Luca Candussi, BSc

Integrating BEM and FEM Software Frameworks with a Focus on Acoustics

AUDIO ENGINEERING PROJECT

for the master's degree programme
Electrical Engineering and Audio Engineering
(Elektrotechnik-Toningenieur)

submitted to

Graz University of Technology
Faculty of Electrical and Information Engineering

supervisor and assessor

Ass.-Prof. Dipl.-Ing. Dr.techn. Stefan Schoder

co-supervisors

Dipl.-Ing. Florian Kraxberger, BSc

Institute of Fundamentals and Theory in Electrical Engineering (IGTE)
Head: Univ.-Prof. Dipl.-Ing. Dr.techn. Manfred Kaltenbacher
Inffeldgasse 18/I, 8010 Graz, Austria

Graz, April 2024

Abstract

The Finite Element Method (FEM) is a common tool for simulating physical problems and is utilized to numerically solve acoustic problems too. A problem is solved by discretizing the geometry and solving the governing partial differential equation (PDE) on the discretized domain. Similar to the FEM, the Boundary Element Method (BEM) is based on solving PDEs with the difference that the BEM further formulates it into integral equations. For this project, the external BEM software *NiHu* will be integrated into the simulation workflow of the already existing FEM environment *openCFS*. This workflow integration includes the possibility of installing *NiHu* within *openCFS* as well as the corresponding testcase. Upon completion of this project it will be possible to numerically solve an PDE problem provided as testcase by utilizing the boundary element method (BEM) in the former FEM only framework.

Kurzfassung

Die Finite-Elemente-Methode (FEM) ist ein gängiges Werkzeug zur Simulation physikalischer Probleme und wird auch zur numerischen Lösung von akustischen Problemen verwendet. Ein Problem wird gelöst, indem die Geometrie diskretisiert und die zugehörige partielle Differentialgleichung (PDE) im diskretisierten Bereich gelöst wird. Ähnlich wie die FEM basiert die Randelementmethode (BEM) auf der Lösung von PDEs, mit dem Unterschied, dass die BEM diese in Integralgleichungen formuliert. Für dieses Projekt wird die externe BEM-Software *NiHu* in den Workflow der bereits bestehenden FEM-Umgebung *openCFS* integriert. Diese Workflow-Integration macht es möglich, *NiHu* als externe Erweiterung von *openCFS* zu installieren. Nach Abschluss dieses Projekts wird es möglich sein, einen bereitgestellten Testcase mittels BEM in *openCFS* numerisch zu lösen.

Contents

1 Introduction

Introducing a new feature *NiHu* [5] to another software program *openCFS* [6] at first does not seam to be that much of a task. In this very case the new functionality can be seen as something like a black box which is built within the installation of the main program *openCFS*. Therefore *openCFS*' build and installation routine has to be edited in a way such that *NiHu* can be utilized as additional feature in *openCFS* properly. On the other hand there is no need - no chance - to conflict with *NiHu*'s code or structure itself. Peter Fiala and Péter Rucz, the founders of *NiHu* explain it in great detail on the *NiHu* website [5] which is where most of the information about how to build *NiHu* is found. How *NiHu* works and the maths behind the solver can be looked up in their paper on that topic [7].
The description of two methods for numerical solution, namely the *finite element method* (Sec. 2.1) and the *boundary element method* (Sec. **??** and their comparison are essential components of the software. This is because they represent the core innovation and significant transformation in the software and its structure, including the incorporation and development process. Both *openCFS* and *NiHu* are mostly developed with a *C++* (Sec. 3.1)back end and their build processes are undertaken by *CMake* (Sec. 3.2). David Cole [1] describes a straight forward way of adding an external *CMake* project to another. The theoretical background of both numerical solvers as summed up in chapter 2 is based on the lecture notes to the class *Computational Acoustics* [8] at IGTE and Manfred Kaltenbacher's *Computational Acoustics* [4].

1.1 Current state of openCFS

openCFS (Coupled Field Simulation) is an open source finite element-based multi-physics modelling and simulation tool used in scientific research and industrial applications. The modelling strategy focuses on physical fields and their respective couplings. `opencfs.org` [6] It is utilized for research in various fields including aero-acoustics and electromagnetics, e.g., at the *Institute of Fundamentals and Theory in Electrical Engineering* (IGTE).

Essentially *openCFS* operates based on an `ProjectName.xml` file containing information for the entire solution of a problem. Therefore it tells *openCFS* file names and where to find those additional files necessary for calculation e.g., the discretized geometry. Therefore to properly execute a simulation *openCFS*, further the "main"`.xml` file is in need of the following files.

- material file
- mesh file

The material file is an .xml file as well which contains material parameters in the calculation. The mesh file contains information about a pre generated mesh needed for the simulation. Several mesh file formats are available, e.g. `.msh, .mesh, .exocus, .cdb, .h5ref, ... .`

Execution/program call

From a user's point of view, the usual *openCFS* program call looks like Lst. 1.1 below with `<input>.xml` as "main".xml and `<output>` as name of the simulation.

```
> cfs -p <input>.xml <output>
```

Listing 1.1: openCFS console call

Apart from generating certain information on the terminal, *openCFS* will create a job.info.xml file containing specifics about the execution, and <output>.cfs within a folder named results_hdf5. This output can be visualized using an external tool called ParaView, which is an open-source post-processing visualization engine developed by kitware [9]. For more information about ParaView look up `www.paraview.org` [10].

When executing an *openCFS* simulation from within an IDE for example *Visual Studio Code*, it is best practice to add the path to *openCFS*' build to a launch.json file looking like this: `"program": "$workspaceFolder/.vscode/build-dir-link/bin/cfs"`. With this modification a simulation can be run inside *Visual Studio Code*.

1.2 Current state of NiHu

NiHu is an open source *C++* and Matlab toolbox used for solving boundary value problems of partial differential equations by means of the *boundary element method* (BEM) see Sec. 2.2. The core module of the toolbox provides a general framework to numerically evaluate weighted residual integrals. *NiHu* in its current version 2.0 provides a unified open source software framework for BEM problems of different kinds. It is capable to generate BEM executables with direct indirect BEM formalisms, with Galerkin, collocational and general solution methods in 2D and 3D and with element types defined in a general manner. Basically *NiHu* defines different BEM problems in a general way, by reflecting mathematics behind boundary elements in its *C++* code `last.hit.bme.hu` [5].

In contrast to *openCFS*, *NiHu* operates by executing a .c file which can be seen as equivalent to *openCFS*' "main".xml file. It contains all information such as material, mesh etc. needed for a proper simulation. As an elegant way to compile and execute such a .c file a simple *CMake* routine can be utilized which makes it compatible with the existing *CMake* structure of *openCFS*.

Executaion

The usual program call of a *NiHu* simulation looks like Lst. 1.2 below.

```
# compile the <input>.cpp file and execute an
# <output> executable
> g++ <input>.cpp -o <output> && ./<output>
```

Listing 1.2: NiHu execution (one line)

Here the compilation and the execution is prompted as one line of commands with && joining the two commands as can be seen in Lst. 1.3.

```
# compile the <input>.cpp file
> g++ <input>.cpp -o <output>

# execute the <output> executable
> ./<output>
```

Listing 1.3: NiHu execution (step by step)

For integrity g++ is the compiler itself which is called to compile a .cpp file, -o is a flag to name the resulting executable - else it would be named after the input file - and ./ is the path to an existing executable, presumed the compilation has worked properly.

For ease of use, a *NiHu* simulation can be built with *CMake* as shown in Lst. 1.4. This is exactly the way a calculation utilizing *NiHu* is called inside *openCFS*. *CMake*'s basic functionality and workflow is described in Sec. 3.2.

```
cmake_minimum_required (VERSION 3.15.0 FATAL_ERROR)

set(CMAKE_CXX_STANDARD 14)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)

message(STATUS "CMAKE_SOURCE_DIR: ${CMAKE_SOURCE_DIR}")
message(STATUS "CMAKE_BINARY_DIR: ${CMAKE_BINARY_DIR}")

project(simulation_NiHu)

get_filename_component(folder_name ${CMAKE_CURRENT_SOURCE_DIR} NAME)
set(source_file "${folder_name}.cc")
set(executable_name "${folder_name}_output")
add_executable(${executable_name} ${source_file})

target_compile_features(${executable_name} PRIVATE cxx_std_14)
target_compile_options(${executable_name} PRIVATE -fconcepts)

target_include_directories(${executable_name} PUBLIC ~/Devel/
    CFS_BIN/build_opt/cfsdeps/nihu/src/NiHu-install)
target_include_directories(${executable_name} PUBLIC ~/Devel/
    CFS_BIN/build_opt/cfsdeps/nihu/src/NiHu-install/include/)
target_include_directories(${executable_name} PUBLIC ~/Devel/
    CFS_BIN/build_opt/cfsdeps/nihu/src/NiHu-install/include/fmm)
target_include_directories(${executable_name} PUBLIC ~/Devel/
    CFS_BIN/build_opt/cfsdeps/nihu/src/NiHu-install/include/library)

target_include_directories(${executable_name} PUBLIC ~/Devel/
    CFS_BIN/build_opt/cfsdeps/nihu/src/NiHu/src/core)
target_include_directories(${executable_name} PUBLIC ~/Devel/
    CFS_BIN/build_opt/cfsdeps/nihu/src/NiHu/src/tmp)

set(NIHU_EXTERNAL_LIBRARY_PATH ~/Devel/CFS_BIN/
    build_opt/cfsdeps/nihu/src/NiHu-build/lib/nihu.a)
set(NIHU_FMM_EXTERNAL_LIBRARY_PATH ~/Devel/CFS_BIN/
    build_opt/cfsdeps/nihu/src/NiHu-build/lib/nihu_fmm.a)

add_library(nihu STATIC IMPORTED)
add_library(nihu_fmm STATIC IMPORTED)

set_target_properties(nihu PROPERTIES IMPORTED_LOCATION
    ${NIHU_EXTERNAL_LIBRARY_PATH})
set_target_properties(nihu_fmm PROPERTIES IMPORTED_LOCATION
    ${NIHU_FMM_EXTERNAL_LIBRARY_PATH})

target_link_libraries(${executable_name} nihu)
target_link_libraries(${executable_name} nihu_fmm)
```

Listing 1.4: NiHu CMake routine

1.3 Structure of this thesis

In the beginning the pre-existing states of the two numerical solvers *openCFS* [6] (*finite element method* Sec. 2.1) and *NiHu* [7] (*boundary element method* Sec. 2.2) are introduced. Both methods' mathematical functionality and structure are explained in chapter 2. Furthermore the way of implementing one solver as additional feature of the other, specifically adding *NiHu* [7] to *openCFS* [6] is handled by slightly modifying *openCFS*' pre-existing *CMake* (Sec. 3.2) structure. A detailed description of this task can be found in chapter 4. The *finite element method* (FEM) (Sec. 2.1) as well as the *boundary element method* (BEM) (Sec. 2.2) are detailed in chapter 2 including necessary tools such as *CMake.* After inclusion of a new feature it is of significance to test its most recent functionality on the one hand and to provide a so called *testcase* (Sec. 4.2.3) to show usage on the other hand. Therefore see chapter 5. The capabilities of *NiHu*'s addition to *openCFS* and the potentials of further working on this project are outlined in chapter 6.

2 Numerical Solution Methods for the Helmholtz Equation

This chapter explains fundamentals and the math behind two fashions of numerically solving partial differential equations - also referred to as PDEs - in physics and engineering, namely the finite element method (short FEM) Sec. 2.1 and the boundary element method (short BEM) Sec. 2.2 and shows similarities as well as differences between the two approaches.
Solving partial differential equations (PDEs) is the main goal of both numerical solution methods. With respect to wave propagation, the solution of the homogeneous Helmholtz equation Eq. 2.3 is the starting point for both the finite and the boundary element method. More precisely the homogeneous Helmholtz equation Eq. 2.3 is a time-independent form of the homogeneous acoustic wave equation Eg. 2.2 in frequency domain. The equation Eq. 2.4 represents the homogeneous Helmholtz equation Eq. 2.3 in *Einstein*'s notation. To further solve these "strong" forms of partial differential equations computationally a respective "weak" form has to be found. The following explanations and theory are mainly based on "Finite Elements for Computational Multiphysics" by Manfred Kaltenbacher[4] and the lecture notes to "Computational Acoustics"[8].

$$p_a = \hat{p}_a e^{j\omega t} \tag{2.1}$$

$$\left(\frac{1}{c_0^2} \frac{\partial^2}{\partial t^2} - \nabla \cdot \nabla \right) p_a = 0 \tag{2.2}$$

$$\nabla \cdot \nabla \hat{p}_a + k^2 \hat{p}_a = 0 \tag{2.3}$$

$$\left(k^2 + \frac{\partial^2}{\partial x_i^2} \right) \hat{p}_a(\mathbf{x}, \omega) = 0 \tag{2.4}$$

p_a is the acoustic pressure (time harmonic ansatz) (Eq. 2.1), c_0 is the speed of sound, $\omega = 2\pi f$ is the circular frequency, $k = \frac{\omega}{c_0}$ the wave number representing the periodicity in space of the modeled wave and $\mathbf{x} \in \Omega$.

2.1 Finite Element Method as implemented in *openCFS*

At first, a “strong” PDE, more precisely in case of an acoustic problem, the Helmholtz equation Eq. 2.3, is multiplied by a so-called test function w. This step is also called “testing”.

$$\begin{aligned} w\left(\nabla\cdot\nabla\hat{p}_a + k^2\hat{p}_a\right) &= 0 \\ \nabla\cdot\nabla\hat{p}_a\, w + k^2\hat{p}_a w &= 0 \end{aligned}$$

Secondly, this expression is integrated over the domain Ω.

$$\int_\Omega \nabla\cdot\nabla\hat{p}_a w d\Omega + k^2\int_\Omega \hat{p}_a w d\Omega = 0$$

Applying the integration by parts yields

$$\int_\Omega \nabla\hat{p}_a\cdot\nabla w d\Omega - \int_\Gamma \hat{p}_a\cdot\mathbf{n} w d\Gamma + k^2\int_\Omega \hat{p}_a w d\Omega = 0,$$

which still represents a continuous system in space still. Therefore the function spaces $\hat{p}_a \to \hat{p}_a^h$ and $w \to w^h$ as well as the computational domain $\Omega \to \Omega^h$ are being discretized. Summarizing the resulting equation the *Galerkin weak form* Eq. 2.5 is obtained.

$$\int_{\Omega^h} \nabla\hat{p}_a^h\cdot\nabla w^h d\Omega^h - \int_{\Gamma^h} \hat{p}_a^h\mathbf{n} w^h d\Gamma^h + k^2\int_{\Omega^h} \hat{p}_a^h w^h d\Omega^h = 0 \tag{2.5}$$

To compute the discretized Galerkin weak form (Eq. 2.5) it has to be formulated as an algebraic system (vector-matrix notation). Considering that the problem is defined over a discretized domain Ω^h, with surface Γ^h, and involving the test function w^h, the equation Eq. 2.6 expresses the Galerkin weak form as discrete matrix equation

$$k^2\boldsymbol{M}\boldsymbol{p} + \boldsymbol{K}\boldsymbol{p} = \boldsymbol{0} \tag{2.6}$$

with

- the stiffness matrix $\boldsymbol{K}$ representing the spatial derivatives of the test functions w^h, representing the term $\int_{\Omega^h} \nabla\hat{p}_a^h\cdot\nabla w^h\, d\Omega^h$,
- the mass matrix $\boldsymbol{M}$ representing the term $k^2\int_{\Omega^h} \hat{p}_a^h w^h\, d\Omega^h$,
- the boundary integral as the term $-\int_{\Gamma^h} \hat{p}_a^h\mathbf{n} w^h\, d\Gamma^h$,
- $\boldsymbol{p}$ the vector of unknowns

and

- k^2 as squared wave number.

2.2 Boundary Element Method as implemented in *NiHu*

In case of BEM the homogeneous Helmholtz equation in *Einstein*'s notation Eq. 2.4 represents the "strong" PDE to begin with. By approaching the solution of a PDE with BEM the goal is to exactly satisfy the equation in its domain and to approximate the solution at the boundary. So called *Green's Functions* $\hat{G}(\mathbf{x}|\mathbf{y}, w)$ represent fundamental solutions of the in-homogeneous wave equation.

$$\left(k^2 + \frac{\partial^2}{\partial y_i^2}\right)\hat{p}_a(\mathbf{y}, \omega) = 0 \qquad | \cdot \hat{G}(\mathbf{x}|\mathbf{y}, \omega) \tag{2.7}$$

$$\left(k^2 + \frac{\partial^2}{\partial y_i^2}\right)\hat{G}(\mathbf{x}|\mathbf{y}, \omega) = \delta(\mathbf{x} - \mathbf{y}) \qquad | \cdot \hat{p}_a \tag{2.8}$$

By subtracting equation Eq. 2.8 from Eq. 2.7

$$\left(\hat{p}_a \frac{\partial^2}{\partial y_i^2}\hat{G} - \hat{G}\frac{\partial^2}{\partial y_i^2}\hat{p}_a\right) = \delta(\mathbf{x} - \mathbf{y})\hat{p}_a(\mathbf{y}, \omega)$$

is obtained which will be integrated over the whole domain Ω.

$$\hat{p}_a(\mathbf{x}, \omega) = \int_\Omega \left(\hat{p}_a \frac{\partial^2}{\partial y_i^2}\hat{G} - \hat{G}\frac{\partial^2}{\partial y_i^2}\hat{p}_a\right) d\mathbf{y}$$

After applying the integration by parts and point collocation the following equation Eq. 2.9 describes the resulting **Boundary Integral Equation**.

$$\hat{p}_a(\mathbf{x}_i, \omega) = \int_\Gamma \left(\hat{p}_a(\mathbf{y}, \omega)\frac{\partial}{\partial y_i}\hat{G}(\mathbf{x}|\mathbf{y}) - \hat{G}(\mathbf{x})|\mathbf{y})\frac{\partial}{\partial y_i}\hat{p}_a(\mathbf{y}, \omega)\right) n_i d\Gamma(\mathbf{y}) \tag{2.9}$$

2.3 Comparison: FEM & BEM

Finite element method

- Mesh over the entire domain
- Sparse but large system matrix
- Problem is solved throughout the entire volume

Boundary element method

- Mesh only on the surface
- Small but densely populated system matrix
- Problem is solved on the boundary surfaces
- Further solutions are determined in post-processing (low computational effort)

3 Software Framework and Programming Basics

Due to the fact that this project's nature is software development aside of numerical solution the basics of how a project such as this is structured and executed are described in this section. A demonstration of how to build a basic `hello_world.cpp` program in C++ (Sec. 3.1) by using CMake (Sec. 3.2) is provided too, as well as a rudimentary description of Gitlab (Sec. 3.3) and its functionality.

3.1 Back End: C++

C++ is a powerful, statically typed, compiled language that is widely used for developing operating systems, browsers, games, and applications requiring high-performance computation and reliability. C++ builds on C, offering object-oriented features such as classes, polymorphism, encapsulation, and inheritance, alongside template programming, which allows for writing generic and reusable code. It is extremely versatile due to its support of both procedural and object-oriented programming paradigms.
A simple `hello_world.cpp` below shall illustrate the looks and general outline of C++.

```cpp
#include <iostream>
using namespace std;

void helloWorld()
{
    cout << "Hello World!" << endl;
}

int main()
{
    helloWorld();
    return 0;
}
```

Listing 3.1: basic `hello_world.cpp`

3.2 CMake

CMake is a makefile generator to simplify a build process for developing projects across different compilers and platforms. It provides a powerful, easy-to-use platform and compiler-independent build system that simplifies the compilation and linking of code across different environments. Its main features are listed below.

- cross-platform compatibility
- compiler independence

- automatic package finding
- testing and packaging support

CMake simplifies the compilation and linking of code across different environments. Instructed by configuration files named `CMakeLists.txt` it generates native build scripts or *makefiles* for a wide variety of compilers.
For example the `hello_world.cpp` (Lst. 3.1) could be built by utilizing CMake as follows. In Lst. 3.2, the minimally required information and commands needed to handle a build using CMake are displayed.

```
# Define the minimum version of CMake
cmake_minimum_required(VERSION 3.10)

# Define your project name and version
project(HelloWorldProject VERSION 1.0)

# Specify the C++ standard
set(CMAKE_CXX_STANDARD 11)

# Add an executable to the project
add_executable(hello_world, hello_world.cpp)
```

Listing 3.2: minimum requirements for `CMakeLists.txt`

The console call for an entire build can be executed as simply as shown in Lst. 3.3 below. The additional configuration step deployed with the command `ccmake` is not a mandatory step. If not used *CMake* will build a program's default configuration. Sometimes a program does not even provide individual customization and this step would be redundant.

```
# Generate makefile and project configuration files
# for the project
> cmake /path/to/project

# Open a pseudo graphical interface to configure the
# project's CMake options interactively
> ccmake /path/to/project  # additional configuration

# Compile the project based on the generated makefile,
# building the executables or libraries
> make                     # actual build process
```

Listing 3.3: CMake console commands

3.3 Version Control: Gitlab

Gitlab is a web-based software development and operations life cycle management tool that provides a Git repository manager providing wiki, issue-tracking, and CI/CD pipeline features. It is designed to help teams collaborate on software development throughout the entire project life cycle, from idea to production and it offers a wide range of features designed to support the development process, including but not limited to:

- **Version Control:** Utilizes Git for source code management, allowing for branching, merging, and version tracking.
- **Continuous Integration/Continuous Deployment (CI/CD):** Automates the process of software testing and deployment, helping teams to detect bugs and release software faster.

- **Issue Tracking:** Provides tools for tracking bugs and managing project tasks and user feedback.
- **Code Review:** Facilitates code review and commentary on commits and merge requests.
- **Wiki:** Enables teams to maintain project documentation in a collaborative manner.
- **Container Registry:** A secure and private registry for Docker images, integrated into the GitLab CI/CD process.
- **Security and Compliance:** Offers features to enhance the security of the codebase and ensure compliance with various standards.

GitLab's basic functionality revolves around the management of Git repositories. It allows users to create and manage repositories, track issues, review code, and collaborate on software projects. The integrated CI/CD pipeline automates the process of testing and deploying software, making it easier to maintain high-quality code standards.
Below are some of the basic Git commands used within GitLab for version control and collaboration.

```
git clone <repository-url>           # Clone a repository into a new directory
git branch                           # List, create, or delete branches
git checkout <branch-name>           # Switch branches or restore working tree files
git add <file>                       # Add file contents to the index
git commit -m "Commit message"       # Record changes to the repository
git push <remote> <branch>           # Update remote refs along with associated objects
git merge <branch>                   # Join two or more development histories together
git pull                             # Fetch from and integrate with another
                                     # repository or a local branch
git fetch <remote>                   # Download objects and refs from another repository
git rebase <basebranch>              # Apply changes from one branch on top of another
git pull --recurse-submodules        # Fetch from and integrate with another
                                     # repository or a local branch, including submodules
```

A usual git call in case of development on *openCFS* could look as follows in Lst. 3.4.

```
# pull latest version from master branch
> git checkout master

# pull the submodule "Testsuite" as well
> git pull --recurse-submodules

# development is done on a feature branch and merged into the master
# branch by a maintainer after code review
> git checkout <feature branch>
```

development or maintenance of source code

```
# Add a new or modified file to the staging area
> git add <newly added or modified file>

# Commit the staged changes with a message
> git commit -m "commit message"

# Push the commit to the remote repository
> git push
```

Listing 3.4: git commands

What is special in Lst. 3.4 is the way the latest version of the master branch is pulled. The flag `-recurse-submodules` simply tells git to pull a sub repository within.

4 Workflow Integration of *NiHu* into *openCFS*

Firstly, *openCFS* [6] is cloned from Git [3] and secondly built locally. Furthermore *NiHu* is cloned from its website [5] and built separately just as its instructions explain, to get to know its build process. This approach will then be included in *openCFS*' CMake build procedure later on. Obviously, the majority of this task takes place in the CMake environment as well as in the C++ back end of *openCFS* particularly in its main *CMakeLists.txt* as shown in Sec. 4.1.3 and its dependencies' directory *cfsdeps* desdcribed in Sec. 4.1.2. After *NiHu* has been successfully integrated into *openCFS* [6] the new functionality can be demonstrated and tested via testcases enlisted in Sec. 4.2.3 which are implemented in the course of this project.

In this particular case the emphasis is on acoustic simulations. Furthermore the basic build and install process is described in conjunction with cross platform compiling of software with particular attention on a C++ back end as described in Sec. 3.1. This is described in great detail such that further work should possibly be picked up quite easily.

4.1 Integration of *NiHu* into the build process of *openCFS*

Due to *openCFS* relying on an `input.xml` file to contain needed information about a simulation on the one hand and *NiHu* on the other hand getting its necessary data from a `input.c` file, calling NiHu from inside of *openCFS* demands a customized kind of call. While a BEM simulation is executed by utilizing a class named `AcousticPDE_BEM` it basically is a function which performs a few console commands to start a respective *CMake* routine to compile and further run the resulting executable.

For *openCFS* to be built and installed properly a couple of dependencies inside and outside the *C++* code itself are required. Therefore when adding a new feature the size of *NiHu*, first of all the main `CMakeLists.txt` must be modified such that it knows of the new feature and where to look for the dependencies needed. This modification demands an additional `External_NiHu.cmake` located in `cfs/cfsdeps/nihu` containing all information needed to clone and add the external project [1] that *NiHu* is. Not every *openCFS* user wants to add a boundary element method solver to their version of the software. Therefore a switch is added to the main `CMakeLists.txt` to allow the user to add *NiHu* in the configuration step of *openCFS* or not. This can be handled in the configuration step `> ccmake` (Sec. 3.2) during the `CMake` build process. Enabling `USE_NIHU` in the `ccmake` configuration step automatically enables `USE_EIGEN` since *NiHu* requires the `Eigen` library which is an optional library of *openCFS* that is disabled by default. Lst. 4.1 shows the changes in *openCFS*' main `CMakeLists.txt`.

```
# NiHu - BoundaryElementMethod (TU Budapest)
option(USE_NIHU "NiHu (BoundaryElementMethod) solver - automatically
    enables USE_EIGEN since NiHu requires the Eigen library."
    ${USE_NIHU_DEFAULT})

# NiHu conditional
if(USE_NIHU)
    # automatically install Eigen with NiHu
    if (NOT USE_EIGEN)
        set(USE_EIGEN ON CACHE BOOL "Install Eigen as a required
        dependency of NiHu." FORCE)
        message(STATUS "Automatically enabling USE_EIGEN because
        NiHu requires the Eigen library.")
    endif()

    # include NiHu configuration file
    include(cfsdeps/nihu/External_NiHu.cmake)
endif()
```

Listing 4.1: main CMakeLists.txt

In Lst. 4.1 above `${USE_NIHU_DEFAULT}` is a default variable representing `OFF` in a dedicated file that hold various default variables such as `${USE_EIGEN_DEFAULT}`. The second last line includes `External_NiHu.cmake`. Its explanation can be found at Sec. 4.2.

Furthermore the code itself has to be adapted. Firstly a new PDE class is added so *openCFS* knows how to interpret a call for a boundary element method solution from an `input.xml`. Secondly the class `Domain.cc` must be adjusted to both call the right PDE (`AcousticPDE_BEM`) and terminate (Sec. 4.2.1) the whole program suitably. As mentioned already, a new PDE class named `AcousticPDE_BEM.cc` including `AcousticPDE_BEM.hh` is introduced which calls the newly added *NiHu* boundary element method functionality.

Each new testcase's addition to the build process has to be enlisted to the whole *CMake* process too.

4.1.1 PDE class: AcousticPDE_BEM

At this point of development the "pseudo" partial differential equation class `AcousticPDE_BEM.cc` and `AcousticPDE_BEM.hh` does not inherent from `SinglePDE.cc` and `SinglePDE.hh` but rather is a method that calls a *CMake* routine to compile and execute a *NiHu* simulation.

4.1.2 *cfsdeps*: Dependency library for *openCFS*

In its entirety *openCFS* stores all of its dependencies - its dependencies' properties - in a directory named `cfsdeps` which is a sub folder of the main folder `cfs` at which the main `CMakeLists.txt` expects all external dependencies with one folder called `nihu` among all the others containing all information for *NiHu* to be built within *openCFS*.

```
# Builds external third party projects.
# The parent script CMakeLists.txt defines the "GLOBAL_OUTPUT_PATH"
# variable, which will be used as output directory for all *.lib,
# *.dll, *.a, *.so, *.pdb files.

###################################################################
# NiHu
###################################################################

set(NIHU_SRC_DIR ${CMAKE_BINARY_DIR}/cfsdeps/nihu/src/NiHu/src)

include(ExternalProject)

ExternalProject_Add(
  NiHu

  # external project's base path
  PREFIX ${CMAKE_BINARY_DIR}/cfsdeps/nihu
  GIT_REPOSITORY "git://last.hit.bme.hu/toolbox/nihu.git"
  GIT_TAG "nightly"

  UPDATE_COMMAND "" # specify if needed
  PATCH_COMMAND ""  # specify if needed

  ## path to NiHu's CMakeLists
  CMAKE_ARGS -H${CMAKE_BINARY_DIR}/cfsdeps/nihu/src

  CMAKE_ARGS -H${NIHU_SRC_DIR}
             -DNIHU_INSTALL_DIR=${CMAKE_BINARY_DIR}/cfsdeps/
                nihu/src/NiHu-install
             -DNIHU_EIGEN_INSTALL=1
)
```

Listing 4.2: External_NiHu.cmake

The listing (Lst. 4.2) above shows how the `external_nihu.cmake` script is used for integrating an external project into a larger *CMake* build system. It makes use of the `ExternalProject_Add` command to download, update, and build the project. First of all the source directory is set with `set(NIHU_SRC_DIR $CMAKE_BINARY_DIR/cfsdeps/nihu/src/NiHu/src)`. This line sets the `NIHU_SRC_DIR` variable to the path where *NiHu*'s source code will be located, relative to the build directory. After that the `ExternalProject` module, which provides the `ExternalProject_Add` function is included, a feature of *CMake* which allows for the inclusion and building of external projects within a *CMake* build process.

Below the content of listing Lst. 4.2 is described in detail.

- `NiHu`: The name of the external project.
- `PREFIX`: The directory where the project will be downloaded and built.
- `GIT_REPOSITORY`: The URL of the Git repository.
- `GIT_TAG`: The specific branch, tag, or commit to checkout.
- `UPDATE_COMMAND` and `PATCH_COMMAND`: Commands for updating and patching, left empty here.
- `CMAKE_ARGS`: Additional arguments passed to *CMake* during the build, specifying the source directory and installation directory.

The `external_nihu.cmake` script is crucial for automating the download, build, and integration of the *NiHu* library into the larger *openCFS* project using CMake.

Looking closely at the listing (Lst. 4.2) one can see how the way the external software is already making use of another recently added dependency `Eigen` (see `-DNIHU_EIGEN_INSTALL=1`). If *openCFS* did not have its own `Eigen` library it would additionally have to clone this very dependency.

4.1.3 CMakeLists

An essential *CMake* requirement is its `CMakeLists.txt` located in the `cfs` folder itself which handles the whole build's overhead by calling sub-CMakeLists and other `.cmake` files, setting up the folder structure and each dependency's location as well as generating the testsuite and all its testcases. Further possible overrides of the user during configuration by executing `ccmake` is being processed there too.

4.2 Integration of *NiHu* into the workflow of *openCFS*

Having cloned, configured and built *openCFS* including *NiHu* as explained in chapter 4 (Sec. 5.1) in more detail makes it possible to solve problems or provided testcases (Sec. 5.2) by applying the boundary element method (Sec. 2.2). Running a BEM simulation inside *openCFS* will result in calling the respective partial differential equation class for the acoustic boundary element method named *AcousticPDE_BEM* which handles the execution of *NiHu*'s functionality by building and running the corresponding `simulation.cc` file according to the instructions provided either via *CMakeLists.txt* or flags within a command line call.
The program's `main()` function basically consists of a couple of lines of code and looks similar to Lst. 4.3 which shows that at its core the object `cfs` is initiated and the method `Run()` of this object is called which mainly performs the following steps.

- `ReadXMLFile();`
- `domain = new Domain( gridInputs, resultHandler, materialHandler, simState, paramNode_, infoNode );`
- `domain->CreateGrid();`
- `SolveProblem();`

At first the `simulation.xml` is read and parsed. At second the domain is set up. At third the data from the grid input files is read. Then the problem is being solved.

```
int main(int argc, const char **argv)
{
  CFS cfs(argc, argv);
  int ret = cfs.Run();
  return ret;
}
```

Listing 4.3: function main() in CFS.cc

Knowing *openCFS* receives all information about a simulation within an `.xml` file which is processed on program start, first of all it will try to find the appropriate PDE tag for example the `<acoustic>`. In case it detects the tag `<AcousticBEM>` it will further call the corresponding “pseudo” PDE class `AcousticPDE_BEM.cc`

```
< ... >
    <pdeList>
        <acousticBEM>
            < ... ></ ... >
        </acousticBEM>
    </pdeList>
 </... >
```

in its PDE classes located in `cfs/source/PDE/AcousticPDE_BEM.cc`. For the greater part the class `Domain.cc` takes care of passing information and calling appropriate functions and classes.

Although *openCFS* usually would instantiate the appropriate PDE class and step by step compute the simulation according to the information provided by the `main.xml` file as well as the material and mesh files, in case of receiving `<acousticBEM>` it will just look for an `input.cc` file in a relative *NiHu* path or directory and build and run it via an attached `CMakeLists.txt`. This fundamentally distinguishes the “pseudo” PDE class `AcousticPDE_BEM.cc` from `AcousticPDE.cc` and all the other PDE classes. Which means that *openCFS*’ program call has to terminate differently as well as discussed in the following.

4.2.1 Termination after BEM simulation (program end)

Since *openCFS* works in a particular way by calling appropriate PDEs to match the information provided via the `input.xml` file its termination - further, the function responsible for a clean program end - is in need of several variables too. Now calling the class `AcousticPDE_BEM.cc` deploys a *CMake* routine to solve a given *boundary element* problem and therefore does not need *openCFS* to do any storing or processing itself. *openCFS* is still in need of a *mesh file* and a *material file* to properly start and terminate a simulation.

4.2.2 XML Scheme

XML (Extensible Markup Language) is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. It provides a framework for data description and is mainly used for data storage and transport as well as configuration and initialization of applications. Its main features are listed below.

- **Extensibility**: XML allows for the creation of custom tags, enabling users to tailor data markup to their specific requirements.
- **Self-Descriptive Nature**: XML documents inherently provide a clear description of their content, enhancing understandability and accessibility.
- **Unicode Support**: XML’s compatibility with Unicode standards ensures it can represent text in virtually any written language, making it universally applicable.
- **Data Sharing Efficiency**: XML excels in facilitating the seamless exchange of data between disparate systems, promoting interoperability.
- **Content-Presentation Distinction**: By emphasizing the structure and meaning of data over its presentation, XML ensures that data management is both flexible and efficient.

In case of **openCFS** all information such as analysis type, type of PDE, etc. and required external files like material and mesh files needed to perform a simulation is stored in an e.g. `<simulation_name.xml>` file. Listing Lst. 4.4 provides the basic XML commands.

```
<!-- XML declaration -->
<?xml version="1.0" encoding="UTF-8"?>

<!-- elements -->
<tagname>content</tagname>

<!-- attributes -->
<element attribute="value">

<!-- comments -->
<!-- This is a comment -->
```

Listing 4.4: basic XML commands

A new tag has been added to the `.xds` Scheme: `<AcousticBEM>`. So, whenever *openCFS* detects this tag it will look for a corresponding `.c` file and compile and execute it.

4.2.3 Testsuite

openCFS' Gitlab (Sec. 3.3) repository contains the sub-repository "Testsuite" which holds a variety of testcases, to test its functionality. Therefore, testcases concerning the new possibility of solving boundary element problems have been added as well, on the one hand to demonstrate the new feature and on the other hand to explain its usage. The corresponding testcase is being described in detail in Sec. 5.2.

5 Usage and Testing

This chapter is the manual for the newly added feature, namely *NiHu* [7]. Firstly, an instruction to the whole process from *cloning* a git repository to the installation of the entire software is provided (see Sec. 5.1). Secondly, the usage of *openCFS* [6] including *NiHu* [7] will be explained (see Sec. 5.1). Thirdly, the workflow of actual simulations are displayed on the basis of a provided testcase.

5.1 Installation of *openCFS* with *NiHu*

The basic *openCFS* setup is explained on the corresponding *GitLab*'s "wiki" which is the main manual and can be found at `gitlab.com/openCFS/cfs/-/wikis` [2]. As described there *openCFS* is cloned and build as follows.

Before getting the source code it is recommended to set up a folder structure as shown in the table below where `CFS_SRC` will contain all dependencies needed for the build itself and `CFS_BIN` will then hold the installed software. Therefore the *openCFS* repository is cloned into `CFS_SRC` which could be understood as a download. The basic build is to be stored in a `build_opt`. Just like the additional BEM feature it is obviously possible to make a debug build which could be built into a folder like `build_debug` and be utilized while working on *openCFS*' source code.

- `~/Devel`
    - `CFS_SRC`
    - `CFS_BIN`
        - `build_opt`
        - `build_debug`

Then, as shown in Sec. 3.2, from within `~/Devel/CFS_BIN/build_opt` proceeding with Lst. 5.1 results in installing *openCFS* ready to use.

```
> cmake ~/Devel/CFS_SRC/cfs
> ccmake ~/Devel/CFS_SRC/cfs
> make
```

Listing 5.1: openCFS installation

As for choosing to set up a debug build e.g. "If you want to have a debug-build, you have to switch `DEBUG=ON` in `ccmake`." [2] Including *NiHu* in an *openCFS* build works just the same. One thing to keep in mind is that *NiHu* requires one additional dependency, *Eigen*, to work properly. Otherwise it will clone an external and redundant *Eigen* library.

So switching `BUILD_NIHU=ON` and `EIGEN=ON` is everything it needs to add the BEM functionality to *openCFS*.

[**!**] At this point of implementation *NiHu*'s `CMakeLists.txt` expects its dependencies to be located in a relative directory to `build_opt`. In case a user wanted to use *NiHu* as a feature of a debug build, it is to be adjusted there.

5.2 Explanation of usage by taking the example of a testcase

Below a testcase is used to demonstrate how a simulation is to be set up in order to solve a *boundary element* problem using *NiHu* as an external feature of *openCFS*. The way the `CMakeLists.txt` responsible for building and running a *BEM* simulation is constructed, prescribes a particular folder structure as explained in Sec. 5.2.1. How a proper simulation will then be executed and terminated is shown with a testcase as example in Sec. 5.2.2 and Sec. 5.2.3.

5.2.1 Folder structure

A simulation or testcase needs a *CMake* routine too and must have the following folder structure.

- `/laplace_2d_transparent_standalone`
  - `mat.xml`
  - `laplace_2d_transparent_standalone.h5`
  - `laplace_2d_transparent_standalone.xml`
  - `/laplace_2d_transparent_standalone_NiHu`
    - `CMakeLists.txt`
    - `laplace_2d_transparent_standalone_NiHu.cc`
    - `/build`
      - ...

The `CMakeLists.txt` file will basically look the same for any simulation or testcase since it links the simulation with the necessary libraries needed for calculation. *openCFS* runs minimum one sequence step during which the *NiHu* simulation is being executed and the first thing *openCFS* does every time is reading the `simulation.xml` file. This `simulation.xml` file requires input such as which *material file* and *mesh file* it shall look for to properly start and execute an *openCFS* simulation. At this point of implementation the `.xml/.xsd` Scheme and *openCFS*' corresponding backend have not been modified such that it would not need input files such as the *material file* and the *mesh file* so these are provided but will not be used for any kind of calculation.

5.2.2 Execution

As shown in Lst. 1.1 a simulation or testcase is executed as shown in Lst. 5.2.2.

```
cfs -p laplace_2d_transparent_standalone.xml output
```

First of all *openCFS* reads the `laplace_2d_transparent_standalone.xml` file and if it detects the tag `<AcousticBEM>` it calls the corresponding "pseudo" PDE class `AcousticPDE_BEM.cc` and start the *NiHu* calculation depending on the `CMakeList.txt` and the `laplace_2d_transparent_standalone_NiHu.cc` files contained in the `laplace_2d_transparent_standalone_NiHu` folder.

NiHu then calculates the problem by compiling the C++ file. Afterwards it executes the resulting executable, prints output to the console and creates a folder called `build` which it stores output files in depending on the C++ file's content.

5.2.3 Termination

After having provided solutions and output of the testcase *NiHu* terminates. If there is no other PDE tag in the `.xml` file *openCFS* leaves the testcase's *NiHu* directory and terminates as well.

5.3 State of implementation

At this stage it is possible to build *openCFS* including the external feature *NiHu* and utilize it to solve *boundary element* problems within *openCFS*. How the build process works is illustrated in Sec. 5.3.1 and a visualization of the workflow is shown in Sec. 5.3.2. The relevant code for the implementation is listed in Appendix A.

5.3.1 Build process

NiHu's implementation in *openCFS*' build process is neatly executed and should provide a tidy base for future work on the topic of adding a *BEM* solver to *openCFS*. It works as shown in Fig. 5.1 below. *openCFS* is cloned from *Gitlab* (see Sec. 3.3) and built utilizing *CMake* (see Sec. 3.2). During the additional configuration step `ccmake` *NiHu* can be set as additional extension to be installed with *openCFS*.

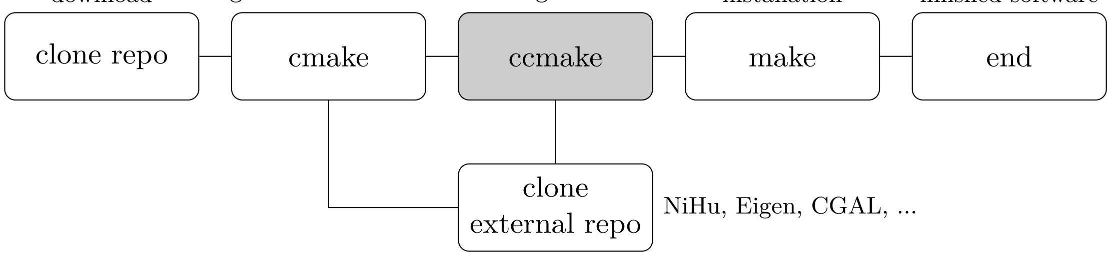

Figure 5.1: build process integration

5.3.2 Workflow

The fact that it is added as external feature to *openCFS*' workflow and it is not to be modified since its source code is simply cloned from a separate repository small compromises have been made during development. In the future, one could take *NiHu*'s source code and directly integrate it in *openCFS*' workflow and the corresponding `.xml` Scheme but until then *openCFS* needs to at least run one sequence step during which NiHu is being executed. It is in need of a *material file* as well as a *mesh file* still, although they are only input files which do not serve any other purpose than being necessary for its proper termination. It did not seem wise modifying too much of *openCFS*' deeper backend on the one hand. On the other hand *NiHu* is supposed to be added as external feature and as that it can only be treated as the “black box” it is. The way the *boundary element method* solver has been implemented is sufficient given its restrictions and mannerisms respectively. Fig. 5.2 below demonstrates *NiHu*'s workflow integration.

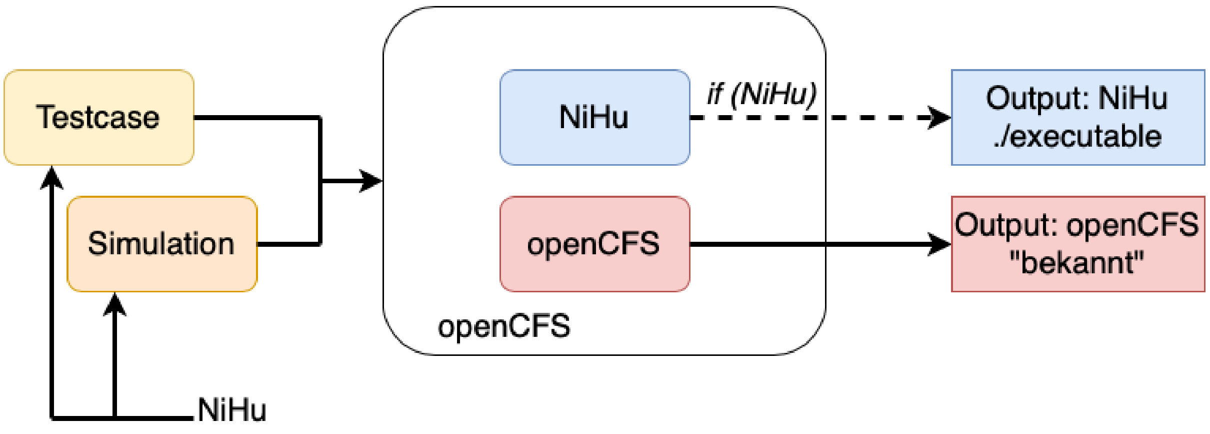

Figure 5.2: workflow integration

6 Discussion

6.1 Conclusion

The fact that *openCFS* and *NiHu* do not operate on similar input patterns meaning *NiHu* does not base its input on an `.xml` scheme as *openCFS* the applied method of implementation was chosen because of simplicity and its straight forward nature despite its possible shortage of elegance, since the main constraints were not to change *NiHu*'s source code.
As shown in this report it is possible to integrate a BEM software into a pre-existing FEM framework. Though external `.cc` files are compiled on runtime the present result provides a working solution as a basis for possible further work with room for improvement. Additionally the documentation as summarized here can contribute to developers' work in various forms in general, ranging from basics regarding the build process utilizing *CMake* to version control manners using *Gitlab*, not to forget finite element and boundary element method's theoretical background as well as their workflow and implementation.

6.2 Outlook

In the future, developing an `.xml` parser allows to integrate *NiHu* into *openCFS*' workflow right away. This would result in not having to compile *NiHu* simulations on runtime. Introducing a new type of program requires a framework that remains untouched, and modifying the existing `.xml` reader, which is one of the essential components of *openCFS*, presents a distinct challenge. But due to *NiHu* being an external "black box" there is no more efficient way of implementing it into *openCFS*' build process and workflow at this point. However, the current implementation offers a solid starting point for several reasons. Firstly, *openCFS* can be constructed to include *NiHu*. Secondly, the properties and dependencies of *NiHu* can be identified and utilized during *openCFS*' runtime. Thirdly, all details regarding how *NiHu* performs a simulation are outlined in the corresponding testcase. Last but not least, this documentation shall be a fundamental summary of *openCFS*' build process and its workflow and help with future work on this or similar topics.

6.2.1 Coupling

Despite the way of *NiHu*'s current state and way of implementation possible projects and/or ideas for future development shall be explained below.

The finite element method (FEM) and boundary element method (BEM) coupling, or FEM-BEM coupling, is a hybrid numerical technique used to solve complex engineering problems. It combines the strengths of both FEM and BEM to provide efficient solutions for problems involving unbounded domains.

Here are each method's advantages and how they are being used when coupled.

- FEM is used for handling the interior domain, where the problem is discretized into a finite number of elements leading to a system of algebraic equations.
- BEM is applied to the boundary of the domain, reducing the dimensionality of the problem by one, and is particularly effective for unbounded domains.
- Coupling occurs by enforcing continuity conditions on the interface between the FEM and BEM regions, ensuring that both displacement and tractions (or their equivalents in other physical fields) match across the boundary.

The process involves three main steps:

1. Discretize the problem domain for using FEM in the interior and BEM on the boundary.
2. Apply the boundary conditions and continuity conditions at the interface between FEM and BEM regions.
3. Solve the coupled system of equations, typically using iterative or direct solvers, to find the unknowns in both FEM and BEM domains.

Forward Coupling

Forward coupling in FEM-BEM in particular is a strategy where the solution progresses from one method to the other, typically from FEM to BEM.

- Initiation of the solution process within the FEM domain, solving for the interior domain.
- Using the FEM results on the boundary to serve as inputs or boundary conditions for the BEM analysis.
- Solving the BEM part to find field values on and beyond the boundary, completing the solution process.

For all practical purposes it works as follows.

1. Solve the interior problem using FEM to obtain boundary values.
2. Transfer these values as boundary conditions to the BEM model.
3. Solve the BEM model to extend the solution to the exterior domain or to capture far-field effects.

Conclusion: Forward coupling allows a comprehensive analysis of an interior problem and its interaction with the surrounding environment.

A Appendix - Source Code

A.1 Class AcousticPDE_BEM

A.1.1 AcousticPDE_BEM.hh

```
#ifndef ACOUSTICPDE_BEM_HH
#define ACOUSTICPDE_BEM_HH

class AcousticPDE_BEM
{
  public:
    static void callNiHu();
};

#endif
```

Listing A.1: AcousticPDE_BEM.hh

A.1.2 AcousticPDE_BEM.cc

```
#include "AcousticPDE_BEM.hh"

#include <boost/lexical_cast.hpp>
#include <boost/filesystem.hpp>
#include <cmath>

#include "General/defs.hh"

void AcousticPDE_BEM::callNiHu()
{
  // output prefix
  const char* prefix_nihu = "++ NIHU >> ";
  const char* error_nihu = "++ NIHU_ERROR >> ";

  // Get the current working directory
  boost::filesystem::path current_directory = boost::filesystem::current_path();

  // Get the name of the current directory
  std::string directory = current_directory.stem().string();

  // Set the output directory based on the current directory name
  std::string output = directory + "_NiHu_output";


  if (chdir((directory + "_NiHu").c_str()) != 0)
  {
    std::cerr << error_nihu << "no directory " + directory + "_NiHu" << std::endl;
    return;
  }

  std::cout << prefix_nihu << "generate directory build" << std::endl;

  if (std::system("mkdir -p build") != 0)
  {
    std::cerr << error_nihu << "cannot generate directory build" << std::endl;
    return;
  }

  std::cout << prefix_nihu << "opening directory build" << std::endl;

  if (chdir("build") != 0)
  {
    std::cerr << error_nihu << "no directory build" << std::endl;
    return;
  }

  std::cout << prefix_nihu << "calling cmake" << std::endl;

  if (std::system("cmake ..") != 0)
  {
    std::cerr << error_nihu << "cannot call cmake properly" << std::endl;
    return;
  }

  std::cout << prefix_nihu << "building simulation" << std::endl;

  if (std::system("make") != 0)
  {
    std::cerr << error_nihu << "cannot execute make properly" << std::endl;
    return;
  }

  std::cout << prefix_nihu << "open result(executable)" << std::endl;

  if (std::system(("./" + output).c_str()) != 0)
  {
    std::cerr << error_nihu << "no executable found" << std::endl;
    return;
  }

  std::cout << prefix_nihu << "leaving directory NiHu/build" << std::endl;

  if (chdir("../..") != 0)
  {
    std::cerr << error_nihu << "possible segmentation fault" << std::endl;
    return;
  }
}
```

Listing A.2: AcousticPDE_BEM.cc

A.2 XML-Scheme `CFS_PDEacousticBEM.xsd`

```
  <?xml version="1.0" encoding="UTF-8"?>
  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.cfs++.org/simulation"
    xmlns="http://www.cfs++.org/simulation"
    xmlns:cfs="http://www.cfs++.org/simulation"
    elementFormDefault="qualified">

    <xsd:annotation>
      <xsd:documentation xml:lang="en">
        Coupled Field Solver project openCFS
        Schema for PDE description for an acoustics PDE BEM
      </xsd:documentation>
    </xsd:annotation>


    <!-- ******************************************************************* -->
    <!--   Definition of element for acoustics BEM PDEs -->
    <!-- ******************************************************************* -->
    <xsd:element name="acoustic_BEM" type="DT_PDEAcousticBEM"
      substitutionGroup="PDEBasic">
      <xsd:unique name="CS_AcousticBEMRegion">
        <xsd:selector xpath="cfs:region"/>
        <xsd:field xpath="@name"/>
      </xsd:unique>
    </xsd:element>


    <!-- ******************************************************************* -->
    <!--   Definition of data type for acoustics PDEs -->
    <!-- ******************************************************************* -->
    <xsd:complexType name="DT_PDEAcousticBEM">
      <xsd:complexContent>
        <xsd:extension base="DT_PDEBasic">
          <xsd:sequence>

            <!-- Regions the PDE lives on -->
            <xsd:element name="regionList" minOccurs="0" maxOccurs="0">
              <xsd:complexType>
                <xsd:sequence>
                  <xsd:element name="region" minOccurs="1" maxOccurs="unbounded">
                    <xsd:complexType>
                      <xsd:attribute name="name" type="xsd:token" use="optional"
                      default="sector"/>
                      <xsd:attribute name="polyId" type="xsd:string" use="optional"
                      default="default"> </xsd:attribute>
                      <xsd:attribute name="integId" type="xsd:string" use="optional"
                      default="default"
                        > </xsd:attribute>
                      <xsd:attribute name="nonLinId" type="xsd:token" use="optional"
                      default=""/>
                      <xsd:attribute name="flowId" type="xsd:token" use="optional"
                      default=""/>
                      <xsd:attribute name="temperatureId" type="xsd:token"
                      use="optional" default=""/>
                      <xsd:attribute name="dampingId" type="xsd:token" use="optional"
                      default=""/>
                      <xsd:attribute name="lambId" type="xsd:token" use="optional"
                      default=""/>
                      <xsd:attribute name="initialFieldId" type="xsd:token"
                      use="optional" default=""/>
                      <xsd:attribute name="initialFieldD1Id" type="xsd:token"
                      use="optional" default=""/>
                      <xsd:attribute name="complexFluid" type="DT_CFSBool"
                      use="optional" default="no"/>
                    </xsd:complexType>
                  </xsd:element>
                </xsd:sequence>
              </xsd:complexType>
            </xsd:element>

            <!-- Desired solution values (optional) -->
            <!-- Currently this is not supported, since it is difficult
            to define in the acoustics case since it depends on the
            boundary conditions -->
            <xsd:element minOccurs="0" name="impedanceList">
              <xsd:complexType>
                <xsd:choice maxOccurs="unbounded">
                  <xsd:element maxOccurs="1" minOccurs="0" name="mpp" type="DT_Mpp"/>
                  <xsd:element maxOccurs="1" minOccurs="0" name="interpolImpedance"
                    type="DT_InterpolImpedance"/>
                  <xsd:element maxOccurs="1" minOccurs="0" name="fctImpedance"
                  type="DT_FctImpedance"
                  />
                </xsd:choice>
              </xsd:complexType>
            </xsd:element>
            <xsd:element name="storeResults" type="DT_AcousticStoreResults"
            minOccurs="0"
              maxOccurs="1"/>

          </xsd:sequence>

          <!-- Type of acoustic formulation (optional) -->
          <xsd:attribute name="formulation" use="optional" default="acouPressure">
            <xsd:simpleType>
              <xsd:restriction base="xsd:token">
                <xsd:enumeration value="acouPressure"/>
                <xsd:enumeration value="acouPressureSOSatLaplace"/>
                <xsd:enumeration value="acouPotential"/>
              </xsd:restriction>
            </xsd:simpleType>
          </xsd:attribute>

          <xsd:attribute name="timeStepAlpha" use="optional" default="0.0"
          type="xsd:double"/>

          <!-- updated Lagrangian formulation -->
          <xsd:attribute name="updatedLagrange" use="optional" default="no">
            <xsd:simpleType>
              <xsd:restriction base="xsd:token">
                <xsd:enumeration value="no"/>
                <xsd:enumeration value="yes"/>
              </xsd:restriction>
            </xsd:simpleType>
          </xsd:attribute>

          <!-- Type of acoustic pde -->
          <xsd:attribute name="subType" use="optional" default="standard">
            <xsd:simpleType>
              <xsd:restriction base="xsd:token">
                <xsd:enumeration value="standard"/>
                <xsd:enumeration value="combustionNoise"/>
              </xsd:restriction>
            </xsd:simpleType>
          </xsd:attribute>
```

```
        <!-- Type of acoustic flow formulation -->
        <xsd:attribute name="flowFormulation" use="optional" default="standard">
          <xsd:simpleType>
            <xsd:restriction base="xsd:token">
              <xsd:enumeration value="standard"/>
              <xsd:enumeration value="withDivergence"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:attribute>

        <!-- Type of function space (Lagrange / Legendre) with
        optional order -->
        <xsd:attribute name="type" use="optional" default="lagrange">
          <xsd:simpleType>
            <xsd:restriction base="xsd:token">
              <xsd:enumeration value="lagrange"/>
              <xsd:enumeration value="legendre"/>
              <xsd:enumeration value="spectral"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:attribute>
        <xsd:attribute name="order" use="optional" type="xsd:positiveInteger"
        default="1"/>

      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <!-- STOP -->

</xsd:schema>
```

Listing A.3: acoustic BEM scheme

A.3 Source code of testcase laplace_2d_transparent_standalone

A.3.1 mat.xml

```
<?xml version='1.0' encoding='utf-8'?>
<cfsMaterialDataBase xmlns="http://www.cfs++.org/material">
  <material name="air">
    <acoustic>
      <density>
        <linear>
          <real> 1.205 </real>
        </linear>
      </density>
      <compressionModulus>
        <linear>
          <real> 1.41767e5 </real>
        </linear>
      </compressionModulus>
    </acoustic>
  </material>
</cfsMaterialDataBase>
```

Listing A.4: dummy material file

A.3.2 laplace_2d_transparent_standalone.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<cfsSimulation xmlns="http://www.cfs++.org/simulation" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance" xsi:schemaLocation="http://www.cfs++.org/simulation
http://cfs-doc.mdmt.tuwien.ac.at/xml/CFS-Simulation/CFS.xsd">

    <fileFormats>
        <input>
            <hdf5 fileName="laplace_2d_transparent_standalone.h5"/>
        </input>
        <output>
            <!-- <hdf5 compressionLevel="9"/> -->
            <hdf5/>
        </output>
        <materialData/>
        <!-- <materialData file="mat.xml" format="xml"/> -->
    </fileFormats>

    <domain geometryType="3d">
        <regionList>
            <region name="sector" material="air"/>
        </regionList>
    </domain>

    <!-- <domain geometryType="3d"/> -->

    <sequenceStep>

        <!-- one formal cycle
                and best of all, it works easiest when transient -->
        <analysis>
            <transient>
                <numSteps>0</numSteps>
                <deltaT>0</deltaT>
            </transient>
        </analysis>

        <pdeList>
            <acoustic_BEM>

                <regionList>
                    <region/>
                </regionList>

                <storeResults/>

            </acoustic_BEM>

            <!-- unfortunately a simple <acoustic_BEM/> call is not sufficient -->

        </pdeList>

    </sequenceStep>

</cfsSimulation>
```

Listing A.5: testcase's xml file

A.3.3 /laplace_2d_transparent_standalone_NiHu/CMakeLists.txt

```
cmake_minimum_required (VERSION 3.15.0 FATAL_ERROR)

# set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD 14)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)

project(Hello_NiHu)

## get folder's name and set names of .cc and executable accordingly
#
#  documentation: The .cc file to create NiHu's executabe has to have
#                 the same name as the NiHu folder!
#

# current source directoy
message(STATUS "current source directory: ${CMAKE_CURRENT_SOURCE_DIR}")

# current source directoy
message(STATUS "current binary directory: ${CMAKE_CURRENT_BINARY_DIR}")

# this will not work properly in case of external simulations
# change this appropriately if necessary
get_filename_component(Folder_Devel
    "${CMAKE_CURRENT_SOURCE_DIR}/../../../../../../../../" ABSOLUTE)
message(STATUS "Devel at: ${Folder_Devel}")

# Get the name of the current directory (excluding the path)
get_filename_component(folder_name ${CMAKE_CURRENT_SOURCE_DIR} NAME)

# Set the source file
set(source_file "${folder_name}.cc")

# Set the executable name
set(executable_name "${folder_name}_output")
```

```
# Create the executable
add_executable(${executable_name} ${source_file})

# enable concepts for the target
target_compile_features(${executable_name} PRIVATE cxx_std_14)
target_compile_options(${executable_name} PRIVATE -fconcepts)

set_target_properties(${executable_name} PROPERTIES LINKER_LANGUAGE CXX)

# find /Devel as base directory
set(NIHU_BASE_PATH ${Folder_Devel}/CFS_BIN/build_opt/cfsdeps/nihu/src/NiHu/src/)
include_directories(${NIHU_BASE_PATH})

set(NIHU_EIGEN_PATH ${Folder_Devel}/CFS_BIN/build_opt/cfsdeps/eigen/install/include/)
include_directories(${NIHU_EIGEN_PATH})

set(NIHU_EXTERNAL_LIBRARY_PATH ${Folder_Devel}/CFS_BIN/build_opt/cfsdeps/nihu/src/
    NiHu-build/lib/nihu.a)
set(NIHU_FMM_EXTERNAL_LIBRARY_PATH ${Folder_Devel}/CFS_BIN/build_opt/cfsdeps/nihu/src/
    NiHu-build/lib/nihu_fmm.a)

# create a new library target for the external library
add_library(nihu STATIC IMPORTED)
add_library(nihu_fmm STATIC IMPORTED)
# add_library(weighted_residual STATIC IMPORTED)

# set the location of the external library file for the target
set_target_properties(nihu PROPERTIES IMPORTED_LOCATION ${NIHU_EXTERNAL_LIBRARY_PATH})
set_target_properties(nihu_fmm PROPERTIES IMPORTED_LOCATION
    ${NIHU_FMM_EXTERNAL_LIBRARY_PATH})

# link executable or library target with the external library
target_link_libraries(${executable_name} nihu)
target_link_libraries(${executable_name} nihu_fmm)
```

Listing A.6: testcase's CMakeLists

A.3.4 /laplace_2d_transparent_standalone_NiHu/laplace_2d_transparent_standalone_NiHu.cc

```
#include <boost/math/constants/constants.hpp>

 #include "core/weighted_residual.hpp"
 #include "library/laplace_2d.hpp"
 #include "../library/lib_element.hpp"

 #include <iostream>

 typedef Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic> dMatrix;
 typedef Eigen::Matrix<unsigned, Eigen::Dynamic, Eigen::Dynamic> uMatrix;

 static NiHu::mesh<tmp::vector<NiHu::line_1_elem> > create_mesh(double r, int N)
 {
     using namespace boost::math::double_constants;

     dMatrix nodes(N,2);
     uMatrix elements(N, 3);
     for (int i = 0; i < N; ++i)
     {
         double phi = i*two_pi/N;
         nodes(i,0) = r * cos(phi);
         nodes(i,1) = r * sin(phi);
         elements(i,0) = NiHu::line_1_elem::id;
         elements(i,1) = i % N;
         elements(i,2) = (i+1) % N;
     }
     return NiHu::create_mesh(nodes, elements, NiHu::line_1_tag());
 }

 template <class TestSpace, class TrialSpace>
 static void tester(TestSpace const &test_space, TrialSpace const &trial_space)
 {
     using namespace boost::math::double_constants;

     // instantiate integral operators
     auto I_op = NiHu::identity_integral_operator();
     auto G_op = NiHu::create_integral_operator(NiHu::laplace_2d_SLP_kernel());
     auto H_op = NiHu::create_integral_operator(NiHu::laplace_2d_DLP_kernel());
     auto Ht_op = NiHu::create_integral_operator(NiHu::laplace_2d_DLPt_kernel());
     auto D_op = NiHu::create_integral_operator(NiHu::laplace_2d_HSP_kernel());

     size_t N = test_space.get_num_dofs();
```

```
    // excitation and response
    dMatrix q0(N, 1), p0(N,1);
    p0.setZero();
    q0.setZero();

    dMatrix x0(2, 2);
    x0 <<
        .2, .3,
        .3, .2;
    dMatrix A(1, 2);
    A << 1.0, -1.0;

    auto const &mesh = trial_space.get_mesh();
    for (unsigned k = 0; k < N; ++k)
    {
        auto const &elem = mesh.template get_elem<NiHu::line_1_elem>(k);
        auto y = elem.get_center();
        auto ny = elem.get_normal().normalized();
        for (int s = 0; s < x0.cols(); ++s)
        {
            auto rvec = y - x0.col(s);
            double r = rvec.norm();
            double rdn = rvec.dot(ny) / r;
            p0(k) += A(s) * -std::log(r) / two_pi;
            q0(k) += A(s) * -(1.0/r) * rdn / two_pi;
        }
    }

    std::cout << "Matrix generation" << std::endl;

    dMatrix G(N, N), H(N, N), Ht(N, N), D(N, N), I(N,N);
    G.setZero();
    H.setZero();
    Ht.setZero();
    D.setZero();
    I.setZero();

    I << test_space * I_op[trial_space];
    G << test_space * G_op[trial_space];
    H << test_space * H_op[trial_space];
    Ht << test_space * Ht_op[trial_space];
    D << test_space * D_op[trial_space];

    // solve with direct Neumann
    std::cout << " Solution with direct Neumann:" << std::endl;
    dMatrix p_direct_neumann = (H - .5 * I).colPivHouseholderQr().solve(G * q0);
    double error_direct_neumann = (p_direct_neumann-p0).norm() / p0.norm();
    std::cout << "Direct Neumann Error: " << error_direct_neumann << std::endl;

    // solve with direct Dirichlet
    std::cout << " Solution with direct Dirichlet:" << std::endl;
    dMatrix q_direct_dirichlet = G.colPivHouseholderQr().solve((H - .5 * I) * p0);
    double error_direct_dirichlet = (q_direct_dirichlet-q0).norm() / q0.norm();
    std::cout << "Direct Dirichlet Error: " << error_direct_dirichlet << std::endl;
}

int main()
{
    int N = 40;
    double r = 1.2;
    auto mesh = create_mesh(r, N);
    auto const &trial_space = NiHu::constant_view(mesh);

    std::cout << "Collocation" << std::endl;
    tester(NiHu::dirac(trial_space), trial_space);

    std::cout << "Galerkin" << std::endl;
    tester(trial_space, trial_space);

    return 0;
}
```

Listing A.7: testcase: laplace_2d_transparent_standalone_NiHu.cc

Bibliography

[1] David Cole. “BUILDING EXTERNAL PROjECTS WITH CMAKE 2.8.” In: (2009).

[2] *gitlab.com/openCFS/cfs/-/wikis*. February 20, 2024.

[3] *git website: https://git-scm.com*. February 20, 2024.

[4] Manfred Kaltenbacher. *Numerical Simulation of Mechatronic Sensors and Actuators: Finite Elements for Computational Multiphysics*. Third. Berlin–Heidelberg: Springer, 2015.

[5] *NiHu Website: last.hit.bme.hu*. February 20, 2024.

[6] *openCFS Website: opencfs.org*. February 20, 2024.

[7] Péter Rucz Peter Fiala. “NiHu: A BEM-FMBEM Matlab toolbox.” In: (2013).

[8] Florian Kraxberger Stefan Schoder Andreas Wurzinger. *Computational Acoustics*. 2022.

[9] *www.kitware.com*. February 20, 2024.

[10] *www.paraview.org*. February 20, 2024.