PhaseFieldSDK walkthrough

+ Software development for our group members

Xiaoxing Cheng, MuPRO LLC

Content

1. MuPRO PhaseFieldSDK
   - history
   - old and new design comparison
   - closer look
2. Tools + Best practices
   - pain points
   - version control systems
   - build systems
   - code editor

MuPRO PhaseFieldSDK

Brief history

- Dr. Chen's library, before 2000?
- Dr. Yulan Li's, around 2002, a single fortran 77 file, several thousand lines of serial code
- Dr. Ben Winchester, around 2011, multiple fortran 90 files, add msolve still serial
- Dr. Jason Britson, around 2013, convert to library and main program structure, add mpi
- Dr. Tiannan Yang, Dr. Bo Wang, Dr. Jianjun Wang, and myself, around 2016, improve the library so that Magnetic, Effective Properties, and Ferroelectric program can share common subroutines from the library
- Around 2019, improve free format parameter input and update documentation, release v1 of MuPRO ferroelectric, magnetic, and effective properties modules
- MuPRO LLC, 2023, use fortran 2008, ditch fftw, a more modular library design

The name and our goal

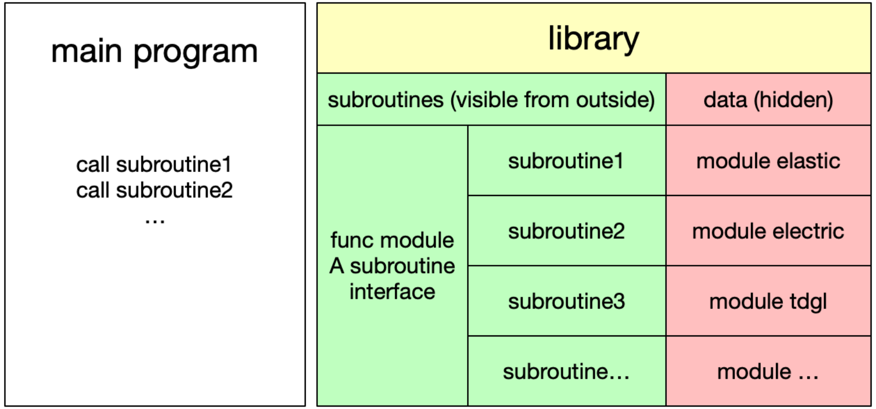
SDK means Software Development Kit

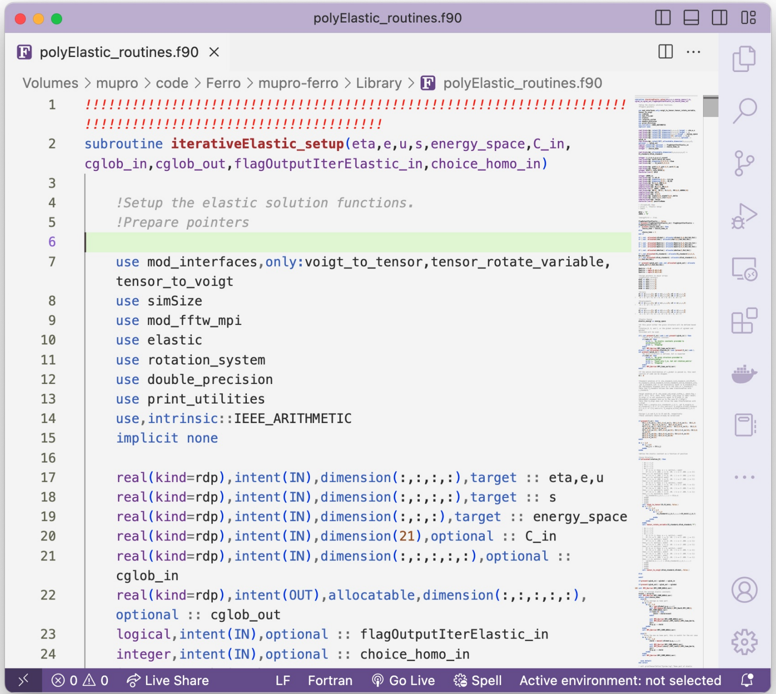
We aim to provide a set of tools to help users develop phase field softwares,

such tools include:

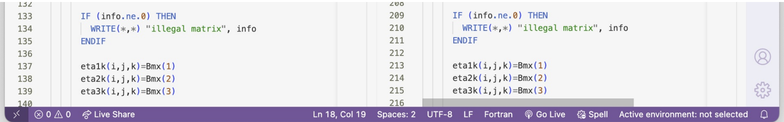
- a core library providing subroutines, derived typed, and modules
- main program starter projects that users can extend and customize
- useful tools for visualization, structure generation, data backup, etc.

Comparison between the old and new design

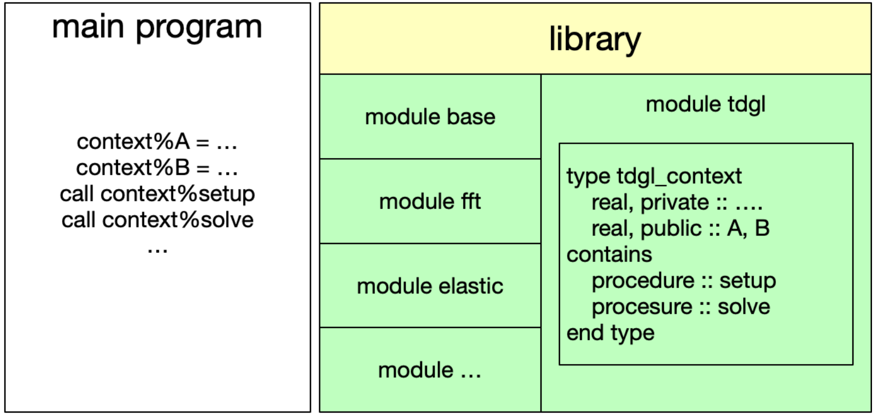




- isolate data in library from the main program
- data are connected through setup subroutines
- all parameter passed as subroutine arguments
- long func_module, also keep growing
- poor code reusability
- poor backward compatibility
- hard to develop



Comparison between the old and new design



- library → module
- library → module → type → data
- library → module → type → subroutine

- [illegible] memory, it
- [illegible] example, [illegible] object of the same
- [illegible] needs to get connected to
- [illegible] defined by [illegible] user's old main program should works fine with any new library

Closer look

```
├── cmake
├── CMakeLists.txt
├── extern
├── L0_Base
├── L0_FFT
├── L0_License
├── L0_Log
├── L1_IO
├── L1_PostProcess
├── L1_StructureGenerator
├── L1_Transform
├── L1_Utilities
├── L2_ACCH
├── L2_Defect
├── L2_Elastic
├── L2_Electric
├── L2_Magnetic
├── L2_Msolve
├── L2_TDGL
├── L3_Ferroelectric
├── L3_Metal
├── L3_Micromagnetic
├── not_used
```

```
xcheng@falcon: PhaseFieldSDK/dev/L2_TDGL$ tree
.
├── CMakeLists.txt
├── interface.f90
├── tdgl_lapack.f90
└── tests
    ├── CMakeLists.txt
    ├── lib.f90
    └── main.f90

1 directory, 6 files
```

Closer look

```
xcheng@falcon: PhaseFieldSDK/dev/L2_TDGL$ tree
.
├── CMakeLists.txt
├── interface.f90
├── tdgl_lapack.f90
└── tests
    ├── CMakeLists.txt
    ├── lib.f90
    └── main.f90

1 directory, 6 files
```

- setup subroutine is only for preparing a few variable for future use
- no repeating subroutines anymore

```
type(type_mupro_TDGLContext) :: polarContext
type(type_mupro_TDGLContext) :: octatiltContext

//! setup in main program
polarContext%G = polar_gradient_coefficient
polarContext%dt = dt_for_polar_tdgl
octatiltContext%G = octatilt_gradient_coefficient
octatiltContext%dt = dt_for_octatilt_tdgl
call polarContext%setup()
call octatiltContext%setup()

...

//! evolve order parameter after other calculations ta
polarContext%rhs => polar_tdgl_driving_force
polarContext%op => polarization_order_parameter
call polarContext%solve()

octatiltContext%rhs => octatilt_tdgl_driving_force
octatiltContext%op => octatilt_order_parameter
call octatiltContext%solve()

  subroutine TDGL_lapack_setup(context)
```

Continue …

- Github PhaseFieldSDK
- https://sdk.mupro.co
- Github Ferroelectric main program
- https://ferroelectric.mupro.co

Tools and Best practices

Tools + Best practices

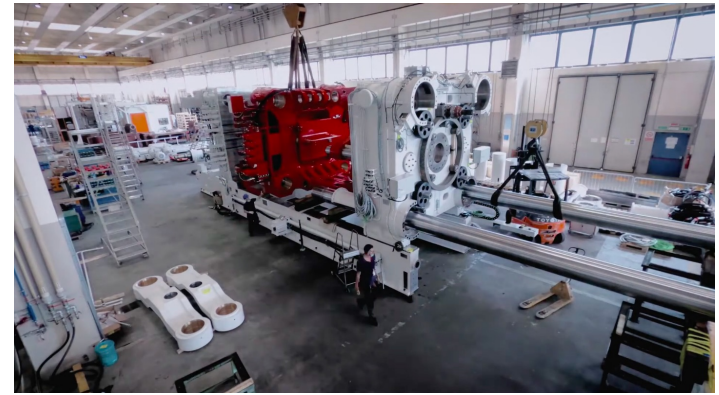
Pain points

- Too many versions for my code
- Cannot reproduce old results
  - code is lost
  - not sure which version to use
- I forget what I changed
- Code no longer build on different environment
- Build speed is slow
- Hard to debug with print *
- Unstable remote connection

Solutions

1. Version control system
2. Build system
3. Code editor

Proper tools are important



Tools - Version control system

Benefits include:

- tracking changes
- collaboration
- backup
- branching and merging

Available systems:

- **Git**
- Mercurial
- SVN

Git service providers:

- **Github**
- Gitlab

Git is a **must use tool** for any kind of code development. In the long run, if your career is in any ways related to code development, you have to know how to use git.

Even if you don't understand why you should use this or feels inconvenient in the beginning, you have to force yourself keep using it.

Git Best practice

- Use gitignore and only track necessary files
- Separate your simulation program from simulation results
- Keep commit small, and commit frequently
- Keep branching simple, and as short life as possible
- Tag your code when necessary
- setup github action to automate building and testing jobs if possible

Check out the repository for this presentation!

Use more!

Tools - Build system

Available systems:

- **CMake** 
- Meson

Benefits include:

- Automation (find dependencies)
- Cross-platform (generate vs, xcode, makefile)
- Faster speed (call ninja instead of make)

I **highly recommend** using CMake for building your code.

CMake might be daunting to learn at first sight, in the long run using CMake can save you a lot of time doing some repetitive jobs and it also has great integration with most code editors.

CMake best practices

- do not just copy and paste any CMake file you find online, many of the examples are for legacy cmake
- only learn Modern CMake, version higher than 3.12 (or 3.20 even better)
- think from the perspective of **targets** and the **dependencies between targets** when writing cmake files
- use CMakePresets to configure, build, and test
- always do out of source build
- gitignore any cmake generate files

Modern CMake!

Tools - code editor/IDE

- Vim
- Emacs
- Notepad++
- Visual Studio
- **Visual Studio Code**
- Xcode
- ...

Benefits of vscode includes:

- auto sync of any settings across all platform
- remote connection through ssh, auto reconnection and caching to ensure no lag
- huge amount of plugins available, such as cmake, intel oneapi, git, etc.
- connect to gdb and provide simple bu easy to use debugging interface
- can remember the status before close

VScode best practices

- setup passwordless connection to the server
- install **Modern Fortran**, **C/C++ Extension Pack**, **Extension Pack for Intel® oneAPI Toolkits**, **Remote - SSH**, **Remote Explorer** plugins
- use a fortran formatter (instruction in Modern Fortran plugin) to format your code
- use debugger instead of print *