Transforming a general cubic elliptic curve equation to Weierstrass form

A Sage implementation

Niels Duif

Technische Universiteit Eindhoven

May 2, 2011

Technische Universiteit
Eindhoven
University of Technology

Contents

1 Introduction

An elliptic curve $\mathcal{E}$ is often given in its affine Weierstrass form:

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6. \tag{1}$$

In this report, the more convenient projective variant is used:

$$Y^2Z + a_1XYZ + a_3YZ^2 = X^3 + a_2X^2Z + a_4XZ^2 + a_6Z^3. \tag{2}$$

The map $(x, y) \to (x : y : 1)$ is an embedding of the affine curve from Equation 1 in the projective Equation 2. Its inverse is the projection $(X : Y : Z) \to (\frac{X}{Z}, \frac{Y}{Z})$, which is defined for $Z \neq 0$.

Alternatively, an elliptic curve may be defined by various other polynomial equations. This report studies those general homogeneous cubic equations

$$F(X : Y : Z) = aX^3+bX^2Y+cXY^2+dY^3+eX^2Z+fXYZ+gY^2Z+hXZ^2+iYZ^2+jZ^3, \tag{3}$$

for which $F = 0$ defines an elliptic curve $\mathcal{E}$. As above, an affine equation is found by setting $x = \frac{X}{Z}$, and $y = \frac{Y}{Z}$. [1] describes how to transform Equation 3 to the Weierstrass form of Equation 2, using a point P on the elliptic curve $\mathcal{E}$. This report explains how this transformation works, and presents an implementation in Sage [2], a free open-source mathematics software system.

An elliptic curve $\mathcal{E}$ has a shorter Weierstrass form than that of Equation 2. In this short form, one or more of the constants $a_1, \ldots, a_6$ is zero. The short form depends on the field over which $\mathcal{E}$ is defined. In Sage, the long Weierstrass form is sufficient to get an elliptic curve object `E`. The short form can be obtained by executing `E.minimal_form()`.

2 Transforming cubic to Weierstrass

Let F be as in Equation 3. If $F = 0$ defines an elliptic curve $\mathcal{E}$, then we can find a birational equivalence from $\mathcal{E}$ to a curve $\mathcal{E}'$, where $\mathcal{E}'$ has the long Weierstrass form of Equation 2. The birational equivalence consists of a rational transformation $\phi : \mathcal{E} \to \mathcal{E}'$, and its rational inverse $\psi : \mathcal{E}' \to \mathcal{E}$. These maps need not be defined for all points on $\mathcal{E}$ and $\mathcal{E}'$, but they should be each others inverses when defined, and they should preserve the group structure of the elliptic curve.

This section describes how to find a birational equivalence from Equation 3 to Equation 2, using a point $P = (P_X : P_Y : P_Z)$ on $\mathcal{E}$. An elliptic curve is a smooth projective curve with at least 1 point, so P exists. We need the tangent line $\mathbf{l}$ at P, which exists because $\mathcal{E}$ is smooth. The equation of the tangent line at P is given by

$$\mathbf{l} : \frac{\partial F}{\partial X}(P)(X - P_X) + \frac{\partial F}{\partial Y}(P)(Y - P_Y) + \frac{\partial F}{\partial Z}(P)(Z - P_Z) = 0. \tag{4}$$

By Bézout's Theorem [3], $\mathbf{l}$ intersects $\mathcal{E}$ with multiplicity 3. $\mathbf{l}$ is the tangent line to $\mathcal{E}$ at P, so there are two cases. The first case is when $\mathbf{l}$ intersects $\mathcal{E}$ at P with multiplicity 3. In that case P is called a *flex*. This case is described in Section 2.1. The second case is when P is not a flex. Then the tangent line $\mathbf{l}$ intersects $\mathcal{E}$ at P with multiplicity 2, and in another point Q with multiplicity 1.

2.1 Rational flex

The elliptic curve $\mathcal{E}$ is defined by the cubic of Equation 3, and the point P is a flex. That means that the tangent $\mathbf{l}$ at P intersects $\mathcal{E}$ in P with multiplicity 3. We will now find a birational equivalence between $\mathcal{E}$ and a Weierstrass curve.

An elliptic curve in the Weierstrass form of Equation 2 has a flex $\mathcal{O} = (0 : 1 : 0)$. For this point the tangent line is $Z = 0$. Substituting this into Equation 2 gives $X^3 = 0$, which shows that $\mathbf{l}$ indeed intersects $\mathcal{E}$ in P with multiplicity 3.

Define the linear transformation α by mapping P to $\mathcal{O} = (0 : 1 : 0)$, and $\mathbf{l}$ to the line $Z = 0$. To map $\mathbf{l}$ to the line $Z = 0$, we use a point $Q \neq P$ on $\mathbf{l}$ but not on $\mathcal{E}$, and map Q to $(1 : 0 : 0)$. The unique line through $(0 : 1 : 0)$ and $(1 : 0 : 0)$ is $Z = 0$, so indeed this maps $\mathbf{l}$ to the line $Z = 0$. It is convenient to first define the inverse transformation $\alpha^{-1} = \beta$. β maps $(0 : 1 : 0)$ to P, and $(1 : 0 : 0)$ to Q. β can be defined by the matrix equation $\beta : R \to \mathbf{M}_\beta R$, where $\mathbf{M}_\beta$ is partially defined by

$$\mathbf{M}_\beta = \begin{pmatrix} Q_X & P_X & \cdot \\ Q_Y & P_Y & \cdot \\ Q_Z & P_Z & \cdot \end{pmatrix}. \tag{5}$$

$\mathbf{M}_\beta$ should be an invertible matrix. This leaves two degrees of freedom for the transformation β. An easy solution is taking the last column equal to $(1, 0, 0)^T$, $(0, 1, 0)^T$, or $(0, 0, 1)^T$, whichever produces an invertible matrix. Of course, this transformation only works if the vectors (P_X, P_Y, P_Z) and (Q_X, Q_Y, Q_Z) are linearly independent. This is always the case, because P and Q are different projective points.

The inverse of $\mathbf{M}_\beta$ is the transformation matrix $\mathbf{M}_\alpha$ that belongs to the map α. This transformation does not yet result in a Weierstrass equation. We apply $\alpha : (X : Y : Z) \to (U : V : T)$ to $\mathcal{E}$ to obtain the equivalent curve $\mathcal{C}$. $\mathcal{C}$ then has curve equation

$$G(U : V : T) = kU^3 + lU^2V + mUV^2 + nV^3 + pU^2T + qUVT + rV^2T + sUT^2 + tVT^2 + uT^3 = 0. \tag{6}$$

We now show that some terms of G are 0. $\mathcal{O} = (0 : 1 : 0)$ is on $\mathcal{C}$, so $G(\mathcal{O}) = n = 0$. The tangent at $\mathcal{O}$ is $T = 0$, so $\frac{\partial G}{\partial U}(P) = m = 0$, and $\frac{\partial G}{\partial T}(P) = r \neq 0$. Intersecting $\mathcal{C}$ with the line $T = 0$ then gives

$$U^2(kU + lV) = 0. \tag{7}$$

$\mathcal{O}$ is a flex, so Equation 7 must have $\mathcal{O}$ as a triple solution. This can only be if $l = 0$, and $k \neq 0$. So $\mathcal{C}$ has curve equation

$$kU^3 + pU^2T + qUVT + rV^2T + sUT^2 + tVT^2 + uT^3 = 0. \tag{8}$$

To obtain Equation 2 we require that $k = 1$, and $r = -1$, which is achieved through scaling: divide Equation 8 by k to get

$$U^3 + c_2U^2T + c_1UVT + c_0V^2T + c_4UT^2 + c_3VT^2 + c_6T^3 = 0, \tag{9}$$

then substitue $T = -\frac{W}{c_0}$ to get

$$V^2W + a_1UVW + a_3VW^2 = U^3 + a_2U^2W + a_4UW^2 + a_6W^3. \tag{10}$$

Finally, an affine equation may be found by substituting $W = 1$.

Example 2.1 *Let $\mathcal{E}$ be defined over $\mathbb{Q}$ by $X^3 + Y^3 + 60Z^3 = 0$. The point $P = (1 : -1 : 0)$ is on $\mathcal{E}$. The tangent $\mathbf{l}$ at P is computed by Equation 4:*

$$\mathbf{l} : X + Y = 0. \tag{11}$$

Substituting $Y = -X$ into $X^3 + Y^3 + 60Z^3 = 0$ gives $60Z^3 = 0$, so P is the only point on $\mathcal{E}$ that is also on $\mathbf{l}$, so P is a flex. For Q we take a different point on $\mathbf{l}$, in this case we use $Q = (1 : -1 : 1)$.

The matrix $\mathbf{M}_\beta = \begin{pmatrix} 1 & 1 & 1 \\ -1 & -1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$, is invertible, with inverse $\mathbf{M}_\alpha = \begin{pmatrix} 0 & 0 & 1 \\ 0 & -1 & -1 \\ 1 & 1 & 0 \end{pmatrix}$.

This gives the transformations $\alpha : (U, V, T) = \mathbf{M}_\alpha(X, Y, Z)^T = (Z, -Y - Z, X + Y)$, and $\beta : (X, Y, Z) = \mathbf{M}_\beta(U, V, T)^T = (U + V + T, -U - V, U)$. Applying the transformation β to $X^3 + Y^3 + 60Z^3 = 0$ gives the equation:

$$60U^3 + 3U^2T + 6UVT + 3V^2T + 3UT^2 + 3VT^2 + T^3 = 0. \tag{12}$$

Dividing by 60 yields

$$U^3 + \frac{1}{20}U^2T + \frac{1}{10}UVT + \frac{1}{20}V^2T + \frac{1}{20}UT^2 + \frac{1}{20}VT^2 + \frac{1}{60}T^3 = 0, \tag{13}$$

and finally, the substitution $T = -20W$ gives

$$V^2W + 2UVW - 20VW^2 = U^3 - U^2W + 20UW^2 - \frac{400}{3}W^3. \tag{14}$$

2.2 No rational flex

The elliptic curve $\mathcal{E}$ is defined by the cubic of Equation 3, and the point P is not a flex. That means that the tangent l at P intersects $\mathcal{E}$ in P with multiplicity 2. We will now find a birational equivalence between $\mathcal{E}$ and a Weierstrass curve.

The tangent $\mathbf{l}$ intersects $\mathcal{E}$ in 3 points by Bézout's Theorem [3]. Therefore, it intersects $\mathcal{E}$ in a point $Q \neq P$. We find this point by intersecting $\mathcal{E}$ with $\mathbf{l}$. This gives three solutions: P is a double solution, and Q is a single solution. From Q we repeat this process, to find a third point R. This method is called the *chord and tangent method*. The chord and tangent method is possible as long as Q is not a flex. If Q is a flex, we proceed as in Section 2.1. Note that $R \neq P$, because P cannot be on $\mathbf{m}$. (Suppose P is on $\mathbf{m}$, then $\mathbf{m} = \mathbf{l}$. So $\mathbf{m}$ is the tangent at both P and Q, so it intersects $\mathcal{E}$ with multiplicity at least 4, which cannot happen by Bézout's Theorem [3].) This also implies that P, Q, and R are not collinear, and therefore

$\det(\mathbf{M}_\gamma) = \det \begin{pmatrix} P_X & Q_X & R_X \\ P_Y & Q_Y & R_Y \\ P_Z & Q_Z & R_Z \end{pmatrix} \neq 0$, so $\mathbf{M}_\gamma$ is invertible.

$\mathbf{M}_\gamma$ defines a linear map. Its inverse is δ, where $\delta : P \to (1 : 0 : 0), Q \to (0 : 1 : 0), R \to (0 : 0 : 1)$. The corresponding matrix is $\mathbf{M}_\delta = {\mathbf{M}_\gamma}^{-1}$. After applying the map δ, we get a curve $\mathcal{C}$ through the points $(1 : 0 : 0)$, $(0 : 1 : 0)$, and $(0 : 0 : 1)$. The general equation of $\mathcal{C}$ is

$$G(U:V:W) = kU^3+lU^2V+mUV^2+nV^3+pU^2W+qUVW+rV^2W+sUW^2+tVW^2+uW^3 = 0. \tag{15}$$

$G((1:0:0)) = 0$, so $k = 0$. $G((0:1:0)) = 0$, so $n = 0$. $G((0:0:1)) = 0$, so $u = 0$. So $\mathcal{C}$ has curve equation

$$lU^2V + mUV^2 + pU^2W + qUVW + rV^2W + sUW^2 + tVW^2 = 0. \tag{16}$$

The tangent at $(1:0:0)$ goes through $(1:0:0)$ and $(0:1:0)$, so $l = 0$, and $p \neq 0$. The tangent at $(0:1:0)$ goes through $(0:1:0)$ and $(0:0:1)$, so $r = 0$, and $m \neq 0$. So the equation of $\mathcal{C}$ is

$$mUV^2 + pU^2W + qUVW + sUW^2 + tVW^2 = 0. \tag{17}$$

Equation 17 has fewer terms than the corresponding equation in Cremona's [1]. Cremona does not note that $u = 0$. His transformation also works for equations with $u \neq 0$.
Setting $(U:V:W) = (K^2 : LN : KN)$, and dividing by K^2N gives

$$vK^3 + wK^2N + xKLN + yL^2N + zLN^2. \tag{18}$$

The quadratic transformation $\mu : (K:L:N) \to (K^2 : LN : KN)$ is called a *Cremona transformation*. These transformations are named after L. Cremona, who studied them around 1863, and not after the author of [1], who has the same last name. The inverse transformation is found by setting $(K:L:N) = (UW : UV : W^2)$, and dividing by UW^2. The line $W = 0$ is mapped to the single point $(0:1:0)$. The transformation $\nu : (U:V:W) \to (UW : UV : W^2)$ is defined everywhere, except for $(U:V:W) = (1:0:0)$ and $(U:V:W) = (0:1:0)$. The transformation μ is defined everywhere, except for $(K:L:N) = (0:1:0)$ and $(K:L:N) = (0:0:1)$. From Equation 18 we divide by v to get

$$K^3 + c_2K^2N + c_1KLN + c_0L^2N + c_3LN^2 = 0, \tag{19}$$

and set $N = -\frac{M}{c_0}$ to get

$$L^2M + a_1KLM + a_3LM^2 = K^3 + a_2K^2M. \tag{20}$$

Again, substituting $M = 1$ gives an affine equation.

Example 2.2 *Let $\mathcal{E}$ be defined over $\mathbb{Q}$ by $X^3 + 7Y^3 + 64Z^3 = 0$. The point $P = (2:2:-1)$ is on $\mathcal{E}$. The tangent $\mathbf{l}$ at P is computed by using Equation 4 and dividing by 12:*

$$\mathbf{l} : X + 7Y + 16Z = 0. \tag{21}$$

Setting $Z = 1$ and substituting $X = -16 - 7Y$ into $X^3 + 7Y^3 + 64Z^3 = 0$ gives $Y = -2$ with multiplicity 2, and $Y = -3$ with multiplicity 1. So P is not a flex, and we find another point of intersection of $\mathbf{l}$ and $\mathcal{E}$. This point is $Q = (5:-3:1)$. The tangent at Q is

$$\mathbf{m} : 25X + 63Y + 64Z = 0. \tag{22}$$

Setting $Z = 1$, and substituting $X = \frac{64-63Y}{25}$ gives $Y = -3$ or $Y = \frac{183}{314}$. Scaling gives the point $R = (-1265 : 183 : 314)$. The three points P, Q, and R define the matrix

$\mathbf{M}_\gamma = \begin{pmatrix} 2 & 5 & -1265 \\ 2 & -3 & 183 \\ -1 & 1 & 314 \end{pmatrix}$, *which has inverse* $\mathbf{M}_\delta = \frac{1}{5040}\begin{pmatrix} 1125 & 2835 & 2880 \\ 811 & 637 & 2896 \\ 1 & 7 & 16 \end{pmatrix}$.

This gives the transformations $\gamma : (X, Y, Z) \to (2U + 5V - 1265W, 2U - 3V + 183W, -U + V + 314W)$*, and* $\delta : (U, V, W) \to \frac{1}{5040}(1125X + 2835Y + 2880Z, 811X + 637Y + 2896Z, X + 7Y + 16Z)$*. Applying* γ *to* $X^3 + 7Y^3 + 64Z^3 = 0$*, then dividing by* 336 *results in*

$$UV^2 + 180U^2W - 722UVW - 23579UW^2 + 121500VW^2. \tag{23}$$

Mapping $(U : V : W) = (K^2 : LN : KN)$ *and dividing by* K^2N *gives*

$$180K^3 - 23579K^2N - 722KLN + L^2N + 121500LN^2. \tag{24}$$

We divide Equation 24 by 180 *to get*

$$K^3 - \frac{23579}{180}K^2N - \frac{361}{90}KLN + \frac{1}{180}L^2N + 675LN^2, \tag{25}$$

and finally, the substitution $N = \frac{-M}{675}$ *gives*

$$L^2M - 722KLM - 21870000LM^2 = K^3 + 23579K^2M. \tag{26}$$

3 Sage code

The Sage implementation follows the approach described in the previous section. The Sage code consists of five parts:

1. A documentation string with several detailed examples. This string starts and ends with `"""`.

2. The main part of the code, which branches into the approach of Section 2.1 and Section 2.2. The main part of the code uses the two auxiliary methods defined below. It starts with `if algorithm == 'sage':`.

3. A part for backwards compatibility, starting with `elif (algorithm == 'magma'):`. This part calls Magma and returns the Weierstrass form of the elliptic curve.

4. The chord and tangent method, starting with `def chord_and_tangent(F, P):`. This method uses a point P on the elliptic curve $\mathcal{E}$ to find more points on $\mathcal{E}$. $\mathcal{E}$ is defined by $F = 0$. The chord and tangent method is described in Section 2.2. If P is a flex, the method returns P.

5. A check for equivalence, starting with `def are_projectively_equivalent(P, Q):`. This method checks whether two points in a projective space are the same.

```
def EllipticCurve_from_cubic(F, P, algorithm = 'sage', equivalence = False):
    r"""
    Construct an elliptic curve from a ternary cubic with a rational point.
```

```
INPUT:

- ``F`` -- a homogeneous cubic in three variables with rational
  coefficients, as a polynomial ring element, defining a smooth
  plane cubic curve.

- ``P`` -- a 3-tuple `(x,y,z)` defining a projective point on the
  curve `F=0`.

- ``algorithm`` -- an optional string (either 'sage' or 'magma')
  that specifies whether to use Magma or the built in Sage method.
  If not specified the default 'sage' is used.

- ``equivalence`` -- an optional boolean (True or False) that
  specifies whether a birational equivalence between F and its
  Weierstrass form should be given. The default is False. This
  only works if the algorithm is 'sage'.

OUTPUT:

(elliptic curve) An elliptic curve (in Weierstrass form)
isomorphic to the curve `F=0`.

If the algorithm 'magma' is used, the ouput is a short Weierstrass
equation, but no birational equivalence is given. If the algorithm
'sage' is used, the output is a long Weierstrass equation. If also
equivalence is True, then a birational equivalence between F and
the Weierstrass curve is printed.

For a more general version, see the function
``EllipticCurve_from_plane_curve()``.

EXAMPLES:

First we find that the Fermat cubic is isomorphic to the curve
with Cremona label 27a1::

sage: var("x y z")
(x, y, z)
sage: R.<x,y,z>=QQ[]
sage: f=R(x^3+y^3+z^3)
sage: P=[1,-1,0]
sage: E=EllipticCurve_from_cubic(f,P)
sage: E
Elliptic Curve defined by y^2 + 2*x*y - 1/3*y = x^3 - x^2 + 1/3*x
- 1/27 over Rational Field
sage: E.cremona_label()
```

```
'27a1'
```

Next we find the minimal model and conductor of the Jacobian of the Selmer curve::

```
sage: var("x y z")
(x, y, z)
sage: R.<x,y,z>=QQ[]
sage: f=R(x^3+y^3+60*z^3)
sage: P=[1,-1,0]
sage: E=EllipticCurve_from_cubic(f,P)
sage: E
Elliptic Curve defined by y^2 + 2*x*y - 20*y = x^3 - x^2 + 20*x - 400/3
over Rational Field
sage: E.minimal_model()
Elliptic Curve defined by y^2 = x^3 - 24300 over Rational Field
sage: E.conductor()
24300
```

We can also get the birational equivalence to and from the Weierstrass form::

```
sage: E = EllipticCurve_from_cubic(f,P,equivalence=True)
Transformation:
x -> x + y - 20
y -> -x - y
z -> x
Then multiply the equation with 1/60
Inverse transformation:
Homogenize the curve equation with respect to z, then map
x -> z
y -> -y - z
z -> -1/20*x - 1/20*y
Then multiply the equation with 60
```

Using Magma gives the same results, but no birational maps::

```
sage: E = EllipticCurve_from_cubic(f, P, algorithm='magma')   # optional - magma
sage: E                # optional - magma
Elliptic Curve defined by y^2  = x^3 - 24300 over Rational Field
sage: E.conductor()    # optional - magma
24300
```

Not all cubics can be transformed to a Weierstrass equation by a linear transformation. The general birational transformation is quadratic::

```
sage: var("x y z")
```

```
(x, y, z)
sage: R.<x,y,z>=QQ[]
sage: f=R(x^3+7*y^3+64*z^3)
sage: P=[2,2,-1]
sage: E=EllipticCurve_from_cubic(f,P,equivalence=True)
Transformation:
x -> -157/45*x^2 - 1265/314*x + 5*y
y -> -157/45*x^2 + 183/314*x - 3*y
z -> 157/90*x^2 + x + y
Then multiply the equation with 15/8792/x^2
Inverse transformation:
Homogenize the curve equation with respect to z, then map
x -> 785/56448*x^2 + 2669/20160*x*y + 157/640*y^2 + 4553/17640*x*z
+ 2041/2520*y*z + 1256/2205*z^2
y -> 4055/112896*x^2 + 4787/40320*x*y + 91/1280*y^2 + 7769/35280*x*z
+ 1993/5040*y*z + 724/2205*z^2
z -> -3869893/571536000*x^2 - 3869893/40824000*x*y - 3869893/11664000*y^2
- 3869893/17860500*x*z - 3869893/2551500*y*z - 7739786/4465125*z^2
Then multiply the equation with 30240/157/(24649/28449792*x^3
+ 1454291/101606400*x^2*y + 1059907/14515200*x*y^2 + 24649/230400*y^3
+ 24649/823200*x^2*z + 468331/1587600*x*y*z + 271139/453600*y^2*z
+ 271139/926100*x*z^2 + 419033/396900*y*z^2 + 394384/694575*z^3)
sage: E
Elliptic Curve defined by y^2 - 361/157*x*y - 2733750/3869893*y
= x^3 + 23579/98596*x^2 over Rational Field
sage: E.minimal_model()
Elliptic Curve defined by y^2 + y = x^3 - 331 over Rational Field
"""

if algorithm == 'sage':

    # check the input
    vars = F.parent().gens()
    if len(vars) != 3:
        raise TypeError, '%s is not a polynomial in three variables'%F
    if not F.is_homogeneous():
        raise TypeError, '%s is not a homogeneous polynomial'%F
    x, y, z = vars
    if len(P) != 3:
        raise TypeError, '%s is not a projective point'%P
    K = F.parent().base_ring()
    R = rings.PolynomialRing(K, 'x,y,z')
    try:
        P = [K(c) for c in P]
    except TypeError:
        raise TypeError, 'cannot coerce %s into %s'%(P,K)
    if F(P) != 0:
```

```
    raise ValueError, '%s is not a point on %s'%(P,F)

# get two additional points on the curve with the chord and tangent method
P2 = chord_and_tangent(F, P)
P3 = chord_and_tangent(F, P2)
# if P2 = P3 then P2 is a flex, and we use a different approach
if (are_projectively_equivalent(P2, P3)):
    # find the tangent to F in P
    dx = F.derivative(x)
    dy = F.derivative(y)
    dz = F.derivative(z)
    dxP2 = dx(P2)
    dyP2 = dy(P2)
    dzP2 = dz(P2)
    # find a second point Q on the tangent line
    if (P2[0] != 0):
        Q0 = P[0];
        if (dy == 0):
            Q1 = P[1] + 1;
            Q2 = P[2];
        else:
            Q1 = P[1]-dz//dy;
            Q2 = P[2]+1;
    elif (P2[1] != 0):
        Q1 = P2[1];
        if (dx == 0):
            Q0 = P[0] + 1;
            Q2 = P[2];
        else:
            Q0 = P[0]-dz//dy;
            Q2 = P[2]+1;
    else:
        Q2 = P[2];
        if (dx == 0):
            Q0 = P[0] + 1;
            Q1 = P[1];
        else:
            Q0 = P[0]-dy//dx;
            Q1 = P[1] + 1;
    # send P to [0:1:0] and Q to [1:0:0]
    M = matrix.matrix([[Q0, Q1, Q2], P2, [1, 0, 0]]).transpose()
    # if M is singular, we change the first column
    if (not M.is_invertible()):
        M = matrix.matrix([[Q0, Q1, Q2], P2, [0, 1, 0]]).transpose()
    if (not M.is_invertible()):
        M = matrix.matrix([[Q0, Q1, Q2], P2, [0, 0, 1]]).transpose()
    F2 = R(M.act_on_polynomial(F))
```

```
            # scale and dehomogenise
            a = F2.coefficient(x**3)
            F3 = F2/a
            b = F3.coefficient(y*y*z)
            F4 = F3.substitute(z=-1/b)
            S = rings.PolynomialRing(K, 'x,y')
            E = EllipticCurve(S(F4))
            # if requested, compute the transformation
            if equivalence:
                trans_x = M[0,0]*x + M[0,1]*y + M[0,2]*-1/b
                trans_y = M[1,0]*x + M[1,1]*y + M[1,2]*-1/b
                trans_z = M[2,0]*x + M[2,1]*y + M[2,2]*-1/b
                print "Transformation:"
                print "%s -> %s"%(vars[0], trans_x)
                print "%s -> %s"%(vars[1], trans_y)
                print "%s -> %s"%(vars[2], trans_z)
                print "Then multiply the equation with %s"%(1/a)
                # compute the inverse transformation
                M = M.inverse()
                trans_x = M[0,0]*x + M[0,1]*y + M[0,2]*z
                trans_y = M[1,0]*x + M[1,1]*y + M[1,2]*z
                trans_z = M[2,0]*x + M[2,1]*y + M[2,2]*z
                print "Inverse transformation:"
                print "Homogenize the curve equation with respect to %s,
then map"%z
                print "%s -> %s"%(vars[0], trans_x)
                print "%s -> %s"%(vars[1], trans_y)
                print "%s -> %s"%(vars[2], -b*trans_z)
                print "Then multiply the equation with %s"%(a)

            return E

        # P and P2 are both not a flex, so P, P2, P3 are different
        # send P, P2, P3 to (1:0:0), (0:1:0), (0:0:1) respectively
        M = matrix.matrix([P, P2, P3]).transpose()
        F2 = M.act_on_polynomial(F)
        # substitute x = U^2, y = V*W, z = U*W, and rename (x,y,z)=(U,V,W)
        F3 = (F2.substitute(x = x**2, y = y*z, z = x*z))//(x**2*z)
        # scale and dehomogenise
        a = F3.coefficient(x**3)
        F4 = F3/a
        b = F4.coefficient(y*y*z)
        F5 = F4.substitute(z=-1/b)
        # change to a polynomial in only two variables
        S = rings.PolynomialRing(K, 'x,y')
        E = EllipticCurve(S(F5))
        # if requested, compute the transformation
```

```
        if equivalence:
            trans_x = (M[0,0]*x*x + M[0,1]*y*-1/b + M[0,2]*x*-1/b)*-b
            trans_y = (M[1,0]*x*x + M[1,1]*y*-1/b + M[1,2]*x*-1/b)*-b
            trans_z = (M[2,0]*x*x + M[2,1]*y*-1/b + M[2,2]*x*-1/b)*-b
            print "Transformation:"
            print "%s -> %s"%(vars[0], trans_x)
            print "%s -> %s"%(vars[1], trans_y)
            print "%s -> %s"%(vars[2], trans_z)
            print "Then multiply the equation with %s"%(1/(a*b**2)/(x**2))
            # compute the inverse transformation
            M = M.inverse()
            trans_x = (M[0,0]*x + M[0,1]*y + M[0,2]*z)
            trans_y = (M[1,0]*x + M[1,1]*y + M[1,2]*z)
            trans_z = (M[2,0]*x + M[2,1]*y + M[2,2]*z)
            print "Inverse transformation:"
            print "Homogenize the curve equation with respect to %s, then
map"%z
            print "%s -> %s"%(vars[0], trans_x*trans_z)
            print "%s -> %s"%(vars[1], trans_x*trans_y)
            print "%s -> %s"%(vars[2], -b*trans_z*trans_z)
            print "Then multiply the equation with %s"%(a/(trans_x*trans_z*
trans_z))

        return E

    elif (algorithm == 'magma'):
        from sage.interfaces.all import magma
        cmd = "P<%s,%s,%s> := ProjectivePlane(RationalField());"%SR(F).
variables()
        magma.eval(cmd)
        cmd = 'aInvariants(MinimalModel(EllipticCurve(Curve(Scheme(P, %s)),
P!%s)));'%(F, P)
        s = magma.eval(cmd)
        return EllipticCurve(rings.RationalField(), eval(s))

    else:
        raise ValueError, 'algorithm %s not supported; options are "sage"
(default) and "magma"'%algorithm

def chord_and_tangent(F, P):

    # check the input
    R = F.parent()
    K = R.base_ring()
    vars = R.gens()
    if len(vars) != 3:
        raise TypeError, '%s is not a polynomial in three variables'%F
```

```
if not F.is_homogeneous():
    raise TypeError, '%s is not a homogeneous polynomial'%F
x, y, z = vars
if len(P) != 3:
    raise TypeError, '%s is not a projective point'%P
try:
    P = [K(c) for c in P]
except TypeError:
    raise TypeError, 'cannot coerce %s into %s'%(P,K)
if F(P) != 0:
    raise ValueError, '%s is not a point on %s'%(P,F)

# find the tangent to F in P
dx = F.derivative(x)
dy = F.derivative(y)
dz = F.derivative(z)
dxP = dx(P)
dyP = dy(P)
dzP = dz(P)
# find the points on F where the derivative is 0
solutions = [[0,0,0], [0,0,0], [0,0,0]]
num_solutions = 0
if dyP == 0:
    # if dF/dy(P) = 0, change variables so that dF/dy != 0
    if (dxP != 0):
        g = F.substitute(x = y, y = x)
        Q = [P[1], P[0], P[2]]
        R = chord_and_tangent(g, Q)
        return [R[1], R[0], R[2]]
    elif (dzP != 0):
        g = F.substitute(y = z, z = y)
        Q = [P[0], P[2], P[1]]
        R = chord_and_tangent(g, Q)
        return [R[0], R[2], R[1]]
    else:
        raise ValueError, '%s is singular at %s'%(F, P)
else:
    # dF/dy(P) != 0

    # scale P to z=1 if possible
    if (P[2] != 0):
        P[0] = P[0]/P[2]
        P[1] = P[1]/P[2]
        P[2] = 1

    # option 1: z = 0
    g = F.substitute(y = -dxP//dyP*x, z = 0)
```

```
        if (g == 0):
            solutions[0][0] = 1
            solutions[0][1] = -dxP//dyP
            solutions[0][2] = 0
            num_solutions = num_solutions + 1

        # check whether the solution is equal to P
        equal_to_P = False
        if (P[2] == 0):
            if (P[1] != 0):
                if (P[0] != 0):
                    if (solutions[0][0]/P[0] == P[1]/solutions[0][1]/P[1]):
                        equal_to_P = True
                elif (solutions[0][0] == 0):
                    equal_to_P = True
                elif (solutions[0][1] == 0):
                    equal_to_P = True

        if (not equal_to_P): # a new point was found
            return solutions[0]

        # option 2: z != 0
        g = F.substitute(y = -dzP//dyP - dxP//dyP*x, z=1)
        assert (g != 0)
        h = g.factor(proof=False)
        for factor in h:
        sol_x = -(factor[0]).constant_coefficient()/(factor[0]).coefficient(x)
        solutions[num_solutions][0] = sol_x
        solutions[num_solutions][1] = -dzP/dyP-dxP/dyP*sol_x
        solutions[num_solutions][2] = 1
        num_solutions = num_solutions + 1
        if ((solutions[0] != [0,0,0]) and not (are_projectively_equivalent(
solutions[0],P))):
            return solutions[0]
        elif ((solutions[1] != [0,0,0]) and not (are_projectively_equivalent(
solutions[1],P))):
            return solutions[1]
        elif ((solutions[2] != [0,0,0]) and not (are_projectively_equivalent(
solutions[2],P))):
            return solutions[2]
        else:
            return P

def are_projectively_equivalent(P, Q):

    length_P = len(P)
    if length_P != len(Q):
```

```
    raise TypeError, '%s and %s do not have the same length'%P,Q
ratio = 0
for i in range(len(P)):
if (ratio != 0):
    if (Q[i] != (P[i])*ratio):
    return False
else:
    if (P[i] == 0):
    if (Q[i] != 0):
        return False
    else:
    ratio = Q[i]/P[i]
return True
```

References

[1] John Cremona - G1CRPC: Rational Points on Curves, Course Notes 2003, Section 4.4, pp 29-31

[2] William A. Stein et al. - Sage Mathematics Software (Version 4.6.2), The Sage Development Team, 2011, http://www.sagemath.org

[3] Wikipedia, the free encyclopedia - Bézout's Theorem, http://en.wikipedia.org/wiki/Bezout's_theorem, 30 April 2011