Automatic Differentiation in R by autodiffr

Changcheng Li, John Nash, Hans Werner Borchers

2018/8/10

Package `autodiffr` provides an R wrapper for Julia packages `ForwardDiff.jl` and `ReverseDiff.jl` through R package `JuliaCall` to do **automatic differentiation** for native R functions. This vignette will walk you through several examples in using `autodiffr`.

Basic usage

When loading the library `autodiffr` it is still necessary to create a process running Julia, to connect to it, to check whether all needed Julia packages are installed resp. to install them in case they are missing. All this is done by the `ad_setup` function. If everything is in place, it will finish in short time, otherwise it may take several minutes (the first time it is called).

```
library(autodiffr)

ad_setup()
## Julia version 0.6.4 at location [...]/julia/bin will be used.
## Loading setup script for JuliaCall...
Finish loading setup script for JuliaCall.
INFO: Precompiling module DiffResults.
INFO: Precompiling module ForwardDiff.
INFO: Precompiling module ReverseDiff.
```

The first time `ad_setup()` is called on your system, these three Julia modules / packages are being precompiled, later on this will not happen again, except when the julia packages get installed anew (e.g., a new version has appeared). Please note that `ad_setup` is required whenever you load the `autodiffr` package into R. And if Julia is not in the path, then `autodiffr` may fail to locate Julia. In this case, use `ad_setup(JULIA_HOME = "the file folder contains the julia binary")`.

Now automatic differentiation is ready to get applied. `autodiffr` has `ad_grad`, `ad_jacobian` and `ad_hessian` to calculate gradient, jacobian and hessian, and has `makeGradFunc`, `makeJacobianFunc` and `makeHessianFunc` to create gradient, jacobian and hessian functions respectively. We can see the basic usage below:

```
## Define a target function with vector input and scalar output
f <- function(x) sum(x^2L)

## Calculate gradient of f at [2,3] by
ad_grad(f, c(2, 3)) ## deriv(f, c(2, 3))
#> [1] 4 6

## Get a gradient function g
g <- makeGradFunc(f)

## Evaluate the gradient function g at [2,3]
g(c(2, 3))
#> [1] 4 6

## Calculate hessian of f at [2,3] by
ad_hessian(f, c(2, 3))
#>      [,1] [,2]
```

```
#> [1,]    2    0
#> [2,]    0    2

## Get a hessian function h
h <- makeHessianFunc(f)

## Evaluate the hessian function h at [2,3]
h(c(2, 3))
#>      [,1] [,2]
#> [1,]    2    0
#> [2,]    0    2

## Define a target function with vector input and vector output
f <- function(x) x^2

## Calculate jacobian of f at [2,3] by
ad_jacobian(f, c(2, 3))
#>      [,1] [,2]
#> [1,]    4    0
#> [2,]    0    6

## Get a jacobian function j
j <- makeJacobianFunc(f)

## Evaluate the gradient function j at [2,3]
j(c(2, 3))
#>      [,1] [,2]
#> [1,]    4    0
#> [2,]    0    6
```

What to do if autodiffr cannot handle your function?

`autodiffr` works by ultilizing R's S3 object system: most of R functions are just compositions of many small operations like `+` and `-`, and these small operations can be dispatched by the S3 object system to Julia, which finishes the real work of automatic differentiation behind the scence. This mechanism can work for many R functions, but it also means that `autodiffr` has same limitations just as R's S3 object system. For example, `autodiffr` can't handle R's internal C code or some external C and Fortran code. In this section, we will talk about how to detect to deal with such problems. We will first talk about the common issues in the next subsection, and then we will see a little example, finally we will show how to use a helper function `ad_variant` provided by `autodiffr` to deal with these common problems at the same time.

Basic R functions that have problems with autodiffr

This is a list of the most common basic R functions that have problems with `autodiffr`:

- `colSums`, `colMeans`, `rowSums`, `rowMeans`: These common functions are all implemented in R's internal C code. And `autodiffr` provides helper functions `cSums`, `cMeans`, `rSums` and `rMeans` to substitute them. Example error message is like "Error in colSums(...) : 'x' must be numeric".
- `%*%`: this function can't be handled with R's S3 system, so `autodiffr` provides `%m%` to use instead. Example error message is like "Error in x %*% y : requires numeric/complex matrix/vector arguments".
- `crossprod`, `tcrossprod`: these functions are also implemented in R's internal C code, users may use `t(x) %m% y` or `x %m% t(y)` instead, or find some other more effective alternatives. Example error

message is like "Error in crossprod(x, y) : requires numeric/complex matrix/vector arguments".

- `diag`: it will also invoke R's internal C code when using `diag` on a vector to create a diagonal matrix and `autodiffr` provides helper functions `diagm` for the same purpose. Example error message is like "Error in diag(x) : long vectors not supported yet: ../../../../R-3.5.1/src/main/array.c:2178".
- `mapply`, `sapply`, `apply`: `autodiffr` provides a helpful function *map*, which can be used similarly to *mapply*.
- `matrix`: it's problematic to use `matrix` to reshape vectors into matrices with `autodiffr`, use R's `array` instead or use the helper function `julia_array` from `autodiffr`. Example error message is like "Error in matrix(x, : 'data' must be of a vector type, was 'environment' ". *`[<-`: if you ever want to do something like `x[i] <- a` or `x[i, j] <- a`, when `x` is a vector or matrix involved in the differentiation process, then it's likely that you hit this problem: "Error in x[i] <- y : incompatible types (from environment to double) in subassignment type fix". To deal with this problem, maybe you need to do `x <- julia_array(x)` first.

A little example to demonstrate the issues

Let us first define a makeup function to show some of the aforementioned issues:

```
fun <- function(x) {
    if (length(x) != 4L) stop("'x' must be a vector of 4 elements.")
    y <- rep(0, 4)
    y[1] <- x[1]; y[2] <- x[2]^2
    y[3] <- x[3]; y[4] <- x[4]^3
    y <- matrix(y, 2, 2)
    det(y)
}
```

If we directly use `autodiffr` to deal with `fun`, we will have:

```
x0 <- c(1.2, 1.4, 1.6, 1.8)

try(ad_grad(fun, x0))

## from the error message
## Error in y[1] <- x[1] :
##  incompatible types (from environment to double) in subassignment type fix
## and the error list in the previous subsection, we can know the problem is from assignment,
## and we can come up with the solution which deal with the issue:

fun1 <- function(x) {
    if (length(x) != 4L) stop("'x' must be a vector of 4 elements.")
    y <- julia_array(rep(0, 4))
    ## we need to use julia_array to wrap the vector
    ## or we can do y <- julia_array(0, 4) which is more elegant.
    y[1] <- x[1]; y[2] <- x[2]^2
    y[3] <- x[3]; y[4] <- x[4]^3
    y <- matrix(y, 2, 2)
    det(y)
}

print(try(ad_grad(fun1, x0)))
#> [1] "Error : Error happens in Julia.\nREvalError: \n"
#> attr(,"class")
#> [1] "try-error"
```

```
#> attr(,"condition")
#> <simpleError: Error happens in Julia.
#> REvalError: >

## and then we have another error,
## from the error message
## Error in matrix(y, 2, 2) :
##   'data' must be of a vector type, was 'environment'
## and the error list in the previous subsection, we can know the problem is from
## using matrix to reshape the vector y,
## and we can come up with the solution which deal with the issue:

fun2 <- function(x) {
    if (length(x) != 4L) stop("'x' must be a vector of 4 elements.")
    ## we need to use julia_array to wrap the vector
    ## or we can do y <- julia_array(0, 4) which is more elegant.
    y <- julia_array(rep(0, 4))
    y[1] <- x[1]; y[2] <- x[2]^2
    y[3] <- x[3]; y[4] <- x[4]^3
    ## we need to use array to reshape the vector,
    ## or we can do y <- julia_array(y, 2, 2).
    y <- array(y, c(2, 2))
    det(y)
}

## This time we will have the correct solution
ad_grad(fun2, x0)
#> [1]  5.832 -4.480 -1.960 11.664

## and we can use numeric approximation to check our result:
numDeriv::grad(fun2, x0)
#> [1]  5.832 -4.480 -1.960 11.664
```

Use `ad_variant` to deal with these issues "automatically"

The above process is a little tedious. Helpfully, `autodiffr` provides a helper function `ad_variant` to deal with these problems simutaneously. And the usage is quite simple:

```
funvariant <- ad_variant(fun)
#> ad_variant will try to wrap your function in a special environment, which redefines some R functions

ad_grad(funvariant, x0)
#> [1]  5.832 -4.480 -1.960 11.664

funvariant1 <- ad_variant(fun, checkArgs = x0)
#> ad_variant will try to wrap your function in a special environment, which redefines some R functions
#> Start checking...
#> assignment in arrays has been replaced to be compatible with arrays in Julia.
#> matrix has been replaced with array.
#> New function gives the same result as the original function at the given arguments. Still need to us
#> Checking finished.

ad_grad(funvariant1, x0)
```

```
#> [1]  5.832 -4.480 -1.960 11.664
```

Chebyquad function

This problem was given prominence in the optimization literature by Fletcher (1965).

First let us define our Chebyquad function. Note that this is for the **vector** x. This version is expressed as the sum of squares of a vector of function values, which provide a nonlinear least squares problem.

```
cyq.res <- function (x) {
# Fletcher's chebyquad function m = n -- residuals
   n<-length(x)
   res<-numeric(n)
   for (i in 1:n) { #loop over resids
     rr<-0.0
     for (k in 1:n) {
    z7<-1.0
    z2<-2.0*x[k]-1.0
        z8<-z2
        j<-1
        while (j<i) {
            z6<-z7
            z7<-z8
            z8<-2*z2*z7-z6 # recurrence to compute Chebyshev polynomial
            j<-j+1
        } # end recurrence loop
        rr<-rr+z8
      } # end loop on k
      rr<-rr/n
      if (2*trunc(i/2) == i) { rr <- rr + 1.0/(i*i - 1) }
      res[i]<-rr
    } # end loop on i
    res
}

cyq.f <- function (x) {
  rv<-cyq.res(x)
  f <- sum(rv^2)
}
```

Let us choose a single value for the number of parameters, and for illustration use $n = 4$.

```
## cyq.setup
n <- 4
x<-1:n
x<-x/(n+1.0) # Initial value suggested by Fletcher
```

For safety, let us check the function and a numerical approximation to the gradient.

```
require(numDeriv)
#> Loading required package: numDeriv
cat("Initial parameters:")
#> Initial parameters:
x
#> [1] 0.2 0.4 0.6 0.8
```

```r
cat("Initial value of the function is ",cyq.f(x),"\n")
#> Initial value of the function is  0.07118393
gn <- numDeriv::grad(cyq.f, x) # using numDeriv
cat("Approximation to gradient at initial point:")
#> Approximation to gradient at initial point:
gn
#> [1]  0.6170624  0.1882112 -0.1882112 -0.6170624
```

We can now try to see if `autodiffr` matches this gradient. However, the following code gives an error in Julia.

Create optimized gradient/jacobian/hessian functions

Bibliography

Fletcher, R. 1965. "Function Minimization Without Calculating Derivatives – a Review." *Computer Journal* 8:33–41.