Rings: A JVM library for Commutative Algebra

S. Poslavsky
(Institute for High Energy Physics NRC "Kurchatov Institute" Protvino, Russia)

stvlpos@mail.ru

Introduction

RINGS is an open-source library written in Java and Scala which implements basic concepts and algorithms from computational commutative algebra. The goal is to provide a high-performance implementation packed into a lightweight *library* (not a full-featured CAS) with a clean API, which meets modern standards of software development.

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.

RINGS is a cross-platform library compatible with modern JVM-based languages (Java, Scala, Closure, Kotlin, Groovy, etc.) or easily interacted with from native applications both on POSIX systems and Windows, either through the Java Native Interface (C/C++) or via simple pipes.

The Scala extension allows for expressive type-safe interaction with the library from within a Scala application or from the REPL console. Both Java and Scala are strongly and statically typed languages, and the mathematical structures used in RINGS also form a fully typed hierarchy, in contrast to many computer-algebra systems and libraries that operate either with untyped or weakly typed objects or use duck typing. 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.

RINGS is hosted on github.com/PoslavskySV/rings. Installation instructions and comprehensive online documentation can be found at rings.readthedocs.io.

Overview

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 construction of Gröbner bases.

The built-in rings include $\mathbb{Z}$, finite fields $\mathbb{Z}_p$ and $\mathbb{GF}(p,k)$ (with arbitrarily large p), field extensions $F(\alpha_1,\ldots,\alpha_s)$ (including e.g. Gaussian numbers $\mathbb{Q}(i)$), $\mathrm{Frac}(R)$ (including e.g. rationals $\mathbb{Q}$ and rational functions), $R[x]$ and the multivariate polynomial ring $R[\vec{X}]$, where R is an arbitrary ground ring which may be either one or any combination of the listed rings.

Below we illustrate the features of RINGS in a step-by-step example. All code snippets are in Scala and can be evaluated directly in REPL. Java examples can be found in the online manual http://rings.readthedocs.io.

Basic concepts

To start our illustration, let's take a finite field $\mathbb{GF}(17,3)$ and perform some basic math in it:

```
1  // Galois field GF(17,3) ("t" is the generator)
2  implicit val gf = GF(17, 3, "t")
3  // parse ring element from string
4  val t  = gf("t")
5  // do some basic math (+-*/)
6  val t1 = 3 + t - t.pow(22)/(1 + t + t.pow(9))
7  // compute minimal polynomial of t1
8  val mpoly = gf.minimalPolynomial(t1)
9  // assert that t1 is a root of mpoly
10 assert( gf(mpoly.composition(t1)).isZero )
```

This very basic example already reveals some important programming concepts implemented in Java and Scala.

The first key point is that each object from the above example has full compile-time type, which is just omitted in our example for shortness but inferred automatically by the compiler. The above lines are in fact equivalent to:

```
val gf : GaloisField64 = ...
val t  : UnivariatePolynomialZp64       = ...
val t1 : UnivariatePolynomialZp64       = ...
val mpoly : UnivariatePolynomialZp64    = ...
```

`GaloisField64` implements the Galois field $\mathbb{GF}(p, q)$ with $p < 2^{64}$ (machine word) and is a subtype of `Ring[UnivariatePolynomialZp64]`. (The interface `Ring[E]` is a supertype for all rings; it defines all mathematical operations on elements of type `E`, plus some methods inherent to rings like `.isField()`, `.characteristic()`, and `.cardinality()`.)

`UnivariatePolynomialZp64` represents univariate polynomials over $\mathbb{Z}_p$ ($p < 2^{64}$) and is used as the actual representation of elements of Galois fields.

The second key point, specific to Scala programming, concerns the concept of *type enrichment* which allows to "enrich" existing classes by adding new functionality. In RINGS it is used to add operator overloading for elements of arbitrary rings in an elegant way: all math operators (e.g. `*` or `+`) work for an arbitrary type `E`, provided that there is an `implicit` instance of `Ring[E]` in the scope:

```
implicit val ring : Ring[E] = ...
val t1 : E = ... ; val t2 : E = ...
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 behavior of math operators:

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

The third point worth mentioning is the syntactic sugar applied in line 4, which is actually `gf.parse("t")` in full. In fact, there are several such methods for different conversions, which may all be called in the same way (this is a common pattern in Scala):

```
// from string
val elem = gf("1 + t^2")
// from Int
val unit = gf(1)
// from elements of other GF fields
val othFieldElement = GF(19, 5).randomElement()
val cast = gf(othFieldElement)
// syntactic sugar for multiple assignment
val (el1, el2) = gf("t + 1", "t + 2")
```

Polynomials, GCDs, Factorization

Our next step is to define some multivariate polynomial ring over the ground ring $\mathbb{GF}(17, 3)$. Below we define such a ring and perform some math operations in the same fashion we did above:

```
11 // multivariate ring GF(17,3)[x,y,z]
12 implicit val ring =
       MultivariateRing(gf, Array("x","y","z"),
       GREVLEX)
13 val (x, y, z) = ring("x", "y", "z")
14 // construct some multivariate polynomials
15 val p1 = (t.pow(2) + 1)*x*y.pow(2)*z +
        (t + 1)*x.pow(5) * z*y.pow(6) + 1
16 val p2 = p1.pow(2) + (t + 1)*x.pow(2)*y.pow(2) +
        (t.pow(9) + 1)*z.pow(7)
17 val p3 = (p1 + p2).pow(2) - 1
```

Again, the `ring` instance is defined `implicit`, hence all math operations on multivariate polynomials of type `MultivariatePolynomial[UnivariatePolynomialZp64]` will be delegated to that instance.

In line 12 we explicitly specified `GREVLEX` monomial ordering for multivariate polynomials. This choice affects 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. Continue our example:

```
18 // GCD of polynomials from GF(17,3)[x,y,z]
19 val gcd1 = ring.gcd(p1 * p3, p2 * p3)
20 assert ( gcd1 % p3 === 0 )
21 val gcd2 = ring.gcd(p1 * p3, p2 * p3 + 1)
22 assert ( gcd2.isConstant )
23
24 // large polynomial from GF(17,3)[x,y,z]
25 // with more than 4 × 10^3 terms and degree 204
26 val hugePoly = p1 * p2.pow(2) * p3.pow(3)
27 // factorize it
28 val factors = ring.factor(hugePoly)
```

One of the key features of the RINGS library is that it does polynomial GCD and factorization of really huge polynomials over different ground rings robustly and fast.

To illustrate how the performance of e.g. polynomial GCD is manifested in applications, suppose we need to solve a system of linear equations with symbolic coefficients. In the continuation of our example, we use $\mathbb{GF}(17, 3)[x, y, z]$ for the ring of coefficients, so the solution belongs to the field of rational functions over 3 variables with coefficients from $\mathbb{GF}(17, 3)$. This can be accomplished in RINGS easily:

```
29 //field of rational functions Frac(GF(17,3)[x,y,z])
30 implicit val ratRing = Frac(ring)
31 // convert x, y, z and t to rationals
32 val (rx, ry, rz, rt) = ratRing(x, y, z, ring(t))
33 // lhs matrix
34 val lhs =  Array(
       Array(rt + rx + rz, ry * rz, rz - rx * ry),
       Array(rx - ry - rt, rx / ry, rz + rx / ry),
       Array(rx * ry / rt, rx + ry, rz / rx + ry))
35 // rhs column
36 val rhs = Array(rx, ry, rz)
37 // solve the system with Gaussian elimination
38 val solution = LinearSolver.solve[ratRing.
       ElementType](ratRing, lhs, rhs)
```

Standard Gaussian elimination needs $O(n^3)$ field operations, which means $O(n^3)$ multivariate polynomial GCDs (since each fraction should be reduced). The speed-up of the polynomial GCD by a factor two reduces the time to solve the entire system by nearly an order of magnitude.

Benchmarks

To compare the speed of GCD with other tools, the following benchmark was used. Polynomials a, b, and g were generated at random and the 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), the exponent of each variable in the monomial was taken uniformly in $0 \le \exp \le 30$. In the second strategy (sharp) the total degree of each monomial was fixed and equal to 50 (so input polynomials were homogeneous). The benchmark was run for different numbers of variables. The performance of RINGS 2.3.2 was compared with that of MATHEMATICA 11.1.1, SINGULAR 4.1.0 [1], FORM 4.2.0 [2], and FERMAT 6.19 [3].

Fig. 1 shows how the performance of different systems compares with increasing number of variables. In all considered problems the performance of RINGS was unmatched. Remarkably, its performance is almost independent of the number of variables in such sparse problems.

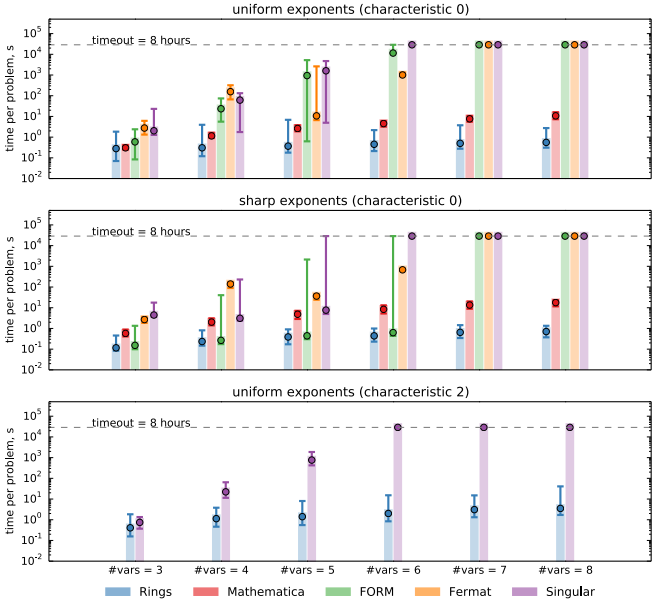

Figure 1: *Dependence of multivariate GCD performance on the number of variables. Each problem set contains 110 problems, points correspond to the median times, and the error bands correspond to the smallest and largest execution time required to compute the GCD within the problem set. If computation of a single GCD took more than 8 hours (timeout) it was aborted and the timeout value was adjoined to the statistics.*

Performance of polynomial factorization was tested using the following benchmark. Polynomials a, b, and c were generated at random and the time needed to compute $\mathrm{factor}(abc+1)$ (trivial) and $\mathrm{factor}(abc)$ (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 a monomial was chosen uniformly in $0 \le \exp \le 30$.

Fig. 2 shows how 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 outliers (typically ten times slower than median values) appear when the number of variables becomes large.

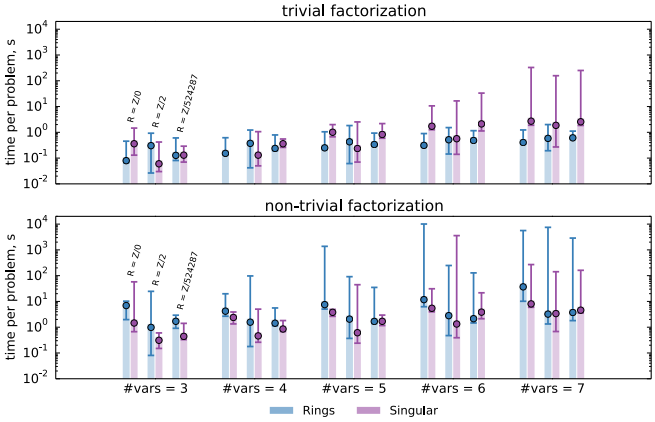

Figure 2: *Dependence of multivariate factorization performance on the number of variables. Each problem set contained 110 problems, points correspond to the median times, and the error bands correspond to the smallest and largest execution time required to compute the factorization within the problem set.*

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

Implementation notes

To achieve the high performance of polynomial GCD and factorization, RINGS uses different algorithms depending on the type of input.

First of all, univariate and multivariate polynomials have different implementations: univariate are represented as dense arrays while multivariate are represented as sparse containers (implemented with a red–black `TreeMap`). Polynomials over the ring $\mathbb{Z}_p$ with $p < 2^{64}$, elements of which can be represented as 64-bit integers, have separate, highly optimized implementations.

Univariate GCD uses the Half-GCD algorithm for polynomials over finite fields and modular algorithms in other cases (i.e. $\mathbb{Q}[x]$ and $\mathbb{Q}(\alpha)[x]$). Univariate polynomial factorization is implemented with the use of the Cantor–Zassenhaus method with optional use of Shoup's baby-step giant-step algorithm [4] (for large polynomials or for finite fields with large characteristic).

Multivariate GCD switches between Zippel's sparse algorithms and Enhanced Extended Zassenhaus algorithm (EEZ-GCD). The latter is used only on very dense problems. Zippel's algorithms require that the ground ring contains a sufficient number of elements (so they will always fail in e.g. $\mathbb{Z}_2[\vec{X}]$). When the cardinality of a ground ring is not sufficiently large, RINGS switches to a Kaltofen–Monagan generic modular algorithm [5]. For polynomials over algebraic number fields, the modular approach with either sparse (Zippel) or dense (EEZ-GCD) interpolation is used with further rational number reconstruction.

Multivariate factorization uses Kaltofen's algorithm [6], with major modifications due to Lee [7]. For factoring bivariate polynomials, the very efficient Bernardin

algorithm [8] is used. Additionally, RINGS performs some fast early checks based on Newton polygons to ensure that there is a nontrivial factorization pattern. Multivariate Hensel lifting is done via a Zippel-like sparse method: the problem of lifting is reduced to a system of (in general non-linear) equations which may be solved efficiently in many cases. For factorization of polynomials over algebraic number fields $\mathbb{Q}(\alpha)$ RINGS uses Trager's algorithm [9].

Ideals and Gröbner bases

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

```
// define a set of polynomial generators
val (i1, i2, i3) = (x + y + z, x - y - z,
                    y.pow(2) - z.pow(2))
// construct Ideal from a set of generators
// (Groebner basis with GREVLEX order will be
    automatically computed)
val ideal = Ideal(Seq(i1, i2, i3))
// 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, p3))
// union of ideals
val union = ideal + othIdeal
// product of ideals
val prod = ideal * othIdeal
// intersection of ideals
val in = ideal intersection othIdeal
// quotient of ideals
val quot = othIdeal :/ ideal
```

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

Problem	Ring	RINGS	MMA	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

Implementation and benchmarks

RINGS implements Faugère's F4 and Buchberger's algorithms for computing Gröbner bases. These implementations show sufficient performance on small and medium problems. Table 1 shows the time needed to compute Gröbner bases of classical Katsura and cyclic systems for RINGS, MATHEMATICA (MMA), 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 [10] (proprietary) or OPENF4 [11] (open source) exist.

Programming with RINGS in Scala

The important feature of RINGS is that it allows to write short and expressive code on top of it using both object-oriented and powerful functional programming. Consider the following short example, which implements a solver for Diophantine equations, i.e. a straightforward generalization of the extended GCD on more than two arguments:

```
/**
 * Solves equation $\sum f_i s_i = \gcd(f_1,\ldots,f_N)$ for
     given $f_i$ and unknown $s_i$
 * @return a tuple (gcd, solution)
 */
def solveDiophantine[E](fi: Seq[E])
                       (implicit ring: Ring[E])=
  fi.foldLeft((ring(0), Seq.empty[E])) {
    case ((gcd, seq), f) =>
      val xgcd = ring.extendedGCD(gcd, f)
      (xgcd(0), seq.map(_ * xgcd(1)) :+ xgcd(2))
  }
```

With this function it is very easy to implement, for example, an efficient algorithm for partial fraction decomposition with just a few lines of code. The resulting function will work with elements of arbitrary fields of fractions:

```
/** Computes partial fraction decomposition of
    given rational */
def apart[E](frac: Rational[E]) = {
  implicit val ring: Ring[E] = frac.ring
  val facs = ring
               .factor(frac.denominator)
               .map {case (f, exp) => f.pow(exp)}
  val (gcd,  nums) = solveDiophantine(facs.map(
    frac.denominator / _))
  val (ints, rats) = (nums zip facs)
    .map { case (num, den) =>
       Rational(frac.numerator * num, den * gcd)
    }
    // extract integral parts
    .flatMap(_.normal)
    // separate integrals and fractions
    .partition(_.isIntegral)\

  // return the result
  rats :+ ints.foldLeft(Rational(ring(0)))(_+_)
}

// partial fraction decomposition for rationals
val qFracs = apart( Q("1234213 / 2341352") )

// partial fraction decomposition for functions
val ufRing = Frac(UnivariateRingZp64(17, "x"))
val expr = ufRing("1 / (3 - 3*x^2 - x^3 + x^5)")
val pFracs = apart(expr)
```

The function `apart[E]` is defined as a generic function which can be applied to fractions over elements of arbitrary rings (that should be Euclidean rings of course). Returning to our initial example where we've constructed a field $\mathrm{Frac}(\mathbb{GF}(17,3)[x,y,z])$, let us adjoin a new variable, say W, and construct the partial fraction decomposition in this complicated field:

```
105 // partial fraction decomposition of rational
        functions
106 // in the ring Frac(GF(17,3)[x,y,z])[W]
107 implicit val uRing = UnivariateRing(ratRing,"W")
108 val W = uRing("W")
109 val fracs = apart(Rational(W + 1,
        (rx/ry + W.pow(2)) * (rz/rx + W.pow(3))))
```

The function call on the last line involves nearly all main components of RINGS library: from very basic algebra to multivariate factorization over sophisticated rings.

The above examples show how powerful features of RINGS in a combination with expressive and type-safe Scala syntax may be used to implement quite non-trivial and generic functionality with little effort.

Summary and outlook

RINGS is a high-performance and lightweight library for commutative algebra that provides both basic methods for manipulating with polynomials and high-level methods including polynomial GCD, factorization, and Gröbner bases over sophisticated ground rings. Special attention was paid to high performance and a well-designed API. High performance is crucial for today's computational problems that arise in many research areas including high-energy physics, commutative algebra, cryptography, etc. 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 (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 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 http://ringsalgebra.io.

References

[1] W. Decker, G.M. Greuel, G. Pfister, H. Schönemann (2018), SINGULAR — A computer algebra system for polynomial computations, `http://www.singular.uni-kl.de`.

[2] B. Ruijl, T. Ueda, J. Vermaseren (2017), FORM version 4.2, arXiv:1707.06453.

[3] R. Lewis (2018), Fermat, `http://home.bway.net/lewis`.

[4] V. Shoup (1995), A new polynomial factorization algorithm and its implementation, *J. Symb. Comput.* 4 (20) 363–397.

[5] E. Kaltofen, M.B. Monagan (1999), On the genericity of the modular polynomial gcd algorithm, in: *Proceedings of ISSAC'99* 59–66, ACM Press.

[6] E. Kaltofen (1985), Sparse hensel lifting, in: *Proceedings of EUROCAL'85* (2) 4–17, Springer.

[7] M.M. Lee (2013), Factorization of multivariate polynomials, PhD thesis, University of Kaiserslautern.

[8] L. Bernardin, M. Monagan (1997), Efficient multivariate factorization over finite fields, on: *Proceedings of AAECC'97* (1255) 15–28, Springer.

[9] B.M. Trager (1976), Algebraic factoring and rational function integration, in: *Proceedings of SYMSAC '76* 219–226, ACM Press.

[10] J.C. Faugère (2010), FGb: a library for computing Gröbner bases, in: *Mathematical Software – ICMS 2010* 84–87.

[11] V.V. Titouan Coladon, A. Joux (2018), OpenF4, `https://github.com/nauotit/openf4`.

Spiegelungen am Kreis: A case for a CAS

J. Meyer
(Hameln)

j.m.meyer@t-online.de

Einführung

Man kann nicht nur an Punkten oder Geraden spiegeln, sondern auch an Kurven. Möchte man das tun, so liegt folgendes Verfahren nahe: Man sucht sich zum Originalpunkt P den nächstgelegenen Kurvenpunkt K und führt anschließend eine Punktspiegelung von P an K aus.
Dies Verfahren wird hier für den Fall untersucht, dass die Kurve ein Kreis um den Ursprung O mit dem Radius r ist und dass eine Gerade abgebildet wird. Es ist zu beachten, dass die hier vorgestellte Spiegelung am Kreis etwas anderes bedeutet als die übliche Kreisinversion.

Ermittlung der Kurvenpunkte

Man bekommt den zu P nächstgelegenen Punkt K_n (der Index n steht für „nah“) des Kreises, indem man die Ursprungsgerade mit dem allgemeinen Punkt $X = t \cdot P$ mit dem Kreis schneidet; das Ergebnis ist $t = \frac{r}{\sqrt{P \cdot P}}$ und damit

$$K_n = \frac{r \cdot P}{\sqrt{P \cdot P}}.$$

Der Kreispunkt K_f (der Index steht für „fern“) mit der größten Entfernung zu P ist $K_f = -K_n$. Die jeweiligen Bildpunkte sind

$$Q_n = 2 \cdot K_n - P$$

und

$$Q_f = 2 \cdot K_f - P = -2 \cdot K_n - P \quad \text{(Abb. 1).}$$

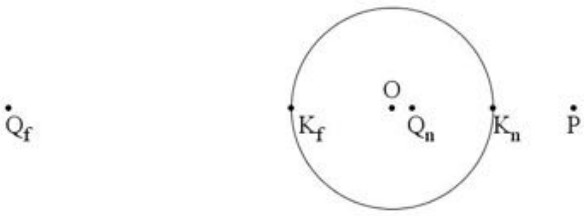

Abbildung 1: *Spiegelung am Kreis*

Nun durchlaufe P eine Gerade. Da beide Achsen jeweils auf sich abgebildet werden, kann man als Geradengleichung $x = x_0$ mit $x_0 > 0$ annehmen. Der allgemeine Punkt der Gerade ist $P(t) = \begin{pmatrix} x_0 \\ t \end{pmatrix}$, und

$$Q_n(t) = \left(\frac{2r}{\sqrt{x_0^2 + t^2}} - 1 \right) \cdot \begin{pmatrix} x_0 \\ t \end{pmatrix}$$

und

$$Q_f(t) = \left(\frac{-2r}{\sqrt{x_0^2 + t^2}} - 1 \right) \cdot \begin{pmatrix} x_0 \\ t \end{pmatrix}$$

durchlaufen Kurven.

Nullstellen

Betrachten wir

$$Q_n(t) = \left(\frac{2r}{\sqrt{x_0^2 + t^2}} - 1 \right) \cdot \begin{pmatrix} x_0 \\ t \end{pmatrix}.$$

Stets führt $t = 0$ zu einer Nullstelle, nämlich zu

$$Q_n(0) = \left(\frac{2r}{x_0} - 1 \right) \cdot \begin{pmatrix} x_0 \\ 0 \end{pmatrix}.$$

Es gibt weitere Nullstellen, wenn der Vorfaktor

$$2r - \sqrt{x_0^2 + t^2}$$

verschwindet; dann liegen diese Nullstellen im Ursprung.
Ist $2r < x_0$, so schneidet die Kurve die x-Achse nur für $t = 0$ (Abb. 2; die dicken Punkte sind Wendepunkte und werden weiter unten erläutert).

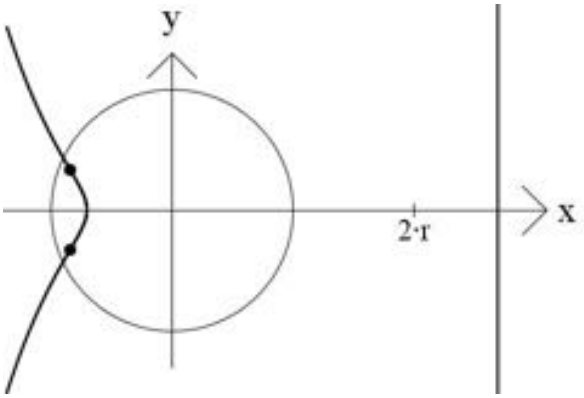

Abbildung 2: *Nur eine Nullstelle*

Ist $0 < x_0 < 2r$, so schneidet die Kurve die x-Achse außerdem für

$$t = \pm\sqrt{4r^2 - x_0^2};$$

die Kurve hat im Ursprung einen *Doppelpunkt* (Abb. 3).

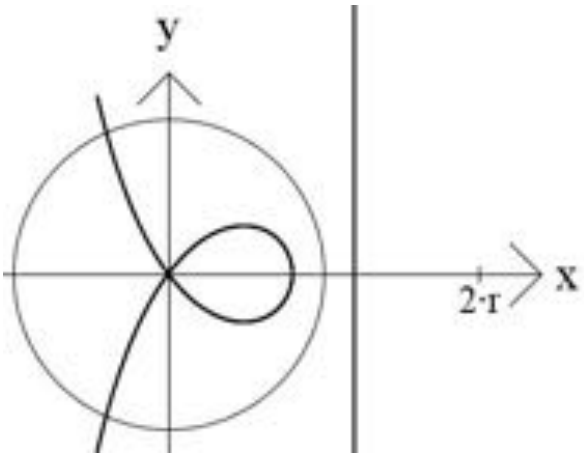

Abbildung 3: *Ein Doppelpunkt*

Ist $x_0 = 2r$, so hat der allgemeine Kurvenpunkt die Gestalt

$$Q_n(t) = \left(\frac{2r}{\sqrt{4r^2 + t^2}} - 1\right) \cdot \begin{pmatrix} 2r \\ t \end{pmatrix}.$$

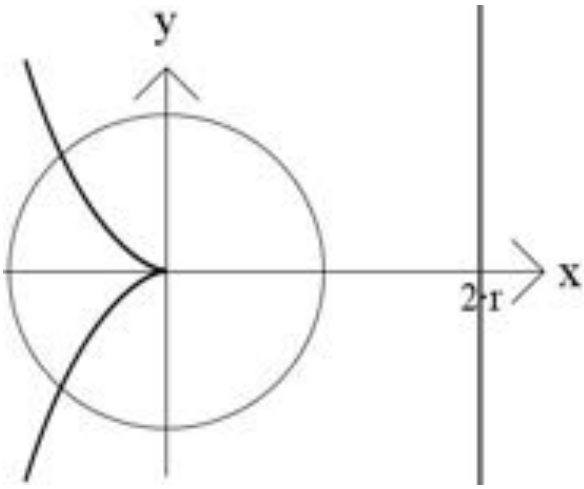

Abbildung 4: *Eine Spitze*

Die Tangente durch $Q_n(t)$ und den Ursprung hat die Steigung $\frac{t}{2r}$; sie hat für $t \to 0$ den Grenzwert 0. Daher hat die Kurve im Ursprung eine *Spitze* (Abb. 4).

Zur Gleichung

Es war

$$Q_n(t) = \left(\frac{2r}{\sqrt{x_0^2 + t^2}} - 1\right) \cdot \begin{pmatrix} x_0 \\ t \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix}.$$

Mit $T := t^2$ bekommt man aus der ersten Zeile die Gleichung

$$\frac{x}{x_0} + 1 = \frac{2r}{\sqrt{x_0^2 + T}},$$

woraus sich nach Quadrieren T ermittelt. Die quadrierte zweite Zeile

$$y^2 = (\frac{x}{x_0})^2 \cdot T$$

liefert die *quartische* Gleichung

$$(x + x_0)^2 \cdot \left(x^2 + y^2\right) = 4r^2x^2.$$

Man sieht, dass es auch beim Einsatz eines CAS einige mathematische Ideen braucht.
An der Gleichung ist zu erkennen, dass es maximal vier Nullstellen geben kann. Das liegt daran, dass auch die *Fernpunkte* $Q_f(t)$ diese Gleichung erfüllen. Die Kurve der Fernpunkte liefert allerdings wenig Variation (Abb. 5, die auch die weiter unten behandelten Wendepunkte zeigt).

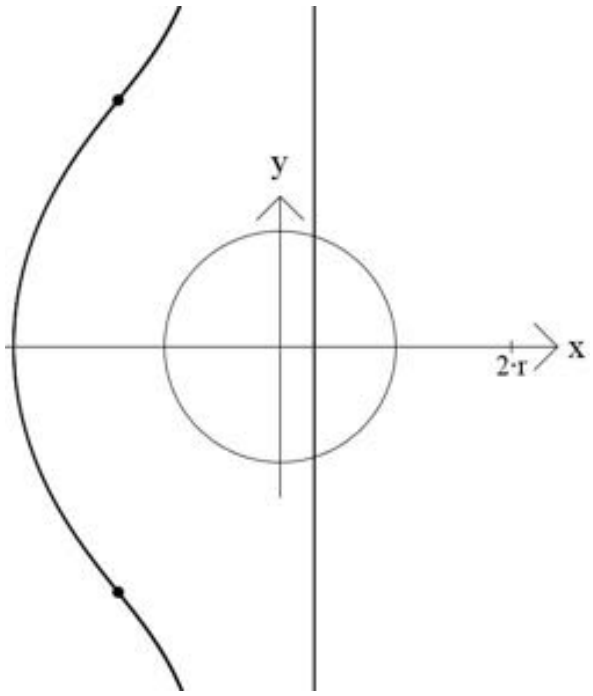

Abbildung 5: *Die Kurve der Fernpunkte*

Exkurs zum Krümmungskreismittelpunkt

Es sieht so aus, als hätten manche Kurven Wendepunkte (wie etwa Abb. 2 oder Abb. 5 zeigen). Dies soll hier näher untersucht werden.

Gegeben sei eine Kurve mit dem allgemeinen Punkt $K(t)$. Man bekommt den *Krümmungskreismittelpunkt* für den Parameter a, indem man die Kurvennormalen zu $K(a)$ und zu $K(a+h)$ miteinander schneidet und anschließend h gegen 0 laufen lässt.

Die Kurvennormale zu $K(a)$ hat den allgemeinen Punkt

$$X = K(a) + \lambda \cdot K'(a)^{\perp},$$

und der allgemeine Punkt X der Kurvennormale zu $K(a+h)$ erfüllt die Gleichung

$$X \cdot K'(a+h) = K(a+h) \cdot K'(a+h) \quad \text{(Hesseform)},$$

was zu

$$\lambda = \frac{(K(a+h) - K(a)) \cdot K'(a+h)}{K'(a+h) \cdot K'(a)^{\perp} - \underbrace{K'(a) \cdot K'(a)^{\perp}}_{0}}$$

$$= \frac{\frac{K(a+h)-K(a)}{h} \cdot K'(a+h)}{\frac{K'(a+h)-K'(a)}{h} \cdot K'(a)^{\perp}}$$

führt, das für $h \to 0$ gegen

$$\frac{K'(a) \cdot K'(a)}{K''(a) \cdot K'(a)^{\perp}}$$

strebt. Damit erhält man den Krümmungskreismittelpunkt

$$K(a) + \frac{K'(a) \cdot K'(a)}{K''(a) \cdot K'(a)^{\perp}} \cdot K'(a)^{\perp}.$$

Ist $K''(a) \cdot K'(a)^{\perp} = 0$ und $K'(a) \neq \begin{pmatrix} 0 \\ 0 \end{pmatrix}$, so ist $K(a)$ ein *Wendepunkt*.

Für welche Parameter liegt ein Wendepunkt vor?

Es war

$$Q_n(t) = \left(\frac{2r}{\sqrt{x_0^2 + t^2}} - 1 \right) \cdot \begin{pmatrix} x_0 \\ t \end{pmatrix}.$$

Die Wendepunktsbedingung führt mit CAS-Hilfe auf die Gleichung

$$\left(x_0^2 - 2t^2\right) \cdot \sqrt{x_0^2 + t^2} - 2r x_0^2 = 0,$$

woraus mit $z := x_0^2$ und $T := \frac{2t^2}{z}$ einerseits die Forderung

$$(1 - T) \cdot \sqrt{z + \frac{z \cdot T}{2}} = 2r$$

und damit $0 < T < 1$ sowie andererseits nach Quadrieren die kubische Gleichung $g(T) = d$ mit

$$g(T) := T^3 - 3T$$

und

$$d := \frac{8 \cdot r^2}{z} - 2$$

folgt. Hier ist das Lösungsverhalten übersichtlich (Abb. 6).

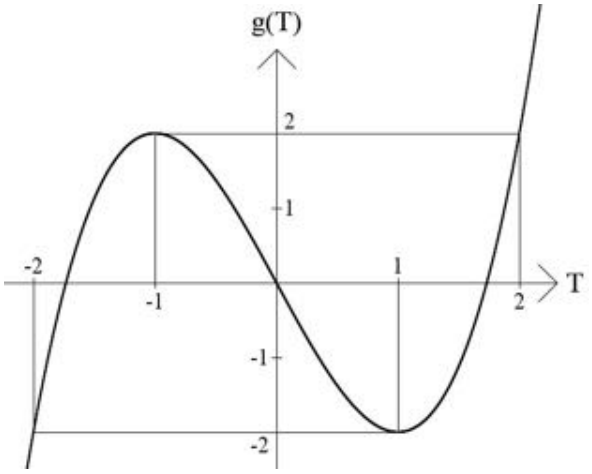

Abbildung 6: *Graph zur kubischen Gleichung*

Ist $0 < T < 1$, so muss $-2 < d < 0$ bzw. $0 < 4r^2 < z$ sein; eine notwendige Bedingung für die Existenz von Wendepunkten ist also $2r < x_0$. Die Lösungen von $T^3 - 3T = d$ sind bekanntlich durch

$$T = 2 \cdot \cos\left(\frac{\arccos(\frac{d}{2})}{3} + k \cdot 120^{\circ} \right)$$

gegeben; für unsere Zwecke ist $0 < T < 1$ notwendig. Die dicken Punkte in Abb. 2 sind die so berechneten Wendepunkte. Auch hier sieht man, dass die mathematischen Überlegungen auch mit Einsatz eines CAS nicht trivial werden.

Die Wendepunkte der Fernkurve

Die Wendepunktsbedingung führt auf

$$\left(x_0^2 - 2t^2\right) \cdot \sqrt{x_0^2 + t^2} + 2r x_0^2 = 0$$

und mit $z := x_0^2$ und $T := \frac{2t^2}{z}$ auf

$$(T - 1) \cdot \sqrt{z + \frac{z \cdot T}{2}} = 2r$$

und somit auf die notwendige Bedingung $T > 1$. Die kubische Gleichung ist dieselbe wie bei den Nahpunkten. Ist $|d| > 2$, so muss man in der Lösungsformel nur cos durch cosh und arccos durch arcosh ersetzen. Es gibt stets Wendepunkte.

Abschlussbemerkungen

Hier bieten sich sofort viele Anschlussfragen an, etwa nach der Kurve, auf der alle Wendepunkte liegen, oder danach, welche Kurven sich ergeben, wenn man nicht eine Gerade, sondern eine Parabel am Kreis spiegelt (Abb. 7 zeigt das Bild einer nach rechts geöffneten Parabel) oder einen anderen Kreis (Abb. 8 zeigt das Bild eines nach rechts verschobenen Kreises).

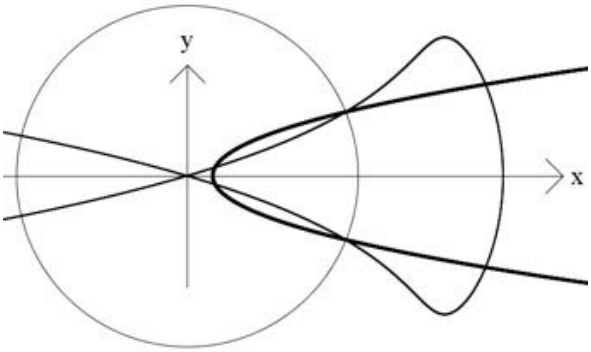

Abbildung 7: *Spiegelung einer Parabel*

Auch wenn die Kurvengleichungen höhergradig werden und nicht mehr mit elementaren Mitteln untersucht werden können, kann der unvermutete Formenreichtum der Kurven zum Erstaunen führen.

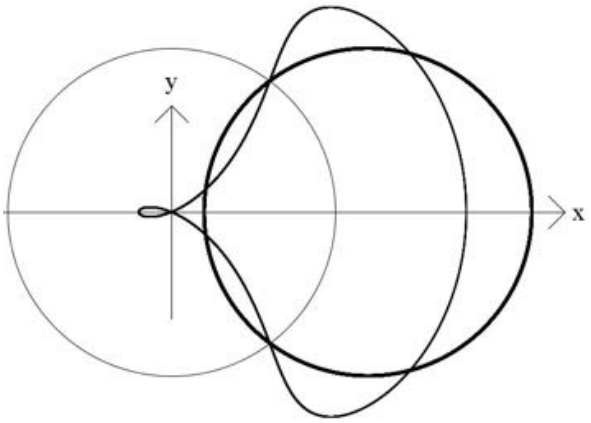

Abbildung 8: *Spiegelung eines Kreises*