Package 'odeintr'

October 14, 2022

Type Package

Title C++ ODE Solvers Compiled on-Demand

Version 1.7.1

Date 2017-03-28

Author Timothy H. Keitt

Maintainer Timothy H. Keitt <tkeitt@gmail.com>

Description Wraps the Boost odeint library for integration of differential equations.

URL https://github.com/thk686/odeintr

BugReports https://github.com/thk686/odeintr/issues

License GPL (>= 2)

Imports Rcpp (>= 0.10.0)

LinkingTo Rcpp, BH

Suggests testthat, BH

RoxygenNote 6.0.1

SystemRequirements C++11

NeedsCompilation yes

Repository CRAN

Date/Publication 2017-03-29 16:16:39 UTC

R topics documented:

`compile_implicit` *Compile Stiff ODE system solver*

Description

Generates a stiff integrator using Rcpp

Usage

```
compile_implicit(name, sys, pars = NULL, const = FALSE, sys_dim = -1L,
  jacobian = JacobianCpp(sys, sys_dim), atol = 1e-06, rtol = 1e-06,
  globals = "", headers = "", footers = "", compile = TRUE,
  env = new.env(), ...)

JacobianCpp(sys, sys_dim = -1L)
```

Arguments

`name`	the name of the generated integration function
`sys`	a string containing C++ expressions
`pars`	a named vector of numbers or a vector of names or number of parameters
`const`	declare parameters const if true
`sys_dim`	length of the state vector
`jacobian`	C++ expressions computing the Jacobian
`atol`	absolute tolerance if using adaptive step size
`rtol`	relative tolerance if using adaptive step size
`globals`	a string with global C++ declarations
`headers`	code to appear before the `odeintr` namespace
`footers`	code to appear after the `odeintr` namespace
`compile`	if false, just return the code
`env`	install functions into this environment
`...`	passed to `sourceCpp`

Details

See `compile_sys` for details. This function behaves identically except that you cannot specify a custom observer and you must provide a Jacobian C++ function body. By default, the Jacobian will be symbollically computed from the system function using the `JacobianCpp` function. This uses `D` to compute the symbolic derivatives. The integration methods is the 4th-order Rosenbrock implicit solver. Step size is adaptive and output is interpolated.

If you provide a hand-written Jacobian, it must update the matrix `J` and the vector `dfdt`. It is perhaps easiest to use `JacobianCpp` to see the required format.

Author(s)

Timothy H. Keitt

See Also

set_optimization, compile_sys

Examples

```
## Not run:
# Lorenz model from odeint examples
pars = c(sigma = 10, R = 28, b = 8 / 3)
Lorenz.sys = '
  dxdt[0] = sigma * (x[1] - x[0]);
  dxdt[1] = R * x[0] - x[1] - x[0] * x[2];
  dxdt[2] = -b * x[2] + x[0] * x[1];
' # Lorenz.sys
cat(JacobianCpp(Lorenz.sys))
compile_implicit("lorenz", Lorenz.sys, pars, TRUE)
x = lorenz(rep(1, 3), 100, 0.001)
plot(x[, c(2, 4)], type = 'l', col = "steelblue")
Sys.sleep(0.5)
# Stiff example from odeint docs
Stiff.sys = '
 dxdt[0] = -101.0 * x[0] - 100.0 * x[1];
 dxdt[1] = x[0];
' # Stiff.sys
cat(JacobianCpp(Stiff.sys))
compile_implicit("stiff", Stiff.sys)
x = stiff(c(2, 1), 7, 0.001)
plot(x[, 1:2], type = "l", lwd = 2, col = "steelblue")
lines(x[, c(1, 3)], lwd = 2, col = "darkred")
# Robertson chemical kinetics problem
Robertson = '
dxdt[0] = -alpha * x[0] + beta * x[1] * x[2];
dxdt[1] = alpha * x[0] - beta * x[1] * x[2] - gamma * x[1] * x[1];
dxdt[2] = gamma * x[1] * x[1];
' # Robertson
pars = c(alpha = 0.04, beta = 1e4, gamma = 3e7)
init.cond = c(1, 0, 0)
cat(JacobianCpp(Robertson))
compile_implicit("robertson", Robertson, pars, TRUE)
at = 10 ^ seq(-5, 5, len = 400)
x = robertson_at(init.cond, at)
par(mfrow = c(3, 1), mar = rep(0.5, 4), oma = rep(5, 4), xpd = NA)
plot(x[, 1:2], type = "l", lwd = 3,
     col = "steelblue", log = "x", axes = F, xlab = NA)
axis(2); box()
plot(x[, c(1, 3)], type = "l", lwd = 3,
     col = "steelblue", log = "x", axes = F, xlab = NA)
axis(4); box()
plot(x[, c(1, 4)], type = "l", lwd = 3,
```

```
     col = "steelblue", log = "x", axes = F)
axis(2); axis(1); box()

## End(Not run)
```

`compile_sys` *Compile ODE system*

Description

Generates an integrator using Rcpp

Usage

```
compile_sys(name, sys, pars = NULL, const = FALSE, method = "rk5_i",
  sys_dim = -1L, atol = 1e-06, rtol = 1e-06, globals = "",
  headers = "", footers = "", compile = TRUE, observer = NULL,
  env = new.env(), ...)
```

Arguments

`name`	the name of the generated integration function
`sys`	a string containing C++ expressions
`pars`	a named vector of numbers or a vector of names or number of parameters
`const`	declare parameters const if true
`method`	a method string (see Details)
`sys_dim`	length of the state vector
`atol`	absolute tolerance if using adaptive step size
`rtol`	relative tolerance if using adaptive step size
`globals`	a string with global C++ declarations
`headers`	code to appear before the `odeintr` namespace
`footers`	code to appear after the `odeintr` namespace
`compile`	if false, just return the code
`observer`	an optional R function to record output
`env`	install functions into this environment
`...`	passed to `sourceCpp`

Details

C++ code is generated and compiled with sourceCpp. The returned function will integrate the system starting from a provided initial condition and initial time to a specified final time. An attempt is made to get the length of the state vector from the system definition. If this fails, the code will likely crash your R session. It is safer to specify `sys_dim` directly.

The C++ expressions must index a state array of length `sys_dim`. The state array is `x` and the derivatives are `dxdt`. The first state value is `x[0]` and the first derivative is `dxdt[0]`.

In the case you use bare `dxdt` and `x`, an attempt will be made to append `[0]` to these variables. This can fail, so do not rely on it. This will also fail if you set `sys_dim` to a positive value.

The `globals` string can be arbitrary C++ code. You can set global named parameter values here. Note that these will be defined within the `odeintr` namespace.

If you supply the `pars` argument, these parameters will be compiled into the code. There are three options: 1) if `pars` is a single number, then you can access a vector of parameters named `pars` of the specified length; 2) if `pars` is a character vectors, then a parameter will be defined for each; and 3) if the character vector is named, then the names will be used for the parameter names and the associated values will be used as defaults. If you specify `const = TRUE`, these named parameters will be declared const. Otherwise parameter getter/setter functions will be defined.

If `observer` is an R function, then this function will be used to record the output of the integration. It is called with signature `obsever(x, t)`. Its return value will be coerced to a list. Observer getter/setter functions will be emitted as well (`name_g(s)et_observer`). You can also get and set an output processing function (`name_g(s)et_output_processor`). It will be passed a 2-element list. The first element is a vector of time points and the 2nd element is a list of lists, one list per time point. The default processor converts this to a data frame.

You can insert arbitrary code outside the `odeintr` name space using `headers` and `footers`. This code can be anything compatible with Rcpp. You could for example define exported Rcpp functions that set simulation paramters. `headers` is inserted right after the Rcpp and ODEINT includes. `footers` is inserted at the end of the code.

The following methods can be used:

Code	Stepper	Type
`euler`	`euler`	Interpolating
`rk4`	`runge_kutta4`	Regular
`rk54`	`runge_kutta_cash_karp54`	Regular
`rk54_a`	`runge_kutta_cash_karp54`	Adaptive
`rk5`	`runge_kutta_dopri5`	Regular
`rk5_a`	`runge_kutta_dopri5`	Adaptive
`rk5_i`	`runge_kutta_dopri5`	Interpolating adaptive
`rk78`	`runge_kutta_fehlberg78`	Regular
`rk78_a`	`runge_kutta_fehlberg78`	Adaptive
`abN`	`adams_bashforth`	Order N multistep
`abmN`	`adams_bashforth_moulton`	Order N multistep
`bs`	`bulirsch_stoer`	Adaptive
`bsd`	`bulirsch_stoer_dense_out`	Interpolating adaptive

These steppers are described at here.

Value

The C++ code invisibly.

The following functions are generated:

Function	Use	Arguments
`name`	regular observer calls	`init, duration, step_size = 1.0, st`
`name_adap`	adaptive observer calls	`init, duration, step_size = 1.0, st`
`name_at`	specified observer calls	`init, times, step_size = 1.0, start`
`name_continue_at`	specified observer calls starting from previous final state	`times, step_size = 1.0`
`name_no_record`	no observer calls	`init, duration, step_size = 1.0, st`
`name_reset_observer`	clear observed record	void
`name_get_state`	get current state	void
`name_set_state`	set current state	`new_state`
`name_get_output`	fetch observed record	void
`name_get_params`	get parameter values	void
`name_set_params`	set parameter values	parameters

Arguments are:

`init`	vector of initial conditions
`duration`	end at start + duration
`step_size`	the integration step size; variable for adaptive methods
`start`	the starting time (always equal 0.0 for `name_at`)
`time`	vector of times as which to call the observer
`new_state`	vector of state values

Note

The c++11 plugin is enabled.

Author(s)

Timothy H. Keitt

See Also

`set_optimization`, `integrate_sys`

Examples

```
## Not run:
# Logistic growth
compile_sys("logistic", "dxdt = x * (1 - x)")
plot(logistic(0.001, 15, 0.1), type = "l", lwd = 2, col = "steelblue")
Sys.sleep(0.5)

# Lotka-Volterra predator-prey equations
pars = c(alpha = 1, beta = 1, gamma = 1, delta = 1)
```

```
LV.sys = '
  dxdt[0] = alpha * x[0] - beta * x[0] * x[1];
  dxdt[1] = gamma * x[0] * x[1] - delta * x[1];
' # LV.sys
compile_sys("preypred", LV.sys, pars, TRUE)
x = preypred(rep(2, 2), 100, 0.01)
plot(x[, 2:3], type = "l", lwd = 2,
     xlab = "Prey", ylab = "Predator",
     col = "steelblue")
Sys.sleep(0.5)

# Lorenz model from odeint examples
pars = c(sigma = 10, R = 28, b = 8 / 3)
Lorenz.sys = '
  dxdt[0] = sigma * (x[1] - x[0]);
  dxdt[1] = R * x[0] - x[1] - x[0] * x[2];
  dxdt[2] = -b * x[2] + x[0] * x[1];
' # Lorenz.sys
compile_sys("lorenz", Lorenz.sys, pars, TRUE)
x = lorenz(rep(1, 3), 100, 0.001)
plot(x[, c(2, 4)], type = 'l', col = "steelblue")

## End(Not run)
```

`compile_sys_openmp` *Compile ODE system with openmp multi-threading*

Description

Generates an integrator using Rcpp and openmp

Usage

```
compile_sys_openmp(name, sys, pars = NULL, const = FALSE,
  method = "rk5_i", sys_dim = -1L, atol = 1e-06, rtol = 1e-06,
  globals = "", headers = "", footers = "", compile = TRUE,
  env = new.env(), ...)
```

Arguments

`name`	the name of the generated integration function
`sys`	a string containing C++ expressions
`pars`	a named vector of numbers or a vector of names or number of parameters
`const`	declare parameters const if true
`method`	a method string (see `compile_sys`)
`sys_dim`	length of the state vector
`atol`	absolute tolerance if using adaptive step size

rtol	relative tolerance if using adaptive step size
globals	a string with global C++ declarations
headers	code to appear before the odeintr namespace
footers	code to appear after the odeintr namespace
compile	if false, just return the code
env	install functions into this environment
...	passed to sourceCpp

Details

This functions behaves identically to compile_sys execpt that it does not allow one to override the default observer. In order to take advantage of openmp multi-threading, you must insert openmp pragmas into your system definition. See the examples.

A special function laplace4 is defined and can be called from your system definition. It will compute a discrete 4-point Laplacian for use in solving PDE via the method of lines. The function takes x as its first argument, dxdt as its second argument and the diffusion coefficient D as its third parameter. This function uses the default openmp scheduling.

Author(s)

Timothy H. Keitt

See Also

set_optimization, compile_sys

Examples

```
## Not run:
M = 200
bistable = '
 laplace4(x, dxdt, D);  // parallel 4-point discrete laplacian
 #pragma omp parallel for
 for (int i = 0; i < N; ++i)
   dxdt[i] += a * x[i] * (1 - x[i]) * (x[i] - b);
' # bistable
compile_sys_openmp("bistable", bistable, sys_dim = M * M,
                   pars = c(D = 0.1, a = 1.0, b = 1/2),
                   const = TRUE)
at = 10 ^ (0:3)
inic = rbinom(M * M, 1, 1/2)
system.time({x = bistable_at(inic, at)})
par(mfrow = rep(2, 2), mar = rep(1, 4), oma = rep(1, 4))
for (i in 1:4){
  image(matrix(unlist(x[i, -1]), M, M),
        asp = 1, col = c("black", "lightgray"),
        axes = FALSE)
  title(main=paste("Time =", x[i, 1]))}
```

```
## End(Not run)
```

`integrate_sys` *Integrate an ODE system using ODEINT*

Description

Numerically integrates an ODE system defined in R

Usage

```
integrate_sys(sys, init, duration, step_size = 1, start = 0,
  adaptive_obs = FALSE, observer = function(x, t) x, atol = 1e-06,
  rtol = 1e-06)
```

Arguments

`sys`	a function with signature function(x, t)
`init`	the initial conditions
`duration`	time-span of the integration
`step_size`	the initial step size (adjusted internally)
`start`	the starting time
`adaptive_obs`	if true, call observer after each adaptive step
`observer`	a function with signature function(x, t) returning values to store in output
`atol`	absolute error tolerance
`rtol`	relative error tolerance

Details

The system will be integrated from `start` to `start` + `duration`. The method is an error controlled 5th-order Dormand-Prince. The time step will be adjusted to within error tolerances (1e-6 absolute and relative).

The observer can return arbitrary data in any form that can be coerced to a list. This could be a single scalar value (no need to wrap the return with `list`!) or a list containing heterogeneous types. These will be inserted into the columns of the returned data frame. If the observer function returns a zero-length object (`NULL` or `list()`), then nothing will be recorded. You can use the `t` argument to selectively sample the output.

Value

A data frame, `NULL` if no samples were recorded and a very complicated list-of-lists if the observer returned objects of different length.

Author(s)

Timothy H. Keitt

See Also

compile_sys

Examples

```
## Not run:
# Lotka-Volterra predator-prey equations
LV.sys = function(x, t)
{
   c(x[1] - 0.1 * x[1] * x[2],
     0.05 * x[1] * x[2] - 0.5 * x[2])
}
null_rec = function(x, t) NULL
system.time(integrate_sys(LV.sys, rep(1, 2), 1e3, observer = null_rec))
named_rec = function(x, t) c(Prey = x[1], Predator = x[2])
x = integrate_sys(LV.sys, rep(1, 2), 100, 0.01, observer = named_rec)
plot(x[, 2:3], type = "l", lwd = 3, col = "steelblue")
Sys.sleep(0.5)

# Lorenz model from odeint examples
Lorenz.sys = function(x, t)
{
 c(10 * (x[2] - x[1]),
   28 * x[1] - x[2] - x[1] * x[3],
   -8/3 * x[3] + x[1] * x[2])
}
system.time(integrate_sys(Lorenz.sys, rep(1, 3), 1e2, obs = null_rec))
x = integrate_sys(Lorenz.sys, rep(1, 3), 100, 0.01)
plot(x[, c(2, 4)], type = 'l', col = "steelblue")

## End(Not run)
```

---

odeintr *Odeintr: Fast and Flexible Integration of Ordinary Differential Equations*

Description

This package is a light-weight wrapper around the Boost ODEINT library. It allows one to specify an ODE system in a few lines of C++. This code is inserted into a template that is compiled. The resulting Rcpp function will integrate the system.

Details

You can also specify the model as an R function. Unlike existing packages, you can also supply an observer function that can return arbitrary data structures.

The main function is `compile_sys`, which takes a snippet of C++ code calculating dervatives and compiles an integrator function.

The function `integrate_sys` accepts an R function defining the system and an observer function to record the output in a data frame or list.

Author(s)

Timothy H. Keitt
http://www.keittlab.org/

Timothy H. Keitt <tkeitt@gmail.com>

References

https://github.com/thk686/odeintr, https://headmyshoulder.github.io/odeint-v2/

`set_optimization` *Set compiler optimization*

Description

Write a user Makevars with updated optimization level

Usage

```
set_optimization(level = 3, omit.debug = FALSE, disable.asserts = FALSE,
  suppress.warnings = FALSE, overwrite = FALSE)

show.Makevars()

rm.Makevars()
```

Arguments

`level`	the compiler optimization level (-O<level>)
`omit.debug`	if true, remove "-g" from flags
`disable.asserts`	if true, set NDEBUG define
`suppress.warnings`	if true, suppress compiler warnings
`overwrite`	if true, overwrite existing Makevars file

Details

This function will change the optimization flags used when compiling code. It will write the file "Makevars" to the ".R" directory in your "$HOME" directory. These setting will effect all subsequent compiles, even package installation, unless you remove or edit the "Makevars" file.

This function assumes that your compiler uses "-O" to indicate optimization level and "-g" to indicate that the compiler should issue debugging symbols.

Please let me know if this function fails for your compiler. (Submit and issue on GitHub.)

Note

Don't go overboard. Levels greater than 3 can be hazardous to numerical accuracy. Some packages will not compile or will give inaccurate results for levels above 2.

A very similar function exists in the RStan package.

Author(s)

Timothy H. Keitt

See Also

compile_sys

Index