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

This work presents a new *Ruby* library for symbolic and automatic differentiation, that exposes minimalistic CAS capabilities — i.e. simplifications, substitutions, evaluations, etc. Library aims at rapid prototyping of numerical interfaces and code generation for different target languages, separating mathematical expression from exportation rules — e.g. models from numerical conditioning best practices.

The library is implemented in pure *Ruby* language and compatible with all *Ruby* interpreter flavours.

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 (also known as *Matz*), 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*) [2] has been published on *GitHub* by Matsumoto, in 2014. The new interpreter is a lightweight implementation, aimed at both low power devices and PC, and complies with

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

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 get problem input definitions.

The *Ruby* code-base exposes a large set of utilities in core and standard library, that can be furthermore expanded through modules, contained in *gems*. Even if a high number of gems are deployed and available, there is no module that implements an **automatic symbolic differentiation** (ASD) [4] engine that handles basic computer algebra routines, compatible with all 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 ASD-capable gem is the foundamental step to rapidly develop specific code generators for well known software — e.g. IPOPT [5].

The module described in this work, is a gem implemented in pure *Ruby* code — compatible with all standardized interpreters — that is able to perform symbolic differentiation (SD) and some computer algebra operations [6]. 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* conditioning rules for numerical issues, changing on request how the code is exported, and how expressions are formulated in the target language;
- *separate mathematical expressions from numerical conditioning and workarounds*;

- 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* [7], *Symbolic* [8]** gems at early stage and with discontinued development status; they offer basic simplification routines. There is no differentiation method, but it is one of the milestones.

***Symengine* [9]** is a wrapper for the C++ library *symengine*. The backend library is very complete, but it is compatible only with the RVM *Ruby* interpreter and has several dependencies. At the moment, the *SciRuby* [10] project reports the gem as broken, and removed it from its codebase. From a direct test, when performing SD of an arbitrary function, the engine always returns `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 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 linked tree, that is the mathematical equivalent of function composition:

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

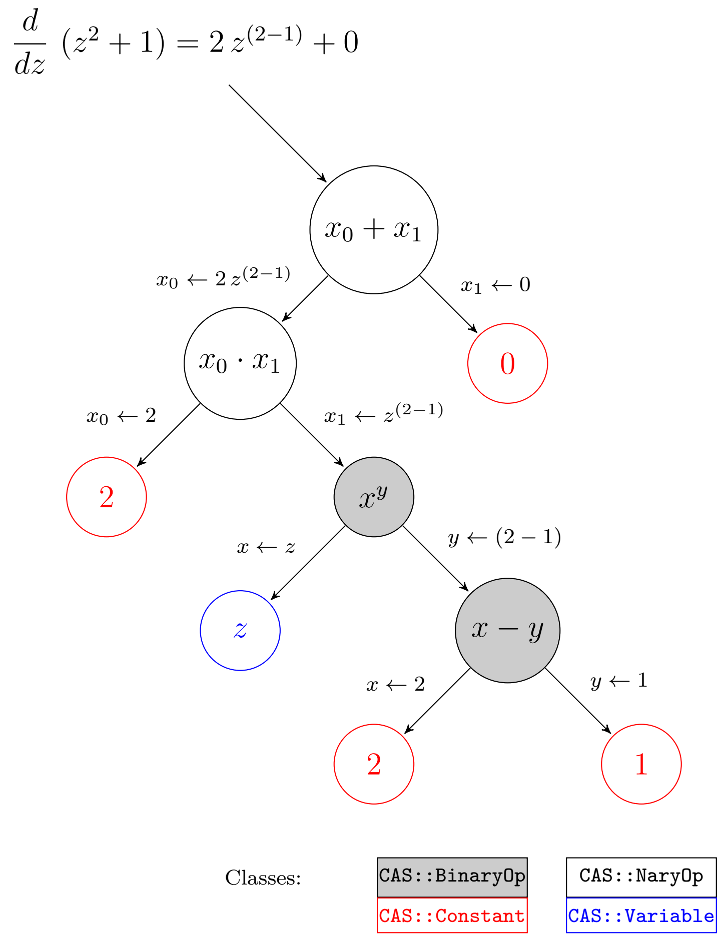

Figure 1: Tree of the expression derived in listing 1

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. Single argument operations — e.g. $\sin(\cdot)$ — have as closest parent the `CAS::Op` class, that links to one sub-tree. Expressions with two arguments — e.g. difference or exponential function — inherit from `CAS::BinaryOp`, that links to two sub-tree. Operations with arbitrary number of arguments — e.g. sum and product

— have as parent the `CAS::NaryOp`[1], that links to an arbitrary number of sub-tree. 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 leafes 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 leafes 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.

2.2. Software Functionalities

2.2.1. Software installation and prerequisites

No additional dependencies are required. The gem can be installed through *rubygems.org* provider[2]. Functionalities must be required runtime using the

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

[2] `gem install ragni-cas`

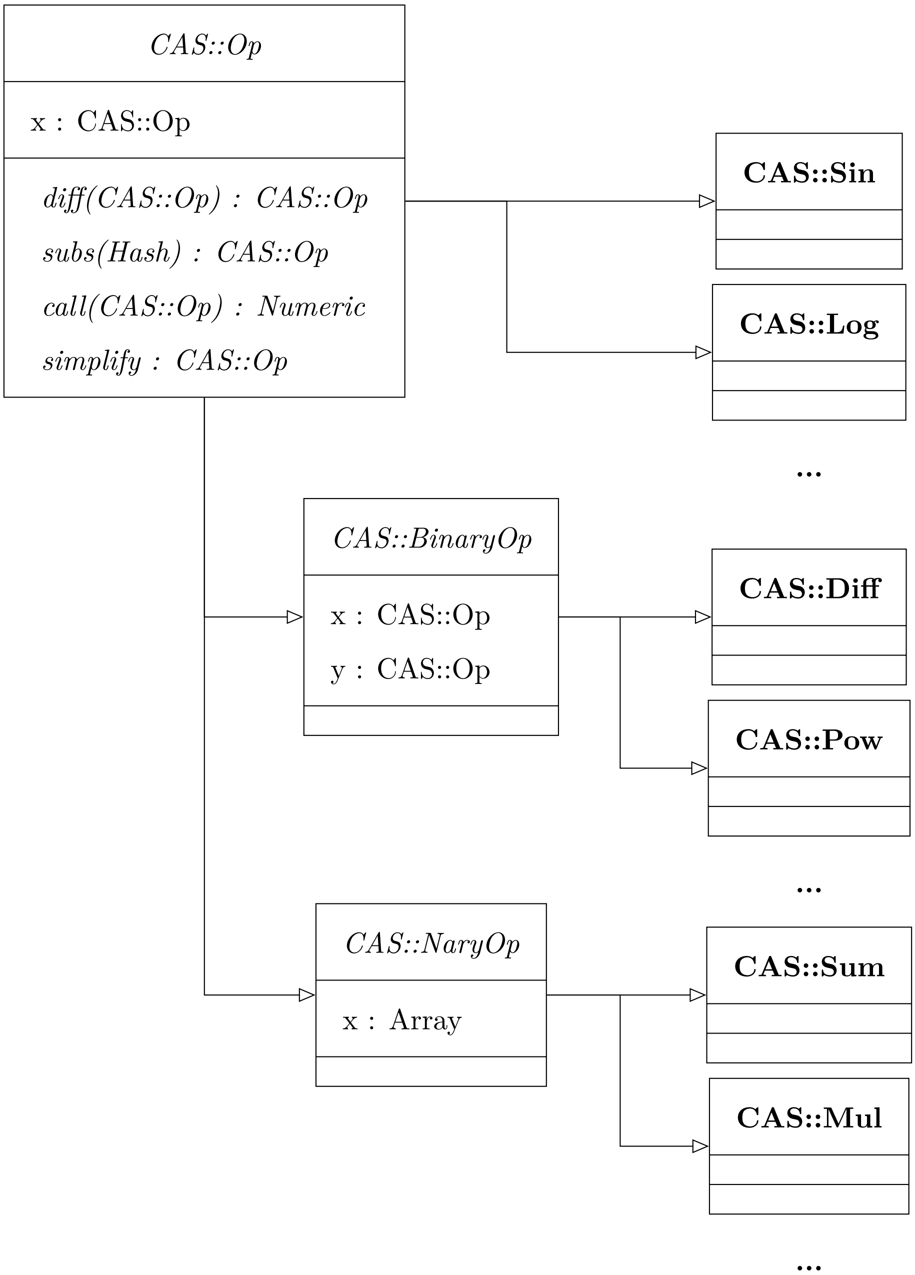

Figure 2: Simplified version of classes interface and inheritance

Kernel method: `require 'ragni-cas'`. All methods and classes are incapsulated in the module `CAS`.

2.2.2. Basic Functionalities

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
# => 2 * z ^ (2 — 1) + 0
g = CAS.declare :g, f      # creates implicit expression
g.diff(z)                  # derivative w.r.t. z
# => (z ^ (2 — 1) * 2) * Dg[0](z ^ 2)
```

Automatic differentiation (AD) is included as plugin and exploits dual numbers [12]. 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**:

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 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             # define a symbolic expression
f.call x => 2              # evaluate for x = 2
# => 5
```

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))
```

2.2.3. *Metaprogramming and Code-Generation*

The library is developed explicitly for **metaprogramming** and **generation of code**. Expressions can be exported as source code or used as prototypes for callable *closures* (`Proc` objects):

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 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
# 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)
```

3. Illustrative Examples

3.1. Code Generation as C Library

TIn 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

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

# 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 class `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.: $\sqrt{\phi(\cdot)} - \sqrt{\psi(\cdot)} = \dfrac{\phi(\cdot) - \psi(\cdot)}{\sqrt{\phi(\cdot)} + \sqrt{\psi(\cdot)}}$

```
  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

clib = 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:

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

This error shows a direct relation:

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

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

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

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

Listing 13: 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 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

There are different complete CAS systems on the market, with complete solutions for analysis of analitical models. But exporting a model, for optimization or any other research activity, requires a lot of work, even with a good CAS software.

This library is a midpoint between a CAS and an AD 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 `ragni-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. This will require time, and the gem's MIT license allows everyone to contribute to the project.

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, the module 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: Computational Science and Its Applications (ICCSA), 2015 15th International Conference on, 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, L. Biegler, Ipopt-an interior point optimizer (2009).

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

[7] J. Lees-Miller, Rucas, https://github.com/jdleesmiller/rucas (2010).

[8] R. Bayramgalin, Symbolic, https://github.com/brainopia/symbolic (2012).

[9] O. C. D. L. Peterson, T. B. Rathnayake, et al., Symengine, https://github.com/symengine/symengine.rb (2016).

[10] T. R. S. Foundation, Sciruby: Tools for scientific computing in ruby, http://sciruby.com (2013).

[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, 2nd Edition, Society for Industrial and Applied Mathematics, 2002. arXiv:http://epubs.siam.org/doi/pdf/10.1137/1.9780898718027, doi:10.1137/1.9780898718027.
URL http://epubs.siam.org/doi/abs/10.1137/1.9780898718027

[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: Sympy 1.0 (mar 2016). `doi:10.5281/zenodo.47274`.
URL `https://doi.org/10.5281/zenodo.47274`

[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/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)