Detailed Documentation

Konrad Krämer

Documentation

The package *ast2ast* translates a R function into a C++ function and returns an external pointer (XPtr) to this function. The scope of *ast2ast* is to generate functions which can be used during solving ode-systems (derivative function or jacobian function) or during optimization. More generally the translated function can be used in fields where it is necessary to evaluate a function very often. Especially if the function is evaluated by C++ the generated external pointer is very sufficient.
First of all the supported objects and functions listed below are explained in detail. Here the arguments which have to be passed to the functions are described and it is explained what the function returns. Furthermore, for each function a small example is given showing how to use it. Moreover, it is explained how the function differ from R equivalents. If other differences were detected please report it.
The last section of the documentation describes how the external pointers produces by *ast2ast* can be used in packages. This information is intended for package authors who want to use *ast2ast* in order to enable a simple user interface as it not necessary anymore for the user to write C++. This paragraph explains how the examples in this vignette are executed. First of all a Rcpp function is created, which executes the output of the function *translate*. If you want to know how this functions work in detail you can go to [Information for Package authors][Information for Package authors]. In the examples only the R code is shown in order to show how to write the code. Sometimes the result of the executed code is also presented.

Supported objects:

- vectors (containing numbers)
- matrices (containing numbers)

Supported functions:

- assignment: = and <-
- allocation: vector and matrix
- information about objects: length and dim
- Basic operations: +, -, *, /
- Indices: [] and at

- mathematical functions: sin, asin, sinh, cos, acos, cosh, tan, atan, tanh, log, ^ and exp
- concatenate objects: c
- control flow: for, if, else if, else
- comparison: ==, !=, >, <, >= and <=
- printing: print
- returning objects: return

Objects

There exists two containers which can be used in *ast2ast* functions. Both containers can only hold the *numeric* type of R (is equivalent to double). The first container are vectors and the second one are matrices. **It is possible to declare a variable of a scalar numeric data type. This is done by adding '_db' (e.g. varname_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!**

Nice features:

- The variables can change the type within a function. This is normally not possible when using C++.
- The index of vectors and matrices starts at 1 as in R.
- The index has to be in the boundaries of the vector or matrix. Even though this is different from the behaviour in R it is a nice feature. If you access a element outside the boundaries of a vector in R *NA* is returned. This is in 99% of the cases not what the user wants.
- The memory of the matrices is columnwise arranged as in R.
- In R arguments passed to a function are always copied. In *ast2ast* functions it is possible to pass only the memory address of a object (called a reference). To do this you have to set the *reference* parameter of the *translate* function to *TRUE*. If you pass a function by reference you can modify the object without returning it (see Example 1). In the Rcpp function the variable x is printed before and after the call of the function *fetr*. Notably, if no *return* is used in the R code translated by *ast2ast* nothing is returnd (in R the last object is returned in this case). You see that x is 10 before the call of the function and it is 1 after the call of the function. But the function does not return anything. Thus, the object *x* is modified in the function without copying it.

Example 1

```
f <- function(variable) {
  variable <- 1
}
library(ast2ast)
fetr <- translate(f)
x <- 10
output <- byref(fetr, x)
```

```
## x before call of function:
## 10
## x after call of function:
## 1
```

```
output
```

```
## NULL
```

Caveats:

- Sometimes large overhead of the containers
  - Variables which are scalars are represented as vectors of length 1. This is also how R handels scalar variables. As in C++ scalar variables are not defined as vectors the speed of the translated R function can be substantially lower compared to a native C++ function.

Variable declaration

In **Example 2** the various ways of declaring variables are presented. To assign a value to a variable you can use <- or =. As already mentioned only numeric values are possible. If you want to assign a vector you can use either the *c* or *vector* function. The *c* function works in the same way as R and can handle any combinations of scalars, vectors or matrices. The function *vector* differs in two ways from the R equivalent. First of all you cannot use terms such as *vector(length = size)* as this is not possible in C++. In contrast you just write *vector(size)*. The R function *rep* is not available in *ast2ast* but it is possible to write *vector(value, size)* which in R would be written as *rep(value, size)*. A third way to use the *vector* function is to pass another vector and the size to it *vector(other_vector, size)*. The *matrix* function works in the same way as the *vector* function with the difference that instead of the size two arguments were needed the number of rows and the number of columns.

Example 2

```
f <- function() {
  a <- 1
  a_db <- 3.14
  b = 2
  c <- c(1, 2, 3)
  d = vector(2)
  e <- vector(3.14, 4)
  f <- vector(c, 3)
  g <- matrix(2, 2)
  h <- matrix(6, 2, 2)
  i <- matrix(e, 2, 2)

  print("a")
  print(a)
  print(a_db)
  print()
  print("b")
  print(b)
  print()
  print("c")
  print(c)
  print()
  print("d")
  print(d)
  print()
  print("e")
  print(e)
  print()
  print("f")
  print(f)
  print()
  print("g")
  print(g)
  print()
  print("h")
  print(h)
  print()
  print("i")
  print(i)
  print()
```

```r
}
library(ast2ast)
fetr <- translate(f)
vardec(fetr)
```

```
## a
## 1
## 3.14
##
## b
## 2
##
## c
## 1
## 2
## 3
##
## d
## 2
## 2
##
## e
## 3.14
## 3.14
## 3.14
## 3.14
##
## f
## 1
## 1
## 1
##
## g
## 4.64629e-310 1.27246e+232
## 0    2.05586
##
## h
## 6    6
## 6    6
##
## i
## 3.14 3.14
## 3.14 3.14
```

Basic arithmetics

As usual in R it is possible to use basic arithmetic operations on scalars, vectors and matrices (**Example 3**).

Example 3

```r
f <- function() {

a <- 2
b <- 3
```

```r
print("scalar operations")
print(a + b)
print(a - b)
print(a / b)
print(a * b)

print()

print("vector & scalar operations")
a <- c(1, 2, 3)
b <- 4
print(a + b)
print(b - a)

print()

print("vector & vector operations (same length)")
a <- 6:8
b <- 1:3
print(a / b)
a <- 1:6
b <- 1:3
print(a / b)
print("vector & vector operations (different length)")
print("longer object length is a multiple of shorter object length")
a <- 1:6
b <- 1:3
print(a / b)
print("longer object length is not a multiple of shorter object length")
a <- 1:5
b <- 1:3
print(a / b) # different to R no warning

print()

print("matrix & scalar operations")
a <- 3
b <- matrix(3, 2, 2)
print(a*b)
print(b + 4)

print()

print("matrix & vector operations")
a <- 5:6
b <- matrix(3, 2, 2)
print(b - a)
print(a / b)

print()

print("matrix & matrix operations")
a <- matrix(3, 2, 2)
```

```r
b <- matrix(4, 2, 1) # difference to R!
print(a + b)

print()

print("mixed operations")
a <- 1
b <- 2:5
c <- matrix(50, 2, 2)
d <- a + b - c/2
print(d)
}

library(ast2ast)
fetr <- translate(f)
call_fct(fetr)
```

```
## scalar operations
## 5
## -1
## 0.666667
## 6
##
## vector & scalar operations
## 5
## 6
## 7
## 3
## 2
## 1
##
## vector & vector operations (same length)
## 6
## 3.5
## 2.66667
## 1
## 1
## 1
## 4
## 2.5
## 2
## vector & vector operations (different length)
## longer object length is a multiple of shorter object length
## 1
## 1
## 1
## 4
## 2.5
## 2
## longer object length is not a multiple of shorter object length
## 1
## 1
## 1
## 4
```

```
## 2.5
##
## matrix & scalar operations
## 9    9
## 9    9
## 7    7
## 7    7
##
## matrix & vector operations
## -2   -2
## -3   -3
## 1.66667  1.66667
## 2    2
##
## matrix & matrix operations
## 7    7
## 7    7
##
## mixed operations
## -22  -20
## -21  -19
```

Subsetting

If you want to subset a vector or a matrix object you can use either *[]* or the *at* function. The *[]* is slower then *at* but more powerful (**Example 4**).

The following objects can be passed to *[]* when using a vector or matrix:

- nothing
- numeric scalar
- logical
- vector
- matrix
- result of comparison
- **caveat:**
- **it is not possible to pass the results of calculations!**

In case of a matrix it is possible to pass one of the above objects to access specific rows or columns respectively ([rows, cols]). In contrast to *at* only a scalar can be passed. Thus, only a single element is accessed by this function! However, this function works faster. The result of *at* cannot be subsetted further.

Example 4

```
f <- function() {

print("pass nothing")
a <- 1:8
print(a)
a[] <- 100
print(a)
print()
```

```r
print("pass logical")
a <- 1:8
print(a)
a[TRUE] <- 100
print(a)
print()

print("pass scalar")
a <- 1:8
print(a)
a[1] <- 100
print(a)
print()


print("pass vector")
a <- 1:8
b <- 2:5
print(a)
a[b] <- 100
print(a)
print()

print("pass result of ==")
a <- 1:8
a[a < 5] <- 100
print(a)
print()


print("pass result of !=")
a <- 1:8
b <- c(1, 2, 3, 0, 0, 0, 0, 8)
a[a != b] <- 100
print(a)
print()


print("pass result of <=")
a <- 1:8
b <- c(1, 2, 3, 0, 0, 0, 0, 8)
a[a <= b] <- 100
print(a)
print()


print("pass result of >=")
a <- 1:8
b <- c(1, 2, 3, 0, 0, 0, 0, 9)
a[a >= b] <- 100
print(a)
print()
```

```r
print("pass result of >")
a <- 1:8
b <- c(0, 2, 3, 0, 0, 0, 0, 9)
a[a > b] <- 100
print(a)
print()


print("pass result of <")
a <- 1:8
b <- c(0, 2, 3, 0, 0, 0, 0, 9)
a[a < b] <- 100
print(a)
print()


print("pass scalar, scalar")
a <- matrix(3, 4, 4)
a[1, 1] <- 100
print(a)
print()


print("pass vector, vector")
a <- matrix(3, 4, 4)
b <- c(1, 3)
c <- c(2, 4)
a[b, c] <- 100
print(a)
print()


print("pass ==, >=")
a <- matrix(1:16, 4, 4)
b <- 1:4
c <- c(1, 8, 3, 8)
a[b == c, b >= c] <- 100
print(a)
print()


print("at")
a <- 1:16
at(a, 2) <- 100
print(a)
print()


print("at")
a <- matrix(1:16, 4, 4)
at(a, 1, 4) <- 100
print(a)
print()
```

```r
}

library(ast2ast)
fetr <- translate(f)
call_fct(fetr)
```

```
## pass nothing
## 1
## 2
## 3
## 4
## 5
## 6
## 7
## 8
## 100
## 100
## 100
## 100
## 100
## 100
## 100
## 100
##
## pass logical
## 1
## 2
## 3
## 4
## 5
## 6
## 7
## 8
## 100
## 100
## 100
## 100
## 100
## 100
## 100
## 100
##
## pass scalar
## 1
## 2
## 3
## 4
## 5
## 6
## 7
## 8
## 100
## 2
## 3
```

```
## 4
## 5
## 6
## 7
## 8
##
## pass vector
## 1
## 2
## 3
## 4
## 5
## 6
## 7
## 8
## 1
## 100
## 100
## 100
## 100
## 6
## 7
## 8
##
## pass result of ==
## 100
## 100
## 100
## 100
## 5
## 6
## 7
## 8
##
## pass result of !=
## 1
## 2
## 3
## 100
## 100
## 100
## 100
## 8
##
## pass result of <=
## 100
## 100
## 100
## 4
## 5
## 6
## 7
## 100
##
```

```
## pass result of >=
## 100
## 100
## 100
## 100
## 100
## 100
## 100
## 8
##
## pass result of >
## 100
## 2
## 3
## 100
## 100
## 100
## 100
## 8
##
## pass result of <
## 1
## 2
## 3
## 4
## 5
## 6
## 7
## 100
##
## pass scalar, scalar
## 100  3   3   3
## 3    3   3   3
## 3    3   3   3
## 3    3   3   3
##
## pass vector, vector
## 3    100 3   100
## 3    3   3   3
## 3    100 3   100
## 3    3   3   3
##
## pass ==, >=
## 100  5   100 13
## 2    6   10  14
## 100  7   100 15
## 4    8   12  16
##
## at
## 1
## 100
## 3
## 4
## 5
```

```
## 6
## 7
## 8
## 9
## 10
## 11
## 12
## 13
## 14
## 15
## 16
##
## at
## 1    5   9   100
## 2    6   10  14
## 3    7   11  15
## 4    8   12  16
```

Helper functions

There exists three helper function. The *length* function returns the number of elements of a vector or matrix. The *dim* function returns the number of rows and columns of a matrix. The *:* function can be used to create a range of numbers. See **Example 5** in order to see how the functions work.

Example 5

```
f <- function() {
  a <- 1:4
  print(a)
  a <- 1.1:5.2
  print(a)

  a <- 1:16
  print(length(a))

  b <- matrix(1:4, 2, 2)
  print(dim(b))
}

library(ast2ast)
fetr <- translate(f)
call_fct(fetr)
```

```
## 1
## 2
## 3
## 4
## 1.1
## 2.1
## 3.1
## 4.1
## 5.1
## 16
## 2
## 2
```

Comparison functions

As usual in R it is possible to compare two objects using one of the following options:

- ==
- <=
- >=
- !=
- <
- >

Control flow

It is possible to write for loops and if, else if, else branches as usual in R (&&, || are not implemented yet):

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

Printing

Using the function print as common in R:

- print() is a difference to R
- print("string")
- print(logical)
- print(scalar)
- print(vector) is different to R
- print(matrix)

Math functions

Following mathematical functions are available:

- sin
- asin
- sinh
- cos
- acos
- cosh
- tan
- atan
- tanh
- log
- ^ and exp

Interpolation

In order to interpolate values the 'cmr' function can be used. The function needs three arguments:

- 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.
- 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.

- 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.

Information for Package authors (not written yet)

- the sexp object
- VEC class

Rcpp Interface

- NumericVector = sexp
- sexp = NumericVector
- NumericMatrix = sexp_mat
- sexp_mat = NumericMatrix

RcppArmadillo Interface

- vec = sexp
- sexp = vec
- mat = sexp_mat
- sexp_mat = mat

Pointer Interface

- sexp(size, pointer, integer) –> integer = 0, 1 or 2 = copy, take ownership, borrow
- sexp(rows, cols, pointer, integer) –> integer = 0, 1 or 2 = copy, take ownership, borrow
- modify NumericVector
- modify NumericMatrix
- modify arma::vec
- modify arma::mat
- modify std::vector

Examples

How to use *ast2ast*

R code is translated

```
add_two <- function(a) {
  return(a + 2)
}
pointer_to_f_cpp <- ast2ast::translate(add_two)
```

Defining C++ Code which uses the translated R Code

```
// [[Rcpp::depends(ast2ast)]]
// [[Rcpp::depends(RcppArmadillo)]]
#include "etr.hpp"

// [[Rcpp::plugins("cpp17")]]

typedef sexp (*fp)(sexp a);

// [[Rcpp::export]]
void call_fct(Rcpp::XPtr<fp> inp) {
```

```
  fp f = *inp;
  sexp a = coca(1, 2, 3);
  print(a);
  a = f(a);
  print("a is now:");
  print(a);
}
```

Calling the final function

```r
#Rcpp::sourceCpp("cpp_code.cpp") # if not run in Rmarkdown
call_fct(pointer_to_f_cpp)
```

```
## 1
## 2
## 3
## a is now:
## 3
## 4
## 5
```

r2sundials

r2sundials code

```r
setwd("/home/konrad/Documents/0Uni/programming")
#install.packages("ast2ast", type = "source", repos = NULL)

library(Rcpp)
library(ast2ast)
library(r2sundials)
```

```
## Loading required package: rmumps
```

```r
library(RcppXPtrUtils)
library(microbenchmark)


# R version
ti <- seq(0, 5, length.out=101)
p <- list(a = 2)
p <- c(nu = 2, a = 1)
y0 <- 0
frhs <- function(t, y, p, psens) -p["nu"]*(y-p["a"])

res_exp <- r2sundials::r2cvodes(y0, ti, frhs, param = p)
attributes(res_exp) <- NULL
#stopifnot(diff(range(1-exp(-p$a*ti) - res_exp)) < 1.e-6)


# External pointer
ptr_exp <- cppXPtr(code = '
int rhs_exp(double t, const vec &y, vec &ydot, RObject &param, NumericVector &psens) {
  double a = 1;
```

```
  double nu = 2;
  ydot[0] = -nu*(y[0] - a);
  return(CV_SUCCESS);
}
', depends=c("RcppArmadillo","r2sundials","rmumps"),
 includes="using namespace arma;\n#include <r2sundials.h>", cacheDir="lib", verbose=FALSE)
pv <- c(a = 1)
res_exp2 <- r2sundials::r2cvodes(y0, ti, ptr_exp, param = pv)
attributes(res_exp2) <- NULL
```

Wrapper code in order to call translated R function (maybe written by package authors)

```
// [[Rcpp::depends(RcppArmadillo)]]
// [[Rcpp::depends(rmumps)]]
// [[Rcpp::depends(r2sundials)]]
// [[Rcpp::depends(ast2ast)]]
// [[Rcpp::plugins("cpp17")]]
#include "etr.hpp"
#include "RcppArmadillo.h"
#include "r2sundials.h"
using namespace arma;

typedef int (*fp)(double t, const vec &y, vec &ydot, RObject &param, NumericVector &psens);

typedef void (*user_fct)(sexp& y_, sexp& ydot_);
user_fct Fct;

int rhs_exp_wrapper(double t, const vec &y, vec &ydot, RObject &param, NumericVector &psens) {
  NumericVector p(param);
  const int size = y.size();
  sexp ydot_(size, ydot.memptr(), 2);
  double* ptr = const_cast<double*>(y.memptr());
  sexp y_(size, ptr, 2);
  Fct(y_, ydot_);
  return(CV_SUCCESS);
}


// [[Rcpp::export]]
Rcpp::NumericVector solve_ode(Rcpp::XPtr<user_fct> inp, NumericVector time, NumericVector y) {
  Fct = *inp;
  Rcpp::XPtr<fp> ptr = Rcpp::XPtr<fp>(new fp(&rhs_exp_wrapper));

  Rcpp::Environment pkg = Rcpp::Environment::namespace_env("r2sundials");
  Rcpp::Function solve = pkg["r2cvodes"];

  Rcpp::NumericVector output = solve(y, time, ptr, time);

  return output;
}
```

Applying *ast2ast*

The speed of pure C++ is not reached. However, the *ast2ast* function is substantially fastern then the R code.

```r
# ast2ast version
ti <- seq(0, 5, length.out=101)
p <- list(a = 2)
p <- c(nu = 2, a = 1)
y0 <- 0

ode <- function(y, ydot) {
  nu_db <- 2
  a_db <- 1
  at(ydot, 1) <- -nu_db*(at(y,1) - a_db)
}

pointer_to_ode <- ast2ast::translate(ode, reference = TRUE, verbose = FALSE)
res_exp3 <- solve_ode(pointer_to_ode, ti, y0)
attributes(res_exp3) <- NULL

head(res_exp)
```

```
## [1] 0.00000000 0.09516259 0.18126923 0.25918179 0.32968002 0.39346945
```

```r
head(res_exp2)
```

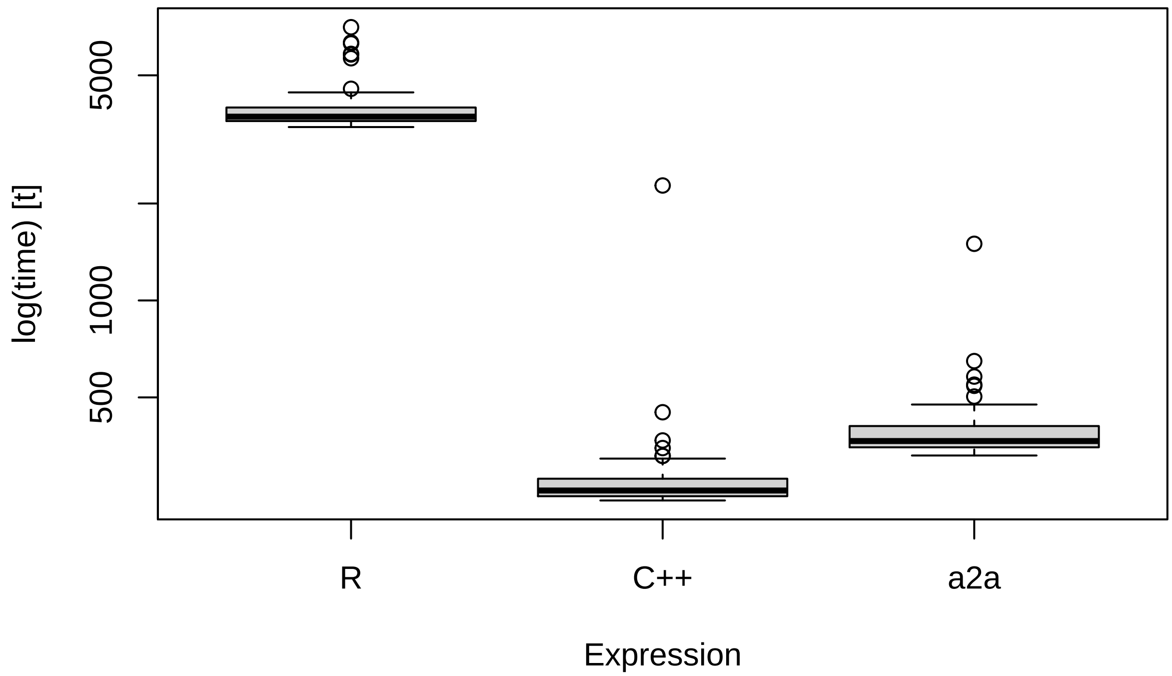
```
## [1] 0.00000000 0.09516259 0.18126923 0.25918179 0.32968002 0.39346945
```

```r
head(res_exp3)
```

```
## [1] 0.00000000 0.09516259 0.18126923 0.25918179 0.32968002 0.39346945
```

```r
out <- microbenchmark(r2cvodes(y0, ti, frhs, param = p), r2cvodes(y0, ti, ptr_exp, param = pv), solve_od

boxplot(out, names = c("R", "C++", "a2a"))
```




```
// [[Rcpp::depends(RcppArmadillo)]]
// [[Rcpp::depends(ast2ast)]]
// [[Rcpp::plugins("cpp17")]]
// [[Rcpp::depends(paropt)]]
#include "etr.hpp"
#include "RcppArmadillo.h"
using namespace Rcpp;

typedef int (*OS)(double &t, std::vector<double> &params, std::vector<double> &states);

int ode_system(double &t, std::vector<double> &params, std::vector<double> & states) {
  double a = params[0];
  double b = params[1];
  double c = params[2];
  double d = params[3];

  double n1 = states[0];
  double n2 = states[1];

  states[0] = n1*c*n2 - n1*d;
  states[1] = n2*a - n2*b*n1;

  return 0;
}

// [[Rcpp::export]]
Rcpp::XPtr<OS> test_optimization() {
  Rcpp::XPtr<OS> xpfun = Rcpp::XPtr<OS>(new OS(&ode_system));
  return xpfun;
}
```

```
typedef int (*paropt_fct)(double &t, std::vector<double> &params, std::vector<double> & states);

typedef void (*user_fct2)(sexp& p, sexp& y);

user_fct2 Fct_paropt;

int ode_system_wrapper(double &t, std::vector<double> &params, std::vector<double> & states) {
  sexp p(params.size(), params.data(), 2);
  sexp s(states.size(), states.data(), 2);
  Fct_paropt(p, s);

  return 0;
}


// [[Rcpp::export]]
Rcpp::List optimize_paropt(Rcpp::XPtr<user_fct2> inp,
                                    NumericVector time, Rcpp::DataFrame lb, Rcpp::DataFrame ub, Rcpp::Da
  Fct_paropt = *inp;
  Rcpp::XPtr<paropt_fct> ptr = Rcpp::XPtr<paropt_fct>(new paropt_fct(&ode_system_wrapper));

  Rcpp::Environment pkg = Rcpp::Environment::namespace_env("paropt");
  Rcpp::Function optim = pkg["optimizer_pointer"];

  Rcpp::NumericVector abs_tols{1e-8, 1e-8};

  Rcpp::List output = optim(time, ptr, 1e-6, abs_tols, lb, ub, states,
                                    40, 1000, 0.0001, "bdf");

  return output;
}
```

```
#states
path <- system.file("examples", package = "paropt")
states <- read.table(paste(path,"/states_LV.txt", sep = ""), header = T)

# parameter
lb <- data.frame(time = 0, a = 0.8, b = 0.3, c = 0.09, d = 0.09)
ub <- data.frame(time = 0, a = 1.3, b = 0.7, c = 0.4, d = 0.7)

# Optimizing
library(paropt)
```

paropt

```
## SUNDIALS - Copyright (c) 2002-2015, Lawrence Livermore National Laboratory.
## Produced at the Lawrence Livermore National Laboratory.
## See - https://computation.llnl.gov/projects/sundials
## PSO: See - https://github.com/kthohr/optim
```

```
set.seed(1)
start_time <- Sys.time()
df_cpp <- optimizer_pointer(integration_times = states$time, ode_sys = test_optimization(),
```

```r
                        relative_tolerance = 1e-6, absolute_tolerances = c(1e-8, 1e-8),
                       lower = lb, upper = ub, states = states,
                       npop = 40, ngen = 1000, error = 0.0001, solvertype = "bdf")
end_time <- Sys.time()
cpp_time <- end_time - start_time



# ast2ast with at and _db
ode <- function(params, states) {
  a_db = at(params, 1)
  b_db = at(params, 2)
  c_db = at(params, 3)
  d_db = at(params, 4)

  n1_db = at(states, 1)
  n2_db = at(states, 2)

  at(states, 1) = n1_db*c_db*n2_db - n1_db*d_db;
  at(states, 2) = n2_db*a_db - n2_db*b_db*n1_db;
}

pointer_to_ode <- ast2ast::translate(ode, reference = TRUE, verbose = FALSE)
set.seed(1)
start_time <- Sys.time()
df_ast2ast <- optimize_paropt(pointer_to_ode, states$time, lb, ub, states)
end_time <- Sys.time()
a2a_time <- end_time - start_time

message("time of the pure C++ code")
```

```
## time of the pure C++ code
```

```r
knitr::kable(cpp_time)
```

x
25.45511 secs

```r
message("parameter results of the pure C++ code")
```

```
## parameter results of the pure C++ code
```

```r
knitr::kable(df_cpp[[8]])
```

0	1.12283	0.4065349	0.0946064	0.3925324

```r
message("time of the ast2ast code")
```

```
## time of the ast2ast code
```

```r
knitr::kable(a2a_time)
```

x
31.88745 secs

```r
message("parameter results of the ast2ast code")
```

```
## parameter results of the ast2ast code
```

```r
knitr::kable(df_ast2ast[[8]])
```

0	1.12283	0.4065349	0.0946064	0.3925324