column sample
        $\mathbf{v}^{l\intercal} := A_{i^l:}$ // the current row sample

        $U_{:l} := \mathbf{u}^l$ // set column l of U to $\mathbf{u}^l$
        $D_{ll} := d^l$ // set the l-th diagonal index of D to d^l
        $V_{l:} := \mathbf{v}^{l\intercal}$ // set row l of V to $\mathbf{v}^{l\intercal}$

        $A^{l+1} := A^l - d^l \mathbf{u}^l \mathbf{v}^{l\intercal}$ // subtract the current rank 1 matrix
    endfor

    return U, D, V
end

Updating the residuum of A via $A^l - d^l \mathbf{u}^l \mathbf{v}^{l\intercal}$ and finding the pivot element costs $O(mn)$ in each iteration. The overall computational cost of the cross approximation with full pivoting is therefore $O(kmn)$.

10.2 Cross approximation with partial pivoting

To reduce computational costs, we confine ourselves to partial pivoting. With partial pivoting we only need implicit knowledge of the matrix that we want to approximate.

The partial pivoting scheme uses a look back strategy to find the next row index of the pivot element by looking at the last sampled (residuum) column; look back in the sense

that the data is already in our approximant. We have to take care, not to re-choose the index of the last sampled row. We calculate the residuum of the new row and the already assembled matrix. We choose the pivot column index from this new data; therefore look ahead.

Algorithm 10.2: partialPivCA(k)

input: implicit $A \in \mathbb{C}^{m \times n}$, $k \in \mathbb{N}$ // target matrix and target rank
output: $U \in \mathbb{C}^{m \times k}$, $D = \mathrm{diag}(D_{11}, \ldots, D_{kk}) \in \mathbb{C}^{k \times k}$, $V \in \mathbb{C}^{k \times n}$
begin
    $U := 0^{m \times k}$, $D := 0^{k \times k}$, $V := 0^{k \times n}$ // initialize the output matrices
    $j^0 := \mathrm{random}\{1, ..., n\}$ // choose a random column index
    $\mathbf{u}^0 := A_{:j_0}$ // calculate the column vector $\mathbf{u}_0$ in $O(m)$

    for $l := 1$ **to** k **do**
        if($l \geq 2$) // if information has already been sampled
            $\mathbf{u}^{l-1}_{i^{l-1}} := 0$ // prevent choosing the last row index again
        endif
        // the max element of the previous residual column
        // determines the current row index
        $i^l := \mathrm{argmax}(|\mathbf{u}^{l-1}|_i)$
        // U, D and V have rank $l - 1$
        $\mathbf{v}^{l\intercal} := A_{i^l:} - (U_{i^l:}D)V$ // calculate the residual row
        // the max element of the current residual row
        // determines the current column index
        $j^l := \mathrm{argmax}(|\mathbf{v}^{l\intercal}|_j)$
        $d^l := 1/\mathbf{v}^{l\intercal}_{j^l}$ // set the pivot element $d^l = 1/(A - UDV)_{i^l j^l}$
        $\mathbf{u}^l := A_{:j^l} - U(DV_{:j^l})$ // calculate the residual column

        if $(1/|d^l| < Tiny)$ // the pivot element is close to zero
            return U, D, V // stop the sampling iteration
        endif
        // store the sampled information in the approximation matrices
        $U_{:l} := \mathbf{u}^l$ // set column l of U to $\mathbf{u}_l$
        $D_{ll} := d^l$ // set the l-th diagonal index of D to d^l
        $V_{l:} := \mathbf{v}^{l\intercal}$ // set row l of V to $\mathbf{v}^{l\intercal}$
    endfor

    return U, D, V
end

The runtime dominant operations in the partially pivoted cross approximation are the matrix-vector/vector-matrix products when calculating the residuum column/row. At iteration l this is done at $O(ln)$ and $O(lm)$ cost respectively. The partially pivoted cross approximation therefore has an overall complexity of $O(k^2(n + m))$. Even though

the algorithm is heuristic, [Sto04, Satz 3.4.9] gives strong convergence estimates for asymptotically smooth kernel functions. In our numerical experiments we found the performance to be very close to the optimal SVD.

10.3 Adaptive cross approximation with partial pivoting

Algorithm (10.2) assumed, that the desired rank of the approximant is a priori known. We give a modified version that stops the approximation at a prescribed relative error.

Algorithm 10.3: ACA($\gamma \times \delta, \epsilon_{error}$)

input: implicit $A \in \mathbb{C}^{\gamma \times \delta}$ // we only require implicit knowledge A
$\epsilon_{error} \in \mathbb{R}^+$ // desired maximum relative error
output: $U \in \mathbb{C}^{\gamma \times k}$, diagonal $D \in \mathbb{C}^{k \times k}$, $V \in \mathbb{C}^{k \times \delta}$ // low-rank matrix
// k is the eventual rank

begin

$U := 0^{\gamma \times 1}$, $D := 0^{1 \times 1}$, $V := 0^{1 \times \delta}$ // initialize empty output matrices

$j^0 :=$ random$\{1, ..., |\delta|\}$ // choose a random column index
$\mathbf{u}^0 := A_{:j_0}$ // calculate the column vector $\mathbf{u}_0$ in $O(|\gamma|)$
maxRank $:= \min\{|\gamma|, |\delta|\}$

for $l := 1$ **to** maxRank **do**
  if$(l \geq 2)$ // if information has already been sampled
    $\mathbf{u}^{l-1}_{i^{l-1}} := 0$ // prevent choosing the last row index again
  endif
  // the max absolute element of the previous residual column
  // determines the current row index
  $i^l := \text{argmax}_{i \in \gamma}(|\mathbf{u}^{l-1}|_i)$
  $\mathbf{v}^{l\intercal} := A_{i^l:} - (U_{i^l:}D)V$ // calculate the residual row
  // the max element of the current residual row
  // determines the current column index
  $j^l := \text{argmax}_{j \in \delta}(|\mathbf{v}^{l\intercal}|_j)$
  $d^l := 1/\mathbf{v}^{l\intercal}_{j^l}$ // set the pivot element $d^l = 1/(A - UDV)_{i^l j^l}$
  $\mathbf{u}^l := A_{:j^l} - U(DV_{:j^l})$ // calculate the residual column

  if $(1/|d^l| < Tiny)$ // the pivot element is close to zero
    return U, D, V // stop the sampling iteration
  endif
  // store the sampled information in the approximation matrices
  $U_{:l} := \mathbf{u}^l$ // set column l of U to $\mathbf{u}_l$
  $D_{ll} := d^l$ // set the l-th diagonal index of D to d^l
  $V_{l:} := \mathbf{v}^{l\intercal}$ // set row l of V to $\mathbf{v}^{l\intercal}$
  // check the error condition

if $(\left\|d^l \mathbf{u}^l \mathbf{v}^{l\intercal}\right\|_F \leq \epsilon_{error} \left\|UDV\right\|_F)$
// the last sampling step gained little additional information
// $\Longrightarrow$ relative accuracy reached
return U, D, V // stop the sampling iteration
endif
// grow U an additional zero column
// grow D (in vector form) an additional zero element
// grow V an additional zero row
endfor

return U, D, V
end

The norm $\left\|d^l \mathbf{u}^l \mathbf{v}^{l\intercal}\right\|_F$ of the rank 1 matrix is efficiently calculated as

$$\left\|d^l \mathbf{u}^l \mathbf{v}^{l\intercal}\right\|_F = |d^l| \left\|\mathbf{u}^l\right\|_F \left\|\mathbf{v}^{l\intercal}\right\|_F .$$

The Frobenius norm of the approximant UDV can be computed economically in a recursive manner: Let $U^{l-1}D^{l-1}V^{l-1}$ be of rank $l-1$ and its norm be known. Then

$$\begin{aligned}\left\|U^l D^l V^l\right\|_F &= \left\|U^{l-1}D^{l-1}V^{l-1} + d^l \mathbf{u}^l \mathbf{v}^{l\intercal}\right\|_F \\ &= \sqrt{\begin{array}{l}\left\|U^{l-1}D^{l-1}V^{l-1}\right\|_F^2 + \left\|d^l \mathbf{u}^l \mathbf{v}^{l\intercal}\right\|_F^2 \\ + \operatorname{Re}\left(2\sum_{i=1}^{l-1}(\overline{d^l} D_{ii})(\mathbf{u}^{l*} U^l_{:i})(V^l_{i:}\overline{\mathbf{v}}^l)\right)\end{array}}\end{aligned}$$

The * stands for the Hermitian adjoint, Re is the real part operator and the $\overline{d^l}$ is the complex conjugate of d^l. The above recursion step has a computational complexity of $O(l(|\gamma|+|\delta|))$. The overall complexity thus remains $O(k^2(|\gamma|+|\delta|))$. The above recursion is an extension of [BR03, page 17] to the complex domain. The underlying heuristic of the stopping criterion is that $\left\|d^l \mathbf{u}^l \mathbf{v}^{l\intercal}\right\|_F \approx \left\|A - U^l D^l V^l\right\|_F$ and $\left\|U^l D^l V^l\right\|_F \approx \|A\|_F$. We see in a later section that the heuristic works well in practice.

11 The complete Helmholtz $\mathcal{H}$-BEM

In the preceding chapters we gathered all the necessary ingredients to replace the arithmetic of the Helmholtz BEM with $\mathcal{H}$-arithmetic. Now we bring all the pieces together in form of the complete algorithm.

Algorithm 11.1: HelmholtzBEM

input: Index set I for the triangular domain and corresponding boundary conditions.
$a_i\phi_i + b_i v_i = f_i \quad \text{for } i \in I$
$\epsilon_{error} \in \mathbb{R}^+$ // error target

output: $\boldsymbol{\phi} \in \mathbb{C}$ // solution to the Helmholtz equation on given triangular domain with given boundary conditions
begin
$T_I \coloneqq \text{constructClusterTree}(I)$
$\text{reindexClusterTree}(T_I)$
$T_{I\times I} \coloneqq \text{constructBlockClusterTree}(T_I, T_I)$
$P \coloneqq \text{minimalAdmissiblePartition}(T_{I\times I})$
$T_{I\times I} \coloneqq T(I \times I, P)$ // trim $T_{I\times I}$

calculate $\boldsymbol{\phi}$ // from substitution via boundary conditions;
// not the actual complete solution

$M_v, M_\phi \in \mathcal{H}(I \times I, P)$ // initialize system matrices
// assemble the partition blocks
for all $b \in P^+$ // approximate admissible blocks
// adaptive cross approximation with prescribed tolerance
$M_v|_b \coloneqq ACA(b, \epsilon_{error})$
$M_\phi|_b \coloneqq ACA(b, \epsilon_{error})$
endfor
for all $b \in P^-$ // calculate full rank blocks
calculate $M_v|_b$ // implicit substitution via boundary conditions
calculate $M_\phi|_b$
endfor

// incident field as right-hand side vector
$\mathbf{f} \coloneqq \boldsymbol{\phi}^{in} + \alpha \boldsymbol{v}^{in}$ // α is the coupling factor

$\text{MVM}(\mathbf{f}, M_\phi, \boldsymbol{\phi}, I \times I)$ // $f \coloneqq f + M_\phi \cdot \phi$

// now solve $M_\upsilon \cdot \boldsymbol{\upsilon} = f$ for $\boldsymbol{\upsilon}$

// low rank LU preconditioning
// create compressed copy of M_υ via block-wise compressedSVD
$M_\upsilon^k \coloneqq \text{compress}(M_\upsilon, k)$ // use very small k
$L^k \in \mathcal{H}(k, I \times I)$ // initialize L and U
$U^k \in \mathcal{H}(k, I \times I)$ // an efficient implementation would write over M_υ^k

$\text{LU}(M_\upsilon^k, L^k, U^k, I \times I)$ // factorize M_υ^k in lower- and upper-triangular $\mathcal{H}$-matrix

// preconditioned GMRES system: $(L^k U^k)^{-1} M_\upsilon \boldsymbol{\upsilon} = (L^k U^k)^{-1} \mathbf{f}$
// calculate right-hand side for the preconditioned system
$\mathbf{f}_{precond} \coloneqq \text{luSolve}(M_\upsilon^k, f)$ // $\mathbf{f}_{precond} \coloneqq (L^k U^k)^{-1} \mathbf{f}$

// use matrix-free GMRES implementation;
// the following three steps replace the usual matrix-vector product:
1. $\text{MVM}(\mathbf{v}', M_\phi, \mathbf{v}_{input}, I \times I)$ // $\mathbf{v}' \coloneqq M_\phi \mathbf{v}_{input}$
2. $\text{hForwardSubstitute}(L^k, \mathbf{v}'', \mathbf{v}', I)$ // $\mathbf{v}'' \coloneqq (L^k)^{-1} \mathbf{v}'$
3. $\text{hBackwardSubstitute}(U^k, \mathbf{v}''', \mathbf{v}'', I)$ // $\mathbf{v}'' \coloneqq (U^k)^{-1} \mathbf{v}'$

// iterative solver with prescribed accuracy and restarts
$\text{GMRES}(L^k, U^k, M_\upsilon, \mathbf{f}_{precond})$
calculate $\boldsymbol{\phi}$ and $\boldsymbol{\upsilon}$ with re−substitution via boundary conditions

end

12 Numerical accuracy

Let $M \in \mathbb{C}^{I \times J}$. The Frobenius norm of M is calculated element-wise:

$$\|M\|_F = \sqrt{\sum_{i \in I, j \in J} |M_{ij}|^2}$$

Let P be a block partition of $I \times J$. The Frobenius norm of M can then be expressed block-wise:

$$\|M\|_F = \sqrt{\sum_{b \in P} \|M|_b\|_F^2}$$

The simplest way to control the relative approximation error ϵ_{relErr} in our $\mathcal{H}$-matrices is to simply enforce ϵ_{relErr} for every admissible block.

$$\begin{aligned} \frac{\|M - M_{\mathcal{H}}\|_F^2}{\|M\|_F^2} &= \frac{\sum_{b=\gamma\times\delta \in P^+} \|M|_b - M_{\mathcal{H}}|_b\|_F^2}{\|M\|_F^2} \\ &\leq \frac{\sum_{b=\gamma\times\delta \in P^+} (\epsilon_{relErr} \|M|_b\|_F)^2}{\sum_{b \in P} \|M|_b\|_F^2} \\ &\leq (\epsilon_{relErr})^2 \end{aligned} \tag{12.1}$$

We do not require explicit knowledge of the block norms $M_{\mathcal{H}}|_b$ for the above approach. We use the strategy from equation (12.1) in our adaptive cross approximation algorithm (10.3) for the block assembly. In our numerical test, the actual approximation error was always one to two orders of magnitude lower than the target value; so there is still room to impove on the estimate.

12.1 Balanced block rank

If we want to control the global relative approximation error ϵ_{relErr} in our $\mathcal{H}$-matrices, so that each admissible block has about the same rank, we can allocate each block a size-dependent absolute error. Let $b = \gamma \times \delta$ be an admissible block and ϵ_{abs}^b its absolute approximation error, with

$$\epsilon_{abs}^b = \|M|_b - M_{\mathcal{H}}|_b\|_F \leq \epsilon_{relErr} \sqrt{\frac{|\gamma| \cdot |\delta|}{\sum_{\gamma' \times \delta' \in P^+} |\gamma'| \cdot |\delta'|}} \|M\|_F \,.$$

Then:

$$\begin{aligned}
\|M - M_{\mathcal{H}}\|_F^2 &= \sum_{b=\gamma\times\delta\in P} \|M|_b - M_{\mathcal{H}}|_b\|_F^2 \\
&= \sum_{b=\gamma\times\delta\in P^+} (\epsilon_{abs}^b)^2 \\
&\leq \sum_{b=\gamma\times\delta\in P^+} \epsilon_{relErr}^2 \frac{|\gamma|\cdot|\delta|}{\sum_{\gamma'\times\delta'\in P^+} |\gamma'|\cdot|\delta'|} \|M\|_F^2 \\
&= \epsilon_{relErr}^2 \|M\|_F^2
\end{aligned}$$

The above approach requires knowledge of the global norm $\|M\|_F^2$. One can iterate over all partition blocks and do one step of ACA on each admissible block; always updating the global norm estimate $\|M_{\mathcal{H}}\|_F \sim \|M\|_F$ in the process.

13 Memory and runtime

In this section we will briefly describe the memory and runtime requirements some of the $\mathcal{H}$-arithmetical operations, that we have introduced above. The authors of ([BGH03, page 94]) introduce the concept of a sparsity constant as a tool to give a measure of sparsity to $\mathcal{H}$-matrices. In case of ordinary sparse matrices the sparsity pattern gives the maximum number of non-zero entries per row or column.

Similarly for a hierarchical matrices the sparsity constant gives the maximum number of blocks per block-row or block-column.

Definition 13.1 (Sparsity constant). *Let $T(I \times I, P)$ be a block cluster tree. We define the sparsity constant C_{sp} of $T(I \times I, P)$ by:*

$$C_{sp} \coloneqq \max \left\{ \begin{matrix} \max\limits_{\gamma \in T_I} |\{\delta \in T_I | \gamma \times \delta \in T(I \times I, P)\}|, \\ \max\limits_{\delta \in T_I} |\{\gamma \in T_I | \gamma \times \delta \in T(I \times I, P)\}| \end{matrix} \right\}. \tag{13.1}$$

We can use the sparsity constant for example to derive an upper bound for the number of partition blocks:

$$\begin{aligned} |P| &= \sum_{b \in P} 1 \\ &= \sum_{\gamma \in T_I} |\{\delta \in T_I | \gamma \times \delta \in P\}| \\ &\leq \sum_{\gamma \in T_I} C_{sp} \\ &\overset{\star}{=} (2|\mathcal{L}(T_I)| - 1) C_{sp} \\ &\leq (2|I|/n_{min} - 1) C_{sp} \end{aligned} \tag{13.2}$$

($\star$ A binary tree with l leaves has $2l - 1$ vertices overall.)

We cite the following definition from ([BGH03, page 103]) as it relates to the sparsity constant:

Definition 13.2 (Locality). *We assume that the triangle supports are locally separated in the sense that there exist two constants C_{sep} and n_{min}, so that:*

$$\max_{i \in I} |\{j \in I | \operatorname{dist}(\triangle_j, \triangle_i) \leq C_{sep}^{-1} \operatorname{diam}(\triangle_i)\}| \leq n_{min} \tag{13.3}$$

the left-hand side describes the maximum number of "rather close" triangles. The bound n_{min} is the same as for the construction of T_I.

This locality assumption on the triangle mesh leads the authors to an upper bound for the sparsity constant by C_{sep} ([BGH03, page 104]):

$$C_{sp} \leq (4 + 16C_{sep} + 8\sqrt{2}\eta^{-1}(1 + C_{sep}))^2.$$

So the sparsity constant is independent of $|I|$.

13.1 Memory requirements

Let $k \in \mathbb{N}$ again be the rank of the low-rank matrices in the far-field blocks ($b \in P^+$). It holds for all full-rank blocks $b = \gamma \times \delta \in P^-$:

$$\min\{|\gamma|, |\delta|\} \leq n_{min}$$

The sparsity constant relates to the memory usage as follows:

$$\begin{aligned}
\mathcal{M}_{memory}(k, P) &= \sum_{\gamma\times\delta\in P^+} k(|\gamma| + |\delta|) + \sum_{\gamma\times\delta\in P^-} (|\gamma| \cdot |\delta|) \\
&\leq \max\{n_{min}, k\} \sum_{\gamma\times\delta\in P} (|\gamma| + |\delta|) \\
&\leq C_{sp} \max\{n_{min}, k\} \left(\sum_{p=1}^{\operatorname{depth}(T)} \sum_{\gamma\in T^{(p)}} |\gamma| + \sum_{p=1}^{\operatorname{depth}(T)} \sum_{\delta\in T^{(p)}} |\delta| \right) \\
&\leq 2C_{sp} \max\{n_{min}, k\} \sum_{p=1}^{\operatorname{depth}(T)} \sum_{\gamma\in T^{(p)}} |\gamma| \\
&= 2C_{sp} \max\{n_{min}, k\} \sum_{p=1}^{\operatorname{depth}(T)} |I| \\
&= 2C_{sp} \max\{n_{min}, k\} \operatorname{depth}(T)|I| \\
&= O(k|I| \log_2(|I|))
\end{aligned}$$

13.2 Runtime of the assembly

Let $b = \gamma \times \delta \in T_{I\times J}$ be a block cluster. The assembly time of b via the cross approximation with partial pivoting was given as $O(k^2|\gamma| + |\delta|)$. Now we want to estimate the complexity of the entire matrix assembly.

$$C_{assembly}(k, P) = \sum_{\gamma\times\delta\in P^+} O(k^2(|\gamma| + |\delta|)) + \sum_{\gamma\times\delta\in P^-} (|\gamma| \cdot |\delta|)$$

If we compare the expression above with the estimate for the memory consumption we, see directly:

$$C_{assembly}(k, P) = O\left(k^2|I| \log_2(|I|)\right)$$

13.3 Runtime of the $\mathcal{H}$-matrix-vector product

The storage requirements of a full matrix $M \in \mathbb{C}^{\gamma \times \delta}$ are $|\gamma||\delta|$. The costs of a full matrix vector product are $8|\gamma||\delta| - 2|\delta|$. The storage requirements of a low rank matrix are $k(|\gamma| + |\delta|)$ and finally the costs of a low-rank matrix vector product are $8k(|\gamma| + |\delta|) - 2(|\delta| + k)$. As an $\mathcal{H}$-matrix consist only of low and full rank blocks, it holds:

$$C_{\mathcal{H}MV}(k, P) \leq 8\mathcal{M}_{memory}(k, P) = O\left(k|I| \log_2 |I|\right)$$

13.4 Runtime of the $\mathcal{H}$-matrix-matrix product and the $\mathcal{H}$-LU-factorization

We cite a very condensed result from [GH03, Theorem 2.24]: The cost of the $\mathcal{H}$-matrix-matrix product is

$$C_{\mathcal{H}MM}(k, P) = O\left(k^3|I| \log_2^2 (|I|)\right)$$

Also [Hac09, Lemma 7.8.23] have shown that the cost of the $\mathcal{H}$-LU-factorization is about one third the cost of the $\mathcal{H}$-matrix-matrix product. Therefore

$$C_{\mathcal{H}LU}(k, P) = O\left(k^3|I| \log_2^2 (|I|)\right)$$

as well.

13.5 Runtime of the $\mathcal{H}$-matrix compression

To construct an efficient preconditioner for the GMRES, we need to truncate the system matrix to a lower rank. This is done by using the compressed SVD (algorithm 4.1) on all admissible blocks. We had a bound for the cost of the compressed SVD:

$$C_{CSVD} = 16k^2(n + m) + 90k^3$$

With that we can estimate the runtime of the matrix compression.

$$\begin{aligned} C_{\text{Compress}}(k, P) &= 90|P^+|k^3 + \sum_{\gamma \times \delta \in P^+} (16k^2(|\gamma| + |\delta|) \\ &= O\left(k^2|I| \log_2 (|I|) + k^3|I|\right) \end{aligned}$$

We have seen in section 8.2, that the expected compression at high frequencies is not optimal. The cubic k term is therefore problematic for larger κ. It would be beneficial then, to just assemble a new low-accuracy $\mathcal{H}$-matrix with ACA for the preconditioner and not do the compression.

13.6 Runtime of the complete Helmholtz $\mathcal{H}$-BEM

Let $|I| = n$. We build up the cost estimate for the complete BEM from its dominant terms. Let k be the prescribed rank for our $\mathcal{H}$-matrix approximation and let k_{prec} be the rank of the preconditioner, with $k_{prec} < k$. Let also l be the number of required GMRES steps. Then we have

$$\begin{aligned} C_{BEM}(k, k_{prec}, P, l) &= C_{assembly}(k, P) + lC_{\mathcal{H}MV}(k, P) + C_{GMRES}(P, l) + C_{\mathcal{H}LU}(k_{prec}, P) \\ &= O\left(k^2 n \log_2(n)\right) + O\left(lkn \log_2 n\right) + O(l^2 n) + O\left(k_{prec}^3 n \log_2^2(n)\right) \\ &= O\Bigg(\left(k(k+l)\log_2(n) + l^2 + k_{prec}^3 \log_2^2(n)\right) n\Bigg) \end{aligned}$$

We conclude that the runtime of the $\mathcal{H}$-BEM is least $O\left(k^2 n \log_2(n)\right)$ in the case of good convergence of the GMRES with very low-rank preconditioning. The runtime is then dominated by the matrix assembly. We have so far no estimates for the relationship between the rank of the preconditioner and the rate of convergence of the GMRES.

14 The implementation

This work is based on an earlier bachelor thesis on the boundary element method [Gar17]. In the course of that thesis the collocation boundary element method for the Helmholtz equation was implemented in a program called 3d_BEM. The program consists of a simple script interpreter, a graphic user interface with 3D viewer and a boundary element solver. The program enables a flexible set-up of small-scale acoustic simulations. One can manually define 3D mesh geometries in scripts or register external mesh files for more complex geometries. The program offers some functionality for mesh manipulation like scaling, rotating and translating. Meshes can also be automatically refined to account for given high-frequency requirements. The program accepts mixed boundary conditions for everything in between pure radiation and pure scattering problems. Incident fields in the form of spherical waves and plane waves can also be supplied.

The novelty that this thesis brings to 3d_BEM is the replacement of the conventional matrix arithmetic in the boundary element solver with $\mathcal{H}$-arithmetic as described in the preceding chapters. This led to a substantial gain in efficiency. Where we were constrained to a few thousand triangle elements before, we have already calculated geometries with hundreds of thousands of elements at low frequencies.

14.1 A sphere

One of our benchmarking problems is a simple sphere on which a spherical wave is scattered. The mesh was generated with GMSH [GR09]. 3d_BEM accepts general flat triangle meshes in the GMSH format.

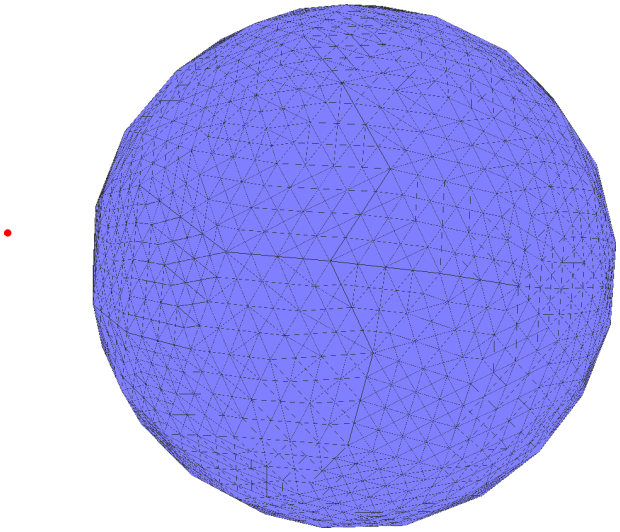


Figure 14.1: Spherical mesh with 8024 triangles

Figure 14.1 shows the spherical mesh. The radius of the sphere is 10 meters. The red dot to the left of the sphere signifies the origin of the spherical wave.

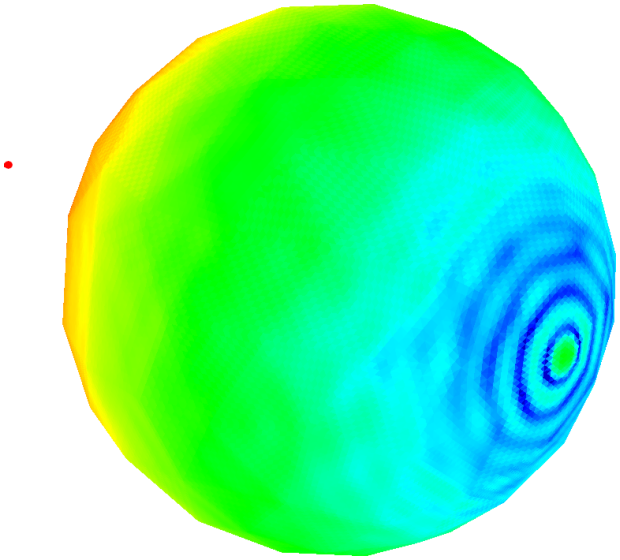


Figure 14.2: Spherical wave scatting at 400 Hz

Figure 14.2 shows the sphere, after a solution to the scattering problem on the surface of the sphere has been calculated. 3d_BEM uses a heat-map gradient to visualize the result. We will now discuss performance aspects of the $\mathcal{H}$-solver implementation.

14.1.1 Sphere with constant frequency

The following charts describe the best case scenario for the solver. The frequency is fixed at $100Hz$. The wavelength of $\lambda = 3.43m$ is in the same order as the dimensions of the geometry. We are therefore in the low to medium frequency regimen and can expect a constant rank per admissible block (see section 8.2 for reference). The targeted approximation tolerance ϵ was set to 0.01. The following chart shows the duration of the matrix assembly via the adaptive cross approximation over on increasingly refined mesh.

Note that we attribute the "kinks" in the following graphs mostly to uneven system performance. The performance data was collected manually and is not averaged. We feel the data still provides easily readable trends.

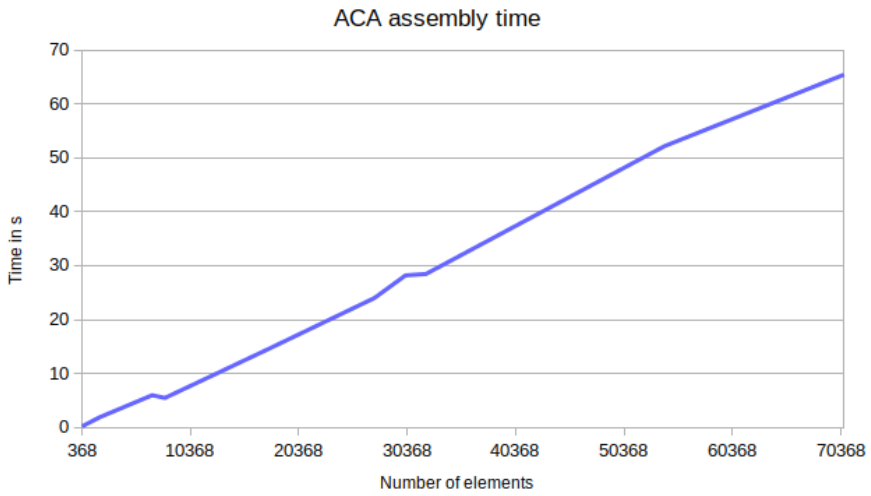
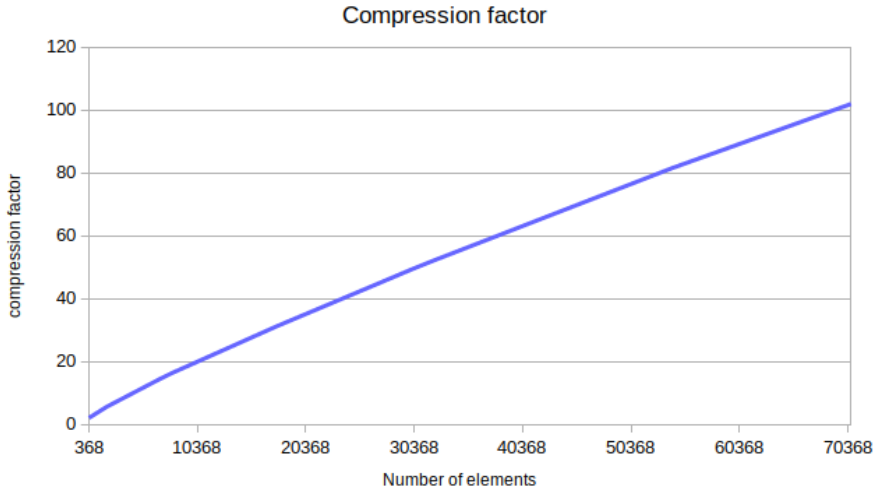


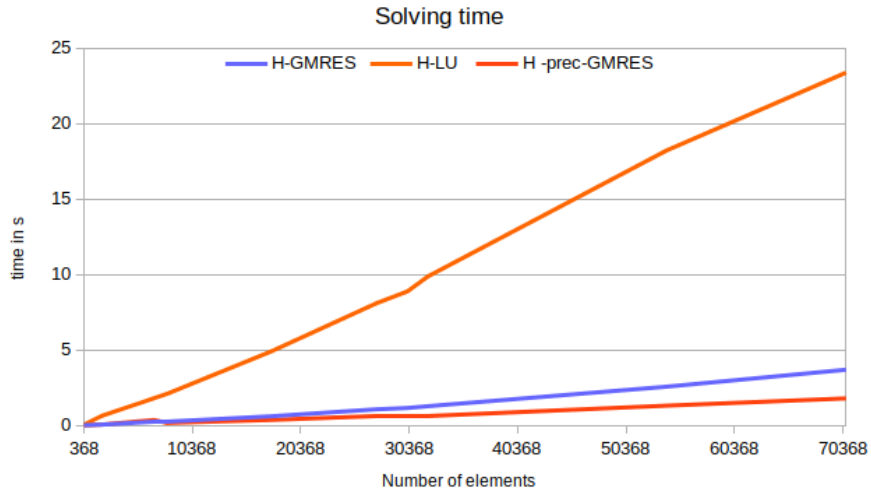
Figure 14.3: constant frequency, increasing mesh refinement

The algorithm duration clearly develops quasi linear with the number of triangles. This is a decent improvement over the full matrix assembly. The new method therefore allows us to simulate sound fields on much more detailed geometries, for the same frequency and geometry dimensions.

We now look at the achieved compression of the method.



The chart above shows the memory compression factor of the $\mathcal{H}$-matrices. It grows slightly less than linear. We observe almost linear memory consumption.



The $\mathcal{H}$-matrix vector product is very fast. Almost negligible in contrast to the assembly time. The preconditioned GMRES is quicker still, but the calculation of the rank-1 preconditioner is comparatively uneconomical here. If one were to solve the same system for round about 15 different right-hand sides, the preconditioner would amortize.

14.1.2 Sphere with rising frequency

We set the same spherical geometry up in a different context. Instead of keeping the frequency constant, we increase the frequency and enforce a maximum triangle edge length of $\lambda/2$, thereby increasing the mesh fineness and keeping the triangle density

roughly constant per wavelength. As we discretize a surface geometry, the number of triangles rises in the order of the squared frequency.

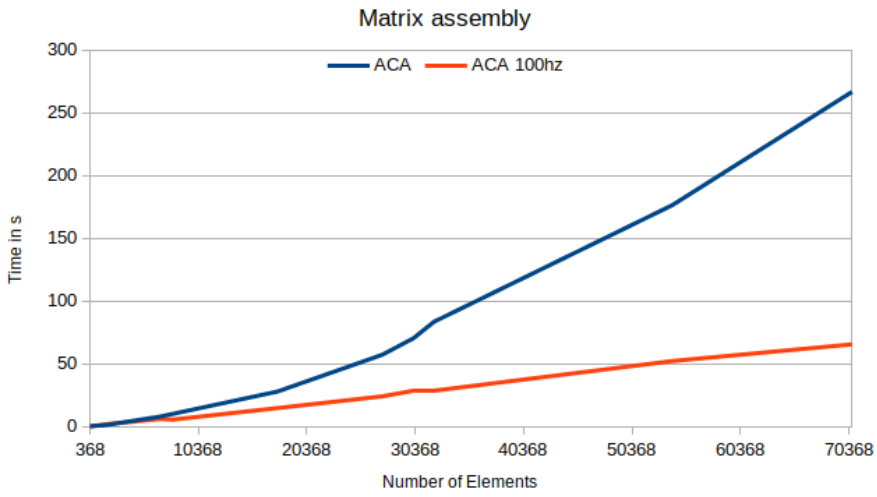


Figure 14.4: rising frequency

The orange graph is from figure 14.3. It shows the assembly time for the same mesh, but at a fixed frequency of 100Hz. The performance of the ACA clearly degrades over a rising frequency.

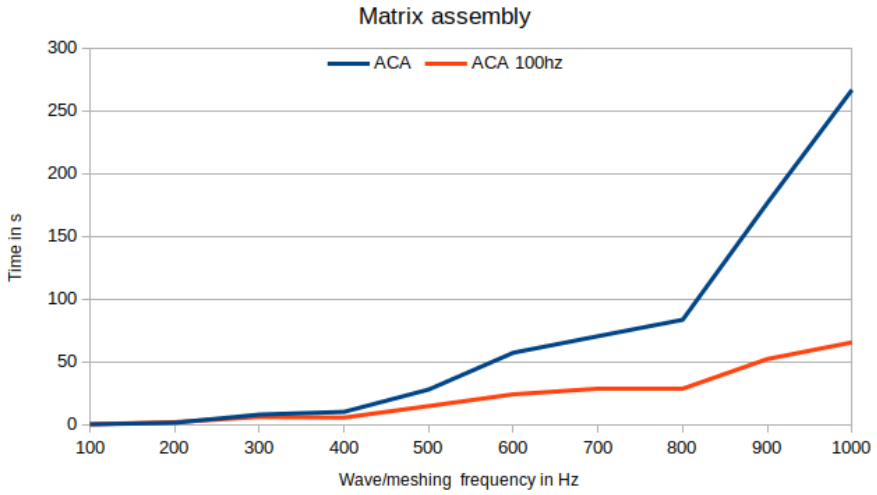


Figure 14.5: rising frequency

The chart above relates to the the same simulations as figure (14.4), but it shows the assembly time as a function of the frequency. The blue graph depicts a rising wave frequency (increasing κ) with a conformant mesh refinement. The orange graph depicts

the same underlying data as figure (14.3) (only mesh refinement, constant wave frequency of 100 Hz). The relationship between the number of triangles and the mesh frequency is only very roughly $O(\kappa^2)$ and is also not continuous. The crude mesh refinement scheme explains the marked kink at 900Hz.

14.2 A dipole loudspeaker

Now to a more practical simulation. We modelled a dipole loudspeaker, basically a speaker driver mounted in a board, in CAD and meshed the model with GMSH. We used a relative approximation error of $\epsilon = 0.01$ as a stopping criterion for the ACA.

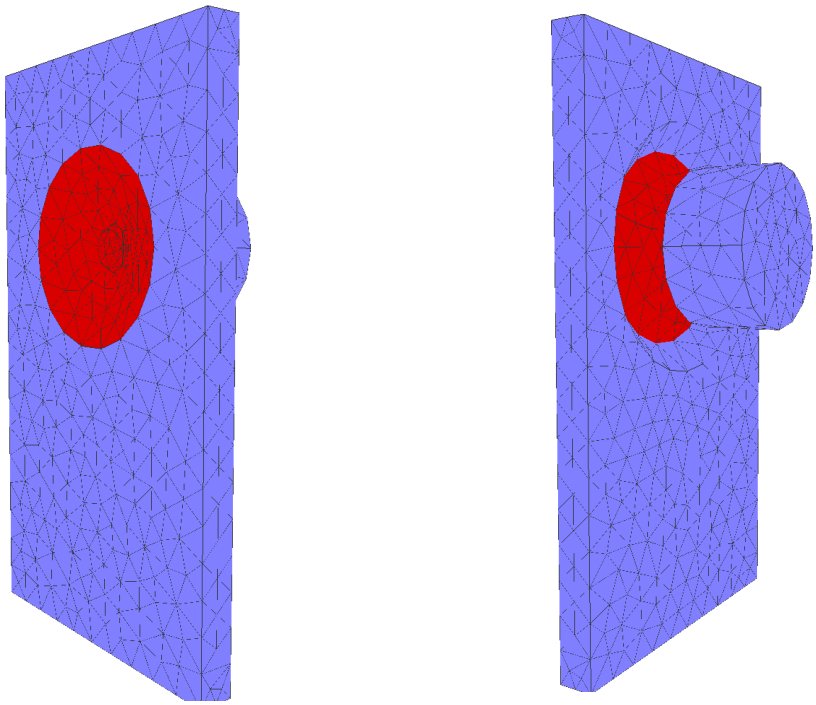


Figure 14.6: Dipole loudspeaker mesh model

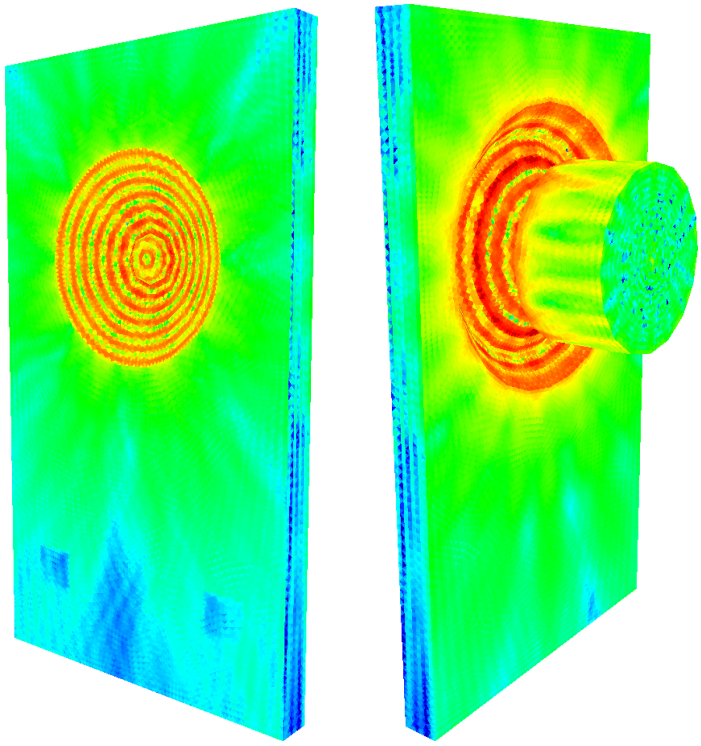


Figure 14.7: Loudspeaker at 40kHz

3d_BEM can compute and visualize sound fields in the domain, after a boundary solution has been calculated.

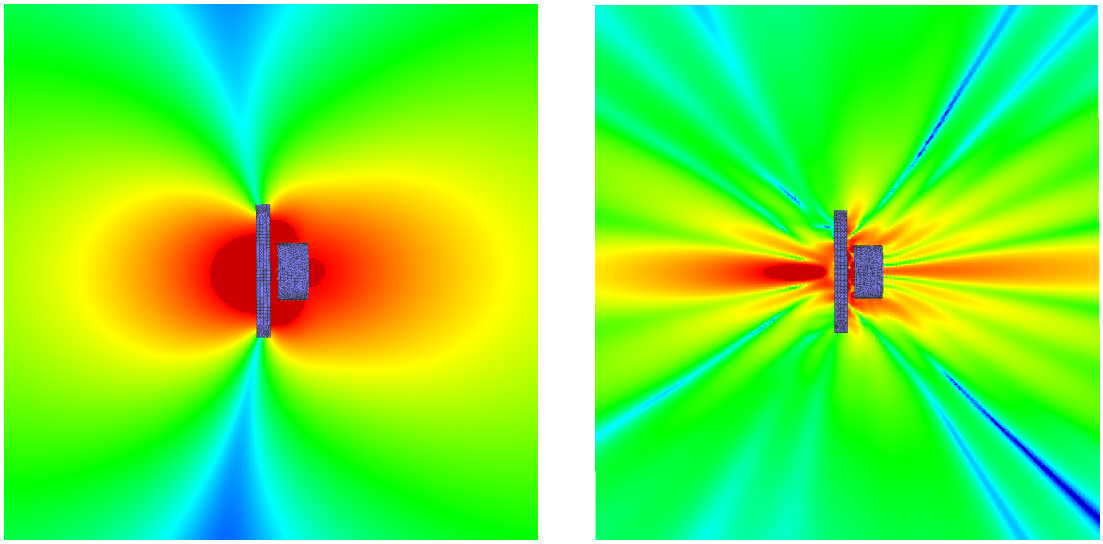
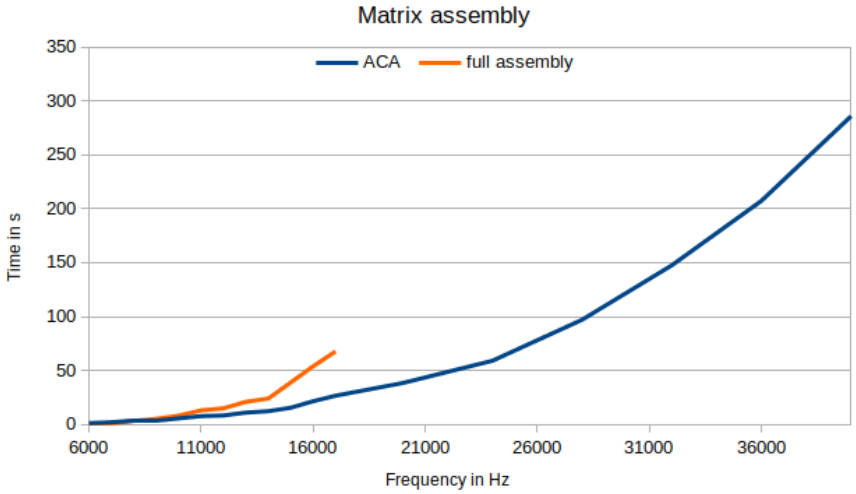


Figure 14.8: Sound field at 1kHz and 10 kHz



We simulated a rising frequency in full and in $\mathcal{H}$-arithmetic ($\epsilon_{ACA} = 0.01$), until each method ran out of memory. The orange graph shows the assembly of the full system. The blue graph visualizes the duration of the ACA assembly. The $\mathcal{H}$-technique clearly extends the workable frequency range.

15 Conclusion

The introduction of $\mathcal{H}$-arithmetic is a clear enhancement of the Helmholtz BEM. The achieved compression extends our reach on larger computable geometries and higher frequencies. Especially the adaptive cross approximation with unpreconditioned H-GMRES is relatively easy to implement and brings for our test cases all the benefits. Some geometries that we simulated had a higher GMRES-step count than others; but in no scenario so far could the preconditioner setup amortise for solving only one right-hand side. The $\mathcal{H}$-method showed stable behaviour. We observed a relative solution error, proportional to the relative assembly error. We used the simple error bound (equation 12.1) as a stopping criterion for our implementation of the adaptive cross approximation. We often found the actual approximation error to be two orders of magnitude lower than targeted. So there is still some room to shed redundancy. The accelerated method worked great at low frequencies. We solved systems with up to half a million unknowns on a laptop within minutes. Its in the high-frequency band, where the method becomes inefficient. A rank dependency of $O(\sqrt{n})$ was given in [CDC17] for the high frequency case. This translates to an asymptotically (large κ) worse assembly time of $O(n^2 \log_2(n))$, but a better solving time and memory complexity of $O(n^{\frac{3}{2}} \log(n))$.

15.1 Outlook

For anyone who wants to go further with the boundary element method for the Helmholtz equation, there is a clear way forward from the standpoint of this thesis. The authors of [Hac09] have developed the $\mathcal{H}$-matrix technique further. The improved constructions are called $\mathcal{H}^2$-matrices and reduce the memory consumption and cost of the matrix-vector product down to $O(n)$ [Bö06]. The main limitation of $\mathcal{H}$-matrices is the still constrained frequency range, due to its loss in compression in the higher-frequency band. $\mathcal{H}^2$-matrices have been adapted to highly oscillatory kernels [BM17]. The restriction to only medium frequencies has been overcome.

Bibliography

[BGH03] Steffen Börm, Lars Grasedyck, and Wolfgang Hackbusch. Hierarchical matrices. Technical report, 2003.

[BH07] Lehel Banjai and Wolfgang Hackbusch. Hierarchical matrix techniques for low- and high-frequency Helmholtz problems. *IMA Journal of Numerical Analysis*, 2007.

[BM71] A. J. Burton and G. F. Miller. A discussion on numerical analysis of partial differential equations - the application of integral equation methods to the numerical solution of some exterior boundary-value problems. *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 1971.

[BM17] Steffen Börm and Jens M. Melenk. Approximation of the high-frequency helmholtz kernel by nested directional interpolation: error analysis. *Numerische Mathematik*, 137, 2017.

[BR03] M. Bebendorf and S. Rjasanow. Adaptive low-rank approximation of collocation matrices. *Computing*, 70, 2003.

[Bö06] Steffen Börm. H2-matrix arithmetics in linear complexity. *Computing*, 2006.

[Bö19] Steffen Börm. Hierarchical matrix arithmetic with accumulated updates, 2019.

[CCH09] Ke Chen, Jin Cheng, and Paul J. Harris. A new study of the burton and miller method for the solution of a 3d helmholtz problem. *IMA Journal of Applied Mathematics*, 2009.

[CDC17] Stéphanie Chaillat, Luca Desiderio, and Patrick Ciarlet. Theory and implementation of $\mathcal{H}$-matrix based iterative and direct solvers for helmholtz and elastodynamic oscillatory kernels. *Journal of Computational Physics*, 351, 2017.

[Des17] Luca Desiderio. H-matrix based solver for 3d elastodynamics boundary integral equations. 2017.

[Gar17] Karl Garcke. The boundary element method - an introduction and application to a three-dimensional acoustical problem, 2017.

[GH03] Lars Grasedyck and Wolfgang Hackbusch. Construction and arithmetics of $\mathcal{H}$-matrices. *Computing*, 2003.

[GR09] Christophe Geuzaine and Jean-François Remacle. Gmsh: A 3-d finite element mesh generator with built-in pre- and post-processing facilities. *International Journal for Numerical Methods in Engineering*, 2009.

[GvL13] Gene H. Golub and Charles F. van Loan. *Matrix Computations*. JHU Press, fourth edition, 2013.

[Hac09] Wolfgang Hackbusch. *Hierarchische Matrizen : Algorithmen und Analysis*. 2009.

[Hac15] Wolfgang Hackbusch. *Hierarchical Matrices: Algorithms and Analysis*. Springer Publishing Company, Incorporated, 1st edition, 2015.

[Juh94] Peter Juhl. The boundary element method for sound field calculations. 1994.

[Kir98] S. Kirkup. *The Boundary Element Method in Acoustics: A Development in Fortran*. Integral equation methods in engineering. Integrated Sound Software, 1998.

[MW08] Steffen Marburg and Ting-Wen Wu. *Treating the Phenomenon of Irregular Frequencies*. Springer Berlin Heidelberg, 2008.

[Sto04] Mirjam Stolper. *Schnelle Randelementmethoden für die Helmholtz-Gleichung*. PhD thesis, Universität des Saarlandes, 2004.

[ZCGD15] Chang-Jun Zheng, Hai-Bo Chen, Hai-Feng Gao, and Lei Du. Is the burton–miller formulation really free of fictitious eigenfrequencies? *Engineering Analysis with Boundary Elements*, 2015.