An Implementation of Elliptic Curves in CASA

Christian Aistleitner
Research Institute for Symbolic Computation
Johannes Kepler University
Linz, Austria

March 11, 2003

Abstract

Parts of the RISC team have spent lots of time dealing with elliptic and hyper-elliptic curves. But until now there has been no single algorithm available for dealing with these types of curves in `CASA`, a computer algebra system, developed by the research group of Franz Winkler at RISC. Since also `Maple`, on which `CASA` is based, offers hardly any functionality, when it comes to elliptic curves, the necessity arose, to enrich the functionality of `CASA` at a first stage in terms of elliptic curves. With this implementation a solid basis should be built, which should carry the advantage of the flexibility of both systems over to the topic of elliptic curves.

Furthermore an approach was made to illustrate the group law on elliptic curves in the most general way. This helps in facilitating demonstrations when it comes to elliptic curves and also conveying the idea of how the group law really works on elliptic curves.

1 An Introduction to Elliptic Curves

An elliptic curve is a projective, non-singular curve of degree 3 over the algebraic closure $\bar{K}$ of a field K. (So every elliptic curve is a curve of genus 1)

That is, the points $(X : Y : Z) \in \mathbb{P}^2(\bar{K})$ on the elliptic curve are the solutions of

$$\begin{aligned} aX^3 + bX^2Y + cX^2Z + dXY^2 + eXYZ + \\ + fXZ^2 + gY^3 + hY^2Z + iYZ^2 + jZ^3 &= 0 \end{aligned}$$

with fixed coefficients $a, b, c, d, e, f, g, i, j \in K$ such that the partial derivatives with respect to X, Y and Z do not all vanish simultaneously and at least one of $a, g, j \neq 0$.

As seen in [10] one can always find a mapping ϕ, between solutions of the cubic equation above to the solutions of a simpler equation, the so-called homogeneous Weierstrass equation

$$E_{hom} : Y^2Z + a_1XYZ + a_3YZ^2 = X^3 + a_2X^2Z + a_4XZ^2 + a_6Z^3$$

over $\mathbb{P}^2(\bar{K})$ where $a_1, a_2, a_3, a_4, a_6 \in K$, and E_{hom} is non-singular (i.e.: not all partial derivatives vanish simultaneously), such that ϕ is a group-isomorphism for the group, which will be defined afterwards.

Since such an elliptic curve is uniquely defined by the coefficients a_1, a_2, a_3, a_4, a_6 (along with the field K, which often are understood, and therefore neglected) lots of algorithms are based on these 5 coefficients, and some constants are defined in order to be able to get shorter expressions

$$\begin{aligned}
b_2 &:= a_1^2 + 4a_2 \\
b_4 &:= a_1a_3 + 2a_4 \\
b_6 &:= a_3^2 + 4a_6 \\
b_8 &:= a_1^2a_6 + 4a_2a_6 - a_1a_3a_4 + a_2a_3^2 - 4a_4^2 \\
c_4 &:= b_2^2 - 24b_4 \\
c_6 &:= -b_2^3 + 36b_2b_4 - 216b_6 \\
\Delta &:= -b_2^2b_8 - 8b_4^3 - 27b_6^2 + 9b_2b_4b_6
\end{aligned}$$

Δ is also referred to as the discriminant of the curve, and plays an important role. For example $\Delta \neq 0$ if and only if the curve is non-singular, as can be seen right away, when combining the partial derivatives of E_{hom}.

Now with an elliptic curve of this homogeneous Weierstrass form at hand, one can easily see that $(0 : 1 : 0)$ is on every such elliptic curve (independent of the chosen a_1, a_2, a_3, a_4, a_6) and furthermore, that it is the only point on this curve with $Z = 0$. Having this in mind, E_{hom} can be de-homogenized with respect to the Z-coordinate and finally the occurring X/Z and Y/Z replaced by X and Y.

Here we choose to de-homogenize with respect to Z. Indeed one can also pick X or Y. This would lead to a different point at infinity, but would not change the structure of the elliptic curve. Resembling the most general cubic equation of an elliptic curve from the beginning of this section, one can also define an elliptic curve, as an affine, non-singular curve of degree 3 together with its unique simple point at infinity (which is determined by the way the homogeneous equation got de-homogenized). Anyway it is used throughout literature to de-homogenize the homogeneous Weierstrass equation E_{hom} with respect to Z. So we also choose Z in order of accordance and finally get the affine Weierstrass equation E_{aff}

$$E_{aff} : Y^2 + a_1XY + a_3Y = X^3 + a_2X^2 + a_4X + a_6$$

It it easily observed that the solutions of E_{aff} are equivalent to those of E_{hom} without $(0 : 1 : 0)$. But for convenience the affine version is used from now on, knowing that one has to add $(0 : 1 : 0)$ to the set of solutions.

Throughout literature the term "elliptic curve" is used for this affine Weierstrass representation of an elliptic curve. And again in order of concordance and simplicity the more general case is neglected, and we want to use the term "elliptic curve" for curves of this affine Weierstrass representation from now on. This does not limit the possible curves one can describe, since every elliptic curve can be brought into this form.

One can now investigate an elliptic curve (which now is in affine Weierstrass form) further and will immediately see, that every straight line intersects an elliptic curve in exactly three, not necessarily distinct points. If you choose two points $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$ on the curve, then let the third intersection point of the curve and the line defined by these two points be denoted by $P * Q$. (Note, that this is NOT a group law on the set of rational points on the curve)

$P * Q = (x_{P*Q}, y_{P*Q})$ can be derived by

- $(x_P \neq x_Q)$

$$\begin{aligned}
&\lambda = \tfrac{y_Q - y_P}{x_Q - x_P} \\
&\mu = \tfrac{y_P x_Q - y_Q x_P}{x_Q - x_P} \\
&x_{P*Q} = \lambda^2 + a_1\lambda - a_2 - x_P - x_Q \\
&y_{P*Q} = \lambda x_{P*Q} + \mu
\end{aligned}$$

- $(x_P = x_Q) \wedge (y_P = y_Q) \wedge (2y_P + a_1 x_P + a_3 \neq 0)$

$$\begin{aligned}
&\lambda = \tfrac{3x_P^2 + 2a_2 x_1 + a_4 - a_1 y_1}{2y_P + a_1 x_P + a_3} \\
&\mu = \tfrac{-x_P^3 + a_4 x_P + 2a_6 - a_3 y_P}{2y_P + a_1 x_P + a_3} \\
&x_{P*Q} = \lambda^2 + a_1\lambda - a_2 - x_P - x_Q \\
&y_{P*Q} = \lambda x_{P*Q} + \mu
\end{aligned}$$

- $(x_P = x_Q) \wedge ((y_P \neq y_Q) \vee (2y_P + a_1 x_P + a_3 = 0))$

$$P * Q = (0 : 1 : 0)$$

But what we actually want to achieve is to turn the set of rational points on an elliptic curve into a group. Therefore first of all a neutral element is needed. So we choose a point on the curve, which should be the neutral element. This point will be denoted by $\mathcal{O}$ and called "origin". Now $(P * Q) * \mathcal{O}$ turns out to be a valid group law, which shall be denoted by $P + Q$.

In general $(0 : 1 : 0)$ is taken to be the origin $\mathcal{O}$. Any other point would do as well, and the result would still be a group law. Of course changing $\mathcal{O}$ does have an impact on the coordinates of $P+Q$, but still it turns out to be a group law.

In order to calculate $P+Q$ we first of all need to get the third intersection point of the curve and the line through P and Q. Then the line of this intersection point and $\mathcal{O}$ is intersected with the elliptic curve and the third intersection point is then $P+Q$. If $\mathcal{O}=(0:1:0)$ then instead of second line intersection, one can also invert the y-coordinate of $P*Q$ which of course is faster.

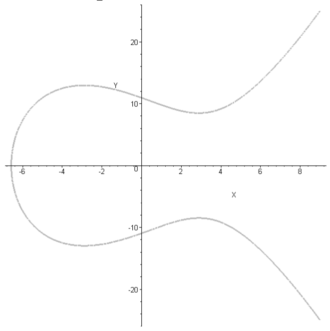
More on explicit formulas of operations and efficient algorithms can be found in [2].

Since operations over the field K generally might be time consuming, it is of great importance to reduce the number of operations needed. Therefore one tries to find isomorphisms onto curves, where some of the coefficients are zero. Then a speed-up is achieved, since some factors no longer have to be computed, because the term would vanish anyway.

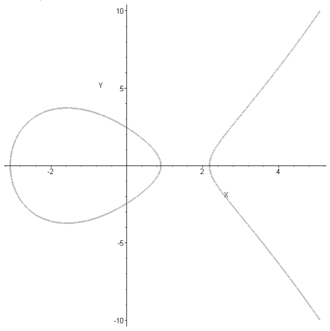
In [10] a proof can be found, that isomorphic curves are equivalent with respect to the j-invariant, which if and only if $\Delta \neq 0$ is defined as

$$j(E) := \frac{c_4^3}{\Delta}$$

So depending on the characteristic of the field K various linear transformations to other isomorphic elliptic curves E' (with the corresponding coefficients $a_1', a_2', a_3', a_4', a_6'$) can be found. In the case, where the characteristic of K is 0 or >3 one can transform the curve into E' with $a_1' = a_2' = a_3' = 0$. Since such a transformation is always possible over $\mathbb{Q}$, most pictures you get to see of elliptic curves will be of a curve with $a_1 = a_2 = a_3 = 0$. There are two possibilities. Either there is just one component in the affine plane, or there are two.



$y^2 = x^3 - 25x + 120$
an elliptic curve with
one component



$y^2 = x^3 - \frac{15}{2}x + 6$
an elliptic curve with
two components

If the characteristic of K is 2, transformations to E' can be found, where either $a_1' = 0$ or $a_1' = 1 \wedge a_3' = 0$ depending on whether $a_1 = 0$

or $a_1 \neq 0$. Finally if the characteristic of K is equal to 3, then an isomorphic E' can be found, with either $a'_1 = a'_3 = a'_4 = 0$ or $a'_1 = a'_2 = a'_3 = 0$ depending on whether $b_2 = 0$ or $b_2 \neq 0$.

In order to take advantage of these isomorphic curves, one maps a point onto the simpler curve. Then the operations are performed on this simpler curve, and finally the result is transformed back onto the original curve. This approach is extremely useful, if intermediate results are not of importance, as for example when it comes to accumulation.

Since we now have a proper group, we are of course also interested in the structure of the group.

A theorem due to Mordell shows, that the group of rational points of an elliptic curve over $\mathbb{Q}$ (denoted by $E(\mathbb{Q})$) is finitely generated.

As a result, we can easily see, that

$$E(\mathbb{Q}) \cong \mathbb{Z}^r \oplus E(\mathbb{Q})_{tors}$$

with r some integer, called the rank of the curve and $E(\mathbb{Q})_{tors}$ a finite group, called torsion subgroup.

Up to now, it is really a problem to determine the rank of an elliptic curve. In general only lower bounds for the rank are found, and even these are extremely hard to derive. Indeed there is a hunt going on nowadays, to find elliptic curves with higher and higher ranks, or rather with higher and higher lower bounds for their ranks.

There is much more known about the torsion subgroup $E(\mathbb{Q})_{tors}$.

The Nagell/Lutz theorem for instance shows, that if $a_1, a_2, a_3 = 0$ and $a_4, a_6 \in \mathbb{Z}$ (which can always be achieved over $\mathbb{Q}$) then every point $P = (x, y) \in E(\mathbb{Q})_{tors}$ which is not at infinity, has integer coordinates, and either $y = 0$ or $y^2 | 16(4a_4^3 + 27a_6^2)$

Furthermore, it is also known, what structure these finitely many points of finite order have.

Mazur's theorem states, if $a_1, a_2, a_3 = 0$ (which again can always be achieved over $\mathbb{Q}$) then $E(\mathbb{Q})_{tors}$ is one of the following groups:

(i) $\mathbb{Z}/n\mathbb{Z}$ with $1 \leq n \leq 10 \vee n = 12$
(ii) $\mathbb{Z}/2n\mathbb{Z} \times \mathbb{Z}/2\mathbb{Z}$ with $1 \leq n \leq 4$

Those who wish to know more details, or more about elliptic curves in general will find lots of usable and interesting information in [10] and [11].

2 Why Elliptic Curves are of Interest

Elliptic curves have found applications mainly in number theory. The most interesting ones surely are public-key cryptography, integer factorization and primality proving.

The motivation for using elliptic curves when it comes to cryptography is, that the discrete logarithm problem for elliptic curves is harder to solve than the discrete logarithm problem for finite fields with about the same size of numbers.

The term discrete logarithm is somehow misleading here, since we write our group additively. In general, the discrete logarithm problem is, that given x and $y = x^m$ one should find m in reasonable time. So the discrete logarithm problem for elliptic curves is to find an integer m in reasonable time by only knowing two points P and $Q = mP$.

How can one design a public-key cryptography system based on the hardness of the discrete logarithm problem for elliptic curves? Lets assume, that Alice and Bob want to exchange information securely. But all of the communication is eavesdropped by Eve. Now Alice and Bob agree on an elliptic curve to use as basis for the encoding, and a way to transform the message into a point on the curve. Then they also choose a point P on the curve. Of course now Eve also knows the elliptic curve, the way to transform the message into a point, and the point P.

Now Alice secretly chooses a sufficiently large integer k_A and tells nobody. Just as Bob, who also chooses a sufficiently large integer k_B in complete privacy. Next Alice sends $k_A P$ to Bob. Bob accordingly sends $k_B P$ to Alice. So now Eve knows P, $k_A P$, and $k_B P$.

And here comes the crucial step. Alice knows k_A, and $k_B P$ (she received $k_B P$ from Bob) and can easily derive $Q := k_A k_B P$ as $k_A(k_B P)$. Bob on the other hand knows k_B, and $k_A P$ (which he got from Alice). Therefore he can also derive Q as $k_B(k_A P)$ right away.

Eve on the other hand knows P, $k_A P$, and $k_B P$, as mentioned above. She cannot derive Q, since she has not got k_A, k_B nor $k_A k_B$. And she cannot get one of these three integers, because she cannot solve the discrete logarithm problem in time. (Which would be determining k_A by knowing P, and $k_A P$ or respectively getting k_B by only having P, and $k_B P$).

So now Alice and Bob have agreed on a secret (the point Q), which Eve does not know. And she has no chance of revealing this secret within reasonable time.

More aspects of cryptography with the help of elliptic curves can be found in [6]

A small example of (de)ciphering using elliptic curves can be found in the examples section (Section 5.4)

But as mentioned above, there are more applications of elliptic curves.

In 1985 factoring integers with the help of elliptic curves was proposed by H.W. Lenstra Jr. and two years later published in [7]. But since this is not so straightforward as public-key cryptography, it is only mentioned that detailed information about this can be found in

[3], [4], or [9].

A good treatment of elliptic curve primality proving can be found in [1] and [8].

3 Objectives of the Implementation

The main objective was, to give the user all the freedom `Maple` and `CASA` provide, and bring it to the domain of elliptic curves. No unnecessary limitations should be put on the user, and the user should be able to interact with the system in a consistent, convenient, and well-known way.

- integration of the implementation into the existing `Maple` and `CASA` environment
- representation of elliptic curves
- generating elliptic curves via the parameters a_1, a_2, a_3, a_4, a_6
- generating elliptic curves via points, which should be on the curve
- generating elliptic curves with an origin $\mathcal{O}$ different to $(0:1:0)$
- generating elliptic curves over finite fields
- doing arithmetic on elliptic curves
- checking if points are on the curve
- direct retrieval of all the necessary parameters of the curve
- generating points on an elliptic curve
- generating animations to illustrate elliptic curves

4 Achievements

The implementation allows the user to generate elliptic curves via the function `mkEllCurve(res,field,origin)`.

`res` is a list of restrictions on the coefficients a_1, a_2, a_3, a_4, a_6 along with points, which should be on the curve.

`field` determines the underlying field of the curve. This optional argument has a default value of 0 to indicate, that the field should be the rational numbers. If a list with a prime and a positive integer is specified, than the curve is taken over a finite field with these parameters.

`origin` is the origin of the curve, which is at $(0:1:0)$, if no other point is supplied.

The result of the `mkEllCurve` is an `EllipticCurve` object, on which the following methods can be invoked:

- `["isDetermined"]()`
  yields true, if there are no free variables in the curve's parameters. In other words, this is true, if you are dealing with a single curve, not a class of curves.
- `["isReal"]()`
  yields true, if all the parameters of the curve are in the domain of the real numbers.
- `["isRational"]()`
  yields true, if all the parameters of the curve are rationals.
- `["isFinite"]()`
  yields true, if the field of the elliptic curve is a finite field.
- `["field"]()` (only defined over finite fields)
  gives you the field over which the curve is defined.
- `["fieldChar"]()`
  results in the characteristic of the field of elliptic curve
- `["fieldPower"]()`
  tells, to which power the characteristic has to be taken, to get the number of field elements.
- `["equation"](x, y)`
  gives the defining equation of the elliptic curve in the variables `x` and `y`. Both arguments are optional.
- `["algebraicSet"](x, y)` (not defined for finite fields)
  gives a `CASA` algebraic set, which can be used for further processing in the `CASA` environment. The optional arguments `x` and `y` are the variables for the representation of the algebraic set.
- `["parameters"]()`
  returns the list of parameters of the elliptic curve of the form a_1, a_2, a_3, a_4, a_6.
- `["discriminant"]()`
  yields the discriminant of the curve.
- `["jInvariant"]()`
  returns the j-invariant of the curve.
- `["allParameters"]()`
  gives all parameters of the elliptic curve as list of the form a_1, a_2, $a_3, a_4, a_6, b_2, b_4, b_6, b_8, c_4, c_6, \Delta$, j-Invariant.
- `["origin"]()`
  shows the origin $\mathcal{O}$ of the curve.

- `["*"](P, Q)`

  If `P` is a point on the curve, then the third intersection point between the curve and the line defined by `P`, and `Q` is the result. If `P` is an integer, then `Q+Q+...+Q` (`P` terms) is returned.

- `["+"](P,Q)`

  is the addition of the two points `P`, and `Q` on the curve.

- `["-"](P)`

  returns the inverse of the point `P`.

- `["onCurve"](P)`

  yields true, if `P` is on the curve.

- `["findPoints"](height,N,verbose)` (over the rationals)

  `["findPoints"](N,verbose)` (over finite fields)

  `height` is the maximal height, to search points for, where the height of a point $P = (\frac{t}{u}, \frac{v}{w})$ is $h(P) := max(|t|, |u|)$.

  `N` is the number of points to search for. If this argument is missing, then the functions searches for all points (up to a certain height for the rationals).

  `verbose` can be set to `true`. The the points are displayed as soon as they are found. This is extremely useful, if lots of points are searched for, and some points are needed right away. If this argument is missing, then there is no intermediate output.

- `["order"](P)` (over the rationals)

  gives the order of `P`.

- `["torsionSubgroup"]()` (over the rationals)

  calculates all the points of finite order, and puts them in a list, along with their orders.

- `["clone"](res)`

  allows to specify restrictions on the class of elliptic curves, or force points to be on the curve by fixing free parameters of a_1, a_2, a_3, a_4, a_6. `res` can be a simple equation or point, or a list of points and equations.

  The result is a new curve from the class of curves, which also fulfills the new restrictions `res`, leaving the current curve unaltered.

Besides the function `mkEllCurve` with the previously listed possibilities, there is also a function `animateEllCurve`. It is used to illustrate and animate elliptic curves and operations on them respectively. The supported operations for single curves are the group law, inversion, and finding the third intersection point. For classes of curves, there is additionally the accumulation, which can be visualized. There is no need to specify bounds, since they are calculated automatically. Furthermore all points in the animations can be labeled, by embedding them in a list along with the text, for the label.

So an animation is generated using
`animateEllCurve(anims, static, frames, borders)`

- `anims`

  is a list of operations to animate. The operations themselves are again lists, which can be of the form

  ```
  [curve,"+",point1,point2,resultdesc,timesdesc,
    origindesc]
  [curve,"*",point1,point2,resultdesc]
  [curve,"-",point1,resultdesc,origindesc]
  ```

  where `curve` is the curve for the operation. `point1`, and `point2` are the points for the operations. `resultdesc`, `timesdesc`, and `origindesc` are optional labels for points which occur during the calculation.

  In order to animate classes of curves, or operations on them, one chooses a parameter for the curve. Values from a specified interval are assigned to this parameter, and the generated class members are plotted. These operations are specified as lists of the form

  ```
  [curve,variable=minval..maxval]
  [curve,variable=minval..maxval,"+",point1,point2,
    resultdesc,timesdesc,origindesc]
  [curve,variable=minval..maxval,"*",point1,point2,
    resultdesc]
  [curve,variable=minval..maxval,"*",n,point1,
    resultdesc]
  [curve,variable=minval..maxval,"-",point1,resultdesc,
    origindesc]
  ```

  where `variable` is the variable which is assigned values from `minval` to `maxval`. `n` is an integer. The other parameters are exactly as those explained above.

- `static`

  is a list of static elements for the animation. These objects can be points, or elliptic curves, and they will be drawn throughout the whole animation.

- `frames`

  is the number of frames to draw per operation.

- `borders`
  is of the form `[[left,top], [right, bottom]]` and defines a rectangle in the affine plane, which should be visible. These are extended automatically.

5 Usage by Examples

5.1 Curve over $\mathbb{Q}$ with $\mathcal{O} = (0 : 1 : 0)$

Generate a curve, by 5 points

```
> curve:=mkEllCurve([[1,2],[1,-3],[5,16],[5,0],[2,2]]);
curve := EllipticCurve
```

Now let's see what curve we have received

```
> curve["parameters"]();
> curve["AlgebraicSet"]();
[-17/4, -17/3, 21/4, 3/2, 55/6]
Implicit_Algebraic_Set([-y^2 + 17/4 x y - 21/4 y + x^3 -
  - 17/3 x^2  + 3/2 x + 55/6],[x, y])
```

Now we want to check, whether some points are on the curve.

```
> curve["onCurve"]([5,16]);
true
> curve["onCurve"]([10,32,2]);
true
> curve["onCurve"]([5,13]);
false
```

`[5,16]` obviously is, since we defined our curve to pass through it.

`[10,32,2]` is a different projective representation of the affine point $(5, 16)$ and should therefore also be on the curve. So in any case, where a point on the curve is expected, a point in projective space can be entered, and will be automatically converted into affine space.

`[5,13]` cannot be on the curve, since then there would be three points, having an x-coordinate of 5 (see generation of the curve)

And now for the operations.

Start with some simple and nice addition

```
> curve["+"]([1,2],[5,16])["coordinates"];
[-71/24, -191/32]
```

Of course also $(0 : 1 : 0)$ is possible wherever the implementation expects a point on the curve

```
> curve["+"]([1,2],[0,1,0])["coordinates"];
[1, 2]
> curve["*"]([1,2],[0,1,0])["coordinates"];
[1, -3]
```

And finally show, that $P+P$ is the same as $2P$

```
> curve["+"]([1,2],[1,2])["coordinates"];
> curve["*"](2,[1,2])["coordinates"];
[85/36, 1007/432]
[85/36, 1007/432]
```

5.2 Curve over $\mathbb{Q}$ with $\mathcal{O} \neq (0:1:0)$

```
> curve:=mkEllCurve([a1=a3,a3=0,[5,16],[2,2],[-1,0]],0,
    [15,48,3]):
> curve["origin"]();
[5, 16]
```

So here we have created a curve, which is symmetric with respect to the x-axis (since $a_1 = a_3 = 0$). Furthermore the origin of the curve is in $(15:48:3)$ which is $(5,16)$ in the affine representation.

```
> curve["-"]([-1,0]);
> curve["+"](%,[-1,0]);
[-6473/841, -572704/73167]
[5, 16]
```

As we can see here, a negated point now usually has a different x-coordinate than the original point. So the change of the origin has tremendous impact on addition and negation

```
> curve["-"]([0,1,0]);
> curve["+"](%,[0,1,0]);
[585/256, -47183/12288]
[5, 16]
```

And now of course also the inverse of $(0:1:0)$ is no longer $(0:1:0)$ itself, but a point on the affine plane.

5.3 Curve over $\mathbb{F}_{3^4}$ with $\mathcal{O} \neq (0:1:0)$

```
> curve:=mkEllCurve([a1=a3,a3=0,[12,0],[2,4],a6=4],
    [3,4],[44, 54]):
> curve["AllParameters"]();
[0, 34, 0, 48, 4, 34, 69, 4, 55, 64, 31]
```

Having generated an elliptic curve over $\mathbb{F}_{3^4}$ with origin at $(44, 54)$, we can retrieve all the necessary parameters for most algorithms. So the first five values are the a_1, a_2, a_3, a_4, a_6, followed by b_2, b_4, b_6, b_8, and finally c_4, c_6, the discriminant and j-invariant.

As can be observed easily, the arithmetic on an elliptic curve over finite fields works just like in the case of the rationals. There is only one difference, namely that the numbers now correspond to elements in a finite field.

```
> curve["-"]([19,50]);
> curve["+"](%,[19,50]);
[34, 20]
[44, 54]

> curve["*"]([63,14],[19,50]);
[19, 50]

> curve["*"](10241,[63,14]);
> curve["*"](10240,[63,14]);
> curve["+"](%,[63,14]);
[63, 14]
[44, 54]
[63, 14]
```

5.4 Cipher/De-Cipher Example

First of all Alice and Bob choose an elliptic curve and a point P on this curve, which also gets known to Eve, the eavesdropper

```
> curve:=mkEllCurve([[171,121],[123,4],[31984,234],
    [1025,1025],a1=3],[2,15]):
> P:=[20438, 24505];
P := [20438, 24505]
```

Then Alice chooses k_A and sends $k_A P$ to Bob

```
> kA:=123;
> kAtimesP:=curve["*"](kA,P);
kA := 123
kAtimesP := [31185, 30162]
```

Then Bob chooses k_B and sends $k_B P$ to Bob

```
> kB:=79;
> kBtimesP:=curve["*"](kB,P);
kB := 79
kBtimesP := [7448, 6253]
```

Alice now knows k_A and $k_B P$, so she can derive the secret key $k_A k_B P$.

```
> curve["*"](kA,kBtimesP);
[9159, 29267]
```

Bob on the other hand knows k_B and $k_A P$ and can also get $k_A k_B P$.

```
> curve["*"](kB,kAtimesP);
[9159, 29267]
```

But Eve in turn only knows $P, k_A P, k_B P$ and she cannot obtain the secret key $k_A k_B P$ in reasonable time, due to the hardness of the discrete logarithm problem of elliptic curves.

5.5 Curve with Free Parameters

In [5] a discussion of elliptic curves of the form $y^2 = x^3 - x + m^2$ can be found. Since this class of curves has some nice properties, as can be seen there, it was chosen as an example for curves with free parameters. Here the class of curves has a degree of freedom 1, but of course you can also use higher ones.

```
> P:=[0,m]:
> Q:=[-1,m]:
> curveclass:=mkEllCurve([a1=0,a3=0,a2=0,a4=-1,a6=m*m]):
> curveclass["parameters"]();
[0, 0, 0, -1, m^2]
```

So now the class of curves has been created. P, and Q should be on every curve of this class. But $(0, 1)$ for example is definitely not on every curve of this class. So the `onCurve` yields `false`. There are curves in this class, where $(0, 1)$ is on the curve, as seen right below.

```
> curveclass["onCurve"](P);
true
> curveclass["onCurve"](Q);
true
> curveclass["onCurve"]([0,1]);
false

> curve:=curveclass["clone"]([m=1]):
> curve["onCurve"]([0,1]);
true
```

And of course, also arithmetic can be done, over such classes of curves.

```
> PpQ:=curveclass["+"](P,Q);
PpQ := [1, -m]
```

This for example tells you, that regardless of what value m has, $P+Q$ always is $(1, m)$

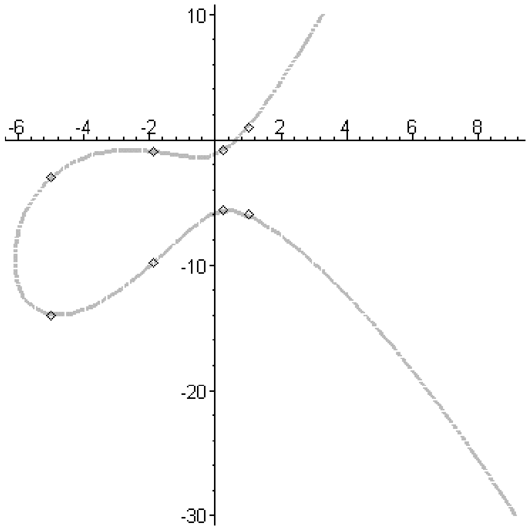
```
> Pp2Q:=expand(curveclass["+"](PpQ,Q));
Pp2Q := [m^2 , m^3]

> expand(curveclass["*"](3,P));
[-8 m^2 + 64 m^6 , 3 m - 96 m^5  + 512 m^9 ]
```

Now we know, that on this class of curves $P+2Q$ and $3P$ are always integer points. As mentioned above, more revealing facts on this class of curves, and ranks of the class-members can be found in [5].

5.6 Finding Points

If we want to find points on a curve over the rationals, then we have to specify a maximum height, up to which points should be searched for.

```
> curve:=mkEllCurve([a1=-2, a2=5, a3=7, a4=7, a6=-7]):
```



rational points of height ≤ 20 on
$y^2 - 2xy + 7y = x^3 + 5x^2 + 7x - 7$

```
> curve["findPoints"](2);
{[1, -6], [1, 1]}

> curve["findPoints"](20);
{[1/4, -7/8], [1/4, -45/8], [-5, -3], [-5, -14], [1, -6],
  [1, 1], [-17/9, -25/27], [-17/9, -266/27]}
```

We can also tell the system to stop, if sufficiently many points have been found. In this example we only search for 2 points on the curve, with a height of at most 20.

```
> curve["findPoints"](20,2);
{[1, -6], [1, 1]}
```

Since searching for points usually is quite time-taking, there is also the possibility to display intermediate results. So the next example searches for all (second parameter is zero) points on the curve with a height of at most 20 while displaying them as soon as they are found.

```
> curve["findPoints"](20,0,true);
{[1, -6], [1, 1]}
{[-5, -3], [-5, -14]}
{[1/4, -7/8], [1/4, -45/8]}
{[-17/9, -25/27], [-17/9, -266/27]}
{[1/4, -7/8], [1/4, -45/8], [-5, -3], [-5, -14], [1, -6],
  [1, 1],[-17/9, -25/27], [-17/9, -266/27]}
```

And of course the same also works over finite fields. But there is no height argument.

```
> curve:=mkEllCurve([a1=-2, a2=5, a3=7, a4=7, a6=-7],
>  [2,4]):
> curve["findPoints"]();
{[2, 1], [2, 2], [12, 3], [4, 6], [4, 9], [3, 3], [3, 2],
  [7, 9], [7, 0], [11, 13], [11, 3], [9, 7], [9, 13],
  [5, 2], [5, 15],[6, 13], [6, 6]}

> curve["findPoints"](2);
{[12, 3], [4, 6]}

> curve["findPoints"](0,false);
{[2, 1], [2, 2], [12, 3], [4, 6], [4, 9], [3, 3], [3, 2],
  [7, 9], [7, 0], [11, 13], [11, 3], [9, 7], [9, 13],
  [5, 2], [5, 15], [6, 13], [6, 6]}

> curve["findPoints"](0,true);
{[12, 3]}
{[4, 6], [4, 9]}
{[3, 3], [3, 2]}
{[7, 9], [7, 0]}
{[11, 13], [11, 3]}
{[9, 7], [9, 13]}
{[5, 2], [5, 15]}
{[2, 1], [2, 2]}
```

```
{[6, 13], [6, 6]}
{[2, 1], [2, 2], [12, 3], [4, 6], [4, 9], [3, 3], [3, 2],
  [7, 9],[7, 0], [11, 13], [11, 3], [9, 7], [9, 13],
  [5, 2], [5, 15], [6, 13], [6, 6]}
```

5.7 Torsion Subgroup and Order

Over the rationals, one can easily get the order of a point with the help of the `order` function.

```
> curve:=mkEllCurve([a1=7,a3=0,a2=0,a4=16,a6=0]);
> curve["order"]([4,4]);
> curve["order"]([0,1,0]);
> curve["order"]([-2,10]);
4
1
8
```

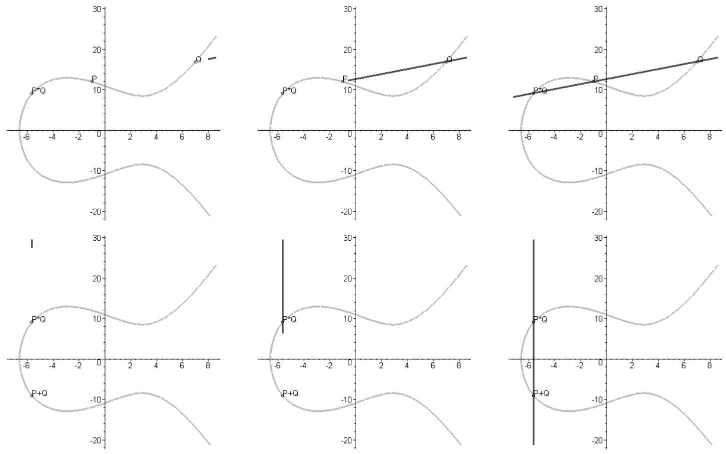
And also the torsion subgroup can be derived right away, with the help of `torsionSubgroup`

```
> curve["TorsionSubgroup"]();
{[[-2, 4], 8], [[0, 0], 2], [[4, 4], 4], [[4, -32], 4],
  [[-8, 40], 8], [[-8, 16], 8], [[0, 1, 0], 1],
  [[-2, 10], 8]}
```

5.8 Animation of Operations

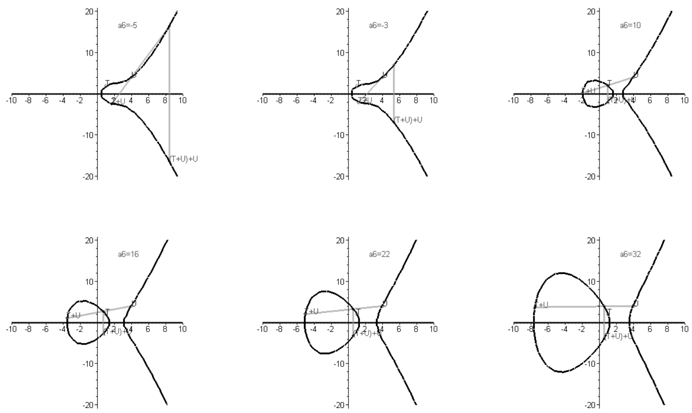
When animating operations on fixed curves, then the derivation of the third intersection point is animated

```
> curve:=mkEllCurve([a1=a3,a3=0,a2=0,a4=-25,a6=120]):
> P:=[-1,12]:
> Q:=curve["*"](6,P):
> animateEllCurve([[curve,"+",[P,"P"],[Q,"Q"],"P+Q","P*Q"]]
>    ,[curve]);
```



The situation changes, when animations are rendered on a class of curves. Then the results of the operations are derived right away, and showed on curves from the specified interval

```
> T:=[1,2]:
> U:=[4,4]:
> curve:=mkEllCurve([T,U,a1=0,a3=0]):
> TpU:=curve["+"](T,U):
> animateEllCurve([[curve,a6=-5..32,"+",[TpU,"T+U"],U,
>   "(T+U)+U"]] ,[[T,"T"],[U,"U"]],38,[[-10,-20],[14,20]]);
```



References

[1] A. O. L. Atkin and F. Morain, Elliptic Curves and Primality Proving, Mathematics of Computation 61 (1993), 29–68

[2] I. Blake, G. Seroussi and N. Smart, Elliptic Curves in Cryptography, Cambridge University Press (1999)

[3] R. P. Brent, Factorization of the tenth Fermat number, Mathematics of Computation 68 (1999), 429–451

[4] R. P. Brent, Some integer factorization algorithms using elliptic curves, Australian Computer Science Communications 8 (1986), 149–163

[5] E. Brown and B. T. Myers, Elliptic Curves from Mordell to Diophantus and Back, American Mathematical Monthly 109 (2002), 639–649

[6] N. Koblitz, Algebraic Aspects of Cryptography, Springer (1998)

[7] H. W. Lenstra, Jr., Factoring Integers with Elliptic Curves, Annaly of Mathematics 2 (1987), 649–673

[8] F. Morain, Primality Proving Using Elliptic Curves: An Update, Lecture Notes in Computer Science 1423 (1998), 111–127

[9] P. L. Montgomery, Speeding the Pollard and Elliptic Curve methods of factorization, Mathematics of Computation 48 (1987), 243–264

[10] H. J. Silverman, The Arithmetic of Elliptic Curves, Springer (1986)

[11] H. J. Silverman and J. Tate, Rational Points on Elliptic Curves, Springer (1992)