Mr.CAS—A minimalistic (pure) Ruby CAS for fast prototyping and code generation

Matteo Ragni

Department of Industrial Engineering, University of Trento, 9, Sommarive, Povo di Trento, Italy

A R T I C L E   I N F O

Article history:
Received 31 January 2017
Received in revised form 9 May 2017
Accepted 12 May 2017

Keywords:
CAS
Code-generation
Ruby

A B S T R A C T

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.

© 2017 The Author. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

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] 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 [3]. 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) [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 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 [5,6].

Mr.CAS is a gem implemented in pure Ruby that supports symbolic differentiation (SD) and fundamentals computer algebra...
operations [7]. 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.

Other CAS libraries for Ruby are available:

Rucas, Symbolic: [8,9] milestone gems, yet at an 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, 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:

\[ (f \circ g) \]

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. \( \sin(x) \).

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 allow to introduce some properties—i.e. associativity and commutativity [11]. 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:

\[ (f \circ g)' = (f' \circ g) \cdot g'. \]  

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. 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 |
|------------------------------------|
| \[ z = \text{CAS}.\text{vars} \{\text{z}\} \] # creates a variable |
| \[ f = z ** 2 + 1 \] expression # define a symbolic |
| \[ f.\text{diff}(z) \] # => \( \text{((z)''(2 - 1)) + 2 \cdot 1 + 0} \) # derivative v.r.t. z |
| \[ g = \text{CAS}.\text{declare} \{\text{g}, f\} \] expression # creates implicit |
| \[ g.\text{diff}(x) \] # => \( \text{(((z)''(2 - 1)) + 2 \cdot 1 + 0) \cdot Dg[0](((z)''(2) + 1))} \) # derivative v.r.t. z |

2 gem install Mr.CAS.

\[ \frac{d}{dx} (x^2 + 1) = 2x^{(2-1)} + 0 \]

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:

\[ (f \circ g) \]

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. \( \sin(x) \).

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 allow to introduce some properties—i.e. associativity and commutativity [11]. 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:

\[ (f \circ g)' = (f' \circ g) \cdot g'. \]  

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. 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 |
|------------------------------------|
| \[ z = \text{CAS}.\text{vars} \{\text{z}\} \] # creates a variable |
| \[ f = z ** 2 + 1 \] expression # define a symbolic |
| \[ f.\text{diff}(z) \] # => \( \text{((z)''(2 - 1)) + 2 \cdot 1 + 0} \) # derivative v.r.t. z |
| \[ g = \text{CAS}.\text{declare} \{\text{g}, f\} \] expression # creates implicit |
| \[ g.\text{diff}(x) \] # => \( \text{(((z)''(2 - 1)) + 2 \cdot 1 + 0) \cdot Dg[0](((z)''(2) + 1))} \) # derivative v.r.t. z |

2 gem install Mr.CAS.
Fig. 2. Reduced version of classes interface and inheritance. The figure depicts
the basic abstract class `CAS::Op`, from which the single argument
operations inherit. `CAS::Op` is also the ancestor for other kind of containers, namely the `CAS::BinaryOp` and `CAS::NaryOp`, the models of container with two and more arguments.

Automatic differentiation (AD) is included as a plugin and exploits the properties of dual numbers to efficiently perform differentiation, see [12] 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 sub-graph, i.e., some trigonometric identities—are not defined for now, but can be easily achieved through substitutions, as shown in Listing 2.

Listing 2: Simplification example

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

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() ≥ g()`. This allow the definition of piecewise functions, in `CAS::Piecewise`. Internally, `max()` and `min()` functions are declared as operations that inherits from `CAS::Piecewise`—for example, `max(f(), g())`. Usage is shown in Listing 4.

Listing 4: Expressions and Piecewise functions

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

```
module CAS
  ...  
  CAS::Variable => Proc.new { "#{name}" }
  CAS::Sin => Proc.new { "sin(#(x.to_fortran))" },
end
end
```

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:

$$f(x, y) = x^p + g(x) \log(\sin(x^p))$$  \hfill (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 $x^p$ is evaluated once, even if it 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

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

# Code Generation
\( g.c_name = 'g_impl' \) # g token
include_local "g_impl" # g header
implement_as "f_impl", f # token for f

CAS::CLib.create "example" do
  # Model
  g = CAS.declare :g, x
  \( g.c_name = 'g_impl' \) # g token
  include_local "g_impl" # g header
  implement_as "f_impl", f # token for f
end
```

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

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

and may suffer from catastrophic numerical cancellation [13] 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 is inserted into the exported 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.

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.

3.2. Using the module as interface

As example, an implementation of an algorithm that estimates 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 for the interval $[a, b]$ is:

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

$$E[n] = F(b) - F(a) - I_n(a, b).$$  \hfill (6)

The error has an asymptotic expansion of the form:

$$E[n] \propto C n^{-p}$$  \hfill (7)

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

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

The Listings 13 and 14 contain the implementation of the described procedure using the proposed gem and the well known Python SymPy [16].

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 [17] is derived. Given an ODE in the form:

$$y’(x) = f(x, y(x))$$  \hfill (9)

the integration step with order $n$ has the form:

$$y(x + h) = y(x) + h y’(x) + \cdots + \frac{h^n}{n!} y^{(n)}(x) + E_n(x)$$  \hfill (10)

where it is possible to substitute Eq. (9):

$$y^{(i)}(x) = \frac{\partial y^{(i-1)}(x)}{\partial x} + \frac{\partial y^{(i-1)}(x)}{\partial y} y’(x).$$  \hfill (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 $y’ = x y$. For the resulting C code of Listing 15, refer to the online version of the examples.

Other examples are available online: (a) adding a user defined CAS \ opp 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. Impact

Mr.CAS is a midpoints 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 [18].

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.

---

3. I.e. $\sqrt{\varphi(n)} - \sqrt{\varphi(i)} = \frac{\varphi(i)-\varphi(1)}{\sqrt{\varphi(i)} - \sqrt{\varphi(1)}}$.

4. http://bit.ly/Mr_CAS_examples.
### Listing 8: C Header

```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
#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;
}
```

### Listing 10: Conditioning in exporting function

```c
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
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)) + sqrt(π(( + x)))
```

### Listing 11: g_impl Header

```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
#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;
}
```
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
```

Listing 15: Generator for ODE integration step

```ruby
$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
```

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 behavior, 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] Flanagan D, Matsumoto Y. The Ruby programming language. O’Reilly Media, Inc.; 2008.
[2] Tanaka K, Nagumanthri AD, Matsumoto Y. mrbry–Rapid Software Development for Embedded Systems. In: 15th International Conference on Computational Science and Its Applications. IEEE; 2015. p. 27–32.
[3] ISO/IEC 30170:2012, ISO/IEC 30170–Information technology – Programming languages – Ruby, 2012, International organization for standardization, 2000, Geneva, CH.
[4] Tolsma JE, Barton PI. On computational differentiation. Comput Chem Eng 1998;22(4):475–90.
[5] Wachter A, Laird C. IPOPT—an interior point OPTimizer, Online; 2009. https://projects.coin-or.org/Ipopt [Accessed: 28 November 2016].
[6] Wächter A, Biegler LT. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. Math. Program. 2006;106(1):25–57.

[7] Von Zur Gathen J, Gerhard J. Modern computer algebra. Cambridge University Press; 2013.

[8] Lees-Miller J, RuCas, GitHub repository, GitHub, Online; commit: 047a38b541966482d1ad0d40d2549683cf193082; 2010. https://github.com/jdleesmiller/rucas.

[9] Bayramgalin R, Symbolic, GitHub repository, GitHub, Online; commit: bbd588e987d5bed0017a3e1900b392e3be435c3; 2012. https://github.com/brainopia symbolic.

[10] Certik O, Peterson DL, Rathnayake TB, SymEngine, GitHub Repository, GitHub, Online; commit: 8cf9e08c972088788c17da9f4e9f22b89e79d93b; 2016. https://github.com/symengine/symengine.rb.

[11] Cohen JS. Computer algebra and symbolic computation: mathematical methods. Universities Press; 2003.

[12] Bartholomew-Biggs M, Brown S, Christianson B, Dixon L. Automatic differentiation of algorithms. J Comput Appl Math 2000;124(1):171–90.

[13] Higham N. Accuracy and stability of numerical algorithms. Society for Industrial and Applied Mathematics; 2002.

[14] Weideman JAC. Numerical integration of periodic functions: a few examples. The American Mathematical Monthly 2002;109(1):21–36.

[15] Van Rossum G, Drake FL. The Python language reference manual. Network Theory Ltd.; 2011.

[16] Smith C, Meurer A, Paprocki M, SymPy 1.0, Online: 2016. https://doi.org/10.5281/zenodo.47274 [Accessed 15 October 2016].

[17] Butcher J. Numerical methods for ordinary differential equations, second ed.; 2008, pp. 1–463, http://dx.doi.org/10.1002/9780470753767.

[18] Biral F, Bertolazzi E, Bosetti P. Notes on numerical methods for solving optimal control problems. IEEE Indus Appl 2016;S(2):154–66.