Octopus: structure of the code

Martin Lüders

Octopus Advanced Course 2023, MPSD Hamburg
Overview

- General structure of the code
- Structure of a calculation: GS and TD
- Real-space representation: and mesh functions and operators
- Multisystems: systems and interactions
- Time propagation
Introduction

Code refactoring:
- big changes to the code, while keeping the code functional
- half-way into the processes
- not everything is where it should be
  e.g. electrons and ions not yet fully in the multisystem framework

This talk: Representation strongly simplified
The global structure
General code structure

- Code is modular. We have components for:
  - I/O: reading and writing data
  - messages: writing information, warnings and error messages
  - parallelism
  - profiler
  - input parser
  - etc.

- Most components have some associated data structures

- Most components have \_\_init() and \_\_end() routines.
  - \_\_init():
    - initialize data structures
    - read related input variables
  - \_\_end():
    - clean up: release memory

- Slowly transitioning to proper classes.
General code structure

Example: the main routine:

    program main
    [...]  ! start code components
    call global_init()  ! initialize the mpi, clocks, etc.
    call parser_init()  ! initialize the input parser
    call messages_init()  ! initialize the message system
    call walltimer_init()  ! initialize the timer module
    call io_init()  ! initialize the I/O subsystem
    call calc_mode_par_init()  ! initialize parallelization strategy
    call profiling_init(global_namespace)  ! initialize and start the profiling system
    call run(global_namespace, inp_calc_mode)  ! pass control to the 'actual code' running the calculation
    ! stop code components
    call profiling_end(global_namespace)
    call calc_mode_par_end()
    call io_end()
    call walltimer_end()
    call messages_end()
    call parser_end()
    call global_end()
    end program
General code structure: Directory structure

What you find in the package:

- **build**: related to build system
- **doc**: Documentation, manuals, tutorials
- **external_libs**: external libraries shipped with Octopus
- **liboct_parser**: the input parser library
- **m4**: m4 macros for autotools
- **scripts**: some analysis scripts
- **share**: pseudopotentials, GPU kernels, recipes, etc.
- **src**: our octopus lives here
- **testsuite**: test files and input files for the tests
- ...
General code structure: Directory structure

Content of src/:

- basic: general routines
- basis_set: atomic orbitals
- classical: classical particle classes
- CmakeList.txt: list of source files for CMake
- common-rules.make
- dftbplus: interface to DFTB+
- electrons: all related to electrons
- fdep: (helper script for automake)
- grid: grid, mesh, etc.
- hamiltonian: Hamiltonain (general, but also electronic, e.g. projectors, v_xc)
- include: macro definitions
- interactions: interaction classes
- ions: ions, boxes, symmetries
- main: main routines
- Makefile.am: list of files for autotools
- Makefile.in: Makefile template for autotools
General code structure: Directory structure

Content of `src/`:

- **math**: mathematical routines, interfaces to blas, fftw, etc.
- **maxwell**: All about Maxwell
- **multisystem**: The multisystem framework (propagator class)
- **opt_control**: optimal control
- **output**: the output module
- **poisson**: the Poisson solver and interface to PSOLVER library
- **scf**: SCF cycle: LCAO, convergence criteria, mixer
- **species**: mainly pseudopotentials
- **states**: wave functions, density, etc.
- **sternheimer**: linear response
- **td**: propagators (old formalism)
- **utils**: external utilities
Files in the `src/main/` folder:

- casida.F90
- casida_inc.F90
- geom_opt.F90
- ground_state.F90
- invert_ks.F90
- main.F90
- phonons_fd.F90
- pulpo.F90
- run.F90
- static_pol.F90
- system_factory.F90
- test.F90
- time_dependent.F90
General code structure

Common objects

- **gr** The object containing the grid
- **mesh** The object containing the grid or mesh
- **space** Description of the periodicity and dimensionality
- **st** The states (i.e. wave functions for electrons)
- **hm** The Hamiltonian
- **scf** An object containing information about the SCF cycle
- **td** An object containing information about time-dependent runs
Finding your way: Doxygen

- Search for:
  - files
  - Modules
  - classes

- Information on:
  - general comments (if provided)
  - class members
  - inheritance
  - function arguments
  - source listing (not working for *.inc.F90)
How a calculation works...
The calculation modes

- **gs**: Calculation of the ground state.
- **unocc**: Calculation of unoccupied/virtual KS states. Can also be used for a non-self-consistent calculation of states at arbitrary k-points, if density.obf from gs is provided in the restart/gs directory.
- **td**: Time-dependent calculation (experimental for periodic systems).
- **go**: Optimization of the geometry.
- **opt_control**: Optimal control.
- **em_resp**: Calculation of the electromagnetic response: electric polarizabilities and hyperpolarizabilities and magnetic susceptibilities (experimental for periodic systems).
- **casida**: Excitations via Casida linear-response TDDFT; for finite systems only.
- **vdw**: Calculate van der Waals coefficients.
- **vib_modes**: Calculation of the vibrational modes.
- **invert ks**: Invert the Kohn-Sham equations (experimental).
- **recipe**: Prints out a tasty recipe.
- **... and others**
The calculation modes

The run() routine:

```fortran
subroutine run(cm):
    integer, intent(in) :: cm
    ...
    select case (calc_mode_id)
    case (OPTION__CALCULATIONMODE__GS) ! ground state
        call ground_state_run(systems, from_scratch)
    case (OPTION__CALCULATIONMODE__UNOCC) ! unoccupied states
        call unocc_run(systems, from_scratch)
    case (OPTION__CALCULATIONMODE__TD) ! time propagation
        call time_dependent_run(systems, from_scratch)
    case (OPTION__CALCULATIONMODE__GO) ! geometry optimization
        call geom_opt_run(systems, from_scratch)
    ...
    end select
    ...
end subroutine run
```

Concentrate on:

- Ground state calculation
- Time propagation
Ground state calculation (electrons only)

- **Startup:**
  - initial wave functions:
    - Restart
    - LCAO from diagonalized pseudo-wavefunctions
  - random wave functions
  - setup initial Hamiltonian

- **SCF cycle:**
  - run the eigenvalue solver
  - calculate new occupations and new density
  - calculate total energy
  - mix potentials or densities
  - update Hamiltonian
  - check convergence criteria
(simplified) SCF cycle: (scf/scf.F90)

do iter = 1, scf%max_iter
   scf%eigens%converged = 0
   call eigensolver_run(scf%eigens, namespace, gr, st, hm, iter)
   call states_elec_fermi(st, namespace, gr%mesh)
   call density_calc(st, gr, st%rho)
   call v_ks_calc(ks, namespace, space, hm, st, ions)
   call mixfield_set_vout(scf%mixfield, hm%vhxc)
   call energy_calc_total(namespace, space, hm, gr, st, iunit = 0)
   call mixing(scf%smix)
   call mixfield_get_vnew(scf%mixfield, hm%vhxc)
   call hamiltonian_elec_update_pot(hm, gr%mesh)
   call mixfield_set_vin(scf%mixfield, hm%vhxc(1:gr%mesh%np, 1:nspin))
   ! check convergence
enddo
Ground state calculation (electrons only)

Eigenvalue problem:

call eigensolver.run(scf%eigens, namespace, gr, st, hm, iter)

- Matrix is huge and sparse: no direct diagonalization
- Iterative schemes:
  - Conjugate gradient: (cg, cg_new)
  - Pre-conditioned Lanczos (plan)
  - Residual minimization scheme, direct inversion in the iterative subspace (rmmdiis)
  - Chebyshev filtering (chebyshev_filter)
The eigensolver contains many applications of the Hamiltonian.
The application of the Hamiltonian needs to be fast.
`hamiltonian_update` collects potentials of the same kind.
Put costly calculations in `hamiltonian_update`.
Time-dependent calculations (for electrons)

- **Startup:**
  - Restart from ground state calculation

- **Propagation:**
  \[
  \varphi_i(r, t + \Delta t) = \hat{T} \exp \left\{ -i \int_t^{t+\Delta t} dt \hat{H} \varphi_i(r, t) \right\}
  \]

- Different ways to approximate
  - the integration
    - TDPropagator for electrons
    - TDSYSTEMPropagator for multisystem framework
  - the exponential
    - TDExponentialMethod

- Still different implementations for matter (electrons + ions) and the multisystem approach (more on new approach later)

- This will change soon...
Modules to write out data:

- file output:
  - larger amounts of data
  - not intended for standard out

- messages: everything for stdout and stderr
  - Information
  - Warnings
  - Error messages
Distinguish between

- **mesh data:**
  - Examples: density $n(\vec{r})$, fields $\vec{E}(\vec{r})$, wave functions $\phi_i(\vec{r})$
  - handled by `output_oct.m` and `io_oct.m`.
  - also used when these functions are time-dependent.

- **time-dependent functions (scalar or vectors, but no fields)**
  - Examples: total energy $E(t)$, magnetization $\vec{m}(t)$
  - handled by `td_write.m`
Data output

Mesh data: output_oct.m and io_oct.m:

- output/output.F90: output_init(outp, namespace, space, st, nst, ks):
  - parse output variable
    (via io_function_read_what_how_when(); new output options need to be implemented here.)
  - parse other output related input variables

- output/output.F90: output_all(outp, namespace, space, dir, gr, ions, iter, st, hm, ks):
  - is automatically called at end of SCF (or similar) calculation, or at specified iteration intervals.
  - calls specific output routines, if requested. (e.g. output_states())
Data output

What, how and when to write?

- **mesh data:**
  From grid/io_function.F90:

  ```fortran
  subroutine io_function_read_what_how_when(namespace, space, what, how, output_interval, &
      what_tag_in, how_tag_in, output_interval_tag_in, ignore_error)
      type(namespace_t), intent(in) :: namespace
      type(space_t), intent(in) :: space
      logical, intent(inout) :: what(MAX_OUTPUT_TYPES) !> which quantities?
      integer(8), intent(out) :: how(0:MAX_OUTPUT_TYPES) !> output format
      integer, intent(out) :: output_interval(0:MAX_OUTPUT_TYPES)
      character(len=__), optional, intent(in) :: what_tag_in
      character(len=__), optional, intent(in) :: how_tag_in
      character(len=__), optional, intent(in) :: output_interval_tag_in
      logical, optional, intent(in) :: ignore_error !> Ignore error check.
  end subroutine io_function_read_what_how_when
  ```

- **time-dependent functions:**
  From grid/io_function.F90:

  ```fortran
  subroutine td_write_init(writ, namespace, space, outp, gr, st, hm, ions, ks, ions_move, &
      with_gauge_field, kick, iter, max_iter, dt, mc)
  end subroutine td_write_init
  ```
Data output

Example: Output of a the density

From output/output_states.inc.F90:

```fortran
subroutine output_states(outp, namespace, space, dir, st, gr, ions, hm, iter)
...
if (outp%what_now(OPTION__OUTPUT__DENSITY, iter)) then
  fn_unit = units_out%length**(space%dim)
  do is = 1, st%d%nspin
    if (st%d%nspin == 1) then
      write(fname, '(a)') 'density'
    else
      write(fname, '(a,i1)') 'density-sp', is
    end if
    call dio_function_output(outp%how(OPTION__OUTPUT__DENSITY), &
      dir, fname, namespace, space, gr%mesh, &
      st%rho(:, is), fn_unit, ierr, ions = ions, &
      grp = st%dom_st_kpt_mpi_grp)
  end do
end if
```

Octopus: code structure
Example: Output of a mesh function

From grid/io_functions.inc.F90:

```fortran
subroutine X(io_function_output) (how, dir, fname, namespace, space, mesh, ff, unit, &
ierr, ions, grp, root, is_global)

integer(8), intent(in) :: how  !< output format descriptor
character(len=*), intent(in) :: dir  !< directory
character(len=*), intent(in) :: fname  !< filename
type(namespace_t), intent(in) :: namespace  !< namespace
type(space_t), intent(in) :: space
type(mesh_t), intent(in) :: mesh
R_TYPE, target, intent(in) :: ff(:)  !< mesh function to be printed
type(unit_t), intent(in) :: unit  !< output units
integer, intent(out) :: ierr

integer, optional, intent(in) :: ions

type(mpi_grp_t), optional, intent(in) :: grp  !< the group that shares the same data,
!< must contain the domains group

integer, optional, intent(in) :: root  !< which process is going to write the data
logical, optional, intent(in) :: is_global  !< Input data is mesh%np_global?
!< And, thus, it has not be gathered
```

This routine deals with domain parallelization.
Implemented in `messages_oct.m`:

- several functions to write messages
  - `messages_info(no_lines, iunit, verbose_limit, stress, all_nodes)` writes information, and can be controlled by `verbose-level`.
  - `messages_warning(no_lines, all_nodes, namespace)` writes warnings (independent of `verbose level`)
  - code continues
  - `messages_fatal(no_lines, only_root_writes, namespace)` writes fatal error message
  - stops the code.

- provides (global) `message array`
- handles parallelism
Messages: Info, Warnings, Errors

Examples:

- **Info**
  ```fortran
  write(message(1), '(a, i4, a)') 'Info: SCF converged in ', iter, ' iterations'
  write(message(2), '(a)')
  call messages_info(2)
  ```

- **Warning**
  ```fortran
  if (ierr /= 0) then
    message(1) = 'Unable to write mixing information.'
    call messages_warning(1)
  end if
  ```

- **Error**
  ```fortran
  select type (system)
  class is (multisystem_basic_t)
    message(1) = "CalculationMode = gs not implemented for multi-system calculations"
    call messages_fatal(1)
  type is (electrons_t)
    call ground_state_run_legacy(system, from_scratch)
  end select
  ```