Octopus: structure of the code

Martin Lüders

Octopus Course 2021, 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
Introduction

Code refactoring:

- big changes to the code, while keeping the code functional
- half-way into the processes
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
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
Introduction

Code refactoring:
- big changes to the code, while keeping the code functional
- half-way into the processes
- not everything is where is 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 very modular. We have components for:
General code structure

- Code is very modular. We have components for:
  - I/O: reading and writing data

Octopus: code structure
General code structure

- Code is very modular. We have components for:
  - I/O: reading and writing data
  - messages: writing information, warnings and error messages

Octopus: code structure
General code structure

- Code is very modular. We have components for:
  - I/O: reading and writing data
  - messages: writing information, warnings and error messages
  - parallelism
General code structure

- Code is very modular. We have components for:
  - I/O: reading and writing data
  - messages: writing information, warnings and error messages
  - parallelism
  - profiler
General code structure

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

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

- Code is very 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
Code is very 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.
General code structure

- Code is very 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():
Code is very 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
Code is very 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
General code structure

- Code is very 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():
General code structure

- Code is very 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
General code structure

Example: the main routine:

```fortran
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() ! initialize and start the profiling system

call run(inp_calc_mode) ! pass control to the 'actual code' running the calculation

! stop code components

call profiling_end()
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

...
Content of `src/`:

- **basic**: general routines
- **basis_set**: atomic orbitals
- **classical**: classical particle classes
- **common-rules.make**: interface to DFTB+
- **dftbplus**: all related to electrons
- **electrons**: (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**
- **Makefile.in**
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 and 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
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
```
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 spurious-wavefunctions
    - random wave functions
  - setup initial Hamiltonian
Ground state calculation (electrons only)

- **Startup:**
  - initial wave functions:
    - Restart
    - LCAO from diagonalized speudo-wavefunctions
    - random wave functions
  - setup initial Hamiltonian
- **SCF cycle:**
  - run the eigenvalue solver
Ground state calculation (electrons only)

- **Startup:**
  - initial wave functions:
    - Restart
    - LCAO from diagonalized speudo-wavefunctions
    - random wave functions
  - setup initial Hamiltonian
- **SCF cycle:**
  - run the eigenvalue solver
  - calculate new occupations and new density
Ground state calculation (electrons only)

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

**SCF cycle:**
- run the eigenvalue solver
- calculate new occupations and new density
- calculate total energy
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
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
Ground state calculation (electrons only)

- **Startup:**
  - initial wave functions:
    - Restart
    - LCAO from diagonalized speudo-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
Ground state calculation (electrons only)

(simplified) SCF cycle: (scf/scf.F90)

```fortran
  do iter = 1, scf%max_iter
```
Ground state calculation (electrons only)

(simplified) SCF cycle: (scf/scf.F90)

```fortran
    do iter = 1, scf%max_iter
        scf%eigens%converged = 0
```

Octopus: code structure
Ground state calculation (electrons only)

(simplified) SCF cycle: (scf/scf.F90)

```fortran
    do iter = 1, scf%max_iter
      scf%eigens%converged = 0
      call eigensolver_run(scf%eigens, namespace, gr, st, hm, iter)
    enddo
```
(simplified) SCF cycle: (scf/scf.F90)

```fortran
    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)
    enddo
```
Ground state calculation (electrons only)

(simplified) SCF cycle: (scf/scf.F90)

```fortran
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)
enddo
```
Ground state calculation (electrons only)

(simplified) SCF cycle: (scf/scf.F90)

```fortran
  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)
  ! check convergence
  enddo
```
Ground state calculation (electrons only)

(simplified) SCF cycle: (scf/scf.F90)

```fortran
   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)
   enddo
```
(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)
Ground state calculation (electrons only)

(simplified) SCF cycle: (scf/scf.F90)

```fortran
  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)
```
Ground state calculation (electrons only)

(simplified) SCF cycle: (scf/scf.F90)

```fortran
  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)
  enddo
```

Octopus: code structure
Ground state calculation (electrons only)

(simplified) SCF cycle: (scf/scf.F90)

```fortran
  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)
  enddo
```
Ground state calculation (electrons only)

(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))
Ground state calculation (electrons only)

(simplified) SCF cycle: (scf/scf.F90)

```fortran
  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
```
(simplified) SCF cycle: (scf/scf.F90)

```fortran
    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)
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 \((rmmdiis)\)
Time-dependent calculations (for electrons)

- **Startup:**
  - Restart from ground state calculation
Time-dependent calculations (for electrons)

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

- **propagation:**

\[ \varphi_i(r, t + \Delta t) = \hat{T} \exp \left\{ -i \int_t^{t+\Delta t} dt \, \hat{H} \varphi_i(r, t) \right\} \]
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
    - TDSysytemPropagator for multisystem framework
  - the exponential
    - TDExponentialMethod
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
    - TDSysstemPropagator for multisystem framework
  - the exponential
    - TDExponentialMethod
- still different implementations for matter (electrons + ions) and the multisystem approach (more on new approach later)
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
    - `TDSysytemPropagator` 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...
The grid
Real-space grid

The grid describes a number of things:

- the mesh (the actual points in space)
- the simulation box (region of space over which the mesh extends)
- the derivatives
- the stencil
Real-space grid

The grid describes a number of things:

- the mesh (the actual points in space)
- the simulation box (region of space over which the mesh extends)
- the derivatives
- the stencil

```fortran
! Components are public by default
type grid_t
  type(simul_box_t) :: sb
  type(mesh_t) :: mesh
  type(derivatives_t) :: der
  class(coordinate_system_t), pointer :: coord_system
  type(stencil_t) :: stencil
  type(symmetries_t) :: symm
end type grid_t
```
Real-space grid

The mesh:

- Usually uniform (curvilinear meshes or double grids are possible)
Real-space grid

The mesh:
- Usually uniform (curvilinear meshes or double grids are possible)
- can be distributed over processes (domain decomposition)
Real-space grid

The mesh:

- Usually uniform (curvilinear meshes or double grids are possible)
- can be distributed over processes (domain decomposition)
- access via linear indices (local and global index)
Real-space grid

The mesh:

- Usually uniform (curvilinear meshes or double grids are possible)
- can be distributed over processes (domain decomposition)
- access via linear indices (local and global index)
- We need some 'extra points':
Real-space grid

The mesh:

- Usually uniform (curvilinear meshes or double grids are possible)
- can be distributed over processes (domain decomposition)
- access via linear indices (local and global index)
- We need some 'extra points':
  - for boundary conditions:
    functions on these points are not updated
Real-space grid

The mesh:

- Usually uniform (curvilinear meshes or double grids are possible)
- can be distributed over processes (domain decomposition)
- access via linear indices (local and global index)
- We need some 'extra points':
  - for boundary conditions:
    functions on these points are not updated
  - halo points (ghost points):
    when using domain decompositions, each process needs access to neighboring domains.
Real-space grid

Memory layout:

- mesh sizes:
Real-space grid

Memory layout:

- mesh sizes:
  - np number of local 'inner' points
Real-space grid

Memory layout:

- mesh sizes:
  - `np` number of local 'inner' points
  - `np_part` number of local 'inner' points + 'ghost' points + boundary points
Real-space grid

Memory layout:

- **mesh sizes**:
  - $\text{np}$: number of local 'inner' points
  - $\text{np_part}$: number of local 'inner' points + 'ghost' points + boundary points
  - $\text{np_global}$: number of global 'inner' points
Real-space grid

Memory layout:

- mesh sizes:
  - \( np \) number of local 'inner' points
  - \( np_{\text{part}} \) number of local 'inner' points + 'ghost' points + boundary points
  - \( np_{\text{global}} \) number of global 'inner' points
  - \( np_{\text{part \_global}} \) number of global 'inner' points + boundary points
Real-space grid

Memory layout:

- mesh sizes:
  - `np` number of local 'inner' points
  - `np_part` number of local 'inner' points + 'ghost' points + boundary points
  - `np_global` number of global 'inner' points
  - `np_part_global` number of global 'inner' points + boundary points

- ordering:
Real-space grid

Memory layout:

- mesh sizes:
  - \( \text{np} \) number of local 'inner' points
  - \( \text{np\_part} \) number of local 'inner' points + 'ghost' points + boundary points
  - \( \text{np\_global} \) number of global 'inner' points
  - \( \text{np\_part\_global} \) number of global 'inner' points + boundary points

- ordering:
  - inner points first [1:np]
Real-space grid

Memory layout:

- mesh sizes:
  - \( np \) number of local 'inner' points
  - \( np_{\text{part}} \) number of local 'inner' points + 'ghost' points + boundary points
  - \( np_{\text{global}} \) number of global 'inner' points
  - \( np_{\text{part\_global}} \) number of global 'inner' points + boundary points

- ordering:
  - inner points first \([1:np]\)
  - ghost and boundary points: \([np+1:np_{\text{part}}]\)
Real-space grid

Memory layout:

- **mesh sizes:**
  - \( np \) number of local 'inner' points
  - \( np_{\text{part}} \) number of local 'inner' points + 'ghost' points + boundary points
  - \( np_{\text{global}} \) number of global 'inner' points
  - \( np_{\text{part}_{\text{global}}} \) number of global 'inner' points + boundary points

- **ordering:**
  - inner points first \([1:np]\)
  - ghost and boundary points: \([np+1:np_{\text{part}}]\)

- **mesh points:** \( \text{mesh\%x}(1:\text{mesh\%np}_{\text{part}}, 1:\text{space\%dim}) \)
## Real-space grid

**Mesh functions:**

- position dependent quantities are stored as so-called mesh functions.

### Examples:

- \( \rho(1:gr\%\text{mesh}\%\text{np}, 1:st\%d\%n\text{spin}) \) (no ghost points needed here)
- \( \text{hm}\%\text{vh}\text{artree}(1:gr\%\text{mesh}\%\text{np} \text{part}) \)
- \( \text{hm}\%\text{a}\text{ind}(1:gr\%\text{mesh}\%\text{np} \text{part}, 1:space\%\text{dim}) \)

**Operations on mesh functions:**

- **Local operations:** point-wise operation, simple loop
- **Integrations:** summation in each domain and reduction over domains
- **Derivatives:** need to consider ghost and boundary points
Real-space grid

Mesh functions:

- position dependent quantities are stored as so-called mesh functions.

Examples:

\[
\begin{align*}
\text{rho}(1:gr\%\text{mesh}\%\text{np}, 1:\text{std}\%\text{nspin}) \text{ (no ghost points needed here)} \\
\text{hm}\%\text{vhartree}(1:gr\%\text{mesh}\%\text{np}\_\text{part}) \\
\text{hm}\%\text{a\_ind}(1:gr\%\text{mesh}\%\text{np}\_\text{part}, 1:\text{space}\%\text{dim})
\end{align*}
\]
Real-space grid

Mesh functions:

- position dependent quantities are stored as so-called mesh functions.
- Examples:
  
  \[ \rho(1:gr\%mesh\%np, 1:st\%d\%nspin) \] (no ghost points needed here)
  
  \[ hm\%vhartree(1:gr\%mesh\%np\_part) \]
  
  \[ hm\%a\_ind(1:gr\%mesh\%np\_part, 1:space\%dim) \]

- wave functions are stored differently \[\rightarrow\] batches
Mesh functions:

- position dependent quantities are stored as so-called mesh functions.
- Examples:
  
  \[
  \rho(1:gr\text{mesh}\text{np}, 1:st\text{d}nspin) \quad \text{(no ghost points needed here)}
  \]
  
  \[
  hm\text{vhartree}(1:gr\text{mesh}\text{np}\text{part})
  \]
  
  \[
  hm\text{a}\text{ind}(1:gr\text{mesh}\text{np}\text{part}, 1:space\text{dim})
  \]

- wave functions are stored differently \(\rightarrow\) batches
Mesh functions:

- position dependent quantities are stored as so-called mesh functions.

Examples:

- \( \rho(1:gr\%\text{mesh}\%np, 1:st\%d\%nspin) \) (no ghost points needed here)
- \( hm\%v\text{hartree}(1:gr\%\text{mesh}\%np\_\text{part}) \)
- \( hm\%a\_\text{ind}(1:gr\%\text{mesh}\%np\_\text{part}, 1:space\%\text{dim}) \)

- wave functions are stored differently → batches

Operations on mesh functions:

- local operations: point-wise operation, simple loop
Real-space grid

Mesh functions:
- position dependent quantities are stored as so-called mesh functions.
- Examples:
  \[
  \text{rho}(1:gr\text{\%}mesh\text{\%}np, 1:st\text{\%}d\text{\%}nspin) \text{ (no ghost points needed here)}
  \]
  \[
  \text{hm\%vhartree}(1:gr\text{\%}mesh\text{\%}np\text{\_}part)
  \]
  \[
  \text{hm\%a\_ind}(1:gr\text{\%}mesh\text{\%}np\text{\_}part, 1:space\text{\%}dim)
  \]
- wave functions are stored differently \(\rightarrow\) batches

Operations on mesh functions:
- local operations: point-wise operation, simple loop
- integrations: summation in each domain and reduction over domains
Real-space grid

Mesh functions:

- position dependent quantities are stored as so-called mesh functions.
- Examples:
  
  \[
  \rho(1:gr\%\text{mesh}\%\text{np}, 1:\text{st}\%\text{d}\%\text{nspin}) \quad (\text{no ghost points needed here})
  \]
  
  \[
  \text{hm}\%\text{vhartree}(1:gr\%\text{mesh}\%\text{np}\_\text{part})
  \]
  
  \[
  \text{hm}\%\text{a}\_\text{ind}(1:gr\%\text{mesh}\%\text{np}\_\text{part}, 1:\text{space}\%\text{dim})
  \]

- wave functions are stored differently $\rightarrow$ batches

Operations on mesh functions:

- local operations: point-wise operation, simple loop
- integrations: summation in each domain and reduction over domains
- derivatives: need to consider ghost and boundary points
Real-space grid

Pre-defined operations on mesh functions:

- **dot product** $X(mf\_dotp)(\text{mesh, f1, f2, reduce, dotu, np})$
Real-space grid

Pre-defined operations on mesh functions:

- **dot product** $X(mf\_dotp)(\text{mesh}, f1, f2, \text{reduce}, \text{dotu}, \text{np})$
- **norm** $X(mf\_nrm2)(\text{mesh}, ff, \text{reduce})$
Real-space grid

Pre-defined operations on mesh functions:

- **dot product**  \( X(\text{mf}_\text{dotp})(\text{mesh}, f1, f2, \text{reduce}, \text{dotu}, \text{np}) \)
- **norm**  \( X(\text{mf}_\text{nrm2})(\text{mesh}, ff, \text{reduce}) \)
- **Laplacian**  \( X(\text{derivatives}_\text{lapl})(\text{der}, ff, \text{op}_ff, \text{ghost}_\text{update}, \text{set}_bc, \text{factor}) \)
Real-space grid

Pre-defined operations on mesh functions:

- **dot product**  \( \text{X(mf\_dotp)}(\text{mesh}, f1, f2, \text{reduce}, \text{dotu}, \text{np}) \)
- **norm**  \( \text{X(mf\_nrm2)}(\text{mesh}, ff, \text{reduce}) \)
- **Laplacian**  \( \text{X(derivatives\_lapl)}(\text{der}, ff, \text{op\_ff}, \text{ghost\_update}, \text{set\_bc}, \text{factor}) \)
- **gradient**  \( \text{X(derivatives\_grad)}(\text{der}, ff, \text{op\_ff}, \text{ghost\_update}, \text{set\_bc}) \)
Real-space grid

Pre-defined operations on mesh functions:

- **dot product**: $X(mf\_dotp)(\text{mesh}, f1, f2, \text{reduce}, \text{dotu}, \text{np})$
- **norm**: $X(mf\_nrm2)(\text{mesh}, ff, \text{reduce})$
- **Laplacian**: $X(\text{derivatives\_lapl})(\text{der}, ff, \text{op\_ff}, \text{ghost\_update}, \text{set\_bc}, \text{factor})$
- **gradient**: $X(\text{derivatives\_grad})(\text{der}, ff, \text{op\_ff}, \text{ghost\_update}, \text{set\_bc})$
Real-space grid

Pre-defined operations on mesh functions:

- **dot product** $X(mf\_dotp)(\text{mesh}, f1, f2, \text{reduce}, \text{dotu}, \text{np})$
- **norm** $X(mf\_nrm2)(\text{mesh}, ff, \text{reduce})$
- **Laplacian** $X(\text{derivatives\_lapl})(\text{der}, ff, \text{op}\_ff, \text{ghost\_update}, \text{set\_bc}, \text{factor})$
- **gradient** $X(\text{derivatives\_grad})(\text{der}, ff, \text{op}\_ff, \text{ghost\_update}, \text{set\_bc})$

However: We are trying to use batches wherever possible.
Batches

- often one has to operate on many mesh functions at once (e.g. wave functions)
Batches

- often one has to operate on many mesh functions at once (e.g. wave functions)
- more effective to swap mesh index and function index: 'packed form': fast index is now over states.
Batches

- often one has to operate on many mesh functions at once (e.g. wave functions)
- more effective to swap mesh index and function index: 'packed form': fast index is now over states.
- excerpt from batch_t:

```fortran
!> unpacked variables; linear variables are pointers with different shapes
FLOAT, pointer, contiguous, public :: dff(:, :, :)
CMPLX, pointer, contiguous, public :: zff(:, :, :)
FLOAT, pointer, contiguous, public :: dff_linear(:, :)
CMPLX, pointer, contiguous, public :: zff_linear(:, :)

!> packed variables; only rank-2 arrays due to padding to powers of 2
FLOAT, pointer, contiguous, public :: dff_pack(:, :)
CMPLX, pointer, contiguous, public :: zff_pack(:, :)
```
Batches

- often one has to operate on many mesh functions at once (e.g. wave functions)
- more effective to swap mesh index and function index: 'packed form': fast index is now over states.
- excerpt from `batch_t`:

```fortran
!> unpacked variables; linear variables are pointers with different shapes
FLOAT, pointer, contiguous, public :: dff(:, :, :)
CMPLX, pointer, contiguous, public :: zff(:, :, :)
FLOAT, pointer, contiguous, public :: dff_linear(:, :)
CMPLX, pointer, contiguous, public :: zff_linear(:, :)
!> packed variables; only rank-2 arrays due to padding to powers of 2
FLOAT, pointer, contiguous, public :: dff_pack(:, :)
CMPLX, pointer, contiguous, public :: zff_pack(:, :)
```
- basic math operations implemented for batches
Batches

- often one has to operate on many mesh functions at once (e.g. wave functions)
- more effective to swap mesh index and function index: 'packed form': fast index is now over states.
- excerpt from batch_t:

```fortran
!> unpacked variables; linear variables are pointers with different shapes
FLOAT, pointer, contiguous, public :: dff(:, :, :)
CMPLX, pointer, contiguous, public :: zff(:, :, :)
FLOAT, pointer, contiguous, public :: dff_linear(:, :)
CMPLX, pointer, contiguous, public :: zff_linear(:, :)

!> packed variables; only rank-2 arrays due to padding to powers of 2
FLOAT, pointer, contiguous, public :: dff_pack(:, :)
CMPLX, pointer, contiguous, public :: zff_pack(:, :)
```

- basic math operations implemented for batches
- more in upcoming lectures.
Operators and observables

Calculating expectation values:

- operators can be expressed in terms of defined math operations
- many terms already implemented in the Hamiltonian
- usually no need to touch low level routines
Operators and observables

Calculating expectation values:

- operators can be expressed in terms of defined math operations
- many terms already implemented in the Hamiltonian
- usually no need to touch low level routines

Let’s look at some code: contributions to the total energy (electrons/energy_calc.F90)

```fortran
subroutine energy_calc_total(namespace, space, hm, gr, st, iunit, full)

FLOAT function X(energy_calc_electronic)(namespace, hm, der, st, terms) result(energy)

subroutine X(calculate_expectation_values)(namespace, hm, der, st, eigen, terms)
```
subroutine energy_calc_total(namespace, space, hm, gr, st, iunit, full)

    type(namespace_t), intent(in) :: namespace
    type(space_t), intent(in) :: space
    type(hamiltonian_elec_t), intent(inout) :: hm
    type(grid_t), intent(in) :: gr
    type(states_elec_t), intent(inout) :: st
    integer, optional, intent(in) :: iunit
    logical, optional, intent(in) :: full

    ...

    hm%energy%eigenvalues = states_elec_eigenvalues_sum(st)

    if (full_ .or. hm%theory_level == HARTREE .or. hm%theory_level == HARTREE_FOCK &
        .or. hm%theory_level == GENERALIZED_KOHN_SHAM_DFT) then
        if (states_are_real(st)) then
            hm%energy%kinetic = denergy_calc_electronic(namespace, hm, gr%der, st, terms=TERM_KINETIC)
            hm%energy%extern_local = denergy_calc_electronic(namespace, hm, gr%der, st, terms=TERM_LOCAL_EXTERNAL)
            hm%energy%extern_non_local = denergy_calc_electronic(namespace, hm, gr%der, st, &
                terms=TERM_NON_LOCAL_POTENTIAL)
        else
            ! same with z prefix
        end if
    else
        hm%energy%extern = hm%energy%extern_local + hm%energy%extern_non_local
    end if
end if
Operators and observables

FLOAT function X(energy_calc_electronic)(namespace, hm, der, st, terms) result(energy)

  type(namespace_t), intent(in) :: namespace
  type(hamiltonian_elec_t), intent(in) :: hm
  type(derivatives_t), intent(in) :: der
  type(states_elec_t), intent(inout) :: st
  integer, intent(in) :: terms

  R_TYPE, allocatable :: tt(:, :)

  PUSH_SUB(X(energy_calc_electronic))

  SAFE_ALLOCATE(tt(st%st_start:st%st_end, st%d%kpt%start:st%d%kpt%end))

  call X(calculate_expectation_values)(namespace, hm, der, st, tt, terms = terms)

  energy = states_elec_eigenvalues_sum(st, TOFLOAT(tt))

  SAFE_DEALLOCATE_A(tt)
  POP_SUB(X(energy_calc_electronic))

end function X(energy_calc_electronic)
subroutine X(calculate_expectation_values)(namespace, hm, der, st, eigen, terms)

    type(namespace_t), intent(in) :: namespace
    type(hamiltonian_elec_t), intent(in) :: hm
    type(derivatives_t), intent(in) :: der
    type(states_elec_t), intent(inout) :: st
    R_TYPE, intent(out) :: eigen(st%st_start:, st%d%kpt%start:) !< (:st%st_end, :st%d%kpt%end)
    integer, optional, intent(in) :: terms

    integer :: ik, minst, maxst, ib
    type(wfs_elec_t) :: hpsib

    do ik = st%d%kpt%start, st%d%kpt%end
        do ib = st%group%block_start, st%group%block_end

            minst = states_elec_block_min(st, ib)
            maxst = states_elec_block_max(st, ib)

            call st%group%psib(ib, ik)%copy_to(hpsib)
            call X(hamiltonian_elec_apply_batch)(hm, namespace, der%mesh, st%group%psib(ib, ik), hpsib, terms = terms)
            call X(mesh_batch_dotp_vector)(der%mesh, st%group%psib(ib, ik), hpsib, eigen(minst:maxst, ik), reduce = .false.)
            call hpsib%end()

        end do
    end do

    if (der%mesh%parallel_in_domains) call der%mesh%allreduce(&
        eigen(st%st_start:st%st_end, st%d%kpt%start:st%d%kpt%end))

end subroutine X(calculate_expectation_values)
Operators and observables

in hamiltonian/hamiltonian_elec_inc.F90:

```fortran
subroutine X(hamiltonian_elec_apply_batch) (hm, namespace, mesh, psib, hpsib, terms, set_bc)

...  
if (bitand(TERM_KINETIC, terms_) /= 0) then
  ASSERT(associated(hm%hm_base%kinetic))
  call profiling_in(prof_kinetic_start, TOSTRING(X(KINETIC_START)))
  call X(derivatives_batch_start)(hm%hm_base%kinetic, hm%der, epsib, hpsib, handle, &
    set_bc = .false., factor = -M_HALF/hm%mass)
  call profiling_out(prof_kinetic_start)
end if

...  
if (bitand(TERM_KINETIC, terms_) /= 0) then
  call profiling_in(prof_kinetic_finish, TOSTRING(X(KINETIC_FINISH)))
  call X(derivatives_batch_finish)(handle)
  call profiling_out(prof_kinetic_finish)
else
  call batch_set_zero(hpsib)
end if
```
Operators and observables

in *hamiltonian/hamiltonian_elec_inc.F90*:

```fortran
subroutine X(hamiltonian_elec_apply_batch) (hm, namespace, mesh, psib, hpsib, terms, set_bc)

... 
if (bitand(TERM_KINETIC, terms_) /= 0) then
    ASSERT(associated(hm%hm_base%kinetic))
    call profiling_in(prof_kinetic_start, TOSTRING(X(KINETIC_START)))
    call X(derivatives_batch_start)(hm%hm_base%kinetic, hm%der, epsib, hpsib, handle, &
        set_bc = .false., factor = -M_HALF/hm%mass)
    call profiling_out(prof_kinetic_start)
end if

... 
if (bitand(TERM_KINETIC, terms_) /= 0) then
    call profiling_in(prof_kinetic_finish, TOSTRING(X(KINETIC_FINISH)))
    call X(derivatives_batch_finish)(handle)
    call profiling_out(prof_kinetic_finish)
else
    call batch_set_zero(hpsib)
end if
```

split in start and finish routine to enable other operations during communication.
Calculations are a (huge!) waste of time...
Calculations are a (huge!) waste of time...

... if we are not writing out the results.
Calculations are a (huge!) waste of time...
... if we are not writing out the results.

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
Data output

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.
Data output

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
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.
```

- **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)
```

Octopus: code structure  Martin Lüders  34 / 58
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.
    !> Used when called from some external utility.
  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)
  ```
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
end subroutine output_states
```
Data output

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
  type(ions_t), 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, Warnings, Errors

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: Info, Warnings, Errors

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: Info, Warnings, Errors

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_rootWrites, namespace)` writes fatal error message stops the code.
Messages: Info, Warnings, Errors

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
Messages: Info, Warnings, Errors

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
Examples:

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

The multisystem framework

- allow calculation of coupled systems
The multisystem framework

- allow calculation of coupled systems
- Examples of systems:
Multisystems

The multisystem framework

- allow calculation of coupled systems
- Examples of systems:
  - maxwell
Multisystems

The multisystem framework

- allow calculation of coupled systems
- Examples of systems:
  - maxwell
  - classical particles
Multisystems

The multisystem framework

- allow calculation of coupled systems
- Examples of systems:
  - maxwell
  - classical particles
  - charged particles
Multisystems

The multisystem framework

- allow calculation of coupled systems
- Examples of systems:
  - maxwell
  - classical particles
  - charged particles
  - ions
Multisystems

The multisystem framework

- allow calculation of coupled systems

Examples of systems:

- maxwell
- classical particles
- charged particles
- ions
- electrons
Multisystems

The multisystem framework

- allow calculation of coupled systems
- Examples of systems:
  - maxwell
  - classical particles
  - charged particles
  - ions
  - electrons
  - tight binding model
Multisystems

The multisystem framework

- allow calculation of coupled systems
- Examples of systems:
  - maxwell
  - classical particles
  - charged particles
  - ions
  - electrons
  - tight binding model
  - etc.
Multisystems

The multisystem framework

- allow calculation of coupled systems
- Examples of systems:
  - maxwell
  - classical particles
  - charged particles
  - ions
  - electrons
  - tight binding model
  - etc.

- re-use as much code as possible between different systems
Multisystems

The multisystem framework

- allow calculation of coupled systems
- Examples of systems:
  - maxwell
  - classical particles
  - charged particles
  - ions
  - electrons
  - tight binding model
  - etc.

- re-use as much code as possible between different systems
- use object oriented approach!
Multisystems

The multisystem framework

- allow calculation of coupled systems
- Examples of systems:
  - maxwell
  - classical particles
  - charged particles
  - ions
  - electrons
  - tight binding model
  - etc.

- re-use as much code as possible between different systems
- use object oriented approach!
- represent systems as classes and use inheritance
System classes

Currently implemented system classes:

- multsystem_t
- system_dftb_t
- classical_particles_t
- linear_medium_t
- electrons_t
- system_mxli_t
- interaction_partner_t
- lasers_t
- external_potential_t
- multsystem_basic_t
- ions_t
- classical_particle_t
- charged_particle_t

Rounded boxes: abstract class
Arrows indicate inheritance.
The abstract class interaction_partner_t:

type, abstract :: interaction_partner_t
private
  type(namespace_t), public :: namespace
  type(clock_t), public :: clock
  type(space_t), public :: space

  type(integer_list_t), public :: supported_interactions_as_partner

  type(quantity_t), public :: quantities(MAX_QUANTITIES) !< Array of all possible quantities.
  !< The elements of the array are accessed using the
  !< quantity’s identifiers.
contains
  procedure(interaction_partner_update_exposed_quantities), deferred :: update_exposed_quantities
  procedure(interaction_partner_update_exposed_quantity), deferred :: update_exposed_quantity
  procedure(interaction_partner_init_interaction_as_partner), deferred :: init_interaction_as_partner
  procedure(interaction_partner_copy_quantities_to_interaction), deferred :: copy_quantities_to_interaction
end type interaction_partner_t
System classes

The abstract class `interaction_partner_t`:

```fortran
type, abstract :: interaction_partner_t
private
  type(namespace_t), public :: namespace
  type(clock_t), public :: clock
  type(space_t), public :: space

  type(integer_list_t), public :: supported_interactions_as_partner

  type(quantity_t), public :: quantities(MAX_QUANTITIES) !< Array of all possible quantities.
  !< The elements of the array are accessed using the
  !< quantity's identifiers.
contains
  procedure(interaction_partner_update_exposed_quantities), deferred :: update_exposed_quantities
  procedure(interaction_partner_update_exposed_quantity), deferred :: update_exposed_quantity
  procedure(interaction_partner_init_interaction_as_partner), deferred :: init_interaction_as_partner
  procedure(interaction_partner_copy_quantities_to_interaction), deferred :: copy_quantities_to_interaction
end type interaction_partner_t
```

- abstract class: cannot be instantiated
The abstract class `interaction_partner_t`:

```fortran
  type, abstract :: interaction_partner_t
  private
    type(namespace_t), public :: namespace
    type(clock_t),       public :: clock
    type(space_t),       public :: space
    type(integer_list_t), public :: supported_interactions_as_partner
    type(quantity_t),    public :: quantities(MAX_QUANTITIES) !< Array of all possible quantities.
                             !< The elements of the array are accessed using the
                             !< quantity's identifiers.
  contains
    procedure(interaction_partner_update_exposed_quantities), deferred :: update_exposed_quantities
    procedure(interaction_partner_update_exposed_quantity), deferred :: update_exposed_quantity
    procedure(interaction_partner_init_interaction_as_partner), deferred :: init_interaction_as_partner
    procedure(interaction_partner_copy_quantities_to_interaction), deferred :: copy_quantities_to_interaction
  end type interaction_partner_t
```

- **abstract class: cannot be instantiated**
- **defines basic variables and interface for all classes which can be partner in an interaction**
System classes

The abstract class interaction_partner_t:

type, abstract :: interaction_partner_t
private
  type(namespace_t), public :: namespace
  type(clock_t), public :: clock
  type(space_t), public :: space

  type(integer_list_t), public :: supported_interactions_as_partner

  type(quantity_t), public :: quantities(MAX_QUANTITIES) !< Array of all possible quantities.
  !< The elements of the array are accessed using the
  !< quantity's identifiers.
contains
  procedure(interaction_partner_update_exposed_quantities), deferred :: update_exposed_quantities
  procedure(interaction_partner_update_exposed_quantity), deferred :: update_exposed_quantity
  procedure(interaction_partner_init_interaction_as_partner), deferred :: init_interaction_as_partner
  procedure(interaction_partner_copy_quantities_to_interaction), deferred :: copy_quantities_to_interaction
end type interaction_partner_t

- abstract class: cannot be instantiated
- defines basic variables and interface for all classes which can be partner in an interaction
- defines list of exposed quantities
System classes

The abstract class system_t:

type, extends(interaction_partner_t), abstract :: system_t

private
  class(propagator_t), pointer, public :: prop => null()

  integer :: accumulated_loop_ticks
  integer, public :: interaction_timing !< parameter to determine if interactions

  type(integer_list_t), public :: supported_interactions
  type(interaction_list_t), public :: interactions !< List with all the interactions of this system

  type(mpi_grp_t), public :: grp !< mpi group for this system

contains
  ! both deferred as actually defined methods
  ! ...
end type system_t
The abstract class `system_t`:

```fortran
  type, extends(interaction_partner_t), abstract :: system_t

  private
  class(propagator_t), pointer, public :: prop => null()

  integer :: accumulated_loop_ticks

  integer, public :: interaction_timing !< parameter to determine if interactions

  type(integer_list_t), public :: supported_interactions
  type(interaction_list_t), public :: interactions !< List with all the interactions of this system

  type(mpi_grp_t), public :: grp !< mpi group for this system

contains
  ! both deferred as actually defined methods
  ! ...
end type system_t
```

- **abstract class: cannot be instantiated**
System classes

The abstract class system_t:

```plaintext
type, extends(interaction_partner_t), abstract :: system_t

private
  class(propagator_t), pointer, public :: prop => null()

  integer :: accumulated_loop_ticks
  integer, public :: interaction_timing !< parameter to determine if interactions

  type(integer_list_t), public :: supported_interactions
  type(interaction_list_t), public :: interactions !< List with all the interactions of this system

  type(mpi_grp_t), public :: grp !< mpi group for this system
contains
  ! both deferred as actually defined methods
  ! ...
end type system_t
```

- abstract class: cannot be instantiated
- inherits all from interaction_partner_t
System classes

The abstract class `system_t`:

```plaintext
type, extends(interaction_partner_t), abstract :: system_t

private
class(propagator_t), pointer, public :: prop => null()

integer :: accumulated_loop_ticks
integer, public :: interaction_timing !< parameter to determine if interactions

type(integer_list_t), public :: supported_interactions
type(interaction_list_t), public :: interactions !< List with all the interactions of this system

type(mpi_grp_t), public :: grp !< mpi group for this system
contains
! both deferred as actually defined methods
! ...
end type system_t
```

- abstract class: cannot be instantiated
- inherits all from `interaction_partner_t`
- defines basic variables and methods for all systems
System classes

The abstract class system_t:

type, extends(interaction_partner_t), abstract :: system_t

private
class(propagator_t), pointer, public :: prop => null()

integer :: accumulated_loop_ticks
integer, public :: interaction_timing !< parameter to determine if interactions

type(integer_list_t), public :: supported_interactions
type(interaction_list_t), public :: interactions !< List with all the interactions of this system

type(mpi_grp_t), public :: grp !< mpi group for this system contains
! both deferred as actually defined methods
! ...
end type system_t

- abstract class: cannot be instantiated
- inherits all from interaction_partner_t
- defines basic variables and methods for all systems
- implements methods which are common to all systems
System classes

The abstract class `system_t`:

```fortran
  type, extends(interaction_partner_t), abstract :: system_t

  private
    class(propagator_t), pointer, public :: prop => null()

    integer             :: accumulated_loop_ticks
    integer, public :: interaction_timing  !< parameter to determine if interactions

  type(integer_list_t), public :: supported_interactions
  type(interaction_list_t), public :: interactions  !< List with all the interactions of this system

  type(mpi_grp_t), public :: grp  !< mpi group for this system
  contains
    ! both deferred as actually defined methods
    ! ...
  end type system_t
```

- abstract class: cannot be instantiated
- inherits all from `interaction_partner_t`
- defines basic variables and methods for all systems
- implements methods which are common to all systems
- defines deferred methods which are common to all systems, but depend on specifics
System classes

The system_t methods:

- procedure :: dt_operation => system_dt_operation
- procedure :: reset_clocks => system_reset_clocks
- procedure :: update_exposed_quantities => system_update_exposed_quantities
- procedure :: init_propagator => system_init_propagator
- procedure :: init_all_interactions => system_init_all_interactions
- procedure :: init_parallelization => system_init_parallelization
- procedure :: update_interactions => system_update_interactions
- procedure :: update_interactions_start => system_update_interactions_start
- procedure :: update_interactions_finish => system_update_interactions_finish
- procedure :: propagation_start => system_propagation_start
- procedure :: propagation_finish => system_propagation_finish
- procedure :: has_reached_final_propagation_time => system_has_reached_final_propagation_time
- procedure :: output_start => system_output_start
- procedure :: output_write => system_output_write
- procedure :: output_finish => system_output_finish
- procedure :: process_is_slave => system_process_is_slave
- procedure :: exec_end_of_timestep_tasks => system_exec_end_of_timestep_tasks

- procedure(system_init_interaction), deferred :: init_interaction
- procedure(system_initial_conditions), deferred :: initial_conditions
- procedure(system_do_td_op), deferred :: do_td_operation
- procedure(system_iteration_info), deferred :: iteration_info
- procedure(system_is_tolerance_reached), deferred :: is_tolerance_reached
- procedure(system_update_quantity), deferred :: update_quantity
System classes

Child classes add more features to the parent class.

- deferred functions can be implemented
- functions of parent can be overridden
System classes

Child classes add more features to the parent class.
- deferred functions can be implemented
- functions of parent can be overridden

Performing a algorithmic step: dt_operation()
- perform general tasks
- call do_td_op() of child class.
Classical particles

classical_particles_t

- any number of classical particles
- described by array of 3-d vector for coordinates
Classicle particles

classical_particles_t

- any number of classical particles
- described by array of 3-d vector for coordinates

classical_particle_t

- specialized to one particle
Classicle particles

type, extends(system_t), abstract :: classical_particles_t

private
integer, public :: np !< Number of particles in the system
FLOAT, allocatable, public :: mass(:) !< Mass of the particles
FLOAT, allocatable, public :: pos(:,:) !< Position of the particles
FLOAT, allocatable, public :: vel(:,:) !< Velocity of the particles
FLOAT, allocatable, public :: tot_force(:,:) !< Total force acting on each particle
logical, allocatable, public :: fixed(:) !< True if a giving particle is to be kept fixed during a !< propagation. The default is to let the particles move.

!> The following variables are work arrays used by the different propagators:
FLOAT, allocatable :: acc(:,:) !< Acceleration of the particles
FLOAT, allocatable :: prev_acc(:,:,,:) !< A storage of the prior times.
FLOAT, allocatable :: save_pos(:,:) !< A storage for the SCF loops
FLOAT, allocatable :: save_vel(:,:) !< A storage for the SCF loops
FLOAT, allocatable :: prev_tot_force(:,:) !< Used for the SCF convergence criterium
FLOAT, allocatable :: prev_pos(:,:,:,:) !< Used for extrapolation
FLOAT, allocatable :: prev_vel(:,:,:,:) !< Used for extrapolation
FLOAT, allocatable :: hamiltonian_elements(:,:)

contains

procedure :: do_td_operation => classical_particles_do_td
procedure :: is_tolerance_reached => classical_particles_is_tolerance_reached
procedure :: copy_quantities_to_interaction => classical_particles_copy_quantities_to_interaction
procedure :: update_interactions_start => classical_particles_update_interactions_start
procedure :: update_interactions_finish => classical_particles_update_interactions_finish

end type classical_particles_t
type, extends(classical_particles_t) :: classical_particle_t

    type(c_ptr) :: output_handle

contains

  procedure :: init_interaction => classical_particle_init_interaction
  procedure :: initial_conditions => classical_particle_initial_conditions
  procedure :: iteration_info => classical_particle_iteration_info
  procedure :: output_start => classical_particle_output_start
  procedure :: output_write => classical_particle_output_write
  procedure :: output_finish => classical_particle_output_finish
  procedure :: update_quantity => classical_particle_update_quantity
  procedure :: update_exposed_quantity => classical_particle_update_exposed_quantity
  procedure :: init_interaction_as_partner => classical_particle_init_interaction_as_partner
  procedure :: copy_quantities_to_interaction => classical_particle_copy_quantities_to_interaction
final :: classical_particle_finalize

end type classical_particle_t
Interaction classes

Currently implemented interaction classes:

- potential_interaction_t: acting on electrons (in development)
- force_interaction_t: acting on classical particles
- linear_medium_em_field_t: acting on Maxwell fields
The abstract class interaction_t:

type, abstract :: interaction_t

private

!> The interaction requires access to some quantities from a system to be evaluated.

integer, public :: n_system_quantities !< Number of quantities needed from the system
integer, allocatable, public :: system_quantities(:) !< Identifiers of the quantities needed from the system
type(clock_t), public :: clock !< Clock storing the time at which the interaction was last updated.
character(len=:), public, allocatable :: label

contains

procedure(interaction_update), deferred :: update
procedure(interaction_calculate), deferred :: calculate

end type interaction_t
The abstract class `interaction_with_partner_t`:

Some interactions involve two systems. In this case the interaction is a unidirectional relationship between those two systems. One of the systems owns the interaction and feels its effects. The other system is referred to as the interaction partner.

type, extends(interaction_t), abstract :: interaction_with_partner_t

    private

    class(interaction_partner_t), public, pointer :: partner
    integer,   public :: n_partner_quantities  !< Number of quantities needed from the partner
    integer, allocatable, public :: partner_quantities(:) !< Identifiers of the quantities needed
                  !< from the partner

contains

    procedure :: update => interaction_with_partner_update

end type interaction_with_partner_t
Interaction classes

The abstract class `force_interaction_t`:

```fortran
  type, extends(interaction_with_partner_t), abstract :: force_interaction_t

  integer :: dim = 0 !< spatial dimensions
  integer :: system_np = 0 !< number of particles in the system that the forces are acting on

  FLOAT, allocatable, public :: force(:,:)
end type force_interaction_t
```
The class `gravity_t`:

`gravity_t` is a class for gravity interaction between two systems of particles. This should be used for testing purposes only. Note that this interaction assumes all quantities are in S.I. units instead of atomic units.

```fortran
type, extends(force_interaction_t) :: gravity_t
    private
    FLOAT, pointer :: system_mass(:) !< pointer to array storing the masses of the particles
    FLOAT, pointer :: system_pos(:,:) !< pointer to array storing the positions of the particles
    integer, public :: partner_np = 0 !< number of particles in the partner system
    FLOAT, allocatable, public :: partner_mass(:) !< array storing a copy of the masses of the partner particles
    FLOAT, allocatable, public :: partner_pos(:,:) !< array storing a copy of the positions of the partner particles
contains
    procedure :: init => gravity_init
    procedure :: calculate => gravity_calculate
    final :: gravity_finalize
end type gravity_t
```
Class hierarchy of propagators:

As propagators are derived from linked lists and algorithms, one can directly use their respective methods.
Propagator implementation

Defining a propagator:

```fortran
function propagator_verlet_constructor(dt) result(this)
    FLOAT, intent(in) :: dt
    type(propagator_verlet_t), pointer :: this

    PUSH_SUB(propagator_verlet_constructor)

    SAFE_ALLOCATE(this)

    this%start_step = OP_VERLET_START
    this%final_step = OP_VERLET_FINISH

    call this%add_operation(OP_VERLET_UPDATE_POS)
    call this%add_operation(OP_UPDATE_INTERACTIONS)
    call this%add_operation(OP_VERLET_COMPUTE_ACC)
    call this%add_operation(OP_VERLET_COMPUTE_VEL)
    call this%add_operation(OP_FINISHED)

    ! Verlet has only one algorithmic step
    this%algo_steps = 1

    this%dt = dt

    POP_SUB(propagator_verlet_constructor)
end function propagator_verlet_constructor
```
Propagator implementation

Defining a propagator:

! Specific verlet propagation operations identifiers
character(len=30), public, parameter :: &
  VERLET_START = 'VERLET_START', &
  VERLET_FINISH = 'VERLET_FINISH', &
  VERLET_UPDATE_POS = 'VERLET_UPDATE_POS', &
  VERLET_COMPUTE_ACC = 'VERLET_COMPUTE_ACC', &
  VERLET_COMPUTE_VEL = 'VERLET_COMPUTE_VEL'

! Specific verlet propagation operations
type(algorithmic_operation_t), public, parameter :: &
  OP_VERLET_START = algorithmic_operation_t(VERLET_START, 'Starting Verlet propagation'),
  OP_VERLET_FINISH = algorithmic_operation_t(VERLET_FINISH, 'Finishing Verlet propagation'),
  OP_VERLET_UPDATE_POS = algorithmic_operation_t(VERLET_UPDATE_POS, 'Propagation step - Updating positions'),
  OP_VERLET_COMPUTE_ACC = algorithmic_operation_t(VERLET_COMPUTE_ACC, 'Propagation step - Computing acceleration'),
  OP_VERLET_COMPUTE_VEL = algorithmic_operation_t(VERLET_COMPUTE_VEL, 'Propagation step - Computing velocity')

These are defined as module variables.
Propagator implementation

Implementing the steps: \texttt{system.t\%do_td\_operation()}

- Actual tasks depend on the specific system.
- the specific function is the same for all implemented algorithms
  $$\Rightarrow$$ implement operations for all implemented propagators
Implementing the steps: system_t%do_td_operation()

```fortran
subroutine classical_particles_do_td(this, operation)
  class(classical_particles_t), intent(inout) :: this
  class(algorithmic_operation_t), intent(in) :: operation
...
  select case (operation%id)
    case (SKIP)
      ! Do nothing
    case (STORE_CURRENT_STATUS)
      this%save_pos(:, 1:this%np) = this%pos(:, 1:this%np)
      this%save_vel(:, 1:this%np) = this%vel(:, 1:this%np)
    case (VERLET_FINISH)
    ...  
    case (BEEMAN_FINISH)
    ...
    case (VERLET_UPDATE_POS)
      this%pos(:, 1:this%np) = this%pos(:, 1:this%np) + this%prop%dt * this%vel(:, 1:this%np) &
        + M_HALF * this%prop%dt**2 * this%acc(:, 1:this%np)
      this%quantities(POSITION)%clock = this%quantities(POSITION)%clock + CLOCK_TICK
    ...
```

Propagator implementation