

Python/C++ Interface for mbsolve Project

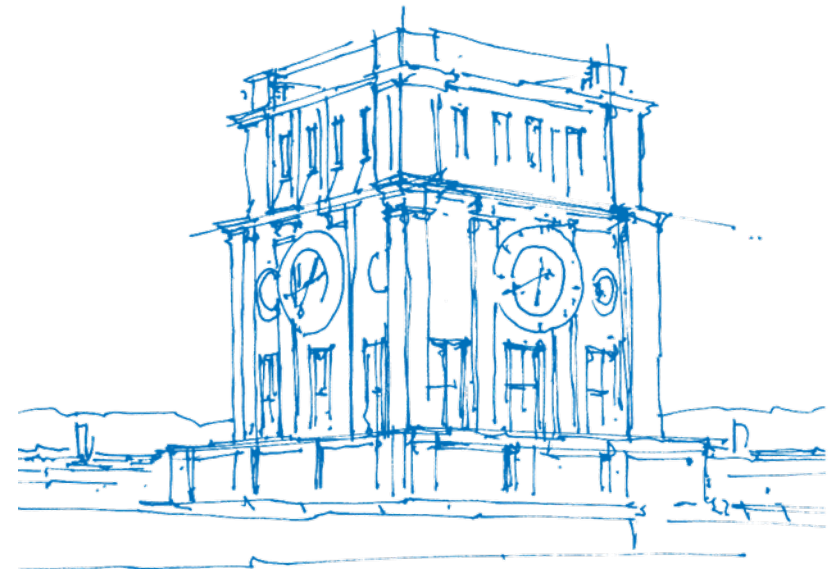
Mariem Kthiri

Technische Universität München

Department of Electrical and Computer Engineering

Professorship of Computational Photonics

München, 9 May, 2017



TUM Uhrenturm

Overview

In this presentation, I would like to

- Put forward the motivation and goal of the project
- Present the steps of the realization of this engineer's practice
- Display the delivered results

Introduction

- Quantum cascade laser is a type of semiconductor emitting mid-infrared portion of the electromagnetic spectrum allowing a multitude of applications.
- The mbsolve project provides the required simulation relying on
 - C++ programming
 - Python user interface

C++	Python
Advantages High performance and speed useful for intensive tasks Parallelization techniques	Advantages Flexibility (fast edit-build-debug cycle) Interactivity (create, change, view objects at runtime)
Drawbacks Non-interactive Writing user-interfaces is complex	Drawbacks relatively slow Limitations with memory intensive tasks Limitations with database access

Table: Characteristics of C++ and Python.

Conception and realization – Python/C++ Interface

- **Python.Boost** huge dependance to GCC and extensive use of C++ template
- **ctypes** allow access to a few functions within a DLL, but not optimal fo bigger libraries
- **SWIG**
 - language neutral
 - tagets many languages (Python, Tcl, MATLAB, etc)
 - simple and completely automated

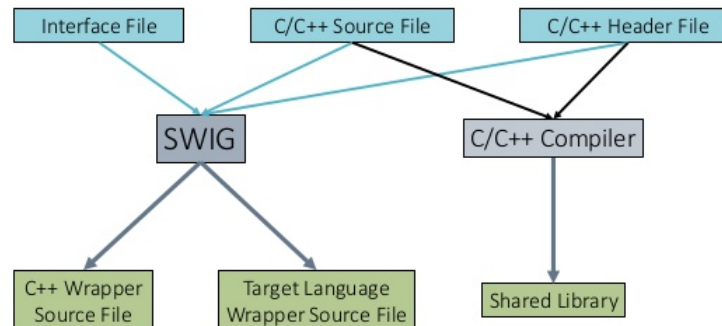


Figure: Functionality of SWIG

Conception and realization – Serializing input parameters

- Use dxml, a simple object-XML mapper for Python
- Create subclasses of the class dxml.Model
- Parse the XML file
- Store the input parameters as instances of the corresponding subclass

Conception and realization – Storing results

Options for file format to store the results

- **XML**
 - concise and unambiguous
 - but unsuitable for large amounts of data
- **VTK**
 - powerful tool to visualize scalars, vectors, complex numbers, etc
 - difficult setup
- **HDF5**
 - suitable for high volume and complex data
 - store and manage data of different types

Conception and realization – Git

A Git project was created to

- register every step of the realization
- have access to the project on any computer
- facilitate data exchange and work coordination between me and my supervisor.

Conception and realization – Makefile/CMake

Objective Organize code compilation, build and manage projects automatically.

- Makefile
 - list of targets with corresponding dependencies and command line(s)
- CMake
 - cross platform build system
 - automatically generates Makefiles using the CMakeListst.txt file

Results – Python/C++ Interface

- C++ library
 - contains the required C++ classes (material, region, record, etc) and the C++ function to calculate the results
- CMakeLists file
 - detect the SWIG executable and find the necessary packages and libraries
 - generate build files such as makefiles, nmake files and Visual Studio projects which call SWIG and compile the generated C++ files into shared objects (.so for UNIX or .pyd for Windows)
- Test program available as script and notebook
 - allow setup of materials, device, scenario, etc
 - extract metadata from an XML file
 - call the C++ function that calculates the required results
 - display and store results and metadata in an HDF5 file

Results

To run the program

- clone the Git repository of the project
- create and access a build directory
- to build run

```
$ cmake .. $ make
```
- bring the test program as well as the settings XML file to the build directory
- execute through

```
$ python project.py
```

Results

Compatible with different versions of Python

- specify the wanted version in the command line while compiling with cmake as follows

```
$ cmake -DWITH_PYTHON3=ON/OFF ..
```

Executable on Windows

- build the extension module using a configuration file (conventionally called setup.py)
- use the source code files generated by SWIG and the original C++ source to create an extension module object
- compile it into a shared object file or DLL (.pyd on Windows) through

```
$ python setup.py build_ext -inplace
```

Results – Restrictions

- A few commands differ from a Python to another (e.g. `raw_input/input`) => 2 testprograms needed
- Building the module extension using `setup.py` on Windows does not allow structuring the project (all files must be in one directory)

Thank you for your attention!