The Sage Project:
Unifying Free Mathematical Software to Create a Viable Alternative to Magma, Maple, Mathematica and MATLAB

Burçin Eröcal[1] and William Stein[2]

[1] Research Institute for Symbolic Computation
Johannes Kepler University,
Linz, Austria
Supported by FWF grants P20347 and DK W1214.
burcin@erocal.org
[2] Department of Mathematics
University of Washington
wstein@uw.edu
Supported by NSF grant DMS-0757627 and DMS-0555776.

Abstract. Sage is a free, open source, self-contained distribution of mathematical software, including a large library that provides a unified interface to the components of this distribution. This library also builds on the components of Sage to implement novel algorithms covering a broad range of mathematical functionality from algebraic combinatorics to number theory and arithmetic geometry.

Keywords: Python, Cython, Sage, Open Source, Interfaces

1 Introduction

In order to use mathematical software for exploration, we often push the boundaries of available computing resources and continuously try to improve our implementations and algorithms. Most mathematical algorithms require basic building blocks, such as multiprecision numbers, fast polynomial arithmetic, exact or numeric linear algebra, or more advanced algorithms such as Gröbner basis computation or integer factorization. Though implementing some of these basic foundations from scratch can be a good exercise, the resulting code may be slow and buggy. Instead, one can build on existing optimized implementations of these basic components, either by using a general computer algebra system, such as Magma, Maple, Mathematica or MATLAB, or by making use of the many high quality open source libraries that provide the desired functionality. These two approaches both have significant drawbacks. This paper is about Sage,[3] which provides an alternative approach to this problem.

[3] http://www.sagemath.org

Having to rely on a closed propriety system can be frustrating, since it is difficult to gain access to the source code of the software, either to correct a bug or include a simple optimization in an algorithm. Sometimes this is by design:

> "Indeed, in almost all practical uses of Mathematica, issues about how Mathematica works inside turn out to be largely irrelevant. You might think that knowing how Mathematica works inside would be necessary [...]" (See [Wol].)

Even if we manage to contact the developers, and they find time to make the changes we request, it might still take months or years before these changes are made available in a new release.

Fundamental questions of correctness, reproducibility and scientific value arise when building a mathematical research program on top of proprietary software (see, e.g., [SJ07]). There are many published refereed papers containing results that rely on computations performed in Magma, Maple, or Mathematica.[4] In some cases, a specific version of Magma is the only software that can carry out the computation. This is not the infrastructure on which we want to build the future of mathematical research.

In sharp contrast, open source libraries provide a great deal of flexibility, since anyone can see and modify the source code as they wish. However, functionality is often segmented into different specialized libraries and advanced algorithms are hidden behind custom interpreted languages. One often runs into trouble trying to install dependencies before being able use an open source software package. Also, converting the output of one package to the input format of another package can present numerous difficulties and introduce subtle errors.

Sage, which started in 2005 (see [SJ05]), attacks this problem by providing:

1. a *self-contained distribution* of mathematical software that installs from source easily, with the only dependency being compiler tools,
2. *unified interfaces* to other mathematical software to make it easier to use all these programs together, and
3. a *new library* that builds on the included software packages and implements a broad range of mathematical functionality.

The rest of this paper goes into more detail about Sage. In Section 1.1, we describe the Sage graphical user interface. Section 1.2 is about the Sage development process, Sage days workshops, mailing lists, and documentation. The subject of Section 2 is the sophisticated way in which Sage is built out of a wide range of open source libraries and software. In Section 2.1 we explain how we use Python and Cython as the glue that binds the compendium of software included in Sage into a unified whole. We then delve deeper into Python, Cython and the Sage preparser in Section 2.2, and illustrate some applications to mathematics in Section 2.3. Sage is actively used for research, and in Section 3 we describe some capabilities of Sage in advanced areas of mathematics.

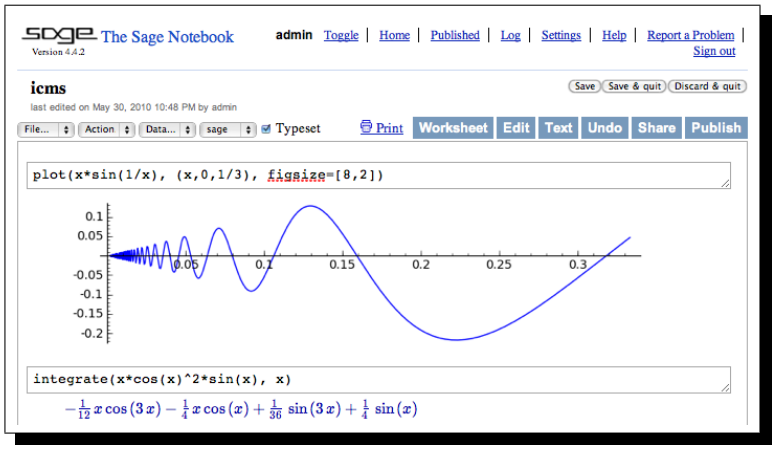


Fig. 1. The Sage Notebook

1.1 The Notebook

As illustrated in Figure 1, the graphical user interface for Sage is a web application, inspired by Google Documents [Goo], which provides convenient access to all capabilities of Sage, including 3D graphics. In single user mode, Sage works like a regular application whose main window happens to be your web browser. In multiuser mode, this architecture allows users to easily set up servers for accessing their work over the Internet as well as sharing and collaborating with colleagues. One can try the Sage notebook by visiting `www.sagenb.org`, where there are over 30,000 user accounts and over 2,000 published worksheets.

Users also download Sage to run it directly on their computers. We track all downloads from `www.sagemath.org`, though there are several other high-profile sites that provide mirrors of our binaries. Recently, people download about 6,000 copies of Sage per month directly from the Sage website.

1.2 The Sage Development Process

There are over 200 developers from across the world who have contributed to the Sage project. People often contribute because they write code using Sage as part of a research project, and in this process find and fix bugs, speed up parts of Sage, or want the code portion of their research to be peer reviewed. Each contribution to Sage is first posted to the Sage Trac server `trac.sagemath.org`; it is then peer reviewed, and finally added to Sage after all issues have been sorted out and all requirements are met. Nothing about this process is anonymous; every step of the peer review process is recorded indefinitely for all to see.

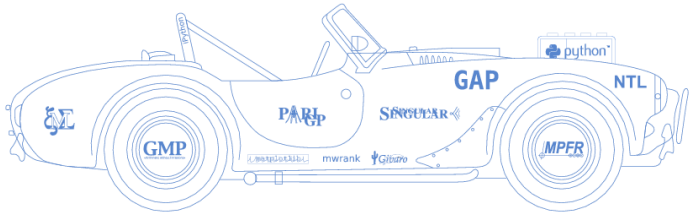
[4] Including by the second author of this paper, e.g., [CES03]!

The Sage Developer's Guide begins with an easy-to-follow tutorial that guides developers through each step involved in contributing code to Sage. Swift feedback is available through the `sage-devel` mailing list, and the `#sage-devel` IRC chat room on `irc.freenode.net` (see `www.sagemath.org/development.html`).

Much development of Sage has taken place at the Sage Days workshops. There have been two dozen Sage Days [Sagb] and many more are planned. These are essential to sustaining the momentum of the Sage project and also help ensure that developers work together toward a common goal, rather than competing with each other and fragmenting our limited community resources.

A major goal is ensuring that there will be many Sage Days workshops for the next couple of years. The topics will depend on funding, but will likely include numerical computation, large-scale bug fixing, L-functions and modular forms, function fields, symbolic computation, topology, and combinatorics. The combination of experienced developers with a group of enthusiastic mathematicians at each of these workshops has rapidly increased the developer community, and we hope that it will continue to do so.

2 Building the Car...



With the motto "building the car instead of reinventing the wheel," Sage brings together numerous open source software packages (see Table 1 and [Saga]).

Many applications of Sage require using these libraries together. Sage handles the conversion of data behind the scenes, automatically using the best tool for the job, and allows the user to concentrate on the problem at hand.

In the following example, which we explain in detail below, Sage uses the FLINT library [HH] for univariate polynomials over the ring $\mathbb{Z}$ of integers, whereas Singular [DGPS10] is used for multivariate polynomials. The option to use the NTL library [Sho] for univariate polynomials is still available, if the user so chooses.

```
sage: R.<x> = ZZ[]
sage: type(R.an_element())
<type 'sage.rings...Polynomial_integer_dense_flint'>
sage: R.<x,y> = ZZ[]
sage: type(R.an_element())
<type 'sage.rings...MPolynomial_libsingular'>
sage: R = PolynomialRing(ZZ, 'x', implementation='NTL')
sage: type(R.an_element())
<type 'sage.rings...Polynomial_integer_dense_ntl'>
```

Table 1. Packages Included With Every Copy of Sage-4.4.2

atlas	gap	libgcrypt	palp	scipy_sandbox
blas	gd	libgpg_error	pari	scons
boehm_gc	gdmodule	libm4ri	pexpect	setuptools
boost	genus2reduction	libpng	pil	singular
cddlib	gfan	linbox	polybori	sphinx
cliquer	ghmm	matplotlib	pycrypto	sqlalchemy
cvxopt	givaro	maxima	pygments	sqlite
cython	gnutls	mercurial	pynac	symmetrica
docutils	gsl	moin	python	sympow
ecl	iconv	mpfi	python_gnutls	sympy
eclib	iml	mpfr	r	tachyon
ecm	ipython	mpir	ratpoints	termcap
f2c	jinja	mpmath	readline	twisted
flint	jinja2	networkx	rubiks	weave
flintqs	lapack	ntl	sagenb	zlib
fortran	lcalc	numpy	sagetex	zn_poly
freetype	libfplll	opencdk	scipy	zodb3

The first line in the example above constructs the univariate polynomial ring `R` $= \mathbb{Z}[x]$, and assigns the variable `x` to be the generator of this ring. Note that $\mathbb{Z}$ is represented by `ZZ` in Sage. The expression `R.<x> = ZZ[]` is not valid Python, but can be used in Sage code as a shorthand as explained in Section 2.2. The next line asks the ring `R` for an element, using the `an_element` function, then uses the builtin Python function `type` to query its type. We learn that it is an instance of the class `Polynomial_integer_dense_flint`. Similarly line 4 constructs `R` $= \mathbb{Z}[x, y]$ and line 7 defines `R` $= \mathbb{Z}[x]$, but this time using the `PolynomialRing` constructor explicitly and specifying that we want the underlying implementation to use the NTL library.

Often these interfaces are used under the hood, without the user having to know anything about the corresponding systems. Nonetheless, there are easy ways to find out what is used by inspecting the source code, and users are strongly encouraged to cite components they use in published papers. The following example illustrates another way to get a list of components used when a specific command is run.

```
sage: from sage.misc.citation import get_systems
sage: get_systems('integrate(x^2, x)')
['ginac', 'Maxima']
sage: R.<x,y,z> = QQ[]
sage: I = R.ideal(x^2+y^2, z^2+y)
sage: get_systems('I.primary_decomposition()')
['Singular']
```

2.1 Interfaces

Sage makes it possible to use a wide range of mathematical software packages together by providing a unified interface that handles data conversion automatically. The complexity and functionality of these interfaces varies greatly, from simple text-based interfaces that call external software for an individual computation, to using a library as the basis for an arithmetic type. The interfaces can also run code from libraries written in the interpreted language of another program. Table 2 lists the interfaces provided by Sage.

Table 2. Sage Interfaces to the above Mathematical Software

Pexpect	axiom, ecm, fricas, frobby, gap, g2red, gfan, gnuplot, gp, kash, lie, lisp, macaulay2, magma, maple, mathematica, matlab, maxima, mupad, mwrank, octave, phc, polymake, povray, qepcad, qsieve, r, rubik, scilab, singular, tachyon
C Library	eclib, fplll, gap (in progress), iml, linbox, maxima, ratpoints, r (via rpy2), singular, symmetrica
C Library arithmetic	flint, mpir, ntl, pari, polybori, pynac, singular

The above interfaces are the result of many years writing Python and Cython [BBS] code to adapt Singular [DGPS10], GAP [L$^+$], Maxima [D$^+$], Pari [PAR], GiNaC/Pynac [B$^+$], NTL [Sho], FLINT [HH], and many other libraries, so that they can be used smoothly and efficiently in a unified way from Python [Ros]. Some of these programs were originally designed to be used only through their own interpreter and made into a library by Sage developers. For example libSingular was created by Martin Albrecht in order to use the fast multivariate polynomial arithmetic in Singular from Sage. The libSingular interface is now used by other projects, including Macaulay2 [GS] and GFan [Jen].

There are other approaches to linking mathematical software together. The recent paper [LHK$^+$] reports on the state of the art using OpenMath. Sage takes a dramatically different approach to this problem. Instead of using a general string-based XML protocol to communicate with other mathematical software, Sage interfaces are tailor made to the specific software and problem at hand. This results in far more efficient and flexible interfaces. The main disadvantage compared to OpenMath is that the interfaces all go through Sage.

Having access to many programs which can perform the same computation, without having to worry about data conversion, also makes it easier to double check results. For example, below we first use Maxima, an open source symbolic computation package distributed with Sage, to integrate a function, then perform the same computation using Maple and Mathematica.

```
sage: var('x')
sage: integrate(sin(x^2), x)
1/8*((I - 1)*sqrt(2)*erf((1/2*I - 1/2)*sqrt(2)*x) + \
```

```
  (I + 1)*sqrt(2)*erf((1/2*I + 1/2)*sqrt(2)*x))*sqrt(pi)
sage: maple(sin(x^2)).integrate(x)
1/2*2^(1/2)*Pi^(1/2)*FresnelS(2^(1/2)/Pi^(1/2)*x)
sage: mathematica(sin(x^2)).Integrate(x)
Sqrt[Pi/2]*FresnelS[Sqrt[2/Pi]*x]
```

The most common type of interface, called a `pexpect` interface, communicates with another command line program by reading and writing strings to a text console, as if another user was in front of the terminal. Even though these are relatively simple to develop, the overhead of having to print and parse strings to represent the data makes this process potentially cumbersome and inefficient. This is the default method of communication with most high level mathematics software, including commercial and open source programs, such as Maple, Mathematica, Magma, KASH or GAP.

Sage provides a framework to represent elements over these interfaces, perform arithmetic with them or apply functions to the given object, as well as using a file to pass the data if the string representation is too big. The following demonstrates arithmetic with GAP elements.

```
sage: a = gap('22')
sage: a*a
484
```

It is also possible to use `pexpect` interfaces over remote consoles. In the following code, we connect to the `localhost` as a different user and call Mathematica functions. Note that the interface can handle indexing vectors as well.

```
sage: mma = Mathematica(server="rmma60@localhost")
sage: mma("2+2")
4
sage: t = mma("Cos[x]")
sage: t.Integrate('x')
Sin[x]
sage: t = mma('{0,1,2,3}')
sage: t[2]
1
```

Sage also includes specialized libraries that are linked directly from compiled code written in Cython. These are used to handle specific problems, such as the characteristic polynomial computation in the example below.

```
sage: M = Matrix(GF(5), 10, 10)
sage: M.randomize()
sage: M.charpoly(algorithm='linbox')
x^10 + 4*x^9 + 4*x^7 + 3*x^4 + 3*x^3 + 3*x^2 + 4*x + 3
```

Many basic arithmetic types also use Cython to directly utilize data structures from efficient arithmetic libraries, such as MPIR or FLINT. An example of this can be seen at the beginning of this section, where elements of the ring $\mathbb{Z}[x]$ are represented by the class `Polynomial_integer_dense_flint`.

The Singular interface is one of the most advanced included in Sage. Singular has a large library of code written in its own language. Previously the only way to access these functions, which include algorithms for Gröbner basis and primary

decomposition, was to call Singular through a `pexpect` interface, passing data back and forth using strings. Recently, due to work of Michael Brickenstein and Martin Albrecht, Sage acquired the ability to call these functions directly.

In the example below, we import the function `primdecSY` from `primdec.lib`, and call it the same way we would call a Python function. The interface handles the conversion of the data to Singular's format and back. Since Sage already uses Singular data structures directly to represent multivariate polynomials and ideals over multivariate polynomial rings, there are no conversion costs. It is only a matter of passing the right pointer.

```
sage: pr = sage.libs.singular.ff.primdec__lib.primdecSY
sage: R.<x,y,z> = QQ[]
sage: p = z^2+1; q = z^3+2
sage: I = R.ideal([p*q^2,y-z^2])
sage: pr(I)
[[[z^2 - y, y^3 + 4*y*z + 4], \
    [z^2 - y, y*z + 2, y^2 + 2*z]], \
    [[y + 1, z^2 + 1], [y + 1, z^2 + 1]]]
```

Efforts are under way to extend these capabilities to other programs, for example to GAP which provides Sage's underlying group theory functionality. Up to now, GAP was only available through its interpreter, through a `pexpect` interface that was written by Steve Linton. As the following example demonstrates, the performance of this interface is far from ideal.[5]

```
sage: b = gap('10')
sage: b*b
100
sage: timeit('b*b')
625 loops, best of 3: 289 microseconds per loop
```

The code snippet above constructs the element `b` in GAP using the `pexpect` interface, and measures the time it takes to square `b`. Compare these numbers to the following example, which uses the library interface to GAP, recently developed by the second author (but *not* included in Sage yet).

```
sage: import sage.libs.gap.gap as g
sage: a = g.libgap('10'); a
10
sage: type(a)
<type 'sage.libs.gap.gap.GapElement'>
sage: a*a
100
sage: timeit('a*a')
625 loops, best of 3: 229 nanoseconds per loop
```

The library interface is about 1,000 times faster than the pexpect interface.

[5] All timings in this paper were performed on an 2.66GHz Intel Xeon X7460 based computer.

2.2 Python - a mainstream language

In line with the principle of not reinventing the wheel, Sage is built on the mainstream programming language Python, both as the main development language and the user language. This frees the Sage developers, who are mainly mathematicians, from the troubles of language design, and gives access to an immense array of general purpose Python libraries and tools.

Python is an interpreted language with a clear, easy to read and learn syntax. Since it is dynamically typed, it is ideal for rapid prototyping, providing an environment to easily test new ideas and algorithms.

A fast interpreter In the following Singular session, we first declare the ring `r` $= \mathbb{Q}[x, y, z]$ and the polynomial `f` $\in$ `r`, then measures the time to square `f` repeatedly, 10,000 times.

```
singular: int t = timer; ring r = 0,(x,y,z), dp;
singular: def f = y^2*z^2-x^2*y^3-x*z^3+x^3*y*z;
singular: int j; def g = f;
singular: for (j = 1; j <= 10^5; j++) { g = f*f; }
singular: (timer-t), system("--ticks-per-sec");
990 1000
```

The elapsed time is 990 milliseconds. Next we use Sage to do the same computation, using the same Singular data structures directly, but without going through the interpreter.

```
sage: R.<x,y,z> = QQ[]
sage: f = y^2*z^2 - x^2*y^3 - x*z^3 + x^3*y*z;  type(f)
<type 'sage.rings.polynomial...MPolynomial_libsingular'>
sage: timeit('for j in xrange(10^5): g = f*f')
5 loops, best of 3: 91.8 ms per loop
```

Sage takes only 91.8 milliseconds for the same operation. This difference is because the Python interpreter is more efficient at performing `for` loops.

Cython - compiled extensions Python alone is too slow to implement a serious mathematical software system. Fortunately, Cython [BBS] makes it easy to optimize parts of your program or access existing C/C++ libraries. It can translate Python code with annotations containing static type information to C/C++ code, which is then compiled as a Python extension module.

Many of the basic arithmetic types in Sage are provided by Cython wrappers of C libraries, such as FLINT for univariate polynomials over $\mathbb{Z}$, Singular for multivariate polynomials, and Pynac for symbolic expressions.

The code segment below defines a Python function to add integers from 0 to N and times the execution of this function with the argument `10^7`.

```
sage: def mysum(N):
....:     s = int(0)
....:     for k in xrange(1,N): s += k
....:     return s
....:
```

```
sage: time mysum(10^7)
CPU times: user 0.52 s, sys: 0.00 s, total: 0.52 s
49999995000000
```

Here is the same function, but the loop index `k` is declared to be a C integer and the accumulator `s` is a C `long long`.

```
sage: cython("""
....: def mysum_cython(N):
....:     cdef int k
....:     cdef long long s = 0
....:     for k in xrange(N): s += k
....:     return s
....: """)
sage: time mysum_cython(10^7)
CPU times: user 0.01 s, sys: 0.00 s, total: 0.01 s
49999995000000L
```

The code is compiled and linked to the interpreter on the fly, and the function `mysum_cython` is available immediately. Note that the run time for the Cython function is 60 times faster than the Python equivalent.

Cython also handles the conversion of Python types to C types automatically. In the following example, we call the C function `sinl` using Cython to wrap it in a Python function named `sin_c_wrap`.

```
sage: cython("""
....: cdef extern from "math.h":
....:         long double sinl(long double)
....: def sin_c_wrap(a):
....:         return sinl(a)
....: """)
sage: sin_c_wrap(3.14)
0.0015926529164868282
sage: sin_c_wrap(1)
0.8414709848078965
sage: sin_c_wrap(1r)
0.8414709848078965
```

Note that the conversion of Sage types in the first two calls to `sin_c_wrap` or the Python type integer in the last call is performed transparently by Cython.

The Preparser While Python has many advantages as a programming and glue language, it also has some undesirable features. Sage hides these problems by using a preparser to change the commands passed to Python in an interactive session (or when running a script with the `.sage` extension). In order to maintain compatibility with Python, changes performed by the preparser are kept to a minimum. Moreover, the Sage library code is not preparsed, and is written in Cython or Python directly.

Python, like C and many other programming languages, performs integer floor division. This means typing `1/2` results in 0, not the rational number 1/2. Sage wraps all numeric literals entered in the command line or the notebook

with its own type declarations, which behave as expected with respect to arithmetic and have the advantage that they are backed by efficient multiprecision arithmetic libraries such as MPIR [H$^+$] and MPFR [Z$^+$], which are thousands of times faster than Python for large integer arithmetic.

To call the preparser directly on a given string, use the `preparse` function.

```
sage: preparse("1/2")
'Integer(1)/Integer(2)'
sage: preparse("1.5")
"RealNumber('1.5')"
```

Adding a trailing `r` after a number indicates that the preparser should leave that as the "raw" literal. The following illustrates division with Python integers.

```
sage: preparse("1r/2r")
'1/2'
sage: 1r/2r
0
```

Here is the result of performing the same division in Sage.

```
sage: 1/2
1/2
sage: type(1/2)
<type 'sage.rings.rational.Rational'>
sage: (1/2).parent()
Rational Field
```

The preparser also changes the `^` sign to the exponentiation operator `**` and provides a shorthand to create new mathematical domains and name their generator in one command.

```
sage: preparse("2^3")
'Integer(2)**Integer(3)'
sage: preparse("R.<x,y> = ZZ[]")
"R = ZZ['x, y']; (x, y,) = R._first_ngens(2)"
```

2.3 Algebraic, Symbolic and Numerical Tools

Sage combines algebraic, symbolic and numerical computation tools under one roof, enabling users to choose the tool that best suits the problem. This combination also makes Sage more accessible to a wide audience—scientists, engineers, pure mathematicians and mathematics teachers can all use the same platform for scientific computation.

While not concentrating on only one of these domains might seem to divide development resources unnecessarily, it actually results in a better overall experience for everyone, since users do not have to come up with makeshift solutions to compensate for the lack of functionality from a different field. Moreover, because Sage is a distributed mostly-volunteer open source project, widening our focus results in substantially more developer resources.

Algebraic Tools: The Coercion System An algebraic framework, similar to that of Magma or Axiom, provides access to efficient data structures and specialized algorithms associated to particular mathematical domains. The Python language allows classes to define how arithmetic operations like `+` and `*` will be handled, in a similar way to how C++ allows overloading of operators. However, the built-in support for overloading in Python is too simple to support operations with a range of objects in a mathematical type hierarchy.

Sage abstracts the process of deciding what an arithmetic operation means, or equivalently, in which domain the operation should be performed, in a framework called the *coercion system*, which was developed and implemented by Robert Bradshaw, David Roe, and many others. Implementations of new mathematical objects only need to define which other domains have a natural embedding to their domain. When performing arithmetic with objects, the coercion system will find a common domain where both arguments can be canonically mapped, perform the necessary type conversions automatically, thus allowing the implementation to only handle the case where both objects have the same parent.

In the following example, the variable `t` is an element of $\mathbb{Z}$ whereas `u` is in $\mathbb{Q}$. In order to perform the addition, the coercion system first deduces that the result should be in $\mathbb{Q}$ from the fact that `t` can be converted to the domain of `u`, namely $\mathbb{Q}$, but canonical conversion in the other direction is not possible. Then the addition is performed with both operands having the same domain $\mathbb{Q}$.

```
sage: t = 1
sage: t.parent()
Integer Ring
sage: u = 1/2
sage: u.parent()
Rational Field
sage: v = t + u; v
3/2
sage: v.parent()
Rational Field
```

Similarly, in the following example, the common domain $\mathbb{Q}[x]$ is found for arguments from $\mathbb{Z}[x]$ and $\mathbb{Q}$. Note that in this case, the result is not in the domain of either of the operands.

```
sage: R.<x> = ZZ[]
sage: r = x + 1/2
sage: r.parent()
Univariate Polynomial Ring in x over Rational Field
sage: 5*r
5*x + 5/2
```

Algebraic Tools: The Category Framework Another abstraction to make implementing mathematical structures easier is the *category framework*, whose development was spearheaded by Nicolas Thiéry and Florent Hivert. Similar in spirit to the mathematical programming facilities developed in Axiom and encapsulated in Aldor, the category framework uses Python's dynamic class

creation capabilities to combine functions relevant for a mathematical object, inherited through a mathematical hierarchy, into a class at run time.

This process greatly simplifies the troubles of having to combine object-oriented programming concepts with mathematical structural concerns, while keeping efficiency in mind. Efficient implementations can keep the inheritance hierarchy imposed by the data structures, while generic methods to compute basic properties are implemented in the *category* and automatically attached to the element classes when they are needed.

Symbolic Tools The symbolic subsystem of Sage provides an environment similar to Maple or Mathematica, where the input is treated only as an expression without any concern about the underlying mathematical structure.

Sage uses Pynac [ES], a hybrid C++ and Cython library built on top of GiNaC [B$^+$], to work with symbolic expressions. High level symbolic calculus problems including symbolic integration, solution of differential equations and Laplace transforms are solved using Maxima behind the scenes.

Here is an example of how to use the symbolic computation facilities in Sage. Note that in contrast to other symbolic software such as Maple, variables must be declared before they are used.

```
sage: x,y,z = var('x,y,z')
sage: sin(x).diff(x)
cos(x)
sage: psi(x).series(x,4)
(-1)*x^(-1) + (-euler_gamma) + (1/6*pi^2)*x + \
    (-zeta(3))*x^2 + (1/90*pi^4)*x^3 + Order(x^4)
sage: w = SR.wild()  # wildcard for symbolic substitutions
sage: ((x^2+y^2+z^2)*zeta(x)).subs({w^2:5})
15*zeta(x)
```

Numerical Tools In addition to code for symbolic computation, the standard numerical Python packages NumPy, SciPy, and Matplotlib are included in Sage, along with the numerical libraries cvxopt, GSL, Mpmath, and R.

For numerical applications, Robert Bradshaw and Carl Witty developed a compiler for Sage that converts symbolic expressions into an internal format suitable for blazingly fast floating point evaluation.

```
sage: f(x,y) = sqrt(x^2 + y^2)
sage: a = float(2)
sage: timeit('float(f(a,a))')
625 loops, best of 3: 216 microseconds per loop
sage: g = fast_float(f)
sage: timeit('float(g(a,a))')
625 loops, best of 3: 0.406 microseconds per loop
```

The `fast_float` feature is automatically used by the `minimize` command.

```
sage: minimize(f, (a,a))
(-5.65756135618e-05, -5.65756135618e-05)
```

Performance is typically within a factor of two from what one gets using a direct implementation in C or Fortran.

3 Afterword

In this article, we have showed that Sage is a powerful platform for developing sophisticated mathematical software. Sage is actively used in research mathematics, and people use Sage to develop state-of-the-art algorithms. Sage is particularly strong in number theory, algebraic combinatorics, and graph theory. For further examples, see the 53 published articles, 11 Ph.D. theses, 10 books, and 30 preprints at `www.sagemath.org/library-publications.html`.

For example, Sage has extensive functionality for computations related to the Birch and Swinnerton-Dyer conjecture. In addition to Mordell-Weil group computations using [Cre] and point counting over large finite fields using the SEA package in [PAR], there is much novel elliptic curve code written directly for Sage. This includes the fastest known algorithm for computation of p-adic heights [Har07,MST06], and code for computing p-adic L-series of elliptic curves at ordinary, supersingular, and split multiplicative primes. Sage combines these capabilities to compute explicitly bounds on Shafarevich-Tate groups of elliptic curves [SW10]. Sage also has code for computation with modular forms, modular abelian varieties, and ideal class groups in quaternion algebras.

The MuPAD-combinat project, which was started by Florent Hivert and Nicolas M. Thiéry in 2000, built the world's preeminent system for algebraic combinatorics on top of MuPAD (see [Des06] and [HT05]). Page 54 of [HT05]: "They [MuPAD] also have promised to release the code source of the library under a well known open-source license, some day." In 2008, MuPAD was instead purchased by MathWorks (makers of MATLAB), so MuPAD is no longer available as a separate product, and will probably never be open source. Instead it now suddenly costs $3000 (commercial) or $700 (academic).

As a result, the MuPAD-combinat group has spent several years reimplementing everything in Sage (see [T$^+$] for the current status). The MuPAD-combinat group was not taken by surprise by the failure of MuPAD, but instead were concerned from the beginning by the inherent risk in building their research program on top of MuPAD. In fact, they decided to switch to Sage two months before the bad news hit, and have made tremendous progress porting:

> "It has been such a relief during the last two years not to have this Damocles sword on our head!"
>
> – Nicolas Thiéry

References

B$^+$. Christian Bauer et al., **Ginac: is not a CAS**, `http://www.ginac.de/`.

BBS. Stefan Behnel, Robert Bradshaw, and Dag Seljebotn, **Cython: C-Extensions for Python**, `http://www.cython.org/`.

CES03. B. Conrad, S. Edixhoven, and W. A. Stein, $J_1(p)$ **Has Connected Fibers**, Documenta Mathematica **8** (2003), 331–408.

Cre. J. E. Cremona, `mwrank` **(computer software)**, `http://www.warwick.ac.uk/staff/J.E.Cremona/mwrank/`.

D$^+$. Robert Dodier et al., **Maxima: A Computer Algebra System**, `http://maxima.sourceforge.net/`.

Des06. Francois Descouens, **Making research on symmetric functions with MuPAD-Combinat**, Mathematical software—ICMS 2006, Lecture Notes in Comput. Sci., vol. 4151, Springer, Berlin, 2006, pp. 407–418.

DGPS10. W. Decker, G.-M. Greuel, G. Pfister, and H. Schönemann, SINGULAR **3-1-1 — A computer algebra system for polynomial computations**, `http://www.singular.uni-kl.de`.

ES. Burcin Erocal and William Stein, **Pynac – symbolic computation with python objects**, `http://pynac.sagemath.org/`.

Goo. Google, **Google Documents**, `http://docs.google.com/`.

GS. Daniel R. Grayson and Michael E. Stillman, **Macaulay2, a software system for research in algebraic geometry**, Available at http://www.math.uiuc.edu/Macaulay2/.

H$^+$. Bill Hart et al., **MPIR: Multiprecision Integers and Rationals**, `http://www.mpir.org/`.

Har07. David Harvey, **Efficient computation of p-adic heights**, `http://arxiv.org/abs/0708.3404`.

HH. Bill Hart and David Harvey, **Flint: Fast library for number theory**, `http://www.flintlib.org/`.

HT05. Florent Hivert and Nicolas M. Thiéry, **MuPAD-Combinat, an open-source package for research in algebraic combinatorics**, Sém. Lothar. Combin. **51** (2004/05), Art. B51z, 70 pp. (electronic), `http://www.emis.de/journals/SLC/wpapers/s51thiery.html`.

Jen. Anders Jensen, **Gfan: software for computing Gröbner fans and tropical varieties**, `http://www.math.tu-berlin.de/~jensen/software/gfan/gfan.html`.

L$^+$. Steve Linton et al., **Gap: Groups, algorithms and programming**, `http://www.gap-system.org/`.

LHK$^+$. S. Linton, K. Hammond, A. Konovalov, et al., **Easy Composition of Symbolic Computation Software: A New Lingua Franca for Symbolic Computation**, `www.win.tue.nl/~droozemo/site/pubs/1004ISSAC2010.pdf`.

MST06. Barry Mazur, William Stein, and John Tate, **Computation of p-adic heights and log convergence**, Doc. Math. (2006), no. Extra Vol., 577–614 (electronic). MR MR2290599 (2007i:11089)

PAR. PARI, **A computer algebra system designed for fast computations in number theory**, `http://pari.math.u-bordeaux.fr/`.

Ros. Guido van Rossum, **Python**, `http://www.python.org`.

Saga. Sage, **Components**, `http://sagemath.org/links-components.html`.

Sagb. ———, **Sage days workshops**, `http://wiki.sagemath.org/Workshops`.

Sho. V. Shoup, **NTL: Number theory library**, `http://www.shoup.net/ntl/`.

SJ05. William Stein and David Joyner, **Open source mathematical software**, ACM SIGSAM Bulletin **39** (2005).

SJ07. ———, **Open source mathematical software**, Notices Amer. Math. Soc. (2007), `http://www.ams.org/notices/200710/tx071001279p.pdf`.

SW10. W. Stein and C. Wuthrich, **Computations About Tate-Shafarevich Groups Using Iwasawa Theory**, In preparation (2010), http://wstein.org/papers/shark/.

T+. N. Thiery et al., **Sage Combinat Roadmap**, http://trac.sagemath.org/sage_trac/wiki/SageCombinatRoadMap.

Wol. Wolfram, **Why you do not usually need to know about internals**, http://reference.wolfram.com/mathematica/tutorial/WhyYouDoNotUsuallyNeedToKnowAboutInternals.html.

Z+. Paul Zimmerman et al., **The MPFR Library**, http://www.mpfr.org/.