BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

AUTOMATIC TRANSLATION AND COMPARISON OF LUMERICAL AND MEEP SIMULATIONS

AUTOMATICKÝ PŘEKLAD A SROVNÁNÍ SIMULACÍ MEZI MEEP A LUMERICAL

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **DANIEL MAČURA**
AUTOR PRÁCE

SUPERVISOR **Ing. TOMÁŠ MILET, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2025

Abstract

This thesis aims to provide a comparison between the Ansys Lumerical and Meep FTDT simulation tools. A self-designed transpiler from Lumerical Scripting Language to the Meep Python interface was implemented. FDTD simulation were first automaticaly results are compared with analytical results.

Abstrakt

Do tohoto odstavce bude zapsán výtah (abstrakt) práce v českém (slovenském) jazyce.

Keywords

CEM, FDTD, Meep, Ansys Lumerical, Transpiler, Source-to-Source Compiler

Klíčová slova

Sem budou zapsána jednotlivá klíčová slova v českém (slovenském) jazyce, oddělená čárkami.

Reference

MAČURA, Daniel. *Automatic translation and comparison of Lumerical and Meep simulations*. Brno, 2025. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Tomáš Milet, Ph.D.

Automatic translation and comparison of Lumerical and Meep simulations

Declaration

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana X... Další informace mi poskytli... Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Daniel Mačura
April 16, 2025

Acknowledgements

V této sekci je možno uvést poděkování vedoucímu práce a těm, kteří poskytli odbornou pomoc (externí zadavatel, konzultant apod.).

Contents

List of Figures

Acronyms

3D Three Dimensions. 8

ABNF Augmented Backus-Naur form. 17

ASDL Zephyr Abstract Syntax Definition Language. 17

AST Abstract Syntax Tree. 17

BNF Backus-Naur form. 17

CEM Computational Electromagnetics. 5, 6, 8

CFG Context Free Grammar. 18

CFL Courtant-Friedrichs-Lewy. 8

DFA Deterministic Finite Automaton. 13, 18, 19

EBNF Extended Backus-Naur form. 17

EM Electro Magnetic. 6, 9

FDTD Finite-Difference Time-Domain. 6, 8

LL Left to Right, Leftmost Derivation. 18

PEG Parsing Expression Grammar. 17

WSN Wirth Syntax Notation. 17

Chapter 1

Note to the Reader

The following notation conventions apply throughout the thesis if not explicitly stated otherwise. In the context pertaining to physics, symbols in **bold** represent vectors and symbols in *italics* represent scalars, unless stated differently. The del operator denoted by the nabla symbol ∇ is used with the dot product and cross product to denote divergence: $\nabla \cdot v$ and the curl: $\nabla \times v$. The expression „iff“ refers to „if and only if“. In the context of regular languages, lower case Latin alphabet letters, such as $a, b, c, \ldots$, refer to terminal symbols. Upper case Latin alphabet letters, such as $A, B, C, \ldots$, denote nonterminal symbols.

Otherwise fairly standard notation in the applicable fields is used throughout the rest of the thesis.

Chapter 2

Introduction

In modern society, humans are reliant on a multitude of technologies, such as the Internet, mobile phones, television, radio, microwave ovens, camera sensors, lasers, light-emitting diodes, electrical motors, medical imaging systems, and many others. All of these rely on electromagnetic devices; electromagnetism undoubtedly plays a key role in day-to-day life as we know it.

The electromagnetic field theory is the underpinning framework for studying the effects of electromagnetic phenomena at scales where quantum effects are negligible. It studies the interactions between electric charges and currents (Currents are often referred to as electric charges in motion). Maxwell's equations are a series of fundamental coupled partial differential equations that form the cornerstone of classical electromagnetism. However, the closed-form analytical solutions are highly complex and available only for simple cases, making them impractical for most real-world applications.

However, with the rapid increase of available computational power, the use of analytically simple but computationally taxing methods was given an extra boost as they became more available. Compared to their counterparts, these methods provide solutions to more general problems. The branch of electromagnetics that focuses on such methods is termed Computational Electromagnetics (CEM). CEM allows us to simulate more complex problems and verify designs before the production of prototypes. They also provide key insight into the operation of electromagnetic devices and even reveal certain information that may be unattainable by classical analytical methods. Moreover, with the ability to tweak the parameters and re-simulate, CEM has caused the advent of design optimization in electromagnetic devices.

Today, many such tools and software packages exist, some commercial and others open-source. It is important to be able to verify the results against other implementations, which allows the researchers to verify whether the results are mere artifacts of a particular solver or physically meaningful. Furthermore, the tools may offer different features and constraints, and their interoperability is desired. This thesis explores the possibility of translating simulation code between two such tools, Ansys Lumerical and Meep.

Chapter 3

Review of Relevant Literature

This chapter aims to give the reader a broader understanding of the relevant subject matter. It provides an overview of the physics behind the aforementioned simulations and an introduction to the basic concepts of compiler design and the underlying formal language theory. This chapter will not delve into the specifics of each subject at hand but will rather provide the reader with the foundations necessary to comprehend the remaining parts of this thesis.

3.1 Computational Electromagnetics

As this thesis focuses on the implementation of a transpiler, an intricate grasp of CEM is not required; however, a surface-level explanation may benefit the reader, and thus it is provided below.

As explained earlier, CEM is a branch of electromagnetics that focuses on computational methods. This process involves modeling the interactions of Electro Magnetic (EM) fields with objects, typically with the computation of the **E** (electric) and **H** (magnetic) fields or the surface current, in the case of Method of Moments, across the simulation domain. These methods discretize the fields in a step called *meshing*, resulting in the subdivision of the problem domain into many smaller elements. Depending on the simulation method and number of dimensions, this may result in a three-dimensional grid, two-dimensional patches or one-dimensional segments. There are multiple different methods, that each have specific use cases, Finite Element Method is useful when facing arbitrary geometry or objects that do not align properly in the Cartesian grid. Method of Moments is a surface-based method that converts integral forms of Maxwell's equations into matrix equations. Finite-Difference Time-Domain (FDTD) solves Maxwell's equations in the time domain and it is one of the most commonly used methods and also a key part of this thesis, so it will be introduced in more detail in the following sections [1].

[[add image of double slit experiment]]

Curl theorem

[[ask if should omit]]

Divergence theorem

[[ask if should omit]]

Maxwell's equations

In the 19th century, James Clerk Maxwell formulated a set of four equations that underpin classical electromagnetic and form a coherent theoretical structure. Maxwell's equations in the integral form tend to be used in the calculation of symmetric problems, such as finding the electric field of a charged plane, sphere, etc., whereas the differential form is more suitable for the calculation of numerical problems as it is simple to obtain the magnetic and electric fields at a single point. The Ampère-Maxwell law states that magnetic fields are generated by electric currents and a change of the electric field over time. Faraday's law states that a changing magnetic field produces a rotating electric field and the other way around. Gauss's law for electric flux states that electric charges generate an electric field. Gauss's law for magnetic flux postulates the nonexistence of magnetic charge, i.e., magnetic monopoles. In 1931, Dirac proposed that the discovery of magnetic charge (a magnetic monopole) would require a modification of Gauss's law for magnetic flux [2]. **[[ask if this citation is OK]]** Despite extensive searches, magnetic monopoles have yet to be detected experimentally.

Ampère-Maxwell law

$$\nabla \times \mathbf{H} = \mathbf{J} + \frac{\partial \mathbf{D}}{\partial t} \tag{3.1}$$

Faraday's law

$$\nabla \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t} \tag{3.2}$$

Gauss's law for electric flux

$$\nabla \cdot \mathbf{D} = \rho \tag{3.3}$$

Gauss's law for magnetic flux

$$\nabla \cdot \mathbf{B} = 0 \tag{3.4}$$

Where $\mathbf{B}$ is the magnetic field, $\mathbf{E}$ is the electric field, $\mathbf{J}$ the current density, μ_0 the vacuum permeability, ϵ_0 = the vacuum permittivity. It also holds that the speed of light $c = (\epsilon_0 \mu_0)^{-\frac{1}{2}}$.

The above equations are in the differential form, and they are first-order linearly coupled. They also have equivalent integral forms, which may be produced by applying the divergence integral theorem and curl integral theorem. Readers are directed to [3, Chapter 2.4] for more details. **[[ask if okay]]**

Constitutive Relations

Since Maxwell's equations contain more unknowns than equations, as shown above, they are undetermined. This is where constitutive relations between field intensities $\mathbf{E}, \mathbf{H}$ and the flux densities $\mathbf{D}, \mathbf{B}$ come into play. For simple mediums, which are *linear*, *isotropic*, and *non-dispersive*, the following equations hold.

$$\mathbf{D} = \epsilon \mathbf{E} \tag{3.5}$$

$$\mathbf{B} = \mu \mathbf{H} \tag{3.6}$$

$$\mathbf{J} = \sigma \mathbf{E} \tag{3.7}$$

Where ϵ, the permittivity of a medium, is given by the permittivity of free space and the relative permittivity respectively $\epsilon = \epsilon_0 \epsilon_r$. Similarly, μ, the permeability of a medium, is

given by the permeability of free space and the relative permeability $\mu = \mu_0\mu_r$. A medium is often defined by its *constitutive parameters* ϵ, μ, σ. A good illustration of this is that a medium is *isotropic* if ϵ does not change with direction, *homogeneous* if ϵ does not change between two different points in space. Another example is that a material is *linear* if $\mathbf{D} = \epsilon\mathbf{E}$ holds and ϵ does not change in respect to $\mathbf{E}$.

Finite Difference Time Domain

The FDTD method is a staple among finite difference techniques. Finite difference methods are numerical techniques that approximate derivatives directly using finite difference quotients. This class of methods is widespread due to its implied simplicity and is widely used for scientific computation.

It is suitable for problems where the electromagnetic wavelengths, along with the geometries, are comparable to the simulation domain. It is a time domain method, thus it may solve for a broad spectrum in a single simulation in a single pass where it provides a direct numerical approximation of the differential operators in Maxwell's curl equations. The spatial domain is discretized into a staggered grid of *cells*; these will be explained in more depth soon. FDTD also discretizes the continuous time into *time steps*, how-ever there is a limit to the largest time step, where the method stays numerically stable, $\Delta t < h/\left(c\sqrt{3}\right)$ for Three Dimensions (3D), where h and c denote the grid dimension and speed of light in the simulation respectively. This is often referred to as the *Courant limit* after the Courtant-Friedrichs-Lewy (CFL) condition.

The user must keep in mind that such a method will never give a precise answer. The accuracy of the simulation is dictated by the resolution. A general rule of thumb in the CEM community is to use at least 10 samples (cells) per the shortest wavelength. Similarly, Dennis M. Sullivan's book Electromagnetic Simulation Using the FDTD Method states: "A good rule of thumb is 10 points per wavelength. Experience has shown this to be adequate, with inaccuracies appearing as soon as the sampling drops below this rate." [4, p.10].[[**ask if this citation is OK**]]

FDTD provides second-order accuracy with the use of first-order numerical differentiation. Thus the error may be defined as Error $\propto (\Delta h)^2$. For example, in a 3D simulation, if the spatial (and thus time) resolution is increased twofold, the accuracy increases 4 times. This is due to the fact Maxwell's curl equations are discretized using central difference approximations, which are inherently second-order accurate.

Another vital part is the staggered grid it self. Kane S. Yee introduced a system of calculating the $\mathbf{E}$ and $\mathbf{H}$ fields using the aforementioned central differences in an offset pair of grids. The $\mathbf{E}$ fields are defined at the edges of a cell while the $\mathbf{H}$ fields are stored at the centers of the faces, thus offset spatially by $\frac{h}{2}$. However since the $\mathbf{E}$ and

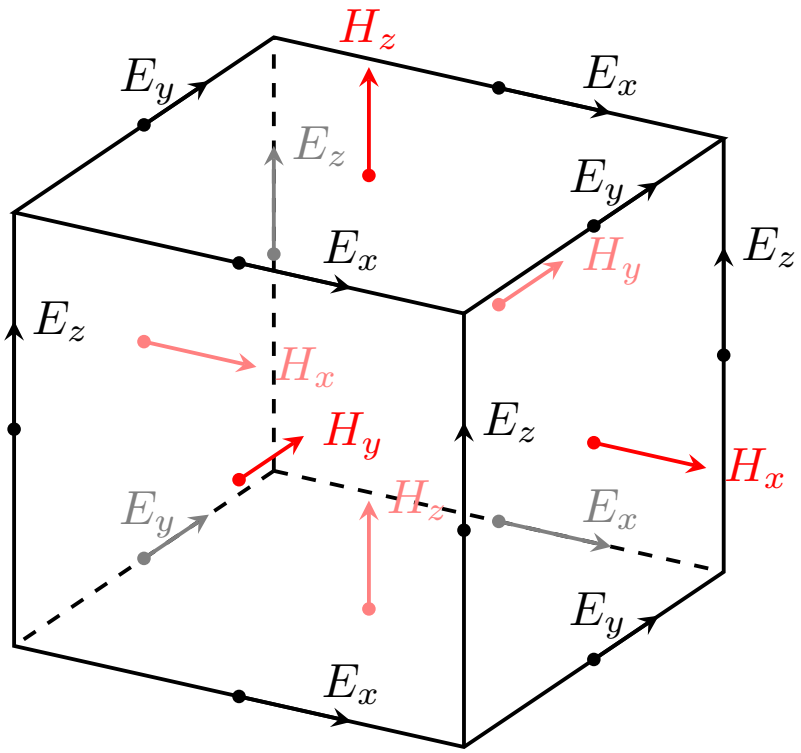

Figure 3.1: Description of the Yee cell.

The basic flow a
[[stair-stepping -> downside]]

3.1.1 Boundary Conditions

3.2 Meep

Meep — MIT Electromagnetic Equation Propagation — is an open-source program for EM simulations.

3.3 Ansys Lumerical

3.4 Formal Language Theory

We use languages like English, Czech, or Slovak in everyday communication exchanges, and these languages are commonly referred to as *natural languages*. The Oxford English Dictionary defines language as a system of spoken or written communication used by a particular country, people or community. How-ever, a more rigorous definition of languages and the tools to operate within them is required. These aspects of language will be tackled in the notion of formal language, which is introduced in the below sections.

3.4.1 Alphabets and Languages

[[add text about languages and alphabets, use English as example]]

Definition 3.4.1 (Alphabet)**.** Let Σ be an *alphabet*, a finite nonempty set of symbols, letters. Then Σ^* defines the set of all sequences w:

$$w = a_1a_2a_3 \dots a_{n-1}a_n, \in \Sigma \text{ for } n \in \mathbb{N}$$

The sequence of symbols w is called a *word*. Word length is given by the number of symbols a, and symbolically annotated as $|w| = n$. The word with a length of 0 is called an *empty word* denoted as ϵ.

Definition 3.4.2 (Language)**.** The set L where $L \subseteq \Sigma^*$ is defined as a *formal language* over the alphabet Σ.

The words $L = \{\epsilon, a, b, aa, ab, bb\}$ are examples of words in language L over the alphabet $\Sigma = \{a, b\}$. Other examples of languages over the alphabet $\Sigma = \{a, b\}$ might include:

- $L_1 = \{\epsilon\}$
- $L_2 = \{a\}$
- $L_3 = \{aaa\}$
- $L_4 = \left\{a^i, b^i; i \in \mathbb{N}\right\}$

Notation conventions

- Iteration
  Let $a^i; a \in \Sigma$ and $i \in \mathbb{Z}$ be the *iteration* of a character a, where $|a^i| = i$.
  Examples bellow:
  - $a^0 = \epsilon$
  - $a^1 = a$
  - $a^2 = aa$
  - $a^i = a_0a_1a_2 \dots a_i; i \in \mathbb{N}$
- Concatenation
  Let $w \cdot w^{'}; w, w^{'} \in \Sigma^*$ be the *concatenation* of the words w and $w^{'}$.
  $w = a_1a_2a_3 \dots a_n; w^{'} = a_1^{'}a_2^{'}a_3^{'} \dots a_m^{'}; n, m \in \mathbb{Z}$ then
  $w \cdot w^{'} = ww^{'} = a_1a_2a_3 \dots a_na_1^{'}a_2^{'}a_3^{'} \dots a_m^{'}$
- **[[Kleene star]]**
- Symbol count
  The number of occurrences of a in w, where $a \in \Sigma, w \in \Sigma^*$, noted as $|w|_a$.

3.4.2 Grammars

Linguists refer to grammar as a set of surface level and deep level rules that specify how a natural language is formed. Due to the polysemous and homonymous nature of natural languages, it is not suited for the description of unambiguous systems. The inherit need to strictly describe languages introduced various language defining mechanisms. Many of these mechanisms are interchangeable, and may describe the same languages. How ever, not all mechanisms are able to describe languages formed by other mechanisms.

The concept of a grammar is a powerful tool for describing languages [5, p. 52]. A simple sentence in English consists of a single independent clause. A clause is typically formed by a subject and a predicate. This can be written as follows.

$$\langle clause \rangle \rightarrow \langle subject \rangle \langle predicate \rangle$$

We can further define the $\langle subject \rangle$ and $\langle predicate \rangle$. One of the possible subjects is a noun phrase, and the predicate may be a simple verb.

$$\langle subject \rangle \rightarrow \langle determiner \rangle \langle premodifier \rangle \langle noun \rangle \langle postmodifier \rangle$$

$$\langle predicate \rangle \rightarrow \langle verb \rangle$$

Associating the $\langle determiner \rangle$ with the article „the“, $\langle premodifier \rangle$ with „fast“ or „slow“, $\langle noun \rangle$ with „athlete“, $\langle postmodifier \rangle$ to „from England“ or to an empty string and finally associating the $\langle verb \rangle$ to „won“ or „lost“ allows us to define a pattern capable of generating an infinite number of clauses/sentences such as „The fast athlete from England won“ or „The slow athlete lost“, which testifies to the principle of Chomskian generative grammar.. These sentences are considered to be *well formed* as far as grammar is concerned, as they resulted from the implementation of grammatical rules.

The premise is to consecutively replace the $\langle clause \rangle$ until only irreducible blocks of the language remain. Generalizing this idea brings about the concept of formal grammars.

Definition 3.4.3 (Grammar)**.** [6] Let an ordered quadruple G define a grammar such that: $G = (N, \Sigma, P, S)$, where:

1. N is a finite set of *nonternminal* symbols
2. Σ is an alphabet, i.e. a finite set of *terminal* symbols, such that $N \cap \Sigma = \varnothing$
3. P is a finite set of rewriting rules known as *productions*, ordered pairs (α, β). P is a subset of the cartesian product of $\alpha = (N \cup \Sigma)^* N (N \cup \Sigma)^*$ and $\beta = (N \cup \Sigma)^*$

   The productions are denoted as $\alpha \rightarrow \beta$. If there are multiple productions with the same left hand side (α), we can group their right hand sides (β).

   $\alpha \rightarrow \beta_1, \alpha \rightarrow \beta_2$ may be written as $\alpha \rightarrow \beta_1 | \beta_2$
4. S is the starting symbol of the grammar G, where $S \in N$

Productions $\alpha \rightarrow \beta$ symbolize, that given the words $V, W, x, y \in (N \cup \Sigma)^*$, the word V can be rewritten as follows: $V \Rightarrow W$ iff there are words, which satisfy the following condition, $V = x\alpha y \wedge W = x\beta y$ and $\alpha \rightarrow \beta \in P$.

Definition 3.4.4 (Derivation)**.** $V \overset{*}{\Rightarrow} W$ iff there is a finite set of words

$$v_0, v_1, v_2, \ldots, v_z; z \in \mathbb{Z}$$

such that $v_0 = V$ and $v_z = W$ where each is rewritten from the previous word. Such a sequence of applications of productions is called a derivation. The length of a derivation is given by z.

Grammars are often represented using formalisms that, often set restrictions on the left and right hand sides of productions. These restrictions further impose limits on the set of languages a grammar may produce; this is known as the *expressive power* of a grammar.

3.4.3 Chomsky hierarchy

When working with formal grammars, the need to compare their expressive power arose. Linguist Noam Chomsky introduced the so called *Chomsky hierarchy* [7]. A set of four classes, each more expressive than the previous, see fig. 3.1.

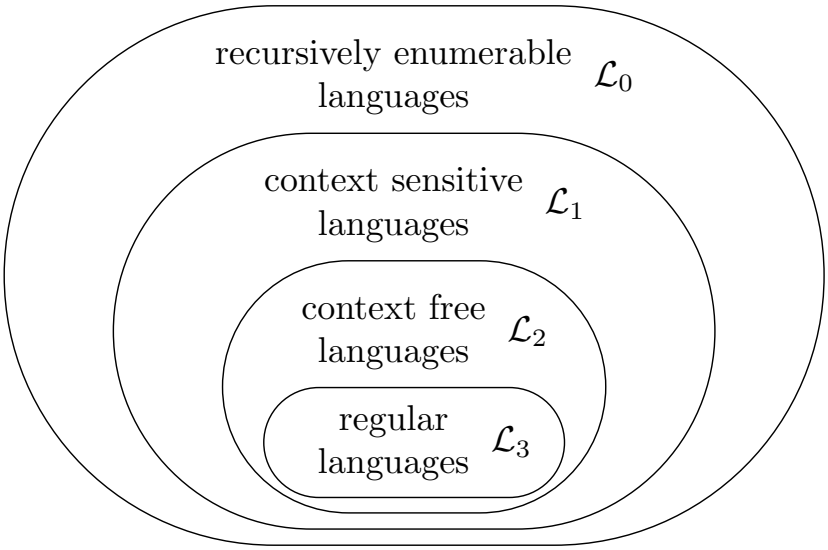

Figure 3.2: Description of the Chomsky hierarchy.

The intricacies of each of said classes are not necessary in the context of this thesis, how-ever regular languages will be used heavily throughout the rest of this thesis, and explained more in depth.

Grammar	Language	Restrictions
$\mathcal{L}_3$	Regular	[[**todo**]]
$\mathcal{L}_2$	Context Free	
$\mathcal{L}_1$	Context Sensitive	
$\mathcal{L}_0$	Recursively Enumerable	

Table 3.1: An overview of the Chomsky hierarchy classes.

3.4.4 Context Free Languages

LL Languages

3.4.5 Regular Languages

As shown above, regular languages are the inner most part of the Chomsky hierarchy, thus they are the most constricted. How-ever, regular languages play a crucial role in lexical analysis, more precisely, in pattern matching. These constrictions lead to many useful closure properties and decisability properties, mainly membership, which will be introduced shortly.

Regular languges may be defined in multiple equivalent manners such as through finite automata or regular expressions.

Finite Automata

Definition 3.4.5 (Deterministic Finite Automata)**.** DFAs are formally defined as 5-tuples

$$M = (Q, \Sigma, \delta, q_0, F),$$

where Q is a finite set of states. Σ is a finite set of symbols, also known as an input alphabet. δ is a transition function between states defined as $\delta : Q \times \Sigma \to Q$. q_0 is a initial state, $q_0 \in Q$. And F is a set of final states, $F \subseteq Q$.

If starting in the state q_0 and moving left to right over the input string such that during each move a single symbol is consumed from the input string and a transition function for the corresponding state and symbol exists, and all symbols from the input string have been read and the Deterministic Finite Automaton (DFA) stopped in a final state, the string is deemed accepted. This is called a deterministic finite acceptor.

Regular Expressions

Definition 3.4.6 (Regular Expressions)**.**

[[show equivalence DFA = NFA = $\mathcal{L}_3$]]

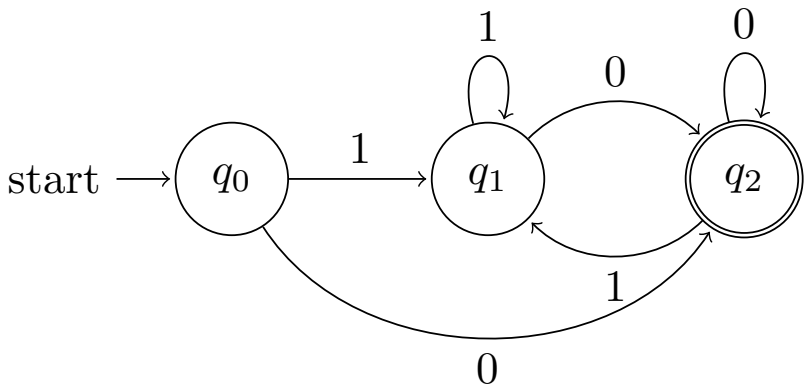

Figure 3.3: Description of a compiler as a single unit.

3.5 Compilers

Today's world relies on software more than ever before. It is imperative for developers to be able to write software efficiently. Software today is written in programming languages, high-level human-readable notations for defining how a program should run. However, before a program can be run on a system, it has to be translated (or compiled) into low-level machine code, which computers can run. The computer program that facilitates this translation is called a *compiler*. See fig. 3.3.

Figure 3.4: Description of a compiler as a single unit.

Compilers are complex programs. It is helpful to break them down into parts, each handling different tasks, which are chained together to form a compiler. Modern compilers are composed of many phases, such as the Lexical Analyzer, Syntax Analyzer, Semantic Analyzer, Intermediate Code Generator, Machine-Independent Code Optimizer, Code Generator, Machine-Dependent Code Optimizer [8, p. 5], however this chapter covers the components relevant to this thesis. See the relevant stages in fig. 4.0.

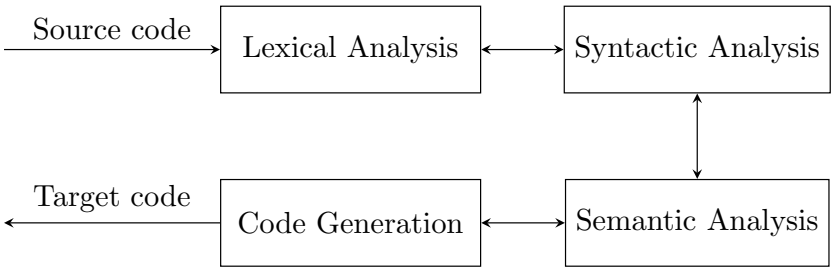

Figure 3.5: Overview of relevant compiler stages.

Lexical Analysis

The first stage of a compiler is lexical analysis, also known as a *lexer* or *tokenizer*. For the remainder of this thesis, the term *lexer* will refer to the lexical analysis stage of a compiler. The lexer consumes a stream of characters, or the *source code*, and returns a stream of *tokens*. A *token* is a lexically indivisible unit, for example, the Python keyword `return`, cannot be divided any further, e.g. into `re turn`. Each token is comprised of characters. The rule that defines which combination of characters constitutes a given token is called a *pattern*. The sequence of characters matching a pattern is called a *lexeme*, which is stored along with the token as a value.

Regular Expressions

Regular expressions, which were introduced in definition 3.4.6 are a compact way to represent the patterns accepting tokens. Regular expressions are an algebraic definition of patterns; they specify *regular languages*, $\mathcal{L}_3$

Syntactic Analysis

LL Parser

Semantic Analysis

Code Generation

Chapter 4

Transpiler Implementation

This chapter introduces the implementation of the source to source compiler designed to convert Lumerical's scripting language to Python combined with the Meep library.

4.1 Choice of Tools

When writing any program, it is important to choose the right tools for the job. This comes down not only to choosing the programming language but also what libraries and packages to utilize and which components to implement to better suit the project's constraints. The source language, Lumerical's scripting language, allows you to automate tasks and analysis such as manipulating simulation objects, launching simulations, and analyzing results [9]. This language how-ever, is not suitable for implementing a transpiler. Looking at the target language, Python is a general-purpose high-level interpreted language. Writing the transpiler in Python will allow for easier editing and debugging due to it's high-level nature. Consequently, and owing to the fact that the implementation language is the same as the target language, this will also allow us to use built in modules for generating the target code. The Python's ecosystem also includes the tools such as Sphinx and Pytest for writing technical documentation and tests respectively.

4.1.1 Sphinx

Sphinx is a tool that automatically generates documentation by converting plain text source files into multiple output formats [10]. Sphinx was chosen because it facilitates the extraction of docstring style comments from the Python code, which may be enriched by the addition of ReStructured Text [11]. Multiple output formats such as Portable Document Format (PDF), LaTeX source code and Hypertext Markup Language (HTML), are supported. Sphinx also includes multiple extensions which allow the parsing of LaTeX into Scalable Vector Graphics (SVG), which are easily rendered in the web.

This is instrumental in providing the user with the necessary insight into the technical operation of the transpiler, and allows us to search and view information in a concise manner.

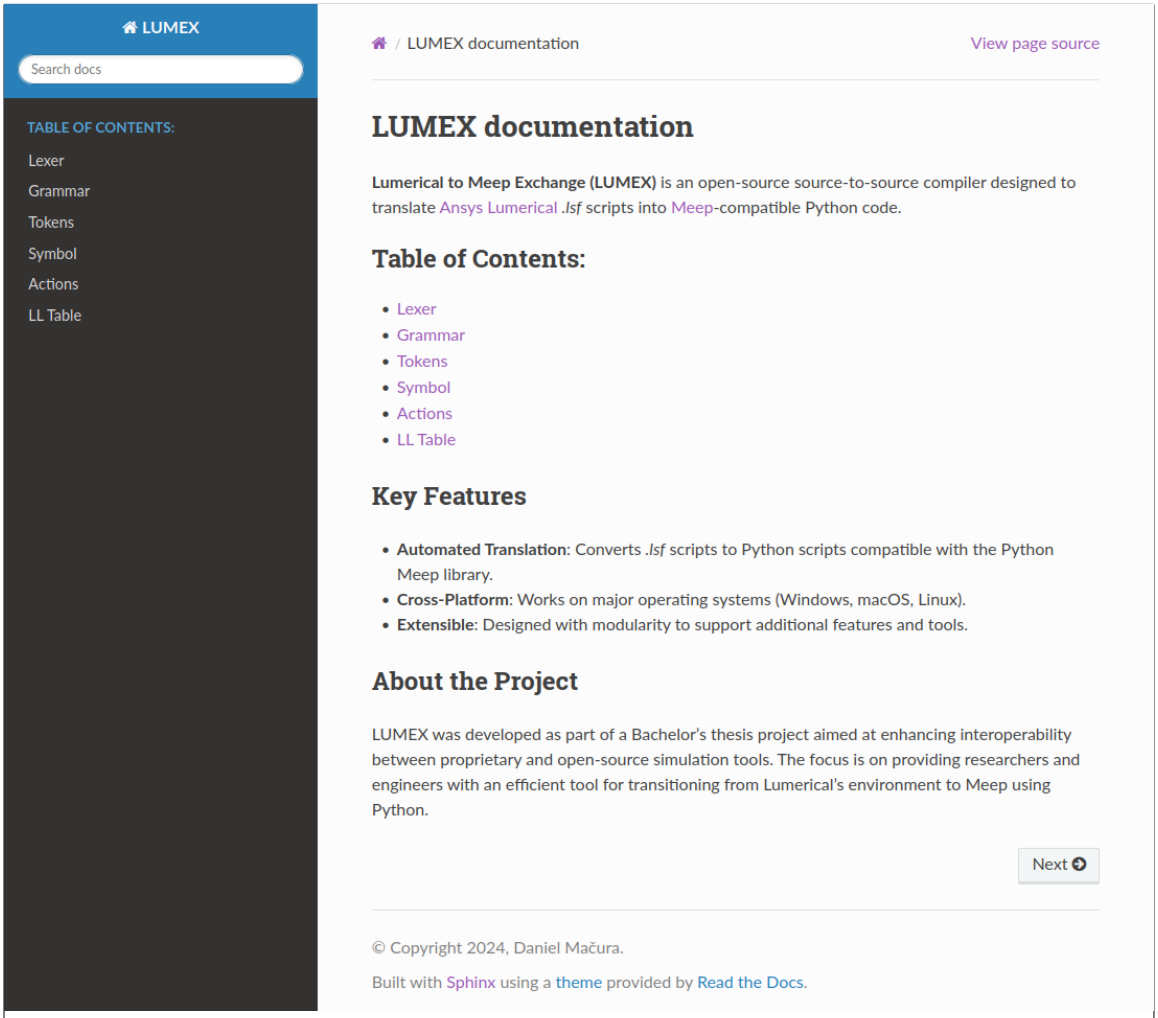

Figure 4.1: Screen-capture of the resulting documentation.

4.1.2 Pytest

Pytest is a widely used testing framework for Python.

[[mention python, sphinx, pytest, explain motivation for implementation of own transpiler]]

4.2 Grammar

Before being able to translate between two languages, it is paramount to understand their grammar. Many languages provide definitions of grammars in a metasyntax format such as Backus-Naur form (BNF), Extended Backus-Naur form (EBNF), Wirth Syntax Notation (WSN), Augmented Backus-Naur form (ABNF), Parsing Expression Grammar (PEG) or Zephyr Abstract Syntax Definition Language (ASDL) [12]. The Python grammar is defined by a mixture of EBNF and PEG as used in the CPython parser [13]. Additionally, the Python Abstract Syntax Tree (AST) module also defines an abstract grammar in ASDL [14].

As, of writing this thesis, Lumerical does not publicly provide any such specification for the Lumerical's scripting language. Thus, it is necessary to analyze the language and derive such a grammar that covers the relevant subset of the language to the scope of this thesis.

Since this thesis focuses primarily on nanophotonic simulation workflows, only essential commands to set up a simulation will be included, e.g. commands to add blocks, sources, monitors and commands to translate and scale the objects. There may be other parts of Lumerical's scripting language that could be utilized for a nanophotonics simulation, however those are considered out of the scope of this thesis due to the broad range of Lumerical's scripting language with around 800 commands and keywords [15]. While Lumerical's scripting language includes hundreds of commands spanning optimization, analysis, and post-processing (e.g., `runoptimization`, `fitlorentzpdf`, `exportcsvresults`), this work prioritizes the core geometry and simulation workflow: `addrect` (block creation), `addgaussian`/`addfdtd` (source/solver definition), `addpower` (monitor placement), and `set` (property configuration).

[[provide steps to reverse-engineer grammar]] There are infinitely many distinct grammars that describe the same language. By example, in a Context Free Grammar (CFG) given as $S \to aB$, one may add an intermediate rule, producing $S \to aC, C \to B$. This process may be repeated indefinitely creating new grammars that describe the same language. To this end, the grammar described bellow is just one of the plethora of possible grammars describing a subset of Lumerical's scripting language. While constructing the grammar, it is beneficial to impose additional constrains and only utilize a subset of CFGs, namely a Left to Right, Leftmost Derivation (LL) grammar. This imposes certain rules as described before, informally speaking, when a parser arrives at a nonterminal, is must be able to decide which production to apply by peeking at at most the next k symbols, this is an LL(k) grammar. **[[collecting samples]]** It is important to first collect samples of code, to study the syntactic structure and the semantics of the code. Examples were taken from the official code

[[observation of keywords, syntactic structures -> tokens]] Lumerical provides a list of all commands [15] **[[formalize rules]]**

[[add grammar listing here]]

4.3 Lexical Analysis

After defining a grammar, the first stage in creating a transpiler is the lexer. There are multiple methods in which a lexer may be modeled, the most straight-forward being the use of a DFA formalism and a loop over all input characters while checking if there is a valid production from the current character to the look-ahead character and greedily continue and backtrack to the last successful match upon not being able to find an applicable production.

Algorithm 1: State Machine Based Lexer

```
current_state ← None
last_accepting_state ← None
while lookahead_character != EOF do
    if has_transition(current_state, lookahead) then
        current_state ← next_state(current_state, lookahead)
        get_next_character()
        if is_accepting_state(current_state) then
            last_accepting_state ← current_state
    else
        if no_accepting_state_visited then
            return None ;                                   // Failure
        else
            revert_to_previous_accepting_state_and_revert_input_characters
            return current_state ;                          // Success
```

This implementation is fine, how-ever, with an ever-growing number of states, which have to be hard-coded; the solution may become rather complex. Indeed, a cleaner method exists. Since any DFA may be transformed into a regular expression, it is sufficient to try to match all regular expressions against the input and return the longest match while removing it from the input.

Algorithm 2: Regular Expression Based Lexer

```
longest_match ← None
for all_regular_expressions do
    if regular_expression_matches_input then
        if regular_expression_is_longer_than_longest_match then
            longest_match ← current_regular_expression
if longest_match != None then
    return matching_token remove_matched_lexeme_from_input
else
    return None ;                                           // Failure
```

This approach is not only more straight forward, but also allows modeling the lexer in a more pythonic manor. Each token is represented as a unique class with holds the regular expression pattern matching the corresponding token. Each specific token class inherits a common token class, after that, it is sufficient for the lexer to enumerate over all children of the token class and search for the one with the longest match resulting in a greedily matched token.

[[mention lexer regex implementation]]

4.4 LL Parser

[[mention LL table calculation according to theory]]

4.5 Code Generation

[[]] [[handling context specific actions such as the selection -> introduction of runtime class]] [[write about transformation of ast]]

Chapter 5

Evaluation of Results

[[talk about methodology]] [[show comparison of leumerical code, generated python and handwritten -> conclusion probably not best method]] [[test examples analytic vs FDTD]]

Chapter 6

Conclusion

[[compare lumerical and meep]] [[test result TBD]]

Bibliography

[1] Davidson, D. B. *Computational Electromagnetics for RF and Microwave Engineering*. Cambridge University Press, Oct 2010. ISBN 9781139492812.

[2] Dirac, P. A. M. Quantised Singularities in the Electromagnetic Field. *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character*. Royal Society, 1931, vol. 133, no. 821, p. 60–72.

[3] Staelin, D. H.; Morgenthaler, A. and Kong, J. A. *Electromagnetics and Applications*. MIT OpenCourseWare, 2009. Available at: <https://phys.libretexts.org/Bookshelves/Electricity_and_Magnetism/Electromagnetics_and_Applications_(Staelin)>.

[4] Sullivan, D. M. *Electromagnetic Simulation Using the FDTD Method*. 2ndth ed. Piscataway, NJ: IEEE Press, 2013. ISBN 9781118588784.

[5] Peter, L. *An Introduction to Formal Languages and Automata*. Jones & Bartlett Publishers, january 2016. ISBN 9781284077247. Available at: <https://books.google.com/books/about/An_Introduction_to_Formal_Languages_and.html?hl=&id=pDaUCwAAQBAJ>.

[6] Salomaa, A. *Formal languages*. Boston: Academic Press, 1987. ISBN 0126157502. Available at: <https://openlibrary.org/books/OL2373365M/Formal_languages>.

[7] Chomsky, N. Three models for the description of language. *IRE Transactions on information theory*. IEEE, 1956, vol. 2, no. 3, p. 113–124.

[8] Aho, A. V.; Lam, M. S.; Sethi, R. and Ullman, J. D. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. 2.th ed. USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN 0321486811.

[9] Ansys Optics. *Lumerical scripting language* <https://developer.ansys.com/docs/lumerical/scripting-language>. 2025. Accessed: March 30, 2025.

[10] Sphinx Documentation Team. *Quickstart — Sphinx Documentation* <https://www.sphinx-doc.org/en/master/usage/quickstart.html>. 2025. Accessed: March 30, 2025.

[11] Docutils Project. *ReStructuredText Markup Specification* <https://docutils.sourceforge.io/rst.html>. 2025. Accessed: March 30, 2025.

[12] Wang, D. C.; Appel, A. W.; Korn, J. L. and Serra, C. S. The Zephyr abstract syntax description language. In: *Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL), 1997*. USA: USENIX Association, 1997, p. 17. DSL'97. Available at: <http://www.usenix.org/publications/library/proceedings/dsl97/full_papers/wang/wang.pdf>. Accessed: March 31, 2025.

[13] Python Software Foundation. *Full Python Grammar Specification*. 2025. Available at: <https://docs.python.org/3/reference/grammar.html>. Accessed: March 31, 2025.

[14] Python Software Foundation. *Python ast — Abstract Syntax Trees*. 2025. Available at: <https://docs.python.org/3/library/ast.html>. Accessed: March 31, 2025.

[15] Ansys Optics. *Lumerical scripting language commands* <https://optics.ansys.com/hc/en-us/sections/360005285973-Commands>. 2025. Accessed: April 5, 2025.