Developing Octopus: an Introduction

Micael Oliveira

Octopus Course 2021, MPSD Hamburg

Scientific Software Development

Unique challenges:

- Translating science into code
- Need to understand the science
- Scientist are often not trained in software engineering
- Software performance is often important
- Many codes need to be enabled for high-performance computing:
  - Parallelism (MPI, OpenMP, etc)
  - GPU's
  - Complex hardware
  - Unusual architectures

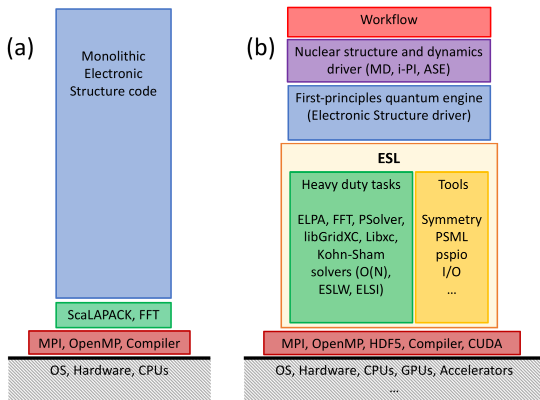
Scientific Software Development

- After a while, cost of maintenance becomes larger than cost of adding new features
- Software engineering good practices are essential!

Some best practices

- Code is the enemy: it can have bugs and it needs maintenance
- Do not reinvent the wheel: reuse code
- Write code that is easy to read and that is mostly self-documented
- Comments about why the code does something are very important
- Test your code
- “Premature optimization is the root of all evil”

Electronic structure "monolithic" and modular coding paradigms



M. J. T. Oliveira, N. Papior, Y. Pouillon, V. Blum, E. Artacho *et al*, J. Chem. Phys. **153**, 024117 (2020)

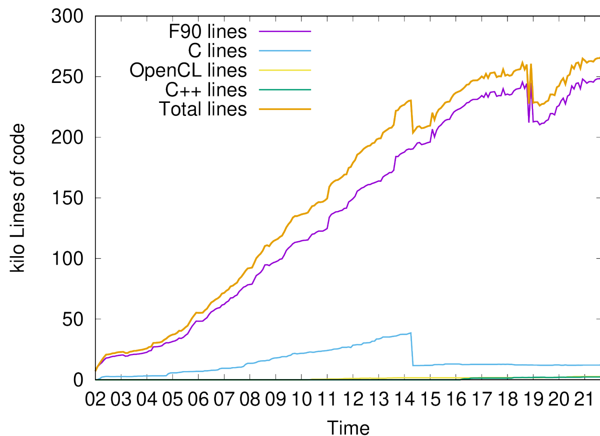
Octopus: Dissecting the Animal

- DFT and TDDFT code
- Some other theories implemented (Hartree-Fock, RDMFT, etc)
- Main focus on excited-state properties
- Real-space representation
- Norm-conserving pseudopotentials

Octopus: Dissecting the Animal

- Project formaly started in 2001
- Free-software (GPL)
- Writen mainly in Fortran 2003
- Fortran sources are preprocessed with cpp
- Some C, C++, perl and Bison (use the right tool for the job!)
- CUDA/OpenCL for GPU support
- Currently over 250,000 lines of code

Octopus: Dissecting the Animal



Octopus: a code for developers

- Not the fastest code around for most problems, but still quite fast
- Real space grid:
  - Good compromise between plane-waves and localized basis-sets
  - Can be as accurate as any other basis
  - Can easily describe excited states
  - Simple and intuitive
- Lots of “exotic” features (e.g., model systems, arbitrary dimensions, etc)
- A framework to implement, develop and test new ideas

octopus-code.org

- Wiki based website
- A new website is under construction
  `octopus-code.org/new-site/develop`
- Ressources for users:
  - Code download
  - Compilation instructions (partially outdated)
  - Manual (outdated)
  - Tutorials
  - Input variable reference
  - ...
- Dedicated section for developers

octopus-code.org/wiki/Developers

- "Starting to develop" guide (must read!)
- Workflow guide (must read!)
- Coding standards
- Input variable reference (development version)
- Some code documentation (partially outdated)
- ...

Git and GitLab

- Octopus uses **git** as version control system
- GitLab provides several important things:
  - Hosts main repository
  - Merge requests
  - Issues

Regression test suite and the Buildbot

- Octopus includes a large collection of regression tests
- Test suite covers $\sim 65\%$ of the code
- Continuous integration (CI) using Buildbot
- Buildbot is interfaced with GitLab

Build system

- Compilation and configuration is based on `autotools`
- Configure script is generated from `configure.ac`
- Makefiles are generated from `Makefile.am` files in each directory
- To generate the configure scripts run `autoreconf -i`
- `VPATH` builds are supported and suggested.

External libraries

- We do not like to reinvent the wheel
- We like to share code
- Octopus uses many external libraries, either optional or mandatory:
    - BLAS/LAPACK
    - FFTW
    - MPI
    - GSL
    - Libxc
    - Libvdwxc
    - PSolver
    - ELPA
    - ...

Coding style

- Set of rules and guidelines for writing code
- Deals with indentation, white spaces, naming conventions, etc
- Makes the code easier to read and understand
- Ideally the code should read like plain English

Bad

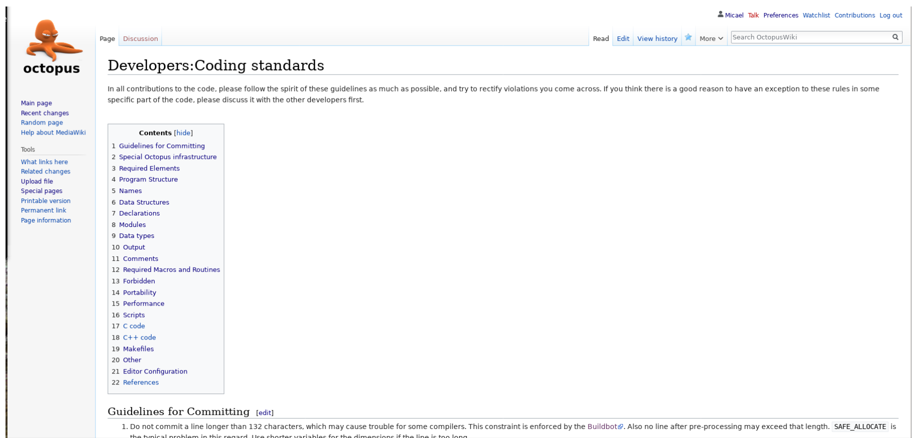
```
if (space%periodic_dim > 0) then
  ...
end if
```

Good

```
if (space%is_periodic()) then
  ...
end if
```

- Helps avoiding some errors

Octopus coding standards



https://octopus-code.org/wiki/Developers:Coding_standards

Octopus coding standards

Some examples:

- Two space indentation
- No single letter variable names
- Module names end with `_oct_m`, derived types with `_t`
- All functions should go inside modules.
- All modules must have `private` and `implicit none` statements
- Intents for subroutine arguments are mandatory
- ...

Preprocessor

- Changes the source before compilation
- We use the C preprocessor:
  - Standard
  - Widely available
  - Requires some tricks to work with Fortran code
  - Imposes (few) limitations on Fortran code
- Several macros generated when running `configure` script
- Conditional compilation:

```
#ifdef HAVE_MPI
   ...
#else
   ...
#endif
```

- Templating to generate same subroutine with different data types (float/complex/integers, scalar/array, etc)

Preprocessor: some useful Octopus specific macros

- SAFE_ALLOCATE()
  - Calls `allocate`
  - Returns error on failure
  - Counts allocated memory for profiling
- PUSH_SUB() / POP_SUB()
  - Generates a call stack used for debugging
- MAX_DIM
  - Maximum dimension the code can run
  - Deprecated
- FLOAT, CMPLX
  - Allow to change the real and complex types at compile time
  - Was introduced to allow compilation in single precision
  - Not really useful anymore; will likely be removed

Preprocessor: "templating"

- _inc.F90 files contain code that is independent of data type
- Files are included with the preprocessor in the following way:

```
#include "undef.F90"
#include "real.F90"
#include "my_function_inc.F90"

#include "undef.F90"
#include "complex.F90"
#include "my_function_inc.F90"
...
```

- Several macros are available to use in the _inc.F90 files

Preprocessor: "templating"

- Function definition:

```
function X(my_function)(arg1, arg2) result(res))
  R_TYPE, intent(in) :: arg1
  R_TYPE, intent(in) :: arg2
  R_TYPE, intent(out) :: res
...
end function X(my_function)
```

- Function call:

```
FLOAT :: da1, da2, dres
CMPLX  :: za1, za2, zres

dres = dmy_function(da1, da2)
zres = zmy_function(za1, za2)
```

- `X(...)`: prepends "type-prefix" (e.g., `d` or `z`) to subroutine name
- `R_TYPE`: templated type in function definition
- Other data types related macros available: `R_TOTYPE()`, `R_TOPREC()`, `R_CONJ()`, etc

Preprocessor: “templating”

real.F90

```
...
#define R_TYPE         FLOAT
#define R_BASE         FLOAT
#define R_DOUBLE       real(8)
#define R_MPITYPE      MPI_FLOAT
#define R_TYPE_VAL     TYPE_FLOAT
#define R_TYPE_CL      'RTYPE_DOUBLE'
#define R_TYPE_IOBINARY TYPE_DOUBLE
#define R_TOTYPE(x)   real(x, REAL_PRECISION)
#define R_TOPREC(x)   real(x, REAL_PRECISION)

#define R_CONJ(x)      (x)
#define R_REAL(x)      (x)
#define R_AIMAG(x)     (M_ZERO)

#define X(x)           d ## x
...
```

Preprocessor: “templating”

complex.F90

```
...
#define R_TYPE          CMPLX
#define R_BASE          FLOAT
#define R_DOUBLE        complex(8)
#define R_MPITYPE       MPI_CMPLX
#define R_TYPE_VAL      TYPE_CMPLX
#define R_TYPE_CL       'RTYPE_COMPLEX'
#define R_TYPE_IOBINARY TYPE_DOUBLE_COMPLEX
#define R_TOTYPE(x)     cmplx(x, M_ZERO, REAL_PRECISION)
#define R_TOPREC(x)     cmplx(real(x), aimag(x), REAL_PRECISION)

#define R_CONJ(x)       conjg(x)
#define R_REAL(x)       real(x)
#define R_AIMAG(x)      aimag(x)
...
#define X(x)            z ## x
...
```

Input file variables

- Octopus uses a parser written in Bison
- Input file is fully parsed at the beginning of the calculation:

```
ierr = parse_init('exec/parser.log', mpi_world%rank)
```

- `exec/parser.log` contains all the variables **accessed** during a calculation
- Input variables can be accessed anywhere in the code
- Avoid reading each variable more than once

Input file variables

- All parser interfaces are defined in the parser_oct_m module
- Scalar variables are accessed with the parse_variable function:

```
call parse_variable(global_namespace, 'CalculationMode', OPTION__CALCULATIONMODE__GS,
  inp_calc_mode)
```

- Reading blocks requires to use a block_t data type
- Blocks must be “opened” and “closed”:

```
type(block_t) :: blk
...
if (parse_block(namespace, 'Lsize', blk) == 0) then
  ! Lsize is specified as a block
  if (parse_block_cols(blk, 0) < space%dim) then
    call messages_input_error(namespace, 'Lsize')
  end if

  do idir = 1, space%dim
    call parse_block_float(blk, 0, idir - 1, sb%lsize(idir), units_inp%length)
    ...
  end do
  call parse_block_end(blk)
  ...
end if
```

Input variables documentation

- Variables are documented in the source code, just before where they are accessed
- Documentation is parsed by a script that generates HTML and plain text output
- Example:

```
!%Variable CalculationMode
!%Type integer
!%Default gs
!%Section Calculation Modes
!%Description
!% Decides what kind of calculation is to be performed.
!%Option gs 01
!% Calculation of the ground state.
!%Option unocc 02
!% Calculation of unoccupied/virtual KS states. Can also be used for a non-self-consist
!% calculation of states at arbitrary k-points, if <tt>density.obf</tt> from <tt>gs</tt
!% is provided in the <tt>restart/gs</tt> directory.
!% ...
!%End
```

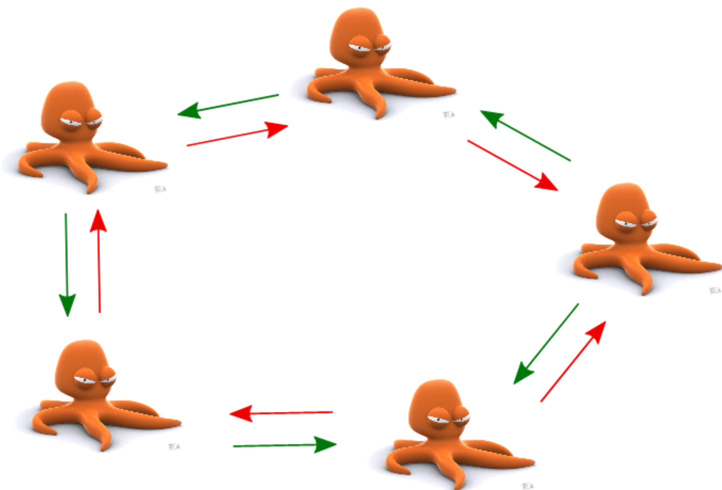
- Options defined in the documentation can be used in the input file

A look at the future: the multi-system framework

- After 20 years of development, the current code structure is starting to show its limits
- New developments are becoming more difficult
- Fortran 2003 introduces lots of new OOP features
- Several "multi-system" features were very hard to implement and maintain:
  - Subsystem DFT
  - Maxwell solver
  - Electronic transport
  - ...

In 2019 it was decided to introduce a new framework and rewrite large portions of Octopus.

What problem are we trying to solve?



What problem are we trying to solve?

- We want to solve a system of **coupled** differential equations
- How to handle arbitrary numbers of equations?
- How to add/remove equations "on-the-fly'?
- How to activate/deactivate couplings "on-the-fly"?

How to code this?

The way NOT to do it:

```
if (system_A%is_electrons) then
...
else if (system_A%is_ions) then
...
end if

if (system_A%has_interaction_X_with_system_B) then
...
end if
if (system_B%has_interaction_X_with_system_A) then
...
end if
if ((system_A%has_interaction_Y_with_system_B) then
...
end if
```

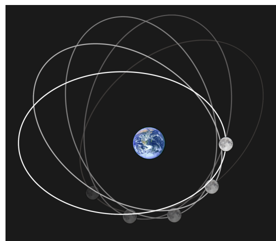
Multi-system framework: Key features

- New framework to handle calculations of coupled systems
- Allows to define many physical systems simultaneously (electrons, ions, lasers, Maxwell, DFTB+, PCM, etc)
- Systems are coupled through interactions (eElectron-ion, Lorentz force, dipole coupling, etc)
- Calculations modes are now "algorithms": a set of state machine atomic operations
- The code automatically handles all the interactions/systems
- New parallelization level: systems
- Current efforts focused on porting SCF and time propagation to new framework

Multi-System Framework: Design

- Focus on extendability and maintainability
- Adding new systems, interactions and algorithms should be as simple as possible
- Flexible algorithms:
  - Time-propagation using different propagators and time-steps for each system
  - Nested SCF loops
- Framework is independent of existing systems and interactions
- Systems do not know about each other directly, instead they know interactions
- Heavy use of object-oriented programming

Test environment: celestial dynamics



- System of Sun, Earth, and Moon as point particles interacting with gravity
- Numerical integration of orbits with different algorithms
- Fast turnover for code development

Test environment: celestial dynamics

inp

```
CalculationMode = td
ExperimentalFeatures = yes

%Systems
 "Sun"   | classical_particle
 "Earth" | classical_particle
 "Moon"  | classical_particle
%

%Interactions
 gravity | all_partners
%
InteractionTiming = timing_retarded

#Initial conditions are taken from https://ssd.jpl.nasa.gov/horizons.cgi#top.
# initial condition at time:
# 2458938.500000000 = A.D. 2020-Mar-30 00:00:00.0000 TDB

Earth.ParticleMass = 5.97237e24
%Earth.ParticleInitialPosition
 -147364661998.16476 | -24608859261.610123 | 1665165.2801353487
%
%Earth.ParticleInitialVelocity
 4431.136612956525 | -29497.611635546345 | 0.343475566161544
%
```

Test environment: celestial dynamics

inp (cont.)

```
Moon.ParticleMass = 7.342e22
%Moon.ParticleInitialPosition
 -147236396732.81906 | -24234200672.857853 | -11062799.286082389
%
%Moon.ParticleInitialVelocity
 3484.6397238565924 | -29221.007409082802 | 82.53526338876684
%

Sun.ParticleMass = 1.98855e30
%Sun.ParticleInitialPosition
 0.0 | 0.0 | 0.0
%
%Sun.ParticleInitialVelocity
 0.0 | 0.0 | 0.0
%

TDSystemPropagator = verlet

sampling = 24 # Time-steps per day
days = 3
seconds_per_day = 24*3600
Sun.TDTimeStep = seconds_per_day/sampling
Earth.TDTimeStep = seconds_per_day/sampling/2
Moon.TDTimeStep = seconds_per_day/sampling/4
TDPropagationTime = days*seconds_per_day
```

New multi-system syntax

Systems block

```
%Systems
 "Sun"   | classical_particle
 "Earth" | classical_particle
 "Moon"  | classical_particle
%
```

Nested systems

```
%Systems
 "Sun"   | classical_particle
 "Earth" | multisystem
%

%Earth.Systems
 "Terra" | classical_particle
 "Luna"  | classical_particle
%
```

New multi-system syntax

Namespaces

```
Sun.ParticleMass = 1.98855e30
Earth.Terra.ParticleMass = 5.97237e24
Luna.ParticleMass = 7.342e22
```

Interactions

```
%Interactions
 gravity       | all_partners
 coulomb_force | no_partners
%

%SystemA.Interactions
 gravity       | no_partners
 coulomb_force | all_partners
%

%SystemB.Interactions
 gravity       | only_partners | "SystemA"
 coulomb_force | all_except    | "SystemC"
%
```

Velocity Verlet

1. Update positions

$$\boldsymbol{x}(t+\Delta t) = \boldsymbol{x}(t) + \boldsymbol{x}(t)\Delta t + \frac{1}{2}\boldsymbol{a}(t)\Delta t^2$$

2. Update interactions with all partners (compute $\boldsymbol{F}(\boldsymbol{x}(t+\Delta t))$)
3. Compute acceleration $\boldsymbol{a}(t+\Delta t)$
4. Compute velocity

$$\boldsymbol{v}(t+\Delta t) = \boldsymbol{v}(t) + \frac{1}{2}(\boldsymbol{a}(t) + \boldsymbol{a}(t+\Delta t))\Delta t$$

Visualizing the multi-system time-stepping algorithm

https://octopus-code.org/new-site/develop/developers/code_documentation/propagators/custom_diagram/

Celestial orbits