Master Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computers

Method of Moments on GPU and Shape Synthesis Using Machine Learning Techniques

Bc. Martin Štrambach

Supervisor: doc. Ing. Miloslav Čapek, Ph.D.
Field of study: Open Informatics
Subfield: Artificial Intelligence
May 2020

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení:	**Štrambach**	Jméno: **Martin**	Osobní číslo: **420786**
Fakulta/ústav:	**Fakulta elektrotechnická**		
Zadávající katedra/ústav:	**Katedra počítačů**		
Studijní program:	**Otevřená informatika**		
Studijní obor:	**Umělá inteligence**		

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Paralelizace metody momentů a na ní založená tvarová syntéza s využitím strojového učení

Název diplomové práce anglicky:

Method of Moments on GPU and Shape Synthesis Using Machine Learning Techniques

Pokyny pro vypracování:

1. Seznamte se problematikou výpočetního řešení vyzařovacích problémů elektromagnetismu metodou momentů, zaměřte se na integrální rovnici pro intenzitu elektrického pole [1].
2. Navrhněte výpočetní model založený na metodě momentů s uvažováním po částech konstantních a lineárních bázových funkcí s využitím Galerkinovy metody [2].
3. Analyzujte možnosti implementace řešení s využitím masivně paralelních výpočetních prostředků [3] a zvolené řešení implementujete. Zhodnoťte případné zrychlení.
4. Studujte využití realizované implementace v řešení úlohy tvarové syntézy [4, 5].
5. Navrhněte využití aproximačních modelů výpočtu použitím metod strojového učení pro urychlení řešení úlohy tvarové syntézy [6, 7].
6. Navržené řešení implementujete a otestujete na zvolené případové studii.

Seznam doporučené literatury:

[1] Harrington, R. F.: Field Computation by Moment Methods, Wiley – IEEE Press, 1993.
[2] Gibson, W. C.: The Method of Moments in Electromagnetics, Chapman and Hall/CRC, 2014.
[3] CUDA by Example: An Introduction to General□Purpose GPU, 2010.
[4] Capek, M., Jelinek, L., Gustafsson, M.: Shape Synthesis Based on Topology Sensitivity, IEEE Transactions on Antennas and Propagation, Vol. 67, No. 6, pp. 3889-3901, June 2019.
[5] Capek, M., Jelinek, L., Gustafsson, M.: Inversion-Free Evaluation of Nearest Neighbors in Method of Moments, IEEE Antennas and Wireless Propagation Letters, Vol. 18, No. 11, pp. 2311-2315, Nov. 2019.
[6] Duda, R. O., Hart, P. E., Stork, D. G.: Pattern Classification (2nd edition), 2000.
[7] Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning, 2015.

Jméno a pracoviště vedoucí(ho) diplomové práce:

doc. Ing. Miloslav Čapek, Ph.D., katedra elektromagnetického pole FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **14.02.2020** Termín odevzdání diplomové práce: **22.05.2020**

Platnost zadání diplomové práce: **30.09.2021**

doc. Ing. Miloslav Čapek, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

.	
Datum převzetí zadání	Podpis studenta

Acknowledgements

I am grateful to many within the Department of Electromagnetic Field for suggestions and contributions made throughout. Foremost, I would like to thank my supervisor and a friend, Miloslav Čapek, for the continuous support of my master's study, for his patience, motivation, countless hours spent answering my questions, immense knowledge, and encouragement necessary to pursue such a work from the outset. To the many friends and family, who have both cheerfully supported and ultimately tolerated my studies – I could not have done it without you, but I promise, THAT'S IT AND NO MORE.

Declaration

Abstract

This thesis aims to describe and implement the method of moments in electromagnetics using graphical processing units utilizing CUDA technology. Speedups are evaluated with respect to the already existing implementation in Antenna Toolbox for MATLAB and verified with respect to commercial software. An effective implementation of the method of moments allows us to speed up automated antenna design process. In the part focused on shape synthesis, we theoretically evaluate and describe its behavior together with experimental use of machine learning algorithms like genetic algorithms and artificial neural networks for data classification.

Keywords: method of moments in electromagnetics, antennas, CUDA, shape synthesis, optimization, machine learning, classification

Supervisor: doc. Ing. Miloslav Čapek, Ph.D.

Abstrakt

Tato práce je zaměřena na popis a implementaci metody momentů v elektromagnetismu pro grafické akcelerátory s technologií CUDA. Je zhodnoceno realné zrychlení oproti již existující implementaci v anténním toolboxu AToM vyvíjeném v prostředí MATLAB. Správnost implementace je porovnána s komerčním softwarem. Efektivní implementace metody momentů navíc umožní zrychlení automatizovaného procesu návrhu antén. V části pracující s tvarovou syntézou antén k automatizovanému návrhu je teoreticky zhodnoceno a popsáno chování algoritmu společně s experimentálním použitím algoritmů umělé inteligence jako jsou genetické algoritmy a umělé neuronové sítě ke klasifikaci dat.

Klíčová slova: metoda momentů v elektromagnetismu, antény, CUDA, tvarová syntéza, optimalizace, strojové učení, klasifikace

Contents

Figures

Tables

Abbreviations

Adam	Adaptive Moment Estimation
ALU	Arithmetic Logic Unit
API	Application Programming Interface
AToM	Antenna Toolbox for MATLAB
AVX-512	Advanced Vector Extensions 512
CAD	Computer-Aided Design
CC	Compute Capability
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
GPGPU	General-Purpose computing on Graphics Processing Units
GPU	Graphics Processing Unit
HPC	High-Performance Computing
MoM	Method of Moments
MPI	Message Passing Interface
NP	Non-Polynomial
NSGA-II	Non-dominated Sorting Genetic Algorithm II
PEC	Perfect Electric Conductor
PTX	Parallel Thread Execution
RAM	Random Access Memory
RCI	Research Center for Informatics
RWG	Rao-Wilton-Glisson
SIMT	Single Instruction Multiple Threads
SMX	Streaming Multiprocessor

Chapter 1

Introduction

Small antennas are ubiquitous, from computers to wearable electronics like smartwatches, RFID chips and so on. It is necessary to make them as efficient as possible, usually with the desired shape and size. It puts high demands on the design of such antennas.

Antenna Toolbox for MATLAB (AToM) [1] is a software developed primarily for antenna design and optimization. It discretizes radiating bodies into triangular meshes [2] and implements electromagnetic solvers for radiating problems, one of them being Method of Moments (MoM) [3].

Most of the algorithms which use matrices (especially large matrices with millions of elements) can exploit speedups on modern graphic cards. From artificial intelligence to the systems of everyday use like weather predictions [4], programmers migrate their computationally expensive parts of the code to parallel execution and especially to Graphics Processing Unit (GPU). GPUs outperform processors in tasks with a huge number of independent computations (millions and more).

The emphasis in Central Processing Unit (CPU) design since 2005 has been put into parallelization rather than an increase of frequency (see Figure 1.1). Even though this change in design led to increased performance of CPUs, trends in the evolution of GPUs followed a similarly steep curve. For comparison of performance evolution, see Figure 1.2 and Figure 1.3.

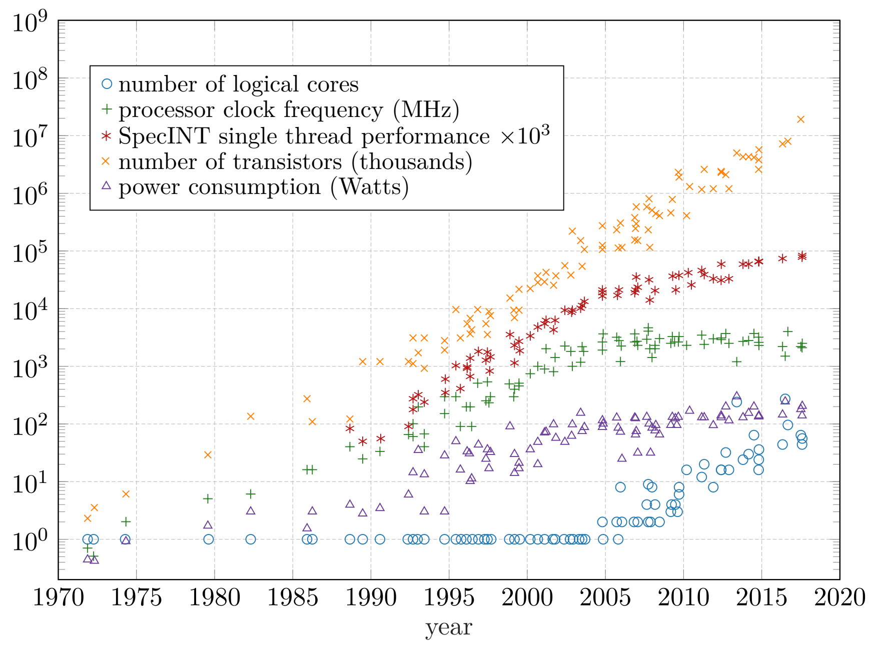


Figure 1.1: Timeline of the CPU development over time[1].

Nowadays, modern clusters already have multiple GPUs installed to computation nodes. The most of them use Nvidia's proprietary technology Compute Unified Device Architecture (CUDA) [6], which evolved into so-called industry standard in the past years [7]. Nodes can communicate using Message Passing Interface (MPI) [8] in order to distribute workload among them. This is supported with the fast development of all kinds of high-performance libraries which offer an Application Programming Interface (API) for parallel computation.

The performance evolution directly motivates the beginning of this work. MATLAB [9] as a scientific computing engine offers wrappers as well as direct access to CUDA with easy-to-use APIs. It is fairly easy to prototype GPU code and even launch compiled CUDA code that can achieve even higher speedups than non-native code. Our aim is to transform computationally demanding CPU code from AToM to a GPU implementation and leverage speedups offered by GPUs.

Ultimately, we want to theoretically study the process of automated antenna design which remains one of the unresolved problems in computational electromagnetics [10]. Namely, we want to study properties of the shape synthesis [11] and employ modern machine learning approaches which can effectively deal with huge datasets produced during the design process.

[1]Series datasets downloaded from [5].

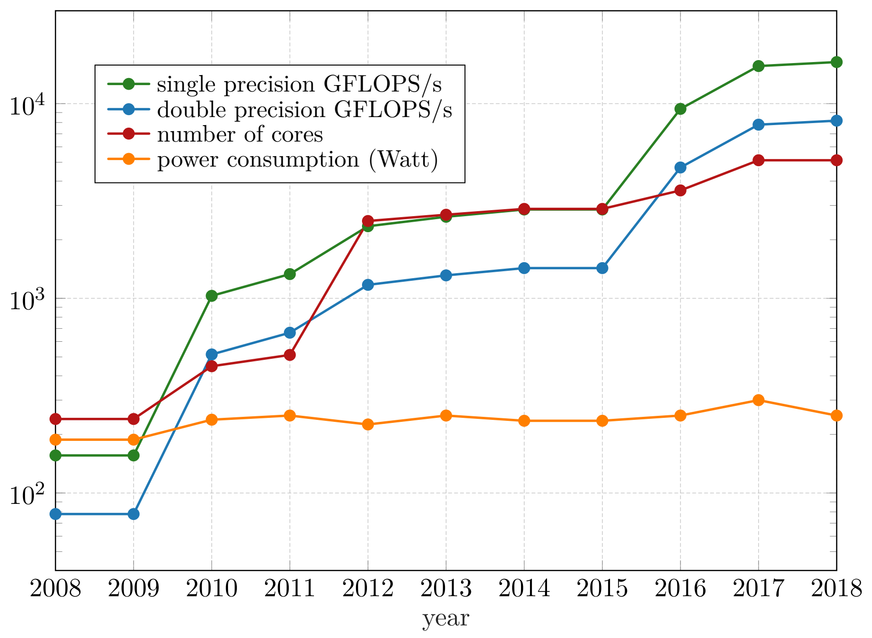


Figure 1.2: Evolution of top GPU models in Nvidia Titan series.

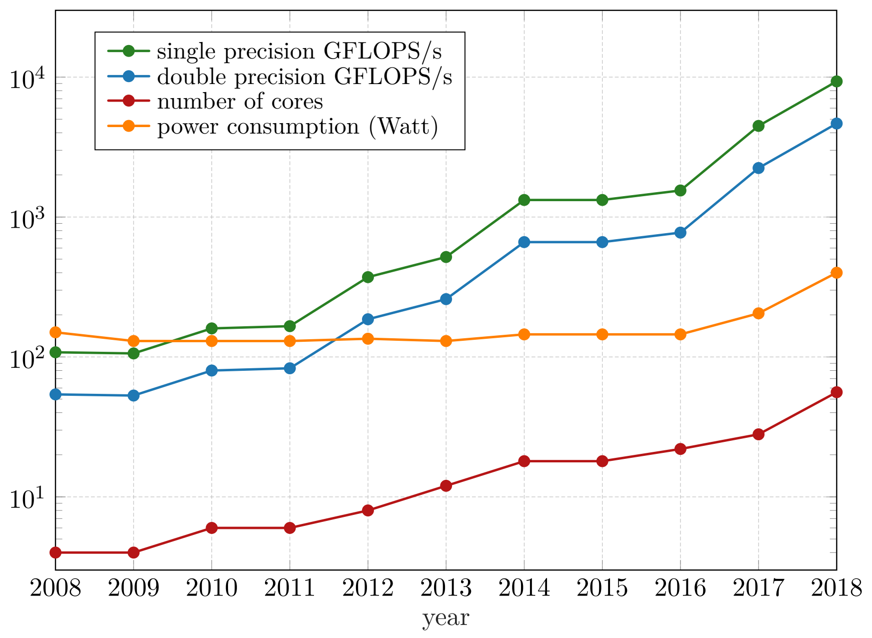


Figure 1.3: Evolution of top CPU models in Intel Xeon series.

Goals of the Thesis

The thesis is focused on the fast implementation of MoM and theoretical study of machine learning methods for antenna shape optimization. For the successful conclusion of the thesis, it is necessary to accomplish the following points:

- To review MoM in electromagnetics and present its discrete form which can be employed in connection with triangular meshes (see Chapter 2).
- To describe modern parallel architecture with emphasis on the use of graphical processing units with CUDA technology (see Chapter 3).
- To implement and evaluate performance of GPU of MoM inside AToM which can compute large (thousands by thousands) impedance matrices for analysis and synthesis of electrically small antennas (see Chapter 4).
- To describe the process of antenna shape optimization, optimized parameters and to identify computationally demanding parts (see Chapter 5).
- To propose use of machine learning algorithms in the process of the shape optimization, evaluate results and discuss potential improvements (see Chapter 6).

Chapter 2

The Method of Moments in Electromagnetics

2.1 Mathematical Formulation

Let us first define a mathematical apparatus which will be used to solve the physical problem. Our goal is to transform integro-differential equations into the algebraic form so that they can be solved by existing solvers (mostly by performing matrix inversion). This will be accomplished by using linear spaces and linear operators.

A linear operator A must satisfy the following conditions [3]

$$A(\boldsymbol{f} + \boldsymbol{g}) = A\boldsymbol{f} + A\boldsymbol{g}, \tag{2.1}$$

$$A(c\boldsymbol{f}) = cA\boldsymbol{f}, \tag{2.2}$$

where c is a constant and $\boldsymbol{f}$ and $\boldsymbol{g}$ are complex vector functions.

In addition, we will also employ the inner (symmetric) product of two complex vector functions as [3]

$$\langle \boldsymbol{f}, \boldsymbol{g} \rangle = \int\limits_V \boldsymbol{f} \cdot \boldsymbol{g} \, \mathrm{d}V, \tag{2.3}$$

which satisfies the following conditions:

$$\langle \boldsymbol{f}, \boldsymbol{g} \rangle = \langle \boldsymbol{g}, \boldsymbol{f} \rangle, \tag{2.4}$$

$$\langle \alpha\boldsymbol{f} + \beta\mathbf{g}, \boldsymbol{h} \rangle = \alpha\langle \boldsymbol{f}, \boldsymbol{h} \rangle + \beta\langle \boldsymbol{g}, \boldsymbol{h} \rangle, \tag{2.5}$$

$$\langle \boldsymbol{f}^*, \boldsymbol{f} \rangle = \begin{cases} > 0 & \text{if } \boldsymbol{f} \neq 0, \\ = 0 & \text{if } \boldsymbol{f} = 0, \end{cases} \tag{2.6}$$

where α and β are scalars and $\boldsymbol{f}^*$ is a complex conjugate of $\boldsymbol{f}$.

2.2 Method of Moments (MoM)

We show a basic scheme for solving linear system called the MoM [3] Firstly, we start with an in-homogeneous equation

$$L(\boldsymbol{f}) = \boldsymbol{g}, \tag{2.7}$$

where L is a linear operator, $\boldsymbol{g}$ is a known function, and $\boldsymbol{f}$ is an unknown function. In electromagnetics, typically, L is an integro-differential operator, $\boldsymbol{g}$ is an excitation and $\boldsymbol{f}$ is a function representing source quantities, *e.g.*, current densities. For the formulation of the MoM, we start from (2.7) and substitute the function $\boldsymbol{f}$ given as a series of functions

$$\boldsymbol{f} = \sum_n \alpha_n \boldsymbol{f}_n, \tag{2.8}$$

where α_n are yet unknown constant coefficients. From now on, we will call the functions $\boldsymbol{f}_n$ basis functions, [12]. The basis functions will approximate[1] the solution on a discrete structure. By substituting (2.8) into (2.7) and utilizing linearity of operator L we get

$$\sum_n \alpha_n L(\boldsymbol{f}_n) \approx \boldsymbol{g}. \tag{2.9}$$

Formula (2.9) forms one equation for N unknowns. Adopting a set of testing (weighting) functions $\boldsymbol{w}_m$ and performing testing via inner product (2.3) gives

$$\sum_n a_n \langle \boldsymbol{w}_m, L(\boldsymbol{f}_n)\rangle = \langle \boldsymbol{w}_m, \boldsymbol{g}\rangle, \tag{2.10}$$

which is a system of linear equations $\mathbf{Ax} = \mathbf{b}$ with

$$A_{mn} = \langle \boldsymbol{w}_m, L(\boldsymbol{f}_n)\rangle, \tag{2.11}$$
$$b_m = \langle \boldsymbol{w}_m, \boldsymbol{g}\rangle. \tag{2.12}$$

2.3 Electromagnetic Problem

Let us start with a radiation equation for the electric field intensity $\boldsymbol{E}$ which is derived for a scatterer made of Perfect Electric Conductor (PEC) and obeying the Maxwell equations [13]. The derivation of the wave equation is thoroughly described, *e.g.*, in [14]. The non-homogeneous wave equation reads [14]

$$\nabla^2 \boldsymbol{E} + k^2 \boldsymbol{E} = \mathrm{j}k Z_0 \boldsymbol{J} - \frac{Z_0}{\mathrm{j}k} \nabla(\nabla \cdot \boldsymbol{J}) - \nabla \times \boldsymbol{M}, \tag{2.13}$$

[1]For rare canonical cases like a sphere with an appropriately chosen basis functions, the MoM can be computed analytically.

where $\boldsymbol{J}$ and $\boldsymbol{M}$ are electric and magnetic current, respectively, k is the wave number, and Z_0 is the vacuum impedance. For PEC problems, the magnetic currents $\boldsymbol{M}$ on the right-hand side of (2.13) are zero [13], therefore, they are further omitted from the derivation.

We want to convert (2.13) into an integral equation. This is done using the Green's function technique which represents our system's impulse response (it can be also called transfer function). We are looking for the Green's function $G(\boldsymbol{r}, \boldsymbol{r}')$ satisfying three-dimensional scalar Helmholtz partial differential equation in a form [15]

$$\nabla^2 G(\boldsymbol{r}, \boldsymbol{r}') + k^2 G(\boldsymbol{r}, \boldsymbol{r}') = -\delta(\boldsymbol{r}, \boldsymbol{r}'). \tag{2.14}$$

With the presumed knowledge of $G(\boldsymbol{r}, \boldsymbol{r}')$ in (2.14) we can transform (2.13) into integral form for scattered field $\boldsymbol{E}_\mathrm{s}(\boldsymbol{r})$

$$\boldsymbol{E}_\mathrm{s}(\boldsymbol{r}) = -\mathrm{j}kZ_0 \int\limits_{\Omega} G(\boldsymbol{r}, \boldsymbol{r}') \left(\mathbf{1} + \frac{1}{k^2}\nabla'\nabla'\cdot\right) \boldsymbol{J}(\boldsymbol{r}')\, \mathrm{d}\boldsymbol{r}' \tag{2.15}$$

with $\mathbf{1}$ being unit dyadic. We need to transform (2.14) into spherical coordinates. Solving three-dimensional case of (2.14) (derivation omitted, can be found in [14]) leads to

$$\frac{\mathrm{d}^2(RG)}{\mathrm{d}R^2} + k^2(RG) = 0, \tag{2.16}$$

where

$$R = |\boldsymbol{r} - \boldsymbol{r}'| \tag{2.17}$$

with $\boldsymbol{r}' \in \Omega$ pointing to the source region Ω and $\boldsymbol{r}$ denoting an observation point. Then the homogeneous solution of (2.16) is

$$G(R) = A\frac{\mathrm{e}^{-\mathrm{j}kR}}{R} + B\frac{\mathrm{e}^{\mathrm{j}kR}}{R}. \tag{2.18}$$

The second term, $B\exp\{\mathrm{j}kR\}/R$, is non-physical because it grows with increasing R. Leaving it out gives

$$G(R) = A\frac{\mathrm{e}^{-\mathrm{j}kR}}{R} \tag{2.19}$$

with the last task being the determination of a constant A. Following the procedure from [14] we get $A = 1/(4\pi)$ and the resulting Green's function is

$$G(R) = \frac{\mathrm{e}^{-\mathrm{j}kR}}{4\pi R}. \tag{2.20}$$

Before we move any further, it is necessary to introduce the principle of surface equivalence [14]. Equations (2.13) and (2.15) can be used when we know the currents $\boldsymbol{J}$ (and $\boldsymbol{M}$).

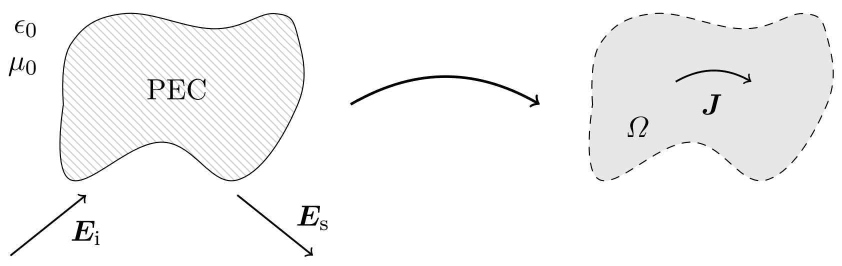


Figure 2.1: Surface equivalence principle.

The surface equivalence equates two specific scenarios, depicted in Figure 2.1. Considering a PEC obstacle with incident field $\boldsymbol{E}_\mathrm{i}$, the scattered field $\boldsymbol{E}_\mathrm{s}$ is produced. Equivalently, it can be shown [3] that the obstacle can be replaced by a surface current $\boldsymbol{J}$ which produces the same scattered field $\boldsymbol{E}_\mathrm{s}$. By applying the boundary condition valid for the PEC obstacle, *i.e.* [13],

$$\hat{n} \times (\boldsymbol{E}_\mathrm{i} + \boldsymbol{E}_\mathrm{s}) = \boldsymbol{0}, \tag{2.21}$$

and assigning the scattered field with (2.15), we receive an integral equation for the unknown current density $\boldsymbol{J}$, which can be solved via MoM for the known incident field $\boldsymbol{E}_\mathrm{i}$.

Substituting now (2.20) into (2.15), applying (2.8) for yet unknown surface currents and applying testing (2.3) with a set of testing functions being equal to the set of the basis functions $\boldsymbol{f}_n$ (*i.e.*, Galerkin method, [3]), we arrive at

$$\mathbf{ZI} = \mathbf{V}, \tag{2.22}$$

where $\mathbf{V} = [V_m]$ is a known excitation vector, defined element-wise as

$$V_m = \int\limits_{\Omega} \boldsymbol{f}_m(\boldsymbol{r}) \cdot \boldsymbol{E}_\mathrm{i} \, \mathrm{d}V \tag{2.23}$$

and $\mathbf{Z} = [Z_{mn}]$ is the impedance matrix, defined element-wise as

$$\begin{aligned} Z_{mn} = \mathrm{j}kZ_0 \int\limits_{\Omega} \boldsymbol{f}_m(\boldsymbol{r}) \cdot \int\limits_{\Omega} \boldsymbol{f}_n(\boldsymbol{r}') \, G(\boldsymbol{r}, \boldsymbol{r}') \, \mathrm{d}\boldsymbol{r}' \, \mathrm{d}\boldsymbol{r} \\ + \frac{\mathrm{j}Z_0}{k} \int\limits_{\Omega} \boldsymbol{f}_m(\boldsymbol{r}) \cdot \left(\nabla\nabla \cdot \int\limits_{\Omega} \boldsymbol{f}_n(\boldsymbol{r}') \, G(\boldsymbol{r}, \boldsymbol{r}') \, \mathrm{d}\boldsymbol{r}' \right) \, \mathrm{d}\boldsymbol{r}. \end{aligned} \tag{2.24}$$

We can see that (2.24) requires a twice differentiable basis functions. This is not very suitable for our problem and we need to rewrite the equation (derivation can be found in [14]) as

$$\begin{aligned} Z_{mn} = \mathrm{j}kZ_0 \int\limits_{\Omega} \boldsymbol{f}_m(\boldsymbol{r}) \cdot \int\limits_{\Omega} \boldsymbol{f}_n(\boldsymbol{r}') \, G(\boldsymbol{r}, \boldsymbol{r}') \, \mathrm{d}\boldsymbol{r}' \, \mathrm{d}\boldsymbol{r} \\ + \frac{\mathrm{j}Z_0}{k} \int\limits_{\Omega} \nabla \cdot \boldsymbol{f}_m(\boldsymbol{r}) \int\limits_{\Omega} \nabla' \cdot \boldsymbol{f}_n(\boldsymbol{r}') \, G(\boldsymbol{r}, \boldsymbol{r}') \, \mathrm{d}\boldsymbol{r}' \, \mathrm{d}\boldsymbol{r}. \end{aligned} \tag{2.25}$$

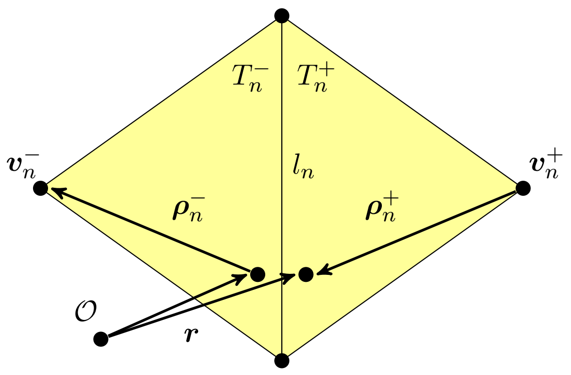


Figure 2.2: RWG basis function.

2.4 Basis Functions

Commonly used basis functions for 3D surface problems are Rao-Wilton-Glisson (RWG) triangular basis functions [16]. We use them to expand the sum in (2.8). It is assumed that an obstacle is discretized into a triangular mesh grid. The RWG functions are defined as

$$\boldsymbol{f}_n(\boldsymbol{r}) = \frac{l_n}{2A_n^+}\boldsymbol{\rho}_n^+(\boldsymbol{r}) \quad \boldsymbol{r} \text{ in } T_n^+, \tag{2.26}$$

$$\boldsymbol{f}_n(\boldsymbol{r}) = \frac{l_n}{2A_n^-}\boldsymbol{\rho}_n^-(\boldsymbol{r}) \quad \boldsymbol{r} \text{ in } T_n^-, \tag{2.27}$$

$$\boldsymbol{f}_n(\boldsymbol{r}) = 0 \quad \text{otherwise,} \tag{2.28}$$

where T_n^+ and T_n^- form a pair of triangles with a shared inner edge n; l_n is the length of the common edge n, position vector $\boldsymbol{r}$ is defined with respect to the coordinate origin $\mathcal{O}$, and vectors $\boldsymbol{\rho}_n^+(\boldsymbol{r})$ and $\boldsymbol{\rho}_n^-(\boldsymbol{r})$ are defined as

$$\boldsymbol{\rho}_n^+(\boldsymbol{r}) = \boldsymbol{v}_n^+ - \boldsymbol{r} \quad \boldsymbol{r} \text{ in } T_n^+, \tag{2.29}$$

$$\boldsymbol{\rho}_n^-(\boldsymbol{r}) = \boldsymbol{r} - \boldsymbol{v}_n^- \quad \boldsymbol{r} \text{ in } T_n^-, \tag{2.30}$$

where $\boldsymbol{v}_n^+$ and $\boldsymbol{v}_n^-$ are the free vertices of the T_n^+ and T_n^- triangles, respectively; see Figure 2.2. For further purposes, the surface divergence of $\boldsymbol{f}_n$ is [16]

$$\nabla \cdot \boldsymbol{f}_n(\boldsymbol{r}) = \frac{l_n}{A_n^+} \quad \boldsymbol{r} \text{ in } T_n^+, \tag{2.31}$$

$$\nabla \cdot \boldsymbol{f}_n(\boldsymbol{r}) = -\frac{l_n}{A_n^-} \quad \boldsymbol{r} \text{ in } T_n^-, \tag{2.32}$$

$$\nabla \cdot \boldsymbol{f}_n(\boldsymbol{r}) = 0 \quad \text{otherwise.} \tag{2.33}$$

2.5 Impedance Matrix Partitioning

The purpose of this section is to normalize the impedance matrix (2.25) with respect to the size of a scatterer and to separate its formulation into smaller blocks to be advantageously evaluated one by one. Let us start with reordering (2.25) and substituting the Green's function (2.20) yielding

$$Z_{mn} = \mathrm{j} Z_0 \int\limits_{\Omega} \int\limits_{\Omega} \left(k \boldsymbol{f}_m(\boldsymbol{r}) \cdot \boldsymbol{f}_n(\boldsymbol{r}') - \frac{1}{k} \nabla \cdot \boldsymbol{f}_m(\boldsymbol{r}) \nabla' \cdot \boldsymbol{f}_n(\boldsymbol{r}') \right) \frac{\mathrm{e}^{-\mathrm{j}kR}}{4\pi R} \, \mathrm{d}\boldsymbol{r}' \, \mathrm{d}\boldsymbol{r}. \tag{2.34}$$

The following decomposition into four matrix products will be further exploited during the implementation

$$\mathbf{Z} = \mathrm{j} Z_0 a^2 \left(ka \left(\mathbf{Z}^{\mathrm{M},k} + \mathbf{Z}^{\mathrm{M},0} \right) - \frac{1}{ka} \left(\mathbf{Z}^{\mathrm{E},k} + \mathbf{Z}^{\mathrm{E},0} \right) \right), \tag{2.35}$$

where the matrices are defined element-wise as

$$Z_{mn}^{\mathrm{M},k} = \frac{1}{a^3} \int\limits_{\Omega} \int\limits_{\Omega} \boldsymbol{f}_m(\boldsymbol{r}) \cdot \boldsymbol{f}_n(\boldsymbol{r}') \frac{\mathrm{e}^{-\mathrm{j}kR} - 1}{4\pi R} \, \mathrm{d}\boldsymbol{r}' \, \mathrm{d}\boldsymbol{r}, \tag{2.36}$$

$$Z_{mn}^{\mathrm{M},0} = \frac{1}{a^3} \int\limits_{\Omega} \int\limits_{\Omega} \boldsymbol{f}_m(\boldsymbol{r}) \cdot \boldsymbol{f}_n(\boldsymbol{r}') \frac{1}{4\pi R} \, \mathrm{d}\boldsymbol{r}' \, \mathrm{d}\boldsymbol{r}, \tag{2.37}$$

$$Z_{mn}^{\mathrm{E},k} = \frac{1}{a} \int\limits_{\Omega} \int\limits_{\Omega} \nabla \cdot \boldsymbol{f}_m(\boldsymbol{r}) \, \nabla' \cdot \boldsymbol{f}_n(\boldsymbol{r}') \frac{\mathrm{e}^{-\mathrm{j}kR} - 1}{4\pi R} \, \mathrm{d}\boldsymbol{r}' \, \mathrm{d}\boldsymbol{r}, \tag{2.38}$$

$$Z_{mn}^{\mathrm{E},0} = \frac{1}{a} \int\limits_{\Omega} \int\limits_{\Omega} \nabla \cdot \boldsymbol{f}_m(\boldsymbol{r}) \, \nabla' \cdot \boldsymbol{f}_n(\boldsymbol{r}') \frac{1}{4\pi R} \, \mathrm{d}\boldsymbol{r}' \, \mathrm{d}\boldsymbol{r}, \tag{2.39}$$

and where variable a is defined as

$$a = \max_{\boldsymbol{r}, \boldsymbol{r}' \in \Omega} = \left\{ \frac{R}{2} \right\}, \tag{2.40}$$

that represents a radius of the smallest sphere fully circumscribing the scatterer Ω. Consequently, the matrix products $\mathbf{Z}^{\mathrm{E},0}, \mathbf{Z}^{\mathrm{E},k}, \mathbf{Z}^{\mathrm{M},0}$ and $\mathbf{Z}^{\mathrm{M},k}$ are dimensionless, which mitigates the problems with the numerical errors associated with computation of very small or very big structures [17]. One can notice that the matrix impedance matrix $\mathbf{Z}$ is symmetric and mostly dense in contrast to the systems in finite element methods. This symmetry property can be further exploited in the implementation.

Considering electrically small region, say $ka < 1$, it is often useful to compute the stored energy matrix [18] which is the normalized derivative of $\mathbf{Z}$ with respect to angular frequency $\omega = kc_0$

$$\omega \frac{\partial \mathbf{Z}}{\partial \omega} = \mathrm{j} Z_0 a^2 \left(ka \left(\mathbf{Z}^{\mathrm{M},k} + \mathbf{Z}^{\mathrm{M},0} - \mathrm{j}ka \mathbf{T}^{\mathrm{M}} \right) - \frac{1}{ka} \left(\mathbf{Z}^{\mathrm{E},k} + \mathbf{Z}^{\mathrm{E},0} + \mathrm{j}ka \mathbf{T}^{\mathrm{E}} \right) \right), \tag{2.41}$$

where the matrices $\mathbf{T}^{\mathrm{M}}$ and $\mathbf{T}^{\mathrm{E}}$ are defined element-wise as

$$T_{mn}^{\mathrm{M}} = \frac{1}{a^4} \int\limits_{\Omega} \int\limits_{\Omega} \boldsymbol{f}_m(\boldsymbol{r}) \cdot \boldsymbol{f}_n(\boldsymbol{r}') \frac{\mathrm{e}^{-\mathrm{j}kR}}{4\pi} \, \mathrm{d}\boldsymbol{r}' \, \mathrm{d}\boldsymbol{r}, \tag{2.42}$$

$$T_{mn}^{\mathrm{E}} = \frac{1}{a^2} \int\limits_{\Omega} \int\limits_{\Omega} \nabla \cdot \boldsymbol{f}_m(\boldsymbol{r}) \, \nabla' \cdot \boldsymbol{f}_n(\boldsymbol{r}') \frac{\mathrm{e}^{-\mathrm{j}kR}}{4\pi} \, \mathrm{d}\boldsymbol{r}' \, \mathrm{d}\boldsymbol{r}. \tag{2.43}$$

2.6 Integration with Numerical Quadrature

Most of the engineering problems are solved in a discrete form because the analytical form is either unknown or it can be found only for canonical cases [19]. The process starts with designing the model in Computer-Aided Design (CAD) software. It allows engineers to represent models in a way which preserves mathematical description, *i.e.*, in the form of B-splines [20]. Once the model is prepared, it is converted to a set of geometric primitives compatible with chosen sets of basis and testing functions. This process is called discretization [21] for which we most commonly choose triangles (for surface objects) or tetrahedra (for volumetric objects) which can well approximate objects of various shapes. Another possibility is, *e.g.*, to use quadrilateral elements [22]. Additionally, we want the discretization to have some favorable properties in order to get optimal results from the underlying numerical solver. These properties are most commonly preserved in the form of Delaunay triangulation [23], *i.e.*, to have ideally equilateral triangles of similar sizes.

By applying M-point numerical quadrature [24], (2.25) can be transformed into

$$Z_{mn}^{XY} = \frac{\mathrm{j}Z_0}{\pi k}\chi_{m,n}^{XY}\sum_{p=1}^{M}\sum_{q=1}^{M} w_p w_q \left(\frac{k^2}{4}\boldsymbol{\rho}_m^X(\boldsymbol{r}_p)\cdot\boldsymbol{\rho}_n^Y(\boldsymbol{r}'_q) - 1\right)\frac{\mathrm{e}^{-\mathrm{j}kR_{mnpq}^{XY}}}{R_{mnpq}^{XY}}, \tag{2.44}$$

where $\boldsymbol{\rho}(\boldsymbol{r})$ are converted to simplex coordinates

$$\boldsymbol{\rho}_m^X(\boldsymbol{r}) = \lambda_1\boldsymbol{v}_{1m} + \lambda_2\boldsymbol{v}_{2m} + \lambda_3\boldsymbol{v}_{3m} + \boldsymbol{v}_m^X, \tag{2.45}$$

$$\boldsymbol{\rho}_n^Y(\boldsymbol{r}') = \lambda'_1\boldsymbol{v}_{1n} + \lambda'_{2n}\boldsymbol{v}_2 + \lambda'_3\boldsymbol{v}_{3n} + \boldsymbol{v}_n^Y, \tag{2.46}$$

with $\boldsymbol{v}_1, \boldsymbol{v}_2$ and $\boldsymbol{v}_3$ being vertex coordinates, $\boldsymbol{v}_m^X$ and $\boldsymbol{v}_n^Y$ being vertices opposite to the shared edge of triangles m and n, and where $R_{m,n,p,q}^{X,Y}$ is distance between quadrature points

$$R_{mnpq}^{XY} = \sqrt{(x_p - x_q)^2 + (y_p - y_q)^2 + (z_p - z_q)^2}, \tag{2.47}$$

with values for X and Y being substituted from Table 2.1.

X	Y	$\chi_{m,n}^{X,Y}$	$\boldsymbol{v}_m^X$	$\boldsymbol{v}_n^Y$
$-$	$-$	$l_m l_n$	$\boldsymbol{v}_m^-$	$\boldsymbol{v}_n^-$
$-$	$+$	$-l_m l_n$	$\boldsymbol{v}_m^-$	$\boldsymbol{v}_n^+$
$+$	$-$	$-l_m l_n$	$\boldsymbol{v}_m^+$	$\boldsymbol{v}_n^-$
$+$	$+$	$l_m l_n$	$\boldsymbol{v}_m^+$	$\boldsymbol{v}_n^+$

Table 2.1: Values of variables for positive and negative triangles.

For the non-singular terms, we get the following equations:

$$T_{mn}^{\mathrm{E},XY} = \frac{\chi_{m,n}^{XY}}{\pi} \sum_{p=1}^{M} \sum_{q=1}^{M} w_p w_q \mathrm{e}^{-\mathrm{j}kR_{mnpq}^{XY}}, \tag{2.48}$$

$$T_{mn}^{\mathrm{M},XY} = \frac{\chi_{m,n}^{XY}}{4\pi} \sum_{p=1}^{M} \sum_{q=1}^{M} w_p w_q \boldsymbol{\rho}_m^X(\boldsymbol{r}) \cdot \boldsymbol{\rho}_n^Y(\boldsymbol{r}') \mathrm{e}^{-\mathrm{j}kR_{mnpq}^{XY}}, \tag{2.49}$$

$$Z_{mn}^{\mathrm{E},k,XY} = \frac{\chi_{m,n}^{XY}}{\pi} \sum_{p=1}^{M} \sum_{q=1}^{M} w_p w_q \frac{\mathrm{e}^{-\mathrm{j}kR_{mnpq}^{XY}} - 1}{R_{mnpq}^{XY}}, \tag{2.50}$$

$$Z_{mn}^{\mathrm{M},k,XY} = \frac{\chi_{m,n}^{XY}}{4\pi} \sum_{p=1}^{M} \sum_{q=1}^{M} w_p w_q \boldsymbol{\rho}_m^X(\boldsymbol{r}) \cdot \boldsymbol{\rho}_n^Y(\boldsymbol{r}') \frac{\mathrm{e}^{-\mathrm{j}kR_{mnpq}^{XY}} - 1}{R_{mnpq}^{XY}}. \tag{2.51}$$

It can be seen that the majority of the terms (triangle areas, quadrature weights, and basis vectors together with radius vectors) can be precomputed. The computationally intensive part consists of the complex exponential evaluation including distances between the integration points. The complexity of evaluation increases quadratically with the number of quadrature points.

2.7 Gaussian Quadrature

Quadrature points are determined according to the symmetrical Gaussian quadrature rule presented in [25] where a number of integration points M does not grow quadratically with the quadrature order N but $M < N^2$. Both points in simplex coordinates and their respective weights are determined, *e.g.*, for quadrature rule of the second order we get three points as shown in Table 2.2. AToM implements quadrature rules up to the order 12 where all of them are tabulated.

i	λ_1	λ_2	λ_3	w
1	$\frac{2}{3}$	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{3}$
2	$\frac{1}{6}$	$\frac{2}{3}$	$\frac{1}{6}$	$\frac{1}{3}$
3	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{2}{3}$	$\frac{1}{3}$

Table 2.2: The Gaussian quadrature points and weights for the order $n = 2$.

2.8 Singularities of Near and Overlapping Terms

In the case of the near or overlapping source and testing triangles the distance R between two integration points limits to zero and the fraction on the right-hand side of (2.39) and (2.37) grows as

$$\lim_{R \to 0} \frac{1}{R} \to \infty. \tag{2.52}$$

Therefore, the solution becomes numerically unstable. The computation of the matrices $\mathbf{Z}^{\mathrm{M},0}$ and $\mathbf{Z}^{\mathrm{E},0}$ is therefore evaluated using different approach presented in [26] where inner integral is computed analytically. This treatment allows us to evaluate impedance matrices even with meshes containing overlapping triangles. This method reduces complexity of (2.39) and (2.37) to just one sum over quadrature points instead of two.

Chapter 3

Modern Parallel Architectures

3.1 GPU Architecture

Graphical processing units started as graphical processors for computer graphics. Nowadays, we use them as multipurpose processors for various types of computations. We often refer to this type of application as General-Purpose computing on Graphics Processing Units (GPGPU). Such cards frequently do not even have any graphical output. GPUs are mainly suitable for matrix computations of independent tasks which can be parallelized.

The state-of-the-art technology for GPGPU computations is CUDA. It gives access to the GPU's instruction set and parallel elements necessary for launching GPU kernels. This technology is proprietary for Nvidia graphic cards. It is designed to work with programming languages C, C++ and Fortran. Nvidia launched the first micro-architecture Tesla in 2006 with CUDA Compute Capability (CC) 1.0. The latest micro-architecture is Ampere (CC 8.0). Over time, many important features were added, namely integer atomic operations (CC 1.1), double precision operations (CC 1.3), half-precision operations (CC 5.3), double atomic addition (CC 6.0) [27]. See evolution of CC and Nvidia architectures over time in Figure 3.1.

CPU vs GPU

Many modern applications like mathematical and physical simulations utilize high level of parallelism. Even though CPU manufacturers stopped increasing frequency of cores and started adding more cores around year 2005 as seen in Figure 1.1, the demand for even higher parallelism was also increasing. The not so obvious solution was to use graphic cards which had been used primarily for graphical computations until that time. Graphical cards offered the possibility to run rather simple computations by thousands of threads in parallel.

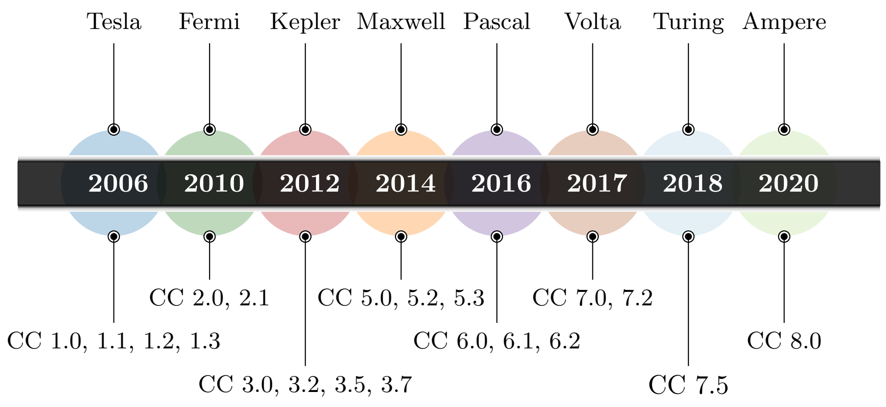
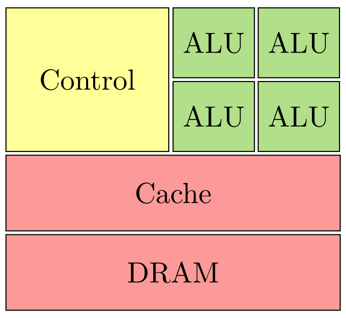


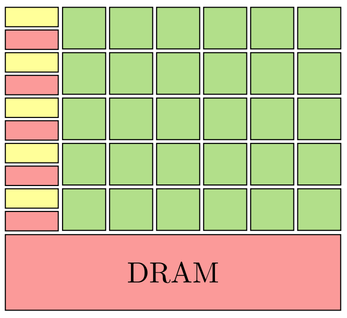
Figure 3.1: Nvidia microarchitecture timeline.

- Multi-core CPUs: Modern CPUs get more cores and wider vector lanes, *e.g.*, Intel Xeon Platinum 9282, 56 cores × 2 threads, Advanced Vector Extensions 512 (AVX-512).
- Many-core GPUs: Highly specialized cores with low control logic and high bandwidth memory, *e.g.*, Nvidia V100 with 5120 cores.

As depicted in Figure 3.2a and Figure 3.2b, GPUs have significantly more chip surface dedicated to computation called Arithmetic Logic Unit (ALU). They contain less control logic than modern CPUs because the simplified execution model does not use speculative and out-of-order execution. CPUs are optimized for low-latency access to the cache memory, GPUs are architecture tolerant to memory latency due to warp scheduling. GPUs need less cache memory due to the missing jump prediction.



(a) Schematics of a CPU chip.



(b) Schematics of a GPU chip.

Figure 3.2: Chip surface visualization of a CPU and a GPU.

3.2 CUDA

Before we start with CUDA implementation of MoM, we need to show some basic properties of this technology. This is crucial for the understanding of different programming paradigms compared to imperative programming for CPUs.

GPU Basics

- Host: the CPU and its memory.
- Device: the GPU and its memory.
- Streaming Multiprocessor (SMX): an operation unit which executes blocks of threads, it has Single Instruction Multiple Threads (SIMT) architecture.
- Threads: execution stream, many threads are executed simultaneously on a GPU.
- Kernel: a function launched on a thread.
- Warp: a minimal batch of threads launched on an SMX. All Nvidia micro architectures operate with fixed warps of size 32. Multiple warps are scheduled to minimize memory latency.
- Block: 3D configuration of threads.
- Grid: 3D configuration of blocks.

Configuration of threads and blocks in a grid can be visualized as a 3D matrix (see Figure 3.3). This allows us to easily map threads to elements of a vector or 2D/3D matrices.

We begin with an example of simple matrix addition, see Listing 3.1. The code adds two square matrices of size n in each dimension and stores the result in the third matrix (output matrix).

```
__global__
void addMatrices(double* result, double* mat1, double* mat2, ...
    int n) {
    const int x = blockDim.x * blockIdx.x + threadIdx.x;
    const int y = blockDim.y * blockIdx.y + threadIdx.y;

    if(x < n && y < n) {
        result[x*n + y] = mat1[x*n + y] + mat2[x*n + y];
    }
}
```

Listing 3.1: CUDA kernel for matrix addition.

Indexing can be performed using special CUDA variables `threadIdx` (index of a thread in a block), `blockIdx` (index of a block in the grid), `blockDim` (number of threads in a given block dimension), `gridDim` (number of blocks in a given grid dimension). All variables can be treated as 3D points by using `.x, .y, .z` structure members.

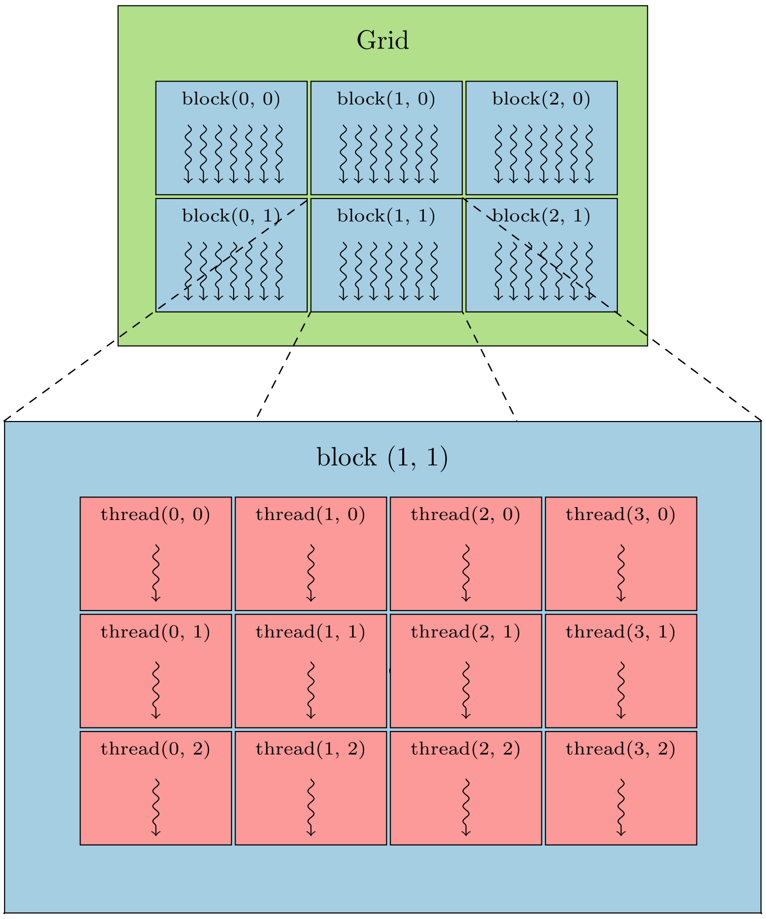


Figure 3.3: Arrangement of the blocks and the threads.

Each function has a declaration specifier. It can be either `__global__`, such a function can be called from both device and host code and must return `void`, or `__device__` which are always called from device code and can have a return value. Variable `dim3` is an integer 3-dimensional vector type that is used in kernel invocation. At least one dimension must have a specified size. Unspecified dimensions are set to one.

```
1 dim3 threadsPerBlock (64, 64) ;
2 int blockX = ceil(n / threadsPerBlock.x);
3 int blockY = ceil(n / threadsPerBlock.y);
4 dim3 numBlocks (blockX, blockY);
5 addMatrices <<<numBlocks , threadsPerBlock>>>(result, mat1, ...
      mat2, n);
```

Listing 3.2: Launch configuration of a kernel adding two matrices.

A kernel launch example is in Listing 3.2. Notice we often launch more blocks/threads than necessary. Code on line 2 in Listing 3.2 rounds the number of blocks to upper bound. This ensures that all matrix elements are covered by spawned threads. Without this treatment, some elements would not be computed when the dimension of a matrix modulo block dimension is not zero. Threads that would operate on elements outside the matrix boundary would skip the actual summation due to the condition on line 6 in Listing 3.1.

Memory

The host has a separate memory from the device. Typically, a host memory spans from tens of gigabytes up to one terabyte in modern cluster nodes. Device memory is much more limited. Consumer GPU cards have only units of gigabytes. A state-of-the-art cluster GPU Nvidia V100 has 32 GB of device memory. Kernels executed on the device can only access device memory. This means that all data must be copied from the host to the device memory before the kernel initialization. After the computation on the device is done (kernel finishes execution) data must be copied back before the host tries to access them. It imposes restrictions to host-device synchronization. Some functions need to explicitly call synchronization routine (*i.e.*, in C++ `cudaDeviceSychronize()`). This ensures all queued operations ahead the synchronization will finish before the execution can continue.

GPUs implement a similar concept of memory hierarchy as CPUs. The operational memory of a typical computer has multiple levels. The fastest and at the same time the smallest memories are directly on a CPU (registers, L caches). If the desired memory block is not found in cache, then Random Access Memory (RAM) or disk space is queried [28]. Similarly, GPUs use memory hierarchy to speed up memory access as described in [29]. The following description corresponds to the hierarchy depicted in Figure 3.4. Each SMX has a very limited number of registers, which are the fastest memory type. Data which do not fit into registers are saved in a local memory that has similar access speeds as the global memory. Shared memory sits speed-wise between registers and local memory. This memory type must be explicitly declared in kernels and allocated before kernel execution. It is two orders of magnitude faster than local and global memory. Global memory is the largest memory space with the highest latency. Last type is constant memory which is read-only inside of kernels. It is always cached because it only contains values that can not be changed inside kernels.

Shared memory is not only used for its speed but also for in-block thread synchronization. It is possible to use `__syncthreads()` barrier which ensures succeeding computations are executed after all threads in a block reach the barrier. There is no communication between streaming multiprocessors nor between different blocks. Each block is always executed on one SMX. All threads within a block can access the same shared memory. Each SMX has its shared memory.

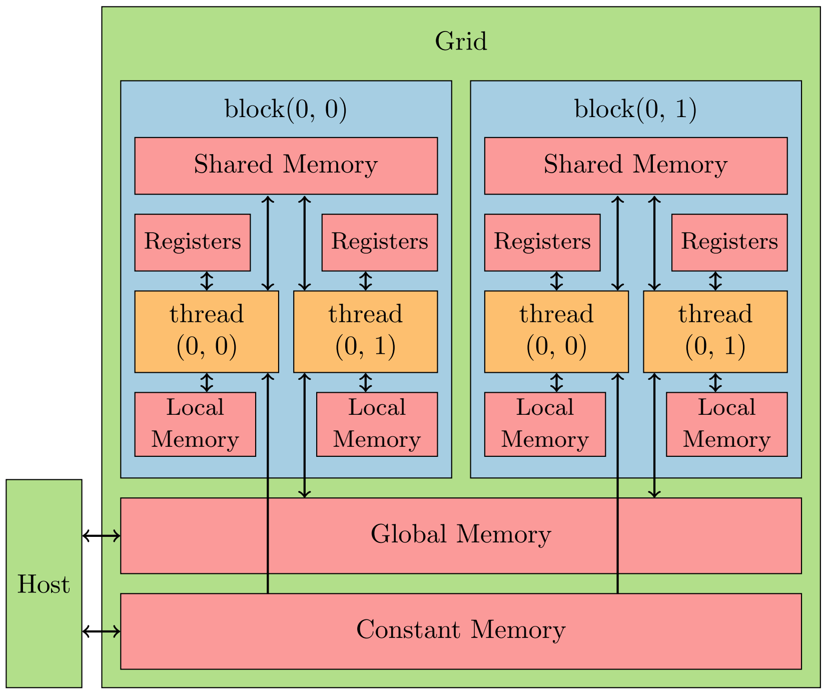


Figure 3.4: GPU memory hierarchy.

Atomic Operations

A typical processor instruction has undefined output when multiple threads write to the same memory space without proper synchronization. If this happens, *e.g.*, the memory can be modified by two write operations and the result is the value of the thread which comes last. In contrary, a series of multiple reads of a single memory space without a write is a safe operation. Atomic operations solve this problem by serializing operations of multiple threads. A read-modify-write operation is performed atomically when no other thread can access the memory space until the operation finishes. CUDA supports 32-bit or 64-bit read-modify-write operations at an address in global or shared memory. These operations are performed serially by thousands of threads and can become a bottleneck. Hardware support of floating-point atomic operations is not available until cards with CC 6.0. For cards with CC lower than 6.0, floating-point atomic operations can be implemented using `long long int` data type.

Chapter 4

GPU Implementation of Method of Moments in MATLAB

We now want to implement MoM derived in Chapter 2 in MATLAB. We want to write most of the code in CUDA which should allow us to evaluate large structures in reasonable time. The code must be compatible with AToM. We review three approaches of GPU programming in MATLAB (see Section 4.1), describe MoM implementation in CUDA (see Section 4.2) and evaluate its performance and correctness (see Sections 4.3 and 4.4).

4.1 GPU Code in MATLAB

Naive GPU Code

MATLAB [9] has a list of GPU-ready functions [30]. These functions can be used with `gpuArray` object which stores data in GPU's memory. Once `gpuArray` is passed to a GPU-ready function, the code is performed on a GPU. The memory works in the same manner as described in Section 3.2, where the workspace memory is the host memory. Basic manipulation with GPU matrices in MATLAB is shown in Listing 4.1.

```
N = 100;
A = eye(N);
% Transfers data from MATLAB workspace to GPU memory.
gpuA = gpuArray(A);

% Directly creates matrix in GPU memory.
gpuB = eye(N, 'gpuArray');

% Transfers data from GPU memory to MATLAB workspace.
B = gather(B);
```

Listing 4.1: Allocation and manipulation with `gpuArray` in MATLAB.

```
% CPU matrix multiplication
A = rand(n,n);
B = rand(n,n);
C = A*B;

% GPU matrix multiplication
A = rand(n,n, 'gpuArray');
B = rand(n,n, 'gpuArray');
C = A*B;
```

Listing 4.2: CPU vs GPU matrix multiplication in MATLAB.

Benchmarks of GPU-Ready Functions

We will demonstrate the speedups of a CPU implementation in MATLAB with respect to their GPU equivalents for some basic functions. The tests were conducted using Intel Xeon Gold 6150 processor and Nvidia Tesla V100. All tests are run without *parpool* which allows us to run CPU code in more threads on multiple cores.

We start with a simple algorithm of matrix multiplication in Listing 4.2.

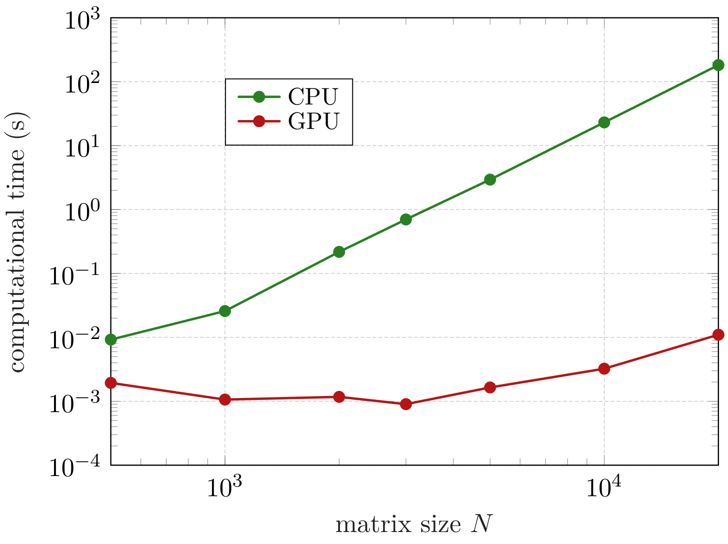


Figure 4.1: Computation time of matrix multiplication on a CPU and a GPU.

We can see from Figure 4.1 that the graphic card is not enough saturated for smaller matrices. On the other hand, CPU computation time grows according to multiplication algorithm complexity which is $\mathcal{O}(N^3)$ where N is the size of a matrix.

Similarly, we can evaluate the performance of matrix inversion and backslash operator (see Listing 4.3) which will be later used for evaluation of the excitation vector $\mathbf{V}$. The performance results of these two operations are in Figure 4.2.

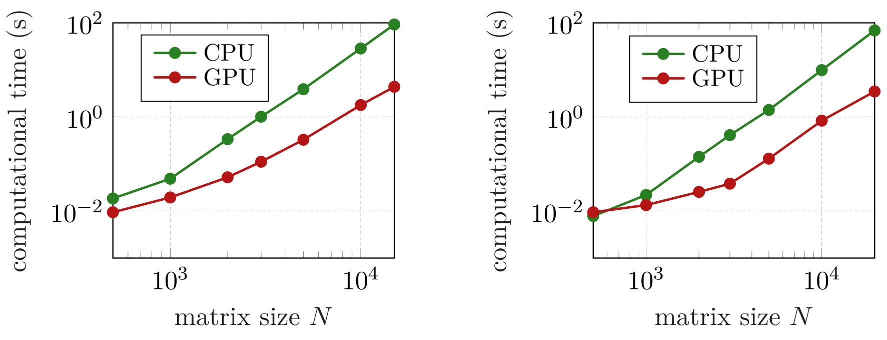
```
% CPU inversion
A = rand(n, n);
B = inv(A);

% GPU inversion
A = rand(n, n, 'gpuArray');
B = inv(A);

%%
% CPU solving system of linear equations
b = rand(n, 1);
A = rand(n, n);
x = A\b;

% GPU solving system of linear equations
b = rand(n, 1, 'gpuArray');
A = rand(n, n, 'gpuArray');
x = A\b;
```

Listing 4.3: CPU vs GPU matrix inversion and solving system of linear equations in MATLAB.



(a) Matrix inversion. **(b)** Solving system of linear equations.

Figure 4.2: Computation time comparison of matrix inversion and solving system of linear equations in MATLAB.

Element-wise Operations

The second approach which can be leveraged in MATLAB is closer to the native CUDA code. The executed code is the same for each matrix element in most of the cases. We can separate our MATLAB code into a helper function which encapsulates instructions performed on GPU. Once called inside the `arrayfun` [31] function with a `gpuArray` parameter, the helper function is compiled into native code. We demonstrate that with a basic linear algebra operation *saxpy* which computes a vector multiplied by a constant plus a vector (see Listing 4.4). This technique still does not involve any CUDA programming knowledge but performs better than the naive version as seen in Figure 4.3.

```
function result = saxpy(x, y, a)
   result = a*x+y;
end

% Create input matrices
N = 1000;
x = gpuArray.randi(100, N);
y = gpuArray.randi(100, N);
a = gpuArray.randi(100, N);

% Run naive MATLAB version
resultNaive = a*x+y;

% Run saxpy function
resultFunction = arrayfun(@saxpy, a, x, y);
```

Listing 4.4: *Saxpy* GPU code using naive implementation and compiled function.

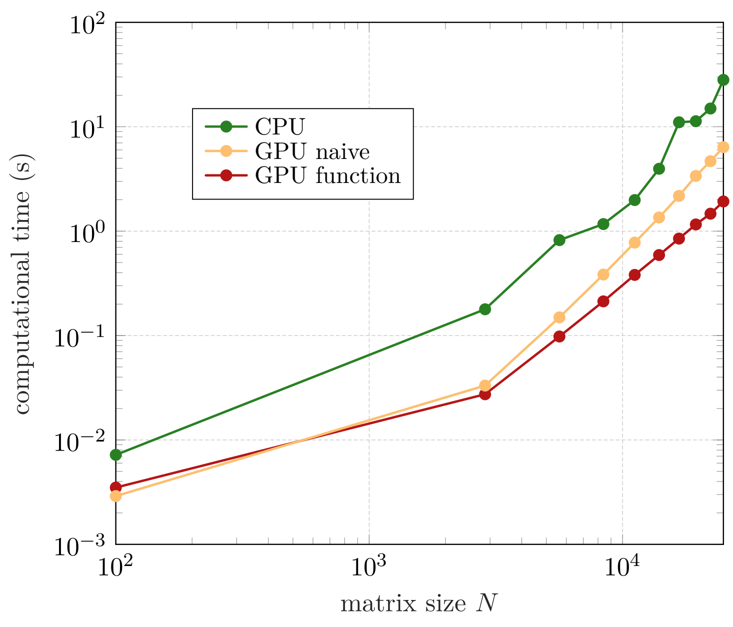


Figure 4.3: Computation time of *saxpy* function from Listing 4.4.

Native CUDA Code

It is also possible to run code that was written using CUDA directly from MATLAB. The native code must be compiled using *nvcc* compiler into a Parallel Thread Execution (PTX) file. This file contains pseudo assembly code that can run on a data-parallel computing device (abstraction of a GPU).

Firstly, we need to create a CUDA kernel object. This is done using a PTX file and the original CUDA source file. Optionally, we can spec-

```
N = 1000;
A = rand(N, N, 'gpuArray');
B = rand(N, N, 'gpuArray');
result = zeros(N, N, 'gpuArray');

kernel = parallel.gpu.CUDAKernel('addMatrices.ptx',
'addMatrices.cu', 'addMatrices');

kernelSize = floor(sqrt(kernel.MaxThreadsPerBlock));
kernel.ThreadBlockSize = [kernelSize, kernelSize, 1];
kernel.GridSize = [ceil(N/kernelSize), ceil(N/kernelSize)];

[result] = feval(kernel, result, A, B, N);

result = gather(result);
```

Listing 4.5: Creating a CUDA kernel from example in Listing 3.1.

ify an entry point. Each kernel has two launch configuration properties – `ThreadBlockSize` and `GridSize`. `ThreadBlockSize` must be set with respect to kernel properties of a GPU device – `MaxThreadsPerBlock` and `MaxThreadBlockSize`. `GridSize` is computed for our problem domain given `ThreadBlockSize`.

The kernel is launched by calling the `feval` [32] function. All kernel parameters must be `gpuArray` objects. Finally, we collect output data from GPU by calling `gather` [33] function which implicitly blocks MATLAB's execution until the kernel finishes. See Listing 4.5. CUDA kernels in MATLAB can only have floating-point, integer, boolean and character type parameters [34]. No user-defined types and structures are allowed compared to CUDA executed from C/C++ code.

4.2 Implementation Details of Method of Moments

We can see from equations (2.48), (2.49), (2.50), (2.51) in Section 2.6 that every element of all electric and magnetic sub-parts needed for assembly of impedance matrix $\mathbf{Z}$ are independent thus they are suitable for parallel GPU implementation. We decided to compute each matrix by a single CUDA kernel. Since all these matrices are symmetric thanks to the Galerkin method, only the upper/lower triangular part is required. The full matrices are completed after the evaluation.

Non-static matrices $\mathbf{T}^{\mathrm{E}}$, $\mathbf{T}^{\mathrm{M}}$, $\mathbf{Z}^{\mathrm{E},k}$, and $\mathbf{Z}^{\mathrm{M},k}$ are complex due to the imaginary unit in the exponential. C++ does not have any explicit support for complex numbers, therefore, we need to use a different approach. Real and imaginary parts are separated into respective matrices and combined into one complex matrix directly in MATLAB which has explicit complex number support. The complex exponential is evaluated using Euler's formula.

Besides, each kernel is written as a template. Every kernel is then compiled for single precision and double precision representation. The user can choose precision for the computation and the respective entry point is then chosen during runtime.

Basis vectors in simplex coordinates, quadrature point weights and triangle areas are precomputed in advance in the preprocessing part, and then reused multiple times. Each element of a decomposed impedance matrix $\mathbf{Z}$ is computed from a variable number of integration points based on the order of the Gaussian quadrature for triangles [25]. AToM supports table values of numerical quadrature up to order 12. This means each element can be computed as the sum of the maximum of 4356 elements (quadrature of order 12 uses 33 points where every combination between neighboring triangles is used four times). It leads to a parallel reduction problem where we want to sum all vector/matrix elements into one scalar value. This problem and its effective implementation were well explained in [35]. Such a technique involves warp unwrapping, usage of shared memory and avoiding bank conflicts in shared memory. Unfortunately, computation of impedance matrix does not have so well structured data alignment for this type of treatment. We implemented a parallel reduction with the above-mentioned improvements for quadrature orders three and eight, where we have 4 and 16 quadrature points, respectively. This would work only for block sizes of 16×16 and 32×32, but unrolling warps in code for specific quadrature extends the binary size of a kernel, thus some kernels hit the limit for registers. When this happens, the maximum number of threads which can be executed in a block decreases. This behavior is not well documented and cannot be easily predicted. It may also differ GPU by GPU. Without a correct block size, we can not assume the in-block alignment of all summed elements. We decided to go just with atomic operations, which seem to be comparably fast for this implementation. Besides, using atomic operations is simpler and easier to read. Furthermore, the kernel memory requirements are lower, *e.g.*, on Nvidia V100 some double-precision kernels increased cap for the maximum number of threads in a block back to 1024, which is also the maximum for this specific card.

Another key aspect of High-Performance Computing (HPC) GPU code is the handling of memory consumption on GPU. Typically, modern cards have either 8 GB or 16 GB of global memory. The state of the art HPC Nvidia GPUs have up to 32 GB of global memory (with larger memories yet to come together with Ampere architecture). Even for a simple operation like matrix addition $\mathbf{C} = \mathbf{A} + \mathbf{B}$, we need to allocate three times the resulting matrix memory (allocate and copy memory from host to device for matrices $\mathbf{A}$ and $\mathbf{B}$, allocate device memory for $\mathbf{C}$). This may be very limiting for large scale problems. We need to either decompose the problem or in the worst case solve the problem in RAM on a CPU. This is the case for matrix inversion on GPU. We use MATLAB's backslash operator for linear equations solution, *i.e.*, direct inversion is employed, expecting that the size of the impedance matrix will be maximally in thousands by thousands. An inversion of very large impedance matrix $\mathbf{Z}$ may in this case hit the memory limit. When this happens, the computation is finished on CPU.

4.3 Evaluation of the Impedance Matrix – Benchmarks

CPU benchmarks were run on Intel Core i7-8750H processor with base frequency 2.20 GHz and maximum turbo frequency 4.10 GHz [36]. All GPU benchmarks were tested on Nvidia Tesla V100 with 32 GB of device memory [37] on the cluster of Research Center for Informatics (RCI) [38].

RCI cluster is HPC infrastructure. It consists of 33 computing nodes where 20 nodes are CPU-only with 24 physical cores each. 12 nodes have 36 physical CPU cores each as well as additional 4 Nvidia Tesla V100 connected with NVLink. Each of these nodes has 384 GB RAM. A node n33 is CPU only and offers up to 192 physical cores with 1536 GB RAM and is suitable for large-scale data structures. In total, 72 CPUs with 864 physical cores and 48 Nvidia V100 GPUs are available with 13824 GB of RAM. Data storage up to 160 TB is utilized.

The evaluation of matrices $\mathbf{T}^{\mathrm{E}}$, $\mathbf{T}^{\mathrm{M}}$, $\mathbf{Z}^{\mathrm{E},k}$, and $\mathbf{Z}^{\mathrm{M},k}$ was measured separately on a CPU opposing to $\mathbf{Z}^{\mathrm{E},0}$ and $\mathbf{Z}^{\mathrm{M},0}$ which share a significant part of the CPU code. The GPU version implements computation of $\mathbf{Z}^{\mathrm{E},0}$ and $\mathbf{Z}^{\mathrm{M},0}$ separately which makes it possible to compute just one of these matrices at the time. Preprocessing contains all data preparation (quadrature weights, triangle areas, and tranformation of basis vectors into simplex coordinates)for impedance matrix computation from triangular mesh discretization. Benchmarks were conducted using an increasing number of basis functions for quadrature orders one (point matching technique [12] utilizing center point of each triangle only) and four.

Single Core Double Precision Performance

We start with the original CPU MoM code written in MATLAB. It evaluates impedance matrix column-wise to take advantage of MATLAB's instruction vectorization. We can see from Figures 4.4 and 4.5 that the algorithm is dominated by the computation of dynamic electric and magnetic sub-products of impedance matrix $\mathbf{Z}$.

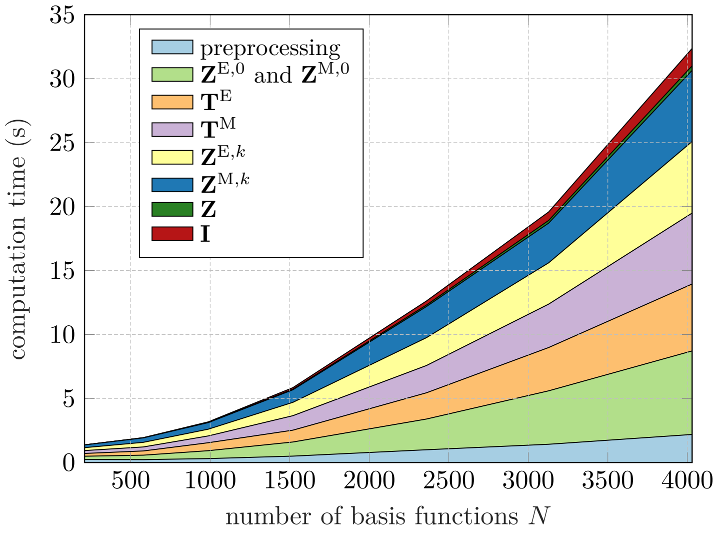


Figure 4.4: The MoM implementation on a CPU with numerical quadrature of the order one.

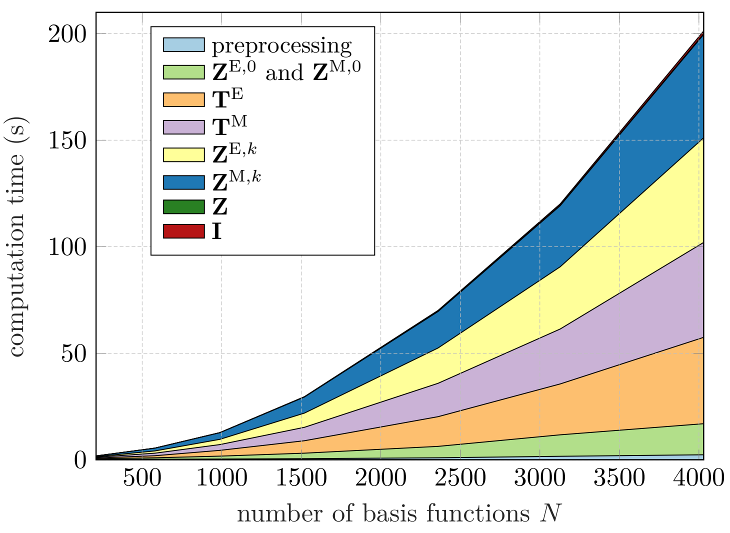


Figure 4.5: The MoM implementation on a CPU with numerical quadrature of the order four.

This is even more visible when we increase the quadrature order where preprocessing, computation of $\mathbf{Z}^{\mathrm{E},0}$ and $\mathbf{Z}^{\mathrm{M},0}$ are negligible. It is mainly due to the ineffective use of memory when the number of quadrature points is increased. The CPU implementation scales approximately quadratically with the increasing number of quadrature points.

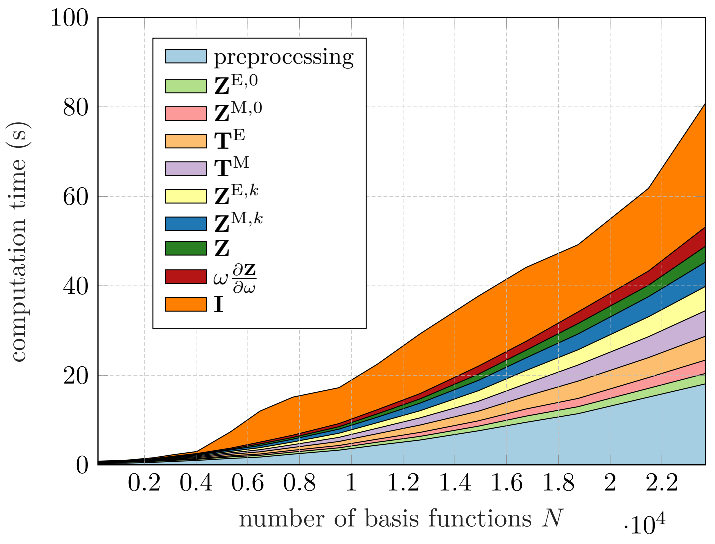


Figure 4.6: The MoM implementation on a GPU with numerical quadrature of the order one.

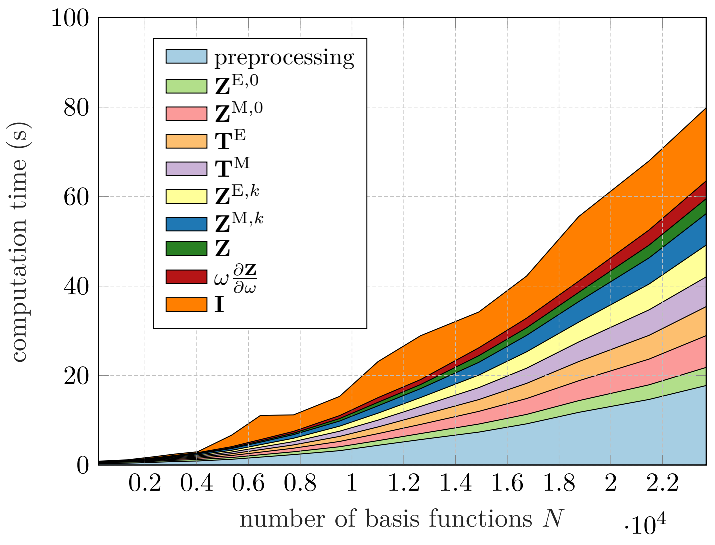


Figure 4.7: The MoM implementation on a GPU with numerical quadrature of the order four.

In comparison to the CPU implementation, preprocessing in the GPU version takes significant amount of computation time because the code is shared with the CPU version. This part of the code prepares data for GPU computation which are later copied to a device memory. The GPU code shows no significant slowdown when the quadrature order is increased from one to four. This can be explained by the massively parallel architecture of a GPU which requires often millions of threads executed at once to fully saturate the card. Every quadrature point is computed on a single thread. The relative speedup of the GPU implementation as compared to the CPU implementation is studied in Figures 4.8 and 4.9. Only the parts that were

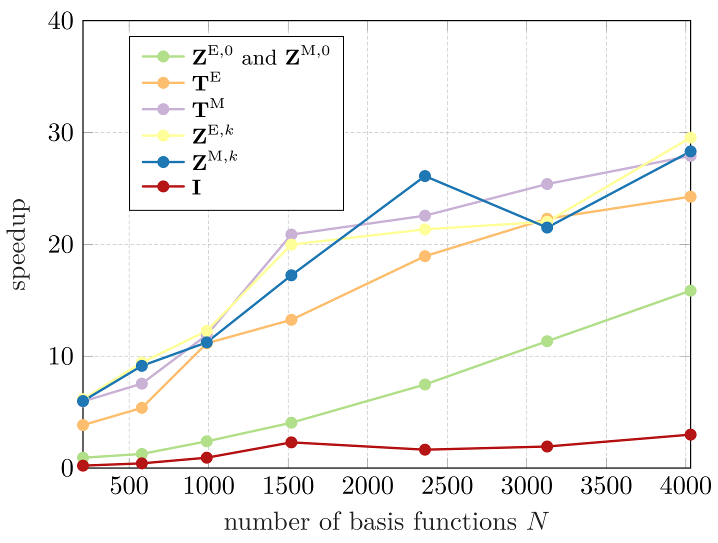


Figure 4.8: Speedup of the GPU implementation as compared to the CPU implementation using numerical quadrature of order one.

offloaded to GPU from CPU are visualized. The GPU is sufficiently saturated around 3000 basis functions. The number of operations executed during the evaluation of static matrices $\mathbf{Z}^{\mathrm{E},0}$ and $\mathbf{Z}^{\mathrm{M},0}$ depends on the number n of quadrature points in a triangle by $\mathcal{O}(n)$ (compared to matrices $\mathbf{T}^{\mathrm{E}}$, $\mathbf{T}^{\mathrm{M}}$, $\mathbf{Z}^{\mathrm{E},k}$, and $\mathbf{Z}^{\mathrm{M},k}$ where the count of operations grows with $\mathcal{O}(n^2)$). This explains why the speedup is lower compared to non-singular matrices (one order of magnitude compared to two orders of magnitude in Figure 4.9).

The vector of expansion coefficients $\mathbf{I}$ is the only part of the code that leverages built-in MATLAB GPU function (namely backslash operator) and the speedup is much lower than other parts that are written in CUDA. This is mainly caused by the fact that we need to solve the system of linear equations which is usually computationally slow, $\mathcal{O}(N^3)$, and memory demanding, $\mathcal{O}(N^2)$, for dense matrices. It is also hard to predict a real memory consumption for this algorithm because MATLAB can switch between multiple implementations based on the properties of the matrix on the input. This part of the code may fail to compute on a GPU for large matrices due to limited memory. If it does, the system is computed on a CPU as a fallback.

Single vs Double Precision Performance on a GPU

As the code is written as a template, we can simply call its single precision version by changing one parameter in MATLAB. From Figure 4.10, we can see that speedups of the kernels $\mathbf{Z}^{\mathrm{E},0}$ and $\mathbf{Z}^{\mathrm{M},0}$ converge close to two. This is what we ideally want to achieve because Nvidia V100 has twice higher single precision performance than double performance [37].

The kernels for $\mathbf{T}^{\mathrm{E}}$, $\mathbf{T}^{\mathrm{M}}$, $\mathbf{Z}^{\mathrm{E},k}$, and $\mathbf{Z}^{\mathrm{M},k}$ matrices have around the same single precision performance as the double precision (speedup oscillates around

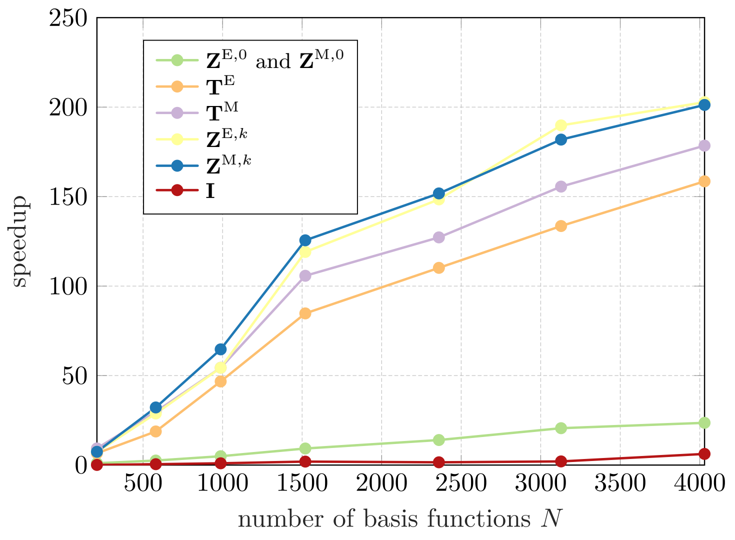


Figure 4.9: Speedup of the GPU implementation as compared to the CPU implementation using numerical quadrature of order four.

one), which is caused by the used *atomic add*. This makes an instruction to serialize and no speedup is achieved. By contrast, no atomic instruction were used in $\mathbf{Z}^{\mathrm{E},0}$ and $\mathbf{Z}^{\mathrm{M},0}$ kernels.

The evaluation of the current vector $\mathbf{I}$ involves backslash operator which is MATLAB's built-in function. Its implementation is proprietary and cannot be further optimized. The speedup of a vector $\mathbf{I}$ computation, depicted in Figure 4.8 jumps up and down very likely due to underlying implementation which uses various heuristics to choose a suitable algorithm for a given matrix.

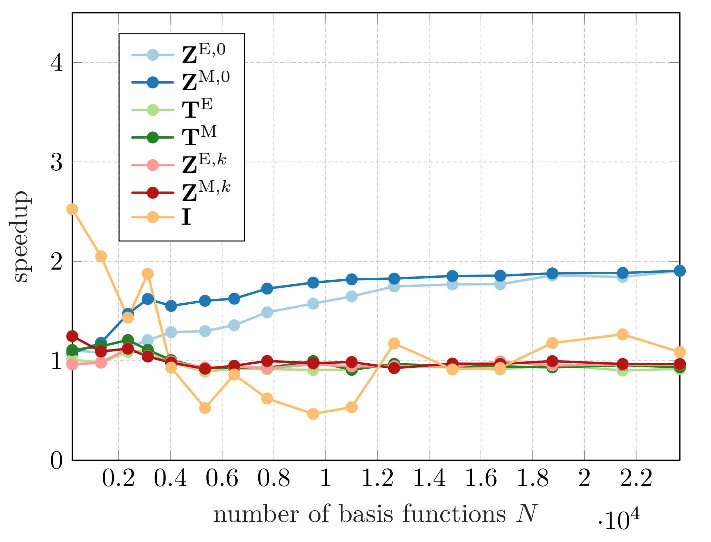


Figure 4.10: Relative speedup of the single precision computation as compared to the double precision. The used numerical quadrature is of the fourth order.

CPU	f_{CPU} (GHz)	N_{CPU}	N_{ins}
i7-4930K	3.4	6	4
Xeon Gold 6150	2.7	18	8

Table 4.1: Specification of used CPUs.

4.4 Comparison with Commercial Software

Comparison with AToM's CPU code in the previous section may be too favorable for the GPU code if CPU implementation is slow by design. Therefore, we also provide comparison with a commercial software which fully implements MoM.

In order to use the same triangularized model for all the computations, the software Feko [39] was chosen to verify the validity of the GPU code and to compare the computational times. Unfortunately, due to limited licensing, we were unable to perform the simulation on the same machine both for GPU implementation in MATLAB and for Feko evaluation. Thus, we decided to relate results to peak performance of a given processing unit. We measured three cases – Feko on a CPU, AToM on a CPU, and a GPU.

Feko version 2019.3-660 was run on a machine with 6 core Intel i7-4930K CPU with 32 GB of RAM memory, running Windows 7, SP1. The AToM's CPU implementation was run on a RCI cluster node with 18 core Intel Xeon Gold 6150 with 384 GB RAM, running CentOS Linux 7. It was restricted only to 6 cores and 32 GB RAM. The AToM's GPU code used a single Nvidia V100 with 32 GB of device memory. The version of MATLAB was 2019b.

Peak performance P of a CPU is computed according to (4.1) with values in Table 4.1.

$$P = f_{\text{CPU}} N_{\text{CPU}} N_{\text{ins}}, \tag{4.1}$$

where f_{CPU} is processor's frequency in GHz, N_{CPU} is number of physical cores and N_{ins} is number of instructions per clock cycle.

The example used for the benchmark consists of a thin-strip dipole oriented above a finite ground plane. Both objects are made of PEC, the dimensions of the plane are 60×60 mm, dimensions of the dipole are 1×42.5 mm. The plane is confined with x-y plane. The dipole is at height 10 mm above it, see Figure 4.11. Radar cross section (RCS) was evaluated to check the validity of the code. The PEC structure was excited by a plane wave of perpendicular incidence angle with polarization alongside the dipole. The computation consisted of the impedance matrix $\mathbf{Z}$ assembly and the solution of $\mathbf{I} = \mathbf{Z}^{-1}\mathbf{V}$. It was repeated for 201 distinct frequencies. Because Feko uses dynamic quadrature inside its computation, we decided to fix quadrature order to five for benchmarks in AToM. Benchmark results are in Table 4.2 with relative time being number of seconds to one GFlops of the maximum chip performance.

[1]limited to 6 cores

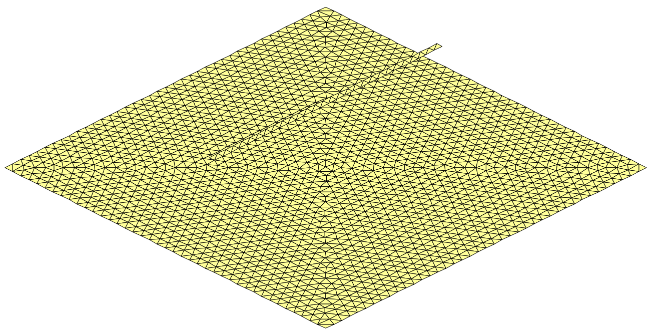


Figure 4.11: A PEC dipole 1×42.5 mm, placed 10 mm above a 60×60 mm PEC plate.

software	performance P (GFlops)	MoM run-time (s)	relative time
Feko	$8.64 \cdot 10^{1}$	$1.75 \cdot 10^{4}$	$2.03 \cdot 10^{2}$
AToM CPU[1]	$1.30 \cdot 10^{2}$	$2.79 \cdot 10^{4}$	$2.15 \cdot 10^{2}$
AToM GPU	$7.80 \cdot 10^{3}$	$1.03 \cdot 10^{3}$	$1.32 \cdot 10^{-1}$

Table 4.2: Comparison of the selected MoM implementations. The benchmark was run with a structure discretized into 3704 triangles (5447 unknowns) and depicted in Figure 4.11. The frequency range spans from 2 GHz to 4 GHz with the step 0.01 GHz.

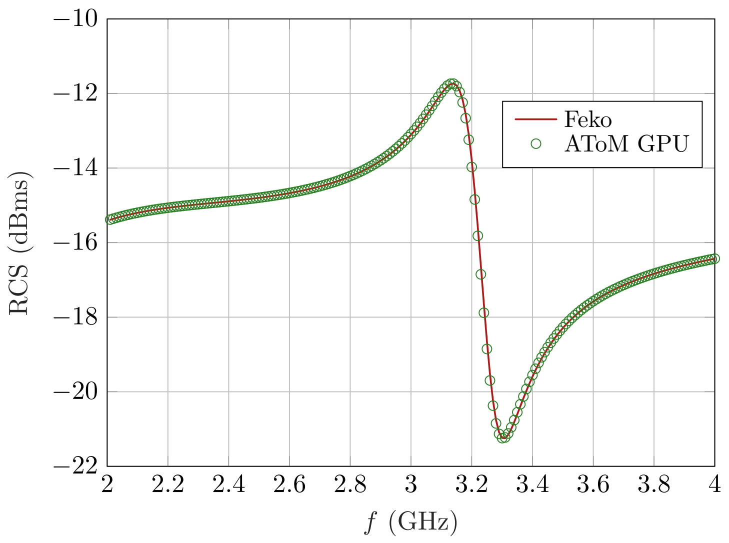


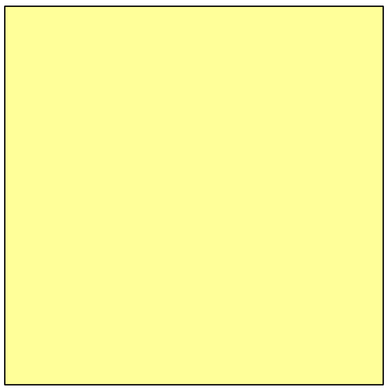
Figure 4.12: Comparison of RCS results for the model depicted in Figure 4.11 and illuminated by a plane wave of perpendicular incidence angle and polarization along the dipole.

Chapter 5

Shape Optimization

A common engineering problem is to find a structure with given properties. This optimization problem is called synthesis [40], an opposite procedure to the analysis. Such tasks may vary from finding thickness of materials to finding optimal shape which can withstand a given load [41]. These problems are usually solved numerically rather than analytically, being encumbered with non-uniqueness, instability [40], and Non-Polynomial (NP)-hardness [42].

Let us start here with a discretization of a region Ω, see Figure 5.1b, and consider it being fixed for the entire optimization. Since the discretized domain allows an exponential number of configurations of enabled/disabled degrees of freedom, we are potentially dealing with an NP-hard problem. It is not feasible to solve this efficiently, *i.e.*, in polynomial time [42]. Rather than finding an optimal solution, we usually settle with a near-optimal solutions and try to find them as fast as possible with as low fitness function value as possible (in the case of a minimization problem).


(a) A domain Ω.

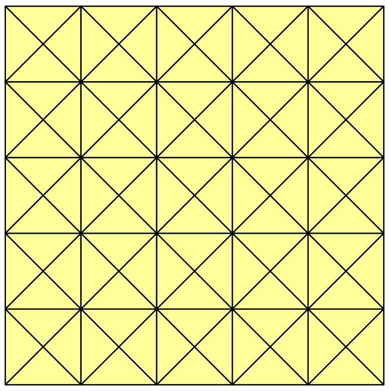

(b) A triangular discretization of the domain $\Omega_{\mathcal{T}}$.

Figure 5.1: A domain Ω and its triangular discretization $\Omega_{\mathcal{T}}$.

5.1 Potential Solutions to the Shape Synthesis

A rigorous definition of the optimization problem can be represented as an integer linear programming problem

$$\begin{aligned} \underset{\mathbf{g}}{\text{minimize}} \quad & \mathbf{I}^{\mathrm{H}}\mathbf{A}(\mathbf{g})\mathbf{I} \\ \text{subject to} \quad & \mathbf{I}^{\mathrm{H}}\mathbf{B}_i(\mathbf{g})\mathbf{I} = p_i \\ & \mathbf{I}^{\mathrm{H}}\mathbf{B}_j(\mathbf{g})\mathbf{I} \leq p_j \\ & \mathbf{Z}(\mathbf{g})\mathbf{I} = \mathbf{V} \\ & \mathbf{g} \in \{0,1\}^N, \end{aligned} \tag{5.1}$$

where $\mathbf{A}$ defines the matrix operator yielding the optimized metric, matrices $\mathbf{B}_i$ and $\mathbf{B}_j$ define the equality and inequality constraints, and $\mathbf{g}$ is a binary representation of the structure (degrees of freedom are either disabled or enabled).

In order to find a solution to (5.1), we can use approximation algorithms that work with graph representations or can try to solve linear programs directly [43]. Furthermore, we often use local or global optimization algorithms. Global algorithms had boom at the turn of the millennium with nature-inspired heuristic algorithms [44]. Among the others, the most prominent class is formed by genetic algorithms [45] which use gene recombination and mutation to achieve a better solution (*e.g.*, Non-dominated Sorting Genetic Algorithm II (NSGA-II) [46]) and swarm optimization which exploits multiple agents which concurrently work towards improving the solution by sharing community information (*e.g.*, Ant-colony optimization [47] or particle swarm optimization [48]). Local techniques involve gradient methods which continuously improve a solution. While they converge rapidly to the local minimum, they often get stuck there and cannot escape it. The adjoint formulation of topology optimization [49] is a good example of such a gradient-based approach.

5.2 Shape Optimization of Antennas

We begin with the domain $\varOmega$, see Figure 5.1a, which is discretized into the set $\mathcal{T}$ of triangles t_i, see Figure 5.1b, which form discretized domain

$$\varOmega_{\mathcal{T}} = \bigcup_{i=1}^{T} t_i, \tag{5.2}$$

with $T = |\mathcal{T}|$ being the number of the triangles in the set $\mathcal{T}$. This triangular discretization is afterwards utilized for introduction of RWG basis functions $\boldsymbol{f}_n(\boldsymbol{r})$, see Section 2.4. The set of basis functions delimits the available degrees of freedom for the memetic algorithm introduced in [11] and used in this work. Its working principle is recapitulated in this section.

Enabled and disabled basis functions are represented in a vector **g** where logical 1 represents enabled and logical 0 disabled respective basis function (see Figure 5.2).

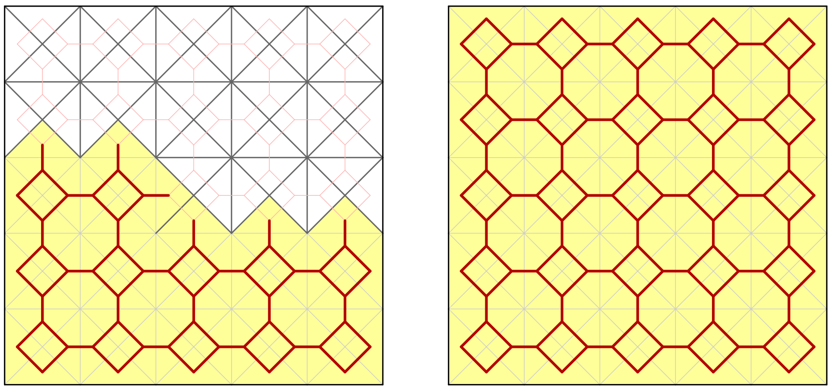

(a) A domain with half of the basis functions enabled. **(b)** A domain with all basis functions enabled.

Figure 5.2: A discretized domain $\Omega_{\mathcal{T}}$ with basis functions.

The same structure can be instead represented via the vector of triangles **t**. It is also a vector of logical zeros and ones, but this time, it marks enabled and disabled triangles instead of basis functions. A mapping from basis functions (**g**) to triangles (**t**) is, generally, unique. The opposite is, however, not the case (the same metallization given by a vector **t** can be formed by different vectors **g**). The mapping matrix between basis functions and triangles is an incidence matrix between the nodes and edges of a Voronoi graph defined for a Delaunay discretization $\Omega_{\mathcal{T}}$.

The topology sensitivity algorithm implemented in AToM involves inversion-free modifications of the impedance matrix [50], based on so-called exact reanalysis procedure [51]. This allows us to follow the direction of the best local improvement and make a step to a neighboring vector of basis functions (one different bit). Inversion-free evaluation of impedance matrix, available thanks to the Shermann-Morisson-Woodbury identity [52], is by orders of magnitude faster than the inversion itself.

The algorithm, depicted in Figure 5.3, starts with a domain Ω discretized into $\mathcal{T}$ where the basis functions are assigned to the inner edges. The impedance matrix $\mathbf{Z}$ is then computed. Since the discretization is fixed during the optimization, the impedance matrix $\mathbf{Z}$ of the initial object (bounding box) fully describing all sub-structures remains unchanged. The next step is the evaluation of all matrices required for the computation of the fitness function ($\mathbf{A}$, $\{\mathbf{B}_i\}$, $\{\mathbf{B}_j\}$). Those will be evaluated only once and their definition for particular optimization problems is described later on. Finally, the optimization procedure combining so-called global and local steps (cycles) starts.

The global cycle repeats until a given number of repetitions N is reached. In every iteration of the global cycle, a new vector of basis functions $\mathbf{g}_i$ is

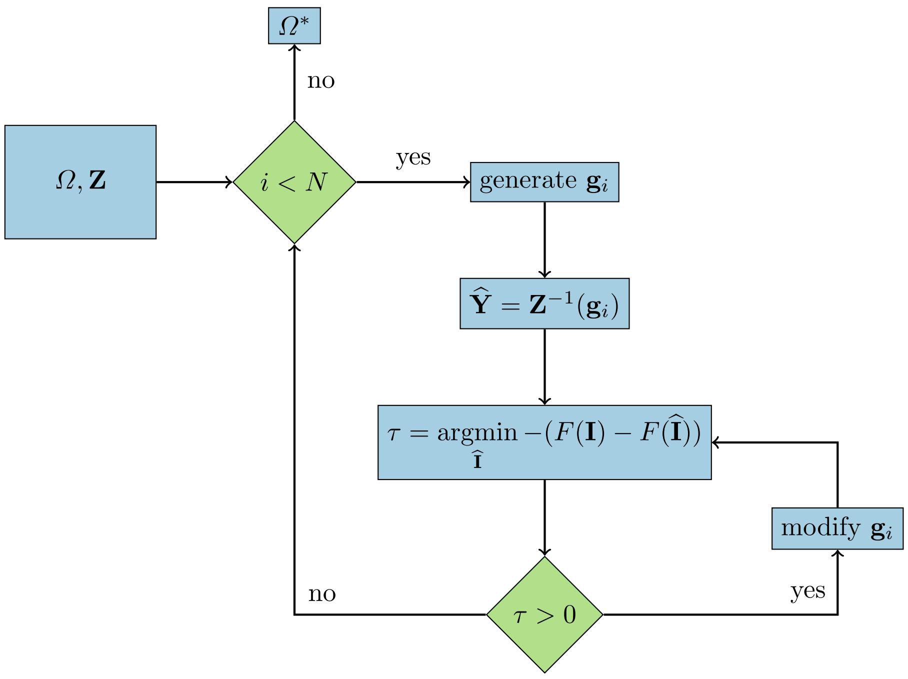

Ω^*
no
$\Omega, \mathbf{Z}$
$i < N$
yes
generate $\mathbf{g}_i$
$\widehat{\mathbf{Y}} = \mathbf{Z}^{-1}(\mathbf{g}_i)$
$\tau = \underset{\widehat{\mathbf{I}}}{\operatorname{argmin}} -(F(\mathbf{I}) - F(\widehat{\mathbf{I}}))$
modify $\mathbf{g}_i$
no
$\tau > 0$
yes

Figure 5.3: Flowchart of the shape optimization algorithm based on exact reanalysis.

generated. This can be done, *e.g.*, with Monte Carlo or a genetic algorithm. The initial impedance matrix $\mathbf{Z}$ is truncated according vector $\mathbf{g}_i$ (the columns and rows corresponding to zeros in $\mathbf{g}_i$ are removed) and inverted.

Within one global cycle, the local cycle is iteratively repeated, see the flowchart in Figure 5.3. The local improvement is based on an evaluation of the topology sensitivity τ [11] via exact reanalysis [51], being able to evaluate all structures differing in vector $\mathbf{g}_i$ by the Hamming distance equal to one from the actual shape, *i.e.*, all the nearest neighbors are evaluated without necessity of inverting the actual impedance matrix. It computes a fitness function F of a candidate's current $\widehat{\mathbf{I}}$ and evaluates improvement to the current best candidate $\mathbf{I}$. The best candidate is chosen, *i.e.*, Greedy algorithm is adapted, the structure is updated, and the optimization continues with the next step. The local cycle is repeated until the termination criterion is met. This can happen either by getting stuck in a local minimum or the fact that the relative improvement is smaller than the preset value. The entire algorithm operates over degrees of freedom, which are the basis functions. Consequently, the basis functions are removed or added back depending on the topology sensitivity and the initial seeds given by the heuristic algorithm.

The inversion-free solution of (5.8) is computed using block inversion [52] which is algebraically derived as

$$\mathbf{Y}_{bb} = \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix}^{-1} = \begin{bmatrix} \mathbf{A}^{-1} + \mathbf{A}^{-1}\mathbf{B}\mathbf{E}^{-1}\mathbf{C}\mathbf{A}^{-1} & -\mathbf{A}^{-1}\mathbf{B}\mathbf{E}^{-1} \\ -\mathbf{E}^{-1}\mathbf{C}\mathbf{A}^{-1} & \mathbf{E}^{-1} \end{bmatrix}, \qquad (5.3)$$

where

$$\mathbf{E} = \mathbf{D} - \mathbf{C}\mathbf{A}^{-1}\mathbf{B}. \tag{5.4}$$

This will allow us to compute current $\mathbf{I}$ as $\mathbf{I} = \mathbf{Y}\mathbf{V}$ for appropriately chosen $\mathbf{Y}$. For addition of an edge $[e]$ to the system of $[a]$ enabled basis functions, we get matrix

$$\mathbf{Y}_{bb} = \mathbf{Z}_{bb}^{-1} = \begin{bmatrix} \mathbf{Z}_{aa} & \mathbf{Z}_{ae} \\ \mathbf{Z}_{ea} & Z_{ee} \end{bmatrix}^{-1}, \tag{5.5}$$

where $\mathbf{Z}_{aa}$ is the original impedance matrix and terms $\mathbf{Z}_{ae}, \mathbf{Z}_{ea}$ and $\mathbf{Z}_{ee}$ are taken from the row and the column belonging to the basis function $[e]$.

Similarly, we can perform a removal of an edge by modification of the admittance matrix $\tilde{\mathbf{Y}}_{bb}$ with known (original) impedance matrix $\mathbf{Z}$

$$\tilde{\mathbf{Y}}_{bb} = \begin{bmatrix} \mathbf{Y}_{aa} + \dfrac{1}{Y_{ee}}\tilde{\mathbf{Y}}_{ae}\tilde{\mathbf{Y}}_{ea} & -\mathbf{Y}_{aa}\mathbf{Z}_{ae}Y_{ee} \\ -Y_{ee}\mathbf{Z}_{ea}\mathbf{Y}_{aa} & Y_{ee} \end{bmatrix}, \tag{5.6}$$

where the edge $[e]$ is removed. The admittance matrix after removal is evaluated as

$$\mathbf{Y}_{aa} = \tilde{\mathbf{Y}}_{aa} - \frac{1}{Y_{ee}}\tilde{\mathbf{Y}}_{ae}\tilde{\mathbf{Y}}_{ea}. \tag{5.7}$$

It can be easily seen that (5.7) is computationally more efficient than (5.5). The modified domain after N iterations is returned as an optimized shape, represented by the vector $\mathbf{g}_i$ which is the unknown from (5.1), of topology sensitivity algorithm.

Both Monte Carlo algorithm and the genetic algorithm can be well parallelized with almost linear speedup. This allows us to evaluate plethora of samples in every iteration and increase the diversity of solutions.

5.3 Optimized Antenna Parameters

Practically all antenna parameters can be defined as either quadratic or linear forms of current, evaluated as

$$\mathbf{I}_i = \mathbf{Z}^{-1}(\mathbf{g}_i)\mathbf{V}. \tag{5.8}$$

Their definitions vary in matrices used and in complexity of formulas to be evaluated. A few examples of interest in electrically small region are presented in this section, see, *e.g.*, [53] or [54] for further details.

Antenna Metrics

Electrically small antennas are known to be narrowband [55]. The bandwidth is inversely proportional to Q-factor [56], therefore, to increase the bandwidth, we have to minimize the Q-factor. It is defined as [57]

$$Q = \frac{\max\{\mathbf{I}^{\mathrm{H}}\mathbf{X}_{\mathrm{m}}\mathbf{I}, \mathbf{I}^{\mathrm{H}}\mathbf{X}_{\mathrm{e}}\mathbf{I}\ \}}{\mathbf{I}^{\mathrm{H}}\mathbf{R}\mathbf{I}}, \tag{5.9}$$

where $\mathbf{X}_{\mathrm{m}}$ and $\mathbf{X}_{\mathrm{e}}$ are computed from the imaginary part of the impedance matrix as

$$\mathbf{X}_{\mathrm{m}} = \frac{1}{2}(\mathbf{W} + \mathbf{X}), \tag{5.10}$$

$$\mathbf{X}_{\mathrm{e}} = \frac{1}{2}(\mathbf{W} - \mathbf{X}), \tag{5.11}$$

$$\mathbf{Z} = \mathbf{R} + \mathrm{j}\mathbf{X}, \tag{5.12}$$

$$\mathbf{W} = \omega\frac{\partial \mathbf{X}}{\partial V}. \tag{5.13}$$

Substituting for unknown currents $\mathbf{I}$ from (5.8), we get

$$Q = \frac{\max\left\{\mathbf{V}^{\mathrm{H}}\mathbf{Z}^{-\mathrm{H}}(\mathbf{g})\mathbf{X}_{\mathrm{m}}\mathbf{Z}^{-1}(\mathbf{g})\mathbf{V}, \mathbf{V}^{\mathrm{H}}\mathbf{Z}^{-\mathrm{H}}(\mathbf{g})\mathbf{X}_{\mathrm{e}}\mathbf{Z}^{-1}(\mathbf{g})\mathbf{V}\right\}}{\mathbf{V}^{\mathrm{H}}\mathbf{Z}^{-\mathrm{H}}(\mathbf{g})\mathbf{R}\mathbf{Z}^{-1}(\mathbf{g})\mathbf{V}}, \tag{5.14}$$

i.e., it is seen that the value of Q-factor is a complicated non-linear function of the shape, represented by a genus $\mathbf{g}$ and excitation $\mathbf{V}$. The evaluation of (5.14) is expensive since the matrix inversion of complexity $\mathcal{O}(N^3)$ and quadruple matrix multiplication of complexity $\mathcal{O}(N^3)$. However, considering fixed excitation $\mathbf{V}$, all the local perturbations can be evaluated in $\mathcal{O}(N^2)$ as proposed in Section 5.2.

The next parameter which we will use in fitness functions is dissipation factor δ defined as [58]

$$\delta = \frac{\mathbf{I}^{\mathrm{H}}\mathbf{L}\mathbf{I}}{\mathbf{I}^{\mathrm{H}}\mathbf{R}\mathbf{I}}, \tag{5.15}$$

where the loss matrix $\mathbf{L} = [L_{mn}]$ is defined element-wise as

$$L_{mn} = \sigma \int\limits_{\Omega} \boldsymbol{f}_n(\boldsymbol{r}) \cdot \boldsymbol{f}_m(\boldsymbol{r})\,\mathrm{d}V, \tag{5.16}$$

where σ is surface resistivity.

Finally, the third parameter, which will be optimized, is called voltage reflection coefficient $\varGamma$ defined as

$$\varGamma = \frac{Z_{\mathrm{in}} - Z_0}{Z_{\mathrm{in}} + Z_0}, \tag{5.17}$$

where Z_0 is characteristic impedance of the transmission line and Z_{in} is antenna input impedance defined as

$$Z_{\mathrm{in}} = \frac{\mathbf{I}^{\mathrm{H}}\mathbf{Z}\mathbf{I}}{|I_{\mathrm{in}}|^2}, \tag{5.18}$$

and where I_{in} is the port input current.

Regularization

Optimization of physical parameters does not tell us anything about how big antenna's surface is or how it is shaped. Let us introduce regularization parameters that can steer the algorithm away from undesirable shapes (see Figure 5.4) or creating unnecessarily large antennas. We start with the relative antenna area A_{rel}. It describes area of the metallic part with respect to the full configuration. Its values are between 0 and 1 and it can be computed as follows

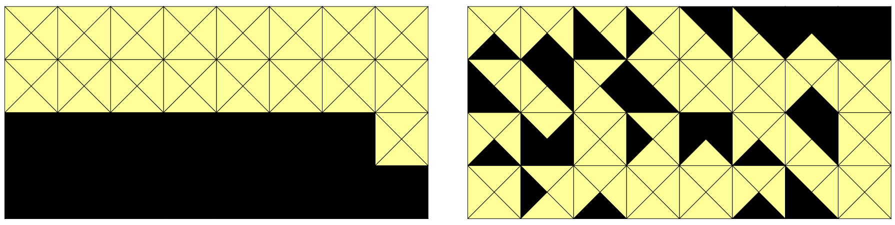
$$A_{\mathrm{rel}}(\mathbf{g}) = \frac{a(\mathbf{g})}{\sum\limits_{n} a_n}, \tag{5.19}$$

where $a(\mathbf{g})$ returns area induced by vector $\mathbf{g}$ and a_n is area of triangle n.

The next parameter is the geometric divergence G. It represents how regular the structure is. The zero value of parameter G is delivered by a continuous structure completely filling the bounding box or by vacuum without any scatterer at all, and the value one is approached by the most irregular curves. It is defined here as

$$G(\mathbf{g}) = \frac{1}{N} \sum_{n=1}^{N} \left(1 - |2\mathbf{B}\mathbf{g} - 1|\right), \tag{5.20}$$

where $\mathbf{B}$ is the adjacency matrix constructed from the connectivity list of the basis functions.



(a) A structure with $G = 0.0844$. **(b)** A structure with $G = 0.5494$.

Figure 5.4: Examples of structure regularity (the black parts are cut-out, *i.e.*, metallization is replaced by vacuum).

Optimized Fitness Functions

In order to demonstrate how to merge the antenna metrics and the geometry regularization together, two fitness functions are introduced, the first being minimized, while the second being maximized.

The first fitness function deals with the reflection coefficient to be minimized and, at the same time, the antenna shape being as regular as possible and spanning as small area as possible. This composite function is written as

$$F_1(\mathbf{g}) = (1 + |\Gamma|^2)(1 + \alpha G(\mathbf{g}))(1 + \beta A_{\mathrm{rel}}(\mathbf{g})). \tag{5.21}$$

where α and β are the hyperparameters weighting the priority in shape regularity and spanned area, respectively. The fitness function is composed as to have one as the minimum and infinity as the maximum. This subsequently leads to the multi-objective optimization.

The second fitness function that we define is the total efficiency which holds [15]

$$F_2(\mathbf{g}) = -\frac{(1 - |\mathit{\Gamma}|^2)}{1 + \delta}(1 - \alpha G(\mathbf{g}))(1 - \beta A_{\mathrm{rel}}(\mathbf{g})). \tag{5.22}$$

It is computationally expensive to evaluate the presented functions but it allows us to generate huge datasets at the same time for problems with exact solutions that can not be found in feasible time.

5.4 Discussion

Before we dive into details of parameter and fitness functions optimization, let us summarize some observations related to the shape synthesis. The problem of the shape synthesis is considered unsolved. This applies to the antenna synthesis as well. The methods like exact reanalysis can help us to understand the underlying problem by extensive testing of its properties. We can ask if the algorithm can converge into a single or multiple minima. It is also necessary to know if a different way of initialization significantly improves the result.

The step of the vector $\mathbf{g}_i$ generation in Figure 5.3 can be implemented in various ways. It may be implemented as Monte Carlo sampling, maximization of vector distances between generations based on a generalized Latin hypercube sampling [59] or a genetic algorithm. The hypothesis which we want to investigate is to replace the existing closed-form with an approximative model. It should predict the behavior of parameters based on learned knowledge from a set of samples. Such a model can be *e.g.*, represented by a neural network.

Chapter 6

Exploitation of Approximation Models in Shape Optimization

The type of optimization problems presented in Chapter 5 is mostly hard to efficiently tackle by any known algorithm. It consists of a huge state space which is searched through by algorithms needing another set of hyperparameters ranging from number of agents to gradient step size. In this chapter, we try to evaluate dependencies between samples, the impact of vector initialization on the genetic algorithm's result and incorporate classification algorithms of artificial intelligence to categorize samples by their optimized parameters.

6.1 State Space Sampling

A problem which we immediately encounter is related to sampling methods. We may generate different initial vectors $\mathbf{g}$ but still get the same (or similar) vector(s) $\mathbf{t}$. Therefore, there exist the same structures (metallizations) with the same definition of enabled triangles which, however, may have different values of parameters and consequently different values of a fitness function. The question is how to sample such a space which is not well behaved in small perturbations to achieve good gene diversity and homogeneity of samples.

Random Sampling

Let us consider random sampling which may give us a rough estimate of the domain. An example in the form of a waterfall diagram is in Figure 6.1a. The first drawback is given by the property of uniform sampling which returns in average a half of the basis functions enabled and a half disabled. In this case, we do not have any guarantee for the maximization of the Hamming distance between multiple samples. The random process may also generate the same sample multiple times, which may be undesirable. Therefore, it may be needed to remember and compare generated samples between generations of the genetic algorithm and throw away repeating samples. This imposes higher memory and computational requirements for storing information about previous initialization vectors.

Equidistant Sampling

Ideally, we would like to sample N-dimensional hypercube vertices aggregated in a set $\mathcal{H}$ in our state space (where each bit of the vector $\mathbf{g}$ represents one dimension), while maximizing the Hamming distance [60] between them. This can be mathematically formulated as follows

$$\mathcal{H} = \underset{\mathbf{G}}{\operatorname{argmax}} \sum_{i,j} d_{\mathrm{H}}(\mathbf{g}_i, \mathbf{g}_j), \tag{6.1}$$

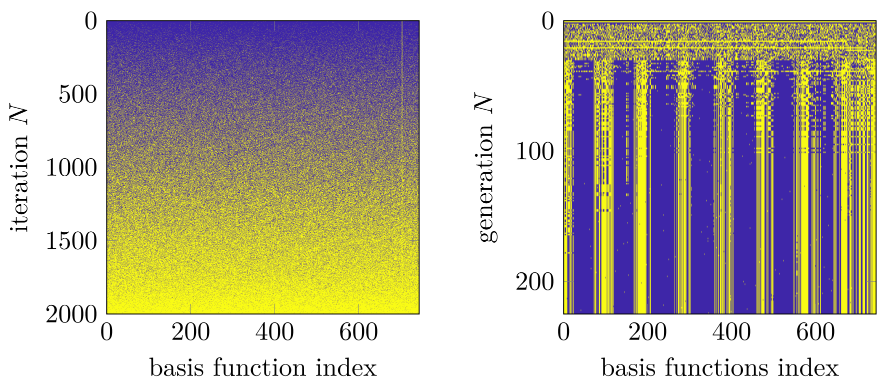
where $\mathbf{G}$ is a set of M samples from N dimensional state space. The Hamming distance between two vectors $\mathbf{g}_i$ and $\mathbf{g}_j$ is denoted as $d_{\mathrm{H}}(\mathbf{g}_i, \mathbf{g}_j)$. It returns the number of positions at which the vectors are different

Generally, there exist exponentially many distinct samples. If we consider all samples from zero enabled basis functions to full configuration of N enabled basis functions, we get formula

$$\sum_{k=0}^{N} \binom{N}{k} = 2^N. \tag{6.2}$$

We assume in this work that one basis function is fixed and serves as a feeding[1], therefore, we have $N-1$ degrees of freedom. The dataset sizes for fixed d_{H} and N were in the past tabulated [61]. The problem of the Hamming distance maximization for bit words of a given length and a given number of samples remains unsolved until nowadays. Due to the complexity of this approach, we will further investigate state spaces by random sampling.

[1]Without loss of generality, we consider only one-port antennas in this work. This restriction is common for electrically small antennas. The theory presented here, however, works without any major changes for multi-port antennas and scattering problems as well.



(a) Monte Carlo initial vectors $\mathbf{g}_i$ sorted by the number of enabled basis functions.

(b) Genetic algorithm initial vectors $\mathbf{g}_i$ sorted by the number of iteration.

Figure 6.1: Initial basis function vectors $\mathbf{g}$.

Probabilistic Distribution of Solutions

We now run the algorithm presented in Section 5.2 with our fitness functions F_1 and F_2. We want find to out probabilities of basis functions in the final designs after the optimization process. For the comparison of the solutions, we also run Monte Carlo algorithm where we expect smoother probability transitions. Figure 6.2 and Figure 6.3 show visualization of probabilities for a 8×16 pixelized structure.

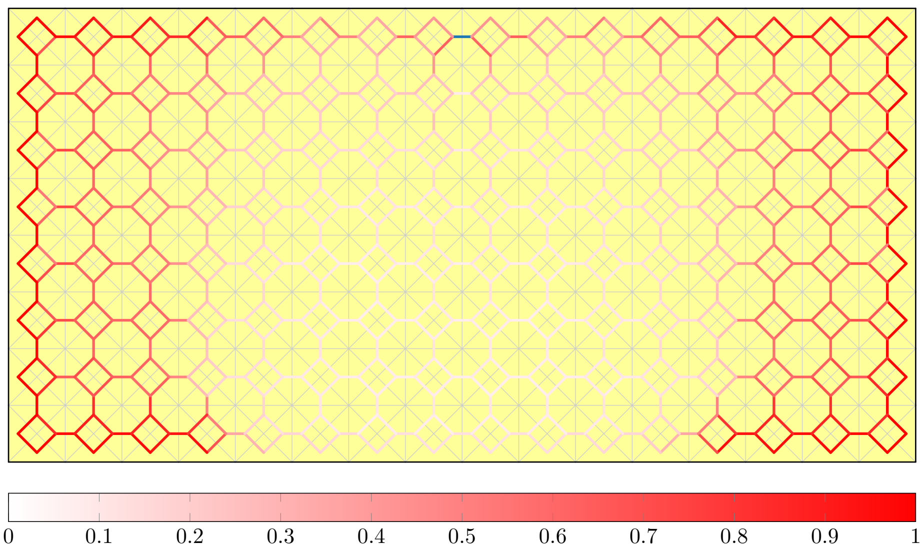


Figure 6.2: Basis function probabilities in the final design evaluated for fitness function F_1, $\alpha = 0$, $\beta = 0$ after 2000 Monte Carlo algorithm iterations.

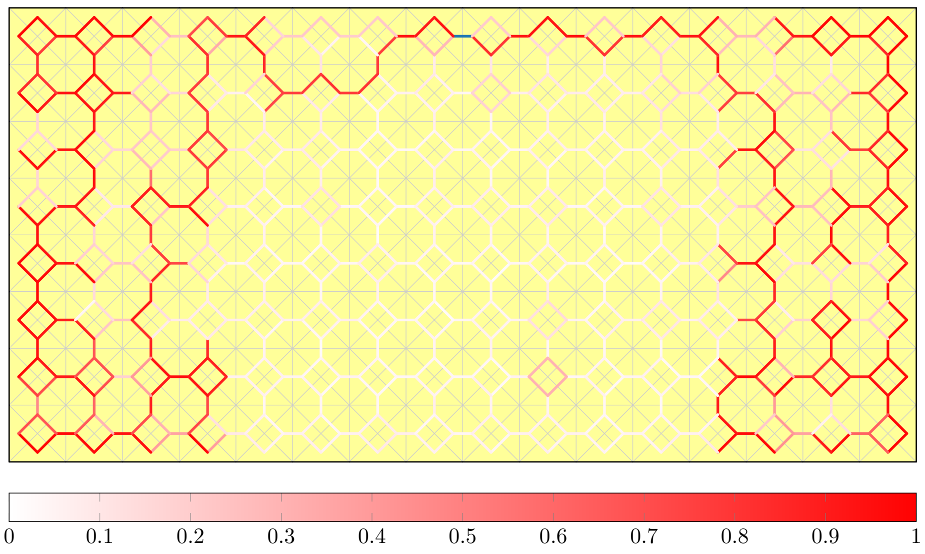


Figure 6.3: Basis function probabilities in the final design evaluated for fitness function F_2, $\alpha = 0$, $\beta = 0$ after 15 iterations of the genetic algorithm with 15 agents.

Probability maps show that minimization of Q-factor is equivalent to maximization of the polarizability of the antenna structure. It is then visible as the charge separation on the sides of a structure [62].

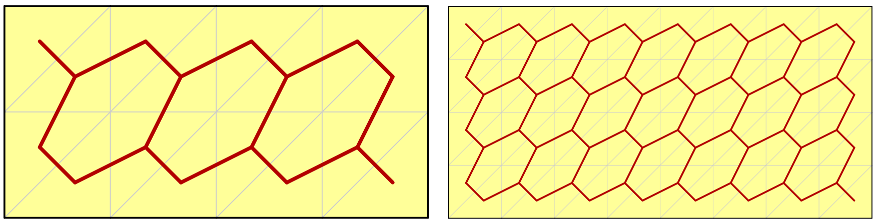
6.2 Classification of Solutions

Our goal is to train a classifier that can filter promising basis vectors in the first step of the optimization process, see Figure 5.3. Its input should be a vector of basis functions $\mathbf{g}$ and it outputs a class based on the values of Q-factor. The choice of Q-factor for the classification was motivated by its non-linearity and computational complexity where its explicit evaluation is the speed bottleneck of optimization algorithms, *cf.* (5.14).

We try to show the feasibility of the concept with two illustrative structures. The first structure is a 2×4 pixelized plate with 18 basis functions in Figure 6.4a. In this case, the small number of basis functions allows us to enumerate all combinations. The second structure is a 4×8 plate of the same physical size but with a denser mesh in Figure 6.4b. It has 84 basis functions and it is already computationally infeasible to enumerate all combinations.

Classification Model

Our chosen model is a multi-layer perceptron [63]. The input layer has the number of inputs based on degrees of freedom – basis functions on a structure.



(a) A 2×4 structure. **(b)** A 4×8 structure.

Figure 6.4: The structures used for classification.

For binary classification, we have a linear layer with sigmoid activation [64] which evaluates as

$$\sigma(x) = \frac{1}{1 + \mathrm{e}^{-x}}. \tag{6.3}$$

Its outputs are scaled on the interval between zero and one. The resulting class can be obtained by thresholding the output. The loss function used for binary classification is binary cross entropy [65] defined as

$$\ell(x, y) = y \ln(x) + (1 - y) \ln(1 - x), \tag{6.4}$$

where x and y are scalar values representing predicted and target values.

The multi-class prediction [66] has the number of outputs equal to the number of classes. The used activation function is softmax

$$S_i(\mathbf{x}) = \frac{e^{x_i}}{\sum\limits_{j=1}^{N} \mathrm{e}^{x_j}} \tag{6.5}$$

which computes class probability for each of the N possible classes. The loss in the N-class case is multinominal cross entropy [65], which is computed as

$$\ell(\mathbf{x}, c) = -\ln(S_c(\mathbf{x})) = -x_c + \ln\left(\sum_{j=1}^{N} \mathrm{e}^{x_j}\right), \tag{6.6}$$

where c is index of the target class. Notice that multinominal cross entropy is defined using the softmax function. This has practical consequences for implementation when both of them are often combined into one layer.

The optimizer of our choice is Adaptive Moment Estimation (Adam) [67]. This optimizer can outperform, by speed and accuracy, other optimization algorithms (*e.g.*, stochastic gradient descent) [68]. We kept the parameters β_1, β_2, and ϵ of Adam optimizer set to their default values, *i.e.*, $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-8}$ as proposed in [67].

Optimized Parameters

A neural network is a complex model with many parameters, some of them need to be chosen beforehand – we call them hyperparameters [69]. Apart from learning of the weights of the neural network by backpropagation [69], we need to select feasible values for the algorithm's hyperparameters. Namely, we optimize the learning rate of the optimizer, the batch size, the number of hidden layers and the size of hidden layers. The size of hidden layers is relaxed as one parameter, therefore all hidden layers have the same size. We restricted hyper-parameters' space to values in Table 6.1.

We can use multiple strategies to find the best combination of parameters.

parameter	minimal value	maximal value
learning rate	10^{-5}	10^{-3}
batch size	64	512
number of hidden layers	1	3
size of a hidden layer	20	1000

Table 6.1: Optimized hyper-parameters space for a neural network.

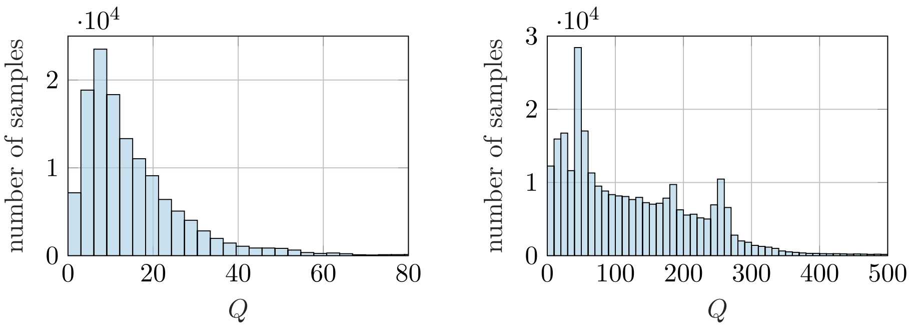
The simplest one for implementation is random search. We generate a random value for each parameter from a given interval and evaluate it. Similarly, we can employ enumerative (grid) search that sweeps parameter intervals with a given step. Both algorithms can be well parallelized as there is no dependency between the iterations.

The algorithm of our choice is heuristical Tree-structured Parzen Estimator implemented in *hyperopt* package [70]. It models probabilities $P(x \mid y)$ and $P(y)$ where y is a quality score (in our case the test loss) and x represents hyper-parameters. As this approach can be computationally very expensive even on a GPU, we run 250 hyperparameter optimization iterations, each with 150 epochs.

Datasets

We now want to create datasets which will contain information about the structures. We created three datasets for each structure where data representation of a structure is in the form of a basis vector $\mathbf{g}$ and its respective Q value. Firstly, we need a training dataset which is used for the optimization of the model's parameters (*e.g.*, weights of a neural network). The second dataset is called validation dataset and it is used for hyperparameters tuning. Lastly, we measure generalization performance of a fully trained model with a test dataset. It is presumed that all three datasets are generated from the same underlying distribution.

The dataset for the 2×4 structure was generated by enumeration of all possible combinations with $2^{18-1} = 131072$ valid samples. The random sampling was chosen for the 4×8 structure where, in total, we generated 281613 unique samples. The distribution of Q-factor across all three datasets can be seen in Figure 6.5.



(a) The 2×4 structure. **(b)** The 4×8 structure.

Figure 6.5: Distribution of Q-factor in generated datasets.

Binary Classification

The datasets for both structures were split into positive and negative samples by thresholding Q values. The small 2×4 structure has positive samples with $Q \leq 15$. Similarly, the large 4×8 structure has positive samples with $Q \leq 50$. The dataset sizes for the small structure are in Table 6.2 and for the large structure in Table 6.3.

class	training	validation	test
positive samples	56088	16089	8087
negative samples	35693	10125	5020

Table 6.2: Datasets for the 2×4 structure.

class	training	validation	test
positive samples	58483	16518	8344
negative samples	138647	39804	19817

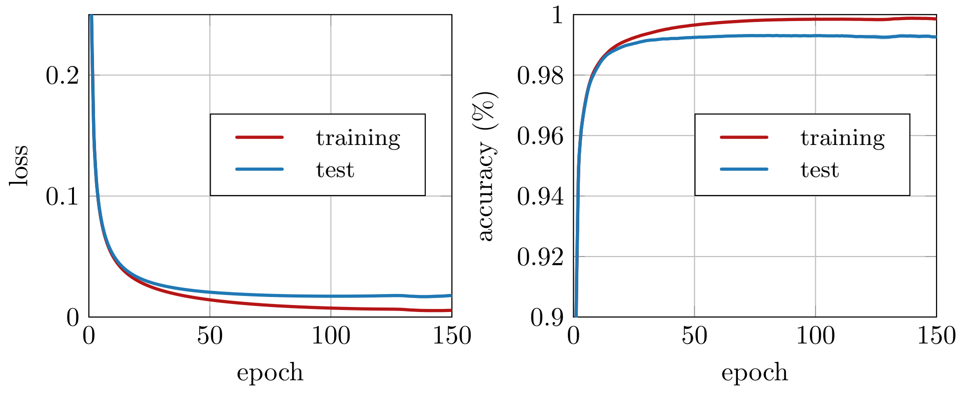
Table 6.3: Datasets for the 4×8 structure.

The optimal values of the hyperparameters for both structures found after 250 optimization iterations can be seen in Table 6.4. We can see that both learning rate values are of the same order of magnitude. On the other hand, the classification of the smaller structure performs better with a shallow neural network with many hidden units, whereas the larger structure can be classified better with a three-layer network with few hidden neurons in each layer.

We then set hyperparameters to the values depicted in Table 6.4 and trained the network for ten cycles. The training progress of the binary classification training for the 2×4 structure can be seen in Figure 6.6. The network performs well for this simple example with the test accuracy $(0.9942 \pm 0.0007)\%$.

parameter	2×4 structure	4×8 structure
learning rate	$7.91 \cdot 10^{-4}$	$9.77 \cdot 10^{-4}$
batch size	496	209
number of hidden layers	1	3
size of a hidden layer	787	21
minimal validation loss	$1.50 \cdot 10^{-2}$	$1.34 \cdot 10^{-1}$

Table 6.4: Hyperparameter values obtained after the optimization with minimal validation loss.

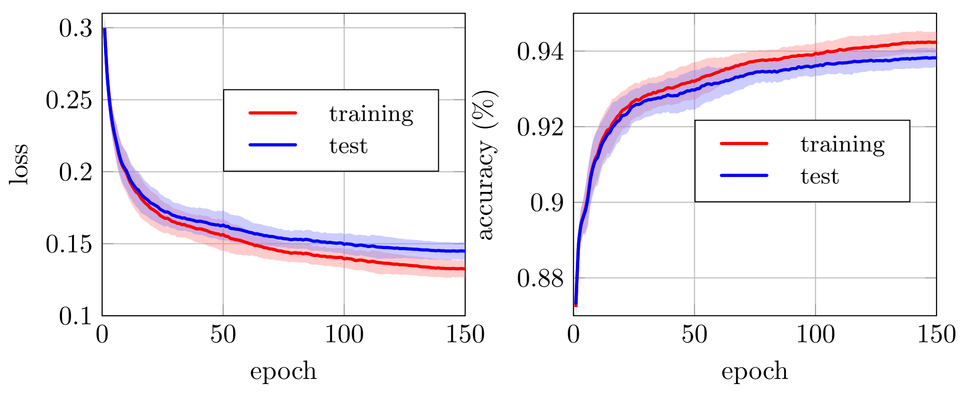


(a) Loss. **(b)** Accuracy.

Figure 6.6: Results of the binary classification for the 2×4 structure.

A neural network for the large structure was evaluated in the same way as for the small structure. Training and testing values of the training process are depicted in Figure 6.7. In this case, the accuracy reached the value of $(0.9382 \pm 0.0024)\%$.

The results show that the binary classification of solutions can be done with fairly good accuracy. Nevertheless, it still may be computationally expensive to train a neural network even for smaller structures.



(a) Loss. **(b)** Accuracy.

Figure 6.7: Results of the binary classification for the 4×8 structure.

Classification into N Classes

Similarly to the previous section where we thresholded Q by a single value, we now split Q into multiple intervals. This will allow us to create multiple classes and transform the problem into N-class classification. The classes and dataset sizes are defined in Table 6.6 for the 2×4 and in Table 6.7 for the 4×8 structure.

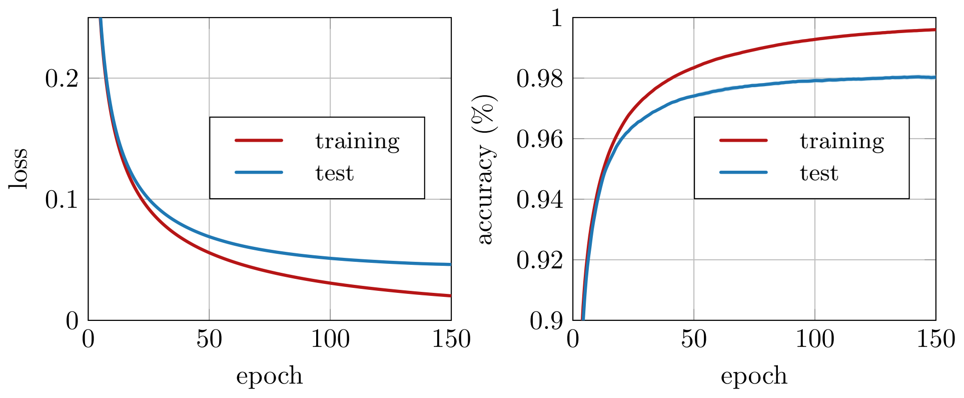
The optimization and evaluation process was the same as for the binary case (see Table 6.5 for optimized hyperparameter values). We achieved accuracy of $\epsilon = (0.9803 \pm 0.0009)\%$ for the small structure. The training process is depicted in Figure 6.8.

parameter	2×4 structure	4×8 structure
learning rate	$5.64 \cdot 10^{-4}$	$7.62 \cdot 10^{-4}$
batch size	340	215
number of hidden layers	1	3
size of a hidden layer	865	48
minimal validation loss	$4.38 \cdot 10^{-2}$	$2.48 \cdot 10^{-1}$

Table 6.5: Hyperparameter values obtained after the optimization with minimal validation loss.

class	training	validation	test
class 1, $Q \in [1, 8]$	28128	8210	4026
class 2, $Q \in (8, 15]$	27960	7879	4061
class 3, $Q \in (15, 30]$	25355	7241	3558
class 4, $Q \in (30, \inf]$	10308	2884	1462

Table 6.6: N-class classification dataset sizes for the 2×4 structure.



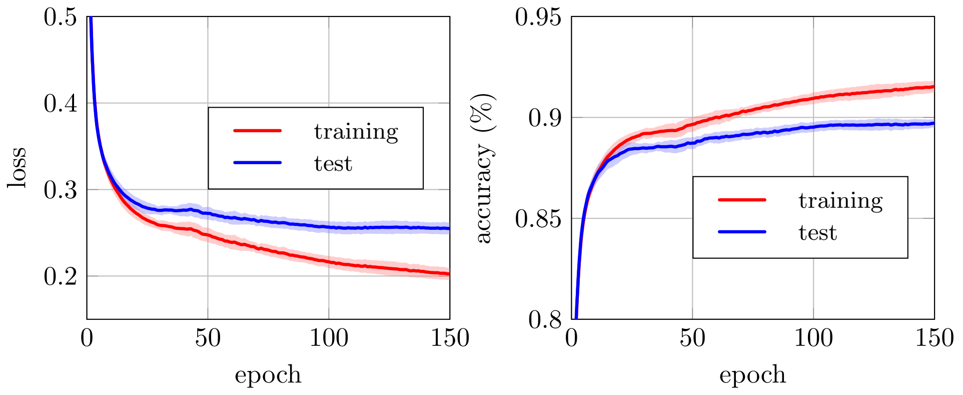
(a) Loss. **(b)** Accuracy.

Figure 6.8: Results of the N-class classification for the 2×4 structure.

The same evaluation process as above was also repeated for the 4×8 structure (see Figure 6.9) with significantly lower accuracy of $(0.8971 \pm 0.0023)\%$.

class	training	validation	test
class 1, $Q \in [1, 20]$	19725	5576	14403
class 2, $Q \in (20, 35]$	16014	4459	8615
class 3, $Q \in (35, 100]$	61326	17494	2310
class 4, $Q \in (100, \inf]$	100065	28793	2833

Table 6.7: N-class classification dataset sizes for 2×4 structure.



(a) Loss. **(b)** Accuracy.

Figure 6.9: Results of the N-class classification for the 4×8 structure.

Discussion

We can notice that the values in Table 6.4 for binary classification and in Table 6.5 for N-class classification are of the same magnitude. It leads us to the conclusion that the complex structures would need deeper architectures for better performance. Additionally, we conclude that accuracy for the large structure is already bellow the bound of usability. A huge drawback of this approach is the time needed for hyperparameter optimization. We need to find optimal parameters for each new structure. This may take hours even with good hardware and it grows rapidly with the depth of a neural network and the increasing size of datasets.

Chapter 7

Conclusion

In this thesis, we briefly reviewed the implementation of the method of moments for perfectly conducting obstacles and its use for electrically small antennas. It was implemented using Nvidia's proprietary technology CUDA, and it can be run on graphical processing units from MATLAB's interface in cooperation with existing code in AToM. Moreover, the validity of the code was proved by comparing the results with the simulation performed in Feko suite. The achieved speedups are by order of magnitude compared to the already existing implementation in AToM. The code can be already downloaded from the AToM website.

In the next part of the thesis, we described the idea of antenna shape optimization and focused on a new algorithm based on the topology sensitivity which significantly speeds up the process. It was shown in Chapter 6 that the current shape optimization algorithm based on the topology sensitivity converges to similar solutions even with different initial vectors. We were unable to satisfyingly extrapolate the underlying dependencies of parameter Q from vectors of basis functions by neural networks. The complexity of the state space seems to be too high for our rather simple model.

Future Work

We have proved it is conceptually a good idea to transform suitable parts of CPU code to GPUs. Another step is to rewrite the volumetric method of moments operating with tetrahedrons, which is developed at the Department of Electromagnetic Field as well. This code needs even more careful treatment as it solves problems by order of magnitude larger than surface MoM. These problems might be already too large for current capacities of GPU memories and it might be necessary to divide computation in smaller computational blocks.

Lastly, we want to incorporate the results and observations from this thesis into the shape synthesis algorithm to accelerate it even more. It will very likely require to rethink the algorithm and rewrite it from scratch.

Appendix A

Bibliography

[1] AToM - Antenna Toolbox for MATLAB [online]. [cit. 2020-05-20]. Available from: http://www.antennatoolbox.com/

[2] Štrambach, M. *Triangulation of planar objects and its implementation into the AToM package*. Bachelor thesis, Czech Technical University, 2017.

[3] Harrington, R. F. *Field Computation by Moment Methods*. Piscataway, New Jersey, United States: Wiley – IEEE Press, 1993.

[4] CSCD. *MeteoSwiss in CSCS [online]*. [cit. 2020-02-12]. Available from: https://www.cscs.ch/computers/kesch-escha-meteoswiss/

[5] Rupp, K. Microprocessor Trend Data. https://github.com/karlrupp/microprocessor-trend-data, 2018.

[6] Nvidia. *CUDA zone [online]*. [cit. 2020-02-12]. Available from: https://developer.nvidia.com/cuda-zone

[7] Memeti, S.; Li, L.; et al. Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: Programming Productivity, Performance, and Energy Consumption. In *Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*, ARMS-CC '17, Association for Computing Machinery, 2017, ISBN 9781450351164, p. 1–6, doi:10.1145/3110355.3110356.

[8] MPI. *Open MPI [online]*. [cit. 2020-02-12]. Available from: https://www.open-mpi.org

[9] The Matlab. 2020. Available from: www.mathworks.com

[10] Chew, W. C. Unsolved Problems in EM and CEM: A Personal Perspective. *IEEE Antennas and Propagation Magazine*, volume 54, no. 6, Dec. 2012: pp. 270–274, doi:10.1109/MAP.2012.6387844.

[11] Capek, M.; Jelinek, L.; et al. Shape Synthesis Based on Topology Sensitivity. *IEEE Trans. Antennas Propag.*, volume 67, no. 6, June 2019: pp. 3889 – 3901, doi:10.1109/TAP.2019.2902749.

[12] Peterson, A. F.; Ray, S. L.; et al. *Computational Methods for Electromagnetics*. Wiley – IEEE Press, 1998.

[13] Harrington, R. F. *Time-Harmonic Electromagnetic Fields*. Wiley – IEEE Press, second edition, 2001.

[14] Gibson, W. C. *The Method of Moments in Electromagnetics*. Chapman and Hall/CRC, second edition, 2014, ISBN 13:978-1-4822-3580-7.

[15] Balanis, C. A. *Advanced Engineering Electromagnetics*. Wiley, 1989.

[16] Rao, S.; Wilton, D.; et al. Electromagnetic scattering by surfaces of arbitrary shape. *IEEE Transactions on Antennas and Propagation*, volume 30, no. 3, 1982: pp. 409–418.

[17] Burden, R.; Faires, J.; et al. *Numerical Analysis*. Cengage Learning, 2015, ISBN 9781305465350.

[18] Cismasu, M.; Gustafsson, M. Antenna Bandwidth Optimization With Single Frequency Simulation. *IEEE Trans. Antennas Propag.*, volume 62, no. 3, 2014: pp. 1304–1311.

[19] Al-Khafaji, A.; Tooley, J. *Numerical Methods in Engineering Practice*. Oxford University Press, 01 1986.

[20] Piegl, L.; Tiller, W. *The NURBS Book*. New York, NY, USA: Springer-Verlag, second edition, 1996.

[21] De Loera, J. A.; Rambau, J.; et al. *Triangulations – Structures for Algorithms and Applications*. Berlin, Germany: Springer, 2010.

[22] WIPL-D d.o.o. WIPL-D. 2020. Available from: http://www.wipl-d.com/

[23] Delaunay, B. Sur la sphère vide. A la mémoire de Georges Voronoı. *Bulletin de l'Académie des Sciences de l'URSS. Classe des sciences mathématiques et na*, volume 6, 1934.

[24] Miller, G. *Numerical Analysis for Engineers and Scientists*. Cambridge University Press, 2014, doi:10.1017/CBO9781139108188.

[25] Dunavant, D. A. High degree efficient symmetrical Gaussian quadrature rules for the triangl. *International Journal for Numerical Methods in Engineering*, volume 21, 1985: pp. 1129–1148.

[26] Graglia, R. D. On the numerical integration of the linear shape functions times the 3-D Green's function or its gradient on a plane triangle. *IEEE Transactions on Antennas and Propagation*, volume 41, no. 10, Oct. 1993: pp. 1448–1455, doi:10.1109/8.247786.

[27] Cheng J., G. M., McKercher T. *Professional CUDA C Programming*. GBR: Wrox Press Ltd., first edition, 2014, ISBN 1118739329.

[28] Toy, W.; Zee, B. *Computer Hardware/Software Architecture*. Prentice Hall, 01 1986, ISBN 978-0-13-163502-9.

[29] Kirk, D. B.; mei W. Hwu, W. *Programming Massively Parallel Processors (Third Edition)*. Morgan Kaufmann, third edition edition, 2017, ISBN 978-0-12-811986-0, doi:https://doi.org/10.1016/B978-0-12-811986-0.00026-1.

[30] MathWorks. *MATLAB GPU functions [online]*. [cit. 2020-02-04]. Available from: https://www.mathworks.com/help/matlab/referencelist.html?type=function&capability=gpuarrays

[31] MathWorks. *MATLAB arrayfun function [online]*. [cit. 2020-04-16]. Available from: https://www.mathworks.com/help/parallel-computing/gpuarray.arrayfun.html

[32] MathWorks. *MATLAB feval function [online]*. [cit. 2020-02-12]. Available from: https://www.mathworks.com/help/matlab/ref/feval.html

[33] MathWorks. *MATLAB gather function [online]*. [cit. 2020-04-16]. Available from: https://www.mathworks.com/help/parallel-computing/gather.html

[34] MathWorks. *MATLAB GPU CUDA code [online]*. [cit. 2020-04-16]. Available from: https://www.mathworks.com/help/parallel-computing/run-cuda-or-ptx-code-on-gpu.html

[35] Harris, M.; et al. Optimizing parallel reduction in CUDA. *Nvidia developer technology, 2.4: 70.*, 2007.

[36] Intel. *Intel Core i7-8750H [online]*. [cit. 2020-02-10]. Available from: https://ark.intel.com/content/www/us/en/ark/products/134906/intel-core-i7-8750h-processor-9m-cache-up-to-4-10-ghz.html

[37] Nvidia. *Nvidia V100 Tensor Core GPU [online]*. [cit. 2020-04-17]. Available from: https://www.nvidia.com/en-us/data-center/v100/

[38] RCI cluster. 2020. Available from: www.rci.cvut.cz

[39] Altair. *Feko [online]*. [cit. 2020-05-7]. Available from: https://altairhyperworks.com/product/FEKO

[40] Deschamps, G.; Cabayan, H. Antenna Synthesis and Solution of Inverse Problems by Regularization Methods. *IEEE Transactions on Antennas and Propagation*, volume 20, no. 3, May 1972: pp. 268–274, doi:10.1109/tap.1972.1140197. Available from: https://doi.org/10.1109/tap.1972.1140197

[41] Bendsoe, M. P.; Sigmund, O. *Topology Optimization: Theory, Methods and Applications*. Springer, 2004, ISBN 9783540429920.

[42] Nemhauser, G. L.; Wolsey, L. A. *Integer an Combinatorial Optimization*. John Wiley & Sons, 1999, ISBN 0-471-35943-2.

[43] Vazirani, V. V. *Approximation Algorithms*. Berlin, Heidelberg: Springer-Verlag, 2001, ISBN 3540653678.

[44] Onwubolu, G. C.; Babu, B. V. *New Optimization Techniques in Engineering*. Berlin, Germany: Springer, 2004.

[45] Deb, K. *Multi-Objective Optimization using Evolutionary Algorithms*. New York, United States: Wiley, 2001.

[46] Deb, K.; Pratap, A.; et al. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, volume 6, no. 2, 2002: pp. 182–197.

[47] Dorigo, M.; Maniezzo, V.; et al. Ant System: Optimization by a colony of cooperating agents. IEEE Trans Syst Man Cybernetics - Part B. *IEEE transactions on systems, man, and cybernetics. Part B, Cybernetics : a publication of the IEEE Systems, Man, and Cybernetics Society*, volume 26, 02 1996: pp. 29–41, doi:10.1109/3477.484436.

[48] Kennedy, J.; Eberhart, R. C. *Swarm Intelligence*. Academic Press, 2001.

[49] Bendsoe, M. P.; Sigmund, O. *Topology Optimization*. Berlin, Germany: Springer, second edition, 2004, ISBN 978-3540429920.

[50] Čapek, M.; Jelinek, L.; et al. Inversion-Free Evaluation of Nearest Neighbors in Method of Moments. *IEEE Antennas and Wireless Propagation Letters*, volume PP, 04 2019: pp. 1–1, doi:10.1109/LAWP.2019.2912459.

[51] Ohsaki, M. *Optimization of Finite Dimensional Structures*. CRC Press, 2011.

[52] Horn, R. A.; Johnson, C. R. *Matrix Analysis*. Cambridge, United Kingdom: Cambridge University Press, 2017.

[53] Gustafsson, M.; Tayli, D.; et al. Antenna Current Optimization using MATLAB and CVX. *FERMAT*, volume 15, no. 5, May–June 2016: pp. 1–29. Available from: http://www.e-fermat.org/articles/gustafsson-art-2016-vol15-may-jun-005/

[54] Jelinek, L.; Capek, M. Optimal Currents on Arbitrarily Shaped Surfaces. *IEEE Trans. Antennas Propag.*, volume 65, no. 1, Jan. 2017: pp. 329–341, doi:10.1109/TAP.2016.2624735.

[55] Volakis, J. L.; Chen, C.; et al. *Small Antennas: Miniaturization Techniques & Applications*. McGraw-Hill, 2010.

[56] Yaghjian, A. D.; Best, S. R. Impedance, Bandwidth and Q of Antennas. *IEEE Trans. Antennas Propag.*, volume 53, no. 4, Apr. 2005: pp. 1298–1324, doi:10.1109/TAP.2005.844443.

[57] Harrington, R.; Mautz, J. Control of radar scattering by reactive loading. *Antennas and Propagation, IEEE Transactions on*, volume 20, 08 1972: pp. 446 – 454, doi:10.1109/TAP.1972.1140234.

[58] Harrington, R. F. Antenna Excitation For Maximum Gain. *IEEE Trans. Antennas Propag.*, volume 13, no. 6, Nov. 1965: pp. 896–903, doi:10.1109/TAP.1965.1138539.

[59] McKay, M. D.; Beckman, R. J.; et al. A Comparison of Three Methods for Selecting Values of Input Variables in the Analysis of Output from a Computer Code. *Technometrics*, volume 21, no. 2, 1979: pp. 239–245, ISSN 00401706.

[60] Roth, R. *Introduction to Coding Theory*. Cambridge University Press, 2006, doi:10.1017/CBO9780511808968.

[61] Best, M.; Brouwer, A.; et al. Bounds for Binary Codes of Length Less Than 25. *IEEE Transactions on Information Theory*, volume 24, no. 1, Jan. 1978: pp. 81–93, ISSN 0018-9448, doi:10.1109/TIT.1978.1055827.

[62] Gustafsson, M.; Tayli, D.; et al. *Physical bounds of antennas*. 2015, pp. 1–32, doi:10.1007/978-981-4560-75-7_18-1.

[63] Rumelhart, D. E.; McClelland, J. L. *Learning Internal Representations by Error Propagation*. 1987, pp. 318–362.

[64] Han, J.; Moraga, C. The Influence of the Sigmoid Function Parameters on the Speed of Backpropagation Learning. In *Natural to Artificial Neural Computation, Lecture Notes in Computer Science*, volume 930, Springer, 1995, ISBN 3-540-59497-3, pp. 195–201.

[65] Goodfellow, I.; Bengio, Y.; et al. *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[66] Bishop, C. M. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006, ISBN 0387310738.

[67] Kingma, D.; Ba, J. Adam: A Method for Stochastic Optimization. *International Conference on Learning Representations*, 12 2014.

[68] Ruder, S. An overview of gradient descent optimization algorithms. *ArXiv*, 09 2016.

[69] Schmidhuber, J. Deep learning in neural networks: An overview. *Neural Networks*, volume 61, Jan 2015: p. 85–117, ISSN 0893-6080, doi:10.1016/j.neunet.2014.09.003. Available from: http://dx.doi.org/10.1016/j.neunet.2014.09.003

[70] Bergstra, J.; Yamins, D.; et al. Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, JMLR.org, 2013, p. I–115–I–123.

Appendix B

Content of the Enclosed CD

```
AToM........................................directory with antenna toolbox
└── +models
    └── +solvers
        └── +MoM2D............................surface MoM implementation
            ├── +gpu..............................GPU MoM implementation
            │   ├── computeBlockSize.m
            │   ├── createKernel.m
            │   ├── solveIVec_GPU.m
            │   ├── solveTE_GPU.m
            │   ├── solveTM_GPU.m
            │   ├── solveZE0_GPU.m
            │   ├── solveZE_GPU.m
            │   ├── solveZM0_GPU.m
            │   ├── solveZM_GPU.m
            │   └── cuda ............................. MoM CUDA source files
            │       ├── atomicAddTemplate.cu
            │       ├── helpers.cu
            │       ├── ITT.cu
            │       ├── mathematicalFunctions.cu
            │       ├── solveTE.cu
            │       ├── solveTM.cu
            │       ├── solveZE.cu
            │       ├── solveZE0.cu
            │       ├── solveZM.cu
            │       ├── solveZM0.cu
            │       ├── symmetrize.cu
            │       └── transpose.cu
            ├── solveGPU.m...........................a user callable function
            └── RCS_GPU_Benchmark.m..................GPU MoM example
```

`Classification`	directory with basis functions classification
`└── classificationBinary24.py`	binary classification 2×4 structure
`└── classificationBinary48.py`	binary classification 4×8 structure
`└── classificationNClass24.py`	N-class classification 2×4 structure
`└── classificationNClass48.py`	N-class classification 4×8 structure
`└── trainSet24.py`	
`└── trainLabels24.py`	
`└── validationSet24.py`	
`└── validationLabels24.py`	
`└── testSet24.py`	
`└── testLabels24.py`	
`└── trainSet48.py`	
`└── trainLabels48.py`	
`└── validationSet48.py`	
`└── validationLabels48.py`	
`└── testSet48.py`	
`└── testLabels48.py`	