Package 'ast2ast'

June 26, 2024

Type Package

Title Translates an R Function to a C++ Function

Version 0.4

Date 2024-06-06

Author Krämer Konrad [aut, cre]

Maintainer Krämer Konrad <konrad_kraemer@yahoo.de>

BugReports https://github.com/Konrad1991/ast2ast

URL https://github.com/Konrad1991/ast2ast

Description Enable translation of a tiny subset of R to C++. The user has to define a R function which gets translated. For a full list of possible functions check the documentation. After translation an R function is returned which is a shallow wrapper around the C++ code. Alternatively an external pointer to the C++ function is returned to the user. The intention of the package is to generate fast functions which can be used as ode-system or during optimization.

License GPL-2

Imports Rcpp (>= 1.0.4), R6, methods, dfdr, rlang, RcppArmadillo

LinkingTo Rcpp, RcppArmadillo

VignetteBuilder knitr

Suggests knitr, kableExtra, rmarkdown, tinytest, microbenchmark, ggplot2, RcppXPtrUtils

Encoding UTF-8

RoxygenNote 7.2.1

NeedsCompilation yes

R topics documented:

J *Calculates the jacobian function and translatesthe resulting function into a C++ function.*

Description

An R function is translated to C++ source code and afterwards the code is compiled.
The result can be an external pointer (*XPtr*) or an *R* function.
The default value is an R function.
Further information can be found in the vignette: *Detailed Documentation*.

Usage

```
J(
  f,
  y,
  x,
  output = "R",
  types_of_args = "SEXP",
  return_type = "SEXP",
  reference = FALSE,
  verbose = FALSE,
  getsource = FALSE
)
```

Arguments

f	The function which should be translated from R to C++.
y	The variables to compute the derivatives of (the dependent variable). For example: df/dx
x	The variables to which respect the variables are calcualted (the independent variable). For example: df/dx
output	If set to "R"" an R function wrapping the C++ code is returned. If output is set to "XPtr"" an external pointer object pointing to the C++ code is returned. The default value is "R".
types_of_args	define the types of the arguments passed to the function as an character vector. This is an optional input if using "XPtr" as output. The default value is "SEXP" as this is the only possibility for output "R". In case one want to use an external pointer the easiest way is to pass "sexp" for types_of_args. Beyond that it is possible to pass "double", "ptr_vec" and "ptr_mat". For more information see below for details and check the vignette *InformationForPackageAuthors*. **Beyond that, be aware that the passed *SEXP* objects are only copied if the object size increases. Thus, R objects can be modified within**

the function! See in section details for an example

`return_type`	is a character defining the type which the function returns. The default value is "SEXP"" as this is the only possibility for output "R". Additionally, the possibilities "sexp" and "void" exist for the external pointer interface.
`reference`	If set to TRUE the arguments are passed by reference (not possible if output is "R").
`verbose`	If set to TRUE the output of the compilation process is printed.
`getsource`	If set to TRUE the function is not compiled and instead the C++ source code itself is returned.

Details

The types *numeric vector* and *numeric matrix* are supported. Notably, it is possible that the variables change the type within the function.
Beyond that, be aware that the passed *SEXP* objects are only copied if the size increases. Thus, R objects can be modified within the function!
For example in the following code the variable *a* contains *1, 2, and 3* before the function call and afterwards *1, 1 and 1*. In contrast for variable *b* the size changes and thus the object within *R* is not modified. Furthermore, the variable *c* is not increased and only the first element is changed.

```
f <- function(a, b, c) {
  a[c(1, 2, 3)] <- 1
  b <- vector(10)
  c <- vector(1)
}
fcpp <- ast2ast::translate(f)
a <- c(1, 2, 3)
b <- c(1, 2, 3)
c <- c(1, 2, 3)
fcpp(a, b,c)
print(a)
print(b)
print(c)
```

It is possible to declare a variable of a scalar numeric data type. This is done by adding *_db* to the end of the variable. Each time *_db* is found the variable is declared as a scalar numeric data type. In this case the object cannot change its type! In the example below the variable *a_db* is of type double whereas *b* is of type "sexp".

```
f <- function() {
  a_db = 3.14
  b = 3.14
}
fcpp <- ast2ast::translate(f, verbose = TRUE)
fcpp()
```

In R every object is under the hood a *SEXP* object. In case an *R* function is created as output only *SEXP* elements can be passed to the function. Furthermore, these functions always return a *SEXP* element. Even if nothing is returned; in this case *NULL* is returned!. Notably, is that only numeric vectors (in R also scalar values are vectors) or numeric matrices can be passed to the function.
In contrast if an external pointer is created other types can be specified which are passed to the function or returned from it. The default value is a variable of type *sexp*. This is the data type which is used in the C++ code. The *ptr_vec* and *ptr_mat* interface work in a different way. If using *ptr_vec* a *double** pointer is expected as first element. Additionally a second argument is needed which is of type *int* and which defines the size of the array. This works in the same way for *ptr_mat*. But instead of the size argument two integers are needed which define the number of rows and columns. Both arguments have to be of type *int*. Notably, the memory is only borrowed. Thus, the memory is not automatically deleted! See vignette *InformationForPackageAuthors* for more information.
The following functions are supported:

1. assignment: = and <-
2. allocation: vector and matrix
3. information about objects: length and dim
4. Basic operations: +, -, *, /
5. Indices: *'[]'*. **The function 'at' cannot be used! Beyond that only integer values are allowed within the brackets.**
6. mathematical functions: sin, asin, sinh, cos, acos, cosh, tan, atan, tanh, sqrt, log, ^ and exp
7. concatenate objects: c
8. control flow: for, if, else if, else
9. comparison: ==, !=, >, <, >= and <=
10. printing: print
11. returning objects: return
12. catmull-rome spline: cmr
13. to get a range of numbers the ':' function can be used
14. is.na and is.infinite can be used to test for NA and Inf.

For more details see: `dfdr::jacobian()`

Value

If output is set to *R* an R function is returned. Thus, the C++ code can directly be called within R. In contrast a function which returns an external pointer is generated if the output is set to *XPtr*.

Examples

```
# Further examples can be found in the vignettes.
 ## Not run:
     # simple example
     f <- function(y) {
       ydot <- vector(length = 2)
```

```
  a <- 1.1
  b <- 0.4
  c <- 0.1
  d <- 0.4
  ydot[1] <- y[1]*a - y[1]*y[2]*b
  ydot[2] <- y[2]*y[1]*c - y[2]*d
  return(ydot)
}
jac <- ast2ast::J(f, ydot, y, verbose = TRUE)
jac(c(10, 11))

## End(Not run)
```

`translate` *Translates an R function into a C++ function.*

Description

An R function is translated to C++ source code and afterwards the code is compiled.
The result can be an external pointer (*XPtr*) or an *R* function.
The default value is an R function.
Further information can be found in the vignette: *Detailed Documentation*.

Usage

```
translate(
  f,
  output = "R",
  types_of_args = "double",
  data_structures = "vector",
  handle_inputs = "copy",
  references = FALSE,
  verbose = FALSE,
  getsource = FALSE
)
```

Arguments

`f`	The function which should be translated from R to C++.
`output`	If set to 'R' an R function wrapping the C++ code is returned. If output is set to 'XPtr' an external pointer object pointing to the C++ code is returned. The default value is 'R'.
`types_of_args`	define the types of the arguments passed to the function as an character vector. The character vector should be either of length 1 or has the same length as the number of arguments to the function.

In case the output is set to 'R' 'logical', 'int' or 'double' are available.
If the 'XPtr' interface is used additionally 'const logical', 'const int' and 'const double' can be chosen.
For more information see below for details and check the vignette *Information-ForPackageAuthors*.

`data_structures`
: defines the data structures of the arguments passed to the function (as an character vector).
The character vector should be either of length 1 or has the same length as the number of arguments to the function.
In case the output is set to 'R' one can chose between 'scalar' and 'vector'.
If the output is set to 'XPtr' one can set a data structure to 'scalar', 'vector' or 'borrow.

`handle_inputs`
: defines how the arguments to the function should be handled as character vector.
The character vector should be either of length 1 or has the same length as the number of arguments to the function.
In case the output is an R function the arguments can be either copied ('copy') or borrowed ('borrow').
If you chose borrow R objects which are passed to the function are modified.
This is in contrast to the usual behaviour of R.
If the output is an XPtr the arguments can be only borrowed ('borrow').
In case only part of the arguments should be borrowed than an empty string ("") can be used to indicate this.

`references`
: defines whether the arguments are passed by reference or whether they are copied. This is indicated by a logical vector.
The logical vector should be either of length 1 or has the same length as the number of arguments to the function.
If set to TRUE the arguments are passed by reference otherwise not. This option can be only used when the output is set to 'XPtr'

`verbose`
: If set to TRUE the output of the compilation process is printed.

`getsource`
: If set to TRUE the function is not compiled and instead the C++ source code itself is returned.

Details

Type system: Each variable has a fixed type in a C++ program.
In *ast2ast* the default type for each variable is a data structure called 'vector'.
Each object in 'vector' is as default of type 'double'. Notably, it is defined at runtime whether a variable is a 'vector' in the sense of on R vector or it is a matrix.

Types of arguments to function: The types of the arguments to the function are set together of:

1. types_of_args; c("int", "int")

2. data_structures; c("vector", "scalar")
3. handle_inputs; c("borrow", "")
4. references; c(TRUE, TRUE)

In this example this results in:

```
f(etr::Vec<int>& argumentNr1Input, int& argumentNr2) {
  etr::Vec<int, etr::Borrow<int>> argumentNr1(argumentNr1Input.d.p,
                                              argumentNr1Input.size());
  ... rest of function code
}
```

Types within the function: As mentioned above the default type is a 'vector' containing 'doubles'
Additionally, it is possible to set specific types for a variable.
However, the type cannot be changed if once defined. It is possible to define the following types:

1. logical
2. int
3. double
4. logical_vector
5. int_vector
6. double_vector

The first three mentioned types are scalar types.
These types cannot be resized. Meaning that the behave like a vector of length 1,
which cannot be extended to have more elements. Notably, the scalar values cannot be subsetted.
The advantage is that scalar values need less memory.

declare variable with type: The variables are declared with the type by using the '::' operator. Here are some examples:

```
f <- function() {
  d::double <- 3.14
  l::logical <- TRUE
  dv::int_vector <- vector(mode = "integer", length = 2)
}
```

Borrowing: As mentioned above it is possible to borrow arguments to a function.
Thus, R objects can be modified within the function.
Please be aware that it is not possible to resize the borrowed variable,
Therefore, the code below throws an error. Here an example:

```
f <- function(a, b, c) {
  a[c(1, 2, 3)] <- 1
  b <- vector(length = 10)
  c <- vector(length = 1)
}
fcpp <- ast2ast::translate(f, handle_inputs = "borrow")
a <- b <- c <- c(1, 2, 3)
fcpp(a, b,c)
```

The following functions are supported::

1. assignment: = and <-
2. allocation: vector, matrix and rep
3. information about objects: length and dim
4. Basic operations: +, -, *, /
5. Indices: *'[]'* and at
6. mathematical functions: sin, asin, sinh, cos, acos, cosh, tan, atan, tanh, sqrt, log, ^ and exp
7. concatenate objects: c
8. control flow: for, if, else if, else
9. comparison: ==, !=, >, <, >= and <=
10. printing: print
11. returning objects: return
12. catmull-rome spline: cmr
13. to get a range of numbers the ':' function can be used
14. is.na and is.infinite can be used to test for NA and Inf.

Some details about the implemented functions:

- For indices squared brackets *'[]'* can be used as common in R. Beyond that the function 'at' exists which accepts as first argument a variable and as the second argument you pass the desired index. The caveat of using 'at' is that only **one** entry can be accessed. The function *'[]'* can return more then one element.
  The *at*-function returns a reference to the vector entry. Therefore variable[index] can behave differently then *at(variable, index)*. If only integers are found within *'[]'* the function *at* is used at the right side of an assignment operator (=). The *at*-function can also be used on the left side of an assignment operator. However, in this case only *at* should be used at the right side. Otherwise the results are wrong.
  Here is a small example presented how to use the subset functions:

```
f <- function() {
  a <- c(1, 2, 3)
  print(at(a, 1))
  print(a[1:2])
 }
fcpp <- ast2ast::translate(f)
fcpp()
```

- For- and while-loops can be written as common in R
  - Nr.1

```
for(index in variable){
# do whatever
}
```

  - Nr.2

```
for(index in 1:length(variable){
# do whatever
}
```

- The print function accepts either a scalar, vector, matrix, string, bool or nothing (empty line).
- In order to return an object use the *return* function (**The last object is not returned automatically as in R**).
- In order to interpolate values the *cmr* function can be used. The function needs three arguments.
  1. the first argument is the point of the independent variable (**x**) for which the dependent variable should be calculated (**y**). This has to be a vector of length one.
  2. the second argument is a vector defining the points of the independent variable (**x**). This has to be a vector of at least length four.
  3. the third argument is a vector defining the points of the dependent variable (**y**). This has to be a vector of at least length four.

  Be aware that the R code is translated to ETR an expression template library which tries to mimic R.
  However, it does not behave exactly like R! Please check your compiled function before using it in a serious project.
  If you want to see how *ast2ast* differs from R in detail check the vignette: *Detailed Documentation*.
  In case you want to know how *ast2ast* works in detail check the vignette: *Information-ForPackageAuthors*.

Value

If output is set to *R* an R function is returned. Thus, the C++ code can directly be called within R. In contrast a function which returns an external pointer is generated if the output is set to *XPtr*.

Examples

```
# Further examples can be found in the vignettes.
   ## Not run:
     f <- function() {
       print("Hello World!")
     }
     fcpp <- ast2ast::translate(f)
     fcpp()

     # Translating to external pointer
     # ------------------------------------------------------------------------
     f <- function() {
       print("Hello World!")
     }
     pointer_to_f_cpp <- ast2ast::translate(f,
                                            output = "XPtr", verbose = TRUE
     )

     Rcpp::sourceCpp(code = "
                     #include <Rcpp.h>
                     typedef void (*fp)();
```

```
                // [[Rcpp::export]]
                void call_fct(Rcpp::XPtr<fp> inp) {
                  fp f = *inp;
                  f(); } ")

call_fct(pointer_to_f_cpp)

# Run sum example:
# ============================================================================

# R version of run sum
# ----------------------------------------------------------------------------
run_sum <- function(x, n) {
  sz <- length(x)

  ov <- vector(mode = "numeric", length = sz)

  ov[n] <- sum(x[1:n])
  for (i in (n + 1):sz) {
    ov[i] <- ov[i - 1] + x[i] - x[i - n]
  }

  ov[1:(n - 1)] <- NA

  return(ov)
}

# translated Version of R function
# ----------------------------------------------------------------------------
run_sum_fast <- function(x, n) {
  sz <- length(x)
  ov <- vector(mode = "numeric", length = sz)

  sum_db <- 0
  for (i in 1:n) {
    sum_db <- sum_db + at(x, i)
  }
  ov[n] <- sum_db

  for (i in (n + 1):sz) {
    ov[i] <- at(ov, i - 1) + at(x, i) - at(x, i - at(n, 1))
  }

  ov[1:(n - 1)] <- NA

  return(ov)
}
run_sum_cpp <- ast2ast::translate(run_sum_fast, verbose = FALSE)
set.seed(42)
x <- rnorm(10000)
n <- 500
one <- run_sum(x, n)
```

```
two <- run_sum_cpp(x, n)

## End(Not run)
```

Index