PICARD GROUPS OF CURVES

David R. Kohel

PICARD GROUPS OF CURVES

PICARD GROUPS OF CURVES

David R. Kohel

§1 Introduction

This document provides some rudimentary examples for a package for working in the degree zero group $\mathrm{Pic}^0(C)$ of a general plane curve C. This package makes use of generic algorithms of Florian Hess for the representation and reduction of divisors on curves (in terms of their function fields).

§2 Creation of the Picard group

`PicardGroup(C,p)`

Given a curve C and a rational point or degree one place p on C, return the *degree zero* divisor class group of C.

§3 Creation of Picard group elements

`J ! q`

Given the Picard group J of a curve C and a rational point or degree one place q on C, return the reduced divisor class element $[q] - [p]$, where p is the base point for divisor reduction.

§4 Arithmetic of elements

`P + Q`

Given points P and Q in $\mathrm{Pic}^0(C)$ find a reduced representative for the sum and and return it as an element of $\mathrm{Pic}^0(C)$.

`P - Q`

Given points P and Q in $\mathrm{Pic}^0(C)$ find a reduced representative for the difference and return it as an element of $\mathrm{Pic}^0(C)$.

`P eq Q`

Given points P and Q in $\mathrm{Pic}^0(C)$ return `true` if and only if they represent the same divisor class.

§5 Picard group examples

Example E1

The Picard group gives an alternative means of constructing and representing points on an elliptic curve, which need not be in Weierstrass form.

```
PP<x,y,z> := ProjectivePlane(RationalField());
// Curve number 1.
C1 := Curve(PP,x^2*y+y^2*z+z^2*x);
p0 := Place(C1![1,0,0]);
p1 := Place(C1![0,1,0]);
p2 := Place(C1![0,0,1]);
J1 := PicardGroup(C1,p0);
P0 := J1!p0;
P1 := J1!p1;
P2 := J1!p2;
for n in [1..6] do n*P1; end for;
P2 eq 2*P1;

> // Curve number 2.
> C2 := Curve(PP,y^2*z-x^3-x*z^2-2*z^3);
> p0 := C2![0,1,0];
> p1 := C2![1,2,1];
> p2 := C2![-1,0,1];
> p3 := C2![1,-2,1];
> J2 := PicardGroup(C2,p0);
> P1 := J2!p1;
> P2 := J2!p2;
> P3 := J2!p3;
> for n in [1..8] do n*P1; end for;
(x - z, y - 2*z)
(x + z, y)
(x - z, y + 2*z)
0
(x - z, y - 2*z)
(x + z, y)
(x - z, y + 2*z)
0
> P3 eq 3*P1;
true

> // Curve number 3.
> C3 := Curve(PP,y^2*z-x^3-x*z^2-3*z^3);
> p0 := Place(C3![0,1,0]);
> p1 := Place(C3![-1,1,1]);
> J3 := PicardGroup(C3,p0);
```

```
> P1 := J3!p1;
> [ n*P1 : n in [1..7] ];
[
    (x + z, y - z),
    (x - 6*z, y + 15*z),
    (x - 11/49*z, y - 617/343*z),
    (x - 1081/900*z, y + 65771/27000*z),
    (x - 179051/80089*z, y - 91814227/22665187*z),
    (x + 6465234/18653761*z, y + 130201927155/80565593759*z),
    (x - 32232855119/1854938761*z, y - 5798155134248651/79890357497509*z),
]
```

Example E2

The Picard group constructor can be used to define and work with points in the Jacobian of a higher genus curve, using generic divisor reduction to find a unique representative for the divisor class.

```
> PP<x,y,z> := ProjectivePlane(RationalField());
> // Curve number 4.
> C4 := KleinQuartic(PP);
> p0 := C4![0,1,0];
> p1 := C4![0,0,1];
> p2 := C4![1,0,0];
> J4 := PicardGroup(C4,p0);
> P1 := J4!p1;
> P2 := J4!p2;
> [ n*P1 : n in [1..7] ];
> P1 eq 5*P2;
> E0 := PP!!EllipticCurve([ 1, -1, 0, -2, -1 ]);
> phi := map< C4->E0 |
> [
>     x^5*y + x^5*z - x^4*y^2 - x^4*y*z - x^4*z^2 - x^3*y^3 - 2*x^3*y^2*z -
>     2*x^3*y*z^2 - x^3*z^3 - x^2*y^4 - 2*x^2*y^3*z - 2*x^2*y*z^3 - x^2*z^4 +
>     x*y^5 - x*y^4*z - 2*x*y^3*z^2 - 2*x*y^2*z^3 - x*y*z^4 + x*z^5 + y^5*z -
>     y^4*z^2 - y^3*z^3 - y^2*z^4 + y*z^5,
>     x^6 - 2*x^5*y - 2*x^5*z - x^4*y^2 - 3*x^4*y*z - x^4*z^2 - x^2*y^4 -
>     3*x^2*y^2*z^2 - x^2*z^4 - 2*x*y^5 - 3*x*y^4*z - 3*x*y*z^4 - 2*x*z^5 +
>     y^6 - 2*y^5*z - y^4*z^2 - y^2*z^4 - 2*y*z^5 + z^6,
>     x^3*y^3 + 3*x^3*y^2*z + 3*x^3*y*z^2 + x^3*z^3 + 3*x^2*y^3*z +
>     6*x^2*y^2*z^2 + 3*x^2*y*z^3 + 3*x*y^3*z^2 + 3*x*y^2*z^3 + y^3*z^3
> ] >;
[
    (x, y),
    (x^2, x*y, y^2),
    (y, z),
    (x*y, x*z, y^2, y*z),
```

```
    (x^2*y, x^2*z, x*y^2, x*y*z, y^3, y^2*z),
    (y^2, y*z, z^2),
    0
]
> E0!0 eq phi(p0);
true
> phi(p1) eq 5*phi(p2);
true

// Curve number 5.
C5 := HyperellipticCurve(x^5+x+2)
    where x := PolynomialRing(Rationals()).1;
WW<x,y,z> := Ambient(C5);
K5 := FunctionField(C5);
// p0 := Place(C5![1,0,0]); // This constructor fails.
p0 := CurvePlace(q0) where
    q0 := Places(K5)!InfinitePlaces(RationalExtensionRepresentation(K5))[1];
t1 := Place(C5![-1,0,1]);
p1 := Place(C5![1,2,1]);
p2 := Place(C5![2,6,1]);
J5 := PicardGroup(C5,p0);
T1 := J5!t1;
P1 := J5!p1;
P2 := J5!p2;
// 2-torsion point
for n in [0..4] do
    n*T1;
end for;
// Independent points on Jacobian.
// Verify that P1 and P2 satisfy no tiny relations:
O := J5!0;
for n in [0..3] do
    for m in [-3..3] do
        if [n,m] ne [0,0] then
            assert (n*P1+m*P2) ne O;
        end if;
    end for;
end for;
// Use Michael Stoll's hyperelliptic package to represent the points.
JJ := Jacobian(C5);
Q1 := JJ![C5![1,2,1],C5![1,0,0]];
Q2 := JJ![C5![2,6,1],C5![1,0,0]];
// Verify that the points are independent using heights:
> HeightPairingMatrix([Q1,Q2]);
[ 0.8329738853736738873135 -0.390756737948500594185370]
[-0.3907567379485005941853  1.337666393013092652613006]
```

A1 PicCrv Package

picard_group.m

```
//////////////////////////////////////////////////////////////////////
//                                                                  //
//                           David Kohel                            //
//                      Picard Groups of Curves                     //
//                                                                  //
//////////////////////////////////////////////////////////////////////
declare verbose PicardGroup, 2;
//////////////////////////////////////////////////////////////////////
//                                                                  //
//                       Attribute declarations                     //
//                                                                  //
//////////////////////////////////////////////////////////////////////
declare attributes PicCrv:  // Picard group
    Curve,                  // Nonsingular curve
    Place;                  // Degree one place for divisor reduction
declare attributes PicCrvElt:
    Parent,
    Element;
//////////////////////////////////////////////////////////////////////
//                                                                  //
//                            Coercions                             //
//                                                                  //
//////////////////////////////////////////////////////////////////////
function PicCrvCreate(J,D)
    P := HackobjCreateRaw(PicCrvElt);
    P`Parent := J;
    P`Element := D + r*O
        where D, r := Reduction(D,O)
        where O := Parent(P)`Place;
    return P;
end function;


intrinsic HackobjCoercePicCrv(J::PicCrv,P::.) -> BoolElt, PicCrvElt
    {}
    if Parent(P) cmpeq J then
        return true, P;
    elif Type(P) eq RngIntElt then
        if P eq 0 then
            K := FunctionField(J`Curve);
```

```
            return true, PicCrvCreate(J,DivisorGroup(K)!0);
        end if;
    elif ISA(Type(P),Pt) then
        if Scheme(P) ne Curve(J) then
            return false, "Argument 2 must be on curve of argument 1.";
        elif Ring(Parent(P)) ne BaseRing(Curve(J)) then
            return false,
                "Argument 2 must be a point over the base field of curve.";
        end if;
        O := ReductionDivisor(J);
        if IsNonsingular(P) then
            P := FunctionFieldPlace(Place(P));
            return true, PicCrvCreate(J,P-O);
        else
            plcs := Places(P);
            vals := &+[ Valuation(P,p) : p in plcs ];
            plcs := [ FunctionFieldPlace(p) : p in plcs ];
            D := &+[ vals[i]*plcs[i] : i in [1..#plcs]] - (&+vals)*O;
        end if;
    elif Type(P) eq PlcCrvElt then
        if Curve(P) ne Curve(J) then
            return false, "Argument 2 must be on curve of argument 1.";
        end if;
        D := 1*FunctionFieldPlace(P) - Degree(P)*ReductionDivisor(J);
    elif Type(P) eq PlcFunElt then
        if FunctionField(P) ne FunctionField(Curve(J)) then
            return false, "Argument 2 must be on curve of argument 1.";
        end if;
        D := P - Degree(P)*ReductionDivisor(J);
    elif Type(P) eq DivCrvElt then
        if Curve(P) ne Curve(J) then
            return false, "Argument 2 must be on curve of argument 1.";
        end if;
        D := FunctionFieldDivisor(P) - Degree(P)*ReductionDivisor(J);
    elif Type(P) eq DivFunElt then
        if FunctionField(P) ne FunctionField(Curve(J)) then
            return false, "Argument 2 must be on curve of argument 1.";
        end if;
        D := P - Degree(P)*ReductionDivisor(J);
    else
        return false, "Invalid coercion";
    end if;
    return true, PicCrvCreate(J,D);
end intrinsic;
```

```
////////////////////////////////////////////////////////////////////////
//                                                                    //
//                         Creation Functions                         //
//                                                                    //
////////////////////////////////////////////////////////////////////////


intrinsic PicardGroup(C::Crv,P::Pt) -> PicCrv
    {}
    require IsProjective(C) and IsField(BaseRing(C)) :
        "Argument 1 must be a projective curve over a field.";
    require Scheme(P) eq C and Ring(Parent(P)) cmpeq BaseRing(C) :
        "Argument 2 must be a point of argument 1 over its base ring.";
    J := HackobjCreateRaw(PicCrv);
    J`Curve := C;
    J`Place := 1*FunctionFieldPlace(Place(P));
    return J;
end intrinsic;


intrinsic PicardGroup(C::Crv,P::PlcCrvElt) -> PicCrv
    {}
    require IsProjective(C) : "Argument 1 must be a projective curve.";
    require Degree(P) eq 1 : "Argument 2 must be of degree one.";
    require C eq Curve(P) :
        "Argument 1 must be the curve of argument 2.";
    J := HackobjCreateRaw(PicCrv);
    J`Curve := C;
    J`Place := 1*FunctionFieldPlace(P);
    return J;
end intrinsic;


intrinsic PicardGroup(P::PlcCrvElt) -> PicCrv
    {}
    C := ProjectiveCurve(P);
    require Degree(P) eq 1 : "Argument 2 must be of degree one.";
    require C eq Curve(P) :
        "Argument 1 must be the curve of argument 2.";
    J := HackobjCreateRaw(PicCrv);
    J`Curve := C;
    J`Place := 1*FunctionFieldPlace(P);
    return J;
```

```
end intrinsic;


intrinsic PicardGroup(P::PlcFunElt) -> PicCrv
    {}
    C := ProjectiveCurve(FunctionField(P));
    require Degree(P) eq 1 : "Argument 2 must be of degree one.";
    J := HackobjCreateRaw(PicCrv);
    J`Curve := C;
    J`Place := 1*P;
    return J;
end intrinsic;
////////////////////////////////////////////////////////////////////////
//                                                                    //
//                            Printing                                //
//                                                                    //
////////////////////////////////////////////////////////////////////////
function SprintDivisor(D)
    if IsZero(D) then
        S := "0";
    else
        fcns := GroebnerBasis(Ideal(CurveDivisor(D)));
        S := "(";
        for i in [1..#fcns] do
            S *:= Sprint(fcns[i]);
            S *:= i lt #fcns select ", " else ")";
        end for;
    end if;
    return S;
end function;


intrinsic HackobjPrintPicCrv(J::PicCrv, level::MonStgElt)
    {}
    printf "Picard group of %o", J`Curve;
end intrinsic;


intrinsic HackobjPrintPicCrvElt(P::PicCrvElt, level::MonStgElt)
    {}
    D := Reduction(P`Element,Parent(P)`Place);
    printf "%o", SprintDivisor(D);
end intrinsic;
```

```
intrinsic HackobjParentPicCrvElt(P::PicCrvElt) -> PicCrv
   {}
   return P`Parent;
end intrinsic;


intrinsic 'eq' (J1::PicCrv,J2::PicCrv) -> BoolElt
   {}
   return J1`Curve cmpeq J2`Curve and J1`Place cmpeq J2`Place;
end intrinsic;


intrinsic 'eq' (P::PicCrvElt,Q::PicCrvElt) -> BoolElt
   {}
   J := P`Parent;
   require J eq Q`Parent : "Arguments must have the same parent.";
   D, r := Reduction(P`Element-Q`Element,J`Place);
   return r eq 0;
end intrinsic;
//////////////////////////////////////////////////////////////////////////
//                                                                      //
//                         Structure Attributes                         //
//                                                                      //
//////////////////////////////////////////////////////////////////////////


intrinsic Divisor(P::PicCrvElt) -> DivCrvElt
    {}
    return CurveDivisor(P`Element);
end intrinsic;


intrinsic Curve(J::PicCrv) -> Crv
    {}
    return J`Curve;
end intrinsic;


intrinsic ReductionDivisor(J::PicCrv) -> Crv
    {}
    return J`Place;
```

```
end intrinsic;


intrinsic Zero(J::PicCrv) -> Crv
    {}
    return J!(DivisorGroup(J`Curve)!0);
end intrinsic;


intrinsic IsZero(P::PicCrvElt) -> Crv
    {}
    return IsZero(P`Element);
end intrinsic;
////////////////////////////////////////////////////////////////////////
//                                                                    //
//                   Arithmetic operations, etc.                      //
//                                                                    //
////////////////////////////////////////////////////////////////////////
function BinaryExpansion(n)
    if n in {0,1} then return [n]; end if;
    return [ n mod 2 ] cat BinaryExpansion(n div 2);
end function;


intrinsic '*'(n::RngIntElt,P::PicCrvElt) -> PicCrvElt
    {}
    Q := Zero(Parent(P));
    if n eq 0 then
        return Q;
    elif n lt 0 then
        P := -P;
        n *:= -1;
    end if;
    if n eq 1 then
        return P;
    elif n eq 2 then
        O := Parent(P)`Place;
        Q`Element := D + r*O where D, r := Reduction(2*P`Element,O);
        return Q;
    end if;
    b := BinaryExpansion(n);
    for i in [1..#b] do
        if b[i] eq 1 then Q := P+Q; end if;
```

```
        P := 2*P;
    end for;
    return Q;
end intrinsic;


intrinsic '+'(P::PicCrvElt,Q::PicCrvElt) -> PicCrvElt
    {}
    require Parent(P) eq Parent(Q) :
        "Arguments must have the same parent.";
    if IsZero(Q`Element) then return P; end if;
    if IsZero(P`Element) then return Q; end if;
    return Parent(P)!(P`Element + Q`Element);
end intrinsic;


intrinsic '-'(P::PicCrvElt) -> PicCrvElt
    {}
    if IsZero(P`Element) then return P; end if;
    return Parent(P)!(-P`Element);
end intrinsic;


intrinsic '-'(P::PicCrvElt,Q::PicCrvElt) -> PicCrvElt
    {}
    require Parent(P) eq Parent(Q) :
        "Arguments must have the same parent.";
    if IsZero(Q`Element) then return P; end if;
    if IsZero(P`Element) then return -Q; end if;
    return Parent(P)!(P`Element - Q`Element);
end intrinsic;
```