LuaCAS: Symbolic computation in LaTeX

Timothy All, Evan Cochrane

Abstract

LuaCAS is a portable computer algebra system, written entirely in Lua, designed for use within LuaLaTeX via the `luacas` package [1]. Features include: arbitrary precision integer and rational arithmetic, number-theoretic algorithms, constructors and algorithms for univariate polynomials defined over various rings, symbolic differentiation and integration, and more.

1 Motivation

Most existing computer algebra systems such as Maple and Mathematica allow for converting their stored expressions to LaTeX code. But this still requires exporting code from LaTeX to another program, converting it to a form that the CAS is expecting, performing the computation, and importing the result, which can be tedious.

In contrast, the `luacas` package allows the user to perform basic symbolic computations from *within* LuaLaTeX without the need for laborious and technical setup. One simply installs the package like any other and adds `\usepackage{luacas}` to the preamble. Indeed, this article, along with all computations contained therein, was prepared in Overleaf.

2 An example

The main method for interacting with LuaCAS from within LuaLaTeX is to use the `CAS` environment. The following example demonstrates typical usage:

```
\begin{CAS}
  vars('x')
  f = int(sin(sqrt(x)),x)
\end{CAS}
\[ \print{f} = \print*{f} \]
```

$$\int \sin\left(\sqrt{x}\right) dx = -2\sqrt{x}\cos\left(\sqrt{x}\right) + 2\sin\left(\sqrt{x}\right)$$

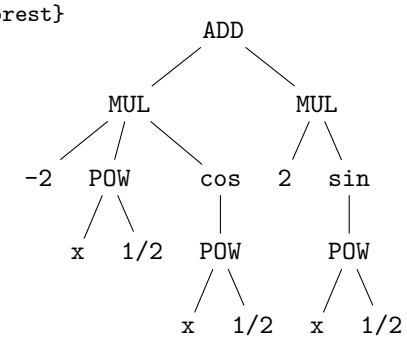
The macro `\print` converts the contents of its argument as-is into a string formatted for LaTeX and prints the result into the document via `tex.print`; the starred variant `\print*` evaluates and performs some basic simplifications on its argument before the conversion to LaTeX step.

3 Development

LuaCAS began as a senior capstone at Rose-Hulman Institute of Technology in the fall of 2021. The primary use case we had in mind early in development was that of a professor creating content with dynamic examples and problems for introductory calculus classes. We thus decided to make symbolic differentiation/integration the end goal of the project, as well as including basic algebraic functionality expected in any CAS.

Our first task was programming the core trees that would be used to store expressions. Mathematically speaking, an expression is a rooted tree. The `luacas` package comes with a pair of macros `\parseforest` (plus a starred variant that evaluates and simplifies the argument) and `@\forestresult` that allow the user to draw these rooted trees with the help of the `forest` package [9]. For example, with `f` defined as in the previous example, we have:

```
\parseforest*{f}
\bracketset{action character=@}
\begin{forest}
    for tree = {font=\ttfamily}
    @\forestresult
\end{forest}
```



Object-oriented Lua was chosen as our programming paradigm, allowing for the functionality of the CAS to be easily extensible without the need for compiling or setting up a complicated build environment.

Given that LuaCAS is written in an interpreted language, one can expect dramatically slower performance when comparing LuaCAS to popular computer algebra systems. Any mathematical operation or structure also gets its own class under this scheme, so there is an inevitable explosion in the number of classes. Despite this, LuaCAS performs well within the scope of its design and motivation.

We decided to split LuaCAS into modules, partly to increase the potential for extensibility and partly to reduce the time to load the CAS when only a subset of its features are needed. At the time of this writing, there are three modules: `core`, `algebra`, and `calculus`. The `luacas` package loads these modules by default.

The `core` module contains all of the interfaces for expressions that other classes need to extend (e.g., expression manipulation, substitution, etc.). The `algebra` module contains polynomial algorithms and common algebraic and trigonometric functions, as well as interfaces for algebraic structures such as

doi.org/10.47397/tb/44-1/tb136all-luacas

rings and fields. Class inheritance was chosen to mirror mathematical structures — fields inherit from Euclidean domains since all fields are Euclidean domains, and Euclidean domains likewise inherit from rings. Concrete classes represent specific rings, such as integers or polynomials, and objects are elements of these rings. Finally, the `calculus` module contains classes for differentiation and integration.

Features were implemented chronologically by an informal notion of how mathematically 'fundamental' they were. Accordingly, expression simplification via algebraic properties came first, then polynomial factoring/root-finding, then symbolic differentiation, and finally symbolic integration. This order turned out to be convenient for testing purposes, since symbolic integration relies on factoring for rational function integration. Algorithms for symbolic factoring and differentiation are well-established [10], but symbolic integration required significantly more tinkering to balance power and efficiency. This took the project beyond its original scope as a senior project; development on the symbolic integrator continued for months. Version 1.0.1 of `luacas` was uploaded to CTAN on November 15, 2022.

4 Features

The `CAS` environment is fundamentally a glorified `\directlua`. Accordingly, the `CAS` environment can be used essentially anywhere in a LaTeX document, and variables declared in one instance of the `CAS` environment will be remembered in the next instance of the `CAS` environment. Thus, expressions can be manipulated naturally throughout a document, as the examples below will illustrate.

4.1 Core

At the core of any computer algebra system is the notion of an *expression*. In LuaCAS, there are *atomic expressions* (e.g., integers, variables) and *composite expressions*. Variables must be declared or initialized before use (like Sage in days of yore). This is done with the `vars()` function within the `CAS` environment. Using a combination of Lua's meta-methods and operator overloading, composite expressions are constructed naturally:

```
\begin{CAS}
  vars('x')
  f,g = 1-x+0*x, 1+1*x
  h = f*g
\end{CAS}
\[ \print{h} \]
```

$$(1 - x + 0 \cdot x)(1 + 1x)$$

Essential functionality for any computer algebra system is the process of *simplification*. Externally, the user expects output from the CAS to be simple and concise. Internally, simplification serves as a sort of normalization procedure for expressions. Expressions can be simplified in a couple of different ways:

```
\begin{CAS}
  ah = h:autosimplify()
  sh = h:simplify()
\end{CAS}
\[ \print{ah} = \print{sh} \]
```

$$(1 + x)(1 - x) = 1 - x^2$$

The `autosimplify` method is designed to be fast, as it is automatically performed on expressions before most other functions, such as factoring and expansion. Accordingly, it perhaps does not go as far as one might expect. For a more rigorous simplification, the `simplify` method makes a rudimentary search for the smallest expression tree equivalent to the input.

The core functionality of LuaCAS allows for other types of expression manipulation including substitutions:

```
\begin{CAS}
  vars('h')
  sh = substitute({[x]=x+h},sh)
\end{CAS}
\[ \print{sh} \]
```

$$1 - (x + h)^2$$

4.2 Algebra

The Algebra module contains constructors for various special classes and related algorithms.

4.2.1 Rings

There are constructors for the following Ring types:

- the integers,
- the integers modulo N,
- rationals (interpreted somewhat broadly), and
- polynomials.

A rudimentary parser wrapped around the contents of the `CAS` environment calls most of these constructors in a natural way:

```
\begin{CAS}
  a,b,c = 65, Mod(65,4), 63/65
\end{CAS}
\[ \lprint{{a,b,c}} \]
```

$$65, 1, \frac{63}{65}$$

But to construct a polynomial requires specific input from the user:

```
\begin{CAS}
  vars('x')
  f,g = x^2+2*x+3, Poly({3,2,1},x)
\end{CAS}
\[ \print{f} \qquad \print{g} \]
```

$$x^2 + 2x + 3 \qquad x^2 + 2x + 3$$

The printouts of `f` and `g` look the same; but internally, LuaCAS handles these expressions quite differently. There are several advantages to having a dedicated `polynomial` class, not least of which is computational speed.

On the other hand, the user more often than not needn't worry about issues pertaining to class types. Many functions in LuaCAS are class-aware in that they will either detect or make some attempt at converting class types for you. For example, the `factor` function applied to $a = 1440$ will detect that the input is an `Integer`, then apply number theoretic algorithms to determine the prime-factorization (specifically a combination of Pollard-Rho and Miller-Rabin). On the other hand, when `factor` is given the expression $f = x^3 + x^2 + x - 3$, it first converts f to the `polynomial` class; from there it uses special algorithms to find the factorization over $\mathbf{Q}$ (specifically a combination of Berlekamp [2] and Zassenhaus [10]).

```
\begin{CAS}
  vars('x')
  a,f = 1440, x^3 + x^2 + x - 3
\end{CAS}
\[ \begin{aligned}
  \print{f} &= \print{factor(f)} \\
  \print{a} &= \print{factor(a)}
\end{aligned} \]
```

$$x^3 + x^2 + x - 3 = (-1 + x)\left(3 + 2x + x^2\right)$$
$$1440 = 3^2 2^5 5^1$$

4.2.2 Ring conversion

Each Ring type comes equipped with a *Ring identifier*. This identifier is used to cast arithmetic performed on differing Ring types to the appropriate Ring. For example, if we ask LuaCAS to add a `polynomial` with integer coefficients to a `rational` number, LuaCAS will fetch the Ring identifiers for both classes and determine that the appropriate Ring into which to cast the arithmetic is the `polynomial` ring with `rational` coefficients:

```
\begin{CAS}
  a,b,c = 2, 4/3, Poly({-3,1,1,1},x)
  d = c+b+a
\end{CAS}
\[ (\print{c}) + \print{b}
  + \print{a} = \print{d}\]
```

$$(x^3 + x^2 + x - 3) + \frac{4}{3} + 2 = x^3 + x^2 + x + \frac{1}{3}$$

4.2.3 Special classes

The Algebra module provides constructors for special classes such as those for trigonometric, radical, and logarithmic expressions, along with support for the expected simplifications of these expressions:

```
\begin{CAS}
  vars('x')
  a,b,c=sin(4*pi/3),ln(e^(x+1)),sqrt(8/9)
\end{CAS}
\[ \begin{aligned}
  \print{a} &= \print*{a} \\
  \print{b} &= \print*{b} \\
  \print{c} &= \print*{c}
\end{aligned} \]
```

$$\sin\left(\frac{4\pi}{3}\right) = -\frac{\sqrt{3}}{2}$$
$$\ln\left(e^{x+1}\right) = 1 + x$$
$$\sqrt{\frac{8}{9}} = \frac{2\sqrt{2}}{3}$$

4.2.4 Number theoretic algorithms

LuaCAS also provides basic number theoretic functionality. For example, LuaCAS can run the extended Euclidean algorithm:

```
\begin{CAS}
  a,b = 42250, 46137
  c,x,y = gcdext(a,b)
\end{CAS}
\[ \print{c} = \print{a} (\print{x})
    +\print{b}(\print{y}) \]
```

$$169 = 42250(-95) + 46137(87)$$

LuaCAS also contains factoring algorithms and primality checking for the `Integer`-class. For primality checking, we use Miller-Rabin [5] and the base set of prime witnesses $p = 2, 3, \ldots, 41$. Accordingly, primality checking can be trusted for integers a bit beyond 10^{24}. For factoring, we use Miller-Rabin combined with Pollard-Rho [4] to search for prime factors recursively:

```
\begin{CAS}
  a = 407808999
  b = factor(a)
\end{CAS}
\[ \print{a} = \print{b} \]
```

$$407808999 = 3^4 31^3 13^2$$

4.2.5 Polynomial algorithms

LuaCAS hosts a number of algorithms for (univariate) polynomial arithmetic over the rationals or modulo a prime, including: extended Euclidean algorithm, factoring, and resultants.

```
\begin{CAS}
  vars('x')
  f = Mod(topoly(x^2+x+1),7)
  ff = factor(f)
\end{CAS}
```

```
\[ \print{f} = \print{ff} \]
```

$$x^2 + x + 1 = 1\,(x+5)^1\,(x+3)^1$$

LuaCAS also contains algorithms for symbolic root finding over the rationals (including supporting algorithms like those for finding decomposition series).

```
\begin{CAS}
  vars('x')
  f = topoly(x^4 + 4*x^3 - 8*x + 3)
  r = roots(f)
\end{CAS}
\[ \left\{ \lprint{r} \right\} \]
```

$$\left\{1, -3, -1+\sqrt{2}, -1-\sqrt{2}\right\}$$

4.3 Calculus

The Calculus module contains constructors for derivatives/integrals and algorithms for symbolic differentiation/integration.

4.3.1 Differentiation

Due to the nature of differentiation, LuaCAS can quickly compute the derivatives of almost any expression that can be represented in LuaCAS.

```
\begin{CAS}
 vars('x', 'y', 'z')
 f,g = 3*ln(y)*sin(x), x^(1/(x*z))*x
 dg, df = diff(g,x), diff(f,{x,3},{y,2})
\end{CAS}
\[ \print{dg} = \print*{dg} \]
\[ \print{df} = \print*{df} \]
```

$$\frac{d}{dx}\left(x^{\frac{1}{xz}}x\right) = x^{1+\frac{1}{xz}}\left(\frac{\left(1+\frac{\frac{1}{x}}{z}\right)}{x} - \frac{\ln(x)}{x^2 z}\right)$$

$$\frac{\partial^5}{\partial y^2 \partial x^3}\left(3\ln(y)\sin(x)\right) = \frac{3\cos(x)}{y^2}$$

4.3.2 Integration

LuaCAS can evaluate a wide variety of definite and indefinite integrals. The integrator mostly works by calling standard methods familiar to any college calculus student recursively (such as u-substitution, integration-by-parts, etc.) and then searching for the appropriate anti-derivative. For integration of rational functions, we use the method of Lazard, Rioboo, Rothstein and Trager [8].

```
\begin{CAS}
  f = e^(2*x)*cos(3*x)
  F = int(f, x,0,pi)
\end{CAS}
\[ \print{F} = \print*{F} \]
```

$$\int_0^{\pi} e^{2x}\cos(3x)\,dx = -\frac{2}{13} - \frac{2e^{2\pi}}{13}$$

However, we cannot guarantee that an integral will be able to be evaluated, even if the expression is integrable in elementary terms.

5 Interaction with the LaTeX ecosystem

Given that the `CAS` environment is just a glorified `lua` environment, LuaCAS interacts very well with TeX primitives and standard macros as well as the Lua language. Indeed, the design of LuaCAS was (in part) inspired by the potential to write reusable code such as:

```
\newcommand{\Euclid}[3]{%
\begin{CAS}
  vars('x')
  a,b,p = #1,#2,#3
  a = Mod(topoly(a),p)
  b = Mod(topoly(b),p)
  tex.print("\\[ \\begin{aligned}")
  while b.degree>0 do
    q,r = a:divremainder(b)
    tex.print(a:tolatex(),
      "&= (",
      b:tolatex(),
      ")(",
      q:tolatex(),
      ")+",
      r:tolatex(),
      "\\\\")
      a,b = b,r
    end
  tex.print("\\end{aligned} \\]")
\end{CAS}%
}
\Euclid{x^4+x^3+x^2+x+1}{x^3+2*x+3}{7}
```

$$x^4 - x^2 + 1 = (x^3 + 2x + 1)(x) + 4x^2 - x + 1$$
$$x^3 + 2x + 1 = (4x^2 - x + 1)(2x + 4) + 4x + 4$$
$$4x^2 - x + 1 = (4x + 4)(x + 4) + 6$$

The macro `\Euclid` displays the Euclidean algorithm applied to the polynomials found in the first and second arguments modulo the prime found in the third argument.

The `luacas` package comes with the macros `\fetch` and `\store`. These macros allow the user to pull content out of LuaCAS in a format that's appropriate for packages like Ti*k*Z/PGF [7] and Asymptote [3]. For example:

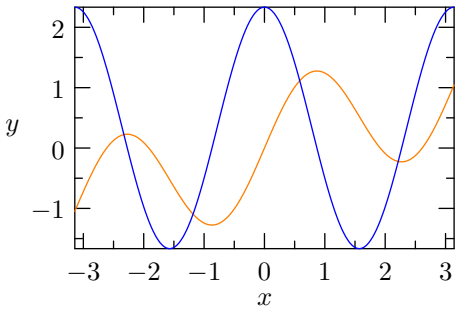
```
\begin{CAS}
  vars('x')
  f = sin(2*x)+x/3
  df = diff(f,x):autosimplify()
\end{CAS}
\store{f}\store{df}
```

The macro `\df` contains the string:

```
1/3 + (2 * (cos(2 * x)))
```

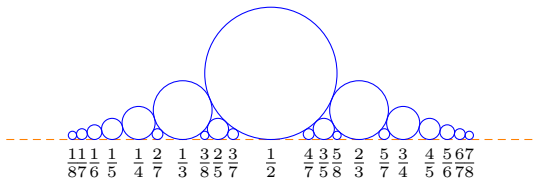
Macros created via the `\store` command can be called into other environments like the `asypicture` environment from the `asypictureB` package [6]:

```
\begin{asypicture}{}
  import graph; size(6cm,0);
  real f(real x){return @f;}
  real df(real x){return @df;}
  draw(graph(f,-pi,pi,operator..),orange);
  draw(graph(df,-pi,pi,operator..),blue);
  xaxis("$x$",BottomTop,LeftTicks);
  yaxis("$y$",LeftRight,RightTicks);
\end{asypicture}
```



The macro `\fetch` does nearly the same thing as `\store` except no macro is created; in other words, `\fetch{df}` can be used wherever we would have used `\df`. This is particularly useful for grabbing values out of tables built with LuaCAS:

```
\begin{tikzpicture}[scale=7]
  \draw[orange,densely dashed] (0,0) -- (1,0);
  \foreach \k in {2,...,22}{
    \draw[blue]
      (\fetch{F[\k]},\fetch{H[\k]})
      circle (\fetch{H[\k]});
    \node[below] at (\fetch{F[\k]},0)
      {\small$\print{F[\k]}$};
  }
\end{tikzpicture}
```



6 Future

In the future, we aim to expand the feature set of LuaCAS and include at least a decent chunk of the functionality common to popular existing computer algebra systems. This may include:

- summation and product expressions
- symbolic limits
- symbolic differential equation solving
- irreducible factorization of multivariate polynomials
- logic & set theory
- symbolic linear algebra
- numeric functionality

On the LaTeX side of things, it would be good to include some amount of externalization so that LuaCAS performs computations only when needed and not at every compile.

7 Acknowledgements

We'd like to thank the Mathematics Department at Rose-Hulman Institute of Technology for their support throughout this project. A special thanks goes to Joseph Eichholz for many helpful discussions and constructive feedback.

References

[1] T. All, E. Cochrane. *The Luacas package.* `ctan.org/pkg/luacas`

[2] E.R. Berlekamp. Factoring polynomials over finite fields. *Bell System Technical Journal* 46(8):1853–1859, 1967.

[3] J. Bowman, A. Hammerlindl. *The Asymptote package.* `asymptote.sourceforge.net`

[4] J.M. Pollard. A Monte Carlo method for factorization. *Nordisk Tidskr. Informationsbehandling (BIT)* 15(3):331–334, 1975.

[5] M.O. Rabin. Probabilistic algorithm for testing primality. *J. Number Theory* 12(1):128–138, 1980.

[6] C. Staats III. *The AsypictureB package.* `ctan.org/pkg/asypictureb`

[7] T. Tantau, C. Feuersänger, et al. *The PGF package.* `ctan.org/pkg/pgf`

[8] B.M. Trager. Algebraic factoring and rational function integration. In *Proceedings of the third ACM symposium on Symbolic and algebraic computation*, pp. 219–226, 1976.

[9] S. Živanović. *The Forest package.* `ctan.org/pkg/forest`

[10] H. Zassenhaus. On Hensel factorization, I. *J. Number Theory* 1(3):291–311, 1969.

◇ Timothy All
Department of Mathematics
Rose-Hulman Institute of Technology
Terre Haute, IN 47803 USA
`timothy.all (at) rose-hulman dot edu`

◇ Evan Cochrane
`cochraef (at) rose-hulman dot edu`