# *Mr.CAS*- A Minimalistic (pure) *Ruby* CAS for Fast Prototyping and Code Generation

Matteo Ragni[a]

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

**Abstract**

There are complete **Computer Algebra System** (CAS) systems on the market with complete solutions for manipulation of analytical models. But exporting a model to a given target language is often a rigid procedure that requires some manual post-processing, even with a good software. This work presents a *Ruby* library that exposes core CAS capabilities—i.e. simplification, substitution, evaluation, etc. The library aims at rapid prototyping of numerical interfaces, and code generation for different target languages, separating mathematical expression from code generation rules supporting best practices for numerical conditioning. The library is implemented in pure *Ruby* language and is compatible with most *Ruby* interpreters.

*Keywords:* CAS, code-generation, Ruby

## 1. Motivation and significance

*Ruby* [1] 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*) [2] has been published on *GitHub*

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

by Matsumoto, in 2014. The new interpreter is a lightweight implementation, aimed at both low power devices and PC, and complies with the standard[3]. *mRuby* has a completely new API, and it is designed to be embedded in complex projects as a front-end interface—e.g., a numerical optimization suite may use *mRuby* to 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 symbolic differentiation** (ASD) [4] 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 though make it intriguing for use in the scientific/technical field. An ASD-capable gem would prove a foundamental step in this direction, including the support for flexible code generation for high-level software—e.g., IPOPT [5, 6].

*Mr.CAS*[1] is a gem implemented in pure *Ruby* that supports symbolic differentiation (SD) and some computer algebra operations [7]. The library aims at:

- support rapid prototyping of numerical algorithms and code generation to different target languages;
- when dealing with mathematical models, support a clean and separate formulation of conditioning rules for numerical issues, in order to support more robust code generation;
- create a complete open-source CAS system for the standard *Ruby* language, as a long-term effort.

[1] Minimalistic Ruby Computer Algebra System

Other CAS libraries for *Ruby* are available:

***Rucas*** [8], ***Symbolic*** [9] : milestone gems, yet at early stage and with discontinued development status. Both offer basic simplification routines, although they lack differentiation.

***Symengine*** [10] : 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, at the moment it seems not working using the *Ruby* 2.x interpreter.

In Section 2, *Mr.CAS* container and tree structure is explained in detail and applied to basic CAS tasks. In Section 3, two 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 some computer algebra routines such as *simplifications* and *substitutions*. When gem is required, it overloads methods of `Fixnum` and `Float` classes, making them compatible with foundamental 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:

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

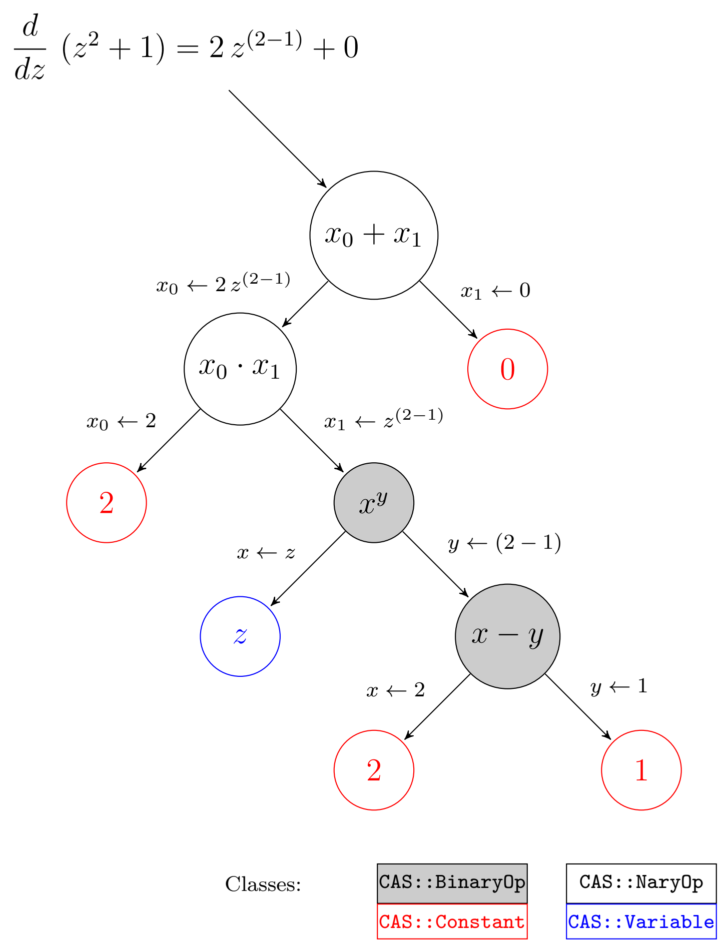


Figure 1: Tree of the expression derived in Listing 1

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`.

Figure 1 contains a graphical representation. The different kind of containers allows to introduce some properties — i.e. *associativity* and *commutativity* for sums and multiplications [11]. Each container exposes the sub-tree as instance properties. Containers interfaces and inheritances are shown in Figure 2.

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. As for now, those leaves exemplify only real scalar expressions, with definition of complex, vectorial and matricial extensions as milestones for the next major release.

SD (`CAS::Op#diff`) crosses 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.

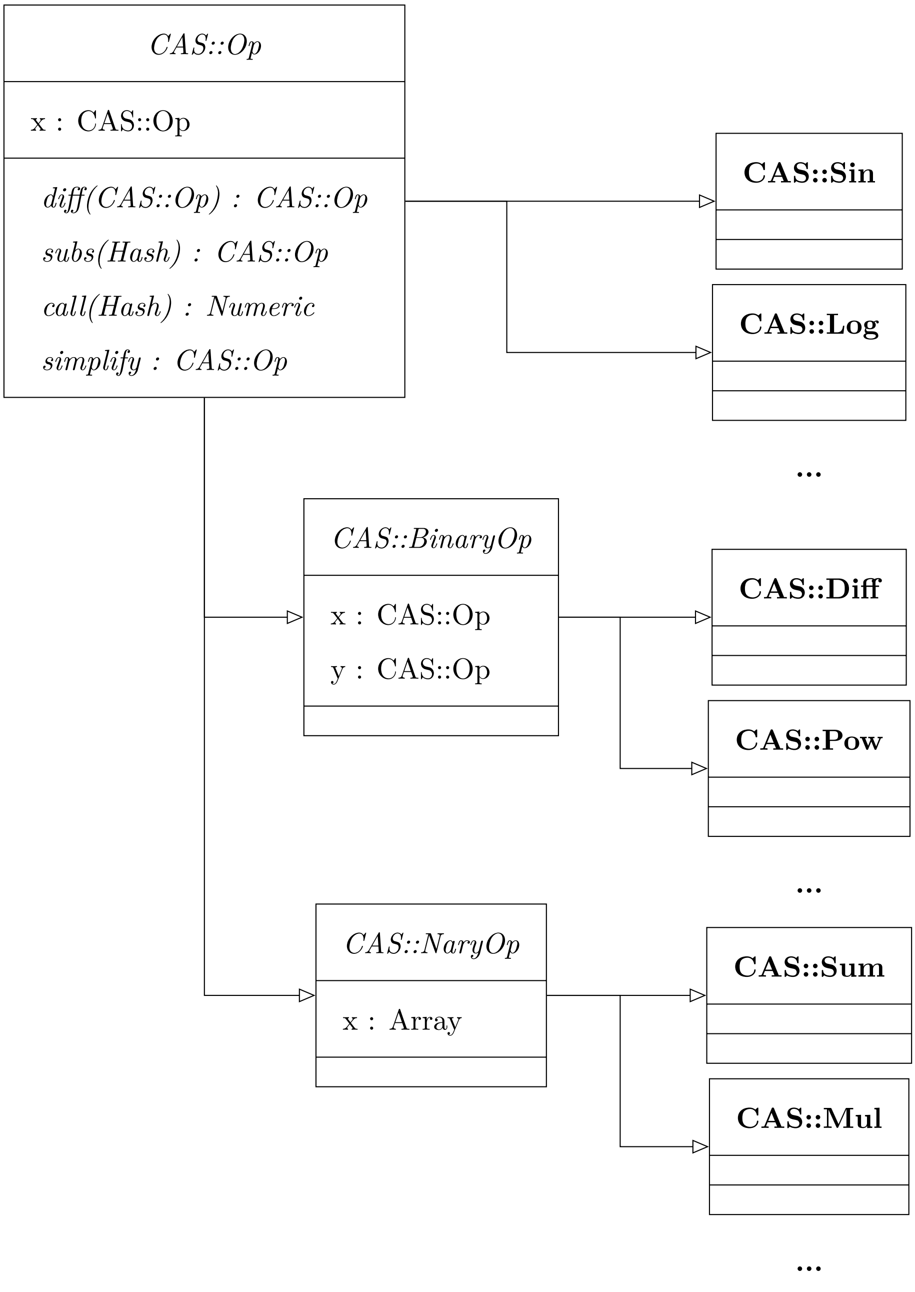


Figure 2: Simplified version of classes interface and inheritance

*2.2. Software Functionalities*

*2.2.1. Basic Functionalities*

*No additional dependencies are required.* The gem can be installed through *rubygems.org* provider[2]. Functionalities must be required runtime using the Kernel method: `require r.CAS`. All methods and classes are incapsulated in the module `CAS`.

**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:

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 plugin and exploits properties of dual numbers to efficiently perform differentiation, see [12] details. This differentiation strategy is useful in case of complex expressions, when explicit derivative's tree may exceed the call stack depth, that is platform dependent.

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

[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 independent variables values are provided in a feed dictionary. The graph is reduced recursively to a single numeric value:

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` — e.g. $f(\cdot) \geq g(\cdot)$. This allow the definition of 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))
```

### *2.2.2. Metaprogramming 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* (`Proc` objects):

Listing 5: Graph evaluation example

```ruby
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 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, transforming it in a *closure* reduces the execution time per iteration.

Code generation should be flexible enough to export expressions' trees 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 code generation plugin

```ruby
# 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*

In this example a model is exported as C library. `c-opt` plugin implements advanced features such as code optimization and generation of libraries.

The library `example` implements the model:

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

Expression $g(x)$ belongs to a external object, declared as `g_impl`, and its interface is described in `g_impl.h` header. The code is optimized: the intermediate operation $x^y$ is evaluated once, even if appears twice in our model. The C function that implements our model $f(x, y)$ is declared with the token `f_impl`. The exporter uses as default type `double` for variables and function returned values.

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
```

Library created by `CLib` contains the following code:

Listing 8: C Header

```c
// 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

```c
// 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 cancellation* [13]. Users can specialize code generation rules for this particular expression, conditioned through rationalization and instead of modifying the model $g(x)$, in Listing 10, the rationalization is extended to all differences of square roots [3]. For more insight about `__to_c` and `__to_c_impl` please refer to the software manual.

Listing 10: Conditioning in exporting function

```ruby

# 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
```

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

```ruby
  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
```

It should be noted the **separation between the model** — that does not contain conditioning — **and the code generation rule** — that overloads, for this particular case and this particular language, the predefined code generation rule. Obviously, a user can decide to apply directly the conditioning on the model. The result of Listing 10 is reported:

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 extimates the *order of convergence* for trapezoidal integration scheme [14] is provided, using the symbolic differentiation as interface.

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

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

with $h = (b-a)/n$, where $n$ mediates the integration's step size. 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:

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

Following Listings contain the implementation of the described procedure using the proposed gem and the well known *Python* [15] library *SymPy* [16].

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
```

## 4. Impact

*Mr.CAS* is a midpoint between a CAS and an ASD library. It allows to manipulate expressions while mantaining the complete control on how the code is exported. Each rule is overloaded and applied runtime, without the need of compilation. Each user's model may include the mathematical description, code generation rules and high level logic that should be intrisic to such a rule — e.g. exporting gradients as **patterns** instead of matrices.

Our research group is including `Mr.CAS` in a solver for optimal control problem with indirect methods, as interface for problems' description [17].

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 metaprogramming. 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 and 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.

## Acknowledgements

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

[1] D. Flanagan, Y. Matsumoto, The ruby programming language, O'Reilly Media, Inc., 2008.

[2] K. Tanaka, A. D. Nagumanthri, Y. Matsumoto, mruby–rapid software development for embedded systems, in: 15th International Conference on Computational Science and Its Applications (ICCSA), IEEE, 2015, pp. 27–32.

[3] ISO/IEC 30170 – Information technology – Programming languages – Ruby, Standard, International Organization for Standardization, Geneva, CH (april 2000).

[4] J. E. Tolsma, P. I. Barton, On computational differentiation, Computers & chemical engineering 22 (4) (1998) 475–490.

[5] A. Wächter, C. Laird, Ipopt-an interior point optimizer, `https://projects.coin-or.org/Ipopt`, online; accessed: 2016-11-28 (2009).

[6] A. Wächter, L. T. Biegler, On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming, Mathematical Programming 106 (1) (2006) 25–57.

[7] J. Von Zur Gathen, J. Gerhard, Modern computer algebra, Cambridge university press, 2013.

[8] J. Lees-Miller, Rucas, `https://github.com/jdleesmiller/rucas`, online; commit: `047a38b541966482d1ad0d40d2549683cf193082` (2010).

[9] R. Bayramgalin, Symbolic, `https://github.com/brainopia/symbolic`, online; commit: `bbd588e8676d5bed0017a3e1900ebc392cfe35c3` (2012).

[10] O. Certik, D. L. Peterson, T. B. Rathnayake, et al., Symengine, `https://github.com/symengine/symengine.rb`, online; commit: `8cf9e08c972085788c17da9f4e9f22898e79d93b` (2016).

[11] J. S. Cohen, Computer algebra and symbolic computation: Mathematical methods, Universities Press, 2003.

[12] M. Bartholomew-Biggs, S. Brown, B. Christianson, L. Dixon, Automatic differentiation of algorithms, Journal of Computational and Applied Mathematics 124 (1) (2000) 171–190.

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

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

[15] G. Van Rossum, F. L. Drake, The python language reference manual, Network Theory Ltd., 2011.

[16] C. Smith, A. Meurer, M. Paprocki, et al., Sympy 1.0, https://-doi.org/10.5281/zenodo.47274, online; accessed: 2016-10-15 (2016).

[17] F. Biral, E. Bertolazzi, P. Bosetti, Notes on numerical methods for solving optimal control problems, IEEJ Journal of Industry Applications 5 (2) (2016) 154–166.

## 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/Mr.CAS |
| 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)