Computer algebra with Rings library

Stanislav Poslavsky
142281, NRC "Kurchatov Institute" - IHEP, Nauki square 1, Protvino, Russia
E-mail: stvlpos@mail.ru

Abstract. Implementation of modern algorithms in computer algebra requires the use of generic and high-performance instruments. Rings is an open-source library, written in Java and Scala programming languages, which implements basic concepts and algorithms from computational commutative algebra while demonstrating quite a high performance among existing software. It rigorously uses generic programming approach, providing a well-designed generic API with a fully typed hierarchy of algebraic structures and algorithms for commutative algebra. Polynomial arithmetic, GCDs, factorization, and Gröbner bases are implemented with the use of modern asymptotically fast algorithms. The use of the Scala language brings a quite novel powerful, strongly typed functional programming model allowing to write short, expressive, and fast code for applications in high-energy physics and other research areas.

1. Introduction

Computer algebra and in particular computational commutative algebra is perhaps the main computational instrument in modern theoretical high-energy physics. For example, rational-function arithmetic is a key component of nearly all symbolic computations in the field. A more advanced concepts, like Gröbner bases, arise frequently in such topics as computation of (multi)loop Feynman diagrams. Efficient implementation of related algorithms and data structures in computer is crucial for successful solution of modern demanding problems.

The key mathematical concept lying in the basis of many algorithms is the concept of *ring homomorphism*. This concept is broadly manifested in e.g. modular techniques and rational reconstruction, which are successfully applied for developing efficient algorithms to solve challenging problems in high-energy physics in the recent few years [1, 2, 3, 4]. To explain these methods, consider the Euclidean algorithm for computing GCD:

```
1 function gcd(a, b)
2     if b = 0
3         return a;
4     else
5         return gcd(b, a mod b);
```

Applying this algorithm to the following quite simple input:

$$\texttt{gcd}(1 - x^2 + x^{20} - x^{200}, 1 - x^3 + x^{30} - x^{300}) = x - 1$$

one will observe extreme growth of intermediate coefficients at each subsequent step. In fact, one will have to operate with numbers containing *thousands* of digits to find a very simple answer $(x-1)$. This problem is called *intermediate expression swell* and it is inherent to many computer

algebra algorithms, like solving linear systems over rational numbers or computing resultants. When the intermediate expression swell takes place, the computation becomes impractically slow due to exponential growth of coefficients.

The way to avoid expression swell is to perform calculations modulo some prime number instead of operating with unbounded integers (or rationals). Once obtained the answer modulo N several prime numbers p_i, it is possible to reconstruct the answer modulo their product $p_1 \times \cdots \times p_N$, using the famous *Chinese Remainder Theorem* known from the ancient times. Taking the sufficient number of primes (so that their product is a sufficiently large number), one can obtain the full result in the initial domain of integers (rationals). Thus, the modular homomorphism

$$\phi_p : Z[x] \to Z_p[x]$$

is used to reduce the problem from *harder* ($Z[x]$) to *simpler* ($Z_p[x]$) domain, where the problem may be solved efficiently.

The same idea is used to reduce problems with multiple variables to univariate. For example, efficient algorithms for multivariate GCD substitute different values for all variables but one, then solve a univariate GCD, and then reconstruct multivariate GCD by doing several different substitutions via polynomial interpolation. Thus, the evaluation homomorphism

$$\phi_I : R[\vec{X}] \to R[x_0]$$

is used to reduce the problem from *harder* ($R[\vec{X}]$) to *simpler* ($R[x_0]$) domain, where the problem may be solved efficiently. Finally, figure 1 shows the homomorphism diagram for Chinese remainder and interpolation algorithms used to solve GCD / factorization / linear system and other problems with polynomials over integer (rational) numbers. The same techniques of ring homomorphism are used for solving problems in more complicated domains: e.g. problems with polynomials over algebraic number fields may be reduced to problems over Galois fields and then the answer may be reconstructed using Chinese remainders.

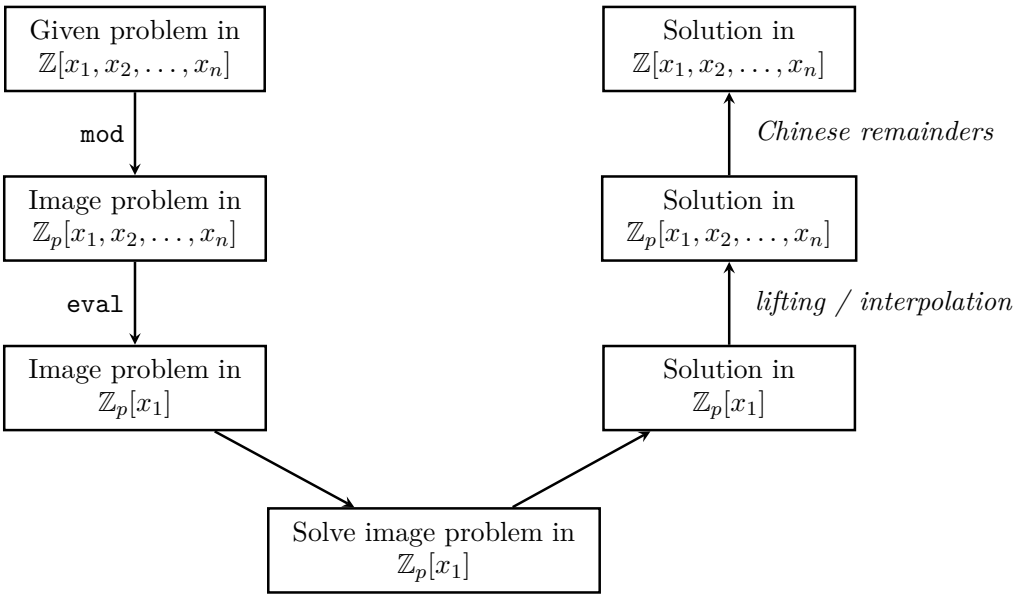

Figure 1. Homomorphism diagram for Chinese remainder and interpolation algorithms.

One of the key aspects in the implementation of above methods concerns the tight connection between abstract mathematics and generic programming. For example, the algorithms for computing GCDs or factoring univariate polynomials over finite fields $F_q[x]$ are formulated independently of the specific type of F_q: they are the same for both modular integers $\mathbb{Z}_q$, their finite extensions $\mathbb{Z}_p[t]/\langle f(t)\rangle$ etc. On the other hand, obviously, the data structures for integers and for finite extensions are completely different. Moreover, one may (and in fact should) use quite special CPU-optimized implementations for $\mathbb{Z}_p$ or $\mathbb{GF}(p,k)$ depending on the values of p and k. That is why the use of generic programming may significantly facilitate the development: one can write and optimize the algorithm only once and it will work with elements of different ring types.

The second key aspect is the performance: among the modern open source software there are only few tools which are able to perform routines such as multivariate polynomial GCD or factorization at a speed sufficient for challenging real-world problems. Existing tools with such functionality are commonly implemented as a computer-algebra systems and each has its own interactive interface and domain-specific programming language, which are in most cases very bad suited (compared to modern industrial object oriented languages) for developing of new generic algorithms.

These two aspects drove the development of Rings — a generic and high-performance library for commutative algebra written in Java and Scala languages [5]. Java is *perhaps* the most widely used language in industry today and combines several programming paradigms including object-oriented, generic, and functional programming. Scala, which is fully interoperable with Java, additionally implements several advanced concepts like pattern matching, an advanced type system, and type enrichment. Use of these concepts in Rings made it possible to implement mathematics in a quite natural and expressive way directly inside the programming environment offered by Java and Scala.

In a nutshell, Rings allows to construct different rings and perform arithmetic in them, including both very basic math operations and advanced methods like polynomial factorization, linear systems solving, and Gröbner bases. The built-in rings provided by the library include: integers $\mathbb{Z}$, modular integers $\mathbb{Z}_p$, finite fields $\mathbb{GF}(p^k)$ (with arbitrary large p and $k < 2^{31}$), algebraic field extensions $F(\alpha_1, \ldots, \alpha_s)$, fractions $Frac(R)$, univariate $R[x]$ and multivariate $R[\vec{X}]$ polynomial rings, where R is an arbitrary ground ring (which may be either one or any combination of the listed rings).

In further sections, we will illustrate the key features of Rings by a few examples given in Scala language. They can be evaluated directly in Rings REPL. The source code of the library is hosted at GitHub https://github.com/PoslavskySV/rings. Installation instructions and comprehensive online documentation (with both Java and Scala examples) can be found at https://rings.readthedocs.io.

2. Rings library overview

The high-level architecture of the Rings library is designed based on two key concepts: the concept of mathematical ring and the concept of generic programming. The use of generic programming allows one to systematically translate abstract mathematical constructions into machine data structures and algorithms. At the same time, the library remains completely type-safe due to the deep use of strong type model of the Scala language.

Generic programming, powered by the advanced type system of Scala language, provides a great level of abstraction when working with different rings. For example, consider the following generic implementation of Euclidean algorithm:

```
def gcd[E](a: E, b: E)(implicit ring: Ring[E]): E =
    if (b == ring(0)) a else gcd(b, a % b)
```

It works with elements of any (Euclidean) ring. E.g. apply it to elements of $\mathbb{Z}$:

```
implicit val zRing = Z // ring of (arbitrary precision) integers
val i1 = zRing(16) // convert machine number to element of ring
val i2 = zRing(18)
val iGcd = gcd(i1, i2)
assert ( iGcd == zRing(2) )
```

E.g. apply it to elements of $\mathbb{Q}[x]$:

```
implicit val pRing = UnivariateRing(Q, "x") // polynomials Q[x]
val p1 = pRing("1 - x^8") // parse poly from string
val p2 = pRing("1 + x^5")
val pGcd = gcd(p1, p2)
assert ( pGcd == pRing("1+x") )
```

Importantly, each object from the above example has complete compile-time type, which is just omitted for shortness, but inferred automatically by the compiler. So in fact, the above lines are effectively expanded to:

```
val pRing : UnivariateRing[Rational[IntZ]] = ...
val p1    : UnivariatePolynomial[Rational[IntZ]] = ...
val pGcd  : UnivariatePolynomial[Rational[IntZ]] = ...
```

⋮

Another key point is the use of `implicit` variables in connection with the Scala concept of "type enrichment". In Rings, it is used to add operator overloading for elements of arbitrary rings in an elegant way: all math operators (like modulo operator `%` used in the above `gcd` definition) work for arbitrary type `E`, provided that there is an `implicit` instance of `Ring[E]` in the scope:

```
implicit val ring : Ring[E] = ...   // implicit ring instance
val t1 : E = ... ; val t2 : E = ... // some ring elements
t1 % t2 // compiles to ring.remainder(t1, t2)
t1 / t2 // compiles to ring.divide(t1, t2)
t1 + t2 // compiles to ring.add(t1, t2)
t1 * t2 // compiles to ring.multiply(t1, t2)
```

⋮

The following example shows how the presence of an `implicit` ring changes the behaviour of math operators:

```
// some arbitrary precision integers
val t1 : IntZ = 12 ; val t2 : IntZ = 13
assert (t1 * t2 == Z(156)) // multiply integers
{
  implicit val ring = Zp(2)
  assert (t1 * t2 == ring(0)) // multiply modulo 2
}
{
  implicit val ring = Zp(17)
  assert (t1 * t2 == ring(3)) // multiply modulo 17
}
```

3. Polynomials, GCDs, and Factorization

Polynomials are the central objects in almost all practical computations. Rings provides separate implementations for univariate (dense) and multivariate (sparse) polynomials: univariate polynomials are represented by dense arrays, while multivariate polynomials are represented by binary trees. Additionally, there are special implementations for polynomials over $\mathbb{Z}_p$ with $p < 2^{64}$ (p fits in machine word). These implementations are highly optimized to achieve the best possible performance using machine intrinsics and special CPU instructions. For example, for univariate polynomials:

```
// univariate ring Z/p[t] with arbitrary large p
val mPrime = Z("2^521 - 1") // Mersenne prime
val uRingZp = UnivariateRing(Zp(mPrime), "t")
// optimized univariate ring Z/p[t] with machine p
val uRingZp64 = UnivariateRingZp64(17, "t")
```

Elements of these two rings have correspondingly different types:

```
val p1 : UnivariatePolynomial[IntZ] = uRingZp("(1 + t)^100")
val p2 : UnivariatePolynomialZp64 = uRingZp64("(1 + t)^100")
```

To further illustrate the features of Rings, let's consider an example of a multivariate polynomial ring over Galois field $\mathbb{GF}(17^3)$:

```
1  //Galois field GF(17^3) ("t" is the generator)
2  implicit val gf = GF(17, 3, "t")
3  val t  = gf("t")
4  val t1 = 3 + t - t.pow(22)/(1 + t + t.pow(9))
5  //compute e.g. minimal polynomial of t1
6  val mpoly = gf.minimalPolynomial(t1)
7  // assert that t1 is a root of mpoly
8  assert( gf(mpoly.composition(t1)).isZero )
```

Elements of this Galois field are internally represented as univariate polynomials over $\mathbb{Z}_{17}$. Define multivariate polynomial ring over the ground ring $\mathbb{GF}(17^3)$:

```
9  // multivariate ring GF(17^3)[x, y, z]
10 implicit val ring = MultivariateRing(gf, Array("x","y","z"), LEX)
11 // construct some multivariate polynomials
12 val p1 = ring("(1 + t + x + y + z)^3 - (1 - x/t - y/t)^3")
13 val p2 = ring("(1 - x^2 - y^2 - z^2)^4 + (1 + z/t)^4 - 1")
14 val p3 = (p1 + p2).pow(2) - 1
```

Again the `ring` instance is defined `implicit`, so all math operations with multivariate polynomials, which have the type

```
MultivariatePolynomial[UnivariatePolynomialZp64]
```

will be delegated to that instance.

In line 10, we explicitly specified to use `LEX` monomial order for multivariate polynomials. This choice affects some algorithms like multivariate division and Gröbner bases. The explicit order may be omitted (`GREVLEX` will be used by default).

Polynomial greatest common divisors and polynomial factorization work for polynomials over all available built-in rings. Now we continue our example:

```
15 // GCD of polynomials from GF(17^3)[x, y, z]
16 val gcd1 = ring.gcd(p1 * p3, p2 * p3)
```

```
17 assert ( gcd1 % p3 == ring(0) )
18 val gcd2 = ring.gcd(p1 * p3, p2 * p3 + 1)
19 assert ( gcd2.isConstant )

20 // large polynomial from GF(17,3)[x,y,z]
21 // with more than 10^4 terms and total degree of 123
22 val bigPoly = p1.pow(3) * p2.pow(2) * p3
23 // factorize it
24 val factors = ring.factor(bigPoly)
```

3.1. Benchmarks

Much attention in the library is paid to the performance of core algorithms. One of the main goals of Rings is to provide really fast implementations of modern algorithms.

To compare the speed of GCD with other tools, the following benchmark was used. Polynomials a, b, and g were generated at random and time needed to compute $gcd(ag, bg)$ was measured. Each polynomial had 40 terms (so the products ag and bg had at most 1600 terms each), and monomial exponents were generated using two strategies. In the first one (uniform), exponent of each variable in monomial was taken uniformly in $0 \leq \exp \leq 30$. In the second strategy (sharp) the total degree of each monomial was fixed and equal to 50 (so input polynomials were homogeneous). Benchmarking was performed for different numbers of variables. The performance of Rings (v2.3.2) was compared to Mathematica (v11.1.1.0), Singular (v4.1.0), FORM (v4.2.0) [6] and Fermat (v6.19) [7].

Figure 2 shows how the performance of different libraries behaves with the increase of the number of variables. In all considered problems performance of Rings was unmatched. Notably, its performance almost doesn't depend on the number of variables in such sparse problems.

Performance of polynomial factorization was tested using the following benchmark. Polynomials a, b, and c were generated at random and time needed to compute $factor(abc + 1)$ (trivial) and $factor(abc)$ (non trivial) was measured. Each polynomial had 20 terms (so the products abc had at most 8000 terms each). The exponent of each variable in monomials was chosen uniformly in $0 \leq \exp \leq 30$.

Figure 3 shows how the performance of multivariate factorization depends on the number of variables. It follows that the median time required to compute factorization changes quite slowly, while some outstanding points (typically ten times slower than median values) appear, when the number of variables becomes large.

The benchmarks shown above involve only sparse problems, which are more frequent in practice (especially in physics). The full set of benchmarks, including dense problems, can be found at https://github.com/PoslavskySV/rings.benchmarks.

4. Ideals and Gröbner Bases

The concept of mathematical ideal is implemented by the `Ideal` class, which computes corresponding Gröbner basis automatically at instantiation. The following code snippet continues our example from the previous section with polynomial ring $\mathbb{GF}(17, 3)[x, y, z]$ and illustrates the main methods provided by the `Ideal` class:

```
62 val (x, y, z) = ring("x", "y", "z")
63 // define a set of polynomial generators
64 val (i1,i2,i3) = (x.pow(16) +y + z, x-y-z, y.pow(8) - z.pow(8))
65 // construct Ideal from a set of generators
66 // (Groebner basis with GREVLEX order will be computed)
67 val ideal = Ideal(Seq(i1, i2, i3), GREVLEX)
```

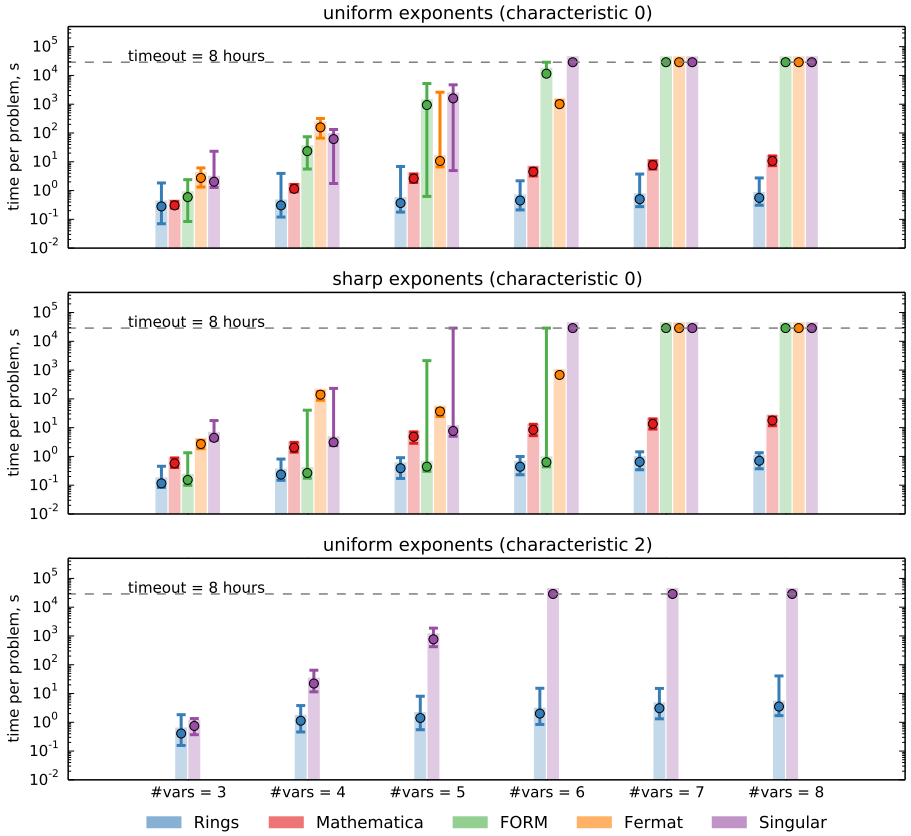

Figure 2. Dependence of multivariate GCD performance on the number of variables. Each problem set contains 110 problems, points correspond to the median times and the error bands correspond to the smallest and largest execution time required to compute the GCD within the problem set. If computation of a single GCD took more than 8 hours (timeout) it was aborted and the timeout value was adjoined to the statistics.

```
// print Groebner basis
println( ideal.groebnerBasis )
// print dimension of ideal
println( ideal.dimension )
// print degree of ideal
println( ideal.degree )
// print Hilbert series of ideal
println( ideal.hilbertSeries )
// reduce poly modulo ideal
val p4 = p2 %% ideal
```

Rings also provides built-in algorithms for manipulating ideals:

```
val othIdeal = Ideal(Seq(p1, p2), GREVLEX)
// union of ideals
val union = ideal + othIdeal
// product of ideals
val prod = ideal * othIdeal
```

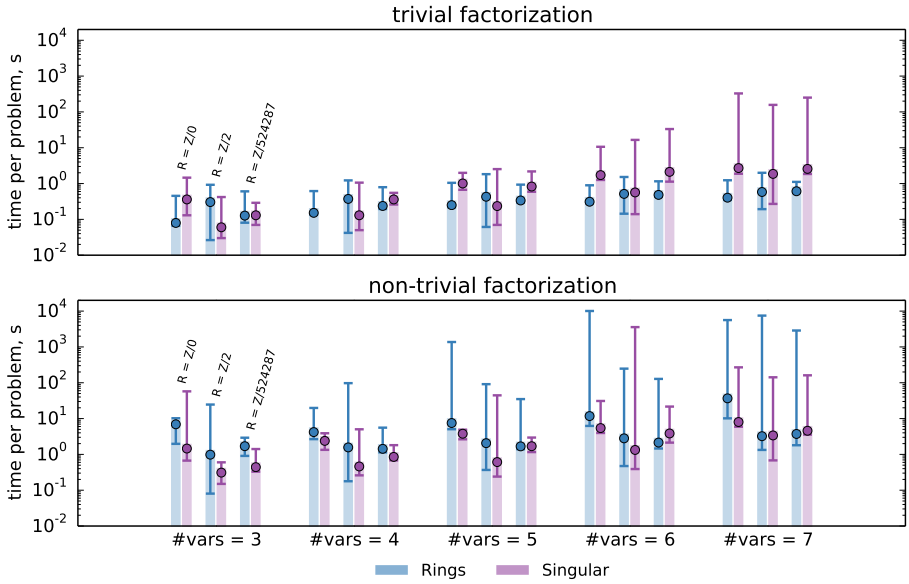

Figure 3. Dependence of multivariate factorization performance on the number of variables. Each problem set contained 110 problems, points correspond to the median times and the error bands correspond to the smallest and largest execution time required to compute the factorization within the problem set.

```
// intersection of ideals
val in = ideal intersection othIdeal
// quotient of ideals
val quot = othIdeal :/ ideal
```

4.1. Benchmarks

Rings implements Faugere's F4 and Buchberger's algorithms for computing Gröbner bases. These implementations show sufficient performance on small and medium problems. Table 1 shows time needed to compute a Gröbner bases of classical Katsura and cyclic systems for Rings, Mathematica and Singular. Timings are in general comparable between Rings and Singular for polynomial ideals over $\mathbb{Z}_p$ while for $\mathbb{Q}$ Rings behaves worse. It should be noted that for very hard problems, much more efficient dedicated tools like FGB [8] (proprietary) or OpenF4 [9] (open source) exist.

5. Conclusion and Future Work

There are two key facets of the programming of modern computer algebra algorithms: the desire to achieve high performance and the use of generic programming. Rings library combines a very generic programming approach with a really high-performance implementations, providing a well-designed generic API with a fully typed hierarchy of algebraic structures and algorithms for commutative algebra. Rings performance is similar or even unmatched in some cases to many advanced open-source and commercial software packages. The API provided by the library allows to write short and expressive code on top of the library, using both object-oriented and functional programming paradigms in a completely type-safe manner.

Some of the planned future work for Rings includes improvement of Gröbner bases algorithms

Table 1. Time required to compute Gröbner basis in graded reverse lexicographic order. In case of $\mathbb{Z}_p$ coefficient ring, value of $p = 1000003$ was used.

Problem	Ring	Rings	Mathematica	Singular
c-7	$\mathbb{Z}_p$	3s	26s	N/A
c-8	$\mathbb{Z}_p$	51s	897s	39s
c-9	$\mathbb{Z}_p$	14603s	∞	8523s
k-7	$\mathbb{Z}_p$	0.5s	2.4s	0.1s
k-8	$\mathbb{Z}_p$	2s	24s	1s
k-9	$\mathbb{Z}_p$	2s	22s	1s
k-10	$\mathbb{Z}_p$	9s	216s	9s
k-11	$\mathbb{Z}_p$	54s	2295s	65s
k-12	$\mathbb{Z}_p$	363s	28234s	677s
k-7	$\mathbb{Q}$	5s	4s	1.2s
k-8	$\mathbb{Q}$	39s	27s	10s
k-9	$\mathbb{Q}$	40s	29s	10s
k-10	$\mathbb{Q}$	1045s	251s	124s

(better implementation of "change of ordering algorithm" and some special improvements for polynomials over $\mathbb{Q}$), optimization of univariate polynomials with more advanced methods for fast multiplication, specific optimized implementation of $\mathbb{GF}(2, k)$ fields which are frequently arise in cryptography, and better built-in support for polynomials over arbitrary-precision real numbers ($\mathbb{R}[\vec{X}]$) and over 64-bit machine floating-point numbers ($\mathbb{R}64[\vec{X}]$).

Rings is an open-source library licensed under Apache 2.0. The source code and comprehensive online manual can be found at https://github.com/PoslavskySV/rings.

6. Acknowledgements

The author would like to thank the organizers of ACAT 2019. The author would like to thank Andrei Kataev for stimulating discussions. The work was supported by the Russian Science Foundation grant #18-72-00070.

7. References

[1] von Manteuffel A and Schabinger R M 2017 *Phys. Rev.* **D95** 034030 (*Preprint* 1611.00795)
[2] Smirnov A V and Chuharev F S 2019 (*Preprint* 1901.07808)
[3] Klappert J and Lange F 2019 (*Preprint* 1904.00009)
[4] Peraro T 2019 (*Preprint* 1905.08019)
[5] Poslavsky S 2019 *Comput. Phys. Commun.* **235** 400–413 (*Preprint* 1712.02329)
[6] Ruijl B, Ueda T and Vermaseren J 2017 (*Preprint* 1707.06453)
[7] Lewis R 2018 Computer Algebra System Fermat http://home.bway.net/lewis
[8] Faugère J C 2010 Fgb: A library for computing gröbner bases *Mathematical Software – ICMS 2010* ed Fukuda Komeiand Hoeven Joris van derand Joswig M T N (Berlin, Heidelberg: Springer Berlin Heidelberg) pp 84–87 ISBN 978-3-642-15582-6
[9] Titouan Coladon V V and Antoine Joux 2018 OpenF4 implementation https://github.com/nauotit/openf4