Abstract

There are Computer Algebra System (CAS) systems on the market with complete solutions for manipulation of analytical models. But exporting a model that implements specific algorithms on specific platforms, for target languages or for particular numerical library, is often a rigid procedure that requires manual post-processing. This work presents a *Ruby* library that exposes core CAS capabilities, i.e. simplification, substitution, evaluation, etc. The library aims at programmers that need to rapidly prototype and generate numerical code for different target languages, while keeping separated mathematical expression from the code generation rules, where best practices for numerical conditioning are implemented. The library is written in pure *Ruby* language and is compatible with most *Ruby* interpreters.

1 Motivation and significance

Ruby [7] is a purely object-oriented scripting language designed in the mid-1990s by Yukihiro Matsumoto, internationally standardized since 2012 as ISO/IEC 30170.

With the advent of the *Internet of Things*, a compact version of the *Ruby* interpreter called *mRuby* (*eMbedded Ruby*) [12] was published on *GitHub* by Matsumoto, in 2014. The new interpreter is a lightweight implementation, aimed at both low power devices and personal computers, and complies with the standard [9]. *mRuby* has a completely new API, and it is designed to be embedded in complex projects as a front-end interface—for example, a numerical optimization suite may use *mRuby* for problem definition.

The *Ruby* code-base exposes a large set of utilities in core and standard libraries, that can be furthermore expanded through third party libraries, or *gems*. Among the large number of available gems, *Ruby* still lacks an Automatic and Symbolic Differentiation (ASD) [13] engine that handles basic computer algebra routines, compatible with all different *Ruby* interpreters.

Nowadays *Ruby* is mainly known thanks to the web-oriented *Rails* framework. Its expressiveness and elegance make it interesting for use in the scientific and technical field. An ASD-capable gem would prove a fundamental step in this direction, including the support for flexible code generation for high-level software—for example, IPOPT [17, 16].

Mr.CAS[1] is a gem implemented in pure *Ruby* that supports symbolic differentiation (SD) and fundamentals computer algebra operations [15]. The library aims at supporting programmers in rapid prototyping of numerical algorithms and in code generation, for different target languages. It permits to implement mathematical models with a clean separation between actual mathematical formulations and conditioning rules for numerical instabilities, in order to support generation of code that is more robust with respect to issues that can be introduced by specific applications. As a long-term effort, it will become a complete open-source CAS system for the standard *Ruby* language.

[1]Minimalistic Ruby Computer Algebra System

Other CAS libraries for *Ruby* are available:

***Rucas* [10], *Symbolic* [2]** : milestone gems, yet at an early stage and with discontinued development status. Both offer basic simplification routines, although they lack differentiation.

***Symengine* [5]** : is a wrapper of the *symengine* C++ library. The back-end library is very complete, but it is compatible only with the *vanilla C Ruby* interpreter and has several dependencies. At best of Author knowledge, the gem does not work with *Ruby* 2.x interpreter.

In Section 2, *Mr.CAS* containers and tree structure are explained in detail and applied to basic CAS tasks. In Section 3, examples on how to use the library as code generator or as interface are described. Finally, the reasons behind the implementation and the long term desired impact are depicted in Section 4. All code listings are available at `http://bit.ly/Mr_CAS_examples`.

2 Software description

2.1 Software Architecture

Mr.CAS is an object oriented ASD gem that supports computer algebra routines such as simplifications and substitutions. When gem is required, it overloads methods of `Fixnum` and `Float` classes, making them compatible with fundamental symbolic classes.

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 tree, that is the mathematical equivalent of function composition:

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

When a new operation is created, it is appended to the tree. The number of branches are determined by the parent container class of the current symbolic function. There are three possible containers:

`CAS::Op` single sub-tree operation—e.g. $\sin(\cdot)$.

`CAS::BinaryOp` dual sub-tree operation—e.g. exponent x^y—that inherits from `CAS::Op`.

`CAS::NaryOp` operation with arbitrary number of sub-tree—e.g. sum $x_1 + \cdots + x_N$—that inherits from `CAS::Op`.

Fig. 1 contains a graphical representation of a expression tree. The different kind of containers allows to introduce some properties—i.e. *associativity* and *commutativity* for sums and multiplications [6]. Each container exposes the sub-tree as instance properties. Basic containers interfaces and inheritances are shown in Fig. 2. For a complete overview of all classes and inheritance, please refer to software documentation.

The terminal leaves of the graph are the classes `CAS::Constant`, `CAS::Variable` and `CAS::Function`. The first 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 implicit functions. Those leaves exemplify only real scalar expressions, with definition of complex, vectorial, and matricial extensions as milestones for the next major release.

The symbolic differentiation (`CAS::Op#diff`) explores the graph until it reaches ending nodes. A 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`), evaluations (`CAS::Op#call`) and code generation.

2.2 Software Functionalities

2.2.1 Basic Functionalities

No additional dependencies are required. The gem can be installed through the *rubygems.org* provider[2]. Gem functionalities are required using the Kernel method: `require 'Mr.CAS'`. All methods and classes are encapsulated in the module `CAS`.

Symbolic Differentiation (SD) is performed with respect to independent variables (`CAS::Variable`) through forward accumulation, even for implicit functions. The differentiation is done by the method `CAS::Op#diff`, having a `CAS::Variable` as argument, as shown in Listing 1.

Listing 1: Differentiation example

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

Automatic differentiation (AD) is included as a plugin and exploits the properties of dual numbers to efficiently perform differentiation, see [1] for further details. The AD strategy is useful in case of complex expressions, where explicit derivative's tree may exceed the call stack depth.

Simplifications are not executed automatically, after differentiation. Each node of the tree knows rules for simplify itself, and rules are called recursively, exactly like ASD. Simplifications that require a *heuristic expansion* of the subgraph—i.e. some trigonometric identities—are not defined for now, but can be easily achieved through substitutions, as shown in Listing 2.

[2]`gem install Mr.CAS`

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)) # performs substitution
f.simplify                       # simplifies expression
# => x
```

The tree is numerically evaluated when the independent variables values are provided in a feed dictionary. The graph is reduced recursively to a single numeric value, as shown in Listing 3.

Listing 3: Tree evaluation example

```
x = CAS.vars 'x'   # creates a variable
f = x ** 2 + 1     # defines a symbolic expression
f.call x => 2      # evaluates for x = 2
# => 5.0
```

Symbolic expressions can be used to create comparative expressions, that are stored in special container classes, modeled by the ancestor `CAS::Condition`—for example, $f(\cdot) \geq g(\cdot)$. This allow the definition of piecewise functions, in `CAS::Piecewise`. Internally, $\max(\cdot)$ and $\min(\cdot)$ functions are declared as operations that inherits from `CAS::Piecewise`—for example, $\max(f(\cdot), g(\cdot))$. Usage is shown in Listing 4.

Listing 4: Expressions and Piecewise functions

```
x, y = CAS.vars 'x', 'y'
f = CAS.declare :f, x
g = CAS.declare :g, x, y
h = CAS.declare :h, y

f.greater_equal g
# => (f(x) >= g(x, y))
pw = CAS::Piecewise.new(f,
       CAS::Piecewise.new(g, h, y.equal(0)),
       x.greater(0))
# => ((x > 0) ? f(x) : ((y ≡ 0) ? g(x, y) : h(y)))
CAS::max f, g
# => ((f(x) >= g(x, y)) ? f(x) : g(x, y))
```

2.2.2 Meta-programming and Code-Generation

Mr.CAS is developed explicitly for metaprogramming and code generation. Expressions can be exported as source code or used as prototypes for callable *closures* (the `Proc` object in Listing 5):

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
```

Compiling a closure of a tree is like making its snapshot, thus any further manipulation of the expression does 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*, transforming it in a *closure* reduces the execution time—for example, in a iterative algorithm, where a closure is called at each iteration.

Code generation should be flexible enough to export expression trees in a user's target language. Generation methods for common languages are included in specific *plugins*. Users can furthermore expand exporting capabilities by writing specific exportation rules, overriding method for existing plugin, or designing their own exporter, like the one shown in Listing 6:

Listing 6: Example of Ruby code generation plugin

```
# Rules definition for Fortran Language
module CAS
  {
    # . . .
    CAS::Variable => Proc.new { "#{name}" }
    CAS::Sin      => Proc.new { "sin(#{x.to_fortran})" },
    # . . .
  }.each do |cls, prc|
    cls.send(:define_method, :to_fortran, &prc)
  end
end

# Usage
x    = CAS.vars 'x'
code = (CAS.sin(x)).to_fortran
# => sin(x)
```

3 Illustrative Examples

3.1 Code Generation as C Library

This example shows how a *user of* Mr.CAS can export a mathematical model as a C library. The `c-opt` plugin implements advanced features such as code optimization and generation of libraries.

The library `example` implements the model:

$$f(x, y) = x^y + g(x) \log(\sin(x^y)) \tag{3}$$

where the expression $g(x)$ belongs to a external object, declared as `g_impl`, which interface is described in `g_impl.h` header. What should be noted is the corpus of the exported code: the intermediate operation x^y is evaluated once, even if appears twice in eq. 3. The C function that implements $f(x, y)$ is declared with the token `f_impl`. The exporter uses as default type `double` for variables and function returned values. Library created by `CLib` contains the code shown in Listing 9.

Listing 7: Calling optimized-C exporter for library generation

```
# Model
x, y = CAS.vars :x, :y
g = CAS.declare :g, x

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

# Code Generation
g.c_name = 'g_impl'             # g token

CAS::CLib.create "example" do
  include_local "g_impl"        # g header
  implements_as "f_impl", f     # token for f
end
```

Listing 8: C Header

```
// Header file for library: example.c

#ifndef example_H
#define example_H

// Standard Libraries
#include <math.h>

// Local Libraries
#include "g_impl"

// Definitions

// Functions
double f_impl(double x, double y);

#endif // example_H
```

Listing 9: C Source

```
// Source file for library: example.c

#include "example.h"


double f_impl(double x, double y) {
  double __t_0 = pow(x, y);
  double __t_1 = g_impl(x);
  double __t_2 = sin(__t_0);
  double __t_3 = log(__t_2);
  double __t_4 = (__t_1 + __t_3);
  double __t_5 = (__t_0 + __t_4);

  return __t_5;
}

// end of example.c
```

The function $g(x)$ models the following operation:

$$g(x) = (\sqrt{x+a} - \sqrt{x}) + \sqrt{\pi + x} \tag{4}$$

and may suffer from *catastrophic numerical cancellation* [8] when the x value is considerably greater than a. The user may decide to specialize code generation rules for this particular expression, stabilizing it through rationalization. Without modifying the actual model, $g(x)$ the rationalization for differences of square roots[3] is inserted into the exportation rules, as in Listing 10. The rules are valid only for the current user script. For more insight about `__to_c` and `__to_c_impl`, refer to the software manual.

[3]i.e.: $\sqrt{\phi(\cdot)} - \sqrt{\psi(\cdot)} = \dfrac{\phi(\cdot) - \psi(\cdot)}{\sqrt{\phi(\cdot)} + \sqrt{\psi(\cdot)}}$

Listing 10: Conditioning in exporting function

```
# Model
a = CAS.declare "PARAM_A"

g = (CAS.sqrt(x + a)— CAS.sqrt(x)) + CAS.sqrt(CAS::Pi + x)

# Particular Code Generation for difference between square roots.
module CAS
  class Diff
    alias :__to_c_impl_old :__to_c_impl

    def __to_c_impl(v)
      if @x.is_a? CAS::Sqrt and @y.is_a? CAS::Sqrt
        "(#{@x.x.__to_c(v)} + #{@y.x.__to_c(v)}) / " +
        "( #{@x.__to_c(v)} + #{@y.__to_c(v)} )"
      else
        self.__to_c_impl_old(v)
      end
    end
  end
end

CAS::CLib.create "g_impl" do
  define "PARAM_A()", 1.0   # Arbitrary value for PARAM_A
  define "M_PI", Math::Pi
  implements_as "g_impl", g
end

puts g
# => ((sqrt((x + PARAM_A()))— sqrt(x)) + sqrtπ(( + x)))
```

It should be noted the separation between the *model*, which does not contain stabilization, and the *code generation rule*. For this particular case, the code generation rule in Listing 10 overloads the predefined one, in order to obtain the conditioned code. Obviously, the user can decide to apply directly the conditioning on the model itself, but this may change the calculus behavior in further manipulation.

Listing 11: `g_impl` Header

```c
// Header file for library: g_impl.c

#ifndef g_impl_H
#define g_impl_H

// Standard Libraries
#include <math.h>

// Local Libraries


// Definitions
#define PARAM_A() 1.0
#define M_PI 3.141592653589793

// Functions
double g_impl(double x);

#endif // g_impl_H
```

Listing 12: `g_impl` Source

```c
// Source file for library: g_impl.c

#include "g_impl.h"

double g_impl(double x) {
  double __t_0 = PARAM_A();
  double __t_1 = (x + __t_0);
  double __t_2 = sqrt(__t_1);
  double __t_3 = sqrt(x);
  double __t_4 = (__t_1 + x) / ( __t_2 + __t_3
       );
  double __t_5 = (M_PI + x);
  double __t_6 = sqrt(__t_5);
  double __t_7 = (__t_4 + __t_6);

  return __t_7;
}

// end of g_impl.c
```

3.2 Using the module as interface

As example, an implementation of an algorithm that estimates the *order of convergence* for trapezoidal integration scheme [18] is provided, using the symbolic differentiation as interface.

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

$$I_n(a,b) = h\left(\frac{f(a)+f(b)}{2} + \sum_{k=1}^{n-1} f\left(a + k\, h\right)\right) \tag{5}$$

with $h = (b-a)/n$, where n mediates the step size of the integration. When exact primitive $F(x)$ is known, approximation error is:

$$E[n] = F(b) - F(a) - I_n(a,b) \tag{6}$$

The error has an asymptotic expansion of the form:

$$E[n] \propto C\, n^{-p} \tag{7}$$

where p is the convergence order. Using a different value for n, for example $2\,n$, the ratio 8 takes the approximate vale:

$$\frac{E[n]}{E[2\,n]} \approx 2^p \quad \rightarrow \quad p \approx \log_2\left(\frac{E[n]}{E[2\,n]}\right) \tag{8}$$

The Listings 13 and 14 contain the implementation of the described procedure using the proposed gem and the well known *Python* [14] library *SymPy* [11].

Listing 13: Ruby version

```ruby
require 'Mr.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 14: 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
```

3.3 ODE Solver with Taylor's series

In this example, a solving step for a specific ordinary differential equation (ODE) using Taylor's series method [4] is derived. Given an ODE in the form:

$$y'(x) = f\left(x, y(x)\right) \tag{9}$$

the integration step with order n has the form:

$$y(x+h) = y(x) + h\,y'(x) + \cdots + \frac{h^n}{n!}\,y^{(n)}(x) + E_n(x) \tag{10}$$

where it is possible to substitute equation 9:

$$y^{(i)}(x) = \frac{\partial y^{(i-1)}(x)}{\partial x} + \frac{\partial y^{(i-1)}(x)}{\partial y} y'(x) \tag{11}$$

For this algorithm, three methods are defined. The first evaluates the factorial, the second evaluates the list of required derivatives, and the third returns the integration step in a symbolic form. The result of the third method is transformed in a C function. In this particular case, the ODE is $y' = xy$. For the resulting C code of Listing 15, refer to the online version of the examples.

Listing 15: Generator for ODE integration step

```
$x, $y, $h = CAS::vars :x, :y, :h
# Evaluates n!
def fact(n); (n < 2 ? 1 : n * fact(n — 1)); end
# Evaluates all derivatives required by the order
def coeff(f, n)
  df = [f]
  for _ in 2..n
    df << df[—1].diff($x).simplify + (df[—1].diff($y).simplify * df[0])
  end
  return df
end
# Generates the symbolic form for a Taylor step
def taylor(f, n)
  df = coeff(f, n)
  y = $y
  for i in 0...df.size
    y = y + (($h ** (i + 1))/(fact(i + 1)) * df[i])
  end
  return y.simplify
end

# Example function for the integrator
f = $x * $y
# Exporting a C function
clib = CAS::CLib.create "taylor" do
  implements_as "taylor_step", taylor(f, 4)
end
```

Other examples are available online[4]: (a) adding a user defined `CAS::Op` that implements the sign$(\cdot)$ function with the appropriate optimized C generation rule; (b) exporting the operation as a continuous function through overloading or substitutions; (c) performing a symbolic Taylor's series; (d) writing an exporter for the LaTeX language; (e) a Newton-Raphson algorithm using automatic differentiation plugin.

[4]`http://bit.ly/Mr_CAS_examples`

4 Impact

Mr.CAS is a midpoint between a CAS and an ASD library. It allows one to manipulate expressions while maintaining the complete control on how the code is exported. Each rule is overloaded and applied run-time, without the need of compilation. Each user's model may include the mathematical description, code generation rules and high level logic that should be intrinsic to such a rule—for example, exporting a Hessian as pattern instead of matrix.

Our research group is including `Mr.CAS` in a solver for optimal control problem with indirect methods, as interface for problems description [3].

As a long term ambitious impact, this library will become a complete CAS for *Ruby* language, filling the empty space reported by *SciRuby* for symbolic math engines.

5 Conclusions

This work presents a pure *Ruby* library that implements a minimalistics CAS with automatic and symbolic differentiation that is aimed at code generation and meta-programming. Although at an early developing stage, *Mr.CAS* has promising feature, some of them shown in Section 3. Also, this is the only gem that implements symbolic manipulation for this language.

Language features and lack of dependencies simplify the use of the module as interface, extending model definition capabilities for numerical algorithms. All core functionalities and basic mathematics are defined, with the plan to include more features in next releases. Reopening a class guarantees a *liquid* behaviour, in which users are free to modify core methods at their needs.

Library is published in *rubygems.org* repository and versioned on *github.com*, under MIT license. It can be included easily in projects and in inline interpreter, or installed as a standalone gem.

References

[1] Michael Bartholomew-Biggs, Steven Brown, Bruce Christianson, and Laurence Dixon. Automatic differentiation of algorithms. *Journal of Computational and Applied Mathematics*, 124(1):171–190, 2000.

[2] Ravil Bayramgalin. Symbolic. `https://github.com/brainopia/symbolic`, 2012. Online; commit: `bbd588e8676d5bed0017a3e1900ebc392cfe35c3`.

[3] Francesco Biral, Enrico Bertolazzi, and Paolo Bosetti. Notes on numerical methods for solving optimal control problems. *IEEJ Journal of Industry Applications*, 5(2):154–166, 2016.

[4] J.C. Butcher. *Numerical Methods for Ordinary Differential Equations, Second Edition.* 2008.

[5] Ondrej Certik, Dale Lukas Peterson, Thilina Bandara Rathnayake, et al. Symengine. https://github.com/symengine/symengine.rb, 2016. Online; commit: 8cf9e08c972085788c17da9f4e9f22898e79d93b.

[6] Joel S Cohen. *Computer algebra and symbolic computation: Mathematical methods*. Universities Press, 2003.

[7] David Flanagan and Yukihiro Matsumoto. *The ruby programming language*. O'Reilly Media, Inc., 2008.

[8] N. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, 2002.

[9] ISO/IEC 30170 – Information technology – Programming languages – Ruby. Standard, International Organization for Standardization, Geneva, CH, April 2000.

[10] John Lees-Miller. Rucas. https://github.com/jdleesmiller/rucas, 2010. Online; commit: 047a38b541966482d1ad0d40d2549683cf193082.

[11] Christopher Smith, Aaron Meurer, Mateusz Paprocki, et al. Sympy 1.0. https://doi.org/10.5281/zenodo.47274, 2016. Online; accessed: 2016-10-15.

[12] Kazuaki Tanaka, Avinash Dev Nagumanthri, and Yukihiro Matsumoto. mruby–rapid software development for embedded systems. In *15th International Conference on Computational Science and Its Applications (ICCSA)*, pages 27–32. IEEE, 2015.

[13] John E Tolsma and Paul I Barton. On computational differentiation. *Computers & chemical engineering*, 22(4):475–490, 1998.

[14] Guido Van Rossum and Fred L Drake. *The Python language reference manual*. Network Theory Ltd., 2011.

[15] Joachim Von Zur Gathen and Jürgen Gerhard. *Modern computer algebra*. Cambridge university press, 2013.

[16] Andreas Wächter and Lorenz T. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming*, 106(1):25–57, 2006.

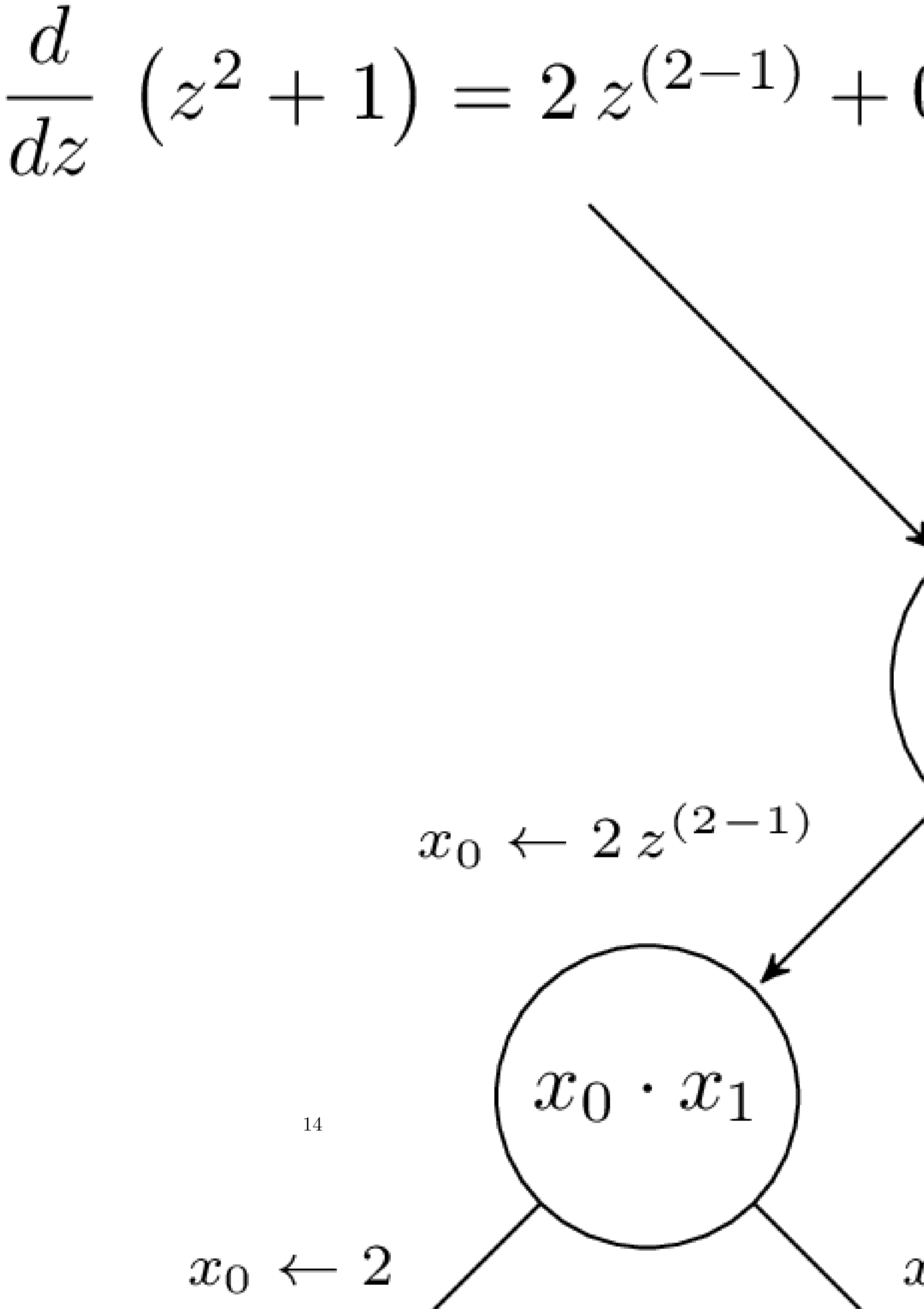
[17] Andreas Wächter and Carl Laird. Ipopt-an interior point optimizer. https://projects.coin-or.org/Ipopt, 2009. Online; accessed: 2016-11-28.

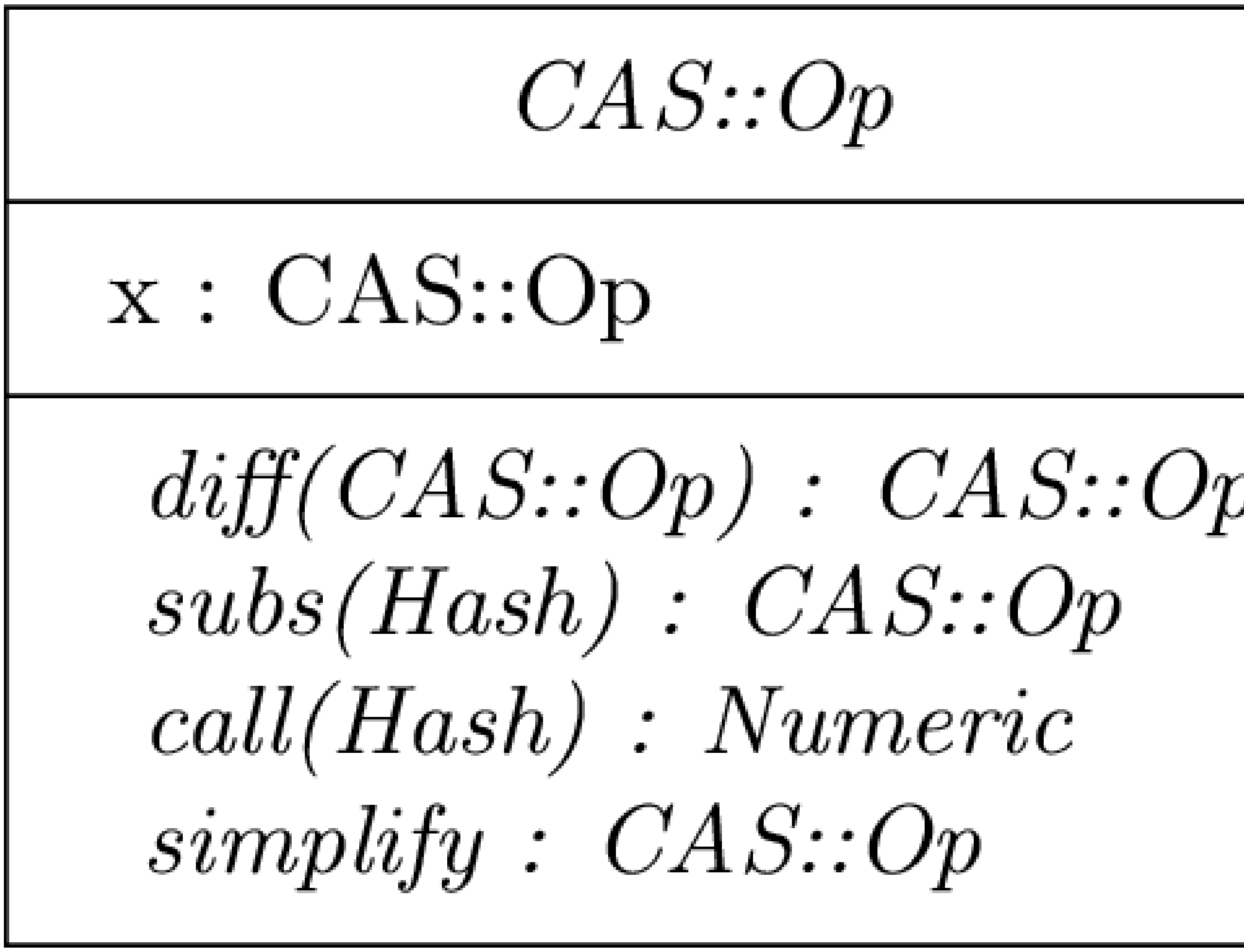
[18] J André C Weideman. Numerical integration of periodic functions: A few examples. *The American mathematical monthly*, 109(1):21–36, 2002.

Current code version

Nr.	**Code metadata description**	**Please fill in this column**
C1	Current code version	0.2.7
C2	Permanent link to code/repository used for this code version	github.com/ ElsevierSoftwareX/SOFTX-D-17-00013
C3	Legal Code License	MIT
C4	Code versioning system used	*git* (GitHub)
C5	Software code languages, tools, and services used	*Ruby* language
C6	Compilation requirements, operating environments	*Ruby* $\geq 2.x$
C7	If available Link to developer documentation/manual	rubydoc.info/gems/Mr.CAS
C8	Support email for questions	info@ragni.me

Table 1: Code metadata (mandatory)


$\frac{d}{dz}\left(z^2+1\right) = 2\,z^{(2-1)} +$
$x_0 \leftarrow 2\,z^{(2-1)}$
$x_0 \cdot x_1$
$x_0 \leftarrow 2$


CAS::Op
x : CAS::Op
diff(CAS::Op) : CAS::Op
subs(Hash) : CAS::Op
call(Hash) : Numeric
simplify : CAS::Op