ReverseDiffSource Documentation

Release 0.1

Frédéric Testard

February 10, 2015

Contents

Contents:

CHAPTER 1

Installation

Install the ReverseDiffSource package at the Julia command line by running:

```
Pkg.add("ReverseDiffSource")
```

This only needs to be done once.

`ReverseDiffSource` has currently no dependency with other packages. However there is a graphical representation function `plot()` that produces a text string in the GraphViz syntax. To produce the plot, you will have to install GraphViz with `Pkg.add`.

Now for each Julia session where ReverseDiffSource is needed, load it with the usual:

```
using ReverseDiffSource
```

You are now ready to go !

CHAPTER 2

The main function : `rdiff()`

The differentiation function is called `rdiff()` and is called with the following parameters:

```
rdiff( ex::Expr; outsym::Symbol; order::Int, init... )
```

2.1 Arguments

ex is a Julia Expression containing the code to derive

outsym (keyword arg, default = nothing) is the symbol of the variable within `ex` containing the expression output (the result whose derivatives are needed). This variable must evaluate to a `Real`. If not specified, `outsym` defaults to `nothing` which signals to `rdiff` that the last statement is the result of interest for derivation.

order (keyword arg, default = 1) is an integer indicating the derivation order (1 for 1st order, etc.). Order 0 is allowed and will produce an expression that is a processed version of `ex` with some variables names rewritten and possibly some optimizations.

init (multiple keyword arguments) is one or several symbol / value pairs indicating which variables appearing in `ex` should be used for the derivation of `outsym`. An example value for each variable needs to be specified in order to fully evaluate `ex`. This is necessary for the derivation algorithm because the generated code will depend on the type of each intermediate result.

evalmod (default=Main) module where the expression is meant to be evaluated. External variables and functions should be evaluable in this module.

debug (default=false) if true `rdiff` dumps the graph of the generating expression, instead of the expression.

allorders (default=true) tells rdiff whether to generate the code for all orders up to `order` (true) or only the last order.

2.2 Output

An expression which, when evaluated, will return a tuple containing the expression value and the derivative at first, second , etc.. order.

2.3 Usage

`rdiff` takes an expression consisting of a subset of Julia statements (assigments, getindex, setindex!, for loops, function calls) and transforms it into a new expression whose evaluation will provide the derivatives at all orders between 0 and the order specified (unless `allorders` is false).

The generated expression will attempt to remove all uneeded calculations (e.g. x + 0) and factorize repeated function calls as much as possible.

All the variables appearing in the init argument are considered as the expression's arguments and a derivative is calculated for it (and cross derivatives if order is >= 2). The other variables, if not defined by the expression, are expected to be top level variables in `evalmod`. If they are not defined there an error will be thrown.

For orders >= 2 *only a single variable, of type Real or Vector, is allowed.* For orders 0 and 1 variables can be of type Real, Vector or Matrix and can be in an unlimited number:

```
julia> rdiff( :(x^3) , x=2.)  # first order
:(begin
    (x^3,3 * x^2.0)
    end)

julia> rdiff( :(x^3) , order = 3, x=2.)  # orders up to 3
:(begin
        (x^3,3 * x^2.0,2.0 * (x * 3),6.0)
    end)
```

`rdiff` runs several simplification heuristics on the generated code to remove neutral statements and factorize repeated calculations. For instance calculating the derivatives of `sin(x)` for large orders will reduce to the calculations of `sin(x)` and `cos(x)`:

```
julia> rdiff( :(sin(x)) , order=10, x=2.)  # derivatives up to order 10
:(begin
        _tmp1 = sin(x)
        _tmp2 = cos(x)
        _tmp3 = -_tmp1
        _tmp4 = -_tmp2
        _tmp5 = -_tmp3
        (_tmp1,_tmp2,_tmp3,_tmp4,_tmp5,_tmp2,_tmp3,_tmp4,_tmp5,_tmp2,_tmp3)
    end)
```

The expression produced can readily be turned into a function with the `@eval` macro:

```
julia> res = rdiff( :(sin(x)) , order=10, x=2.)
julia> @eval foo(x) = $res
julia> foo(2.)
(0.9092974268256817,-0.4161468365471424,-0.9092974268256817,0.4161468365471424,0.9092974268256817,-0.
```

When a second derivative expression is needed, only a single derivation variable is allowed. If you are dealing with a function of several (scalar) variables you will have you aggregate them into a vector:

```
julia> ex = :( (1 - x[1])^2 + 100(x[2] - x[1]^2)^2 )  # the rosenbrock function
julia> res = rdiff(ex, x=zeros(2), order=2)
:(begin
    _tmp1 = 1
    _tmp2 = 2
    _tmp3 = 100.0
    _tmp4 = _tmp1 - x[_tmp1]
    _tmp5 = length(x)
    _tmp6 = zeros(size(x))
```

```
    _tmp7 = x[_tmp2] - x[_tmp1] ^ _tmp2
    _tmp8 = zeros((_tmp5,_tmp5))
    _tmp9 = _tmp2 * (_tmp7 * _tmp3)
    _tmp10 = -_tmp9
    _tmp6[_tmp1] = _tmp6[_tmp1] + (_tmp2 * (x[_tmp1] * _tmp10) + -(_tmp2 * _tmp4))
    _tmp6[_tmp2] = _tmp6[_tmp2] + _tmp9
    for _idx1 = _tmp1:_tmp5
        _tmp11 = zeros(size(_tmp6))
        _tmp12 = zeros(size(x))
        _tmp11[_idx1] = _tmp11[_idx1] + 1.0
        _tmp13 = _tmp11[_tmp2]
        _tmp11[_tmp2] = 0.0
        _tmp11[_tmp2] = _tmp11[_tmp2] + _tmp13
        _tmp14 = _tmp2 * _tmp11[_tmp1]
        _tmp15 = _tmp3 * (_tmp2 * (_tmp13 + -(x[_tmp1] * _tmp14)))
        _tmp12[_tmp1] = _tmp12[_tmp1] + ((_tmp10 * _tmp14 + _tmp2 * (x[_tmp1] * -_tmp15)) + -(_tmp2 *
        _tmp12[_tmp2] = _tmp12[_tmp2] + _tmp15
        _tmp8[(_idx1 - 1) * _tmp5 + 1:_idx1 * _tmp5] = _tmp12
    end
    (_tmp4 ^ _tmp2 + 100 * _tmp7 ^ _tmp2,_tmp6,_tmp8)
    end)
julia> @eval foo(x) = $res
julia> foo([0.5, 2.])
    (306.5,[-351.0,350.0],
    2x2 Array{Float64,2}:
     -498.0  -200.0
     -200.0   200.0)
```

`foo(x)` returns a tuple containing respectively the value of the expression at `x`, the gradient (a 2-vector) and the hessian (a 2x2 matrix)

2.4 Limitations

- The canonical implementation of `for` loops derivation in reverse accumulation requires the caching of the complete state of each iteration which makes the generated code complex and memory intensive. The current algorithm uses a simpler approach that limits the kind of loops that can be correctly derived : in short, loops should not have any kind of recursivity in them (the calculations of each iteration should not depend on the calculations of previous iterations):

```
# will work
for i in 1:n
    a = f(x[i])
    b = a + g(y[i])
    c[i] = b
end

# will (probably) not work
for i in 1:n
    c[i] = f( c[i-1] )
end
```

However simple accumulations are an instance of recursive calculations that will work:

```
# will work
for i in 1:n
    a += b[i]    # new a value depends on previous a
```

```
end
```

- `for` loops are limited to a single index. If you have a `for i,j in 1:10, 1:10` in your expression you will have to translate it to nested loops as a workaround
- All variables should be type-stable (not change from a scalar to a vector for example).
- Only a limited set of Julia semantics are supported at this stage. Some frequently used statements such as comprehensions, `if else`, `while` loops cannot be used in the expression.
- Mutating functions cannot be used (with the exception of `setindex!` and `setfield!`).

CHAPTER 3

Calling `rdiff()` with a function

Warning: Currently `rdiff()` will not work with `for` loops in the function definition. This is caused by `uncompressed_ast` call not giving access to the original function definition but to an interpreted version that is more challenging to parse.

Calling syntax:

```
rdiff( func::Function, init::Tuple; order::Int)
```

3.1 Arguments

func is a Julia generic function.

init is a tuple containing initial values for each parameter of `func`.

order (keyword arg, default = 1) is an integer indicating the derivation order (1 for 1st order, etc.). Order 0 is allowed and will produce a function that is a processed version of `ex` with some variables names rewritten and possibly some optimizations.

evalmod (default=Main) module where the expression is meant to be evaluated. External variables and functions should be evaluable in this module.

debug (default=false) if true `rdiff` dumps the graph of the generating expression, instead of the expression.

allorders (default=true) tells rdiff whether to generate the code for all orders up to `order` (true) or only the last order.

3.2 Output

A function, evaluated in the same module that `func` is from and returning a tuple containing the expression value and the derivative at first, second , etc.. order.

3.3 Usage

`rdiff` takes a function defined with the same subset of Julia statements (assigments, getindex, setindex!, for loops, function calls) as the Expression variant of `rdiff()` and transforms it into another function whose call will return the derivatives at all orders between 0 and the order specified:

```
julia> rosenbrock(x) = (1 - x[1])^2 + 100(x[2] - x[1]^2)^2   # function to be derived
julia> rosen2 = rdiff(rosenbrock, (ones(2),), order=2)       # orders up to 2
        (anonymous function)
julia> rosen2([1,2])
        (100,[-400.0,200.0],
        2x2 Array{Float64,2}:
          402.0  -400.0
         -400.0   200.0)
```

The generated function will attempt to remove all uneeded calculations (e.g. x + 0) and factorize repeated function calls as much as possible.

All the variables appearing in the init argument are considered as the expression's arguments and a derivative is calculated for it (and cross derivatives if order is >= 2).

For orders >= 2 *only a single variable, of type Real or Vector, is allowed*. For orders 0 and 1 variables can be of type Real, Vector or Matrix and can be in an unlimited number. If you are dealing with a function of several (scalar) variables you will have you aggregate them into a vector (as in the example above).

3.4 Limitations

- No `for` loops allowed for this `rdiff` version.
- All variables should be type-stable (not change from a scalar to a vector for example).
- Only a limited set of Julia semantics are supported at this stage. Some frequently used statements such as comprehensions, `if else`, `while` loops cannot be used in the expression.
- Mutating functions cannot be used (with the exception of `setindex!` and `setfield!`).

CHAPTER 4

Defining new functions : `@deriv_rule()`

ReverseDiffSource comes with the derivations instructions for a limited set of functions such as `*`, `+`, `/`, `transpose`, `exp`, `log`, ... You can 'teach' the package derivation methods for new functions with the macro call `@deriv_rule`:

```
@deriv_rule ex::Expr var::Symbol rule::Expr
```

4.1 Arguments

ex is the function signature, with each argument specified

var is the symbol of the argument you derive for.

rule is an expression to calculate the value to be added to the derivative accumulator for variable `var`.

4.2 Usage

`rule` should contain an expression that can be parsed by ReverseDiffSource (syntax limitations mentionned in previous chapter apply here). All symbols in it should either be one of the arguments in the function signature or the special symbol `ds` that represents the derivative accumulator of the function.

Julia's multiple dispatch rules apply to the definition : you can define different rules for a given function depending on the type of its arguments:

```
@deriv_rule *(x::Real         , y::Real )            y     x * ds
@deriv_rule *(x::Real         , y::AbstractArray)    y     x .* ds
@deriv_rule *(x::AbstractArray, y::Real )            y     sum(x .* ds)
@deriv_rule *(x::AbstractArray, y::AbstractArray)    y     x' * ds
```

4.3 Example

Suppose you defined a function `foo(x)`:

```
foo(x) = log(1+sin(x))
```

This function is in turn used in the expression you want to derive:

```
ex = :( 2 ^ foo(x) )
```

Define the derivation of `foo` by its argument:

```
@deriv_rule foo(x)   x   cos(x) / ( 1 + sin(x)) * ds
```

You can now derive `ex`:

```
julia> rdiff( :( 2 ^ foo(x) ) , x=1)
        :(begin
                _tmp1 = 2^foo(x)
                (_tmp1,((cos(x) / (1.0 + sin(x))) * (0.6931471805599453_tmp1),))
            end)
```

CHAPTER 5

Working with composite types

When encountering a composite type, ReverseDiffSource builds a `Vector{Any}` to hold its derivative accumulator. Its structure is derived from the fields of the composite type: Float for a Real number, an array of Floats for Arrays, or another `Vector{Any}` if the field is a type. No special declaration has to be made beforehand to ReverseDiffSource.

However you do need to declare how each function using the composite type changes its derivative accumulator.

Suppose you have type `Bar` defined as:

```
type Bar
    x
    y
end
```

And an associated function `norm(z::Bar)`:

```
norm(z::Bar) = z.x*z.x + z.y*z.y
```

And finally an expression to derive making use of `Bar` and `norm()`:

```
ex = :( z = Bar(2^a, sin(a)) ; norm(z) )
```

You need to declare how both the constructor `Bar` and the function `norm` behave regarding the derivative accumulator (which will be a 2 element vector of type `Any` for the two fields `x` and‘‘y‘‘):

```
@deriv_rule  Bar(x,y)      x  ds[1]   # Derivative accumulator of x is increased by ds[1]
@deriv_rule  Bar(x,y)      y  ds[2]   # Derivative accumulator of y is increased by ds[2]

@deriv_rule  norm(z::Bar)  z  Any[ 2*z.x*ds , 2*z.y*ds ]  # Note : produces a 2-vector since z is a B
```

We are now ready to derive:

```
julia> res = rdiff(ex, a=0.)
julia> @eval df(a) = $res

julia> df(1)
        (4.708073418273571,6.454474871305244)
```

CHAPTER 6

ReverseDiffSource internals

All the core of the functions in the package (differentiation, removal of neutral statements, factorization of identical calls) rely on 2 structures:

1. **The ExNode composite type that represents either:**
    - a single operation (a function call)
    - an external reference (a variable that can be a parameter for derivation or a reference to a variable outside the scope of the expression)
    - a constant

   ExNodes have parents which are typically the arguments of the function. Collectively they make a DAG but with several a

    - the order of arguments (parent nodes) is significant (`a ^ b` is not the same as `b ^ a`)
    - there needs to be additionnal ordering information as statements not related sometimes need to execute in a specific order, this information is in the `precedence` field.
2. **The ExGraph composite type that stores**
    - ExNodes in a vector (in the order of execution),
    - information on how to map ExNodes to variable names (used and set),
    - and optionnaly information on how to map nodes to 'outer' nodes. This last mapping is necessary when the ExGraph is embedded in another parent graph (for example the inner scope of `for` loops is represented as a subgraph).

6.1 Showing the code graph

Starting from an expression, it is possible to have a dump of the nodes composing its equivalent graph with the (unexported) `tograph()` call :

ex = quote a = 1 + x 2 * exp(-a)

end

g = ReverseDiffSource.tograph(ex)

node | symbol | ext ? | type | parents | precedence | main | value | —- | ———- | —- | ———- | ——- | ———- | —- | ———- | 1 | | | [constant] | | | 1 | (Float64) NaN | 2 | x >> | | [external] | | | :x | (Float64) NaN | 3 | a << | | [call] | 11, 1, 2 | | :call | (Float64) NaN | 4 | | | [constant] | | | 2 | (Float64) NaN | 5 | | | [call] | 8, 3 | | :call | (Float64) NaN | 6 | | | [call] | 10, 5 | | :call | (Float64) NaN | 7 | nothing << | | [call] | 9, 4, 6 | |

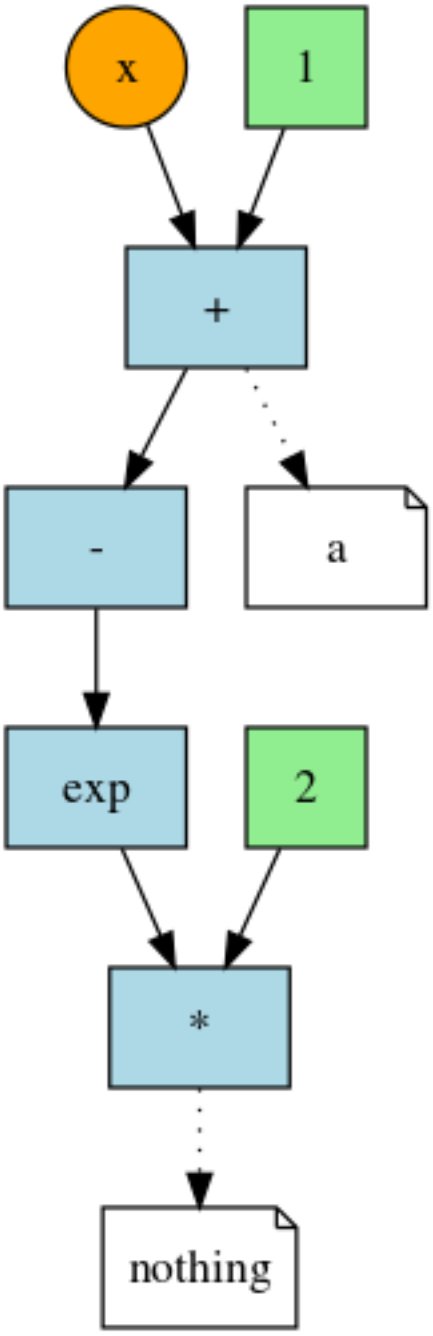
:call | (Float64) NaN | 8 | | | [constant] | | | - | (Float64) NaN | 9 | | | [constant] | | | * | (Float64) NaN | 10 | | | [constant] | | | exp | (Float64) NaN | 11 | | | [constant] | | | + | (Float64) NaN |

Additionnaly, the `plot()` function (also unexported) will generate a GrapViz compatible graph description :

using GraphViz

Graph(ReverseDiffSource.plot(g))

Should produce :



CHAPTER 7

Indices and tables

- *genindex*
- *modindex*
- *search*