Package ‘diffeqr’

September 22, 2019

Type Package

Title Solving Differential Equations (ODEs, SDEs, DDEs, DAEs)

Version 0.1.3

Description An interface to 'DifferentialEquations.jl' <http://docs.juliadiffeq.org/latest/> from the R programming language.
It has unique high performance methods for solving ordinary differential equations (ODE), stochastic differential equations (SDE), delay differential equations (DDE), differential-algebraic equations (DAE), and more. Much of the functionality, including features like adaptive time stepping in SDEs, are unique and allow for multiple orders of magnitude speedup over more common methods.
'diffeqr' attaches an R interface onto the package, allowing seamless use of this tooling by R users.

Depends R (>= 3.4.0)

Encoding UTF-8

License MIT + file LICENSE

URL https://github.com/JuliaDiffEq/diffeqr

LazyData true

SystemRequirements Julia (>= 0.6.0), DifferentialEquations.jl

Imports JuliaCall, stringr

RoxygenNote 6.1.1

Suggests testthat, knitr, rmarkdown

VignetteBuilder knitr

NeedsCompilation no

Author Christopher Rackauckas [aut, cre]

Maintainer Christopher Rackauckas <me@chrisrackauckas.com>

Repository CRAN

Date/Publication 2019-09-22 16:50:06 UTC
R topics documented:

dae.solve ................................................................. 2
dde.solve ................................................................. 3
diffeq_setup ............................................................ 5
ode.solve ............................................................... 5
sde.solve ............................................................... 7

Index 10

dae.solve Solve Differential-Algebraic Equations (DAE)

Description

Solves a DAE with f(du,u,p,t)=0 for u(0)=u0 over the tspan

Usage

dae.solve(f, du0, u0, tspan, p = NULL, alg = "nothing",
         reltol = 0.001, abstol = 1e-06, maxiters = 1e+06, saveat = NULL,
         differential_vars = NULL)

Arguments

f the implicit ODE function.

du0 the initial derivative. Can be a number or (arbitrary dimension) array.

u0 the initial condition. Can be a number or (arbitrary dimension) array.

tspan the timespan to solve over. Should be a list of two values: (initial time, end time).

p the parameters. Defaults to no parameters. Can be a number or an array.

alg the algorithm used to solve the differential equation. Defaults to an adaptive choice. Algorithm choices are done through a string which matches the DifferentialEquations.jl form.

reltol the relative tolerance of the ODE solver. Defaults to 1e-3.

abstol the absolute tolerance of the ODE solver. Defaults to 1e-6.

maxiters the maximum number of iterations the adaptive solver is allowed to try before exiting. Default value is 1000000.

saveat the time points to save values at. Should be an array of times. Defaults to automatic.

differential_vars boolean array declaring which variables are differential. All falses correspond to purely algebraic variables.
Value

sol. Has the sol$t for the time points and sol$u for the values.

Examples

```r
## diffeq_setup() is time-consuming and requires Julia+DifferentialEquations.jl
diffeqr::diffeq_setup()

f <- function (du,u,p,t) {
  c(resid1, resid2, resid3)
}
u0 = c(1.0, 0, 0)
du0 = c(-0.04, 0.04, 0.0)
tspan = list(0.0,100000.0)
differential_vars = c(TRUE,TRUE,FALSE)
sol = diffeqr::dae.solve(f,du0,u0,tspan,differential_vars=differential_vars)
udf = as.data.frame(sol$u)
#plotly::plot_ly(udf, x = sol$t, y = ~V1, type = "scatter", mode = 'lines') %>%
#plotly::add_trace(y = ~V2) %>%
#plotly::add_trace(y = ~V3)

f = JuliaCall::julia_eval("function f(out,du,u,p,t)
end")
sol = diffeqr::dae.solve('f',du0,u0,tspan,differential_vars=differential_vars)
```

---

dde.solve  Solve Delay Differential Equations (DDE)

description

Solves a DDE with f(u,p,t)=0 for u(0)=u0 over the tspan

Usage

```r
dde.solve(f, u0, h, tspan, p = NULL, alg = "nothing", reltol = 0.001,
  abstol = 1e-06, maxiters = 1e+06, saveat = NULL,
  constant_lags = NULL)
```
Arguments

- **f**: the implicit ODE function.
- **u0**: the initial condition. Can be a number or (arbitrary dimension) array.
- **h**: is the history function (p,t) which gives values of the solution before the initial time point.
- **tspan**: the timespan to solve over. Should be a list of two values: (initial time, end time).
- **p**: the parameters. Defaults to no parameters. Can be a number or an array.
- **alg**: the algorithm used to solve the differential equation. Defaults to an adaptive choice. Algorithm choices are done through a string which matches the DifferentialEquations.jl form.
- **reltol**: the relative tolerance of the ODE solver. Defaults to 1e-3.
- **abstol**: the absolute tolerance of the ODE solver. Defaults to 1e-6.
- **maxiters**: the maximum number of iterations the adaptive solver is allowed to try before exiting. Default value is 1000000.
- **saveat**: the time points to save values at. Should be an array of times. Defaults to automatic.
- **constant_lags**: a vector of floats for the constant-time lags. Defaults to NULL.

Value

- **sol**: Has the sol$t for the time points and sol$u for the values.

Examples

```r
## diffeq_setup() is time-consuming and requires Julia+DifferentialEquations.jl
diffeqr::diffeq_setup()

f = JuliaCall::julia_eval("function f(du, u, h, p, t)
    du[1] = 1.1/(1 + sqrt(10)*(h(p, t-20)[1])^(5/4)) - 10*u[1]/(1 + 40*u[2])
end")

u0 = c(1.05767027/3, 1.030713491/3)

h <- function (p,t){
    c(1.05767027/3, 1.030713491/3)
}

tspan = list(0.0, 100.0)

constant_lags = c(20.0)

sol = diffeqr::dde.solve('f',u0,h,tspan,constant_lags=constant_lags)

udf = as.data.frame(sol$u)

#plotly::plot_ly(udf, x = sol$t, y = ~V1, type = 'scatter', mode = 'lines')
#plotly::plot_ly(udf, x = sol$t, y = ~V2)
```
diffeq_setup

Description

This function initializes Julia and the DifferentialEquations.jl package. The first time will be long since it includes precompilation.

Usage

diffeq_setup(...)

Arguments

... Parameters are passed down to JuliaCall::julia_setup

Examples

## diffeq_setup() is time-consuming and requires Julia+DifferentialEquations.jl

diffeqr::diffeq_setup()

ode.solve

Solve Ordinary Differential Equations (ODE)

Description

Solves an ODE with u’=f(u,p,t), for u(0)=u0 over the tspan

Usage

ode.solve(f, u0, tspan, p = NULL, alg = "nothing", reltol = 0.001,
abstol = 1e-06, maxiters = 1e+06, saveat = NULL)

Arguments

f the derivative function.
u0 the initial condition. Can be a number or (arbitrary dimension) array.
tspan the timespan to solve over. Should be a list of two values: (initial time, end time).
p the parameters. Defaults to no parameters. Can be a number or an array.
alg  the algorithm used to solve the differential equation. Defaults to an adaptive choice. Algorithm choices are done through a string which matches the DifferentialEquations.jl form.
reltol  the relative tolerance of the ODE solver. Defaults to 1e-3.
abstol  the absolute tolerance of the ODE solver. Defaults to 1e-6.
maxiters  the maximum number of iterations the adaptive solver is allowed to try before exiting. Default value is 1000000.
saveat  the time points to save values at. Should be an array of times. Defaults to automatic.

Value

sol. Has the sol$t for the time points and sol$u for the values.

Examples

```r
## diffeq_setup() is time-consuming and requires Julia+DifferentialEquations.jl
diffeqr::diffeq_setup()

# Scalar ODEs
f <- function(u,p,t) {
  return(1.01*u)
}
u0 = 1/2
tspan <- list(0.0,1.0)
sol = diffeqr::ode.solve(f,u0,tspan)
plot(sol$t,sol$u,"l")

saveat=1:10/10
sol2 = diffeqr::ode.solve(f,u0,tspan,saveat=saveat)
sol3 = diffeqr::ode.solve(f,u0,tspan,alg="Vern9()")
sol4 = diffeqr::ode.solve(f,u0,tspan,alg="Rosenbrock23()")

# Systems of ODEs
f <- function(u,p,t) {
  du1 = p[1]*(u[2]-u[1])
  du2 = u[1]*(p[2]-u[3]) - u[2]
  return(c(du1,du2,du3))
}
u0 = c(1.0,0.0,0.0)
tspan <- list(0.0,100.0)
p = c(10.0,28.0,8/3)
sol = diffeqr::ode.solve(f,u0,tspan,p=p)
udf = as.data.frame(sol$u)
matplot(sol$t,udf,"l",col=1:3)
```
#plotly::plot_ly(udf, x = ~V1, y = ~V2, z = ~V3, type = 'scatter3d', mode = 'lines')

f <- JuliaCall::julia_eval("function f(du,u,p,t)
    du[1] = 10.0*(u[2]-u[1])
end")
sol = diffeqr::ode.solve('f',u0,tspan)

---

### sde.solve

**Solve Stochastic Differential Equations (SDE)**

**Description**

Solves an SDE with \( du = f(u,p,t)dt + g(u,p,t)dW_t \), for \( u(0) = u_0 \) over the tspan

**Usage**

```r
sde.solve(f, g, u0, tspan, p = NULL, alg = "nothing",
            noise.dims = NULL, maxiters = 1e+06, reltol = 0.01,
            abstol = 0.01, saveat = NULL, seed = 0)
```

**Arguments**

- **f**
  - the drift function.
- **g**
  - the diffusion function.
- **u0**
  - the initial condition. Can be a number or (arbitrary dimension) array.
- **tspan**
  - the timespan to solve over. Should be a list of two values: (initial time, end time).
- **p**
  - the parameters. Defaults to no parameters. Can be a number or an array.
- **alg**
  - the algorithm used to solve the differential equation. Defaults to an adaptive choice. Algorithm choices are done through a string which matches the DifferentialEquations.jl form.
- **noise.dims**
  - list of the dimensions for the noise rate term. Defaults to NULL which gives diagonal noise.
- **maxiters**
  - the maximum number of iterations the adaptive solver is allowed to try before exiting. Defualt value is 1000000.
- **reltol**
  - the relative tolerance of the ODE solver. Defaults to 1e-3.
- **abstol**
  - the absolute tolerance of the ODE solver. Defaults to 1e-6.
- **saveat**
  - the time points to save values at. Should be an array of times. Defaults to automatic.
- **seed**
  - the seed for the random numbers. Defaults to zero for a random seed.
**Value**

sol. Has the sol$t for the time points and sol$u for the values.

**Examples**

```r
## diffeq_setup() is time-consuming and requires Julia+DifferentialEquations.jl
diffeqr::diffeq_setup()

# Scalar SDEs
f <- function(u,p,t) {
  return(1.01*u)
}
g <- function(u,p,t) {
  return(0.87*u)
}
u0 = 1 / 2
tspan <- list(0.0,1.0)
sol = diffeqr::sde.solve(f,g,u0,tspan)
#plotly::plot_ly(udf, x = sol$t, y = sol$u, type = 'scatter', mode = 'lines')

# Diagonal Noise SDEs
f <- JuliaCall::julia_eval("function f(du,u,p,t)
  du[1] = 10.0*(u[2]-u[1])
end")
g <- JuliaCall::julia_eval("function g(du,u,p,t)
  du[1] = 0.3*u[1]
  du[2] = 0.3*u[2]
  du[3] = 0.3*u[3]
end")
tspan <- list(0.0,100.0)
sol = diffeqr::sde.solve('f','g',u0,tspan,p=p,saveat=0.05)
udf = as.data.frame(sol$u)
#plotly::plot_ly(udf, x = ~V1, y = ~V2, z = ~V3, type = 'scatter3d', mode = 'lines')

# Non-Diagonal Noise SDEs
f <- JuliaCall::julia_eval("function f(du,u,p,t)
  du[1] = 10.0*(u[2]-u[1])
end")
g <- JuliaCall::julia_eval("
function g(du,u,p,t)
    du[1,1] = 0.3u[1]
    du[2,1] = 0.6u[1]
    du[3,1] = 0.2u[1]
    du[1,2] = 1.2u[2]
    du[2,2] = 0.2u[2]
    du[3,2] = 0.3u[2]
end"

u0 = c(1.0,0.0,0.0)
tspan <- list(0.0,100.0)
noise.dims = list(3,2)
sol = diffeqr::sde.solve('f','g',u0,tspan,saveat=0.005,noise.dims=noise.dims)
udf = as.data.frame(sol$u)
#plotly::plot_ly(udf, x = ~V1, y = ~V2, z = ~V3, type = 'scatter3d', mode = 'lines')
Index

dae.solve, 2
dde.solve, 3
diffeq_setup, 5
ode.solve, 5
sde.solve, 7