CAAM 419/519, Homework #2

`hc54`

September 30, 2022

1 Verification of the correctness of the two methods

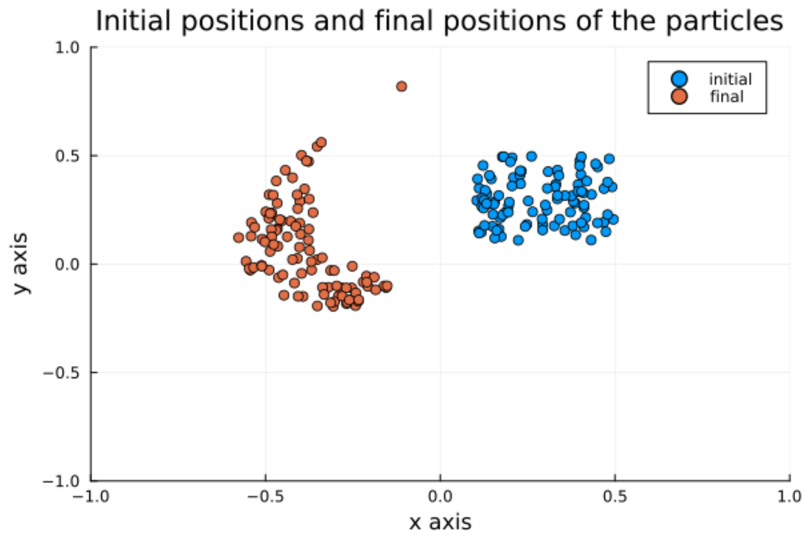


Figure 1: Positions of the particles

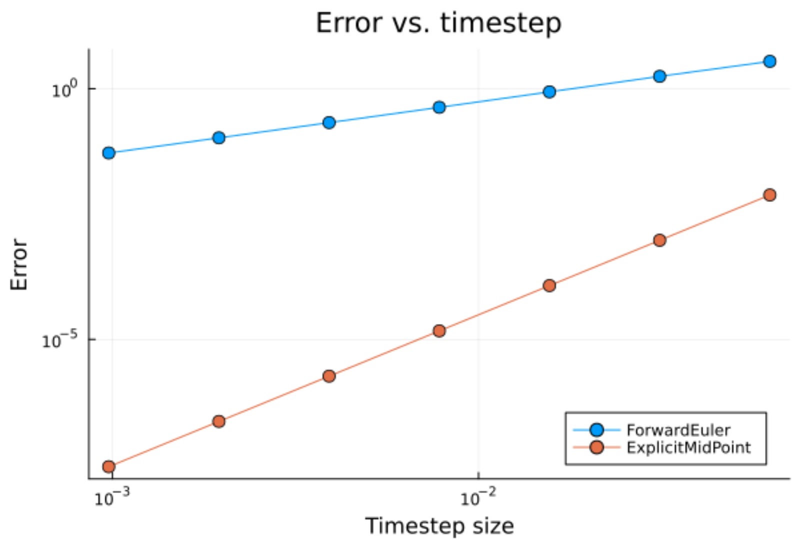
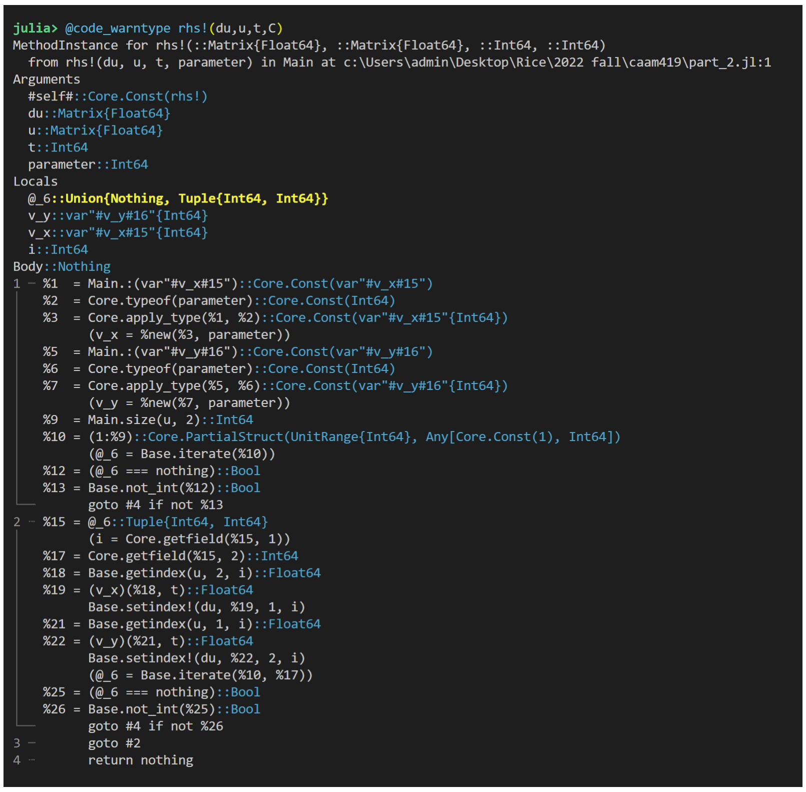


Figure 2: Error vs dt

Explicit Midpoint Method performs better than Forward Euler Method. As timestep size decreases, the error of Explicit Midpoint Method decreases faster than Forward Euler Method and both error converges to 0. In addition, the error created by Explicit Midpoint Method is always less than the one created by Forward Euler among the seven tested timestep sizes.

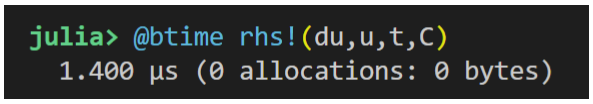
2 Efficiency of rhs! function



```
julia> @code_warntype rhs!(du,u,t,C)
MethodInstance for rhs!(::Matrix{Float64}, ::Matrix{Float64}, ::Int64, ::Int64)
  from rhs!(du, u, t, parameter) in Main at c:\Users\admin\Desktop\Rice\2022 fall\caam419\part_2.jl:1
Arguments
  #self#::Core.Const(rhs!)
  du::Matrix{Float64}
  u::Matrix{Float64}
  t::Int64
  parameter::Int64
Locals
  @_6::Union{Nothing, Tuple{Int64, Int64}}
  v_y::var"#v_y#16"{Int64}
  v_x::var"#v_x#15"{Int64}
  i::Int64
Body::Nothing
1 ─ %1  = Main.:(var"#v_x#15")::Core.Const(var"#v_x#15")
│   %2  = Core.typeof(parameter)::Core.Const(Int64)
│   %3  = Core.apply_type(%1, %2)::Core.Const(var"#v_x#15"{Int64})
│         (v_x = %new(%3, parameter))
│   %5  = Main.:(var"#v_y#16")::Core.Const(var"#v_y#16")
│   %6  = Core.typeof(parameter)::Core.Const(Int64)
│   %7  = Core.apply_type(%5, %6)::Core.Const(var"#v_y#16"{Int64})
│         (v_y = %new(%7, parameter))
│   %9  = Main.size(u, 2)::Int64
│   %10 = (1:%9)::Core.PartialStruct(UnitRange{Int64}, Any[Core.Const(1), Int64])
│         (@_6 = Base.iterate(%10))
│   %12 = (@_6 === nothing)::Bool
│   %13 = Base.not_int(%12)::Bool
└──       goto #4 if not %13
2 ┄ %15 = @_6::Tuple{Int64, Int64}
│         (i = Core.getfield(%15, 1))
│   %17 = Core.getfield(%15, 2)::Int64
│   %18 = Base.getindex(u, 2, i)::Float64
│   %19 = (v_x)(%18, t)::Float64
│         Base.setindex!(du, %19, 1, i)
│   %21 = Base.getindex(u, 1, i)::Float64
│   %22 = (v_y)(%21, t)::Float64
│         Base.setindex!(du, %22, 2, i)
│         (@_6 = Base.iterate(%10, %17))
│   %25 = (@_6 === nothing)::Bool
│   %26 = Base.not_int(%25)::Bool
└──       goto #4 if not %26
3 ─       goto #2
4 ┄       return nothing
```

Figure 3: Type stability

According to the output above, this function is type stable.



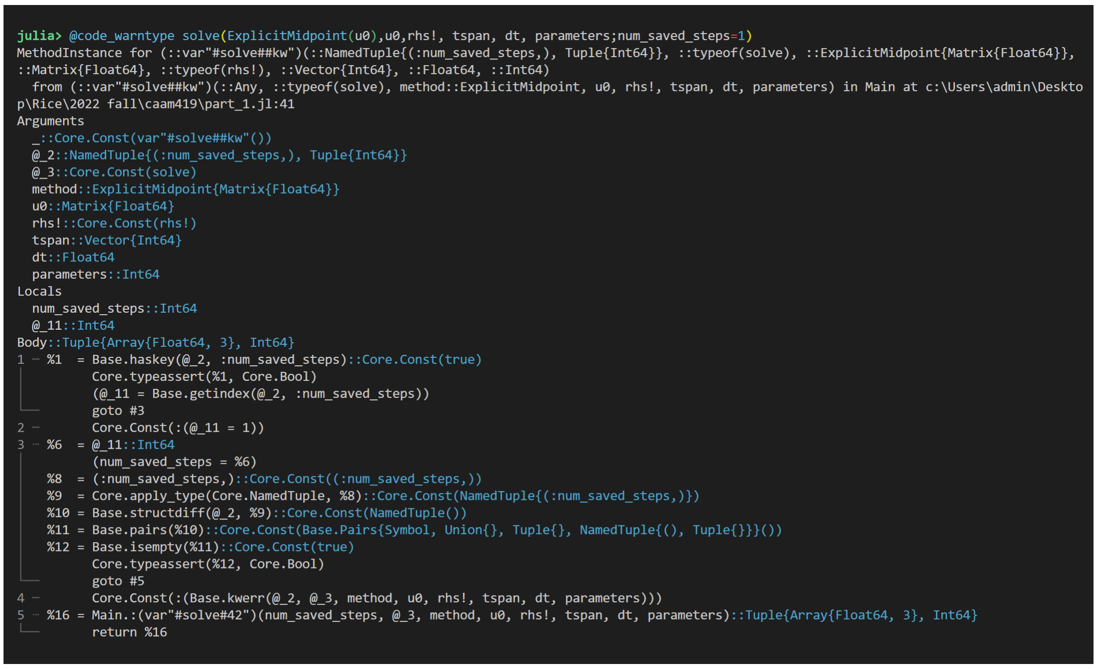
```
julia> @btime rhs!(du,u,t,C)
  1.400 μs (0 allocations: 0 bytes)
```

Figure 4: Allocation

According to the output above, this function is allocation-free.

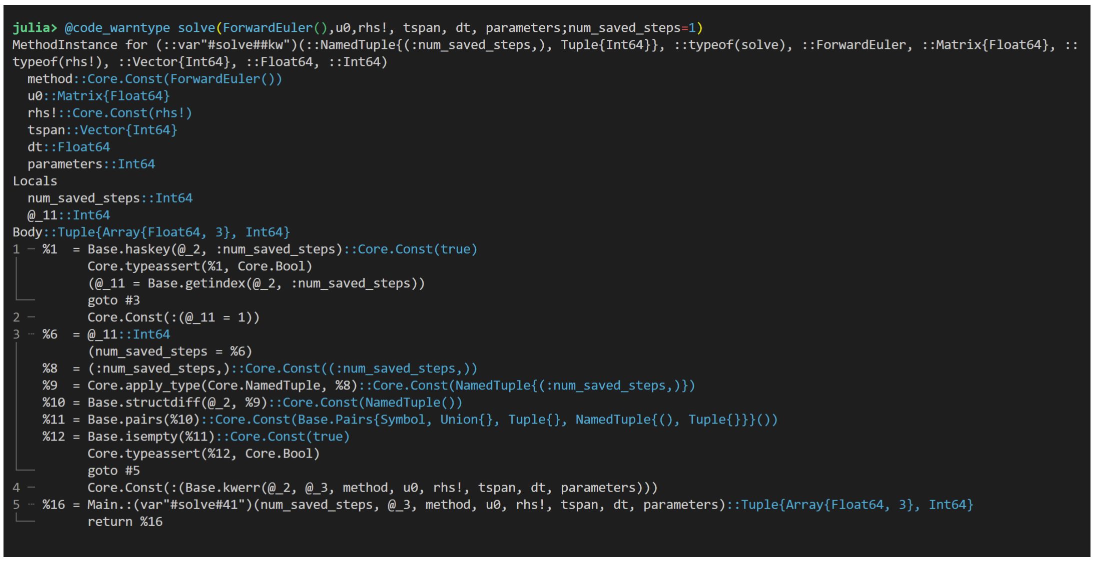
3 Efficiency of solver function

According to the outputs, the solver is type stable.



```
julia> @code_warntype solve(ExplicitMidpoint(u0),u0,rhs!, tspan, dt, parameters;num_saved_steps=1)
MethodInstance for (::var"#solve##kw")(::NamedTuple{(:num_saved_steps,), Tuple{Int64}}, ::typeof(solve), ::ExplicitMidpoint{Matrix{Float64}},
::Matrix{Float64}, ::typeof(rhs!), ::Vector{Int64}, ::Float64, ::Int64)
  from (::var"#solve##kw")(::Any, ::typeof(solve), method::ExplicitMidpoint, u0, rhs!, tspan, dt, parameters) in Main at c:\Users\admin\Deskto
p\Rice\2022 fall\caam419\part_1.jl:41
Arguments
  _::Core.Const(var"#solve##kw"())
  @_2::NamedTuple{(:num_saved_steps,), Tuple{Int64}}
  @_3::Core.Const(solve)
  method::ExplicitMidpoint{Matrix{Float64}}
  u0::Matrix{Float64}
  rhs!::Core.Const(rhs!)
  tspan::Vector{Int64}
  dt::Float64
  parameters::Int64
Locals
  num_saved_steps::Int64
  @_11::Int64
Body::Tuple{Array{Float64, 3}, Int64}
1 ─ %1  = Base.haskey(@_2, :num_saved_steps)::Core.Const(true)
│         Core.typeassert(%1, Core.Bool)
│         (@_11 = Base.getindex(@_2, :num_saved_steps))
└──       goto #3
2 ─       Core.Const(:(@_11 = 1))
3 ┄ %6  = @_11::Int64
│         (num_saved_steps = %6)
│   %8  = (:num_saved_steps,)::Core.Const((:num_saved_steps,))
│   %9  = Core.apply_type(Core.NamedTuple, %8)::Core.Const(NamedTuple{(:num_saved_steps,)})
│   %10 = Base.structdiff(@_2, %9)::Core.Const(NamedTuple())
│   %11 = Base.pairs(%10)::Core.Const(Base.Pairs{Symbol, Union{}, Tuple{}, NamedTuple{(), Tuple{}}}())
│   %12 = Base.isempty(%11)::Core.Const(true)
│         Core.typeassert(%12, Core.Bool)
└──       goto #5
4 ─       Core.Const(:(Base.kwerr(@_2, @_3, method, u0, rhs!, tspan, dt, parameters)))
5 ┄ %16 = Main.:(var"#solve#42")(num_saved_steps, @_3, method, u0, rhs!, tspan, dt, parameters)::Tuple{Array{Float64, 3}, Int64}
└──       return %16
```

Figure 5: Type stability for solver



```
julia> @code_warntype solve(ForwardEuler(),u0,rhs!, tspan, dt, parameters;num_saved_steps=1)
MethodInstance for (::var"#solve##kw")(::NamedTuple{(:num_saved_steps,), Tuple{Int64}}, ::typeof(solve), ::ForwardEuler, ::Matrix{Float64}, ::
typeof(rhs!), ::Vector{Int64}, ::Float64, ::Int64)
  method::Core.Const(ForwardEuler())
  u0::Matrix{Float64}
  rhs!::Core.Const(rhs!)
  tspan::Vector{Int64}
  dt::Float64
  parameters::Int64
Locals
  num_saved_steps::Int64
  @_11::Int64
Body::Tuple{Array{Float64, 3}, Int64}
1 ─ %1  = Base.haskey(@_2, :num_saved_steps)::Core.Const(true)
│         Core.typeassert(%1, Core.Bool)
│         (@_11 = Base.getindex(@_2, :num_saved_steps))
└──       goto #3
2 ─       Core.Const(:(@_11 = 1))
3 ┄ %6  = @_11::Int64
│         (num_saved_steps = %6)
│   %8  = (:num_saved_steps,)::Core.Const((:num_saved_steps,))
│   %9  = Core.apply_type(Core.NamedTuple, %8)::Core.Const(NamedTuple{(:num_saved_steps,)})
│   %10 = Base.structdiff(@_2, %9)::Core.Const(NamedTuple())
│   %11 = Base.pairs(%10)::Core.Const(Base.Pairs{Symbol, Union{}, Tuple{}, NamedTuple{(), Tuple{}}}())
│   %12 = Base.isempty(%11)::Core.Const(true)
│         Core.typeassert(%12, Core.Bool)
└──       goto #5
4 ─       Core.Const(:(Base.kwerr(@_2, @_3, method, u0, rhs!, tspan, dt, parameters)))
5 ┄ %16 = Main.:(var"#solve#41")(num_saved_steps, @_3, method, u0, rhs!, tspan, dt, parameters)::Tuple{Array{Float64, 3}, Int64}
└──       return %16
```

Figure 6: Type stability for solver with another method

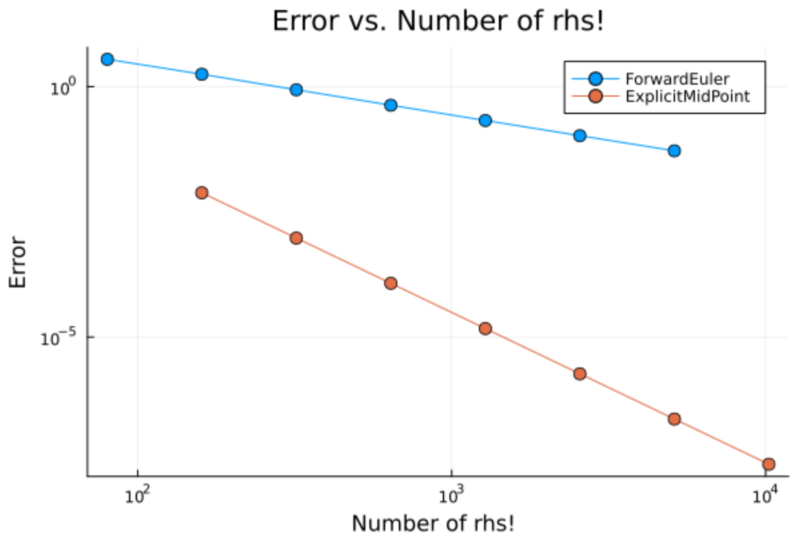


Figure 7: Error vs number of rhs! evaluations