Octopus: Multisystem Example

Martin Lüders and the Octopus developers

Octopus Advanced Course 2023, MPSD Hamburg
Testing the framework: celestial dynamics

- System of planets and moons as point particles interacting with gravity
- Numerical integration of orbits with different algorithms
- Easy to validate results
- Fast turnover for code development
Testing the framework: the velocity Verlet propagator

1. Update positions

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

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

3. Compute acceleration \( a(t + \Delta t) \)

4. Compute velocity

   \[ v(t + \Delta t) = v(t) + \frac{1}{2}(a(t) + a(t + \Delta t))\Delta t \]
Testing the framework: the velocity Verlet propagator

! -----
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_STEP_DONE)
call this%add_operation(OP_REWIND_ALGORITHM)

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

this%dt = dt

POP_SUB(propagator_verlet_constructor)
end function propagator_verlet_constructor
Testing the framework: the velocity Verlet propagator

```fortran
logical function classical_particles_do_algorithmic_operation(this, operation, updated_quantities)
  class(classical_particles_t), intent(inout) :: this
  class(algorithmic_operation_t), intent(in) :: operation
  integer, allocatable, intent(out) :: updated_quantities(:)

  select case (operation % id)
  case (VERLET_START)
    if (.not. this % prop_data % initialized) then
      call this % prop_data % initialize(prop, this % space % dim, this % np)
      do ip = 1, this % np
        if (this % fixed(ip)) then
          this % prop_data % acc(:, ip) = M_ZERO
        else
          this % prop_data % acc(:, ip) = this % tot_force(:, ip) / this % mass(ip)
        end if
      end do
    end if
  case (VERLET_FINISH, BEEMAN_FINISH)
    call this % prop_data % end()
  case (VERLET_UPDATE_POS)
    this % pos(:, 1: this % np) = this % pos(:, 1: this % np) + prop % dt * this % vel(:, 1: this % np) &
    + M_HALF * prop % dt ** 2 * this % prop_data % acc(:, 1: this % np)
    updated_quantities = [POSITION]
  end select
end function classical_particles_do_algorithmic_operation
```
Testing the framework: a simple case

Input

# List of systems to simulate, giving each one a name and declaring what type of system it is
%Syste%sys (I am a block. Each line can have multiple columns. Columns are separated by a vertical bar)
"Sun"   | classical_particle
"Earth"  | classical_particle
"Moon"   | classical_particle
%

# Each system will interact through gravity with all possible partners
%Interactions
   gravity | all_partners
%

# We use velocity verlet for all systems
TDSyste%systemPropagator = verlet

# Time step and total simulation time
TDTimeStep = 3600   # one hour
TDPPropagationTime = 3*24*3600  # three days

# Next come the initial conditions, masses, etc
...
Testing the framework: a simple case

But how does this work in practice?
Visualizing the multi-system time-stepping algorithm

https://octopus-code.org/documentation/main/developers/code_documentation/propagators/custom_diagram/
Testing the framework: different time-steps

Input

# List of systems to simulate, giving each one a name and declaring what type of system it is
%Systems
"Sun" | classical_particle
"Earth" | classical_particle
"Moon" | classical_particle
%

# Each system will interact through gravity with all possible partners
%Interactions
  gravity | all_partners
%

# We use velocity verlet for all systems
TDSys템Propagator = verlet

# Time step and total simulation time
Sun.TDTimeStep = 3600  # one hour
Earth.TDTimeStep = 3600/2  # 30 minutes
Moon.TDTimeStep = 3600/4  # 15 minutes
TDPropagationTime = 3*24*3600  # three days

# We need to allow for the interactions to use information that is behind in time
# Another way to handle this would be to use interpolation
InteractionTiming = timing_retarded

# Next come the initial conditions, masses, etc
...
Testing the framework: multi-systems and nesting

Input

# Top-level list of systems
%Systems
"Sun" | classical_particle
"Earth" | multisystem # This is a system of systems
%

# Here we specify the systems contained in the "Earth" system
%Earth.Systems
"Terra" | classical_particle
"Luna" | classical_particle
%

# Each system will interact through gravity with all possible partners
%Interactions
gravity | all_partners
%

# We use velocity verlet for all systems
TDSYSTEMPROPAGATOR = verlet

# Time step and total simulation time
TDTimeStep = 3600 # one hour
TDPropagationTime = 3*24*3600 # three days

# Next come the initial conditions, masses, etc...
...
Testing the framework: different algorithms

Input

# Top-level list of systems
%Systems
"Sun" | classical_particle
"Earth" | multisystem # This is a system of systems
%

# Here we specify the systems contained in the "Earth" system
%Earth.Systems
"Terra" | classical_particle
"Luna" | classical_particle
%

# Each system will interact through gravity with all possible partners
%Interactions
  gravity | all_partners
%

# Use exponential-midpoint for Sun (NB: this propagator requires the evaluation of the forces at dt/2 and at dt)
Sun.TDSystemPropagator = exp_mid_2step
Earth.TDSystemPropagator = verlet

# Time step and total simulation time
Sum.TDTimeStep = 3600 # one hour
Earth.TDTimeStep = 3600/2 # 30 minutes

TDPropagationTime = 3*24*3600 # three days

# Next come the initial conditions, masses, etc.
Examples

Classical dynamics class diagram

Octopus: Multisystem Example