Symbolic Differentiation of R Code with Deriv Package

RUG meeting in Toulouse

Serguei Sokol, IR INRA / UMR LISBP

25 septembre 2018

Introduction

Existent solutions for symbolic computation

Exact and economic derivative calculation is useful e.g. for optimization codes.

Symbolic computation (including differentiation) is proposed for a long time in various maths software:

- MAPPLE
- Matematica
- SAGE
- MATLAB (dedicated toolbox)
- Python (SymPy)
- ...

In R, we have `stats::deriv()` and various packages `rSymPy`, `SymEngin.R` (not on CRAN), `Ryacas` (no more available)

So why to develop a new package `Deriv`?

Why a new package Deriv?

- Symbolic computation packages often operate on their own data type (not R's variable), e.g.

```
> library(rSymPy)
> x = Var("x")
> sympy("diff(sin(2*x), x, 1)")
## [1] "2*cos(2*x)"
```

Why a new package Deriv?

- deriv() can differentiate R expression e.g.

```
> deriv(~sin(2*x), "x")
## expression({
##     .expr1 <- 2 * x
##     .value <- sin(.expr1)
##     .grad <- array(0, c(length(.value), 1L), list(NULL, c("x"
##     .grad[, "x"] <- cos(.expr1) * 2
##     attr(.value, "gradient") <- .grad
##     .value
## })
```

but ...

Why a new package `Deriv`?

- . . . but not a user's function body:

```
> y = function(x) sin(2*x)
> deriv(~y(x), "x")
## Error in deriv.formula(~y(x), "x"): La fonction 'y' n'es
```

while `Deriv()` can inspect users' functions:

```
> library(Deriv)
> Deriv(~y(x), "x")
## 2 * cos(2 * x)
```

Why a new package `Deriv`?

- `deriv()` return an expression that user must integrate in its code while `Deriv()` return a function that can be called

```
> yd = Deriv(y, "x")
> class(yd)
## [1] "function"
```

```
> yd(pi/2.)
## [1] -2
```

Why a new package Deriv?

- Deriv() rules' table is quite complete but can be extended by user's custom derivative rules

```
> deriv(~besselI(x, 1), "x")
## Error in deriv.formula(~besselI(x, 1), "x"): La fonctio
```

```
> Deriv(~besselI(x, 1), "x")
## 0.5 * (besselI(x, 0) + besselI(x, 2))
```

- Deriv() offers many other useful features for R programmers. We will see some of them today.

R introspection tools

R code as data

R code can be presented and manipulated as R data structures.

```
> e = quote(sin(2 * x))
> class(e)
## [1] "call"
```

```
> (eli = as.list(e))
## [[1]]
## sin
##
## [[2]]
## 2 * x
```

```
> lapply(eli, class)
## [[1]]
## [1] "name"
##
## [[2]]
## [1] "call"
```

R code as data

```
> eli[[1]] = as.symbol("cos")
> (e = as.call(eli))
## cos(2 * x)
```

Few of functions for code manipulation:

```
> substitute(sin(2*x), as.environment(list(x=as.symbol("z"))))
## sin(2 * z)
```

```
> substitute(e, as.environment(list(x=as.symbol("z"))))
## e
```

```
> do.call(substitute, list(e, as.environment(list(x=as.symbol("z
## cos(2 * z)
```

R code as data

- we can explore an R function with `args()` and `body()`;
- create functions “on the fly” with `as.function()`
- and to know what each parameter became in a real call

```
> args("*")
## function (e1, e2)
## NULL
```

```
> body(y)
## sin(2 * x)
```

```
> as.function(alist(a = , b = 2, a+b))
## function (a, b = 2)
## a + b
## <environment: 0x2b303a8>
```

```
> as.list(match.call(args("*"), quote(2*x)))
## [[1]]
## `*`
##
## $e1
## [1] 2
##
## $e2
## x
```

`Deriv` insides

Derivation algorithm for last leafs in AST

Let `e` an R language element in the last leaf of Abstract Syntax Tree (AST) and `x` is symbol by which we want to differentiate.

```
if (e == x) 1 else 0
```

Derivative rules

Next, we'll need a table of derivative rules for calls

```
# linear functions, i.e. d(f(x))/dx == f(d(arg)/dx)
dlin=c("+", "-", "c", "t", "sum", "cbind", "rbind", "list")

# rule table
# arithmetics
drule[["*"]] <- alist(e1=e2, e2=e1)
drule[["^"]] <- alist(e1=e2*e1^(e2-1), e2=e1^e2*log(e1))
drule[["/"]] <- alist(e1=1/e2, e2=-e1/e2^2)
# log, exp, sqrt
drule[["sqrt"]] <- alist(x=0.5/sqrt(x))
drule[["log"]] <- alist(x=1/(x*log(base)), base=-log(x, base)/(base*log
...
drule[["dnorm"]] <- alist(x=-(x-mean)/sd^2*if (log) 1 else dnorm(x, mea
    mean=(x-mean)/sd^2*if (log) 1 else dnorm(x, mean, sd),
    sd=(((x - mean)/sd)^2 - 1)/sd * if (log) 1 else dnorm(x, mean, sd),
    log=NULL)
...
```

Differentiate an expression

If a call is in `drule` table, then it will be replaced by a sum of all partial derivatives read from `drule`. Each partial derivative is multiplied by a derivative of corresponding argument by `x`.

```
> drule[["sin"]]
## $x
## cos(x)
```

```
> drule[["*"]]
## $e1
## e2
##
## $e2
## e1
```

Expression	Derivative
`sin(2*x)`	`cos(2*x)*d(2*x)/dx`
`2*x`	`x*d(2)/dx + 2*d(x)/dx`

Examples

Calculate a Hessian for least squares functional

$$L = \Sigma_i (a * x_i - b - y_i)^2$$

$$\min_{a,b} L$$

```
> L=function(a,b, x=NULL,y=NULL) sum((a*x+b-y)**2)
> gradL=Deriv(L, c("a", "b"))
> hessL=Deriv(gradL, c("a", "b"), combine="cbind")
> set.seed(7); a=1; b=2; x=1:10; y=a*x+b+rnorm(x, 0, 0.1)
> (sol=solve(hessL(0, 0, x, y), -gradL(0, 0, x, y)))
##          a          b
##  1.3766662 -0.6126639
```

Oh-oh!

What's wrong with our Hessian?

```
> hessL
## function (a, b, x = NULL, y = NULL)
## {
##     .e2 <- sum(2 * x)
##     cbind(a = c(a = sum(2 * x^2), b = .e2), b = c(a = .e2, b = 2))
## }
## <environment: 0x2b303a8>
```

```
> hessL(0, 0, x, y)
##     a   b
## a 770 110
## b 110   2
```

The problem is in $\partial^2 L/\partial b^2$ term which is set to 2 instead of $2n$.

Why we've got a wrong term?

Because of implicite R rule of recycling scalar arguments to fit needed vector length. We have to do it explicitly!

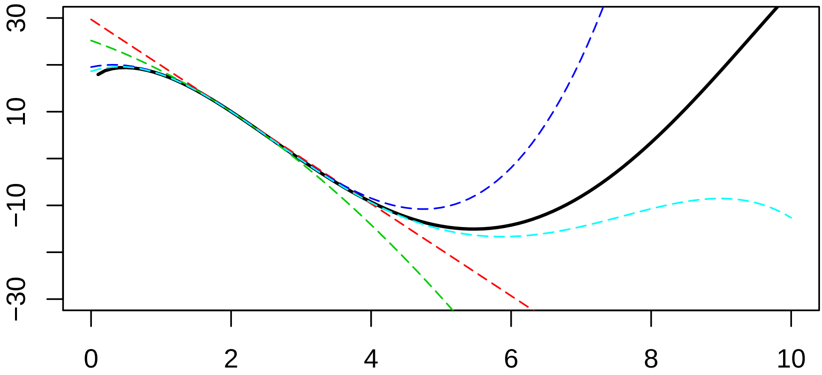
Re-calculate the Hessian (the good one)

```
> L=function(a,b, x=NULL,y=NULL) sum((a*x+rep.int(b, length(x))-y)**2)
> gradL=Deriv(L, c("a", "b"))
> hessL=Deriv(gradL, c("a", "b"), combine="cbind")
> set.seed(7); a=1; b=2; n=10; x=1:10; y=a*x+b+rnorm(x, 0, 0.1)
> (sol=solve(hessL(0, 0, x, y), -gradL(0, 0, x, y)))
##        a        b
## 1.009068 1.960524
```

```
> hessL
## function (a, b, x = NULL, y = NULL)
## {
##     .e2 <- rep.int(1, length(x))
##     .e4 <- sum(2 * (x * .e2))
##     cbind(a = c(a = sum(2 * x^2), b = .e4), b = c(a = .e4, b = sum(2 *
##         .e2^2)))
## }
## <environment: 0x2b303a8>
```

Taylor series for user defined functions [1]

```
> TS=function(f, x, x0, k) {
+   vd=unlist(Deriv(f, "x", nderiv=0:k)(x0))
+   return(sapply(x, function(xi) sum(vd*(xi-x0)**(0:k)/factorial(0:k))))
+ }
> f = function(x) log(x) - x^4/24 + x^3 - 6*x^2 + 3*x + 20
> x=seq(0, 15, length.out = 101); x0=2
> curve(f(x), xlim=c(0,10), ylim=c(-30, 30), xlab="", ylab="", lwd=2)
> tmp=lapply(1:4, function(nd) curve(TS(f,x,x0,nd), xlim=c(0,10), add=T, col=nd+1, lty=5))
```



[1] Credits: Bertrand Koebel, Louis Pasteur University Strasbourg

User defined derivative rules

Often classical scalar derivative applies to vector arguments, term by term. But not always:

```
> le=function(x) log(sum(exp(x)))
> Deriv(le)
## function (x)
## 1
## <environment: 0x2b303a8>
```

```
> # Oops...
> # Let repair it by a special rule for `le()` (thx @chuanwen)
> le2=function(x) {xm=max(x); log(sum(exp(x-xm)))+xm}
> dle2=function(x) {e_x=exp(x-max(x)); e_x/sum(e_x)}
> drule[["le2"]]=alist(x=dle2(x))
> le(710:711)
## [1] Inf
```

```
> le2(710:711)
## [1] 711.3133
```

```
> Deriv(le2)(710:711)
## [1] 0.2689414 0.7310586
```

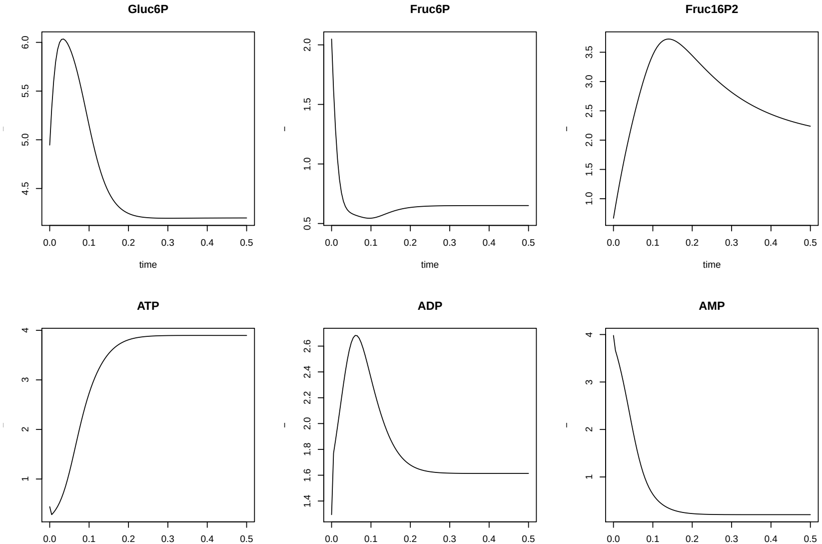
Closer to real world example

```
> # Upper glycolysis model taken from "CSB Lecture flux balance analysis"
> #  which cites E. Klipp, Systems Biology in Practice, 2005
> upglyc=function(t, co, p) {
+   # provide derivative vector for ODE solving
+   # flux as function of conc and kin. parameters
+   nu1=p["Vmax1"]*co["ATP"]*p["Glucose"]/(1+co["ATP"]/p["Katp1"]+p["Glucose"]/
+   nu2=p["k2"]*co["ATP"]*co["Gluc6P"]
+   nu3=((p["Vfmax3"]/p["Kgluc6p3"])*co["Gluc6P"]-(p["Vrmax3"]/p["Kfruc6p3"])*c
+     (1+co["Gluc6P"]/p["Kgluc6p3"]+co["Fruc6P"]/p["Kfruc6p3"])
+   nu4=p["Vmax4"]*co["Fruc6P"]**2/(p["Kfruc6p4"]*(1+p["ka"]*(co["ATP"]/co["AMP
+   nu5=p["k5"]*co["Fruc16P2"]
+   nu6=p["k6"]*co["ADP"]
+   nu7=p["k7"]*co["ATP"]
+   nu8=p["k8f"]*co["ATP"]*co["AMP"]-p["k8r"]*co["ADP"]**2
+   # first derivatives in time of concentration vector co (named)
+   Gluc6P=nu1-nu2-nu3
+   Fruc6P=nu3-nu4
+   Fruc16P2=nu4-nu5
+   ATP=-nu1-nu2-nu4+nu6-nu7-nu8
+   ADP=nu1+nu2+nu4-nu6+nu7+2*nu8
+   AMP=-nu8
+   list(c(Gluc6P, Fruc6P, Fruc16P2, ATP, ADP, AMP))
+ }
```

Parameter sensitivity analysis

Function `upglyc()` is written to be usable with `DeSolve` package.

```
> source("upper_glyco_model.inc.R")
```



Parameter sensitivity analysis

```
> Deriv(upglyc, x=c(p="Katp1"))
## function (t, co, p)
## {
##     .e6 <- (1/p["Katp1"]^2 + p["Glucose"] * p["Kglucose1"]/(p["Katp1
##         p["Kglucose1"])^2) * co["ATP"]^2 * p["Glucose"] * p["Vmax1"]
##         p["Glucose"]/p["Kglucose1"]) * co["ATP"]/p["Katp1"] +
##         1 + p["Glucose"]/p["Kglucose1"])^2
##     list(c(.e6, 0, 0, -.e6, .e6, 0))
## }
## <environment: 0x2b303a8>
```

Features remained uncovered

Some useful features remained out of scope of this presentation:

- simplifications: they make the code to look "presentable" but do have corner cases;
- caching : it makes the code to be efficient (no re-calculated expression), can be disabled;

Conclusions

Conclusions

- `Deriv` can differentiate not only standalone R expressions or formula but a real mathematical code wrapped in a function;
- result is computationally efficient due to symbolic simplification and caching;
- can be helpful in *accurate* parameter sensitivity analysis

Questions? Comments?

Thank you for your attention!