ragni-cas - A Pure *Ruby* Automatic Differentiation Library for Fast Prototyping of Interfaces

Matteo Ragni[a]

[a] *Department of Industrial Engineering, University of Trento, 9, Sommarive, Povo di Trento, Italy*

Abstract

Ca. 100 words

Keywords: CAS, code-generation, Ruby

1. Motivation and significance

Ruby is a purely object-oriented scripting language designed in the mid-1990s by Yukihiro Matsumoto (also known as *Matz*). It is internationally standardized since 2012 as ISO/IEC 30170.

With the advent of the *Internet of Things*, a written from scratch version of the *Ruby* interpreter called *mRuby* (*eMbedded Ruby*) has been published on *GitHub* by Matsumoto in 2014. The new interpreter is a lightweight implementation aimed at both low power devices and personal computer that complies with the standard. *mRuby* has a completely new API, and it is designed to be embedded in a complex project as a front-end interface — e.g. a numerical optimization suite may use *mRuby* to get problem input definitions.

The *Ruby* code-base exposes a a large set of utilities in core and standard library, that can be furthermore expanded through libraries, also known as *gems*. Even the high number of gems deployed and available, there is no

Email address: `matteo.ragni@unitn.it` (Matteo Ragni)

library that implements a **automatic symbolic differentiation** (ASD) engine that handles some basic computer algebra routines, that is also cross compatible with all the different *Ruby* interpreters flavours.

Ruby has matured its fame as a web oriented language with *Rails*, and can efficiently generate code in other languages. An AD-capable gem is the foundamental step to rapidly develop a specific code generator for well known software — e.g. IPOPT.

The library described in this work, is a gem implemented in pure *Ruby* code — compatible with all standardized interpreters — that is able to perform symbolic ASD and some computer algebra operations. The library aims at:

- be an instrument for rapid development of prototype interface for numerical algorithms and exporting code generated in different target languages;
- generate rapidly descriptions of mathematical models, with *easy to implement* workaround for numerical issues, changing on request how the code is exported, and how expressions are formulated in the target language;
- create a complete open-source CAS system for the standard *Ruby* language, as a long-term ambitious impact.

This is not the first gem that tries to implement a CAS. The available computer algebra library for *Ruby* are:

Rucas*, *Symbolic gems at early stage and with discontinued developing status; they implement basic simplification routines. There is no AD method, but it is one of the milestones. The development for both is currently discontinued.

Symengine is a wrapper for the C++ library *symengine*. The back-end library is very complete, but it is compatible only with the mainstream *Ruby* interpreter. At the moment, the *SciRuby* project reports the gem as broken, and removed it from its codebase. From a direct test, when performing AD of a function, the engine returns always `nil`.

2. Software description

2.1. Software Architecture

ragni-cas is an object oriented ASD gem that supports some computer algebra routines such as *simplifications* and *substitutions*. When gem is required, it automatically overloads methods of the `Fixnum` and `Float` classes, to make them compatible with the foundamental symbolic class.

Each symbolic expression (or operation) is the instance of an object, that inherits from a common virtual ancestor: `CAS::Op`. An operation encapsulates sub-operations recursively, building a linked graph, that is the mathematical equivalent of function composition:

$$(f \circ g) \tag{1}$$

When a new operation is created, it is appended to the graph. The number of branches are determined by the parent container class of the current symbolic function. There are three possible containers. Single argument operations — e.g. $\sin(\cdot)$ — have as closest parent the `CAS::Op` class, that links to one sub-graph. Expressions with two arguments — e.g. difference or exponential function — inherit from `CAS::BinaryOp`, that links to two subgraphs. Operations with arbitrary number of arguments — e.g. sum and product — have as parent the `CAS::NaryOp`[1], that links to an arbitrary

[1] Please note that this container is still at experimental stage

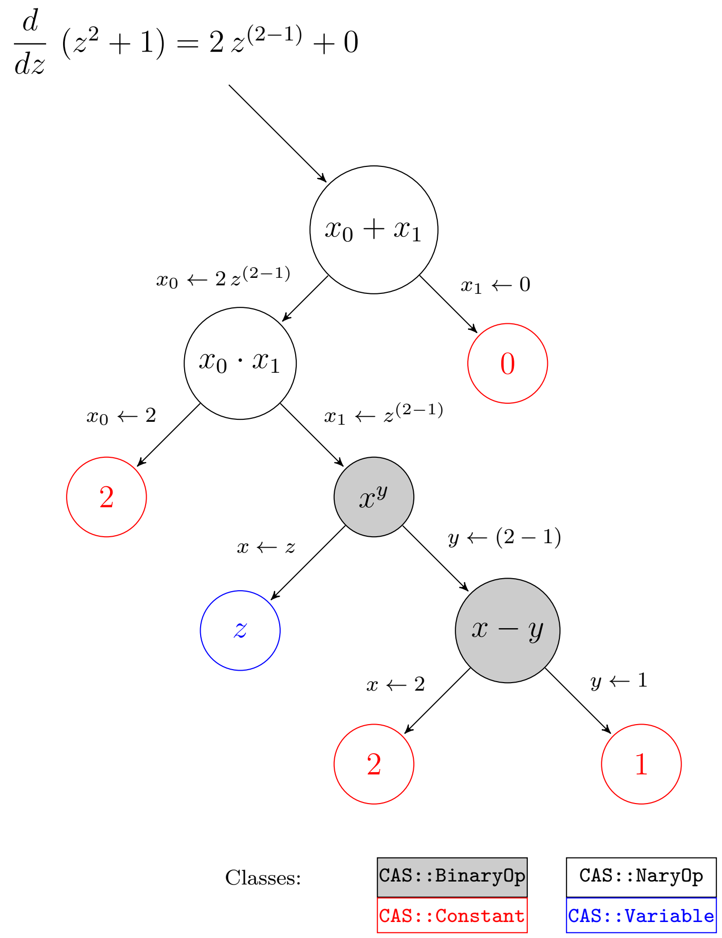

Figure 1: Example graph from the first function reported in listing 1

number of subgraph. Figure 2.1 contains an example of graph. The different kind of containers allows to introduce some properties like *associativity* and *commutativity*. Each container exposes the subgraphs as instance properties. Containers structure is shown in Figure 2.1.

Terminal leafes of the graph are the classes `CAS::Constant`, `CAS::Variable` and `CAS::Function`. The first is models a simple numerical value, while the second represents an independent variable, that can be used to perform derivatives and evaluations, and the latter is a prototype of an implicit function. As for now, those leafes exemplify only real scalar expressions, with plans to define also the complex, vectorial and matricial extensions in the next major release.

Automatic differentiation (`CAS::Op#diff`) crosses the graph until it reaches the ending node. The terminal node is the starting point for derivatives accumulation, the mathematical equivalent of the chain rule:

$$(f \circ g)' = (f' \circ g)\ g' \tag{2}$$

The recursiveness is used also for simplifications (`CAS::Op#simplify`), substitutions (`CAS::Op#subs`) and evaluations (`CAS::Op#call`).

2.2. Software Functionalities

2.2.1. Basic Functionalities

ASD can be performed with respect to an independent variable (`CAS::Variable`) through forward accumulation, even for implicit functions. The differentiation is done by a method of the `CAS::Op`, having a `CAS::Variable` as argument:

Listing 1: Differentiation example

```
x = CAS.vars 'x'           # creates a variable
f = x ** 2 + 1             # define a symbolic expression
f.diff(x)                  # derivative w.r.t. x
# => 2 * x ^ (2 — 1) + 0
g = CAS.declare :g, f      # creates implicit expression
g.diff(x)                  # derivative w.r.t. x
# => (x ^ (2 — 1) * 2) * Dg[0](x ^ 2)
```

Simplifications are not executed automatically, after differentiations. Each node of the graph knowns rules for simplify itself, and rules are called recursively inside the graph, exactly like ASD. Simplifications that require an *heuristic expansion* of the subgraph — i.e. some trigonometric identities — are not defined for now, but they can be easily achieved through **substitutions**:

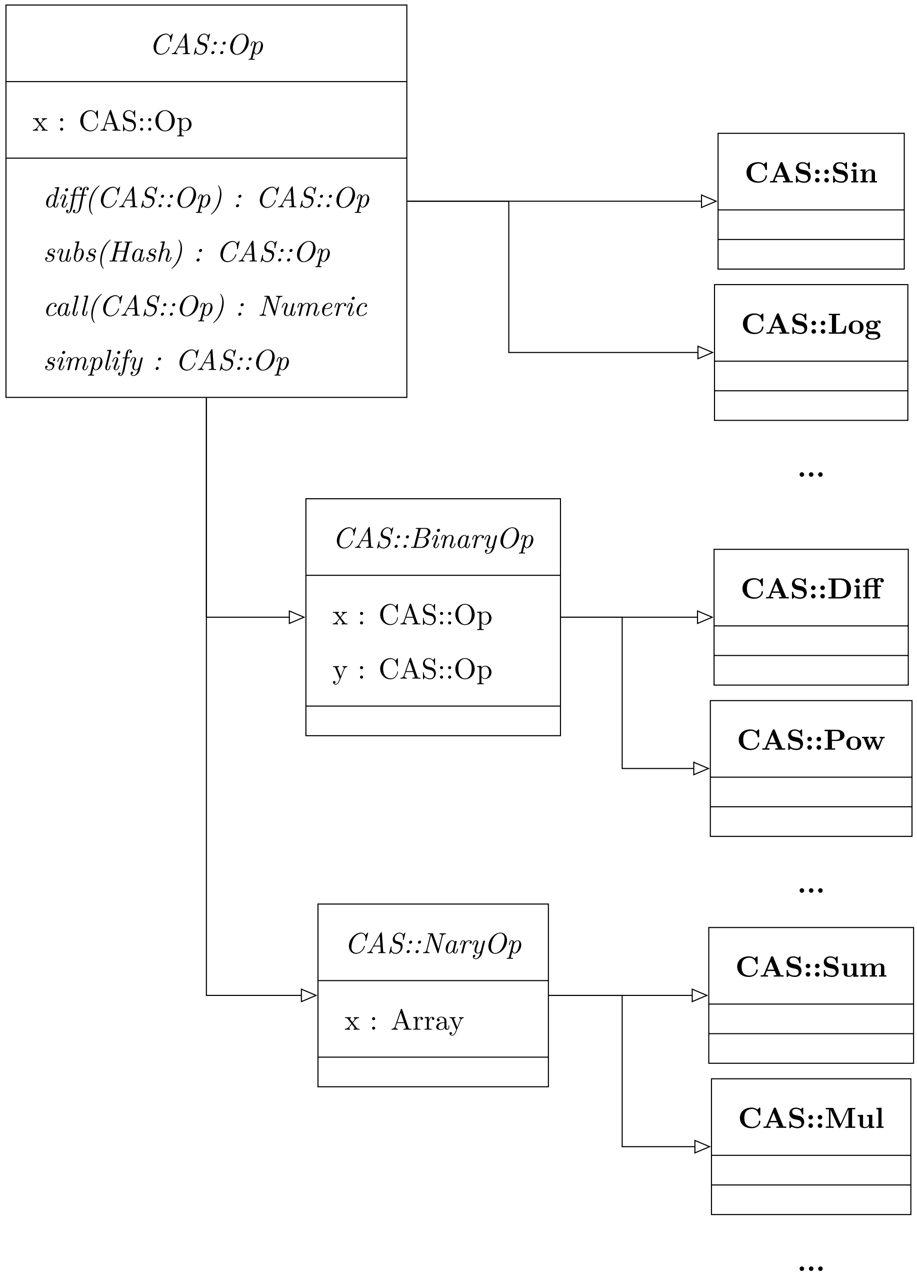

Figure 2: Simplified version of classes interface and inheritance

Listing 2: Simplification example

```
x, y = CAS::vars 'x', 'y'        # creates two variables
f = CAS.log( CAS.sin( y ) )      # symbolic expression
```

```
f.subs  y: CAS.asin(CAS.exp(x))  # perform substitution
f.simplify                       # simplify expression
# => x
```

The graph can be numerically **evaluated** when independent variables values are provided in a feed dictionary. The graph is recursively reduced to a single numeric value:

Listing 3: Graph evaluation example

```
x = CAS.vars 'x'          # creates a variable
f = x ** 2 + 1             # define a symbolic expression
f.call x => 2              # evaluate for x = 2
# => 5
```

Symbolic functions can be used to create comparative expressions — e.g. $f(\cdot) \geq g(\cdot)$ — or piecewise functions — e.g. $\max(f(\cdot), g(\cdot))$:

Listing 4: Expressions and Piecewise functions

```
x, y = CAS.vars 'x', 'y'
f = CAS.declare :f, x
g = CAS.declare :g, x, y
f.greater_equal g
# => (f(x) >= g(x, y))
CAS::max f, g
# => ((f(x) >= g(x, y)) ? f(x) : g(x, y))
```

Comparative expression are stored in a special container classes, modeled by the ancestor `CAS::Condition`.

2.2.2. Metaprogramming and Code-Generation

The library is developed explicitly for **generation of code** for a target language, and **metaprogramming**. Expressions, once manipulated, can be exported as plain source code or used as a prototype for a callable *closure* (`Proc` object):

Listing 5: Graph evaluation example

```
x = CAS::vars 'x'             # creates a variable
```

```
f = CAS::log(CAS::sin(x))     # define a symbolic function

proc = f.as_proc              # exports callable lambda
proc.call 'x' => Math::PI/2
# => 0.0
```

Composing a closure of a graph is like making its snapshot, thus any further manipulation to the expression do not update the callable object. This drawback is balanced by the faster execution time of a `Proc`: when a graph needs only to be evaluated in a iterative algorithm, and not to be manipulated, transforming it in a *closure* reduces the execution time per iteration.

Code generation should be flexible enough to export a graph in a user's target language. Generation methods for common languages are included in specific plugins. Users can furthemore expand exporting capabilites by writing specific exportation rules, overriding method for existing plugin, or desining their own exporter:

Listing 6: Example of Ruby exportation plugin

```
# Definition
module CAS
  {
    # . . .
    CAS::Variable => Proc.new { "#{name}" }
    CAS::Sin      => Proc.new { "Math.sin(#{x.to_ruby})" },
    # . . .
  }.each do |cls, prc|
    cls.send(:define_method, :to_ruby, &prc)
  end
end

# Usage
x = CAS.vars 'x'
(CAS.sin(x)).to_ruby
# => Math.sin(x)
```

Included plugins may implement some advanced features such as code optimization and generation of libraries: this is an example with the *C* plugin:

Listing 7: Calling optimized-C exporter for library generation

```
require 'ragni-cas/c-opt'

x, y = CAS.vars :x, :y
g = CAS.declare :g, x

g.cname = 'g_impl'
f = x ** y + g * CAS.log(CAS.sin(x ** y))

CLib.create "example" do
  include_local "g_impl"
  implements_as "f_impl", f
  implements_as "my_pow", (x ** y)
end
```

library created contains the following source (header is omitted for brevity):

Listing 8: Calling optimized-C exporter

```
[[[[[ TODO Must be written again ]]]]]
[[[[[ ADD header                 ]]]]]
// Source file for library: example.c

#include "example.h"

double func(double x, double y) {
  double __t_0 = pow(x, y);
  double __t_1 = sin(__t_0);
  double __t_2 = log(__t_1);
  double __t_3 = (__t_0 + __t_2);

  return __t_3;
}

// end of example.c
```

3. Illustrative Examples

As example, an implementation of a algorithm that extimates the *order of convergence* for trapezoidal integration scheme will be provided, using the automatic differentiation as interface.

Given a function $f(x)$, the trapezoidal rule for primitive estimation in the interval $[a,\, b]$ is:

$$I_n(a,b) = \frac{b-a}{n}\left(\frac{f(a)+f(b)}{2} + \sum_{k=1}^{n-1} f\left(a + k\frac{b-a}{n}\right)\right) \tag{3}$$

where n mediates the step size of the integration. The error of the approximation is, when the exact primitive $F(x)$ is known:

$$E[n] = F(b) - F(a) - I_n(a,b) \tag{4}$$

This error shows a direct relation:

$$E[n] \propto C\, n^{-p} \tag{5}$$

where p is the convergence order. Using a different value for n, for example $2\,n$:

$$\frac{E[n]}{E[2\,n]} \approx 2^p \quad \rightarrow \quad p \approx log_2\left(\frac{E[n]}{E[2\,n]}\right) \tag{6}$$

Following listings contain the implementation of the described procedure using the described gem and the well known *Python* library *sympy*.

Listing 9: Ruby version

```ruby
require 'ragni-cas'


def integrate(f, a, b, n)
  h = (b - a) / n

  func = f.as_proc

  sum = ((func.call 'x' => a) +
        (func.call 'x' => b)) / 2.0

  for i in (1...n)
    sum += (func.call 'x' => (a + i*h))
  end
  return sum * h
end

def order(f, a, b, n)
  x = CAS.vars 'x'

  f_ab = (f.call x => b) -
         (f.call x => a)
  df   = f.diff(x).simplify
  f_1n = integrate(df, a, b, n)
  f_2n = integrate(df, a, b, 2 * n)

  return Math.log(
    (f_ab - f_1n) /
    (f_ab - f_2n),
  2)
end

x = CAS.vars 'x'
f = CAS.arctan x

puts(order f, -1.0, 1.0, 100)
# => 1.9999999974244451
```

Listing 10: Python version

```python
import sympy
import math

def integrate(f, a, b, n):
    h = (b - a)/n
    x = sympy.symbols('x')
    func = sympy.lambdify((x), f)

    sums = (func(a) +
            func(b)) / 2.0

    for i in range(1, n):
        sums += func(a + i*h)

    return sums * h


def order(f, a, b, n):
    x = sympy.symbols('x')

    f_ab = sympy.Subs(f, (x), (b)).n()-\
           sympy.Subs(f, (x), (a)).n()
    df   = f.diff(x)
    f_1n = integrate(df, a, b, n)
    f_2n = integrate(df, a, b, 2 * n)

    return math.log(
      (f_ab - f_1n) /
      (f_ab - f_2n),
    2)


x = sympy.symbols('x')
f = sympy.atan(x)

print(order(f, -1.0, 1.0, 100))
# => 1.9999999974244451
```

4. Impact

5. Conclusions

Acknowledgements

This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

Current code version

Nr.	Code metadata description	Please fill in this column
C1	Current code version	0.0.0
C2	Permanent link to code/repository used for this code version	github.com/MatteoRagni/cas-rb & rubygems.org/gems/ragni-cas
C3	Legal Code License	MIT
C4	Code versioning system used	*git* (GitHub)
C5	Software code languages, tools, and services used	*Ruby*
C6	Compilation requirements, operating environments	*Ruby* $\geq 2.x$, *pry* for testing console (optional)
C7	If available Link to developer documentation/manual	rubydoc.info/gems/ragni-cas
C8	Support email for questions	info@ragni.me

Table 1: Code metadata (mandatory)