Octopus Multi-system Framework

Micael Oliveira

Fortran OOP Seminar, January 14, 2022
Motivation

- After 20 years of development, the old code structure was starting to show its limits
- New developments were becoming more difficult
- Previous attempts to go beyond the “simple” electrons+nuclei implementation were not very successful
- Fortran 2003 introduces lots of new OOP features

In 2019 it was decided to introduce a new framework and rewrite large portions of Octopus using Fortran OOP.
What problem are we trying to solve?
What problem are we trying to solve?

- We want to solve a system of **coupled** differential equations

Example: TDDFT coupled with classical nuclei

\[
i \frac{\partial}{\partial t} \varphi_i(r, t) = \left\{ v_{\text{ext}}(r; R) + v_H[n](r, t) + v_{\text{xc}}[n](r, t) \right\} \varphi_i(r, t) \]

\[
m_I \frac{\partial^2}{\partial t^2} R_I(t) = \sum_J F_{IJ}(R_I, R_J) + F_{Ie}(R_I; n) \]

- Couplings
What problem are we trying to solve?

- How to handle arbitrary numbers of equations?
- How to add/remove equations “on-the-fly”?
- How to activate/deactivate couplings “on-the-fly”? 
How to code this?

The way **NOT** to do it:

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

if (system_A%has_interaction_X_with_system_B) then
    ...
end if

if (system_B%has_interaction_X_with_system_A) then
    ...
end if

if ((system_A%has_interaction_Y_with_system_B) then
    ...
end if
```
Multi-system framework: Key features

- Allows to define many physical systems simultaneously (electrons, ions, lasers, Maxwell, DFTB+, PCM, etc)
- Systems are coupled through interactions (Electron-ion, Lorentz force, dipole coupling, etc)
- The code automatically handles all the interactions/systems
- New parallelization level over systems
- Flexible algorithms:
  - Time-propagation using different propagators and time-steps for each system
  - Nested SCF loops
  - ...
- Adding new systems, interactions and algorithms should be as simple as possible
Multi-System Framework: Design

- Focus on extendability and maintainability
- Heavy use of object-oriented programming
- Framework is independent of existing systems and interactions
- Systems do not know about each other directly, instead they only know interactions
- Calculations modes are now “algorithms”, which are coded as a set of state machine atomic operations
Test environment: celestial dynamics

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

inp

CalculationMode = td
ExperimentalFeatures = yes

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

%Interactions
gravity | all_partners
%
InteractionTiming = timing_retarded

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

Earth.ParticleMass = 5.97237e24
%Earth.ParticleInitialPosition
-147364661998.16476 | -24608859261.610123 | 1665165.2801353487
%
%Earth.ParticleInitialVelocity
4431.136612956525 | -29497.611635546345 | 0.343475566161544
%
Test environment: celestial dynamics

inp (cont.)

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

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

TDSystemPropagator = verlet

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

### Systems block

<table>
<thead>
<tr>
<th>%Systems</th>
<th>classical_particle</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Sun&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;Earth&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;Moon&quot;</td>
<td></td>
</tr>
</tbody>
</table>

### Nested systems

<table>
<thead>
<tr>
<th>%Systems</th>
<th>classical_particle</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Sun&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;Earth&quot;</td>
<td>multisystem</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>%Earth.Systems</th>
<th>classical_particle</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Terra&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;Luna&quot;</td>
<td></td>
</tr>
</tbody>
</table>
New multi-system syntax

**Namespaces**

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

**Interactions**

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

%SystemA.Interactions
  gravity   | no_partners
  coulomb_force | all_partners
%

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

1. Update positions

\[ \mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \mathbf{v}(t)\Delta t + \frac{1}{2}\mathbf{a}(t)\Delta t^2 \]

2. Update interactions with all partners (compute \( F(\mathbf{x}(t + \Delta t)) \))

3. Compute acceleration \( \mathbf{a}(t + \Delta t) \)

4. Compute velocity

\[ \mathbf{v}(t + \Delta t) = \mathbf{v}(t) + \frac{1}{2}(\mathbf{a}(t) + \mathbf{a}(t + \Delta t))\Delta t \]
Visualizing the multi-system time-stepping algorithm

https://octopus-code.org/new-site/develop/developers/code_documentation/propagators/custom_diagram/
Celestial orbits
System classes

- 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:

Rounded boxes: abstract class
Arrows indicate inheritance.
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
- Defines list of exposed quantities
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 !< should use the quantities at the exact time or if retardation is allowed

    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
    type(barrier_t) :: barrier(NUMBER_BARRIERS)
contains
    ...
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 :: restart_write => system_restart_write
- procedure :: restart_read => system_restart_read
- 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 :: start_barrier => system_start_barrier
- procedure :: end_barrier => system_end_barrier
- procedure :: arrived_at_barrier => system_arrived_at_barrier
- procedure :: arrived_at_any_barrier => system_arrived_at_any_barrier

- 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

Performing an algorithmic step: `dt_operation()`
- Perform general tasks
- Call `do_td_op()` of child class.
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
- Used for testing
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.

  type(propagator_data_t),public :: prop_data

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
  procedure :: restart_write_data => classical_particles_restart_write_data
  procedure :: restart_read_data => classical_particles_restart_read_data

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
Currently implemented interaction classes:
Interaction classes

The abstract class interaction_t:

type, abstract :: interaction_t
    private
        !> The interaction requires access to some quantities from a system to be evaluated.
        logical, public :: intra_interaction !< Is this an interaction of a system with itself?
        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
Interaction classes

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.

```fortran
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
```
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
```
Interaction classes

The class gravity_t:

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
module gravity
  private
  type, extends(force_interaction_t) :: gravity_t
    private
    type 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
    end type private
    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
end module gravity
```


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
class(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.
Implementing the steps: `system_t%do_td_operation()`

- Actual tasks depend on the specific system.
- The specific function is the same for all implemented algorithms
  - Implement operations for all implemented propagators
Propagator implementation

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(:, 1:1:np) = this%pos(:, 1:1:np)
        this%save_vel(:, 1:1:np) = this%vel(:, 1:1:np)
    case (VERLET_FINISH)
    ... case (BEEMAN_FINISH)
    ... case (VERLET_UPDATE_POS)
        this%pos(:, 1:1:np) = this%pos(:, 1:1:np) + this%prop%dt * this%vel(:, 1:1:np) &
        + M_HALF * this%prop%dt**2 * this%acc(:, 1:1:np)
        this%quantities(POSITION)%clock = this%quantities(POSITION)%clock + CLOCK_TICK
        ...