Package 'rODE'

May 28, 2017

Type Package

Title Ordinary Differential Equation (ODE) Solvers Written in R Using S4 Classes

Version 0.99.4

Author Alfonso R. Reyes <alfonso.reyes@oilgainsanalytics.com>

Maintainer Alfonso R. Reyes <alfonso.reyes@oilgainsanalytics.com>

Description Show Physics and engineering students how an ODE solver is made and how effective classes can be for the construction of the equations that describe how effective classes can be for the construction of equations that describe the natural phenomena. Most of the ideas come from the book on ``Computer Simulations in Physics'' by Harvey Gould, Jan Tobochnik, and Wolfgang Christian. Book link: <http://www.compadre.org/osp/items/detail.cfm?ID=7375>.

Depends R (>= 3.3.0)

License GPL-2

Encoding UTF-8

Imports methods, data.table

LazyData true

Suggests knitr, testthat, rmarkdown, ggplot2, dplyr, tidyr

RoxygenNote 6.0.1

Collate 'ode_generics.R' 'ODESolver.R' 'ODE.R' 'AbstractODESolver.R' 'ODEAdaptiveSolver.R' 'DormandPrince45.R' 'Euler.R' 'EulerRichardson.R' 'RK4.R' 'RK45.R' 'Verlet.R' 'rODE-package.r' 'utils.R'

VignetteBuilder knitr

NeedsCompilation no

Repository CRAN

Date/Publication 2017-05-28 06:20:06 UTC

R topics documented:

`AbstractODESolver` *AbstractODESolver class*

Description

Defines the basic methods for all the ODE solvers.

Usage

```
AbstractODESolver(ode, ...)
```

Arguments

ode	an ODE object
...	additional parameters

Details

Inherits from: ODESolver class

Examples

```
# This is how we start defining a new ODE solver: Euler
.Euler <- setClass("Euler",              # Euler solver very simple; no slots
     contains = c("AbstractODESolver"))



# Here we define the ODE solver Verlet
.Verlet <- setClass("Verlet", slots = c(
    rate1 = "numeric",                          # Verlet calculates two rates
    rate2 = "numeric",
    rateCounter = "numeric"),
contains = c("AbstractODESolver"))



# This is the definition of the ODE solver Runge-Kutta 4
.RK4 <- setClass("RK4", slots = c(       # On the other hand RK4 uses 4 rates
   rate1 = "numeric",
   rate2 = "numeric",
   rate3 = "numeric",
   rate4 = "numeric",
   estimated_state = "numeric"),         # and estimates another state
contains = c("AbstractODESolver"))
```

AbstractODESolver,missing-method
AbstractODESolver constructor 'missing"

Description

AbstractODESolver constructor 'missing"

Usage

```
## S4 method for signature 'missing'
AbstractODESolver(ode, ...)
```

Arguments

ode	an ODE object
...	additional parameters

`AbstractODESolver,ODE-method` *AbstractODESolver constructor 'ODE'*

Description

Uses this constructor when ODE object is passed

Usage

```
## S4 method for signature 'ODE'
AbstractODESolver(ode, ...)
```

Arguments

ode	an ODE object
...	additional parameters

`DormandPrince45` *DormandPrince45 constructor*

Description

DormandPrince45 constructor

Usage

```
DormandPrince45(.ode)
```

Arguments

.ode	an ODE object

Examples

```
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  base class: KeplerVerlet.R

setClass("Kepler", slots = c(
    GM = "numeric",
    odeSolver = "DormandPrince45",
    counter = "numeric"
    ),
    contains = c("ODE")
)

setMethod("initialize", "Kepler", function(.Object, ...) {
    .Object@GM <- 4 * pi * pi         # gravitation constant times combined mass
    .Object@state <- vector("numeric", 5)  # x, vx, y, vy, t
    .Object@odeSolver <- DormandPrince45(.Object)
    .Object@counter <- 0
    return(.Object)
})

setMethod("doStep", "Kepler", function(object, ...) {
    object@odeSolver <- step(object@odeSolver)
    object@state <- object@odeSolver@ode@state
    object
})

setMethod("getTime", "Kepler", function(object, ...) {
    return(object@state[5])
})

setMethod("getEnergy", "Kepler", function(object, ...) {
    ke <- 0.5 * (object@state[2] * object@state[2] +
                     object@state[4] * object@state[4])
    pe <- -object@GM / sqrt(object@state[1] * object@state[1] +
                                object@state[3] * object@state[3])
    return(pe+ke)
})

setMethod("init", "Kepler", function(object, initState, ...) {
    object@state <- initState
    object@odeSolver <- init(object@odeSolver, getStepSize(object@odeSolver))
    object@counter <- 0
    object
})

setMethod("getRate", "Kepler", function(object, state, ...) {
    # Computes the rate using the given state.
    r2 <- state[1] * state[1] + state[3] * state[3]  # distance squared
    r3 <- r2 * sqrt(r2)   # distance cubed
    object@rate[1] <- state[2]
    object@rate[2] <- (- object@GM * state[1]) / r3
    object@rate[3] <- state[4]
    object@rate[4] <- (- object@GM * state[3]) / r3
```

```
    object@rate[5] <- 1   # time derivative

    object@counter <- object@counter + 1
    object@rate

})

setMethod("getState", "Kepler", function(object, ...) {
    # Gets the state variables.
    return(object@state)
})

# constructor
Kepler <- function() {
    kepler <- new("Kepler")
    return(kepler)
}
```

---

doStep *Perform a step*

Description

Perform a step

Usage

```
doStep(object, ...)
```

Arguments

object	a class object
...	additional parameters

Examples

```
# ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++  example: PlanetApp.R
# Simulation of Earth orbiting around the SUn using the Euler ODE solver

importFromExamples("Planet.R")      # source the class

PlanetApp <- function(verbose = FALSE) {
    # x =  1, AU or Astronomical Units. Length of semimajor axis or the orbit
    # of the Earth around the Sun.
    x <- 1; vx <- 0; y <- 0; vy <- 6.28; t <- 0
    state <- c(x, vx, y, vy, t)
    dt <-  0.01
    planet <- Planet()
    planet@odeSolver <- setStepSize(planet@odeSolver, dt)
    planet <- init(planet, initState = state)
```

```
    rowvec <- vector("list")
    i <- 1
    # run infinite loop. stop with ESCAPE.
    while (planet@state[5] <= 90) {     # Earth orbit is 365 days around the sun
        rowvec[[i]] <- list(t  = planet@state[5],     # just doing 3 months
                            x  = planet@state[1],     # to speed up for CRAN
                            vx = planet@state[2],
                            y  = planet@state[3],
                            vy = planet@state[4])
        for (j in 1:5) {                # advances time
            planet <- doStep(planet)
        }
        i <- i + 1
    }
    DT <- data.table::rbindlist(rowvec)
    return(DT)
}
# run the application
solution <- PlanetApp()
select_rows <- seq(1, nrow(solution), 10)      # do not overplot
solution <- solution[select_rows,]
plot(solution)

# +++++++++++++++++++++++++++++++++++++++++++++++++   application:  Logistic.R
# Simulates the logistic equation

setClass("Logistic", slots = c(
    K = "numeric",
    r = "numeric",
    odeSolver = "Verlet",
    counter = "numeric"
),
contains = c("ODE")
)

setMethod("initialize", "Logistic", function(.Object, ...) {
    .Object@K <- 10
    .Object@r <- 1.0
    .Object@state <- vector("numeric", 3)  # x, vx
    .Object@odeSolver <- Verlet(.Object)
    .Object@counter <- 0
    return(.Object)
})

setMethod("doStep", "Logistic", function(object, ...) {
    # cat("state@doStep=", object@state, "\n")
    object@odeSolver <- step(object@odeSolver)
    object@state <- object@odeSolver@ode@state
    object
})

setMethod("getTime", "Logistic", function(object, ...) {
    return(object@state[3])
```

```
})


setMethod("init", "Logistic", function(object, initState, r, K, ...) {
    object@r <- r
    object@K <- K
    object@state <- initState
    object@odeSolver <- init(object@odeSolver, getStepSize(object@odeSolver))
    object@counter <- 0
    object
})


setMethod("getRate", "Logistic", function(object, state, ...) {
    # Computes the rate using the given state.
    object@rate[1] <- state[2]
    object@rate[2] <- object@r * state[1] * (1 - state[1] / object@K)
    object@rate[3] <- 1   # time derivative
    object@counter <- object@counter + 1
    object@rate

})


setMethod("getState", "Logistic", function(object, ...) {
    # Gets the state variables.
    return(object@state)
})

# constructor
Logistic <- function() {
    logistic <- new("Logistic")
    return(logistic)
}

# Run the application
LogisticVerletApp <- function(verbose = FALSE) {
    x  <- 0.1
    vx <- 0
    r  <- 2        # Malthusian parameter (rate of maximum population growth)
    K  <- 10.0     # carrying capacity of the environment
    dt   <- 0.01; tol  <- 1e-3; tmax <- 10
    population <- Logistic()
    population <- init(population, c(x, vx, 0), r, K)
    odeSolver <- Verlet(population)
    odeSolver <- init(odeSolver, dt)
    population@odeSolver <- odeSolver
    rowVector <- vector("list")
    i <- 1
    while (getTime(population) <= tmax) {
        rowVector[[i]] <- list(t = getTime(population),
                               s1 = population@state[1],
                               s2 = population@state[2])
```

```
        population <- doStep(population)
        i <- i + 1
    }
    DT <- data.table::rbindlist(rowVector)
    return(DT)
}
# show solution
solution <- LogisticVerletApp()
plot(solution)
```

---

enableRuntimeExceptions *Enable Runtime Exceptions*

Description

Enable Runtime Exceptions

DormandPrince45 class

Usage

```
enableRuntimeExceptions(object, enable, ...)

## S4 method for signature 'DormandPrince45'
enableRuntimeExceptions(object, enable)
```

Arguments

object	a class object
enable	a logical flag
...	additional parameters

Examples

```
setMethod("enableRuntimeExceptions", "DormandPrince45", function(object, enable) {
    object@enableExceptions <- enable
})
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  base class: KeplerVerlet.R

setClass("Kepler", slots = c(
    GM = "numeric",
    odeSolver = "DormandPrince45",
    counter = "numeric"
    ),
    contains = c("ODE")
)

setMethod("initialize", "Kepler", function(.Object, ...) {
```

```
    .Object@GM <- 4 * pi * pi         # gravitation constant times combined mass
    .Object@state <- vector("numeric", 5)  # x, vx, y, vy, t
    .Object@odeSolver <- DormandPrince45(.Object)
    .Object@counter <- 0
    return(.Object)
})

setMethod("doStep", "Kepler", function(object, ...) {
    object@odeSolver <- step(object@odeSolver)
    object@state <- object@odeSolver@ode@state
    object
})

setMethod("getTime", "Kepler", function(object, ...) {
    return(object@state[5])
})

setMethod("getEnergy", "Kepler", function(object, ...) {
    ke <- 0.5 * (object@state[2] * object@state[2] +
                     object@state[4] * object@state[4])
    pe <- -object@GM / sqrt(object@state[1] * object@state[1] +
                                object@state[3] * object@state[3])
    return(pe+ke)
})

setMethod("init", "Kepler", function(object, initState, ...) {
    object@state <- initState
    object@odeSolver <- init(object@odeSolver, getStepSize(object@odeSolver))
    object@counter <- 0
    object
})

setMethod("getRate", "Kepler", function(object, state, ...) {
    # Computes the rate using the given state.
    r2 <- state[1] * state[1] + state[3] * state[3]  # distance squared
    r3 <- r2 * sqrt(r2)   # distance cubed
    object@rate[1] <- state[2]
    object@rate[2] <- (- object@GM * state[1]) / r3
    object@rate[3] <- state[4]
    object@rate[4] <- (- object@GM * state[3]) / r3
    object@rate[5] <- 1   # time derivative

    object@counter <- object@counter + 1
    object@rate

})

setMethod("getState", "Kepler", function(object, ...) {
    # Gets the state variables.
    return(object@state)
})

# constructor
```

```
Kepler <- function() {
    kepler <- new("Kepler")
    return(kepler)
}
```

`Euler` *Euler class*

Description

Euler class

Euler constructor when 'ODE' passed

Euler constructor 'missing' is passed

Usage

```
Euler(ode, ...)

## S4 method for signature 'ODE'
Euler(ode, ...)

## S4 method for signature 'missing'
Euler(ode, ...)
```

Arguments

ode	an ODE object
...	additional parameters

Examples

```
# ++++++++++++++++++++++++++++++++++++++++++++++ application: RigidBodyNXFApp.R
# example of a nonstiff system is the system of equations describing
# the motion of a rigid body without external forces.

setClass("RigidBodyNXF", slots = c(
    g = "numeric"
),
prototype = prototype(
    g = 9.8
),
contains = c("ODE")
)


setMethod("initialize", "RigidBodyNXF", function(.Object, ...) {
    .Object@state <- vector("numeric", 5)
    return(.Object)
```

```
})


setMethod("getState", "RigidBodyNXF", function(object, ...) {
    # Gets the state variables.
    return(object@state)
})


setMethod("getRate", "RigidBodyNXF", function(object, state, ...) {
    # Gets the rate of change using the argument's state variables.
    object@rate[1] <- state[2] * state[3]
    object@rate[2] <- - state[1] * state[3]
    object@rate[3] <- -0.51 * state[1] * state[2]
    object@rate[4] <- 1

    object@rate
})


# constructor
RigidBodyNXF <- function(y1, y2, y3) {
    .RigidBodyNXF <- new("RigidBodyNXF")
    .RigidBodyNXF@state[1] <- y1
    .RigidBodyNXF@state[2] <- y2
    .RigidBodyNXF@state[3] <- y3
    .RigidBodyNXF@state[4] <- 0
    .RigidBodyNXF
}

# run the application
RigidBodyNXFApp <- function(verbose = FALSE) {
    # load the R class that sets up the solver for this application
    y1 <- 0   # initial y1 value
    y2 <- 1    # initial y2 value
    y3 <- 1    # initial y3 value
    dt        <- 0.01 # delta time for step

    body <- RigidBodyNXF(y1, y2, y3)
    solver <- Euler(body)
    solver <- setStepSize(solver, dt)
    rowVector <- vector("list")
    i <- 1
    # stop loop when the body hits the ground
    while (body@state[4] <= 12) {
        rowVector[[i]] <- list(t  = body@state[4],
                               y1 = body@state[1],
                               y2 = body@state[2],
                               y3 = body@state[3])
        solver <- step(solver)
        body <- solver@ode
        i <- i + 1
    }
```

```
    DT <- data.table::rbindlist(rowVector)
    return(DT)
}

# get the data table from the app
solution <- RigidBodyNXFApp()
plot(solution)

# ++++++++++++++++++++++++++++++++++++++++++++  example: FallingParticleApp.R
# Application that simulates the free fall of a ball using Euler ODE solver

importFromExamples("FallingParticleODE.R")      # source the class

FallingParticleODEApp <- function(verbose = FALSE) {
    # initial values
    initial_y <- 10
    initial_v <- 0
    dt <- 0.01
    ball <- FallingParticleODE(initial_y, initial_v)
    solver <- Euler(ball)                        # set the ODE solver
    solver <- setStepSize(solver, dt)            # set the step
    rowVector <- vector("list")
    i <- 1
    # stop loop when the ball hits the ground, state[1] is the vertical position
    while (ball@state[1] > 0) {
        rowVector[[i]] <- list(t  = ball@state[3],
                               y  = ball@state[1],
                               vy = ball@state[2])
        solver <- step(solver)                   # move one step at a time
        ball <- solver@ode                       # update the ball state
        i <- i + 1
    }
    DT <- data.table::rbindlist(rowVector)
    return(DT)
}
# show solution
solution <- FallingParticleODEApp()
plot(solution)
# KeplerVerlet.R



setClass("Kepler", slots = c(
    GM = "numeric",
    odeSolver = "Euler",
    counter = "numeric"
    ),
    contains = c("ODE")
)

setMethod("initialize", "Kepler", function(.Object, ...) {
    .Object@GM <- 4 * pi * pi                # gravitation constant times combined mass
    .Object@state <- vector("numeric", 5)  # x, vx, y, vy, t
```

```
    .Object@odeSolver <- Euler(.Object)
    .Object@counter <- 0
    return(.Object)
})

setMethod("doStep", "Kepler", function(object, ...) {
    # cat("state@doStep=", object@state, "\n")
    object@odeSolver <- step(object@odeSolver)

    object@state <- object@odeSolver@ode@state

    # object@rate <- object@odeSolver@ode@rate
    # cat("\t", object@odeSolver@ode@state)
    object
})

setMethod("getTime", "Kepler", function(object, ...) {
    return(object@state[5])
})

setMethod("getEnergy", "Kepler", function(object, ...) {
    ke <- 0.5 * (object@state[2] * object@state[2] +
                     object@state[4] * object@state[4])
    pe <- -object@GM / sqrt(object@state[1] * object@state[1] +
                                object@state[3] * object@state[3])
    return(pe+ke)
})

setMethod("init", "Kepler", function(object, initState, ...) {
    object@state <- initState
    object@odeSolver <- init(object@odeSolver, getStepSize(object@odeSolver))

    # object@rate  <- object@odeSolver@ode@rate
    # object@state <- object@odeSolver@ode@state

    object@counter <- 0
    object
})

setMethod("getRate", "Kepler", function(object, state, ...) {
    # Computes the rate using the given state.
    r2 <- state[1] * state[1] + state[3] * state[3]  # distance squared
    r3 <- r2 * sqrt(r2)   # distance cubed
    object@rate[1] <- state[2]
    object@rate[2] <- (- object@GM * state[1]) / r3
    object@rate[3] <- state[4]
    object@rate[4] <- (- object@GM * state[3]) / r3
    object@rate[5] <- 1   # time derivative

    # object@state <- object@odeSolver@ode@state <- state
    # object@state <- state
    object@counter <- object@counter + 1
    object@rate
```

```
})

setMethod("getState", "Kepler", function(object, ...) {
    # Gets the state variables.
    return(object@state)
})

# constructor
Kepler <- function() {
    kepler <- new("Kepler")
    return(kepler)
}
# +++++++++++++++++++++++++++++++++++++++++++++++++++++++++  example: PlanetApp.R
# Simulation of Earth orbiting around the SUn using the Euler ODE solver

importFromExamples("Planet.R")      # source the class

PlanetApp <- function(verbose = FALSE) {
    # x =  1, AU or Astronomical Units. Length of semimajor axis or the orbit
    # of the Earth around the Sun.
    x <- 1; vx <- 0; y <- 0; vy <- 6.28; t <- 0
    state <- c(x, vx, y, vy, t)
    dt <-  0.01
    planet <- Planet()
    planet@odeSolver <- setStepSize(planet@odeSolver, dt)
    planet <- init(planet, initState = state)
    rowvec <- vector("list")
    i <- 1
    # run infinite loop. stop with ESCAPE.
    while (planet@state[5] <= 90) {     # Earth orbit is 365 days around the sun
        rowvec[[i]] <- list(t  = planet@state[5],     # just doing 3 months
                            x  = planet@state[1],     # to speed up for CRAN
                            vx = planet@state[2],
                            y  = planet@state[3],
                            vy = planet@state[4])
        for (j in 1:5) {                # advances time
            planet <- doStep(planet)
        }
        i <- i + 1
    }
    DT <- data.table::rbindlist(rowvec)
    return(DT)
}
# run the application
solution <- PlanetApp()
select_rows <- seq(1, nrow(solution), 10)      # do not overplot
solution <- solution[select_rows,]
plot(solution)

# +++++++++++++++++++++++++++++++++++++++++++++++ application: RigidBodyNXFApp.R
# example of a nonstiff system is the system of equations describing
# the motion of a rigid body without external forces.
```

```
setClass("RigidBodyNXF", slots = c(
    g = "numeric"
),
prototype = prototype(
    g = 9.8
),
contains = c("ODE")
)


setMethod("initialize", "RigidBodyNXF", function(.Object, ...) {
    .Object@state <- vector("numeric", 5)
    return(.Object)
})


setMethod("getState", "RigidBodyNXF", function(object, ...) {
    # Gets the state variables.
    return(object@state)
})


setMethod("getRate", "RigidBodyNXF", function(object, state, ...) {
    # Gets the rate of change using the argument's state variables.
    object@rate[1] <- state[2] * state[3]
    object@rate[2] <- - state[1] * state[3]
    object@rate[3] <- -0.51 * state[1] * state[2]
    object@rate[4] <- 1

    object@rate
})


# constructor
RigidBodyNXF <- function(y1, y2, y3) {
    .RigidBodyNXF <- new("RigidBodyNXF")
    .RigidBodyNXF@state[1] <- y1
    .RigidBodyNXF@state[2] <- y2
    .RigidBodyNXF@state[3] <- y3
    .RigidBodyNXF@state[4] <- 0
    .RigidBodyNXF
}

# run the application
RigidBodyNXFApp <- function(verbose = FALSE) {
    # load the R class that sets up the solver for this application
    y1 <- 0   # initial y1 value
    y2 <- 1    # initial y2 value
    y3 <- 1    # initial y3 value
    dt        <- 0.01 # delta time for step

    body <- RigidBodyNXF(y1, y2, y3)
```

```
    solver <- Euler(body)
    solver <- setStepSize(solver, dt)
    rowVector <- vector("list")
    i <- 1
    # stop loop when the body hits the ground
    while (body@state[4] <= 12) {
        rowVector[[i]] <- list(t  = body@state[4],
                               y1 = body@state[1],
                               y2 = body@state[2],
                               y3 = body@state[3])
        solver <- step(solver)
        body <- solver@ode
        i <- i + 1
    }
    DT <- data.table::rbindlist(rowVector)
    return(DT)
}

# get the data table from the app
solution <- RigidBodyNXFApp()
plot(solution)
```

`EulerRichardson` *EulerRichardson class*

Description

EulerRichardson class

Usage

```
EulerRichardson(ode, ...)
```

Arguments

ode	an ODE object
...	additional parameters

Examples

```
# ++++++++++++++++++++++++++++++++++++++++++++++++      example: PendulumApp.R
# Simulation of a pendulum using the EulerRichardson ODE solver

suppressPackageStartupMessages(library(ggplot2))

importFromExamples("Pendulum.R")      # source the class

PendulumApp <- function(verbose = FALSE) {
    # initial values
```

```
    theta <- 0.2
    thetaDot <- 0
    dt <- 0.1
    ode <- new("ODE")
    pendulum <- Pendulum()
    pendulum@state[3] <- 0      # set time to zero, t = 0
    pendulum <- setState(pendulum, theta, thetaDot)
    pendulum <- setStepSize(pendulum, dt = dt) # using stepSize in RK4
    pendulum@odeSolver <- setStepSize(pendulum@odeSolver, dt) # set new step size
    rowvec <- vector("list")
    i <- 1
    while (pendulum@state[3] <= 40)    {
        rowvec[[i]] <- list(t  = pendulum@state[3],    # time
                            theta = pendulum@state[1], # angle
                            thetadot = pendulum@state[2]) # derivative of angle
        pendulum <- step(pendulum)
        i <- i + 1
    }
    DT <- data.table::rbindlist(rowvec)
    return(DT)
}
# show solution
solution <- PendulumApp()
plot(solution)
```

`EulerRichardson,ODE-method`

EulerRichardson constructor ODE

Description

EulerRichardson constructor ODE

Usage

```
## S4 method for signature 'ODE'
EulerRichardson(ode, ...)
```

Arguments

ode	an ODE object
...	additional parameters

getEnergy *Get the calculated energy level*

Description

Get the calculated energy level

Usage

```
getEnergy(object, ...)
```

Arguments

object	a class object
...	additional parameters

Examples

```
# KeplerEnergy.R
#

setClass("KeplerEnergy", slots = c(
    GM        = "numeric",
    odeSolver = "Verlet",
    counter   = "numeric"
    ),
    contains = c("ODE")
)

setMethod("initialize", "KeplerEnergy", function(.Object, ...) {
    .Object@GM <- 4 * pi * pi         # gravitation constant times combined mass
    .Object@state <- vector("numeric", 5)  # x, vx, y, vy, t
    # .Object@odeSolver <- Verlet(ode = .Object)
    .Object@odeSolver <- Verlet(.Object)
    .Object@counter <- 0
    return(.Object)
})

setMethod("doStep", "KeplerEnergy", function(object, ...) {
    object@odeSolver <- step(object@odeSolver)
    object@state <- object@odeSolver@ode@state
    object
})

setMethod("getTime", "KeplerEnergy", function(object, ...) {
    return(object@state[5])
})

setMethod("getEnergy", "KeplerEnergy", function(object, ...) {
```

```
    ke <- 0.5 * (object@state[2] * object@state[2] +
                     object@state[4] * object@state[4])
    pe <- -object@GM / sqrt(object@state[1] * object@state[1] +
                                object@state[3] * object@state[3])
    return(pe+ke)
})


setMethod("init", "KeplerEnergy", function(object, initState, ...) {
    object@state <- initState
    object@odeSolver <- init(object@odeSolver, getStepSize(object@odeSolver))
    object@counter <- 0
    object
})


setMethod("getRate", "KeplerEnergy", function(object, state, ...) {
    # Computes the rate using the given state.
    r2 <- state[1] * state[1] + state[3] * state[3]  # distance squared
    r3 <- r2 * sqrt(r2)   # distance cubed
    object@rate[1] <- state[2]
    object@rate[2] <- (- object@GM * state[1]) / r3
    object@rate[3] <- state[4]
    object@rate[4] <- (- object@GM * state[3]) / r3
    object@rate[5] <- 1   # time derivative

    object@counter <- object@counter + 1
    object@rate

})


setMethod("getState", "KeplerEnergy", function(object, ...) {
    # Gets the state variables.
    return(object@state)
})

# constructor
KeplerEnergy <- function() {
    kepler <- new("KeplerEnergy")
    return(kepler)
}
```

`getErrorCode` *Get an error code*

Description

Get an error code

Usage

```
getErrorCode(object, tol, ...)

## S4 method for signature 'ODEAdaptiveSolver'
getErrorCode(object)

## S4 method for signature 'DormandPrince45'
getErrorCode(object)
```

Arguments

object	a class object
tol	tolerance
...	additional parameters

See Also

Other adaptive solver methods: getTolerance, setTolerance

Examples

```
setMethod("getErrorCode", "DormandPrince45", function(object) {
return(object@error_code)
})
```

---

getExactSolution *Compare analytical and calculated solutions*

Description

Compare analytical and calculated solutions

Usage

```
getExactSolution(object, t, ...)
```

Arguments

object	a class object
t	time ath what we are performing the evaluation
...	additional parameters

Examples

```
# ++++++++++++++++++++++++++++++++++++++++++++++  example: ComparisonRK45App.R
# Compares the solution by the RK45 ODE solver versus the analytical solution
# Example file: ComparisonRK45App.R
# ODE Solver:   Runge-Kutta 45
# Class:        RK45

importFromExamples("ODETest.R")

 ComparisonRK45App <- function(verbose = FALSE) {
     ode <- new("ODETest")                    # create an `ODETest` object
     ode_solver <- RK45(ode)                  # select the ODE solver
     ode_solver <- setStepSize(ode_solver, 1)      # set the step
     ode_solver <- setTolerance(ode_solver, 1e-8)  # set the tolerance
     time <-  0
     rowVector <- vector("list")
     i <- 1
     while (time < 50) {
         rowVector[[i]] <- list(t  = ode_solver@ode@state[2],
                                s1 = getState(ode_solver@ode)[1],
                                s2 = getState(ode_solver@ode)[2],
                                xs = getExactSolution(ode_solver@ode, time),
                                rc = getRateCounts(ode),
                                time = time)
         ode_solver <- step(ode_solver)       # advance one step
         stepSize <-  ode_solver@stepSize     # update the step size
         time <- time + stepSize
         state <- getState(ode_solver@ode)    # get the `state` vector
         i <- i + 1
     }
     DT <- data.table::rbindlist(rowVector)    # a data table with the results
     return(DT)
 }
# show solution
solution <- ComparisonRK45App()                          # run the example
plot(solution)
# ODETest.R
# Base class for examples:
#                         ComparisonRK45App.R
#                         ComparisonRK45ODEApp.R

#' ODETest as an example of ODE class inheritance
#'
#' ODETest is a base class for examples ComparisonRK45App.R and
#' ComparisonRK45ODEApp.R
#'
#' @rdname ODE-class-example
#' @include ODE.R
setClass("ODETest", slots = c(
    n     = "numeric",           # counts the number of getRate evaluations
    stack = "environment"        # environnment to keep stack
    ),
```

```
    contains = c("ODE")
    )


setMethod("initialize", "ODETest", function(.Object, ...) {
    .Object@stack$rateCounts <-  0              # counter for rate calculations
    .Object@state <- c(5.0, 0.0)
    return(.Object)
})

#' @rdname getExactSolution-method
setMethod("getExactSolution", "ODETest", function(object, t, ...) {
    return(5.0 * exp(-t))
})

#' @rdname getState-method
setMethod("getState", "ODETest", function(object, ...) {
    object@state
})

#' @rdname getRate-method
setMethod("getRate", "ODETest", function(object, state, ...) {
    object@rate[1] <- - state[1]
    object@rate[2] <-  1            # rate of change of time, dt/dt
    # accumulate how many the rate has been called to calculate
    object@stack$rateCounts <- object@stack$rateCounts + 1          # use stack
    object@state <- state
    object@rate
})


#' @rdname getRateCounts-method
setMethod("getRateCounts", "ODETest", function(object, ...) {
    # use environment stack to accumulate rate counts
    object@stack$rateCounts
})


# constructor
ODETest <- function() {
    odetest <- new("ODETest")
    odetest
}
```

---

getRate *Get a new rate given a state*

Description

Get a new rate given a state

Usage

```
getRate(object, state, ...)

## S4 method for signature 'ODE'
getRate(object, state, ...)
```

Arguments

object	a class object
state	current state
...	additional parameters

Examples

```
#  Kepler models Keplerian orbits of a mass moving under the influence of an
#  inverse square force by implementing the ODE interface.
#  Kepler.R
#

setClass("Kepler", slots = c(
    GM = "numeric"
    ),
    contains = c("ODE")
)

setMethod("initialize", "Kepler", function(.Object, ...) {
    .Object@GM <- 1.0                # gravitation constant times combined mass
    .Object@state <- vector("numeric", 5)  # x, vx, y, vy, t
    return(.Object)
})

setMethod("getState", "Kepler", function(object, ...) {
    # Gets the state variables.
    return(object@state)
})

setMethod("getRate", "Kepler", function(object, state, ...) {
    # Computes the rate using the given state.
    r2 <- state[1] * state[1] + state[3] * state[3]  # distance squared
    r3 <- r2 * sqrt(r2)   # distance cubed
    object@rate[1] <- state[2]
    object@rate[2] <- (- object@GM * state[1]) / r3
    object@rate[3] <- state[4]
    object@rate[4] <- (- object@GM * state[3]) / r3
    object@rate[5] <- 1   # time derivative

    object@rate

})

# constructor
```

```
Kepler <- function(r, v) {
    kepler <- new("Kepler")
    kepler@state[1] = r[1]
    kepler@state[2] = v[1]
    kepler@state[3] = r[2]
    kepler@state[4] = v[2]
    kepler@state[5] = 0
    return(kepler)
}
#  ++++++++++++++++++++++++++++++++++++++++++++++++++++++++ example KeplerApp.R
#  KeplerApp solves an inverse-square law model (Kepler model) using an adaptive
#  stepsize algorithm.
#  Application showing two planet orbiting
#  File in examples: KeplerApp.R

importFromExamples("Kepler.R") # source the class Kepler

KeplerApp <- function(verbose = FALSE) {

    # set the orbit into a predefined state.
    r <- c(2, 0)                                    # orbit radius
    v <- c(0, 0.25)                                 # velocity
    dt <- 0.1

    planet <- Kepler(r, v)
    solver <- RK45(planet)

    rowVector <- vector("list")
    i <- 1
    while (planet@state[5] <= 10) {
        rowVector[[i]] <- list(t  = planet@state[5],
                               planet1.r = planet@state[1],
                               p1anet1.v = planet@state[2],
                               planet2.r = planet@state[3],
                               p1anet2.v = planet@state[4])
        solver <- step(solver)
        planet <- solver@ode
        i <-  i + 1
    }
    DT <- data.table::rbindlist(rowVector)

    return(DT)
}


solution <- KeplerApp()
plot(solution)


# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ base class: FallingParticleODE.R
# Class definition for application FallingParticleODEApp.R

setClass("FallingParticleODE", slots = c(
```

```
        g = "numeric"
        ),
        prototype = prototype(
            g = 9.8
        ),
        contains = c("ODE")
        )


setMethod("initialize", "FallingParticleODE", function(.Object, ...) {
    .Object@state <- vector("numeric", 3)
    return(.Object)
})

setMethod("getState", "FallingParticleODE", function(object, ...) {
    # Gets the state variables.
    return(object@state)
})

setMethod("getRate", "FallingParticleODE", function(object, state, ...) {
    # Gets the rate of change using the argument's state variables.
    object@rate[1] <- state[2]
    object@rate[2] <- - object@g
    object@rate[3] <- 1

    object@rate
})

# constructor
FallingParticleODE <- function(y, v) {
    .FallingParticleODE <- new("FallingParticleODE")
    .FallingParticleODE@state[1] <- y
    .FallingParticleODE@state[2] <- v
    .FallingParticleODE@state[3] <- 0
    .FallingParticleODE
}
```

`getRateCounts` *Get the number of times that the rate has been calculated*

Description

Get the number of times that the rate has been calculated

Usage

```
getRateCounts(object, ...)
```

Arguments

object	a class object
...	additional parameters

`getState` *Get current state of the system*

Description

Get current state of the system

Usage

```
getState(object, ...)

## S4 method for signature 'ODE'
getState(object, ...)
```

Arguments

object	a class object
...	additional parameters

Examples

```
# ++++++++++++++++++++++++++++++++++++++++++++++++++      example: VanderPol.R
# Solution of the Van der Pol equation
#


setClass("VanderPol", slots = c(
    mu = "numeric"
),
contains = c("ODE")
)

setMethod("initialize", "VanderPol", function(.Object, ...) {
    .Object@mu <- 1.0                # gravitation constant times combined mass
    .Object@state <- vector("numeric", 3)  # y1, y2, t
    return(.Object)
})


setMethod("getState", "VanderPol", function(object, ...) {
    # Gets the state variables.
    return(object@state)
})
```

```
setMethod("getRate", "VanderPol", function(object, state, ...) {
    # Computes the rate using the given state.
    object@rate[1] <- state[2]
    object@rate[2] <-  object@mu* (1 - state[1]^2) * state[2] - state[1]
    object@rate[3] <- 1

    object@rate
})

# constructor
VanderPol <- function(y1, y2) {
    VanderPol <- new("VanderPol")
    VanderPol@state[1] = y1
    VanderPol@state[2] = y2
    VanderPol@state[3] = 0
    return(VanderPol)
}

# run the application
VanderPolApp <- function(verbose = FALSE) {
    # set the orbit into a predefined state.
    y1 <- 2; y2 <- 0; dt <- 0.1;
    rigid_body <- VanderPol(y1, y2)
    solver <- RK45(rigid_body)
    rowVector <- vector("list")
    i <- 1
    while (rigid_body@state[3] <= 20) {
        rowVector[[i]] <- list(t =  rigid_body@state[3],
                               y1 = rigid_body@state[1],
                               y2 = rigid_body@state[2])
        solver <- step(solver)
        rigid_body <- solver@ode
        i <-  i + 1
    }
    DT <- data.table::rbindlist(rowVector)
    return(DT)

}
# show solution
solution <- VanderPolApp()
plot(solution)

# ++++++++++++++++++++++++++++++++++++++++++++++++   application:  SpringRK4.R
# Simulation of a spring considering no friction
library(rODE)


setClass("SpringRK4", slots = c(
    # we should improve this by letting the user entered these values
    K         = "numeric",
    mu        = "numeric",
    mass      = "numeric",
```

```
        state     = "numeric",
        odeSolver = "RK4"
        ),
        prototype = prototype(
            K = 1,
            state = c(0, 0, 0)
        ),
        contains = c("ODE")
        )

    setMethod("initialize", "SpringRK4", function(.Object) {
        # we should improve this by letting the user entered these values
        .Object@K    <- 1.0
        .Object@mu   <- 1.5
        .Object@mass <- 20
        .Object@odeSolver <- RK4(.Object)
        return(.Object)
    })

    setMethod("setStepSize", signature("SpringRK4"), function(object, dt, ...) {
        # use explicit parameter declaration
        # setStepSize generic may use two different step parameters: stepSize and dt
        object@odeSolver <- setStepSize(object@odeSolver, dt)
        object
    })


    setMethod("step", "SpringRK4", function(object) {
        object@odeSolver <- step(object@odeSolver)
        object@rate  <- object@odeSolver@ode@rate
        object@state <- object@odeSolver@ode@state
        object
    })

    setMethod("setState", "SpringRK4", function(object, theta, thetaDot) {
        object@state[1] <- theta     # angle
        object@state[2] <- thetaDot  # derivative of the angle
        #                              state[3] is time
        object@odeSolver@ode@state <- object@state      # set state on solver
        object
    })

    setMethod("getState", "SpringRK4", function(object) {
        object@state
    })

    setMethod("getRate", "SpringRK4", function(object, state, ...) {
        # enter the derivatives here
        object@rate[1] <- state[2]     # rate of change of angle
        object@rate[2] <- -object@mu /  object@mass * state[2] - object@K * state[1]
        object@rate[3] <- 1            # rate of change of time, dt/dt

        object@rate
```

```
})

# constructor
SpringRK4 <- function()  new("SpringRK4")


# run application
SpringRK4App <- function(verbose = FALSE) {
    theta    <- 0
    thetaDot <- -0.2
    tmax     <- 22; dt <- 0.1
    ode <- new("ODE")
    spring <- SpringRK4()
    spring@state[3] <- 0      # set time to zero, t = 0
    spring <- setState(spring, theta, thetaDot)
    spring <- setStepSize(spring, dt = dt) # using stepSize in RK4
    spring@odeSolver <- setStepSize(spring@odeSolver, dt) # set new step size
    rowvec <- vector("list")
    i <- 1
    while (spring@state[3] <= tmax)    {
        rowvec[[i]] <- list(t  = spring@state[3],      # angle
                            y1 = spring@state[1],      # derivative of the angle
                            y2 = spring@state[2])      # time
        i <- i + 1
        spring <- step(spring)
    }
    DT <- data.table::rbindlist(rowvec)
    return(DT)
}

# show solution
solution <- SpringRK4App(TRUE)
plot(solution)
```

`getStepSize` *Get the step size*

Description

Get the step size

Usage

```
getStepSize(object, ...)

## S4 method for signature 'ODESolver'
getStepSize(object, ...)

## S4 method for signature 'AbstractODESolver'
getStepSize(object, ...)
```

```
## S4 method for signature 'DormandPrince45'
getStepSize(object, ...)
```

Arguments

object	a class object
...	additional parameters

Examples

```
# ++++++++++++++++++++++++++++++++++++++++ Example:      ComparisonRK45ODEApp.R
# Updates the ODE state instead of using the internal state in the ODE solver
# Also plots the solver solution versus the analytical solution at a
# tolerance of 1e-6
# ODE Solver:   Runge-Kutta 45
# Class:        RK45

importFromExamples("ODETest.R")

ComparisonRK45ODEApp <- function(verbose = FALSE) {
    ode <- new("ODETest")                          # new ODE instance
    ode_solver <- RK45(ode)                        # select ODE solver
    ode_solver <- setStepSize(ode_solver, 1)       # set the step
    ode_solver <- setTolerance(ode_solver, 1e-6)  # set the tolerance
    time <-  0
    rowVector <- vector("list")                    # row vector
    i <- 1    # counter
    while (time < 50) {
        # add solution objects to a row vector
        rowVector[[i]] <- list(t = ode_solver@ode@state[2],
                               ODE = getState(ode_solver@ode)[1],
                               s2  = getState(ode_solver@ode)[2],
                               exact = getExactSolution(ode_solver@ode, time),
                               rate.counts = getRateCounts(ode),
                               time = time )
        ode_solver <- step(ode_solver)             # advance solver one step
        stepSize <-  getStepSize(ode_solver)       # get the current step size
        time <- time + stepSize
        ode <- ode_solver@ode                      # get updated ODE object
        state <- getState(ode)                     # get the `state` vector
        i <- i + 1                                 # add a row vector
    }
    DT <- data.table::rbindlist(rowVector)         # create data table
    return(DT)
}

solution <- ComparisonRK45ODEApp()
plot(solution)
library(ggplot2)
library(dplyr)
library(tidyr)
```

```
solution.multi <- solution %>%
    select(t, ODE, exact)
plot(solution.multi)
solution.2x1 <- solution.multi %>%
    gather(key, value, -t)
g <- ggplot(solution.2x1, mapping = aes(x = t, y = value, color = key))
g <-  g + geom_line(size = 1) + labs(title = "ODE vs Exact solution",
                                     subtitle = "tolerance = 1E-6")
print(g)
```

---

getTime *Get the elapsed time*

Description

Get the elapsed time

Usage

```
getTime(object, ...)
```

Arguments

object	a class object
...	additional parameters

Examples

```
# ++++++++++++++++++++++++++++++++++++++++++++++++++   application:  Logistic.R
# Simulates the logistic equation

setClass("Logistic", slots = c(
    K = "numeric",
    r = "numeric",
    odeSolver = "Verlet",
    counter = "numeric"
),
contains = c("ODE")
)

setMethod("initialize", "Logistic", function(.Object, ...) {
    .Object@K <- 10
    .Object@r <- 1.0
    .Object@state <- vector("numeric", 3)  # x, vx
    .Object@odeSolver <- Verlet(.Object)
    .Object@counter <- 0
```

```
    return(.Object)
})

setMethod("doStep", "Logistic", function(object, ...) {
    # cat("state@doStep=", object@state, "\n")
    object@odeSolver <- step(object@odeSolver)
    object@state <- object@odeSolver@ode@state
    object
})

setMethod("getTime", "Logistic", function(object, ...) {
    return(object@state[3])
})


setMethod("init", "Logistic", function(object, initState, r, K, ...) {
    object@r <- r
    object@K <- K
    object@state <- initState
    object@odeSolver <- init(object@odeSolver, getStepSize(object@odeSolver))
    object@counter <- 0
    object
})


setMethod("getRate", "Logistic", function(object, state, ...) {
    # Computes the rate using the given state.
    object@rate[1] <- state[2]
    object@rate[2] <- object@r * state[1] * (1 - state[1] / object@K)
    object@rate[3] <- 1   # time derivative
    object@counter <- object@counter + 1
    object@rate

})


setMethod("getState", "Logistic", function(object, ...) {
    # Gets the state variables.
    return(object@state)
})

# constructor
Logistic <- function() {
    logistic <- new("Logistic")
    return(logistic)
}

# Run the application
LogisticVerletApp <- function(verbose = FALSE) {
    x  <- 0.1
    vx <- 0
    r  <- 2        # Malthusian parameter (rate of maximum population growth)
    K  <- 10.0     # carrying capacity of the environment
```

```
    dt   <- 0.01; tol  <- 1e-3; tmax <- 10
    population <- Logistic()
    population <- init(population, c(x, vx, 0), r, K)
    odeSolver <- Verlet(population)
    odeSolver <- init(odeSolver, dt)
    population@odeSolver <- odeSolver
    rowVector <- vector("list")
    i <- 1
    while (getTime(population) <= tmax) {
        rowVector[[i]] <- list(t = getTime(population),
                               s1 = population@state[1],
                               s2 = population@state[2])
        population <- doStep(population)
        i <- i + 1
    }
    DT <- data.table::rbindlist(rowVector)
    return(DT)
}
# show solution
solution <- LogisticVerletApp()
plot(solution)
# KeplerEnergy.R
#

setClass("KeplerEnergy", slots = c(
    GM        = "numeric",
    odeSolver = "Verlet",
    counter   = "numeric"
    ),
    contains = c("ODE")
)

setMethod("initialize", "KeplerEnergy", function(.Object, ...) {
    .Object@GM <- 4 * pi * pi         # gravitation constant times combined mass
    .Object@state <- vector("numeric", 5)  # x, vx, y, vy, t
    # .Object@odeSolver <- Verlet(ode = .Object)
    .Object@odeSolver <- Verlet(.Object)
    .Object@counter <- 0
    return(.Object)
})

setMethod("doStep", "KeplerEnergy", function(object, ...) {
    object@odeSolver <- step(object@odeSolver)
    object@state <- object@odeSolver@ode@state
    object
})

setMethod("getTime", "KeplerEnergy", function(object, ...) {
    return(object@state[5])
})

setMethod("getEnergy", "KeplerEnergy", function(object, ...) {
    ke <- 0.5 * (object@state[2] * object@state[2] +
```

```
                    object@state[4] * object@state[4])
    pe <- -object@GM / sqrt(object@state[1] * object@state[1] +
                                object@state[3] * object@state[3])
    return(pe+ke)
})


setMethod("init", "KeplerEnergy", function(object, initState, ...) {
    object@state <- initState
    object@odeSolver <- init(object@odeSolver, getStepSize(object@odeSolver))
    object@counter <- 0
    object
})


setMethod("getRate", "KeplerEnergy", function(object, state, ...) {
    # Computes the rate using the given state.
    r2 <- state[1] * state[1] + state[3] * state[3]  # distance squared
    r3 <- r2 * sqrt(r2)   # distance cubed
    object@rate[1] <- state[2]
    object@rate[2] <- (- object@GM * state[1]) / r3
    object@rate[3] <- state[4]
    object@rate[4] <- (- object@GM * state[3]) / r3
    object@rate[5] <- 1   # time derivative

    object@counter <- object@counter + 1
    object@rate

})


setMethod("getState", "KeplerEnergy", function(object, ...) {
    # Gets the state variables.
    return(object@state)
})

# constructor
KeplerEnergy <- function() {
    kepler <- new("KeplerEnergy")
    return(kepler)
}
```

`getTolerance` *Get the tolerance for the solver*

Description

Get the tolerance for the solver

Usage

```
getTolerance(object, ...)

## S4 method for signature 'ODEAdaptiveSolver'
getTolerance(object)

## S4 method for signature 'DormandPrince45'
getTolerance(object)
```

Arguments

object	a class object
...	additional parameters

See Also

Other adaptive solver methods: getErrorCode, setTolerance

---

importFromExamples *Source the R script*

Description

Source the R script

Usage

```
importFromExamples(aClassFile, aFolder = "examples")
```

Arguments

aClassFile	a file containing one or more classes
aFolder	a folder where examples are located

`init` *Set initial values before starting the ODE solver*

Description

Not all super classes require an init method.

Set initial values and get ready to start the solver

Usage

```
init(object, ...)

## S4 method for signature 'ODESolver'
init(object, stepSize, ...)

## S4 method for signature 'AbstractODESolver'
init(object, stepSize, ...)

## S4 method for signature 'DormandPrince45'
init(object, stepSize, ...)

## S4 method for signature 'Euler'
init(object, stepSize, ...)

## S4 method for signature 'EulerRichardson'
init(object, stepSize, ...)

## S4 method for signature 'RK4'
init(object, stepSize, ...)

## S4 method for signature 'Verlet'
init(object, stepSize, ...)
```

Arguments

`object`	a class object
`...`	additional parameters
`stepSize`	size of the step

Examples

```
# init method in Kepler.R
setMethod("init", "Kepler", function(object, initState, ...) {
    object@state <- initState
    object@odeSolver <- init(object@odeSolver, getStepSize(object@odeSolver))
    object@counter <- 0
    object
```

```
})

# init method in Logistic.R
setMethod("init", "Logistic", function(object, initState, r, K, ...) {
    object@r <- r
    object@K <- K
    object@state <- initState
    object@odeSolver <- init(object@odeSolver, getStepSize(object@odeSolver))
    object@counter <- 0
    object
})

# init method in Planet.R
setMethod("init", "Planet", function(object, initState, ...) {
    object@state <- object@odeSolver@ode@state <- initState
    # initialize providing the step size
    object@odeSolver <- init(object@odeSolver, getStepSize(object@odeSolver))
    object@rate <- object@odeSolver@ode@rate
    object@state <- object@odeSolver@ode@state
    object
})
```

ODE-class *Defines an ODE object for any solver*

Description

Defines an ODE object for any solver

Examples

```
# ++++++++++++++++++++++++++++++++++++++++++++++++++      example: PendulumApp.R
# Simulation of a pendulum using the EulerRichardson ODE solver

suppressPackageStartupMessages(library(ggplot2))

importFromExamples("Pendulum.R")      # source the class

PendulumApp <- function(verbose = FALSE) {
    # initial values
    theta <- 0.2
    thetaDot <- 0
    dt <- 0.1
    ode <- new("ODE")
    pendulum <- Pendulum()
    pendulum@state[3] <- 0      # set time to zero, t = 0
    pendulum <- setState(pendulum, theta, thetaDot)
    pendulum <- setStepSize(pendulum, dt = dt) # using stepSize in RK4
    pendulum@odeSolver <- setStepSize(pendulum@odeSolver, dt) # set new step size
    rowvec <- vector("list")
```

```
    i <- 1
    while (pendulum@state[3] <= 40)    {
        rowvec[[i]] <- list(t  = pendulum@state[3],    # time
                            theta = pendulum@state[1], # angle
                            thetadot = pendulum@state[2]) # derivative of angle
        pendulum <- step(pendulum)
        i <- i + 1
    }
    DT <- data.table::rbindlist(rowvec)
    return(DT)
}
# show solution
solution <- PendulumApp()
plot(solution)
# ++++++++++++++++++++++++++++++++++++++++++++++++  example: PendulumEulerApp.R
# Pendulum simulation with the Euler ODE solver
# Notice how Euler is not applicable in this case as it diverges very quickly
# even when it is using a very small `delta t``?ODE

importFromExamples("PendulumEuler.R")      # source the class

PendulumEulerApp <- function(verbose = FALSE) {
    # initial values
    theta <- 0.2
    thetaDot <- 0
    dt <- 0.01
    ode <- new("ODE")
    pendulum <- PendulumEuler()
    pendulum@state[3] <- 0      # set time to zero, t = 0
    pendulum <- setState(pendulum, theta, thetaDot)
    stepSize <- dt
    pendulum <- setStepSize(pendulum, stepSize)
    pendulum@odeSolver <- setStepSize(pendulum@odeSolver, dt) # set new step size
    rowvec <- vector("list")
    i <- 1
    while (pendulum@state[3] <= 50)    {
        rowvec[[i]] <- list(t = pendulum@state[3],
                            theta = pendulum@state[1],
                            thetaDot = pendulum@state[2])
        pendulum <- step(pendulum)
        i <- i + 1
    }
    DT <- data.table::rbindlist(rowvec)
    return(DT)
}

solution <- PendulumEulerApp()
plot(solution)
```

ODESolver-class *ODESolver virtual class*

Description

A virtual class inherited by AbstractODESolver

Arguments

object	a class object
stepSize	size of the step

RK4 *RK4.R*

Description

RK4 ODE solver

RK4 class constructor

Usage

```
RK4(ode, ...)

## S4 method for signature 'ODE'
RK4(ode, ...)
```

Arguments

ode	an ODE object
...	additional parameters

Details

RK4 class

Examples

```
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  base class: Projectile.R
# Projectile class to be solved with Euler method


setClass("Projectile", slots = c(
    g = "numeric",
    odeSolver = "RK4"
    ),
    prototype = prototype(
        g = 9.8
    ),
    contains = c("ODE")
    )
```

```
setMethod("initialize", "Projectile", function(.Object) {
    .Object@odeSolver <- RK4(.Object)
    return(.Object)
})

setMethod("setStepSize", "Projectile", function(object, stepSize, ...) {
    # use explicit parameter declaration
    # setStepSize generic has two step parameters: stepSize and dt
    object@odeSolver <- setStepSize(object@odeSolver, stepSize)
    object
})


setMethod("step", "Projectile", function(object) {
    object@odeSolver <- step(object@odeSolver)
    object@rate  <- object@odeSolver@ode@rate
    object@state <- object@odeSolver@ode@state
    object
})


setMethod("setState", signature("Projectile"), function(object, x, vx, y, vy, ...) {
    object@state[1] <- x
    object@state[2] <- vx
    object@state[3] <- y
    object@state[4] <- vy
    object@state[5] <- 0     # t + dt
    object@odeSolver@ode@state <- object@state
    object
})


setMethod("getState", "Projectile", function(object) {
    object@state
})


setMethod("getRate", "Projectile", function(object, state, ...) {
    object@rate[1] <- state[2]     # rate of change of x
    object@rate[2] <- 0            # rate of change of vx
    object@rate[3] <- state[4]     # rate of change of y
    object@rate[4] <- - object@g   # rate of change of vy
    object@rate[5] <- 1            # dt/dt = 1

    object@rate
})


# constructor
Projectile <- function()  new("Projectile")
# ++++++++++++++++++++++++++++++++++++++++++++++++++      example: PendulumApp.R
# Simulation of a pendulum using the EulerRichardson ODE solver
```

```
suppressPackageStartupMessages(library(ggplot2))

importFromExamples("Pendulum.R")      # source the class

PendulumApp <- function(verbose = FALSE) {
    # initial values
    theta <- 0.2
    thetaDot <- 0
    dt <- 0.1
    ode <- new("ODE")
    pendulum <- Pendulum()
    pendulum@state[3] <- 0      # set time to zero, t = 0
    pendulum <- setState(pendulum, theta, thetaDot)
    pendulum <- setStepSize(pendulum, dt = dt) # using stepSize in RK4
    pendulum@odeSolver <- setStepSize(pendulum@odeSolver, dt) # set new step size
    rowvec <- vector("list")
    i <- 1
    while (pendulum@state[3] <= 40)    {
        rowvec[[i]] <- list(t  = pendulum@state[3],    # time
                            theta = pendulum@state[1], # angle
                            thetadot = pendulum@state[2]) # derivative of angle
        pendulum <- step(pendulum)
        i <- i + 1
    }
    DT <- data.table::rbindlist(rowvec)
    return(DT)
}
# show solution
solution <- PendulumApp()
plot(solution)
# ++++++++++++++++++++++++++++++++++++++++++++++++++ application: ReactionApp.R
# ReactionApp solves an autocatalytic oscillating chemical
# reaction (Brusselator model) using
# a fourth-order Runge-Kutta algorithm.

importFromExamples("Reaction.R")      # source the class

ReactionApp <- function(verbose = FALSE) {
    X <- 1; Y <- 5;
    dt <- 0.1

    reaction <- Reaction(c(X, Y, 0))
    solver <- RK4(reaction)
    rowvec <- vector("list")
    i <- 1
    while (solver@ode@state[3] < 100) {             # stop at t = 100
        rowvec[[i]] <- list(t = solver@ode@state[3],
                            X = solver@ode@state[1],
                            Y = solver@ode@state[2])
        solver <- step(solver)
        i <-  i + 1
    }
    DT <- data.table::rbindlist(rowvec)
```

```
    return(DT)
}


solution <- ReactionApp()
plot(solution)
```

RK45-class *RK45 class*

Description

RK45 class

RK45 class constructor

Usage

```
RK45(ode)
```

Arguments

ode	and ODE object

Examples

```
# ++++++++++++++++++++++++++++++++++++++++++++++  example: ComparisonRK45App.R
# Compares the solution by the RK45 ODE solver versus the analytical solution
# Example file: ComparisonRK45App.R
# ODE Solver:   Runge-Kutta 45
# Class:        RK45

importFromExamples("ODETest.R")

 ComparisonRK45App <- function(verbose = FALSE) {
     ode <- new("ODETest")                    # create an `ODETest` object
     ode_solver <- RK45(ode)                  # select the ODE solver
     ode_solver <- setStepSize(ode_solver, 1)      # set the step
     ode_solver <- setTolerance(ode_solver, 1e-8)  # set the tolerance
     time <-  0
     rowVector <- vector("list")
     i <- 1
     while (time < 50) {
         rowVector[[i]] <- list(t  = ode_solver@ode@state[2],
                                s1 = getState(ode_solver@ode)[1],
                                s2 = getState(ode_solver@ode)[2],
                                xs = getExactSolution(ode_solver@ode, time),
                                rc = getRateCounts(ode),
                                time = time)
```

```
        ode_solver <- step(ode_solver)       # advance one step
        stepSize <-  ode_solver@stepSize     # update the step size
        time <- time + stepSize
        state <- getState(ode_solver@ode)    # get the `state` vector
        i <- i + 1
    }
    DT <- data.table::rbindlist(rowVector)    # a data table with the results
    return(DT)
}
# show solution
solution <- ComparisonRK45App()                          # run the example
plot(solution)
#  +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ example KeplerApp.R
#  KeplerApp solves an inverse-square law model (Kepler model) using an adaptive
#  stepsize algorithm.
#  Application showing two planet orbiting
#  File in examples: KeplerApp.R

importFromExamples("Kepler.R") # source the class Kepler

KeplerApp <- function(verbose = FALSE) {

    # set the orbit into a predefined state.
    r <- c(2, 0)                                  # orbit radius
    v <- c(0, 0.25)                               # velocity
    dt <- 0.1

    planet <- Kepler(r, v)
    solver <- RK45(planet)

    rowVector <- vector("list")
    i <- 1
    while (planet@state[5] <= 10) {
        rowVector[[i]] <- list(t  = planet@state[5],
                               planet1.r = planet@state[1],
                               p1anet1.v = planet@state[2],
                               planet2.r = planet@state[3],
                               p1anet2.v = planet@state[4])
        solver <- step(solver)
        planet <- solver@ode
        i <-  i + 1
    }
    DT <- data.table::rbindlist(rowVector)

    return(DT)
}


solution <- KeplerApp()
plot(solution)
```

`rODE` *rODE.*

Description

rODE.

`run_test_examples` *Run test of all the examples*

Description

Run test of all the examples

Usage

```
run_test_examples()
```

`setState` *New setState that should work with different methods "theta", "thetaDot": used in PendulumApp "x", "vx", "y", "vy": used in ProjectileApp*

Description

New setState that should work with different methods "theta", "thetaDot": used in PendulumApp "x", "vx", "y", "vy": used in ProjectileApp

Usage

```
setState(object, ...)
```

Arguments

`object`	a class object
`...`	additional parameters

Examples

```
# ++++++++++++++++++++++++++++++++++++++++++++++++ application: ProjectileApp.R
#                                                    test Projectile with RK4
#                                                    originally uses Euler

# suppressMessages(library(data.table))

importFromExamples("Projectile.R")      # source the class

ProjectileApp <- function(verbose = FALSE) {
    # initial values
    x <- 0; vx <- 10; y <- 0; vy <- 10
    state <- c(x, vx, y, vy, 0)                       # state vector
    dt <- 0.01

    projectile <- Projectile()
    projectile <- setState(projectile, x, vx, y, vy)
    projectile@odeSolver <- init(projectile@odeSolver, 0.123)
    projectile@odeSolver <- setStepSize(projectile@odeSolver, dt)
    rowV <- vector("list")
    i <- 1
    while (projectile@state[3] >= 0)    {
        rowV[[i]] <- list(t = projectile@state[5],
                          x  = projectile@state[1],
                          vx = projectile@state[2],
                          y  = projectile@state[3],     # vertical position
                          vy = projectile@state[4])
        projectile <- step(projectile)
        i <- i + 1
    }
    DT <- data.table::rbindlist(rowV)
    return(DT)
}


solution <- ProjectileApp()
plot(solution)
# ++++++++++++++++++++++++++++++++++++++++++++++++      example: PendulumApp.R
# Simulation of a pendulum using the EulerRichardson ODE solver

suppressPackageStartupMessages(library(ggplot2))

importFromExamples("Pendulum.R")      # source the class

PendulumApp <- function(verbose = FALSE) {
    # initial values
    theta <- 0.2
    thetaDot <- 0
    dt <- 0.1
    ode <- new("ODE")
    pendulum <- Pendulum()
    pendulum@state[3] <- 0      # set time to zero, t = 0
```

```
    pendulum <- setState(pendulum, theta, thetaDot)
    pendulum <- setStepSize(pendulum, dt = dt) # using stepSize in RK4
    pendulum@odeSolver <- setStepSize(pendulum@odeSolver, dt) # set new step size
    rowvec <- vector("list")
    i <- 1
    while (pendulum@state[3] <= 40)    {
        rowvec[[i]] <- list(t  = pendulum@state[3],    # time
                            theta = pendulum@state[1], # angle
                            thetadot = pendulum@state[2]) # derivative of angle
        pendulum <- step(pendulum)
        i <- i + 1
    }
    DT <- data.table::rbindlist(rowvec)
    return(DT)
}
# show solution
solution <- PendulumApp()
plot(solution)
```

`setStepSize` *setStepSize uses either of two step parameters: stepSize and dt 'stepSize" works for most of the applications 'dt" is used in Pendulum*

Description

setStepSize uses either of two step parameters: stepSize and dt 'stepSize" works for most of the applications 'dt" is used in Pendulum

Set the size of the step

Usage

```
setStepSize(object, ...)

## S4 method for signature 'ODESolver'
setStepSize(object, stepSize, ...)

## S4 method for signature 'AbstractODESolver'
setStepSize(object, stepSize, ...)

## S4 method for signature 'DormandPrince45'
setStepSize(object, stepSize, ...)

## S4 method for signature 'Euler'
setStepSize(object, stepSize, ...)

## S4 method for signature 'Euler'
getStepSize(object, ...)
```

Arguments

object	a class object
...	additional parameters
stepSize	size of the step

Examples

```
# ++++++++++++++++++++++++++++++++++++++++++++++++++   application:  SpringRK4.R
# Simulation of a spring considering no friction
library(rODE)


setClass("SpringRK4", slots = c(
    # we should improve this by letting the user entered these values
    K         = "numeric",
    mu        = "numeric",
    mass      = "numeric",
    state     = "numeric",
    odeSolver = "RK4"
    ),
    prototype = prototype(
        K = 1,
        state = c(0, 0, 0)
    ),
    contains = c("ODE")
    )

setMethod("initialize", "SpringRK4", function(.Object) {
    # we should improve this by letting the user entered these values
    .Object@K    <- 1.0
    .Object@mu   <- 1.5
    .Object@mass <- 20
    .Object@odeSolver <- RK4(.Object)
    return(.Object)
})

setMethod("setStepSize", signature("SpringRK4"), function(object, dt, ...) {
    # use explicit parameter declaration
    # setStepSize generic may use two different step parameters: stepSize and dt
    object@odeSolver <- setStepSize(object@odeSolver, dt)
    object
})


setMethod("step", "SpringRK4", function(object) {
    object@odeSolver <- step(object@odeSolver)
    object@rate  <- object@odeSolver@ode@rate
    object@state <- object@odeSolver@ode@state
    object
})
```

```
setMethod("setState", "SpringRK4", function(object, theta, thetaDot) {
    object@state[1] <- theta     # angle
    object@state[2] <- thetaDot  # derivative of the angle
    #                              state[3] is time
    object@odeSolver@ode@state <- object@state      # set state on solver
    object
})

setMethod("getState", "SpringRK4", function(object) {
    object@state
})

setMethod("getRate", "SpringRK4", function(object, state, ...) {
    # enter the derivatives here
    object@rate[1] <- state[2]     # rate of change of angle
    object@rate[2] <- -object@mu /  object@mass * state[2] - object@K * state[1]
    object@rate[3] <- 1            # rate of change of time, dt/dt

    object@rate
})

# constructor
SpringRK4 <- function()  new("SpringRK4")


# run application
SpringRK4App <- function(verbose = FALSE) {
    theta    <- 0
    thetaDot <- -0.2
    tmax     <- 22; dt <- 0.1
    ode <- new("ODE")
    spring <- SpringRK4()
    spring@state[3] <- 0      # set time to zero, t = 0
    spring <- setState(spring, theta, thetaDot)
    spring <- setStepSize(spring, dt = dt) # using stepSize in RK4
    spring@odeSolver <- setStepSize(spring@odeSolver, dt) # set new step size
    rowvec <- vector("list")
    i <- 1
    while (spring@state[3] <= tmax)    {
        rowvec[[i]] <- list(t  = spring@state[3],      # angle
                            y1 = spring@state[1],      # derivative of the angle
                            y2 = spring@state[2])      # time
        i <- i + 1
        spring <- step(spring)
    }
    DT <- data.table::rbindlist(rowvec)
    return(DT)
}

# show solution
solution <- SpringRK4App(TRUE)
plot(solution)
# ++++++++++++++++++++++++++++++++++++++++++++++  example: ComparisonRK45App.R
```

```
# Compares the solution by the RK45 ODE solver versus the analytical solution
# Example file: ComparisonRK45App.R
# ODE Solver:   Runge-Kutta 45
# Class:        RK45

importFromExamples("ODETest.R")

 ComparisonRK45App <- function(verbose = FALSE) {
     ode <- new("ODETest")                    # create an `ODETest` object
     ode_solver <- RK45(ode)                  # select the ODE solver
     ode_solver <- setStepSize(ode_solver, 1)      # set the step
     ode_solver <- setTolerance(ode_solver, 1e-8)  # set the tolerance
     time <-  0
     rowVector <- vector("list")
     i <- 1
     while (time < 50) {
         rowVector[[i]] <- list(t  = ode_solver@ode@state[2],
                                s1 = getState(ode_solver@ode)[1],
                                s2 = getState(ode_solver@ode)[2],
                                xs = getExactSolution(ode_solver@ode, time),
                                rc = getRateCounts(ode),
                                time = time)
         ode_solver <- step(ode_solver)       # advance one step
         stepSize <-  ode_solver@stepSize     # update the step size
         time <- time + stepSize
         state <- getState(ode_solver@ode)    # get the `state` vector
         i <- i + 1
     }
     DT <- data.table::rbindlist(rowVector)    # a data table with the results
     return(DT)
 }
# show solution
solution <- ComparisonRK45App()                         # run the example
plot(solution)
```

`setTolerance` *Set the tolerance for the solver*

Description

Set the tolerance for the solver

Usage

```
setTolerance(object, tol, ...)

## S4 method for signature 'ODEAdaptiveSolver'
setTolerance(object, tol)

## S4 method for signature 'DormandPrince45'
setTolerance(object, tol)
```

Arguments

object	a class object
tol	tolerance
...	additional parameters

See Also

Other adaptive solver methods: getErrorCode, getTolerance

Examples

```
# ++++++++++++++++++++++++++++++++++++++++++++++  example: ComparisonRK45App.R
# Compares the solution by the RK45 ODE solver versus the analytical solution
# Example file: ComparisonRK45App.R
# ODE Solver:   Runge-Kutta 45
# Class:        RK45

importFromExamples("ODETest.R")

 ComparisonRK45App <- function(verbose = FALSE) {
     ode <- new("ODETest")                    # create an `ODETest` object
     ode_solver <- RK45(ode)                  # select the ODE solver
     ode_solver <- setStepSize(ode_solver, 1)      # set the step
     ode_solver <- setTolerance(ode_solver, 1e-8)  # set the tolerance
     time <-  0
     rowVector <- vector("list")
     i <- 1
     while (time < 50) {
         rowVector[[i]] <- list(t  = ode_solver@ode@state[2],
                                s1 = getState(ode_solver@ode)[1],
                                s2 = getState(ode_solver@ode)[2],
                                xs = getExactSolution(ode_solver@ode, time),
                                rc = getRateCounts(ode),
                                time = time)
         ode_solver <- step(ode_solver)       # advance one step
         stepSize <-  ode_solver@stepSize     # update the step size
         time <- time + stepSize
         state <- getState(ode_solver@ode)    # get the `state` vector
         i <- i + 1
     }
     DT <- data.table::rbindlist(rowVector)    # a data table with the results
     return(DT)
 }
# show solution
solution <- ComparisonRK45App()                         # run the example
plot(solution)
# ++++++++++++++++++++++++++++++++++++++++ example: KeplerDormandPrince45App.R
# Demostration of the use of ODE solver RK45 for a particle subjected to
# a inverse-law force. The difference with the example KeplerApp is we are
# seeing the effect in thex and y axis on the particle.
# The original routine used the Verlet ODE solver
```

```
importFromExamples("KeplerDormandPrince45.R")

KeplerDormandPrince45App <- function(verbose = FALSE) {
    # values for the examples
    x  <- 1
    vx <- 0
    y  <- 0
    vy <- 2 * pi
    dt <- 0.01          # step size
    tol <- 1e-3         # tolerance
    particle <- Kepler()                            # use class Kepler
    particle <- init(particle, c(x, vx, y, vy, 0))  # enter state vector
    odeSolver <- DormandPrince45(particle)      # select the ODE solver
    odeSolver <- init(odeSolver, dt)            # start the solver
    odeSolver <- setTolerance(odeSolver, tol)   # this works for adaptive solvers
    particle@odeSolver <- odeSolver             # copy the solver to ODE object
    initialEnergy <- getEnergy(particle)        # calculate the energy
    rowVector <- vector("list")
    i <- 1
    while (getTime(particle) < 1.5) {
        rowVector[[i]] <- list(t  = particle@state[5],
                               x  = particle@state[1],
                               vx = particle@state[2],
                               y  = particle@state[3],
                               vx = particle@state[4],
                               energy = getEnergy(particle) )
        particle <- doStep(particle)            # advance one step
        energy   <- getEnergy(particle)         # calculate energy
        i <- i + 1
    }
    DT <- data.table::rbindlist(rowVector)
    return(DT)
}


solution <- KeplerDormandPrince45App()
plot(solution)

# ++++++++++++++++++++++++++++++++++++++++ Example:      ComparisonRK45ODEApp.R
# Updates the ODE state instead of using the internal state in the ODE solver
# Also plots the solver solution versus the analytical solution at a
# tolerance of 1e-6
# ODE Solver:   Runge-Kutta 45
# Class:        RK45

importFromExamples("ODETest.R")

ComparisonRK45ODEApp <- function(verbose = FALSE) {
    ode <- new("ODETest")                       # new ODE instance
    ode_solver <- RK45(ode)                     # select ODE solver
    ode_solver <- setStepSize(ode_solver, 1)    # set the step
    ode_solver <- setTolerance(ode_solver, 1e-6)  # set the tolerance
```

```
    time <-  0
    rowVector <- vector("list")                  # row vector
    i <- 1    # counter
    while (time < 50) {
        # add solution objects to a row vector
        rowVector[[i]] <- list(t = ode_solver@ode@state[2],
                               ODE = getState(ode_solver@ode)[1],
                               s2  = getState(ode_solver@ode)[2],
                               exact = getExactSolution(ode_solver@ode, time),
                               rate.counts = getRateCounts(ode),
                               time = time )
        ode_solver <- step(ode_solver)           # advance solver one step
        stepSize <-  getStepSize(ode_solver)     # get the current step size
        time <- time + stepSize
        ode <- ode_solver@ode                    # get updated ODE object
        state <- getState(ode)                   # get the `state` vector
        i <- i + 1                               # add a row vector
    }
    DT <- data.table::rbindlist(rowVector)       # create data table
    return(DT)
}

solution <- ComparisonRK45ODEApp()
plot(solution)
library(ggplot2)
library(dplyr)
library(tidyr)
solution.multi <- solution %>%
    select(t, ODE, exact)
plot(solution.multi)
solution.2x1 <- solution.multi %>%
    gather(key, value, -t)
g <- ggplot(solution.2x1, mapping = aes(x = t, y = value, color = key))
g <-  g + geom_line(size = 1) + labs(title = "ODE vs Exact solution",
                                     subtitle = "tolerance = 1E-6")
print(g)
```

`showMethods2` *Get the methods in a class*

Description

But only those specific to the class

Usage

```
showMethods2(theClass)
```

Arguments

theClass	class to analyze

---

step *Advances a step in the ODE solver*

Description

Advances a step in the ODE solver

Usage

```
step(object, ...)

## S4 method for signature 'ODESolver'
step(object, ...)

## S4 method for signature 'AbstractODESolver'
step(object, ...)

## S4 method for signature 'DormandPrince45'
step(object, ...)

## S4 method for signature 'Euler'
step(object, ...)

## S4 method for signature 'EulerRichardson'
step(object, ...)

## S4 method for signature 'RK4'
step(object, ...)

## S4 method for signature 'Verlet'
step(object, ...)
```

Arguments

object	a class object
...	additional parameters

Examples

```
# ++++++++++++++++++++++++++++++++++++++++++++++++++ application: ReactionApp.R
# ReactionApp solves an autocatalytic oscillating chemical
# reaction (Brusselator model) using
# a fourth-order Runge-Kutta algorithm.
```

```
importFromExamples("Reaction.R")      # source the class

ReactionApp <- function(verbose = FALSE) {
    X <- 1; Y <- 5;
    dt <- 0.1

    reaction <- Reaction(c(X, Y, 0))
    solver <- RK4(reaction)
    rowvec <- vector("list")
    i <- 1
    while (solver@ode@state[3] < 100) {             # stop at t = 100
        rowvec[[i]] <- list(t = solver@ode@state[3],
                            X = solver@ode@state[1],
                            Y = solver@ode@state[2])
        solver <- step(solver)
        i <-  i + 1
    }
    DT <- data.table::rbindlist(rowvec)
    return(DT)
}


solution <- ReactionApp()
plot(solution)
```

`Verlet` *Get the rate counter*

Description

How many time the rate has changed with a step

Verlet class

Verlet class constructor ODE

Usage

```
Verlet(ode, ...)

getRateCounter(object, ...)

## S4 method for signature 'Verlet'
getRateCounter(object, ...)

## S4 method for signature 'ODE'
Verlet(ode, ...)
```

Arguments

ode	an ODE object
...	additional parameters
object	a class object

Examples

```
# ++++++++++++++++++++++++++++++++++++++++++++++++++   application:  Logistic.R
# Simulates the logistic equation

setClass("Logistic", slots = c(
    K = "numeric",
    r = "numeric",
    odeSolver = "Verlet",
    counter = "numeric"
),
contains = c("ODE")
)

setMethod("initialize", "Logistic", function(.Object, ...) {
    .Object@K <- 10
    .Object@r <- 1.0
    .Object@state <- vector("numeric", 3)  # x, vx
    .Object@odeSolver <- Verlet(.Object)
    .Object@counter <- 0
    return(.Object)
})

setMethod("doStep", "Logistic", function(object, ...) {
    # cat("state@doStep=", object@state, "\n")
    object@odeSolver <- step(object@odeSolver)
    object@state <- object@odeSolver@ode@state
    object
})

setMethod("getTime", "Logistic", function(object, ...) {
    return(object@state[3])
})


setMethod("init", "Logistic", function(object, initState, r, K, ...) {
    object@r <- r
    object@K <- K
    object@state <- initState
    object@odeSolver <- init(object@odeSolver, getStepSize(object@odeSolver))
    object@counter <- 0
    object
})


setMethod("getRate", "Logistic", function(object, state, ...) {
```

```
    # Computes the rate using the given state.
    object@rate[1] <- state[2]
    object@rate[2] <- object@r * state[1] * (1 - state[1] / object@K)
    object@rate[3] <- 1   # time derivative
    object@counter <- object@counter + 1
    object@rate

})


setMethod("getState", "Logistic", function(object, ...) {
    # Gets the state variables.
    return(object@state)
})

# constructor
Logistic <- function() {
    logistic <- new("Logistic")
    return(logistic)
}

# Run the application
LogisticVerletApp <- function(verbose = FALSE) {
    x  <- 0.1
    vx <- 0
    r  <- 2        # Malthusian parameter (rate of maximum population growth)
    K  <- 10.0     # carrying capacity of the environment
    dt   <- 0.01; tol  <- 1e-3; tmax <- 10
    population <- Logistic()
    population <- init(population, c(x, vx, 0), r, K)
    odeSolver <- Verlet(population)
    odeSolver <- init(odeSolver, dt)
    population@odeSolver <- odeSolver
    rowVector <- vector("list")
    i <- 1
    while (getTime(population) <= tmax) {
        rowVector[[i]] <- list(t = getTime(population),
                               s1 = population@state[1],
                               s2 = population@state[2])
        population <- doStep(population)
        i <- i + 1
    }
    DT <- data.table::rbindlist(rowVector)
    return(DT)
}
# show solution
solution <- LogisticVerletApp()
plot(solution)
# ++++++++++++++++++++++++++++++++++++++++++++++++  example: ComparisonRK45App.R
# Compares the solution by the RK45 ODE solver versus the analytical solution
# Example file: ComparisonRK45App.R
# ODE Solver:   Runge-Kutta 45
# Class:        RK45
```

```
importFromExamples("ODETest.R")

 ComparisonRK45App <- function(verbose = FALSE) {
     ode <- new("ODETest")                    # create an `ODETest` object
     ode_solver <- RK45(ode)                  # select the ODE solver
     ode_solver <- setStepSize(ode_solver, 1)      # set the step
     ode_solver <- setTolerance(ode_solver, 1e-8)  # set the tolerance
     time <-  0
     rowVector <- vector("list")
     i <- 1
     while (time < 50) {
         rowVector[[i]] <- list(t  = ode_solver@ode@state[2],
                                s1 = getState(ode_solver@ode)[1],
                                s2 = getState(ode_solver@ode)[2],
                                xs = getExactSolution(ode_solver@ode, time),
                                rc = getRateCounts(ode),
                                time = time)
         ode_solver <- step(ode_solver)       # advance one step
         stepSize <-  ode_solver@stepSize     # update the step size
         time <- time + stepSize
         state <- getState(ode_solver@ode)    # get the `state` vector
         i <- i + 1
     }
     DT <- data.table::rbindlist(rowVector)    # a data table with the results
     return(DT)
 }
# show solution
solution <- ComparisonRK45App()                         # run the example
plot(solution)
# ++++++++++++++++++++++++++++++++++++++++++++++++  example: KeplerEnergyApp.R
# Demostration of the use of the Verlet ODE solver
#

importFromExamples("KeplerEnergy.R") # source the class Kepler

KeplerEnergyApp <- function(verbose = FALSE) {
    # initial values
    x  <- 1
    vx <- 0
    y  <- 0
    vy <- 2 * pi
    dt <- 0.01
    tol <- 1e-3
    particle <- KeplerEnergy()
    particle <- init(particle, c(x, vx, y, vy, 0))
    odeSolver <- Verlet(particle)
    odeSolver <- init(odeSolver, dt)
    particle@odeSolver <- odeSolver
    initialEnergy <- getEnergy(particle)
    rowVector <- vector("list")
    i <- 1
    while (getTime(particle) <= 1.20) {
```

```
        rowVector[[i]] <- list(t  = particle@state[5],
                               x  = particle@state[1],
                               vx = particle@state[2],
                               y  = particle@state[3],
                               vy = particle@state[4],
                               E  = getEnergy(particle))
        particle <- doStep(particle)
        energy <- getEnergy(particle)
        i <- i + 1
    }
    DT <- data.table::rbindlist(rowVector)
    return(DT)
}


solution <- KeplerEnergyApp()
plot(solution)

# ++++++++++++++++++++++++++++++++++++++++++++++++  example: KeplerEnergyApp.R
# Demostration of the use of the Verlet ODE solver
#

importFromExamples("KeplerEnergy.R") # source the class Kepler

KeplerEnergyApp <- function(verbose = FALSE) {
    # initial values
    x  <- 1
    vx <- 0
    y  <- 0
    vy <- 2 * pi
    dt <- 0.01
    tol <- 1e-3
    particle <- KeplerEnergy()
    particle <- init(particle, c(x, vx, y, vy, 0))
    odeSolver <- Verlet(particle)
    odeSolver <- init(odeSolver, dt)
    particle@odeSolver <- odeSolver
    initialEnergy <- getEnergy(particle)
    rowVector <- vector("list")
    i <- 1
    while (getTime(particle) <= 1.20) {
        rowVector[[i]] <- list(t  = particle@state[5],
                               x  = particle@state[1],
                               vx = particle@state[2],
                               y  = particle@state[3],
                               vy = particle@state[4],
                               E  = getEnergy(particle))
        particle <- doStep(particle)
        energy <- getEnergy(particle)
        i <- i + 1
    }
    DT <- data.table::rbindlist(rowVector)
    return(DT)
```

```
}


solution <- KeplerEnergyApp()
plot(solution)

# +++++++++++++++++++++++++++++++++++++++++++++++++++   application:  Logistic.R
# Simulates the logistic equation

setClass("Logistic", slots = c(
    K = "numeric",
    r = "numeric",
    odeSolver = "Verlet",
    counter = "numeric"
),
contains = c("ODE")
)

setMethod("initialize", "Logistic", function(.Object, ...) {
    .Object@K <- 10
    .Object@r <- 1.0
    .Object@state <- vector("numeric", 3)  # x, vx
    .Object@odeSolver <- Verlet(.Object)
    .Object@counter <- 0
    return(.Object)
})

setMethod("doStep", "Logistic", function(object, ...) {
    # cat("state@doStep=", object@state, "\n")
    object@odeSolver <- step(object@odeSolver)
    object@state <- object@odeSolver@ode@state
    object
})

setMethod("getTime", "Logistic", function(object, ...) {
    return(object@state[3])
})


setMethod("init", "Logistic", function(object, initState, r, K, ...) {
    object@r <- r
    object@K <- K
    object@state <- initState
    object@odeSolver <- init(object@odeSolver, getStepSize(object@odeSolver))
    object@counter <- 0
    object
})


setMethod("getRate", "Logistic", function(object, state, ...) {
    # Computes the rate using the given state.
    object@rate[1] <- state[2]
    object@rate[2] <- object@r * state[1] * (1 - state[1] / object@K)
```

```
    object@rate[3] <- 1   # time derivative
    object@counter <- object@counter + 1
    object@rate

})


setMethod("getState", "Logistic", function(object, ...) {
    # Gets the state variables.
    return(object@state)
})

# constructor
Logistic <- function() {
    logistic <- new("Logistic")
    return(logistic)
}

# Run the application
LogisticVerletApp <- function(verbose = FALSE) {
    x  <- 0.1
    vx <- 0
    r  <- 2        # Malthusian parameter (rate of maximum population growth)
    K  <- 10.0     # carrying capacity of the environment
    dt   <- 0.01; tol  <- 1e-3; tmax <- 10
    population <- Logistic()
    population <- init(population, c(x, vx, 0), r, K)
    odeSolver <- Verlet(population)
    odeSolver <- init(odeSolver, dt)
    population@odeSolver <- odeSolver
    rowVector <- vector("list")
    i <- 1
    while (getTime(population) <= tmax) {
        rowVector[[i]] <- list(t = getTime(population),
                               s1 = population@state[1],
                               s2 = population@state[2])
        population <- doStep(population)
        i <- i + 1
    }
    DT <- data.table::rbindlist(rowVector)
    return(DT)
}
# show solution
solution <- LogisticVerletApp()
plot(solution)
```

Index