UPPSALA
UNIVERSITET

ISRN UTH-INGUTB-EX-E-2025/001-SE
Examensarbete 15 hp
June 2025

Framework for Rotating Electrical Machines using FEM

Elias Lunderbye

Högskoleingenjörsprogrammet i elektroteknik

UPPSALA
UNIVERSITET

Framework for Rotating Electrical Machines using FEM

Elias Lunderbye

Abstract

This project addresses the modernization and porting of a finite-element-method framework for rotating electrical machines from Ubuntu 16 to Ubuntu 24. The codebase comprising of C, C++, Fortran, Lex and Yacc relied on outdated compiler behaviors and build scripts that are incompatible with modern toolchains. The goal was to update and replace K&R-style declarations with modern ANSI-style, legacy Fortran syntax, incorrect header files and resolve linker errors between different working directories. Lex/Yacc syntax were recompiled with Flex/Bison and build scripts were used to speed up the compiling process. The updated framework now compiles on Ubuntu 24, producing the interactive GUI (AceMain) and analysis routines (Acek01). Simulations for magnetic field plotting and mesh visualization run without error. This successful port enables continued use of the framework for two-dimensional steady-state FEM analysis on modern systems.

Teknisk-naturvetenskapliga fakulteten

Uppsala universitet, Utgivningsort Uppsala/Visby

Handledare: Urban Lundin Ämnesgranskare: Irina Temiz

Examinator: Andrej Savin

Contents

Chapter 1

Introduction

1.1 Purpose and Goals

The aim of this project is to analyze and update an existing framework for simulating rotating electrical machines using Finite Element Method (FEM). The framework is outdated and was previously functional up to Ubuntu 16 but is no longer compatible with newer versions such as Ubuntu 24.

To address this issue, we will investigate the framework's source code which includes multiple programming languages such as C, C++, Fortran, Yacc and Lex. The objective is to identify the root cause of the incompatibility with Ubuntu 24 and propose solutions to restore and improve the framework's functionality on modern systems.

1.2 Limitations

Prior to commencing this project, we recognized that the existing framework is both expansive in scope and complex in its internal structure, presenting numerous challenges in terms of modernization. To ensure a manageable workload and meet our project timeline, we have deliberately focused our efforts solely on restoring basic functionality under Ubuntu 24. Specifically:

- **Serial Execution Only:** We will not implement or evaluate parallel-processing optimizations in this project. Introducing multi-threading or distributed computation would require extensive redesign of the code base substantially increasing both development time and verification efforts.
- **Stationary Simulations:** The framework's present architecture supports only steady-state electromagnetic analyses. Time-dependent

(transient) simulations, while important for certain dynamic applications are beyond the current project's scope.

Chapter 2

Background

2.1 Rotating Electrical Machines

Given that the finite element method provides a powerful framework for predicting electromagnetic behavior in complex geometries, it is essential to ground our FEM analysis in the physical principles of the machines we are studying. Rotating electrical machines, both motors and generators, derive their function from the precise distribution of magnetic flux and currents in the stator and rotor, which FEM is uniquely suited to capture. By simulating how field windings, core materials, and air gap geometries interact under load, FEM allows us to optimize torque, efficiency, and thermal performance long before a prototype is built. In other words, understanding the core factors that govern machine performance (magnetic flux paths, winding configurations, saturation of steel, and so on) is what makes FEM not just a numerical tool, but a critical design enabler for modern applications, from grid-scale power plants to electric vehicles and railway traction.

2.1.1 Operating Principles

An electric motor consist of two mechanical components: the rotor, which rotates, and the stator, which remains stationary. The rotor is the moving part that delivers the mechanical power. It typically holds windings carrying currents; in a non–permanent-magnet machine, these rotor currents interact with the stator's magnetic field to produce torque that turns the shaft [1]. The stator is the stationary component that both channels energy to and from the rotor, supplying the magnetic field as a motor and as a generator having its own windings cut by the rotor's rotating magnetic field so that an electromotive force (EMF) is induced [2]. The air gap between these allow the rotor to turn. In general the air gap should be kept as small as possible to achieve the best performance but this also comes with its respective drawbacks. A small air gap may create friction in addition to noise while a large air gap leads to weakened performance [3].

To generate electricity the motor is made up of two elements, the field magnets and the armature. The typical configurations consists of the field magnets being places on the stator and the armature on the rotor, though their roles can be swapped. Together, these create a magnetic circuit [4]. The magnets can be either electromagnets or permanent magnets. They create a magnetic field that passes through the armature. The armature typically consists of wire shaped into a coil, this conductor wound in successive turns of insulated wire form an inductor that produces a magnetic field when energized [5].

In brief, the rotor and stator with their air gap form the magnetic circuit powered by field magnets and the armature winding. These elements together enable the conversion between electrical and mechanical energy.

2.1.2 Common Machine Types

Rotating electrical machines can be classified into several broad families. The three most common types are presented below.

Induction machines or asynchronous machines are widely used in the current industry. In a three-phase motor, a rotating magnetic field produced in the stator induces currents in the short-circuited conductors of the rotor. The interaction between stator field and induced rotor currents produces torque. Because the rotor always turns slightly slower than the stator field, these machines are inherently self-starting. Induction generators operate on the same principle when driven above synchronous speed and are often used in wind turbines and small hydropower plants [6].

In synchronous machines, the magnetic field of the rotor, produced by direct current excitation in a wound rotor or by permanent magnets, is locked in speed to the rotating stator field. This "synchronous" operation enables precise speed control and correction of the power factor. The two main subtypes are: Salient-pole synchronous machines, which feature large poles projecting from the rotor and are common in hydroelectric generators and Cylindrical-rotor machines, with smooth cylindrical rotors and distributed field windings, typically employed in high-speed steam-turbine generators [6].

DC machines convert between electrical and mechanical energy via a commutator and brushes that reverse current in the rotor windings each half-turn. There are three classic DC-machine configurations: series (high starting torque), shunt (good speed regulation), and compound (mixed characteristics) [6].

2.1.3 Maxwell's Equations

When engineers build a FEM model of a motor or generator, all of Maxwell's equations are solved together.

$$\nabla \cdot \mathbf{E} \;=\; \frac{1}{\varepsilon_0}\,\rho, \tag{2.1}$$

Gauss's law states that the net electric flux through any closed surface is proportional to the total charge enclosed. In a FEM model, this ensures that charge distributions in conductors or dielectrics produce the correct electric field everywhere in the machine [7].

$$\nabla \cdot \mathbf{B} \;=\; 0, \tag{2.2}$$

Gauss's law for magnetism indicates that there are no isolated magnetic charges, magnetic field lines $\mathbf{B}$ always form closed loops. In a motor or generator, this condition enforces continuity of the magnetic flux around the iron, air-gap, and return paths [7].

$$\nabla \times \mathbf{B} = \mu_0\,\mathbf{J}, \tag{2.3}$$

Using Ampère's Law, electric currents in the stator and rotor windings create the magnetic fields that drive the machine. By arranging and energizing these windings, we shape the field in the air-gap to produce smooth torque in a motor or steady voltage in a generator [7].

$$\nabla \times \mathbf{E} = -\,\frac{\partial \mathbf{B}}{\partial t}, \tag{2.4}$$

Using Faraday's law, whenever the magnetic field through a coil changes, whether the rotor is spinning or the current in the stator is alternating, a voltage is induced in that coil. In a generator, the induced voltage serves as the output. In a motor, applying voltage to the stator produces a rotating magnetic field that drives the rotor [7].

$$\mathbf{F} = q\Big(\mathbf{E} + \mathbf{v} \times \mathbf{B}\Big), \tag{2.5}$$

Once you have a magnetic field and currents flowing in conductors, the interaction between them generates a force that can be calculated using Lorentz law. In a motor, these forces around the air-gap add up to produce the torque that turns the shaft. In detailed FEM models, we calculate this force directly from the field solution [7].

Therefore, charge distributions determine the electric field (Gauss's law), and the absence of magnetic monopoles enforces that magnetic flux lines form closed loops (Gauss's law for magnetism). Currents tell us the field (Ampère's law) and the changing field tells us the voltages (Faraday's law). Finally, the combined electric and magnetic fields produce forces on moving charges (Lorentz's law). By linking that electromagnetic solution to

thermal and mechanical analyses, we can predict losses, temperature rise, structural stresses and then adjust the design and material choices to optimize electromagnetic performance while satisfying thermal, mechanical and manufacturability constraints.

2.2 Finite Element Method (FEM)

Designing high-performance rotating machines demands a multi-physics approach that couples electromagnetic, thermal, and mechanical analyses. The Finite Element Method (FEM) is the tool of choice: by discretizing the machine geometry into an interconnected mesh of small elements, FEM solvers compute detailed magnetic field distributions and estimate core and winding losses. These electromagnetic loss maps serve as input to thermal models, which predict temperature gradients and evaluate cooling efficacy. Mechanical simulations using the same or adapted meshes then assess stress concentrations, deformation under load, and modal frequencies. Iterating this process within an optimization loop allows engineers to adjust geometry, material selection, and winding configurations in order to maximize torque density, minimize losses, or meet specified thermal limits, all while ensuring structural robustness [8].

Below, the two-dimensional finite-element (FE) field model for electrical machine analysis is presented. We begin with the partial differential equations, proceed to their variational (energy) formulation, describe the discretization, and then discuss material property assignment and boundary condition treatment.

Governing Equations in Finite-Element Field Modeling

In the 2-D magnetostatic formulation the magnetic field is expressed via the axial component of the vector potential, $A_z(x, y, t)$, such that

$$B_x = \frac{\partial A_z}{\partial y}, \quad B_y = -\frac{\partial A_z}{\partial x}, \quad B_z = 0. \tag{2.6}$$

Applying Ampère's law and Ohm's law in conductor regions yields the following PDE for each subdomain [9]:

Conducting subdomains:

$$\frac{\partial}{\partial x}\Big(\nu \frac{\partial A_z}{\partial x}\Big) + \frac{\partial}{\partial y}\Big(\nu \frac{\partial A_z}{\partial y}\Big) = \sigma \frac{\partial A_z}{\partial t} + \sigma \frac{\partial V}{\partial z} \tag{2.7}$$

Non-conducting regions (iron, air):

$$\frac{\partial}{\partial x}\Big(\nu \frac{\partial A_z}{\partial x}\Big) + \frac{\partial}{\partial y}\Big(\nu \frac{\partial A_z}{\partial y}\Big) = 0 \tag{2.8}$$

Here,

$$\nu = \frac{1}{\mu_r \mu_0} \tag{2.9}$$

is the reluctivity, μ_0 free-space permeability, μ_r relative permeability, σ the electrical conductivity, and V the applied electric potential. The right-hand side of (2.7) represents the total current density: the induced term $\sigma\, \partial A_z/\partial t$ and the applied term $\sigma\, \partial V/\partial z$ [9].

Variational (Energy) Formulation

To derive a FE solution, (2.7)–(2.8) are recast as a stationary point of the electromagnetic energy functional $\mathcal{F}$:

$$\mathcal{F}[A_z] = \iint_S \left(\int_0^B \mathbf{H} \cdot d\mathbf{B} \; - \; J\, A_z \right) dS \tag{2.10}$$

where S is the 2-D calculation domain, $\mathbf{H}$ the field intensity, $\mathbf{B}$ the flux density, and J the total current density. A trial solution for A_z is expressed in terms of nodal coefficients $A_j(t)$ and base functions $\vartheta_j(x, y)$:

$$A_z^*(x, y, t) = \sum_{j=1}^{N} A_j(t)\, \vartheta_j(x, y). \tag{2.11}$$

Substituting (2.11) into (2.10) and enforcing $\delta\mathcal{F} = 0$ yields the weak form from which the FE equations are assembled [9].

Finite-Element Discretization

The calculation geometry is partitioned into small, non-overlapping triangular elements. Each element's vertices define the mesh nodes, and base functions ϑ_j are chosen to be nonzero only within their associated element (typically first- or second-order polynomials in x, y). By applying the variational form element-wise and assembling over all elements, one obtains a system of ordinary differential algebraic equations in the unknown nodal potentials $A_j(t)$. Time integration (e.g., implicit Euler) is then used for the induced currents in conducting regions [9].

Material Property Assignment

The FE calculation geometry is subdivided into distinct subdomains, each assigned appropriate electromagnetic properties:

- **Iron cores (rotor, stator):** Nonlinear ferromagnetic material characterized by single-valued $B-H$ curves $\mu_r(B)$.
- **Conductors (field winding, stator winding):** Constant electrical conductivity σ. Windings are modelled as filamentary conductors with uniformly distributed current density per time step.

- **Air (air-gap, exterior):** Vacuum properties ($\mu_r = 1$, $\sigma = 0$).

Nonlinear subdomains require iterative updating of element reluctivities at each time step [9].

Boundary Conditions

To complete the field problem, the external boundaries of the reduced pole-pitch model are constrained as follows [9]:

1. **Homogeneous Dirichlet:** $A_z = 0$ on the outer iron boundary, simulating a perfect magnetic insulator at infinity.
2. **Periodic (odd) condition:** Opposite radial boundaries satisfy $A_z^{\text{upper}} = -A_z^{\text{lower}}$ to enforce anti-symmetry of one pole pitch.
3. **Sliding-mesh interface:** In the air-gap, separate rotor and stator meshes are linked via an interpolation algorithm at each time step, modelling relative rotation without remeshing.

2.3 Ubuntu VM Framework

All of the framework's source code, mesh definitions, solver binaries, and post-processing scripts reside on an Ubuntu installation, making it the natural home for this FEM-based rotating electrical machine framework. By running Ubuntu inside a Oracle VM VirtualBox, we get access to all Linux utilities—shell scripting, packages, and file-system tools that the framework's build and execution workflows depend upon. Attempting to force such a complex project into Windows would not only complicate installation but also risk breaking key components such as path conventions, linking libraries and shell environments that assume a UNIX-based OS.

Moreover, our overarching research goal is to migrate the legacy code from its original Ubuntu 16 environment to Ubuntu 24. By maintaining two parallel VMs—one running different OS versions we can then directly compare build logs and runtime errors. When algorithms that once executed flawlessly on Ubuntu 16 fail under Ubuntu 24, we capture precise error messages, library mismatches, or API deprecations, then iterate on code modifications or adjust compiler flags until compatibility is restored.

Chapter 3

Theory

3.1 LINUX Interface

The framework runs on a Virtual Machine using Ubuntu. We are mostly interested in the console were we can access all the files for the framework. To navigate the console and perform different actions there are several commands that the user can use. Below we will list the most common commands used during the project:

Table 3.1: Common Linux Commands

Command	**Description**	**Example**
`ls`	List directory contents	`ls */*.h`
`cd`	Change current working directory	`cd IPL/bin`
`mkdir`	Create a new directory	`mkdir IPL_Backup`
`cp`	Copy files or directories (use `-r` for directories)	`cp file.y */yaccTest`
`mv`	Move or rename files and directories	`mv oldname.txt newname.txt`
`rm`	Remove files or directories (use `-r` for directories)	`rm -r IPL_Backup`
`cat`	Concatenate and print file contents	`cat GÖR_SÅ_HÄR`
`find`	Search for files and directories by various criteria	`find . -type f -name "*.pdf"`
`grep`	Search inside files for lines matching a pattern (recursive)	`grep -R "TODO" .`

3.1.1 VIM

When you open a code file in the LINUX console you can access it through different editors. One that is commonly used and that we mostly worked in during the project is VIM. VIM is a very flexible code editor with alot of different commands to master. Below we will list the most common commands used during the project:

Table 3.2: Common Vim Commands

Command	Description	Example
`i`	Enter Insert mode before the cursor	Press `i`, then type text
`Esc`	Return to Normal mode	Press `Esc` to stop inserting
`:w`	Write (save) current file	`:w`
`:q`	Quit Vim (fails if unsaved changes)	`:q`
`:wq`	Write and quit	`:wq`
`u`	Undo last change	Press `u` in Normal mode
`Ctrl+r`	Redo (undo the undo)	Press `Ctrl+r`
`dd`	Delete (cut) entire current line	`dd`
`yy`	Yank (copy) entire current line	`yy`
`p`	Put (paste) after the cursor	`p`
`n`	Repeat last search forward	Press `n`

3.2 Fortran Code

Fortran (Formula Translation) is one of the oldest programming languages, designed in the 1950s for scientific and engineering solutions. The language best use case is to write formulas and numeric algorithms in a more natural way than assembly language, while still achieving efficient machine code execution. Early Fortran included a built-in complex number data type, which made it especially appealing for engineering and scientific applications requiring complex arithmetic [10].

Fortran has now been in continuous use for over half a century, and remains highly relevant in computationally intensive fields such as numerical weather prediction, climate modeling, finite element analysis, computational fluid dynamics, astrophysics, and computational chemistry [11]. The language's design, especially its handling of arrays and non-aliasing semantics, allows compilers to optimize numeric code aggressively. High-performance Fortran compilers can automatically vectorize loops and make use of advanced CPU instructions, often achieving performance that is hard to match in languages like C or C++ without careful manual tuning.

This is one reason Fortran is sometimes preferred for numerically intensive kernels.

Fortran has developed over the years with new features and upgrades to its syntax to better represent today's programming standards. We will talk about how Fortran code has evolved, comparing older Fortran styles to modern Fortran. The key differences are in the code structure, syntax, and capabilities between classic FORTRAN (e.g., 1970s FORTRAN 77 style) and modern Fortran (Fortran 90 and later). In our program we were sometimes required to update the outdated syntax in certain code parts with more modern syntax to help the code through the compiler.

Vector Addition in Fortran

One of the biggest productivity boosts in modern Fortran is the ability to operate on whole arrays without writing explicit loops. In FORTRAN 77, if you wanted to add two arrays element by element, you had to loop over the indices manually. In Fortran 90 and later, you can do it with one assignment statement using array expressions[12]

FORTRAN 77 style (explicit loop)

```
      REAL A(5), B(5), C(5)
      INTEGER I
      C(1) = 0.0
      B(1) = 1.0
      ! ... (assume A and B are initialized with some values)
          ...
      DO 100 I = 1, 5
         C(I) = A(I) + B(I)
 100  CONTINUE
      PRINT *, 'Result vector:', C
      END
```

Listing 3.1: FORTRAN 77 vector addition

In this old code, note the fixed columns and the loop construct. The loop is labeled with `100` and ended by a `CONTINUE` statement with the matching label. The addition is performed element by element inside the loop.

Modern Fortran style (array expression)

```
program vector_add
    implicit none
    integer, parameter :: N = 5
    real :: A(N), B(N), C(N)
    ! Initialize A and B (for example purposes)
    A = [1.0, 2.0, 3.0, 4.0, 5.0]
    B = 1.0         ! broadcast to all elements (B = [1,1,1,1,1])
    ! Element-wise addition of arrays:
```

```
    C = A + B
    print *, "Result vector:", C
end program vector_add
```

Listing 3.2: Modern Fortran vector addition

Here we see free-form source (no column restrictions) and lowercase code (Fortran is case-insensitive). The array addition `C = A + B` replaces the entire loop; the Fortran runtime/compiler handles the element-wise operation efficiently. Also, we used array constructors `([ ])` and constant broadcast (`B = 1.0` sets every element of B to 1.0) which are conveniences provided by Fortran 90 and later. This code is clearer and less error-prone, and a good compiler will optimize the array operation, potentially using vector instructions [12].

Sharing Data

In older Fortran, sharing variables or constants across multiple program units was often done with COMMON blocks. This approach, however, could lead to maintenance issues as programs grew. Modern Fortran encourages the use of modules to share data and constants in a controlled way (encapsulation and explicit interfaces). The following demonstrates how one might define and use a constant pi in FORTRAN 77 versus in modern Fortran.

FORTRAN 77 style (COMMON block)

```
      COMMON /ConstBlock/ PI
      DATA PI /3.14159/

      SUBROUTINE compute_area(radius, area)
          REAL radius, area
          COMMON /ConstBlock/ PI
          area = PI * radius * radius
          RETURN
      END
```

Listing 3.3: Fortran COMMON block

In this F77 code, a named common block `/ConstBlock/` is used to hold the constant `PI`. The `DATA` statement in the main program (or BLOCK DATA subprogram) initializes `PI` with the value 3.14159. Any subroutine that needs `PI` includes the same `COMMON /ConstBlock/ PI` line to access that variable. This mechanism works, but it relies on all pieces of code using the common block consistently. If, for example, one subroutine forgot to list `PI` in the common or used a different size for the common block, it could cause errors. It is also not obvious just from the subroutine code that `PI` is a constant, one has to know it comes from a common block.

Modern Fortran style (using a module)

```
module ConstModule
    implicit none
    real, parameter :: PI = 3.14159
end module ConstModule

program circle_area
    use ConstModule          ! import the constant PI
    implicit none
    real :: radius, area
    print *, "Enter a radius:"
    read *, radius
    area = PI * radius**2    ! use the shared constant PI
    print *, "Area of circle =", area
end program circle_area
```

Listing 3.4: Modern Fortran module

In the modern code, we define a module `ConstModule` that contains a named constant `PI`. The `real, parameter :: PI = 3.14159` establishes `PI` as a constant accessible to any program unit that does `use ConstModule`. The main program (or any subroutine) can then use that module to get `PI`. This method has several advantages: the value of `PI` is set in one place (inside the module) and cannot be accidentally changed at runtime (because it is a `parameter`). If multiple parts of the program use `PI`, they all refer to the same module definition, ensuring consistency. Moreover, if we decided to increase precision or change the value, we do it in the module and recompile, without hunting through code. Modules also allow specifying private vs public variables; in this simple case, `PI` is public by default so it is accessible to any user of the module. This approach is much cleaner than COMMON blocks. In fact, modules essentially replace COMMON blocks in modern Fortran for sharing data or global variables. The compiler can also check that the types and shapes of data in a module match when you use them, which prevents many errors that could occur with mismatched COMMON block declarations [12].

Control Flow and Syntax Improvements

A brief note on general syntax differences: modern Fortran also improved readability in control flow. For example, compare a conditional in an older style vs a modern style:

Fortran 66/77 style conditional

One might use an arithmetic IF or computed GOTO for multi-way branching, e.g. IF (X) 10, 20, 30 which branches to label 10, 20, or 30 depending on $X < 0$, $X = 0$, or $X > 0$ respectively [13]. This is cryptic by today's standards. By FORTRAN 77, one could at least do:

```
      IF (I .GT. 0) THEN
          CALL positive_branch()
      ELSE
          CALL nonpositive_branch()
      END IF
```

Listing 3.5: IF-ELSE statement in old Fortran

which is clearer. However, FORTRAN 77 still required labels for loops. A loop to iterate N times would be:

```
      INTEGER J
      J = 1
 200  IF (J .LE. N) THEN
          ... do something ...
          J = J + 1
          GOTO 200
      END IF
```

Listing 3.6: Loop iteration in old Fortran

Modern loop and conditional

In Fortran 90+, one writes:

```
      do j = 1, N
         ! do something
      end do
```

Listing 3.7: Modern end-do Fortran loop

with no labels required, and can exit loops via exit or skip to next iteration via cycle. Conditionals are written with matching END IF, and one can use SELECT CASE for multi-way branching on an integer or character value, instead of computing GOTO indices. These changes greatly reduce the need for labels and GO TOs, making code easier to follow and prove correct.

3.3 C/C++ Code

First, let us give a quick rundown of C coding. A C program is composed of functions where execution starts in main(). The codebase usually consists of variables declarations such as [14]:

- **char** stores single characters
- **int** stores integers (whole numbers)
- **float** stores fractional numbers, containing one or more decimals. Sufficient for storing 6-7 decimal digits

- **double** stores fractional numbers, containing one or more decimals. Sufficient for storing 15 decimal digits
- **pointers (type *)** a variable that stores the memory address of another variable as its value

But to utilize these and perform something with the code we have to use conditionals like if-else statements and loops [14]:

- **for** statement to specify a block of code to be executed if a condition is true
- **switch** statement that selects one of many code blocks to be executed
- **while** loops through a block of code as long as a specified condition is true
- **do-while** loop that will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.

But most of these things are very common in most general purpose programming languages today. What makes C coding unique is its memory management opportunities. This is also very important to understand with C coding other than pointers to have a smooth program that runs properly [14].

- **malloc()** function has one parameter, size, which specifies how much memory to allocate
- **calloc()** function has two parameters, amount - Specifies the amount of items to allocate, size - Specifies the size of each item measured in bytes
- **realloc()** reallocate memory to mae it larger if the amount of memory originally reserved is not enough
- **free()** to deallocate memory

For this project there was alot of old and deprecated C code that needed syntax updating for our modern compilers. Below you will see the most common syntax differences between old-style C and modern C.

Table 3.3: Syntax differences between old-style C and modern C

Area	**Old-Style (K&R / Pre-ANSI)**	**Modern C (C99+)**
Function Declarations	Omitted prototypes; definitions like `int foo(a, b) int a; char *b; { ... }`	Require prototypes; e.g. `int foo(int a, int b);` and `int foo(int a, char *b){ ... }`
Implicit `int` Return Type	Functions default to returning `int`.	Implicit `int` removed must declare return type explicitly.
Variable Declarations	All declarations at top of a block before any statements.	Can declare anywhere; mix declarations and code.
`for` Loop Declarations	Loop index declared beforehand: `int i; for (i = 0; i < n; i++) ...`	Loop-local declarations allowed: `for (int i = 0; i < n; i++) { ... }`
Comments	Only `/* ... */`, cannot nest.	Adds `// ...` single-line comments.
Compound Literals & Designators	No designated initializers.	Supports both: `struct Point p = {.y=2, .x=1};` and compound literals `(int[]){1,2,3}`
`const` & `volatile`	Rare or unsupported.	Widely used for safer APIs and optimization hints.
`inline` Functions	Not part of K&R.	Introduced in C99 to suggest inlining.
Fixed-Width Integers	Only `short`, `int`, `long`.	`<stdint.h>` provides `int8_t`, `uint32_t`, etc.
Variable-Length Arrays	Absent you would `malloc`.	C99 VLA: `int arr[n];` with runtime size.
`restrict` Keyword	Not available.	C99-only hint for pointer aliasing.

3.4 Yacc Code

Yacc (Yet Another Compiler-Compiler) is not a general purpose language like C, Python or Java, it acts like a parser generator. You write a declarative description of your grammar (the syntax of the language you want to parse) plus snippets of C code for the actions to take when rules match; Yacc then spits out a C program that implements an LALR(1) parser for you[15].

A typical Yacc code is divided into three sections separated by %% lines.

- **Definition Section:** Defines macros and imports header files written in C. It is also possible to write any C code here, which will be copied verbatim into the generated source file[15].
- **Rules Section:** Associates regular expression patterns with C statements. When the lexer sees text in the input matching a given pattern, it will execute the associated C code[15].
- **Subroutines and C Code Section:** Contains C statements and functions that are copied verbatim to the generated source file. These statements presumably contain code called by the rules in the rules section[15].

```
1  /* 1. Definitions Section */
2  %{
3    /* C code to copy verbatim into the generated .c */
4    #include <stdio.h>
5  %}
6  %token NUMBER PLUS MINUS
7
8  /* Operator precedence/associativity */
9  %left PLUS MINUS
10
11 %%
12
13 /* 2. Grammar Rules Section */
14 expr:
15       expr PLUS expr    { $$ = $1 + $3; }
16     | expr MINUS expr   { $$ = $1 - $3; }
17     | NUMBER            { $$ = $1; }
18     ;
19
20 %%
21
22 /* 3. User Subroutines Section/C Code */
23 int yylex(void) { /* your lexer (often from Lex) */ }
24 void yyerror(const char *s) { fprintf(stderr, "Error: %s\n", s);
       }
25 int main(void) { return yyparse(); }
```

Listing 3.8: Typical Yacc code structure

3.5 Lex Code

Lex (lexical analyzer generator) is a tool that produces lexical analyzers. It is typically paired with the Yacc parser generator and is the default lexer generator on many Unix and Unix-like systems. Given a description of token patterns, Lex reads that specification from an input stream and outputs C source code implementing the scanner[15].

Just as Yacc, Lex code is divided intro the same three sections separated by %% lines.

```
/* 1. Definitions Section */
%{
  /* C code copied verbatim: headers, globals */
  #include "y.tab.h"        /* token definitions from Yacc */
  extern int yylval;        /* semantic value for tokens */
%}

/* Definitions and options */
DIGIT    [0-9]
ID       [A-Za-z_][A-Za-z0-9_]*

%%

/* 2. Rules Section: pattern   action */
{DIGIT}+            { yylval = atoi(yytext); return NUMBER; }
"+"                 { return PLUS; }
"-"                 { return MINUS; }
[ \t\n]+            { /* skip whitespace */ }
{ID}                { yylval = strdup(yytext); return IDENT; }
.                   { /* catch-all: ignore unknown */ }

%%

/* 3. User Subroutines Section */
int yywrap(void) { return 1; }
```

Listing 3.9: Typical Lex code structure

3.6 Compilers

A compiler transforms the clear, human-readable instructions you write into the precise machine code your computer's processor understands. By parsing your high-level source code, checking for errors, and applying optimizations, the compiler not only ensures your program is correct and safe but also tunes it for peak performance. Compilers let you focus on coding while they handle the intricate details of translation, portability, and efficiency behind the scenes.

We used four different compilers during the project, gcc, gfortran, flex and bison:

The GNU Compiler Collection (**GCC**) is a versatile, open-source compiler originally designed for C and C++, but now supporting multiple other languages. **GCC** takes source code through multiple stages like preprocessing, parsing, optimization, assembly, and linking to produce highly efficient machine code. Its strong optimization passes and active development have made it a standard compiler for Unix-like systems[16].

GFortran is the Fortran front-end of the GCC project, offering full compliance with modern Fortran standards and access to GCC's optimization infrastructure. Like GCC, it leverages the same back-end code generators and optimization passes, but it is tailored to Fortran's syntax and its emphasis on numerical computations and array operations. **GFortran** thus brings Fortran developers the same performance tuning that C and C++ users enjoy[16].

Flex (Fast Lexical Analyzer) is a tool for generating C code that performs lexical analysis, breaking an input stream of characters into meaningful tokens such as keywords, identifiers, numbers, and operators. Users write a simple specification of regular-expression patterns and associated actions; **Flex** then converts these into a state machine in C, like lex.yy.c that scans text efficiently. **Flex** focuses exclusively on tokenization and leaves grammar and structure handling to a parser generator[15].

Bison is a parser generator that constructs a bottom-up (LALR(1)) parser in C or C++ from a user defined grammar. Given a set of production rules that describe how tokens form expressions and statements, **Bison** produces code that builds parse trees or invokes user actions as it recognizes syntactic patterns. In typical projects, **Bison** works hand in hand with Flex: Flex tokenizes the input, and **Bison** assembles those tokens into a higher-level syntactic structure[15].

While GCC and GFortran are full compilers that translate high-level source directly into machine executables, Flex and Bison are compiler-builder tools: Flex handles the front-end task of tokenizing raw text, and Bison manages the syntactic parsing stage. In a custom language implementation, one often uses Flex to produce a stream of tokens, Bison to recognize grammatical structure, and finally GCC or GFortran to compile the resulting intermediate C code (or link against runtime libraries) into optimized binaries.

Chapter 4

Method

4.1 Project Efforts

4.1.1 Initial Program Load (IPL)

I methodically worked through the IPL codebase folder by folder, calling the make_dir build script in each to pinpoint compilation failures. Early on, many error messages ranging from obsolete K&R-style C declarations to legacy Fortran constructs flooded the output. I cleaned these up first, replacing deprecated syntax (e.g., old-style function prototypes, implicit int) with ANSI-compliant declarations, then re-ran the build and filtered the remaining diagnostics with grep to focus on the last blockers.

Here is one example of a syntax update for a C file. The left part is the updated code and the right part is the old C-style:

Figure 4.1: Comparison between old C-code to modern C rotated 90 degrees

Figure 4.2: Comparison between old C-code to modern C rotated 90 degrees

Once the basic syntax was corrected, the compiler began reporting "multiple definition" errors, each prefaced by "first defined here." These occur when the same global symbol (variable or function) is defined in more than one place, causing the linker to complain that it does not know which copy to use. To resolve this, each conflicting symbols were traced back to its header-file declaration and then converted it's redundant definitions in the .c/.f files into extern declarations. Initially, only simple globals were applied as extern (e.g., int error_flag; → extern int error_flag;), but the linker still found duplicates for pointers and struct instances. Extending the extern treatment to those types cleared most conflicts but not all.

At this stage, "undefined reference" errors persisted for certain symbols, indicating that although duplicate definitions had been removed, the linker now could not find any definition at all. Each case was resolved by ensuring that exactly one source file provided the actual (non-extern) definition. Finally, to accommodate some legacy object files that default to the older "common" linkage model (where uninitialized globals are implicitly merged), the -fcommon flag was added to the compiler options. This flag tells GCC to revert to the pre-C99 behavior of allowing tentative definitions in multiple files to be merged at link time. With extern declarations in certain headers and -fcommon enabled, the entire IPL framework now links cleanly and runs without error.

4.1.2 Analysis

Once the IPL was operational, we turned our attention to the analysis module and its associated issues. We experimented with various compiler flag configurations (e.g, -std=legacy and -ansi) to force older code to compile on current compilers. We even applied the same flags to both IPL and the analysis code since they originally used different settings but this yielded no improvement.

Next, to address the undefined reference errors in the analysis build when compiling, we duplicated IPL source files that triggered these errors. These errors stemmed from inconsistent Fortran–C interfaces, some calls and variable names required an underscore suffix, while others did not. On further inspection, we discovered that the IPL libraries (dblib and Acelib), which are linked into IPL, had not been included in the analysis linking stage. Once we added those libraries, all undefined reference errors disappeared; we then manually removed a few remaining underscore mismatches in the code.

With those fixes in place, both IPL and the analysis module compiled and executed successfully, producing the expected output files, AceMain for IPL and Acek01 for analysis confirming that the framework was functioning correctly.

4.2 Error Handling

Once the compiler has done its work of preprocessing, parsing, optimizing, and generating object files, it hands you back a report of any problems it encountered. Errors like syntax mistakes, type mismatches, or missing symbols must be addressed before your code will link or run correctly. Warnings, on the other hand, flag questionable constructs (unused variables, implicit conversions, deprecated functions) that do not stop the build but can hide subtle bugs or reduce performance. In our workflow, we review and correct all errors immediately to restore a clean compile; warnings are inspected but can often be deferred or even suppressed with certain flags. By distinguishing critical errors from lower-priority warnings we keep the build output focused on the issues that truly matter, speeding both development of the framework and also the extensive code review required.

4.3 Compiler Flags

After the code has been written by the user, produced to readable format through the compiler to the processor, and adjusted for errors. If the code or framework is still giving unreasonable runtime implications the user can sort to using compiler flags to fix issues instead.

During the project we got a working solution using -fcommon, -DYY_ALWAYS_INTERACTIVE, -Wno-implicit-function-declaration -no-pie. Down below we will provide a detailed description of their necessary usage in the project:

By default in older GCC versions, uninitialized global variables with the same name across different translation units were merged into a single "common" symbol at link time. The **-fcommon** flag explicitly tells GCC to keep that behavior: any tentative definitions are placed in a common block, allowing multiple such definitions to resolve to one. In contrast, -fno-common (the newer default) places each tentative definition in its own section, and will produce a linker error if more than one definition exists. That is why we opted to use **-fcommon** during the project to avoid this linker error with multiple definitions.

This flag defines the preprocessor macro **YY_ALWAYS_INTERACTIVE** as if you had written #define **YY_ALWAYS_INTERACTIVE** 1 at the top of every source file. It is commonly used with tools like Flex: when **YY_ALWAYS_INTERACTIVE** is set, the generated scanner forces interactive (character-by-character) input rather than buffered, block-mode input. You would use it if you need your lexer to respond to each character as it arrives. This flag was efficient during the project so our lex code worked as intended.

GCC's -Wimplicit-function-declaration warning notifies you when you

call a function without first declaring it. In modern C, that is an error; in C89 it was allowed. By passing **-Wno-implicit-function-declaration**, you silence that warning, telling GCC not to warn (or error) if you call undeclared functions. This was useful when compiling old C code during the project that lacked proper modern prototype, but it risked hidden bugs if a function name was declared not properly or a call to it was with the wrong signature.

Position-Independent Executables (PIEs) are binaries whose code can be loaded at any address in memory, useful for security but sometimes unwanted (for performance or compatibility). The **-no-pie** flag tells GCC to produce a normal, fixed address executable rather than a PIE. That means the final binary will not include the extra relocation machinery needed for runtime address rebasing, and it will load at its default link time address.

4.4 Automated Build Scripts

In large software projects where hundreds or even thousands of source files, libraries, and resources must come together, manual compilation and linking quickly becomes error-prone, tedious, and virtually impossible to keep consistent across the project.

In our project, we begin by loading all necessary compiler flags from a centralized script in home/ace/IPL/bin. Once on that path we write . profile to load the compiler flags which helps in using the exact same optimization settings, warning options and include paths for all the files in the project. Once those flags are in place, we invoke our build script, which traverses the source tree and compiles each module in turn, recompiling all the neccessary files. This automated compilation phase guarantees consistency and speed, sparing us from manually typing long gcc or gfortran commands for every file. Finally, the build script hands off the resulting object files to the linker, which resolves symbols across libraries, embeds resources, and produces the final executable. By structuring the workflow this way, flags first, then compilation, then linking, we achieve a reproducible, efficient build process that scales effortlessly as the codebase grows.

Chapter 5

Results and Discussion

5.1 Ace Framework

Figure 5.1 illustrates the framework front page generated by the IPL. If the IPL fails to compile, the program will have no interface. A successful build produces AceMain, which provides the graphical user interface. Although the program can still run in batch-mode without a functional IPL, batch-mode likewise depends on AceMain to manage geometry, mesh, material properties, boundary conditions, and sources for the FEM simulation. Therefore, any incomplete IPL build could cause certain elements of AceMain to fail, and with it, even batch-mode execution.

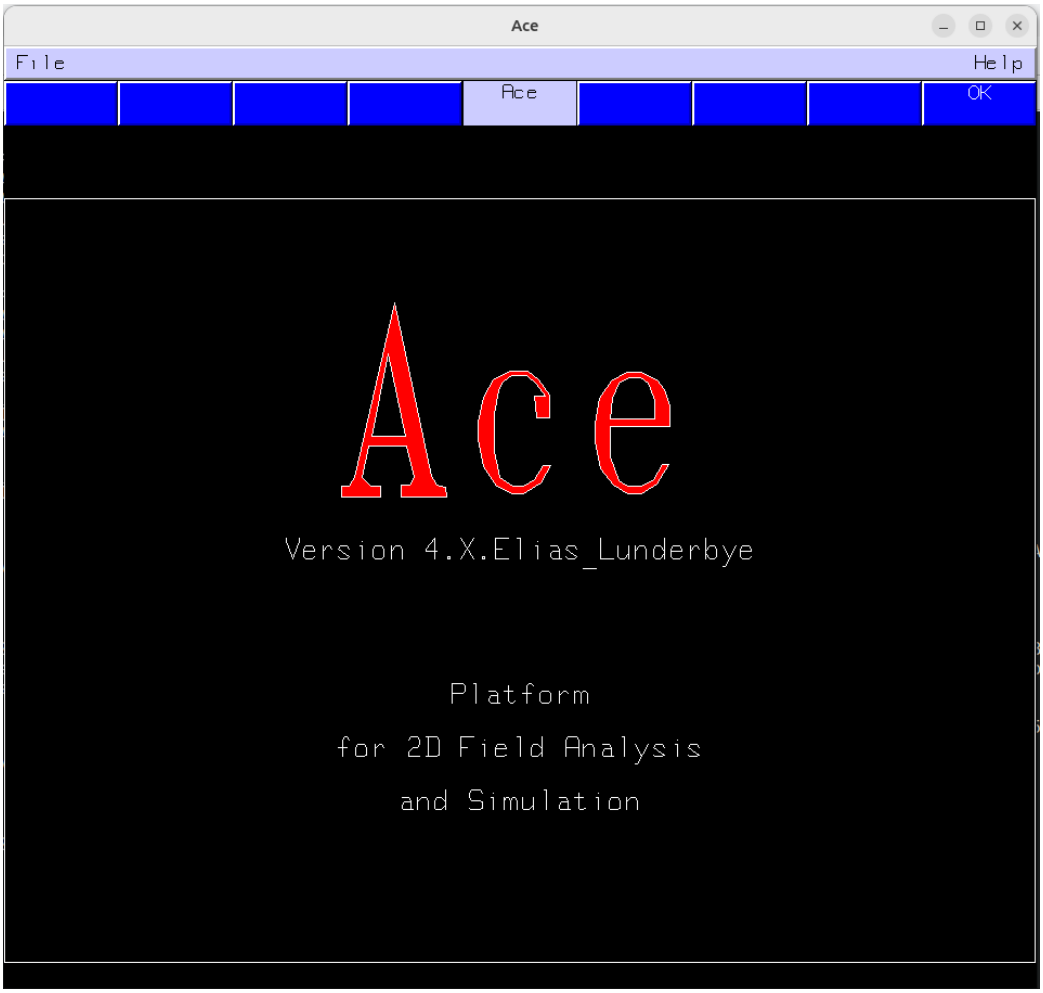

Figure 5.1: First page of Ace Framework

Clicking “OK” in the top-right corner of the front page opens the electrical-machine configuration screen (see Figure 5.2), where the Analysis component takes over. While some functions may run without a fully operational analysis module, critical calculations and visualizations require it to be complete. Once the Analysis module is fully compiled, it generates Acek01, the analysis counterpart to AceMain, which ensures that our simulation results are displayed correctly.

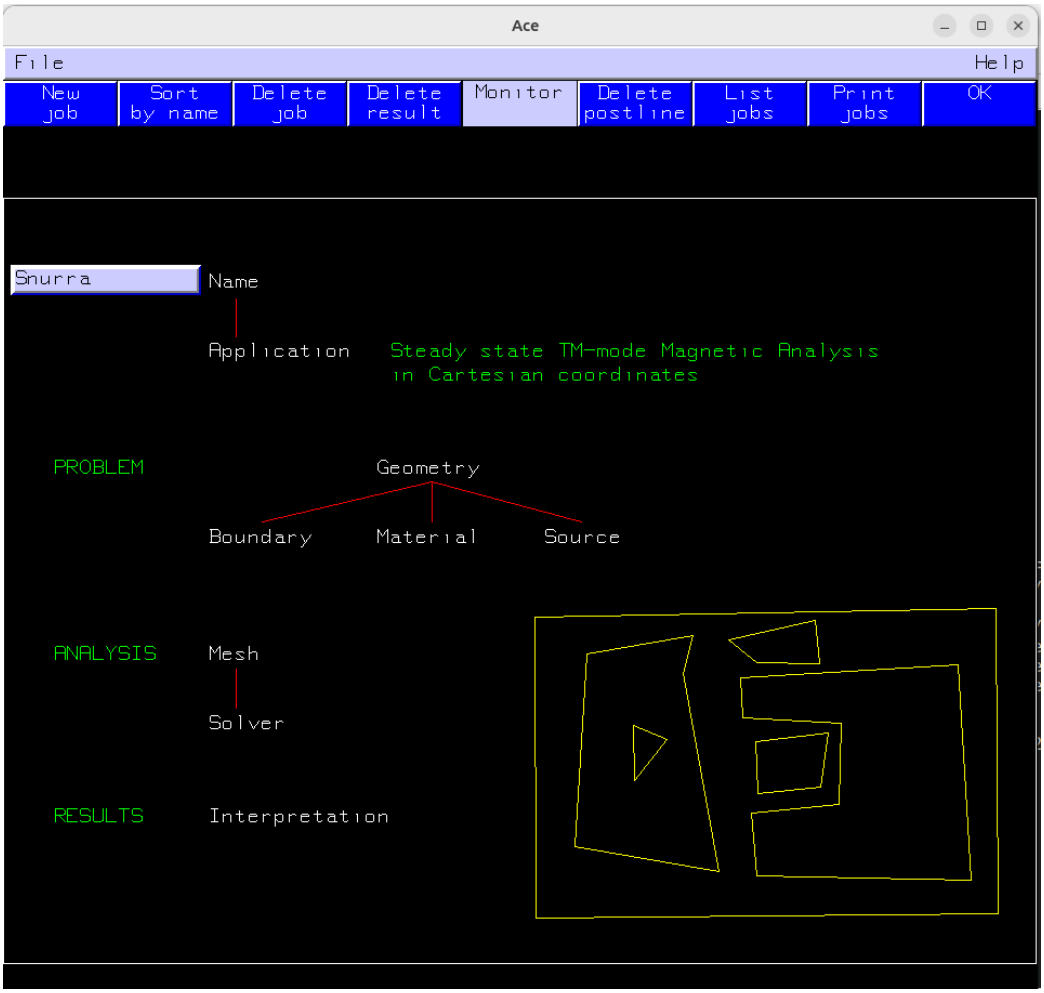

Figure 5.2: Ace Framework configurations page

The final results of a test simulation will be presented below with a direct comparison of Ubuntu 16 and Ubuntu 24 versions for validation.

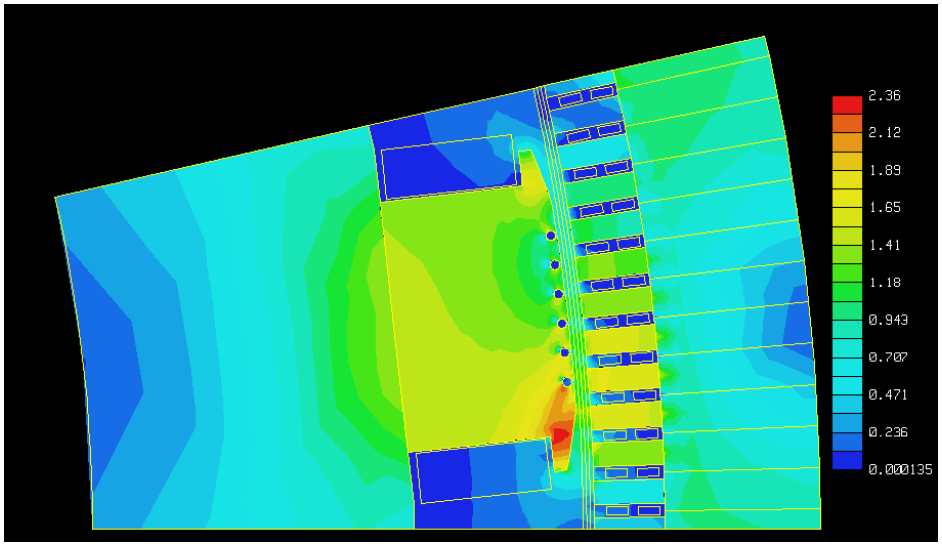

Figure 5.3: Ace Framework magnetic flux density for simulation test case Ubuntu 16

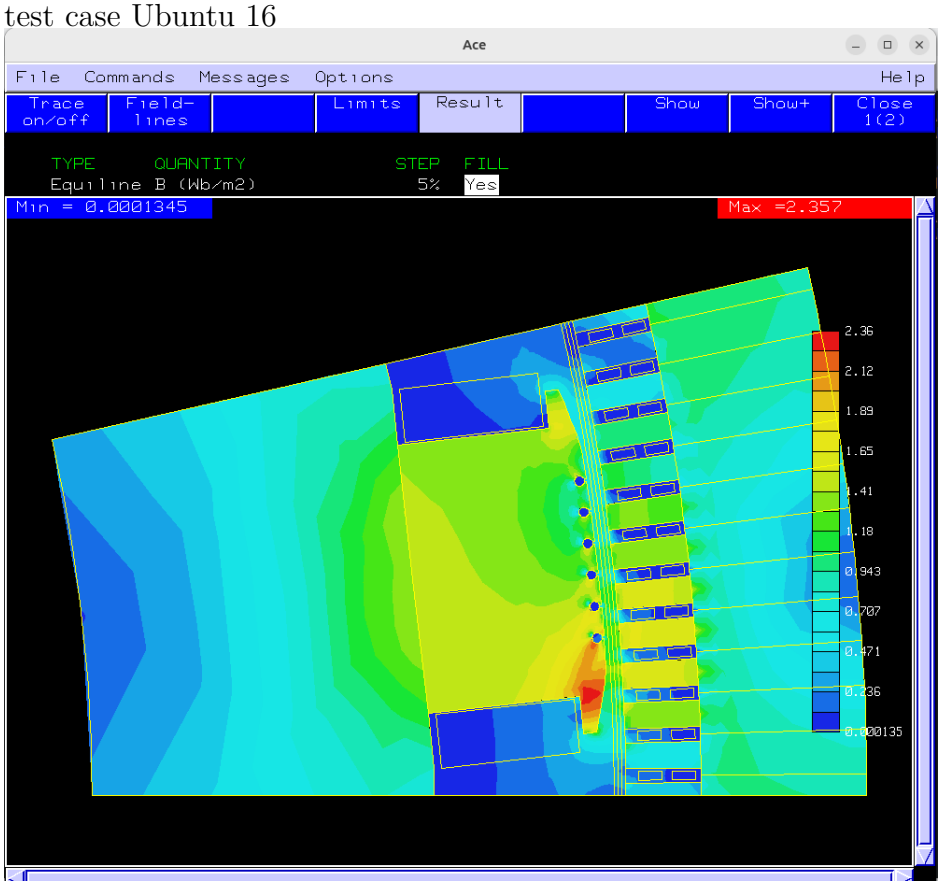

Figure 5.4: Ace Framework magnetic flux density for simulation test case Ubuntu 24

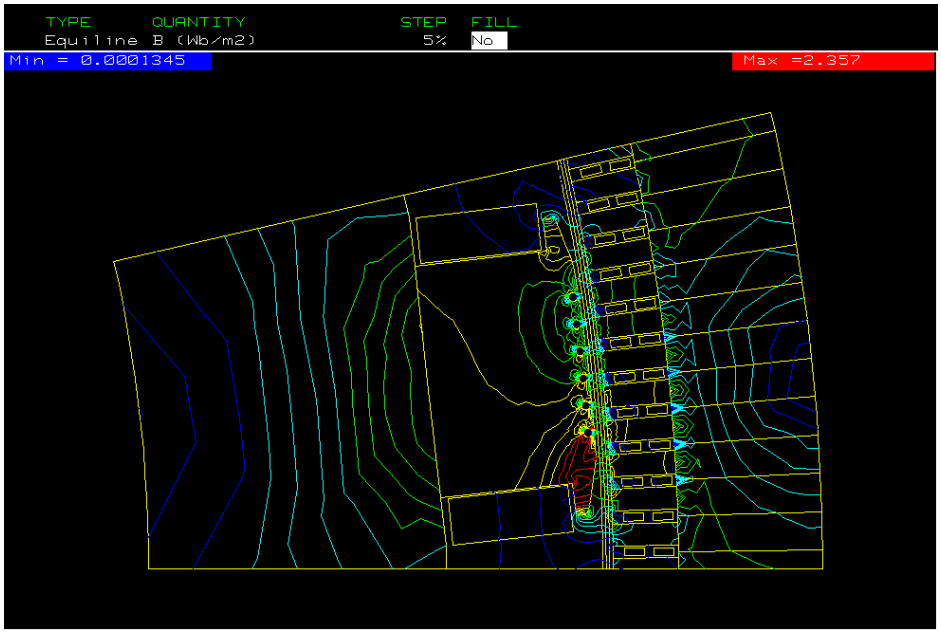

Figure 5.5: Ace Framework field lines for simulation test case Ubuntu 16

Figure 5.6: Ace Framework field lines for simulation test case Ubuntu 24

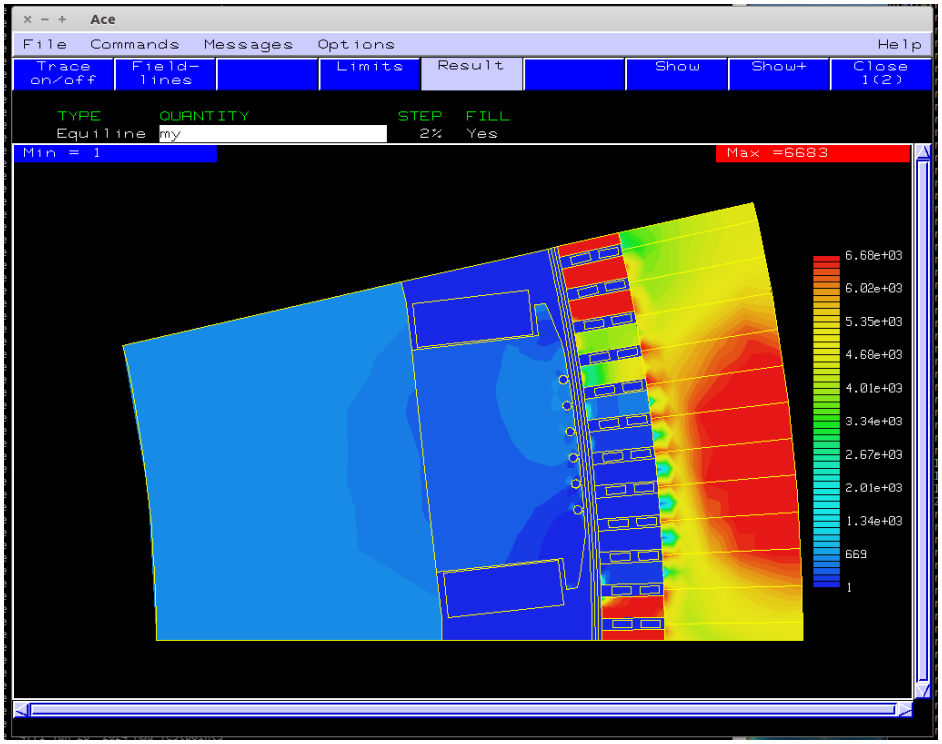

Figure 5.7: Ace Framework magnetic permeability for simulation test case Ubuntu 16

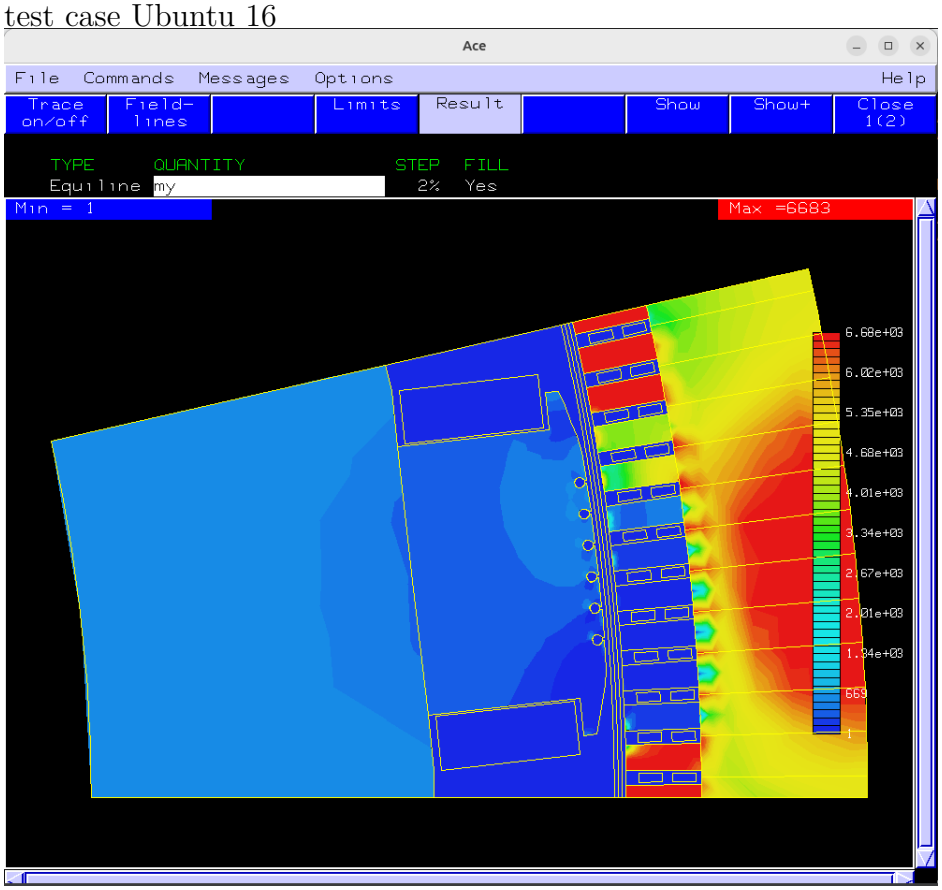

Figure 5.8: Ace Framework magnetic permeability for simulation test case Ubuntu 24

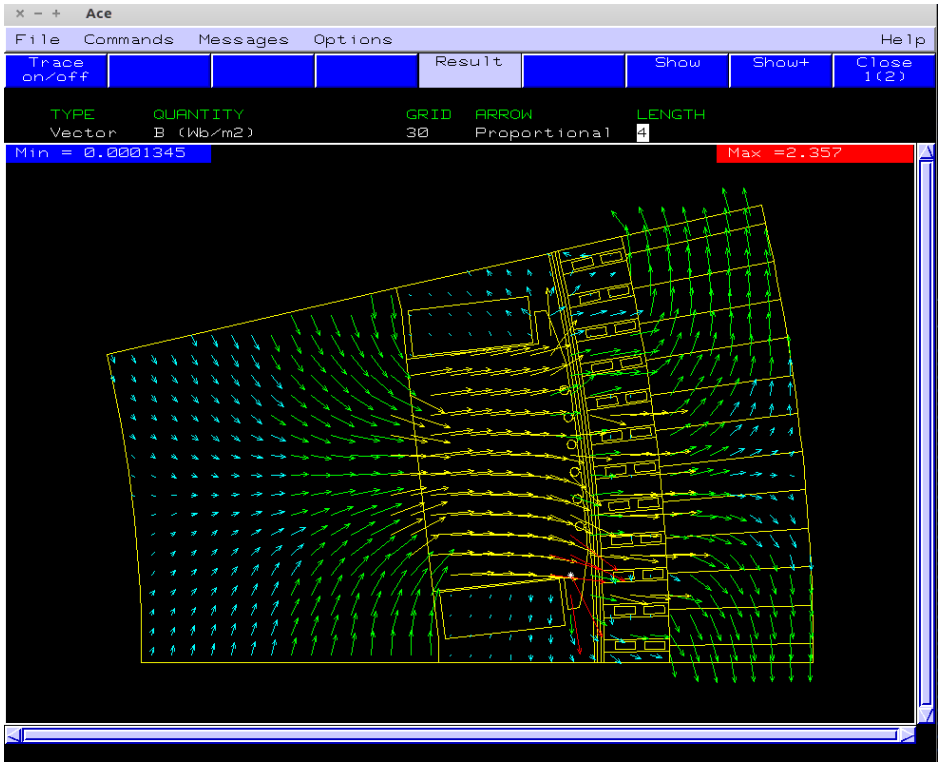

Figure 5.9: Ace Framework vector representation of magnetic flux for simulation test case Ubuntu 16

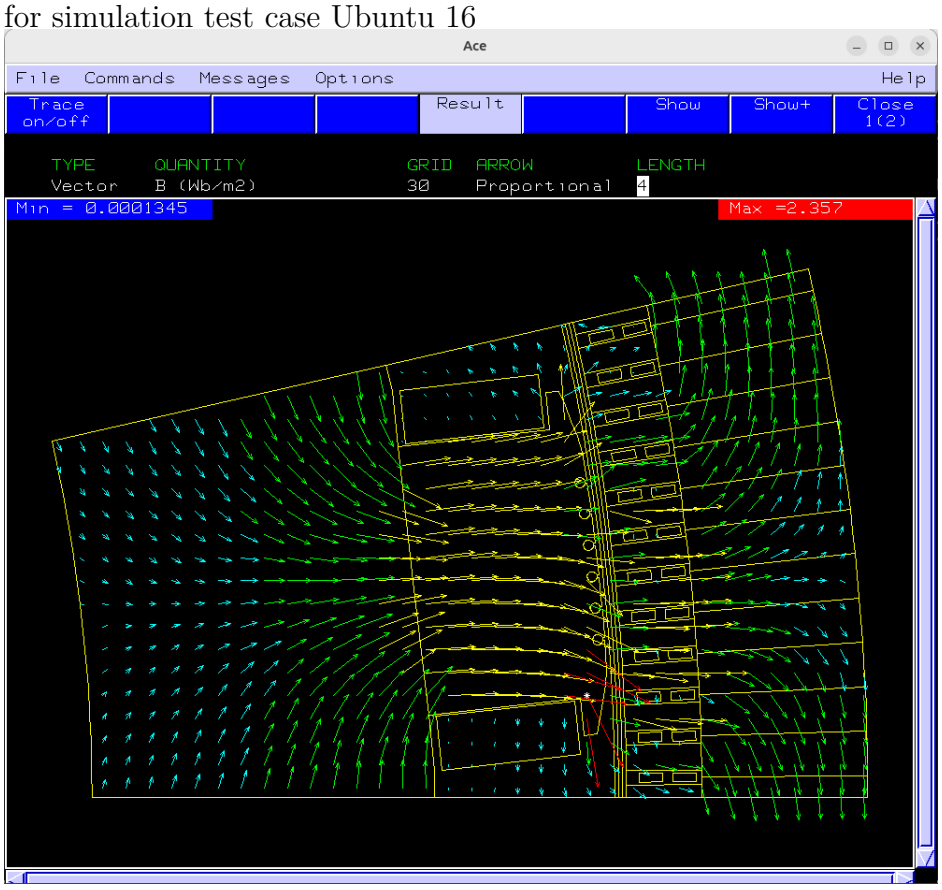

Figure 5.10: Ace Framework vector representation of magnetic flux for simulation test case Ubuntu 24

5.2 Challenges in Legacy Code Porting

Porting our compiler framework from Ubuntu 16 to Ubuntu 24 revealed a host of challenges inherent to a software that was first created in the 1970s. First and foremost, the original codebase relies on older syntax and conventions, K&R-style function definitions, implicit int declarations, and non-ANSI-standard headers that modern GCC and GFortran versions no longer accept by default. To bridge this gap, we had to introduce compatibility fixes and invoke legacy-mode compiler flags (e.g. -fcommon and -Wno-implicit-function-declaration), yet even with these measures, dozens of subtle parsing errors and warnings surfaced during our initial compilation pass.

Beyond language syntax, the framework's build infrastructure assumed obsolete toolchains and directory layouts. Our automated script responsible for loading flags, compiling modules, and linking, encountered missing utilities, renamed libraries, and relocated system headers. We addressed these by installing backward-compatible packages, crafting wrapper scripts to emulate legacy tool behavior, and updating include paths in our build definitions. Even then, differences in default linker behavior, Ubuntu 24's insistence on PIE binaries versus Ubuntu 16's fixed-address executables required the addition of -no-pie to our linking stage to maintain the runtime semantics the original code expected.

Finally, compiler warnings demanded disciplined error-handling policies. While errors halted our automated build until corrected, warnings, often dozens per file, threatened to drown genuine faults. Therefore, we had to compile specific files or directories of files that we knew were troublesome and compile those separately from the rest to retrieve the important errors. Legacy code porting is not merely a matter of "make it compile again," but rather an exercise in understanding decades-old design decisions, reconciling them with modern toolchains, and systematically hardening the software for future maintenance on Ubuntu 24 and beyond.

5.3 Impact of Modern Compiler Features

The leap from old toolchains to the latest GCC and GFortran on Ubuntu 24 brought a wave of powerful compiler features that significantly improved the performance of our framework. Advanced optimization passes such as link-time optimization and improved auto-vectorization allowed us to squeeze much better runtime speed out of the same algorithms, without touching a single line of legacy code.

When bringing a codebase written in the 1970s up to Ubuntu 24 with today's GCC and GFortran, the usage of compatibility flags can mean the difference between "it compiles" and "it actually works." Flags such as -fcommon restore the old behavior of merging tentative definitions for

globals, preventing linker errors on variables declared in multiple modules, while -std=gnu89 or -std=legacy can relax strict ANSI rules so K&R-style function definitions and implicit int declarations still parse correctly. Similarly, adding -no-pie counteracts Ubuntu 24's default of building position-independent executables, preserving assumptions in the original runtime loader. Without these targeted switches, the compiler simply refuses to accept outdated syntax or defaults to newer binary formats that break legacy expectations.

However, it is crucial to recognize that many flags merely suppress errors and warnings rather than resolve root causes. For example, -Wno-implicit-function-declaration silences warnings about undeclared functions yet the underlying issue of missing prototypes remains, leaving room for mismatched signatures and hard-to-trace bugs. Likewise, -Wno-unused-variable hides unused-variable notices, but the dead code still inflates the binary and clutters future maintenance. Over reliance on blanket suppression could lure you into a false sense of security.

By striking this balance, leveraging flags to bridge genuine incompatibilities while refusing to treat warnings as mere nuisances, we ensure that our ported framework does more than just run again: it becomes more maintainable, and better aligned with the guarantees of modern compilers.

Chapter 6

Conclusion

In this project, we successfully modernized and ported a FEM framework for simulating rotating electrical machines from Ubuntu 16 to Ubuntu 24. To achieve this, we delved deeply into the existing Fortran, C, C++, Yacc and Lex codebase, identifying and updating numerous deprecated constructs—old-style function prototypes, implicit typing, legacy Fortran features and more to ANSI-compliant, maintainable syntax.

Throughout the migration we also navigated Linux tooling shell commands, ‘make_dir‘ build scripts and VIM to streamline our workflow. We systematically addressed compilation errors by filtering and analyzing diagnostics with ‘grep‘, refactored header (‘.h‘) files to replace multiple definitions with proper ‘extern‘ declarations, and experimented with compiler flags (notably ‘-fcommon‘) to accommodate legacy object files. To accelerate iterative builds across the many source directories, we developed automated build scripts that accelerated compilation.

The code now compiles and links on Ubuntu 24. Moreover, with the Analysis module fully integrated, the framework not only produces AceMain for the IPL interface but also generates the Acek01 file, ensuring that simulation results are correctly post-processed and displayed. Looking ahead, this modernization lays the groundwork for further enhancements such as performance optimizations through parallel FEM calculations, and with modern GPUs, accelerate this process even more. Ensuring the framework remains a robust and efficient design tool for motors and generators in the decades to come.

Chapter 7

Sammanfattning

I det här projektet moderniserade och överförde vi ett FEM-program för simulering av roterande elektriska maskiner från Ubuntu 16 till Ubuntu 24. För att åstadkomma detta undersökte vi den befintliga koden som var skriven i Fortran, C, C++, Yacc och Lex, där vi identifierade och uppdaterade en mängd gammal kod som inkluderade gammaldags funktionsprototyper, implicita deklareringar, äldre Fortran funktioner med mera till ANSI-kompatibel och äldre syntax stiler.

Under överföringen hanterade vi även olika Linux-verktyg, shellkommandon, make_dir-byggskript och VIM för att effektivisera arbetsflödet. Vi åtgärdade systematiskt kompileringsfel genom att filtrera och analysera med hjälp av grep, omgjorda headerfiler (.h) för att ersätta dubbla definitioner med korrekta extern deklarationer och experimenterade med kompilatorflaggor (särskilt -fcommon) för att hantera äldre objektfiler. För att effektivisera kompileringen över alla katalogerna med filer skapades ett automatiserat byggskript som effektiviserade kompileringsprocessen.

Koden kompilerar och länkar nu på Ubuntu 24. Dessutom, med Analysismodulen fullt integrerad, producerar programmet inte bara AceMain för IPL-gränssnittet utan genererar även Acek01-filen, vilket säkerställer att simuleringsresultaten bearbetas och visualiseras korrekt. Denna modernisering lägger grunden för framtida ytterligare förbättringar såsom prestandaoptimeringar via parallella FEM-beräkningar och med moderna GPU:er kan processen accelereras ytterligare. Detta säkerställer att programmet förblir ett robust och effektivt verktyg för motorer och generatorer framöver.

Bibliography

[1] Huynh, C.(2021, April 26). Permanent magnet vs. induction motors. *Turbo Machinery Magazine.* https://www.turbomachinerymag.com/view/permanent-magnet-vs-induction-motors

[2] Klempner, G & Kerszenbaum, I.(2004).Operation and Maintenance of Large Turbo-Generators.John Wiley & Sons https://books.google.se/books?id=DPdkLp_crLAC&dq=stator&pg=PA34&redir_esc=y#v=onepage&q=stator&f=false

[3] Wikipedia contributors. (2025, May 17). Electric motor. *Wikipedia*, https://en.wikipedia.org/wiki/Electric_motor

[4] Scarpino, M.Motors for Makers: A Guide to Steppers, Servos, and Other Electrical Machines. Que Publishing https://books.google.se/books?id=FpEFCwAAQBAJ&q=%22electric+motor%22+armature+lorentz+field+winding&redir_esc=y#v=onepage&q&f=false

[5] Brooke Stauffer, H.(2005).NFPA's Pocket Dictionary of Electrical Terms.National Fire Protection Association https://books.google.se/books?id=9sW_knz5P0wC&pg=PA36&redir_esc=y#v=onepage&q&f=false

[6] Umans, D, S.(2014).Electric Mahinery.McGraw-Hill https://mrce.in/ebooks/Electric%20Machinery%20Fitzgerald%20&%20Kingsley%E2%80%99s%207th%20Ed.pdf

[7] Griffiths, J, D & College, R.(1999).Introduction to Electrodynamics.Prentice Hall https://www.hlevkin.com/hlevkin/90MathPhysBioBooks/Physics/Physics/Electrodynamics/David%20J.%20Griffiths%20-%20Introduction%20to%20Electrodynamics-Prentice%20Hall%20(1999).pdf

[8] O.C. Zienkiewicz, R.L. Taylor & J.Z. Zhu.(2005).The Finite Element Method: Its Basis and Fundamentals.Butterworth-Heinemann

[9] Ranlöf, M.(2011).Electromagnetic Analysis of Hydroelectric Generators.Uppsala University

[10] Kupferschmid, M.(2009).Classical FORTRAN, 2nd ed.CRC Press

[11] GeeksforGeeks. (2022, February 25). Difference between COBOL and FORTRAN. *GeeksforGeeks*, https://www.geeksforgeeks.org/difference-between-cobol-and-fortran/.

[12] Layton, J. (n.d.). Modern Fortran: Fortran 90 *ADMIN Magazine*, https://www.admin-magazine.com/HPC/Articles/Modern-Fortran-Part-1#:~:text=match%20at%20L284%20old%20common,blocks%20feature%20of%20Fortran%2077.

[13] Modernizing Old Fortran in Fortran Wiki. (n.d.). *Fortran Wiki*, https://fortranwiki.org/fortran/show/Modernizing+Old+Fortran#:~:text=Arithmetic%20IF.

[14] C Tutorial. *W3Schools*, https://www.w3schools.com/c/index.php.

[15] Levine, R, J, Mason, T & Brown, D.(1992).lex & yacc.O'REILLY

[16] Wikipedia contributors. (2025, May 26). Compiler. *Wikipedia*, https://en.wikipedia.org/wiki/Compiler.