Developing Octopus

Xavier Andrade*

European Theoretical Spectroscopy Facility
and
Departamento de Física de Materiales
Universidad del País Vasco, Spain

San Sebastian, January 2009

*xavier@tddft.org
Outline

1 Introduction
2 Development tools
3 The code
4 Main data-structures
5 Conclusions
Outline

1. Introduction
2. Development tools
3. The code
4. Main data-structures
5. Conclusions
Outline

1. Introduction
2. Development tools
3. The code
4. Main data-structures
5. Conclusions
Outline

1. Introduction
2. Development tools
3. The code
4. Main data-structures
5. Conclusions
**Octopus**

- DFT and TDDFT code.
- Mainly focused on excited states properties.
- Real space representation.
- Pseudo-potential approximation for ions.
Octopus

- DFT and TDDFT code.
- Mainly focused on excited states properties.
- Real space representation.
- Pseudo-potential approximation for ions.
- DFT and TDDFT code.
- Mainly focused on excited states properties.
- Real space representation.
- Pseudo-potential approximation for ions.
Octopus

- DFT and TDDFT code.
- Mainly focused on excited states properties.
- Real space representation.
- Pseudo-potential approximation for ions.
Free software - GPL license.

100k lines of Fortran 95 and C.

Around 10 developers.

4 main research groups:

- Universidad del País Vasco, San Sebastian, Spain
- Freie Universität Berlin, Germany
- Universidade de Coimbra, Portugal
- Université Lyon I, France
Octopus: development

- Free software - GPL license.
- 100k lines of Fortran 95 and C.
- Around 10 developers.
- 4 main research groups:
  - Universidad del País Vasco, San Sebastian, Spain
  - Freie Universität Berlin, Germany
  - Universidade de Coimbra, Portugal
  - Université Lyon I, France
Octopus: development

- Free software - GPL license.
- 100k lines of Fortran 95 and C.
- Around 10 developers.
- 4 main research groups:
  - Universidad del País Vasco, San Sebastian, Spain
  - Freie Universität Berlin, Germany
  - Universidade de Coimbra, Portugal
  - Université Lyon I, France
Octopus: development

- Free software - GPL license.
- 100k lines of Fortran 95 and C.
- Around 10 developers.
- 4 main research groups:
  - Universidad del País Vasco, San Sebastian, Spain
  - Freie Universität Berlin, Germany
  - Universidade de Coimbra, Portugal
  - Université Lyon I, France
Octopus: development

- Free software - GPL license.
- 100k lines of Fortran 95 and C.
- Around 10 developers.
- 4 main research groups:
  - Universidad del País Vasco, San Sebastian, Spain
  - Freie Universität Berlin, Germany
  - Universidade de Coimbra, Portugal
  - Université Lyon I, France
Octopus: development

- Free software - GPL license.
- 100k lines of Fortran 95 and C.
- Around 10 developers.
- 4 main research groups:
  - Universidad del País Vasco, San Sebastian, Spain
  - Freie Universität Berlin, Germany
  - Universidade de Coimbra, Portugal
  - Université Lyon I, France
Octopus: development

- Free software - GPL license.
- 100k lines of Fortran 95 and C.
- Around 10 developers.
- 4 main research groups:
  - Universidad del País Vasco, San Sebastian, Spain
  - Freie Universität Berlin, Germany
  - Universidade de Coimbra, Portugal
  - Université Lyon I, France
Octopus: development

- Free software - GPL license.
- 100k lines of Fortran 95 and C.
- Around 10 developers.
- 4 main research groups:
  - Universidad del País Vasco, San Sebastian, Spain
  - Freie Universität Berlin, Germany
  - Universidade de Coimbra, Portugal
  - Université Lyon I, France
About this guide

- Basic knowledge to start.
- Understand and read the code.
- Main reference is existing code.
About this guide

- Basic knowledge to start.
- Understand and read the code.
- Main reference is existing code.
About this guide

- Basic knowledge to start.
- Understand and read the code.
- Main reference is existing code.
Octopus webpage

- http://www.tddft.org/programs/octopus/
- Wiki: anyone can edit the page.
- Documentation:
  - Tutorial
  - Manual
  - Reference manual: documentation of the input variables (generated "automatically")
  - FAQ
  - Information for developers
http://www.tddft.org/programs/octopus/

Wiki: anyone can edit the page.

Documentation:
- Tutorial
- Manual
- Reference manual: documentation of the input variables (generated "automatically")
- FAQ
- Information for developers
Octopus webpage

- Wiki: anyone can edit the page.
- **Documentation:**
  - Tutorial
  - Manual
  - Reference manual: documentation of the input variables (generated “automatically”).
  - FAQ
  - Information for developers.
Octopus webpage

- http://www.tddft.org/programs/octopus/
- Wiki: anyone can edit the page.
- Documentation:
  - Tutorial
  - Manual
  - Reference manual: documentation of the input variables (generated “automatically”).
  - FAQ
  - Information for developers.
http://www.tddft.org/programs/octopus/

Wiki: anyone can edit the page.

Documentation:
- Tutorial
- Manual
- Reference manual: documentation of the input variables (generated “automatically”).
- FAQ
- Information for developers.
Octopus webpage

- http://www.tddft.org/programs/octopus/
- Wiki: anyone can edit the page.
- Documentation:
  - Tutorial
  - Manual
  - Reference manual: documentation of the input variables (generated “automatically”).
  - FAQ
  - Information for developers.
http://www.tddft.org/programs/octopus/

Wiki: anyone can edit the page.

Documentation:
- Tutorial
- Manual
- Reference manual: documentation of the input variables (generated “automatically”).
- FAQ
- Information for developers.
Octopus webpage

- http://www.tddft.org/programs/octopus/
- Wiki: anyone can edit the page.
- Documentation:
  - Tutorial
  - Manual
  - Reference manual: documentation of the input variables (generated “automatically”).
  - FAQ
  - Information for developers.
Communication

- **Mailing lists:**
  - **octopus-announce**: Used to announce new versions.
  - **octopus-devel**: Communication between developers and notification of SVN commits.
  - **octopus-users**: Communication with users.

- **IRC meetings**: #octopus channel on irc.freshmeat.net.
Mailing lists:
  - octopus-announce: Used to announce new versions.
  - octopus-devel: Communication between developers and notification of SVN commits.
  - octopus-users: Communication with users.

IRC meetings: #octopus channel on irc.freshmeat.net.
SVN repository

- Version control system used by Octopus.
  - Get the code:
    `svn co http://www.tddft.org/svn/octopus/trunk/ octopus`
  - Get current versions: `svn update`
  - Check your changes: `svn status` and `svn diff`.
  - Send changes to server: `svn commit` (you need an account).
SVN repository

- Version control system used by Octopus.
- Get the code:
  
  ```bash
  svn co http://www.tddft.org/svn/octopus/trunk/ octopus
  ```
- Get current versions: `svn update`
- Check your changes: `svn status` and `svn diff`.
- Send changes to server: `svn commit` (you need an account).
SVN repository

- Version control system used by Octopus.
- Get the code:
  
  `svn co http://www.tddft.org/svn/octopus/trunk/ octopus`

- Get current versions: `svn update`

- Check your changes: `svn status` and `svn diff`.

- Send changes to server: `svn commit` (you need an account).
SVN repository

- Version control system used by Octopus.
- Get the code:
  ```
  svn co http://www.tddft.org/svn/octopus/trunk/ octopus
  ```
- Get current versions: `svn update`
- Check your changes: `svn status and svn diff`.
- Send changes to server: `svn commit` (you need an account).
SVN repository

- Version control system used by Octopus.
- Get the code:
  `svn co http://www.tddft.org/svn/octopus/trunk/ octopus`
- Get current versions: `svn update`
- Check your changes: `svn status` and `svn diff`.
- Send changes to server: `svn commit` (you need an account).
Trac

- Code management system: http://www.tddft.org/trac/octopus/
- Code browser.
- SVN interface.
- Bug tracking system.
Trac

- Code management system: http://www.tddft.org/trac/octopus/
- Code browser.
- SVN interface.
- Bug tracking system.
Trac

- Code management system: http://www.tddft.org/trac/octopus/
- Code browser.
- SVN interface.
- Bug tracking system.
- Code management system: http://www.tddft.org/trac/octopus/
- Code browser.
- SVN interface.
- Bug tracking system.
Documentation for input variables

- Documentation is placed in the code, where the variable is read.
- This is parsed by a script that generates HTML and plain text output.
- This is an example:

```
%!Variable Example
%!Type integer
%!Default first_option
%!Section Execution::Examples
%!Description
%! Here goes a description of the variable. It can be several
%! lines long.
%!Option first_option 1
%! Here goes the description for the option.
%!Option second_option 6
%! Same for this variable.
%! Again it can take several lines.
%!End
```

- The options can be used in the input file.
Documentation for input variables

- Documentation is placed in the code, where the variable is read.
- This is parsed by a script that generates HTML and plain text output.

This is an example:

```
!%Variable Example
!%Type integer
!%Default first_option
!%Section Execution::Examples
!%Description
!% Here goes a description of the variable. It can be several lines long.
!%Option first_option 1
!% Here goes the description for the option.
!%Option second_option 6
!% Same for this variable.
!% Again it can take several lines.
!%End
```

- The options can be used in the input file.
Documentation for input variables

- Documentation is placed in the code, where the variable is read.
- This is parsed by a script that generates HTML and plain text output.

This is an example:

```
%!Variable Example
%!Type integer
%!Default first_option
%!Section Execution::Examples
%!Description
%!Here goes a description of the variable. It can be several
%!lines long.
%!Option first_option 1
%!Here goes the description for the option.
%!Option second_option 6
%!Same for this variable.
%!Again it can take several lines.
%!End
```

- The options can be used in the input file.
Documentation for input variables

- Documentation is placed in the code, where the variable is read.
- This is parsed by a script that generates HTML and plain text output.
- This is an example:

  `!%Variable Example
  !%Type integer
  !%Default first_option
  !%Section Execution::Examples
  !%Description
  !% Here goes a description of the variable. It can be several lines long.
  !%Option first_option 1
  !% Here goes the description for the option.
  !%Option second_option 6
  !% Same for this variable.
  !% Again it can take several lines.
  !%End`

- The options can be used in the input file.
Set of tests to verify Octopus results.

Several sub-directories for different system types.

A test is composed of:
- One or more input files .inp extension.
- A file with information to run the tests and reference results .test extension.

Tests are executed with the oct-run_regression_test.pl script.

Set of tests: oct-run_testsuite.

You can use make check and check-full.

Verify that your changes do not break tests.
Testsuite

- Set of tests to verify Octopus results.
- Several sub-directories for different system types.
- A test is composed of:
  - One or more input files .inp extension.
  - A file with information to run the tests and reference results .test extension.
- Test are executed with the oct-run_regression_test.pl script.
- Set of tests: oct-run_testsuite.
- You can use make check and check-full.
- Verify that your changes do not break tests.
Testsuite

- Set of tests to verify Octopus results.
- Several sub-directories for different system types.
- A test is composed of:
  - One or more input files .inp extension.
  - A file with information to run the tests and reference results .test extension.
- Test are executed with the oct-run_regression_test.pl script.
- Set of tests: oct-run_testsuite.
- You can use make check and check-full.
- Verify that your changes do not break tests.
Testsuite

- Set of tests to verify Octopus results.
- Several sub-directories for different system types.
- A test is composed of:
  - One or more input files `.inp` extension.
  - A file with information to run the tests and reference results `.test` extension.
- Tests are executed with the `oct-run_regression_test.pl` script.
- Set of tests: `oct-run_testsuite`.
- You can use `make check` and `check-full`.
- Verify that your changes do not break tests.
Set of tests to verify Octopus results.

Several sub-directories for different system types.

A test is composed of:

- One or more input files .inp extension.
- A file with information to run the tests and reference results .test extension.

Tests are executed with the `oct-run_regression_test.pl` script.

Set of tests: `oct-run_testsuite`.

You can use `make check` and `check-full`.

Verify that your changes do not break tests.
Testsuite

- Set of tests to verify Octopus results.
- Several sub-directories for different system types.
- A test is composed of:
  - One or more input files .inp extension.
  - A file with information to run the tests and reference results .test extension.
- Test are executed with the oct-run_regression_test.pl script.
- Set of tests: oct-run_testsuite.
- You can use make check and check-full.
- Verify that your changes do not break tests.
Set of tests to verify Octopus results.
Several sub-directories for different system types.
A test is composed of:
  One or more input files .inp extension.
  A file with information to run the tests and reference results .test extension.
Test are executed with the oct-run_regression_test.pl script.
Set of tests: oct-run_testsuite.
You can use make check and check-full.
Verify that your changes do not break tests.
Testsuite

- Set of tests to verify Octopus results.
- Several sub-directories for different system types.
- A test is composed of:
  - One or more input files \texttt{.inp} extension.
  - A file with information to run the tests and reference results \texttt{.test} extension.
- Test are executed with the \texttt{oct-run_regression_test.pl} script.
- Set of tests: \texttt{oct-run_testsuite}.

You can use \texttt{make check} and \texttt{check-full}.

Verify that your changes do not break tests.
Testsuite

- Set of tests to verify Octopus results.
- Several sub-directories for different system types.
- A test is composed of:
  - One or more input files .inp extension.
  - A file with information to run the tests and reference results .test extension.
- Test are executed with the oct-run_regression_test.pl script.
- Set of tests: oct-run_testsuite.
- You can use make check and check-full.
- Verify that your changes do not break tests.
Buildbot

- Continuous integration tool.
  - Compiles Octopus periodically in different systems.
  - Runs the testsuite.
  - Reports error to the mailing list.
  - (Supposedly) Integrated into trac.
Buildbot

- Continuous integration tool.
- Compiles Octopus periodically in different systems.
- Runs the testsuite.
- Reports error to the mailing list.
- (Supposedly) Integrated into trac.
Buildbot

- Continuous integration tool.
- Compiles Octopus periodically in different systems.
- Runs the testsuite.
- Reports error to the mailing list.
- (Supposedly) Integrated into trac.
Buildbot

- Continuous integration tool.
- Compiles Octopus periodically in different systems.
- Runs the testsuite.
- Reports error to the mailing list.
- (Supposedly) Integrated into trac.
Buildbot

- Continuous integration tool.
-Compiles Octopus periodically in different systems.
-Runs the test suite.
-Reports error to the mailing list.
-(Supposedly) Integrated into trac.
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.
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.*
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.
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.
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.
Mainly Fortran 95 with C Preprocessor.
Object oriented programming (as possible in F95).
Extensive use of derived data-types.
Modules and explicit interfaces.
C for special requirements.
Perl for scripts.
Mainly Fortran 95 with C Preprocessor.

Object oriented programming (as possible in F95).

- Extensive use of derived data-types.
- Modules and explicit interfaces.
- C for special requirements.
- Perl for scripts.
Mainly Fortran 95 with C Preprocessor.

Object oriented programming (as possible in F95).

Extensive use of derived data-types.

Modules and explicit interfaces.

C for special requirements.

Perl for scripts.
Mainly Fortran 95 with C Preprocessor.
Object oriented programming (as possible in F95).
Extensive use of derived data-types.
Modules and explicit interfaces.
C for special requirements.
Perl for scripts.
Mainly Fortran 95 with C Preprocessor.

Object oriented programming (as possible in F95).

Extensive use of derived data-types.

Modules and explicit interfaces.

C for special requirements.

Perl for scripts.
The code

- Mainly Fortran 95 with C Preprocessor.
- Object oriented programming (as possible in F95).
- Extensive use of derived data-types.
- Modules and explicit interfaces.
- C for special requirements.
- Perl for scripts.
Directory structure

- **src/**: The Octopus code.
- **libxc/**: The exchange and correlation functionals library.
- **liboct_parser/**: The parser library.
- **testsuite/**: Set of tests to verify that Octopus is working properly.
- **external_libs/**: Libraries distributed with Octopus.
- **etc.**
Directory structure

- *src/*: The Octopus code.
- *libxc/*: The exchange and correlation functionals library.
- *liboct_parser/*: The parser library.
- *testsuite/*: Set of tests to verify that Octopus is working properly.
- *external_libs/*: Libraries distributed with Octopus.
- etc.
Directory structure

- **src/**: The Octopus code.
- **libxc/**: The exchange and correlation functionals library.
- **liboct_parser/**: The parser library.
- **testsuite/**: Set of test to verify that Octopus is working properly.
- **external_libs/**: Libraries distributed with Octopus.
- **etc.**
Directory structure

- src/: The Octopus code.
- libxc/: The exchange and correlation functionals library.
- liboct_parser/: The parser library.
- testsuite/: Set of test to verify that Octopus is working properly.
- external_libs/: libraries distributed with Octopus.
- etc.
Directory structure

- `src/`: The Octopus code.
- `libxc/`: The exchange and correlation functionals library.
- `liboct_parser/`: The parser library.
- `testsuite/`: Set of test to verify that Octopus is working properly.
- `external_libs/`: Libraries distributed with Octopus.
- etc.
Directory structure

- `src/`: The Octopus code.
- `libxc/`: The exchange and correlation functionals library.
- `liboct_parser/`: The parser library.
- `testsuite/`: Set of test to verify that Octopus is working properly.
- `external_libs/`: Libraries distributed with Octopus.
- `etc.`
Sub-directories of `src/`

- `include/` Includes and preprocessor macros.
- `basic/` Basic and utilitary routines.
- `math/` Mathematical routines.
- `species/` The pseudo-potentials and other types of species.
- `ions/` Classical stuff, ions and the simulation box.
- `beak/` Optimized and platform dependent code.
- `grid/` The grid and operations on it.
- `poisson/` Poisson solvers.
- `states/` The wave-functions and operations over them.
- `xc/` The interface to libxc.
- `hamiltonian/` The code to apply the hamiltonian.
- `system/` Routines that depend on the hamiltonian.
- `scf/` Ground state DFT routines and eigensolvers.
- `td/` The time propagators.
- `opt_control/` Optimal control theory.
- `sternheimer/` Linear response code.
- `main/` The main routine and other run modes.
- `utils/` Octopus auxiliary utilities.
Coding guidelines

- No strict rules (for the moment).
- No single letter variable names.
- Module names end with _m derived types with _t
- Two space indentation.
- All functions should go inside modules.
- All modules must be declared private and implicit none.
- Always declare intent for arguments.
- Good code ≠ code with comments.
- Avoid global module variables.
- Fortran limitations:
  - Lines: 132 characters long
  - Variables: 31 characters long
Coding guidelines

- No strict rules (for the moment).
- **No single letter variable names.**
  - Module names end with _m derived types with _t
  - Two space indentation.
  - All functions should go inside modules.
  - All modules must be declared private and implicit none.
  - Always declare intent for arguments.
  - Good code ≠ code with comments.
  - Avoid global module variables.
- Fortran limitations:
  - Lines: 132 characters long
  - Variables: 31 characters long
Coding guidelines

- No strict rules (for the moment).
- No single letter variable names.
- Module names end with `_m` derived types with `_t`.
  - Two space indentation.
  - All functions should go inside modules.
  - All modules must be declared private and implicit none.
  - Always declare intent for arguments.
  - Good code ≠ code with comments.
  - Avoid global module variables.
- Fortran limitations:
  - Lines: 132 characters long
  - Variables: 31 characters long
Coding guidelines

- No strict rules (for the moment).
- No single letter variable names.
- Module names end with \_m derived types with \_t
- **Two space indentation.**
- All functions should go inside modules.
- All modules must be declared private and implicit none.
- Always declare intent for arguments.
- Good code ≠ code with comments.
- Avoid global module variables.
- Fortran limitations:
  - Lines: 132 characters long
  - Variables: 31 characters long
Coding guidelines

- No strict rules (for the moment).
- No single letter variable names.
- Module names end with _m derived types with _t.
- Two space indentation.
- All functions should go inside modules.
- All modules must be declared private and implicit none.
- Always declare intent for arguments.
- Good code ≠ code with comments.
- Avoid global module variables.
- Fortran limitations:
  - Lines: 132 characters long
  - Variables: 31 characters long
Coding guidelines

- No strict rules (for the moment).
- No single letter variable names.
- Module names end with _m derived types with _t
- Two space indentation.
- All functions should go inside modules.
- All modules must be declared private and implicit none.
- Always declare intent for arguments.
- Good code ≠ code with comments.
- Avoid global module variables.
- Fortran limitations:
  - Lines: 132 characters long
  - Variables: 31 characters long
Coding guidelines

- No strict rules (for the moment).
- No single letter variable names.
- Module names end with _m derived types with _t
- Two space indentation.
- All functions should go inside modules.
- All modules must be declared private and implicit none.
- Always declare intent for arguments.

- Good code ≠ code with comments.
- Avoid global module variables.
- Fortran limitations:
  - Lines: 132 characters long
  - Variables: 31 characters long
Coding guidelines

- No strict rules (for the moment).
- No single letter variable names.
- Module names end with _m derived types with _t
- Two space indentation.
- All functions should go inside modules.
- All modules must be declared private and implicit none.
- Always declare intent for arguments.
- **Good code ≠ code with comments.**
- Avoid global module variables.
- Fortran limitations:
  - Lines: 132 characters long
  - Variables: 31 characters long
Coding guidelines

- No strict rules (for the moment).
- No single letter variable names.
- Module names end with _m derived types with _t
- Two space indentation.
- All functions should go inside modules.
- All modules must be declared private and implicit none.
- Always declare intent for arguments.
- Good code ≠ code with comments.
- Avoid global module variables.

Fortran limitations:
- Lines: 132 characters long.
- Variables: 31 characters long.
Coding guidelines

- No strict rules (for the moment).
- No single letter variable names.
- Module names end with \_m derived types with \_t
- Two space indentation.
- All functions should go inside modules.
- All modules must be declared private and implicit none.
- Always declare intent for arguments.
- Good code \(\neq\) code with comments.
- Avoid global module variables.
- **Fortran limitations:**
  - Lines: 132 characters long.
  - Variables: 31 characters long.
Coding guidelines

- No strict rules (for the moment).
- No single letter variable names.
- Module names end with \_m derived types with \_t
- Two space indentation.
- All functions should go inside modules.
- All modules must be declared private and implicit none.
- Always declare intent for arguments.
- Good code $\neq$ code with comments.
- Avoid global module variables.
- Fortran limitations:
  - Lines: 132 characters long.
  - Variables: 31 characters long.
Coding guidelines

- No strict rules (for the moment).
- No single letter variable names.
- Module names end with _m derived types with _t.
- Two space indentation.
- All functions should go inside modules.
- All modules must be declared private and implicit none.
- Always declare intent for arguments.
- Good code ≠ code with comments.
- Avoid global module variables.
- Fortran limitations:
  - Lines: 132 characters long.
  - Variables: 31 characters long.
A typical Octopus object

- Module `algo_m` in `algo.F90`.
- Type `algo_t`.
- Constructor `algo_init` and `algo_end` functions.
- Function names start with `algo_`.
- The parameter of type `algo_t` goes first.
- Access functions for members (unless critical for performance).
A typical Octopus object

- **Module** `algo_m` in `algo.F90`.
- **Type** `algo_t`.
  - Constructor `algo_init` and `algo_end` functions.
  - Function names start with `algo_`.
  - The parameter of type `algo_t` goes first.
  - Access functions for members (unless critical for performance).

X. Andrade (EHU/UPV)  Developing Octopus  San Sebastian, January 2009  18 / 36
A typical Octopus object

- Module \texttt{algo.m} in \texttt{algo.F90}.
- Type \texttt{algo_t}.
- Constructor \texttt{algo_init} and \texttt{algo_end} functions.
  - Function names start with \texttt{algo_}.
  - The parameter of type \texttt{algo_t} goes first.
  - Access functions for members (unless critical for performance).
A typical Octopus object

- Module `algo_m` in `algo.F90`.
- Type `algo_t`.
- Constructor `algo_init` and `algo_end` functions.
- Function names start with `algo_`.
- The parameter of type `algo_t` goes first.
- Access functions for members (unless critical for performance).
A typical Octopus object

- Module `algo.m` in `algo.F90`.
- Type `algo_t`.
- Constructor `algo_init` and `algo_end` functions.
- Function names start with `algo_`.
- The parameter of type `algo_t` goes first.
- Access functions for members (unless critical for performance).
Module `algo_m` in `algo.F90`.

Type `algo_t`.

Constructor `algo_init` and `algo_end` functions.

Function names start with `algo_`.

The parameter of type `algo_t` goes first.

Access functions for members (unless critical for performance).
Octopus relies on preprocessors macros to overcome Fortran limitations.

- Macros for numerical data types: FLOAT and CMPLX.
- Numerical literals: CNST(1.0).
- Maximum dimension the code can run: MAX_DIM.
- Configure macros.
- Parallelization enabled: HAVE_MPI.
Octopus relies on preprocessors macros to overcome Fortran limitations.

- Macros for numerical data types: FLOAT and CMPLX.
- Numerical literals: CNST(1.0).
- Maximum dimension the code can run: MAX_DIM.
- Configure macros.
- Parallelization enabled: HAVE_MPI.
Preprocessor macros

- Octopus relies on preprocessors macros to overcome Fortran limitations.
- Macros for numerical data types: FLOAT and CMPLX.
- Numerical literals: CNST(1.0).
- Maximum dimension the code can run: MAX_DIM.
- Configure macros.
- Parallelization enabled: HAVE_MPI.
Octopus relies on preprocessors macros to overcome Fortran limitations.

- Macros for numerical data types: FLOAT and CMPLX.
- Numerical literals: CNST(1.0).
- Maximum dimension the code can run: MAX_DIM.

- Configure macros.
- Parallelization enabled: HAVE_MPI.
Octopus relies on preprocessors macros to overcome Fortran limitations.

Macros for numerical data types: FLOAT and CMPLX.

Numerical literals: CNST(1.0).

Maximum dimension the code can run: MAX_DIM.

Configure macros.

Parallelization enabled: HAVE_MPI.
Octopus relies on preprocessors macros to overcome Fortran limitations.

- Macros for numerical data types: `FLOAT` and `CMPLX`.
- Numerical literals: `CNST(1.0)`.
- Maximum dimension the code can run: `MAX_DIM`.
- Configure macros.
- Parallelization enabled: `HAVE_MPI`. 
Generic code than can be compiled for different data-types.

Code with preprocessor macros (_inc.F90 extension).

Use the macro R_TYPE as the data-type.

The macro X() appends d or z.

Include the file several times:

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

```
#include "undef.F90"
#include "complex.F90"
#include "mesh_function_inc.F90"
```
Poor man’s templates

- Generic code than can be compiled for different data-types.
- **Code with preprocessor macros (`_inc.F90` extension).**
  - Use the macro `R_TYPE` as the data-type.
  - The macro `X()` appends `d` or `z`.
  - Include the file several times:
    ```
    #include "undef.F90"
    #include "real.F90"
    #include "mesh_function_inc.F90"
    #include "undef.F90"
    #include "complex.F90"
    #include "mesh_function_inc.F90"
    ```
Poor man’s templates

- Generic code that can be compiled for different data-types.
- Code with preprocessor macros (.inc.F90 extension).
- Use the macro R_TYPE as the data-type.
- The macro X() appends d or z.
- Include the file several times:
  
  ```
  #include "undef.F90"
  #include "real.F90"
  #include "mesh_function_inc.F90"

  #include "undef.F90"
  #include "complex.F90"
  #include "mesh_function_inc.F90"
  ```
Poor man’s templates

- Generic code than can be compiled for different data-types.
- Code with preprocessor macros (_inc.F90 extension).
- Use the macro R_TYPE as the data-type.
- The macro X() appends d or z.

Include the file several times:

```c
#include "undef.F90"
#include "real.F90"
#include "mesh_function_inc.F90"
#include "undef.F90"
#include "complex.F90"
#include "mesh_function_inc.F90"
```
Poor man's templates

- Generic code than can be compiled for different data-types.
- Code with preprocessor macros (\_inc.F90 extension).
- Use the macro R\_TYPE as the data-type.
- The macro X() appends d or z.

- Include the file several times:
  
  ```
  #include "undef.F90"
  #include "real.F90"
  #include "mesh\_function\_inc.F90"
  ```

  ```
  #include "undef.F90"
  #include "complex.F90"
  #include "mesh\_function\_inc.F90"
  ```
**Allocation macro**

- **ALLOCATE(a, size)**
  - Wrapper around Fortran `allocate`.
  - `a`: the variable to be allocated.
  - `size`: the number of elements allocated.
  - Returns an error if allocation failed.
Allocation macro

- **ALLOCATE**(*a*, *size*)
- **Wrapper around Fortran allocate.**
  - *a*: the variable to be allocated.
  - *size*: the number of elements allocated.
  - Returns an error if allocation failed.
Allocation macro

- ALLOCATE(a, size)
- Wrapper around Fortran allocate.
- a: the variable to be allocated.
- size: the number of elements allocated.
- Returns an error if allocation failed.
**Allocation macro**

- **ALLOCATE(a, size)**
- Wrapper around Fortran `allocate`.
- **a**: the variable to be allocated.
- **size**: the number of elements allocated.
- Returns an error if allocation failed.
ALLOCATE(a, size)

Wrapper around Fortran allocate.

- **a**: the variable to be allocated.
- **size**: the number of elements allocated.

Returns an error if allocation failed.
Octopus has its own debugging routines.

- With `DebugLevel` greater than 3 it will print when the code enters and exit a function.
- With `DebugLevel` greater than 100 info will go to files under `debug/` (one per processor, useful for parallel runs).
- You have to mark the entry point:
  ```
call push_sub('file.function')
  ```
- and the exit point:
  ```
call pop_sub()
  ```
Octopus has its own debugging routines. With `DebugLevel` greater than 3 it will print when the code enters and exit a function.

With `DebugLevel` greater than 100 info will go to files under `debug/` (one per processor, useful for parallel runs).

You have to mark the entry point:
```
call push_sub('file.function')
```

and the exit point:
```
call pop_sub()
```
Octopus has its own debugging routines.

With DebugLevel greater than 3 it will print when the code enters and exit a function.

With DebugLevel greater than 100 info will go to files under debug/ (one per processor, useful for parallel runs).

You have to mark the entry point:
```
call push_sub('file.function')
```

and the exit point:
```
call pop_sub()
```
Octopus has its own debugging routines.

With `DebugLevel` greater than 3 it will print when the code enters and exit a function.

With `DebugLevel` greater than 100 info will go to files under `debug/` (one per processor, useful for parallel runs).

You have to mark the entry point:
```
call push_sub('file.function')
```

and the exit point:
```
call pop_sub()
```
Octopus has its own debugging routines.

With `DebugLevel` greater than 3 it will print when the code enters and exit a function.

With `DebugLevel` greater than 100 info will go to files under `debug/` (one per processor, useful for parallel runs).

You have to mark the entry point:
```c
call push_sub('file.function')
```

and the exit point:
```c
call pop_sub()
```
Input variables can be read anywhere in the code.

- Avoid reading them more than once.
- Use `loct_parser_m` and `datasets_m` modules.
- For scalar variables use `loct_parse_*`. For example:
  ```oct
  call loct_parse_int(datasets_check('Dimensions'), 3, calc_dim)
  ```
- For blocks:
  ```oct
  block t object.
  Initialize with `loct_parse_block`, destroy with `loct_parse_end`.
  Read components with `loct_parse_block_*`
  ```
- Debug: check `exec/out.oct`.
Reading input variables

- Input variables can be read anywhere in the code.
- Avoid reading them more than once.
- Use `loct_parser_m` and `datasets_m` modules.
- For scalar variables use `loct_parse_*`. For example:
  ```octave
  call loct_parse_int(datasets_check('Dimensions'), 3, calc_dim)
  ```
- For blocks:
  - Block is object.
  - Initialize with `loct_parse_block`, destroy with `loct_parse_end`.
  - Read components with `loct_parse_block_*`.
- Debug: check `exec/out.oct`.

X. Andrade (EHU/UPV)  Developing Octopus  San Sebastian, January 2009  23 / 36
Reading input variables

- Input variables can be read anywhere in the code.
- Avoid reading them more than once.
- Use `loct_parser.m` and `datasets.m` modules.
- For scalar variables use `loct_parse_*`. For example:
  ```octave
  call loct_parse_int(datasets_check('Dimensions'), 3, calc_dim)
  ```
- For blocks:
  - Block object.
  - Initialize with `loct_parse_block`, destroy with `loct_parse_end`.
- Read components with `loct_parse_block`.
- Debug: check `exec/out.oct`. 
Reading input variables

- Input variables can be read anywhere in the code.
- Avoid reading them more than once.
- Use `loct_parser_m` and `datasets_m` modules.
- For scalar variables use `loct_parse_*`. For example:
  ```oct
call loct_parse_int(datasets_check('Dimensions'), 3, calc_dim)
```
- For blocks:
  - Block `t` object.
  - Initialize with `loct_parse_block`, destroy with `loct_parse_end`.
  - Read components with `loct_parse_block_*`.
- Debug: check `exec/out.oct`.

X. Andrade (EHU/UPV)  Developing Octopus  San Sebastian, January 2009  23 / 36
Reading input variables

- Input variables can be read anywhere in the code.
- Avoid reading them more than once.
- Use `loct_parser_m` and `datasets_m` modules.
- For scalar variables use `loct_parse_*`. For example:
  ```oct
  call loct_parse_int(datasets_check('Dimensions'), 3, calc_dim)
  ```
- For blocks:
  - `block_t` object.
  - Initialize with `loct_parse_block`, destroy with `loct_parse_end`.
  - Read components with `loct_parse_block_*`
- Debug: check `exec/out.oct`. 

X. Andrade (EHU/UPV)  
Developing Octopus  
San Sebastian, January 2009
Reading input variables

- Input variables can be read anywhere in the code.
- Avoid reading them more than once.
- Use `loct_parser_m` and `datasets_m` modules.
- For scalar variables use `loct_parse_*`. For example:
  ```oct
call loct_parse_int(datasets_check('Dimensions'), 3, calc_dim)
```
- For blocks:
  - `block_t` object.
  - Initialize with `loct_parse_block`, destroy with `loct_parse_end`.
  - Read components with `loct_parse_block_*`
- Debug: check `exec/out.oct`
Input variables can be read anywhere in the code.
Avoid reading them more than once.
Use `loct_parser.m` and `datasets.m` modules.
For scalar variables use `loct_parse_*`. For example:
```
call loct_parse_int(datasets_check('Dimensions'), 3, calc_dim)
```
For blocks:
- `block_t` object.
- Initialize with `loct_parse_block`, destroy with `loct_parse_end`.
- Read components with `loct_parse_block_*`
Debug: check `exec/out.oct`
Input variables can be read anywhere in the code.
Avoid reading them more than once.
Use `loct_parser_m` and `datasets_m` modules.
For scalar variables use `loct_parse_*`. For example:
call `loct_parse_int`(`datasets_check('Dimensions'), 3, calc_dim)
For blocks:
- `block_t` object.
- Initialize with `loct_parse_block`, destroy with `loct_parse_end`.
- Read components with `loct_parse_block_*`

Debug: check `exec/out.oct`.
Reading input variables

- Input variables can be read anywhere in the code.
- Avoid reading them more than once.
- Use `loct_parser.m` and `datasets.m` modules.
- For scalar variables use `loct_parse_*`. For example:
  ```octave
  call loct_parse_int(datasets_check('Dimensions'), 3, calc_dim)
  ```
- For blocks:
  - `block_t` object.
  - Initialize with `loct_parse_block`, destroy with `loct_parse_end`.
  - Read components with `loct_parse_block_*`
- Debug: check `exec/out.oct`.
Octopus data structures

- Review of the main objects and routines.
- Ordered by complexity.
Octopus data structures

- Review of the main objects and routines.
- Ordered by complexity.
The information about the ions is contained in a `geometry_t` object.

- The number of atoms is `geometry_t%natoms`.
- `geometry_t%atoms` is an array of `atom_t`.
- The position is contained by `atom_t%x(1:MAX_DIM)`.
The information about the ions is contained in a `geometry_t` object.

- **The number of atoms is** `geometry_t%natoms`.
- `geometry_t%atoms` is an array of `atom_t`.
- The position is contained by `atom_t%x(1:MAX_DIM)`.
The information about the ions is contained in a `geometry_t` object.
The number of atoms is `geometry_t%natoms`.
`geometry_t%atoms` is an array of `atom_t`.
The position is contained by `atom_t%x(1:MAX_DIM)`.
The information about the ions is contained in a `geometry_t` object.

The number of atoms is `geometry_t%natoms`.

`geometry_t%atoms` is an array of `atom_t`.

The position is contained by `atom_t%x(1:MAX_DIM)`.
Simulation box

- Information about the box are in `simul_box_t`.
- Dimension of the space: `simul_box_t%dim`.
- Periodic dimensions: `simul_box_t%periodic_dim`.
Information about the box are in `simul_box_t`.

Dimension of the space: `simul_box_t%dim`.

Periodic dimensions: `simul_box_t%periodic_dim`.
Simulation box

- Information about the box are in `simul_box_t`.
- Dimension of the space: `simul_box_t%dim`.
- Periodic dimensions: `simul_box_t%periodic_dim`.
The mesh

- Main component of Octopus.
  - Flexible and general approach.
  - Routines to treat the mesh are complicated.
  - The rest of the code is simple.
  - Object: \texttt{mesh\_t}.
The mesh

- Main component of Octopus.
- **Flexible and general approach.**
- Routines to treat the mesh are complicated.
- The rest of the code is simple.
- Object: `mesh_t`.
The mesh

- Main component of Octopus.
- Flexible and general approach.
- **Routines to treat the mesh are complicated.**
- The rest of the code is simple.
- Object: `mesh_t`.

The mesh

- Main component of Octopus.
- Flexible and general approach.
- Routines to treat the mesh are complicated.
- The rest of the code is simple.

Object: mesh_t.
The mesh

- Main component of Octopus.
- Flexible and general approach.
- Routines to treat the mesh are complicated.
- The rest of the code is simple.
- Object: `mesh_t`.
Functions in the mesh

- Functions are simple one index arrays.
- Size is `mesh_t%np` for normal functions.
- To calculate derivatives: size `mesh_t%np_part`.
- Each component is the value in a point.
- The position of each point is given by `mesh_t%x(1:MAX_DIM, :)`
Functions in the mesh

- Functions are simple one index arrays.
- **Size is** `mesh_t%np` for normal functions.
- To calculate derivatives: size `mesh_t%np_part`.
- Each component is the value in a point.
- The position of each point is given by `mesh_t%x(1:MAX_DIM, :)`.
Functions in the mesh

- Functions are simple one index arrays.
- Size is `mesh_t%np` for normal functions.
- To calculate derivatives: size `mesh_t%np_part`.
- Each component is the value in a point.
- The position of each point is given by `mesh_t%x(1:MAX_DIM,:)`.
Functions in the mesh

- Functions are simple one index arrays.
- Size is `mesh_t%np` for normal functions.
- To calculate derivatives: size `mesh_t%np_part`.
- Each component is the value in a point.
- The position of each point is given by `mesh_t%x(1:MAX_DIM, :)`
Functions in the mesh

- Functions are simple one index arrays.
- Size is `mesh_t%np` for normal functions.
- To calculate derivatives: size `mesh_t%np_part`.
- Each component is the value in a point.
- The position of each point is given by `mesh_t%x(1:MAX_DIM, :)`.

X. Andrade (EHU/UPV)  Developing Octopus  San Sebastian, January 2009  28 / 36
Operating over functions

- Local operations are just performed per component.
- For other operations we have to consider:
  - Curvilinear coordinates.
  - Parallelization in domains.
- Reduction operations (from mesh_functions.m):
  - Integration \( X(mf\_integrate) \)
  - Dot product \( X(mf\_dotp) \)
  - Norm \( X(mf\_nrm2) \)
Local operations are just performed per component.

For other operations we have to consider:
- Curvilinear coordinates.
- Parallelization in domains.

Reduction operations (from `mesh_functions.m`):

<table>
<thead>
<tr>
<th>Operation</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integration</td>
<td><code>X(mf_integrate)</code></td>
</tr>
<tr>
<td>Dot product</td>
<td><code>X(mf_dotp)</code></td>
</tr>
<tr>
<td>Norm</td>
<td><code>X(mf_nrm2)</code></td>
</tr>
</tbody>
</table>
Operating over functions

- Local operations are just performed per component.
- For other operations we have to consider:
  - **Curvilinear coordinates.**
  - Parallelization in domains.

Reduction operations (from `mesh_functions.n`):

<table>
<thead>
<tr>
<th>Operation</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integration</td>
<td><code>X(mf_integrate)</code></td>
</tr>
<tr>
<td>Dot product</td>
<td><code>X(mf_dotp)</code></td>
</tr>
<tr>
<td>Norm</td>
<td><code>X(mf_nrm2)</code></td>
</tr>
</tbody>
</table>
Operating over functions

- Local operations are just performed per component.
- For other operations we have to consider:
  - Curvilinear coordinates.
  - Parallelization in domains.

- Reduction operations (from mesh_functions.m):
  - Integration: \[ X(mf\_integrate) \]
  - Dot product: \[ X(mf\_dotp) \]
  - Norm: \[ X(mf\_nrm2) \]
Operating over functions

- Local operations are just performed per component.
- For other operations we have to consider:
  - Curvilinear coordinates.
  - Parallelization in domains.

- **Reduction operations (from `mesh_functions.m`):**
  - Integration: `X(mf_integrate)`
  - Dot product: `X(mf_dotp)`
  - Norm: `X(mf_nrm2)`
Derivatives

Derivatives are calculated using finite differences. This is performed by the `derivatives.m` module:

- Laplacian: `X(derivatives_lapl)
- Gradient: `X(derivatives_grad)
- Divergence: `X(derivatives_div)
- Curl: `X(derivatives_curl)
Derivatives

- Derivatives are calculated using finite differences.
- This is performed by the `derivatives.m` module:

<table>
<thead>
<tr>
<th>Derivative</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Laplacian</td>
<td><code>X(derivatives_lapl)</code></td>
</tr>
<tr>
<td>Gradient</td>
<td><code>X(derivatives_grad)</code></td>
</tr>
<tr>
<td>Divergence</td>
<td><code>X(derivatives_div)</code></td>
</tr>
<tr>
<td>Curl</td>
<td><code>X(derivatives_curl)</code></td>
</tr>
</tbody>
</table>
The grid

- `grid_t`: container for all mesh related objects.
- Passed to many function as the `gr` argument.
- Most used components:
  
  ```
  type(simul_box_t) :: sb
  type(mesh_t) :: mesh
  type(derivatives_t) :: der
  ```

- Shortcut macros:
  
  ```
  NP           | gr%mesh%np
  NP_PART      | gr%mesh%np_part
  NDIM         | gr%sb%dim
  ```

- Avoid passing `gr` if you only need `gr%mesh`. 

X. Andrade (EHU/UPV)  Developing Octopus  San Sebastian, January 2009
The grid

- **grid_t**: container for all mesh related objects.
- **Passed to many function as the gr argument.**
- Most used components:
  
  ```
  type(simul_box_t) :: sb
  type(mesh_t) :: mesh
  type(derivatives_t) :: der
  ```

- **Shortcut macros:**
  
  ```
  NP | gr%mesh%np
  NP_PART | gr%mesh%np_part
  NDIM | gr%sb%dim
  ```

- **Avoid passing gr if you only need gr%mesh.**
The grid

- **grid_t**: container for all mesh related objects.
- Passed to many function as the `gr` argument.
- **Most used components:**
  
  ```fortran
  type(simul_box_t) :: sb
  type(mesh_t) :: mesh
  type(derivatives_t) :: der
  ```

- **Shortcut macros:**
  
  ```fortran
  NP | gr%mesh%np
  NP_PART | gr%mesh%np_part
  NDIM | gr%sb%dim
  ```

- Avoid passing `gr` if you only need `gr%mesh`. 
**The grid**

- `grid_t`: container for all mesh related objects.
- Passed to many function as the `gr` argument.
- Most used components:
  ```fortran
  type(simul_box_t) :: sb
  type(mesh_t) :: mesh
  type(derivatives_t) :: der
  ```
- **Shortcut macros:**
  ```
  NP | gr%mesh%np
  NP_PART | gr%mesh%np_part
  NDIM | gr%sb%dim
  ```
- Avoid passing `gr` if you only need `gr%mesh`. 
grid_t: container for all mesh related objects.

Passed to many function as the gr argument.

Most used components:

\[
\begin{align*}
\text{type}(&\text{simul\_box\_t}) : : \text{sb} \\
\text{type}(&\text{mesh\_t}) : : \text{mesh} \\
\text{type}(&\text{derivatives\_t}) : : \text{der}
\end{align*}
\]

Shortcut macros:

\[
\begin{align*}
\text{NP} & \quad \text{gr\%mesh\%np} \\
\text{NP\_PART} & \quad \text{gr\%mesh\%np\_part} \\
\text{NDIM} & \quad \text{gr\%sb\%dim}
\end{align*}
\]

Avoid passing gr if you only need gr\%mesh.
The states (orbitals)

- The states are stored in a special class: `states_t`
  - Represented by a two dimensional array.
  - First component is the mesh index `1:gr%mesh%np_part`.
  - Second component is the spinor index `1:st%d%dim`.
  - Each state is stored in `states_t%X(psi)(:, :, ist, ik)`.
  - Indexes are:
    - `ist`: `st%st_start:st%st_end` (state index)
    - `ik`: `st%d%kpt%start:st%d%kpt%end` (k-point/spin index)
  - States can be real or complex, use `states_are_real` or `states_are_complex` functions to know.
  - Eigenvalues are `states_t%eigenval(ist, ik)`.
  - Occupations are `states_t%occ(ist, ik)`.
The states (orbitals)

- The states are stored in a special class: `states_t`.
- Represented by a two dimensional array.
  - First component is the mesh index (1:gr%mesh%np_part).
  - Second component is the spinor index (1:st%d%dim).
  - Each state is stored in `states_t%X(psi)(:, :, ist, ik)`.
  - Indexes are:
    - `ist`: `st%st_start:st%st_end` state index
    - `ik`: `st%d%kpt%start:st%d%kpt%end` k-point/spin index
  - States can be real or complex, use `states_are_real` or `states_are_complex` functions to know.
  - Eigenvalues are `states_t%eigenval(ist, ik)`.
  - Occupations are `states_t%occ(ist, ik)`. 
The states (orbitals)

- The states are stored in a special class: `states_t`.
- Represented by a two dimensional array.
- **First component is the mesh index (1:gr%mesh%np_part).**
- Second component is the spinor index (1:st%d%dim).
- Each state is stored in `states_t%X(psi)(:, :, ist, ik)`.
- Indexes are:
  - `ist`: `st%st_start:st%st_end` - state index
  - `ik`: `st%d%kpt_start:st%d%kpt_end` - k-point/spin index
- States can be real or complex, use `states_are_real` or `states_are_complex` functions to know.
- Eigenvalues are `states_t%eigenval(ist, ik)`.
- Occupations are `states_t%occ(ist, ik)`.
The states (orbitals)

- The states are stored in a special class: `states_t`.
- Represented by a two dimensional array.
- First component is the mesh index ($1:\text{gr\%mesh\%np\_part}$).
- **Second component is the spinor index ($1:\text{st\%d\%dim}$).**
- Each state is stored in `states_t\%X(psi)(:, :, ist, ik)`.
- Indexes are:
  
<table>
<thead>
<tr>
<th>ist</th>
<th>st%st_start:st%st_end</th>
<th>state index</th>
</tr>
</thead>
<tbody>
<tr>
<td>ik</td>
<td>st%d%kpt%start:st%d%kpt%end</td>
<td>k-point/spin index</td>
</tr>
</tbody>
</table>
- States can be real or complex, use `states\_are\_real` or `states\_are\_complex` functions to know.
- Eigenvalues are `states_t\%eigenval(ist, ik)`.
- Occupations are `states_t\%occ(ist, ik)`.
The states (orbitals)

- The states are stored in a special class: `states_t`.
- Represented by a two dimensional array.
- First component is the mesh index (1:gr%mesh%np_part).
- Second component is the spinor index (1:st%d%dim).
- Each state is stored in `states_t%X(psi)(:, :, ist, ik)`.

Indexes are:

<table>
<thead>
<tr>
<th>ist</th>
<th>st%st_start:st%st_end</th>
<th>state index</th>
</tr>
</thead>
<tbody>
<tr>
<td>ik</td>
<td>st%d%kpt%start:st%d%kpt%end</td>
<td>k-point/spin index</td>
</tr>
</tbody>
</table>

- States can be real or complex, use `states_are_real` or `states_are_complex` functions to know.
- Eigenvalues are `states_t%eigenval(ist, ik)`.
- Occupations are `states_t%occ(ist, ik)`.
The states (orbitals)

- The states are stored in a special class: `states_t`.
- Represented by a two dimensional array.
- First component is the mesh index `(1:gr%mesh%np_part)`.
- Second component is the spinor index `(1:st%d%dim)`.
- Each state is stored in `states_t%X(psi)(:, :, ist, ik)`.
- Indexes are:
  
<table>
<thead>
<tr>
<th>ist</th>
<th>st%st_start:st%st_end</th>
<th>state index</th>
</tr>
</thead>
<tbody>
<tr>
<td>ik</td>
<td>st%d%kpt%start:st%d%kpt%end</td>
<td>k-point/spin index</td>
</tr>
</tbody>
</table>

- States can be real or complex, use `states_are_real` or `states_are_complex` functions to know.
- Eigenvalues are `states_t%eigenval(ist, ik)`.
- Occupations are `states_t%occ(ist, ik)`.
The states (orbitals)

The states are stored in a special class: `states_t`.

- Represented by a two dimensional array.
- First component is the mesh index (1:`gr%mesh%n_p_part`).
- Second component is the spinor index (1:`st%d%dim`).
- Each state is stored in `states_t%X(psi)(:, :, ist, ik)`.

Indexes are:

- `ist` `st%st_start:st%st_end` | state index
- `ik` `st%d%kpt%start:st%d%kpt%end` | k-point/spin index

- States can be real or complex, use `states_are_real` or `states_are_complex` functions to know.
- Eigenvalues are `states_t%eigenval(ist, ik)`.
- Occupations are `states_t%occ(ist, ik)`.
The states (orbitals)

- The states are stored in a special class: `states_t`.
- Represented by a two dimensional array.
- First component is the mesh index ($1:gr\%mesh\%np\_part$).
- Second component is the spinor index ($1:st\%d\%dim$).
- Each state is stored in `states_t\%X(psi)(:, :, ist, ik)`.
- Indexes are:
  - `ist` | `st\%st\_start:\st\%st\_end` | state index
  - `ik` | `st\%d\%kpt\_start:\st\%d\%kpt\_end` | k-point/spin index
- States can be real or complex, use `states\_are\_real` or `states\_are\_complex` functions to know.
- **Eigenvalues** are `states\_t\%eigenval(ist, ik)`.
- Occupations are `states\_t\%occ(ist, ik)`.
The states (orbitals)

- The states are stored in a special class: `states_t`.
- Represented by a two dimensional array.
- First component is the mesh index (1:gr%mesh%np_part).
- Second component is the spinor index (1:st%d%dim).
- Each state is stored in `states_t%X(psi)(:, :, ist, ik)`.

Indexes are:

<table>
<thead>
<tr>
<th>ist</th>
<th>st%st_start:st%st_end</th>
<th>state index</th>
</tr>
</thead>
<tbody>
<tr>
<td>ik</td>
<td>st%d%kpt%start:st%d%kpt%end</td>
<td>k-point/spin index</td>
</tr>
</tbody>
</table>

- States can be real or complex, use `states_are_real` or `states_are_complex` functions to know.
- Eigenvalues are `states_t%eigenval(ist, ik)`.
- Occupations are `states_t%occ(ist, ik)`.
The `system_t` object

- Container for system related objects.

```plaintext
type(geometry_t) :: geo
type(grid_t) :: gr
type(states_t) :: st
```

- `system_init` takes care of initialization.
The `system_t` object

- Container for system related objects.
  
  type(geometry_t) :: geo  
  type(grid_t) :: gr  
  type(states_t) :: st  

- `system_init` takes care of initialization.
The Hamiltonian is contained in the `hamiltonian.m` module.

To apply it use the `X(hamiltonian_apply)` function.
The Hamiltonian

- The Hamiltonian is contained in the `hamiltonian.m` module.
- To apply it use the `X(hamiltonian_apply)` function.
Octopus program is in `main.F90`.

- It calls `run` in `run.m`.
- Takes care of initialization.
- Each calculation mode provides a `run` function.
- Arguments are `system_t sys` and `hamiltonian_t hm`.
Calculation Modes

- Octopus program is in `main.F90`.
- It calls `run` in `run.m`.
- Takes care of initialization.
- Each calculation mode provides a `run` function.
- Arguments are `system_t sys` and `hamiltonian_t hm`.
Calculation Modes

- Octopus program is in main.F90.
- It calls run in run.m.
- Takes care of initialization.
- Each calculation mode provides a run functions.
- Arguments are system_t sys and hamiltonian_t hm.
Octopus program is in `main.F90`.
It calls `run` in `run.m`.
Takes care of initialization.
Each calculation mode provides a run functions.
Arguments are `system_t sys` and `hamiltonian_t hm`.
Octopus program is in main.F90.
It calls run in run.m.
Takes care of initialization.
Each calculation mode provides a run functions.
Arguments are system_t sys and hamiltonian_t hm.
Conclusions

- Not the usual Fortran code.
- There is much more.
- Look at the code an ask.
- Octopus offers a framework to implement and test new ideas.
- Developers are welcome.
Conclusions

- Not the usual Fortran code.
- **There is much more.**
- Look at the code and ask.
- Octopus offers a framework to implement and test new ideas.
- Developers are welcome.
Conclusions

- Not the usual Fortran code.
- There is much more.
- **Look at the code an ask.**
- Octopus offers a framework to implement and test new ideas.
- Developers are welcome.
Conclusions

- Not the usual Fortran code.
- There is much more.
- Look at the code and ask.
- Octopus offers a framework to implement and test new ideas.
- Developers are welcome.
Conclusions

- Not the usual Fortran code.
- There is much more.
- Look at the code and ask.
- Octopus offers a framework to implement and test new ideas.
- Developers are welcome.