Module array.g

1999/4/5, Nils Bruin

Implementation of *Array* data type. It provides support (although not very efficiently) for multidimensional arrays. The data structure is a record with the following fields.

`String`	`Type`	Constant "Array"
`Int`	`dim`	dimension of array (number of indices)
`List`	`L`	nested list of elements.

Arrays are indexed including 0. The index type is a list of integers. The first index refers to the position in L.

Function ArrInit

Creates an Array data structure.

SYNTAX. `A:=ArrInit(d,L);`

`Array`	`A`	resulting Array
`Int`	`d`	dimension
`List`	`L`	appropriate nested list, describing initial values

EXPLANATION. Returns an Array of given dimension, initialised with L, which should be a nested list of appropriate depth. The entries of L are NOT copied. The call ArrInit(d,[]) is valid and creates an uninitialised Array of dimension d.

Function ArrSetElt

set an element of an Array

SYNTAX. `ArrSetElt(A,I,obj);`

`Array`	`A`	array to set the element in
`List`	`I`	list of integers: index in A
`object`	`obj`	the value

EXPLANATION. It is an error if Length(I)<>A.dim

Function ArrElt

Retrieve an element from an array

SYNTAX. `obj:=ArrElt(A,I);`

`object`	`obj`	the value
`Array`	`A`	array to set the element in
`List`	`I`	list of integers: index in A

Function ListAugFront

returns a list with element added in front

SYNTAX. `L:=ListAugFront(obj,M);`

`List`	`L,M`
`object`	`obj`

EXPLANATION. Result is undefined if entries from L are duplicated or identically appended.

Function ArrLookup

Look up particular value in Array

SYNTAX. `I:=ArrLookup(A,obj);`

`List | Boolean`	`I`	index of obj in A or false
`Array`	`A`	Array to look up obj in
`object`	`obj`	value to look up

EXPLANATION. returns index of obj in A or false, if it does not occur. If obj occurs more than once, the lexicographically smallest index is returned (i.e. the index that occurs first if A is written out as list of lists).

Function ArrLookupAll

Look up all occurences of some value in an array.

SYNTAX. `L:=ArrLookupAll(A,obj);`

`List`	`L`	List of indices of obj in A
`Array`	`A`	Array to look up obj in
`object`	`obj`	value to look up

Function ArrApply

Apply a function to all entries of an Array

SYNTAX. `A:=ArrApply(B,f);`

`Array`	`A,B`	
`function`	`f`	function taking one parameter and returning a value

EXPLANATION. The returned Array is a copy of the original. ArrApply(A,x$\rightarrow$x) will return an array with identical elements, though.

Function ArrToList

Nested list representation of Array.

SYNTAX. `L:=ArrToList(A);`

`List`	`L`
`Array`	`A`

EXPLANATION. Beware that the returned nested list actually corresponds to the Array. It is not a copy.

Function ArrAddRow

Add a row to an array

SYNTAX.
```
ArrAddRow(A,B);
ArrAddRow(A,obj);

Array   A,B
object  obj
```

EXPLANATION. If A is of dimension 1, then the second argument is simply added at the end of A. If A is of higher dimension, then B should be of one dimension less and then the entries of B are added (not copied) as an extra row.

Module desc2.g

2000/11/4, Nils Bruin

2(-isogeny) descent on elliptic curves over number fields. This module will be part of a bigger module, handling general 2-descent on elliptic curves. However, since none of these methods are actually algorithms, the programs might occasionally need some help from the user and therefore it is useful to have documentation that digs into the bowels of the implementation.

The routines presented here (culminating in EllGensMod2) deal with elliptic curves of the form $\gamma Y^2 = X^3 + a_2X^2 + a_4X$. Computation intensive data is stored in the Ell data structures in some additional fields.

`Ell`	`ec2iso`	the 2-isogenous curve
`function`	`psi2`	the 2-isogeny to ec2iso
`function`	`psi2inv`	lift back from ec2iso (otherwise false)
`List`	`selmer2isosupp`	for EllSelmer2isoSupp
`function`	`mu2iso`	selmer map
`List`	`mu2isox`	generating x-coordinates
`function`	`homspc2iso`	quartic defining homspc for delta
`function`	`homspc2isoToX`	x-coordinate for a point on a homspc

For data structures for full 2-descent, see description of Ell2DescInit.

WARNING!!! Selmer group computations also involve places at infinity. Precision settings may affect their outcome. The curve $y^2 = x^3 - 1042011$ is an example of this phenomenon. With OrderPrec(50), you get a rank bound of 3, where with OrderPrec(300) the bound is 4, which is the rank of the curve. This is due to incorrect answers returned by EltCon. The remedy suggested by the KANT-team is: use HEAP(O, "USE_ELT_PROD_REP", 50); where O is the order of definition for 2-iso descent or O is Aa for full 2-descent. This is done if Desc2Init is called without presupplied Aa (if you do supply Aa, you'll have to take care of it yourself)

The archimedian computations may lead to another error as well. Sometimes, roundoff-errors are so bad that the program has trouble figuring out how the 2-torsion over C matches up with the algebraic (exact) structures. In that case, you can increase _EllSelRPtoll. It is probably a good idea to increase Prec() and OrderPrec() as well then.

For finding local generators of $E(K_p)/2E(K_p)$ and such, just randomly trying points is often a better strategy than sytematically combing through the homogeneous spaces. For full 2-descents, this is even the only tool implemented in this program. For 2-isogeny descent, this strategy is tried for a limited time before resorting to systematically working through the homogeneous spaces. For the full descent, a few attempts are made. If this does not result in enough points, then the program gives up. The fould points are still stored, so actually rerunning might be succesful. If you want the program to be more persistent, then change the value of the global variable EllShootBound. It is used by both 2-isogeny and full 2-descent.

Function Ell2Iso

Returns the 2-isogenous curve

SYNTAX. `E2:=Ell2Iso(E1);`

```
P  Ell:  E1,E2
```

EXPLANATION. This routine has memory, so subsequent calls will return an identical answer. The action is the following. The 2-isogenous curve (by dividing out by (0:0:1)) is computed and returned. Additionally several maps are installed.

`function`	`E1.psi2`	maps EllPnts on E1 to E2
`function`	`E1.psi2inv`	lifts point back to E1, if possible
`function`	`E1.mu2iso`	maps EllPnts on E1 to $K^*/(K^*)^2$
`function`	`E1.homspc2iso`	delta to a quartic describing a homspc
`function`	`E1.homspc2isoToX`	x in the homspace back to E1

For E2, the same maps are installed, such that E2.psi2 maps to E1 and that the combination of these two maps gives the multiplication-by-two map.

For non-localised Ells, his routine calls EllSelmer2isoSupp(E1) and installs localisations at those places, together with 2-isogenies to localisations of E2. This means that for even class orders, one may need a call to the routine EllSelmer2isoSetSupp first.

Function EllLookupHints

Returns advised X-coordinates to look at for finding interesting points.

SYNTAX. `L:=EllLookupHints(E);`

```
List  L
Ell   EllLookupHints
```

EXPLANATION. For looking up the hints, a mild equality test is done on elliptic curves: only to coefficient lists are compared. For localised curves, also the global hints are returned. If E.a6=0, then 0 is always returned as a hint.

For global hint lookups with E.a6=0, there is a special facility. Hints for the isogenous curve are also looked at. If they have trivial image under mu2iso, then they are lifted to x-coordinates on E itself. This means that global point searching on the isogenous curve can be helped by specifying the images on the original curve. This help when checking that a suspected set of Mordell-Weil generators actually generates $E(K)/E(K)[2]$.

Hints should not be trusted to be correct. They don't even have to correspond with rational points.

Function EllAddHint

Register a suggestion for searching routines

SYNTAX. `EllAddHint(E,x);`

`Ell`	`E`	elliptic curve to register hint for
`Ring element	List`	`x`	suggestion for searching

EXPLANATION. Searching for points on elliptic curves (globally or on localised curves) is a hard job. Sometimes, you already know some points. It is a great timesaver if such points can be used by the computer as well. This routine enables you to tell the searching routines first to look at a given x-coordinate before trying systematic searching. It is only that, so registering false hints will not produce wrong answers. If you want to register multiple hints for a curve, just call the routine repeatedly or call with a list of values.

Hints for localised curves can save the computer a systematic search for points. After a succesful 2-isogeny descent, the "mu2isox" fields in all associated Ells form a full set of hints.

Function HomSpcLocalPoint

Returns an X-coordinate of a local point on a homogeneous space or false.

SYNTAX. `x:=HomSpcLocalPoint(plc,P);`
`x:=HomSpcLocalPoint(plc,L);`

`Boolean	Elt`	`x`	
`Place`	`plc`	
`Poly`	`P`	even quartic
`List`	`L`	list of coefficients of $x^4, x^2, 1$.

EXPLANATION. Routine interfaces with the internals of HypEllFindLocX and is a little more efficient. It tries to flatten out the newton polygon, for instance.

Function EllMu2isoGenX

Find X-coordinates of curve mod image of 2-isogeny.

SYNTAX. `L:=EllMu2isoGenX(E)`

`List	String`	`L`	list of x-coordinates or "unknown".
`Ell`	`E`	elliptic curve over an order or localised curve.

EXPLANATION. If E is localised, this routine searches for local points generating the group of local points modulo the image of the 2-isogeny, both on

E and the 2-isogenous curve. This process may be labour-intensive but is in principle effective. HomSpcLocalPoint is used and is not completely fool-proof. Especially it there is p-adic 2-torsion that is not global, it might get stuck. Some hints should help then.

If E is over an order, then EllSelmer2iso is used to find a basis for the 2-isogeny selmer group. This routine tries to find points that generate the thing, first by using hints and then by searching homogeneous spaces. Searching is not done very well. For instance, denominators are not tried. If not enough points are found, then "unknown" is returned. This may either signify that points exist outside the searched area or that Sha has a 2-part.

Function EllSelmer2isoSetSupp

Sets the places that support the 2-isogeny selmer group of a curve

SYNTAX. `EllSelmer2isoSetSupp(E,L);`
`EllSelmer2isoSetSupp(E,I);`

`Ell`	`E`	
`List`	`L`	list of places
`Ideal`	`I`	ideal giving the support

EXPLANATION. For even class number, there is no canonical choice of this support, since the class group part can be represented by many ideals. This routine does not check the validity of the input, so be careful! A sufficient support (but possibly too large) is: the support of 2*EllDisc(E), extended with representatives of the 2-part of the class group.

This routine also sets the support for the isogenous curve. It should be called before Ell2Iso is called, because that routine depends on the answer of EllSelmer2isoSupp, which tries to find a support itself if none is given beforehand.

Because of the far-reaching consequences of changing this support, this call purges any stored 2-isogeny (descent) data in the elliptic curve. As a consequence, the curve returned by Ell2Iso(E) after setting the support is not identical to the one before. Summarising: call this routine right after constructing E.

Note that an erroneous call to this routine in purging of the information anyway.

Function EllSelmer2isoSupp

Returns the support for the 2 isogeny selmer group.

SYNTAX. `L:=EllSelmer2isoSupp(E);`

`List`	`L`	list of places
`Ell`	`E`	

EXPLANATION. If support is not preset, then return the the places above 2 together with places that divide the elliptic discriminant

Function EllSelmer2iso

Returns a basis of the 2-isogeny selmer group.

SYNTAX. `L:=EllSelmer2iso(E);`

`List`	`L`	list of Elts
`Ell`	`E`	

EXPLANATION. Uses localisations of E to bound the selmer group.

Function Ell2DescInit

Prepares an ell for a 2-descent.

SYNTAX. `Ell2DescInit(ec[,Aa[,thetaa][,plclist]][,Ar[,thetar]);`
`ec:=Ell2DescInit(ec);`

`Ell`	`ec`	elliptic curve (same curve is returned)
`Order`	`Ar`	relative order ec.R[x of 2-torsion]
`Order`	`Aa`	absolute version of Ar
`Elt`	`thetar`	Elt of Ar; x-coordinate of 2-torsion
`Elt`	`thetaa`	Elt of Aa; absolute version of thetar
`List`	`plclist`	List of places of Aa

EXPLANATION. This function installs several necessary data structures in ec for functions like EllSelmer2. Apart from ec, all arguments are optional and can be given in arbitrary order (their type identifies them). If both Aa and Ar are given, then KASH should be able to figure out morphisms between them. This is not trivial, since OrderIsSubfield is only implemented for orders over Z. If Ar=OrderRelativeOrder(Aa,ec.R) or Aa=OrderMaximal(OrderAbs(Ar)) then things should be OK. Aa and ec.R should be known to be maximal and have known class groups. This routine installs an embedding of ec.R in Aa that is compatible with the one of Ar in Aa. (and assumes that, as a result, kash makes sure that all other arrows commute) This may be a problem if Ar is galois over ec.R

If plclist is supplied, then this is taken as support for the selmer group (otherwise, this is set to the list of places over 2*EllDisc(ec), enlarged with places such that the even part of the class ggroup of Aa is supported by it. obviously, this parameter is only used for non-localised curves.

After this call, the following fields are installed

`Order`	`Aa`	
`Order`	`Ar`	
`Elt`	`thetaa`	
`Elt`	`thetar`	
`function`	`mu2`	map to Aa
`List`	`mu2x`	

The function ec.mu2 can either be called with an EllPnt or the x-coordinate of a point.

For non-localised curves, also

`List`	`selmersupp`	List of places of ec.R

is set. Additionally, the localisations at those places are computed and Ell2DescInit is called for those localisations with identical Aa, Ar, thetaa and thetar.

For localised curves, we have

`List`	`placelift`	List of places of Aa above ec.Place
`function`	`sqrvec`	concat of PlcSqrReps of places in placelift
`List`	`tor2x`	list of (approximations to) x-coords of p-adic 2-torsion.
`List`	`mu2spec`	associative list with values of mu2x for 2-torsion.

In this case, mu2x returns a *list* of values, corresponding to the places in placelift. Additionally, if mu2x is fed an approximation to an x-coordinate of a 2-torsion point (of sufficient precision to tell it apart from other points), mu2x returns an appropriate value, to be found in mu2spec.

Determining the classgroup of Aa is generally most expensive. It really pays off (if possible) to precompute Aa and its class group and supply it. The hints for Aa, Ar, thetaa and thetar can also be passed in the appropriate fields of ec, i.e. calling Ell2DescInit twice on the same curve does not lead to recomputing the class group.

Function EllMu2GenX

Tries to determine X-coordinates of points that generate E(K)/2E(K)

SYNTAX. `L:=EllMu2GenX(ec)`

`List`	`L`	list of x-coordinates
`Ell`	`ec`	elliptic curve

EXPLANATION. For localised curves, this routine computes the number of generators needed and tries to find them. First, the image of 2-torsion and hints are tried. If that is not enough then it randomly tries points (using EllRandomX). If it has no success after (currently) 50 tries, it gives up with an error.

For non-localised curves, EllSelmer2 is used to get a bound on the number of generators. Then the routine tries to find some points using HypEllFindX on the curve itself. Chances of succesfully finding generators are quite slim, so a negative answer shouldn't be interpreted as probable sha (but there might be ...)

Function XOrderPrimElt

Returns a generating element of the order (i.e. an element with as minimal polynomial the defining polynomial of the underlying equation order.

SYNTAX. `e:=XOrderPrimElt(O);`

```
Elt    e
Order  O
```

Function SigDif

Returns 10t̂he number of digits that is significant in the difference of two numbers of the same order of magnitude.

SYNTAX. `d:=SigDif(a,b);`

```
P  Real:  d,a,b;
```

EXPLANATION. returns Abs(a/b-1)*10P̂rec(). If a and b are not of the same order of magnitude then SigDif does not quite return what it's supposed to return, but it returns a large number anyway.

Function EllAa

Returns an absolute version of the order obtained by adjoining a 2-torsion point.

SYNTAX. `O:=EllAa(ec)`

```
Order  O
Ell    ec
```

EXPLANATION. The returned order is known to be maximal.

Function EllSelmer2

Computes basis for 2-selmer group represented in ec.Aa.

SYNTAX. `L:=EllSelmer2(ec);`

`List  L` list of elements of ec.Aa
`Ell   ec`

EXPLANATION. If Ell2DescInit is not yet called on ec, then this is done first. Note however that hints cannot be passed to EllSelmer2. ec.Aa can be set beforehand, though. Uses information at places in selmersupp as well as at infinity to determine the selmer group. Also, elements returned are required to have global squares for norms in ec.R.

Function EllGensMod2

Returns generators of E(K)/2E(K) or fails.

SYNTAX. `L:=EllGensMod2(E);`

`List`	`L`	list of EllPnts on E
`Ell`	`E`	

EXPLANATION. Does an EllMu2isoGenX on both E and Ell2Iso(E) and pushes the generators on the isogenous curve to E. Beware that this routine does not give generator that decompose the generated group in the direct sum of a torsion and a free part.

Function EllRankBound

Computes an upper bound for the rank of the curve by Selmer-computations

SYNTAX. `N:=EllRankBound(ec)`

`Int`	`N`	rank bound
`Ell`	`ec`	curve

EXPLANATION. At the moment, only implemen

Module domain.g

2000/1/21, Nils Bruin

Some routines to facilitate writing routines for generic domains. Z, Q, Orders and finite fields are mainly supported now. Main reason is to get representatives for 0 and 1 right.

Function Tail

Returns a list minus first element

SYNTAX. `M:=Tail(L);`

```
List  L,M
```

Function IsFFElt

Tests if an object is an FFElt

SYNTAX. `B:=IsFFElt(obj);`

```
Boolean  B
object   obj
```

Function IsFF

Tests if an object is an FF

SYNTAX. `B:=IsFF(obj);`

```
Boolean  B
object   obj
```

Function EltRing

Returns ring of definition of an object.

SYNTAX. `R:=EltRing(obj);`

```
Ring  R
P     object:  obj
```

EXPLANATION. returns ring of definition. Note that Z is NOT an order according to KASH. Thus, Z and Q are handled differently. Perhaps Z should be returned for rational numbers as well. This is not so critical since KASH automatically casts between integers and rationals. (Not so for orders !)

Function RingZero

Returns the zero element of a ring.

SYNTAX. `elt:=RingZero(R);`

`Element of R`	`elt`	equal to 0
`Ring`	`R`	

Function RingOne

Returns the one element of a ring. (These rings have one.)

SYNTAX. `elt:=RingZero(R);`

`Element of R`	`elt`	equal to 1
`Ring`	`R`	

Function RingForce

Tries casting an element to a given ring.

SYNTAX. `a:=RingForce(R,b)`

`Element of R`	`a`
`Ring`	`R`
`object`	`b`

EXPLANATION. Only works if KASH can figure out the cast itself. Not usable for lifting or descending from an order to Z. In those cases, no error is generated, but just the original element is returned. This routine should therefore not be considered user interface. Either put up with it or don't use it.

Function InitPolyAlg

Defines a multivariate polynomial ring and sets global variables to the monomials.

SYNTAX. `A:=InitPolyAlg(R,L)`

`PolyAlg`	`A`	
`Ring`	`R`	
`List`	`L`	list of strings designating variable names

Module ellbase.g

2001/7/30, Nils Bruin

Implementation of Ell, EllPnt and EllPntApprox data types and several useful routines. These types implement elliptic curves, point on them and p-adic approximations to them respectively. We consider curves of the form $\gamma Y^2 = X^3 + a_2X^2 + a_4X + a_6$ The fields present in Ell may vary and other packages are free to add to them. We always have

`String`	`Type`	constant "Ell"
`Rec`	`operations`	constant _EllOps
`Ring`	`R`	field of definition
`Element of R`	`gamma,a2,a4,a6`	coefficients
`List`	`isom`	list of pairs [ec,f] (memory for EllIsom)

We may have (by EllGenInit or inheritance through EllLocalise or EllRed)

`List`	`G`	list of EllPnt
`Int`	`r`	supposed rank of G

EllLocalise creates in curves over orders (signifies curves over number fields)

`List`	`lcl`	list of Ell over "local" fields

An Ell produced by EllLocalise has (this acts as a flag for "localised" curves)

`Place`	`Place`	place giving localisation
`Ell`	`red`	reduced curve (produced once EllRed is called)
`List`	`kerZ`	logarithms of generators of kernel of reduction

An Ell over a finite field may have

`Int`	`size`	set by EllSize
`Array`	`group`	set by EllGroup, EllKernel
`List`	`kerlat`	list of relation vectors, set by EllGroup, EllKernel

The Quar routine supplies the following functions

`function`	`fromquar`	map from the quartic to Ell
`function`	`toquar`	inverse map
`function`	`fromquarrev`	map from the reversed quartic to Ell
`function`	`toquar`	inverse map

Not all ring types are supported or give sensible results in all routines. In general, orders and finite fields are supported. Z and Q are supported for as far as KASH allows them to be treated as orders (if this is not good enough for you, don't complain but do a better job yourself)

For EllPnt, we have the following fields

`String`	`Type`	constant "EllPnt"
`Rec`	`operations`	constant _EllPntOps
`Ell`	`ec`	elliptic curve on which the point lies
`Ring element`	`x,y,d`	projective coordinates describing the point

Operations defined for a point are addition, subtraction, scalar multiplication with integers, comparison and reduction mod a Place.

For EllPntApprox, we have the following fields

`String`	`Type`	constant "EllPntApprox"
`Rec`	`operations`	constant _EllPntApproxOps
`Ell`	`ec`	localised elliptic curve
`Ring element`	`x,y,d`	(approximate) coordinates
`Int`	`prec`	padic precision.

This module uses place.g, array.g and pntsrch.g

Function IsEll

Test if object is an Ell

SYNTAX. `B:=IsEll(obj);`

```
Boolean  B
P        object:  obj
```

Function EllDisc

compute discriminant of elliptic curve

SYNTAX. `D:=EllDisc(E);`

```
Ring element  D
Ell           E
```

EXPLANATION. If E.gamma=1, then this routine computes the discriminant of the given model. For other gamma, the discriminant is computed for the the model as if gamma were 1 and that value is multiplied by gamma. This ensures that primes of bad reduction do divide the discriminant (provided that all coefficients are integral).

Function EllJay

Compute j-ivariant of curve, classifying the geometric isomorphism class of the curve.

SYNTAX. `j:=EllJay(E)`

```
Ring element  j
Ell           E
```

Function Ell

Initialise an Ell data structure

SYNTAX. `E:=Ell(gamma,a1,a2,a3,a4,a6)`
`E:=Ell([gamma,a1,a2,a3,a4,a6])`

```
Ell           E
Ring element  gamma,a2,a4,a6
Ring element  a1,a3            constant 0
```

EXPLANATION. initialises the elliptic curve $\gamma Y^2 = X^3 + a_2X^2 + a_4X + a_6$. It is an error if a1 and a3 are not 0. The ring type of E is determined from the type of the given parameters. The parameters can be specified both as a list and as separate parameters

Function EllMonic

returns a true weierstrass model (i.e. with gamma=1)

SYNTAX. `ec1:=EllMonic(ec2);`

```
Ell  ec1,ec2
```

Function IsEllPnt

Test if an object is an EllPnt

SYNTAX. `B:=IsEllPnt(obj);`

```
Boolean  B
object   obj
```

Function EllPntCheck

Sanity check on EllPnt (is called upon creation of an EllPnt)

SYNTAX. `B:=EllPntCheck(P)`

```
Boolean  B
EllPnt   P
```

Function ProjPointNormal

bring a projective point in normal form

SYNTAX. `L:=ProjPointNormal(R,M)`

```
List  L   normalised point
Ring  R   ring of definition
List  M   3 element list
```

EXPLANATION. Divides the list by the non-zero entry with highest index

Function EllPnt

Defines an EllPnt on an Ell

SYNTAX.
```
P:=EllPnt(E,x,y);
P:=EllPnt(E,x,y,d);
P:=EllPnt(E,L);
```

`EllPnt`	`P`
`Ell`	`E`
`Ring element`	`x,y,z`
`List`	`L`

EXPLANATION. Depending upon the called form, initialises the point with coordinates (x,y) or (x/d,y/d). If the second argument is a list, then this is treated as [x,y] or [x,y,d]. It is an error if the point does not lie on E.

Function EllPntToList

returns list of projective coordinates describing point

SYNTAX. `L:=EllPntToList(P);`

`List`	`L`	coordinates
`EllPnt`	`P`	

Function EllPntNeg

Negates an EllPnt

SYNTAX. `Q:=EllPntNeg(P)`

`EllPnt`	`P,Q`

Function EllZero

Returns the neutral element on an Ell (the point at infinity)

SYNTAX. `P:=EllZero(E)`

`EllPnt`	`P`
`Ell`	`E`

Function EllPntAdd

addition of EllPnts

SYNTAX. `P:=EllPntAdd(Q,R)`

`EllPnt`	`P,Q,R`

Function EllPntMul

Multiplication of an EllPnt with an integer (by repeated doubling)

SYNTAX. `Q:=EllPntMul(n,P)`

`EllPnt`	`P,Q`
`Int`	`n`

Function EllYsqr

Returns $(x^3 + a_2x^2 + a_4x + a_6)/\gamma$

SYNTAX. `y:=EllYsqr(E,x);`

```
Ring element  x,y
Ell           E
```

EXPLANATION. Useful for testing if an X-coordinate belongs to a rational point

Function EllGenInit

Initialises fixed finitely generated group on Ell

SYNTAX. `EllGenInit(L,r);`

`List`	`L`	List of EllPnts
`Int`	`r`	supposed rank

EXPLANATION. L should be a non-empty list of EllPnts, all on the same curve. The generated group is used by a lot of routines as if it were the Mordell-Weil group. By convention, the first r points should generate a free group and the other points should be torsion. These points are not expected to change in the life of an Ell. For instance, EllLocalise and EllRed inherit these points. As a consequence, this routine purges any precomputed data on Ell.

Function EllVecToPnt

Compute a linear combination of generators defined by EllGenInit

SYNTAX. `P:=EllVecToPnt(E,L)`

`EllPnt`	`P`	
`Ell`	`E`	
`List`	`L`	list of integers

EXPLANATION. The length of L should be the same as the number of points given to EllGenInit

Function EllLocalise

Get the elliptic curve over a "local field"

SYNTAX. `E2:=EllLocalise(E1,plc)`

```
Ell    E1,E2
Place  plc
```

EXPLANATION. E1 should be an Ell over an order and plc a Place over the same order. The curve E2 is a curve over the "completion" of the order at that place. It is a place holder for local information. E2 inherits generators installed by EllGenInit. Ells remember localisations: Subsequent calls with equal places give identical results unless the memory was purged for some reason.

Function IsEllPntApprox

Test for EllPntApprox

SYNTAX. `B:=IsEllPntApprox(obj)`

```
Boolean  B
object   obj
```

Function EllRed

Return reduction of Ell at place of good reduction

SYNTAX. `Er:=EllRed(E);`
`Er:=EllRed(E,plc);`

```
Ell    Er,E
Place  plc
```

EXPLANATION. If E is localised and plc is given, then plc should match the place at which E is localised. If E is global, then plc must be specified. In the latter case, the call is equivalent to EllRed(EllLocalise(E,plc)). The reduced curve inherits generators installed by EllGenInit. Localised curves have a memory for EllRed (and EllLocalise has memory as well, so indirectly, curves over number fields have memory as well), so subsequent calls produce identical answers, unless the memory is purged.

This action is invoked if E mod plc is evaluated.

Function EllPntRed

Reduces a point

SYNTAX. `Pr:=EllPntRed(P);`
`Pr:=EllPntRed(P,plc);`

```
EllPnt | EllPntApprox  P
EllPnt                 Pr
Place                  plc
```

EXPLANATION. Returns the image of an EllPnt under the reduction map. This action is invoked when evaluating P mod plc.

Function EllSize

Count points on an Ell over a finite field

SYNTAX. `N:=EllSize(E);`

`Int`	`N`	Size of E(E.R)
`Ell`	`E`	Ell over finite field

EXPLANATION. No fancy algorithm, but there is a memory.

Function EllSpan

Return (finite) group generated by given elements

SYNTAX. `R:=Ell(E,L)`

`List`	`R`	list [M,A]
`List`	`M`	list of list. Rows give relations.
`Array`	`A`	points in span, indexed by linear combinations of L
`Ell`	`E`	curve to work on
`List`	`L`	list of points on E

EXPLANATION. the returned "matrix" M is in row HNF. If the span is infinite, then this routine does not return (and will run out of memory).

Function EllGroup

Returns the group generated by installed generators.

SYNTAX. `A:=EllGroup(E);`

`Array`	`A`	group elements
`Ell`	`E`	Ell over finite field

EXPLANATION. The returned value is NOT a copy. Do not change it. This routine has memory. EllKernel is computed and remembered as well.

Function EllKernel

Returns the relation matrix on the installed generators

SYNTAX. `M:=EllKernel(E);`

`Mat`	`M`	matrix over Z; rows are relations
`Ell`	`E`	Ell over finite field.

EXPLANATION. If E is the reduction of a curve with installed generators, then this relation matrix gives the combinations that generate the kernel of reduction. This routine has memory, but the returned matrix is a copy of the stored data. EllGroup is computed and remembered as well.

Function EllGrpIndex

Computes EllSize(ec)/MatDet(EllKernel(ec)).

SYNTAX. `N:=EllGrpIndex(ec);`

`Int`	`N`	index
`Ell`	`ec`	Ell over finite field

EXPLANATION. Computes the index of the registered subgroup in the whole group. If ec is the reduction of a curve over a number field and the registered group is known to be a group of index prime to N in the Mordell-Weil group, then both groups have the same image under reduction.

Function EllPntApprox

Creates an approximate point on a localised Ell.

SYNTAX. `Q:=EllPntApprox(E,n,L);`
`Q:=EllPntApprox(E,n,x,y);`
`Q:=EllPntApprox(E,n,x,y,d);`
`Q:=EllPntApprox(P,n);`

`EllPntApprox`	`Q`	
`Ell`	`E`	localised Ell
`Int`	`n`	level of precision
`Ring element`	`x,y,d`	
`EllPnt`	`P`	point on a localised Ell

Function EllPntApproxAdd

Add approximate points with limited p-adic precision

SYNTAX. `R:=EllPntApproxAdd(P,Q)`

`EllPntApprox`	`P,Q,R`

EXPLANATION. The implementation of this routine does not work for all points. The routine can basically just add points that are distant from each other and from the neutral element. If P and Q does not satisfy this demand, it tries to translate one of the points by one of the installed generators. If this does not help, an error is generated. This is a weakness. Furthermore, we should have good reduction.

The resulting precision is the minimum of the precisions of P and Q
This routine is called if P+Q is evaluated.

Function EllPntApproxMul

Tries to approximate a multiple of an approximate point

SYNTAX. `Q:=EllPntApproxMul(n,P);`

`EllPntApprox`	`P,Q`
`Int`	`n`

EXPLANATION. Uses repeated doubling and EllPntApproxAdd. If EllPntApproxMul fails, then there may be addition chains that do lead to a valid answer.

Returned precision is that of P.
This routine is called if n*P is evaluated.

Function EllVecToPntApprox

Computes a p-adic approximation to a linear combination of the installed generators.

SYNTAX. `P:=EllVecToPntApprox(E,n,L)`

`EllPntApprox`	`P`	
`Int`	`n`	desired precision level
`List`	`L`	list of integers, giving linear combination.

EXPLANATION. This routine might fail, since it is based on the PntApprox addition and multiplication routines, which are not complete.

Function EllKerZ

Computes approximations to the logarithms of the generators of the kernel of reduction

SYNTAX. `L:=EllKerZ(E);`

`List`	`L`	list of approximations to elliptic logarithms
`Ell`	`E`	localised Ell

EXPLANATION. This routine takes EllKernel(E) and computes approximations to those linear combinations of the installed generators. It returns a list of the quotients of the x and y coordinates. The answer returned is defined modulo the place squared.

If EllKernel(ec) is particularly small, i.e. if the span of the registered points in ec span a particularly small group in reduction, then it might not be possible to find a suitable translation point in EllPntApproxAdd. For such groups, EllKerZ resorts to computing the elements in the kernel with exact precision. This is justified by the fact that then this procedure will probably not take too much time.

Function EllCoefList

Returns the coefficients in a list

SYNTAX. `L:=EllCoefList(E);`

`List`	`L`
`Ell`	`E`

EXPLANATION. The list is of the form $[\gamma, 0, a_2, 0, a_4, a_6]$.

Function EllXtoPnt

Returns a point with given X-coordinate

SYNTAX. `P:=EllXtoPnt(E,x);`

`EllPnt`	`P`
`Ell`	`E`
`Ring Element`	`x`

EXPLANATION. It is an error if x is not the x-coordinate of a rational point.

Function EllRandomX

produces a random X-coordinate of a point on the curve

SYNTAX. `x:=EllRandomX(ec);`

`Elt`	`x`	returned x-coordinate or false
`Ell`	`ell`	

EXPLANATION. Currently implemented only for localised elliptic curves. Future versions may include facilities for selecting the kind of point returned. Currently, it is not guaranteed that every kind of point has a chance of being returned. If the routine fails to find a point, false is returned.

Function EllIsIsom

Test if two elliptic curves are isomorphic over their rings of definition

SYNTAX. `B:=EllIsIsom(E1,E2)`

`Boolean`	`B`
`Ell`	`E1,E2`

EXPLANATION. This routine has memory for isomorphic curves

Function EllIsom

Returns an isomorphism between isomorphic curves

SYNTAX. `f:=EllIsom(E1,E2)`

`function | Boolean`	`f`	map from E1 to E2 or false
`Ell`	`E1,E2`	

EXPLANATION. The returned function either takes an EllPnt and returns one or takes a 3 element list(representing a point in the projective plane) and returns one. If no such map exists then false is returned. This routine has memory, so two calls with identical parameters give identical answers. The call EllIsom(E2,E1) returns the inverse map.

Function EllTorBound

Bounds size of the torsion subgroup by counting points in reduction

SYNTAX. `N:=EllTorBound(ec [,plclist]);`

```
Int   N
Ell   ec
List  plclist
```

Function EllPntMove

Moves a point to an isomorphic curve.

SYNTAX. `Q:=EllPntMove(P,E)`

`EllPnt`	`P`	
`EllPnt`	`Q`	the image of P on E
`Ell`	`E`	the curve to move to

EXPLANATION. In it an error if E is not isomorphic to the curve of P.

Function EllMWrank

calls MWrank for curves over Z and prints output.

SYNTAX. `EllMWRank(ec);`

```
Ell  ec
```

EXPLANATION. at the moment, the curve should be over Z. Gammas are taken care of.

Function Quar

Determine Weierstrass form of a curve of the form $Y^2 = Quartic(X)$

SYNTAX. `E:=Quar(P [, X | L][, Eref]);`

`Ell`	`E`	resulting E
`Poly`	`P`	quartic polynomial
`Elt`	`X`	x-coordinate for the point to use
`List`	`L`	list [x,y] describing (affine) point to use
`List`	`L`	list [x,y,d] projective point to use
`Ell`	`Eref`	reference E.

EXPLANATION. If no point is supplied, Quar tries points at infinity, points with X=0 and other points, in that order. It searches up to GlobalSearchBound using HypEllFindX. It is an error if Quar can't find a point.

The returned Ell has fields toquar, fromquar, toquarrev and fromquarrev, being functions that give maps between the quartic and the elliptic curve. The "from" functions try to return an EllPnt, if possible. Otherwise a triplet is returned (eg, for points with polynomial coordinates). The "to" functions accept both triplets and EllPnts. The "quar" function have to do with the quartic, the "quarrev" routines with the reversed quartic.

If Eref is supplied, then Quar checks if the curve is isomorphic to Eref. If it is, it equips Eref with the maps, overwriting possible present ones, and returns Eref. An error is generated if the curve is not isomorphic.

In general, Quar is not guaranteed to return an integral model. One situation in which it does, is

$$\mathrm{Quar}((1/a)x^3 + (b/a)x^2 + (c/a)x + (d/a))$$

with a, b, c, d integral.

Function Cubic

Initialises elliptic curve from cubic planar curve with a given point

SYNTAX. `E:=Cubic(P,L [,E2]);`

`Ell`	`E`	initialised curve
`Poly`	`P`	trivariate cubic homogeneous polynomial
`List`	`L`	homogeneous coordinates of point on curve
`Ell`	`E2`	prescribed curve for birationality test

EXPLANATION. The returned curve has a function E.tocub installed that maps points back to the cubic model. If E2 is specified, It is tested if E2 is birational to the cubic curve. If so, E2 with tocub installed (or overwritten) is returned. Otherwise, false is returned.

Function Ell2DivPol

Returns a 2-division polynomial

SYNTAX. `P:=Ell2DivPol(E,x0)`

`Ell`	`E`	elliptic curve
`Poly`	`P`	degree 4 polynomial in x
`List`	`x0`	x-coordinate of point to divide

EXPLANATION. Suppose (x, y) is a point on E such that $2(x, y) = (x0, y0) \neq \infty$. This routine returns a polynomial that x should satisfy.

Module ellcov.g

2001/7/30, Nils Bruin

This module implements a data structure EllCov, representing an elliptic cover of the projective line. For now, mainly double covers are supported.

Function IsEllCov

Test if an object is an EllCov.

SYNTAX. `B:=IsEllCov(obj)`

```
Boolean  B
object   obj
```

Function IsEllDblCov

Test if an object is an Ell Double cover (optimised form of an EllCov)

SYNTAX. `B:=IsEllDblCov(obj)`

```
Boolean  B
object   obj
```

Function EllCov

Constructs a generic EllCov

SYNTAX. `cov:=EllCov(ec,f)`

`EllCov`	`cov`	
`Ell`	`ec`	
`List`	`f`	pair of trivariate homogeneous polynomials

EXPLANATION. Constructs an EllCov that uses EllFunEval to compute values. f describes the covering map.

Function EllDblCov

Creates an EllCov of degree 2 with given coefficients for the cover

SYNTAX. `cov:=EllDblCov(ec,c)`

`EllCov`	`cov`	
`Ell`	`ec`	
`List`	`c`	List [[c11,c12,c13],[c21,c22,c23]]

EXPLANATION. The returned data structure represents the cover from the curve to the projective line given by

$$(x, y) \mapsto \frac{c_{11}x + c_{12}y + c_{13}}{c_{21}x + c_{22}y + c_{23}}$$

Function QuarCov

Returns a degree 2 EllCov representing the X-map on $Y^2 = Quartic(X)$.

SYNTAX. `cov:=QuarCov(P,L [,ec]);`

`EllCov`	`cov`	
`Poly`	`P`	valid input for Quar
`Elt | List`	`L`	point description on quartic passed to Quar
`Ell`	`ec`	prescribed elliptic curve to use

EXPLANATION. Basically, this routine is a wrapper around Quar. It returns an EllCov instead of an Ell, thus capturing more of the original quartic structure. Since the EllCov data structures are mainly for elliptic curve Chabauty methods, it is very important that the point L chosen satisfies certain rationality conditions. Therefore, it is mandatory to prescribe it (or at least the X-coordinate)

Function EllCovRed

Returns an EllCov reduced at a place of good reduction.

SYNTAX. `redcov:=EllCovRed(cov,plc)`

`EllCov`	`redcov`	cover over residue field
`EllCov`	`cov`	cover over number field
`Place`	`plc`	place to reduce at

Function EllCovFibRed

returns the fibers above the points over the primitive field.

SYNTAX. `L:=EllCovFibRed(cov);`

`List`	`L`	list [M,L'] of a lattice and list describing fibers
`EllCov`	`cov`	cover over finite field

EXPLANATION. L is a list of vectors wrt. installed gens in ec. Positions 1 .. p-1 contain the fibers of 1 .. p-1. 0 is encoded as p and infinity as p+1. Note that these vectors should be interpreted "mod EllKernel", which is M.

Function EllCovFibTranslate

Translates a value of one of the Fiber routines in a more readable answer

SYNTAX. `L2:=EllCovFibTranslate(L1)`

`List`	`L1`	list as given by EllCovFibers
`List`	`L2`	list [p,[x-coordinates (projective)]]

Function EllCovFibStrict

Returns result of intersection argument on rational fibers of equal characteristic.

SYNTAX. `L:=EllCovFibStrict(cov,places);`

`List`	`L`	list L with same format as for EllCovFibRed
`EllCov`	`cov`	
`List`	`places`	list of places of equal characteristic

EXPLANATION. Note that, since the returned lattice is the intersection of the kernels of reduction, fibers might actually contain more than 2 points mod the lattice.

Function EllCovFibers

uses local intersection arguments to bound fibers containing points with rational image under phi.

SYNTAX. `L:=EllCovFibers(cov,plclist1 [,plclist2]);`

`List`	`L`	list [M,L'] of lattice and fibers (in vectors)
`EllCov`	`cov`	
`List`	`plclist1`	list of places of same characteristic
`List`	`plclist2`	list of auxiliary places

EXPLANATION. This routine calls EllCovFibStrict for plclist1. It sorts plclist2 by characteristic and does an EllCovFibStrict with that data as well. It then discards vectors from the first result that are not congruent (mod the appropriate lattice) to any of the vectors in such a second EllCovFibStrict. Therefore, adding places to plclist2 will not change the returned lattice, but might reduce the number of vectors in the returned fiber list.

Function EllCovFibSect

Does an intersection argument on "Fiber" data structures.

SYNTAX. `L3:=EllCovFibSect(L1,L2);`

`List`	`L1,L2,L3`	fiber data structure

EXPLANATION. Given two fiber data structure L1,L2 (output from EllCovFibers or EllCovFibStrict), returns a new structure derived from L1, where entries that do not agree with L2 (for congruence reasons), are removed. This routine allows finer control that EllCovFibers, as sometimes, intersecting two auxiliary structures before intersecting with the main one may yield a sharper result than just intersecting with the main structure.

Function EllZB

Returns matrix of elliptic logarithms of intersections of kernels of reduction.

SYNTAX. `M2:=EllZB(ec,L[,M1]);`

`Mat`	`M2`	matrix over cov.ec.R, rows are EllKerZ's
`Ell`	`ec`	
`List`	`L`	list of places
`Mat`	`M1`	sublattice of kernels of reduction

Function EllCovTheta

Computes the "magic" Theta power series that bounds the number of points with rational image under phi near G.

SYNTAX. `L:=EllCovTheta(cov,plclist,G)`

`List`	`L`	list describing linear or quadratic approximation
`EllCov`	`cov`	
`List`	`plclist`	list of places of same characteristic
`EllPnt`	`G`	point of curve belonging to cov.

Function QuadHasSols

given a list of matrices over a finite field, describing quadratic forms, determines if these simultaneously represent 0 in a non-trivial way.

SYNTAX. `B:=QuadHasSols(L);`

`Boolean`	`B`	
`List`	`L`	List of matrices over a finite field

Function EllCovThetaTest

Tests whether EllTheta around G proves that G is only point with rational phi in its fiber of reduction.

SYNTAX. `B:=EllCovThetaTest(cov,L,G)`

`Boolean`	`B`	
`EllCov`	`cov`	
`List`	`L`	list of places of equal characteristic
`EllPnt`	`G`	Point to look around

Function EllCovChab

This routine checks if the full set of rational x-coordinates (or the full set of points which map to the rational points) for the given cover is known.

SYNTAX.
```
B:=EllCovChab(cov [,xlist |
  ,pntlist],placelist1[,placelist2]]);
```

`EllCov`	`cov`	
`List`	`xlist`	list of x-coordinates
`List`	`pntlist`	list of points on cov.ec
`List | elt | ideal`	`placelist1`	
`List | elt | ideal`	`placelist2`	

EXPLANATION. The format for xlist is quite lenient - the string "infinity" or "infty" is understood to mean the point at infinity, rationals and integers are interpreted as affine coordinates and a list if considered a projective point ([1,0] denoting the point at infinity).

Specifying the rational fibers downstairs using xlist is only allowed for EllDblCov covers. If an xlist nor a pntlist is specified, then this routine uses EllCovSearch to find points with rational image.

If plclist1 is specified as a list of places, then the first place determines the characteristic for the chabauty-computations. Places of a different characteristic in plclist1 will be added to plclist2.

Module ellform.g

2000/1/31, Nils Bruin

This package implements several routines that have to do with formal power series and elliptic curves.

Function EllPSR

Returns a power series based on a univariate polynomial algebra over the ring of definition of an elliptic curve.

SYNTAX. `R:=EllPSR(ec);`

```
PowSerRing  R
Ell         ec
```

EXPLANATION. This routine has memory. When called repeatedly with identical elliptic curves, then the answer is identical. If called for the first time, the ring is created with variable name z. Precision with which the ring is created is $EllPSR_Prec + 3$, where $EllPSR_Prec$ is a global variable. Note that, once called, the ring precision cannot be changed.

Since it is convenient that power series can have denominators in their coefficients, the base ring for R is Q if ec is over Z.

Function EllW

Returns power series solution $W = z^3 * (1 + R(z))$ to $gamma * W = z^3 + a2 * z^2 * W + a4 * z * W^2 + a6 * W^3$.

SYNTAX. `S:=EllW(ec [,prec]);`

```
PowSer  S
Ell     ec
int     prec
```

EXPLANATION. If prec is present, then an answer of that precision (if EllPSR allows it) is returned. Otherwise, an answer of maximal precision is returned (well ... strictly, the precision could be a little bit higher) This call uses EllPSR, so if you want a non-standard PSR associated to ec, call EllPSR first.

Function EllOmega

Returns a power series expansion of a normalised invariant differential on ec.

SYNTAX. `S:=EllOmega(ec);`

```
PowSer  S
Ell     ec
```

EXPLANATION. If $w = EllW(ec)$ then $EllOmega(ec) = -(w - z * dw)/w/2$.

Function EllPowSerPoint

Returns a neighbourhood on ec around the given point by translation of the point $[z : -1 : EllW(ec)]$.

SYNTAX. `S:=EllPowSerPoint(ec,[ P | L ]);`

`List`	`S`	3 components describing homogeneous coordinates.
`Ell`	`ec`	
`EllPnt`	`P`	
`List`	`L`	either [x,y] or [x,y,d]

EXPLANATION. Note that evaluation of EllPowSerPoint in $z = EllExp(ec)$ gives a useful parametrisation of the kernel of reduction. The components of S are not necessarily power series.

Function EllLog

returns p-adic elliptic logarithm.

SYNTAX. `S:=EllLog(ec);`

`PowSer`	`S`
`Ell`	`ec`

EXPLANATION. Returns PowSerInit(EllOmega(ec)).

Function EllExp

returns inverse of EllLog (map-inverse, i.e. PowSerMapInv)

SYNTAX. `S:=PowSerExp(ec);`

`PowSer`	`S`
`Ell`	`ec`

Function EllFunEval

Evaluates a rational function on an elliptic curve, resolving singularities if necessary.

SYNTAX. `v:=EllFunEval(ec,f,[P | L]);`

`object`	`v`	
`Ell`	`ec`	
`List`	`f`	pair of trivariate homogeneous polynomials.
`EllPnt`	`P`	
`List`	`L`	list [x,y] or [x,y,d]

EXPLANATION. If evaluation of f in P leads to [0,0], then f is evaluated in PowSerPoint(ec,P), common factors of z are divided out and the resulting pair of power series is evaluated in $z = 0$. Obviously, singularities of too high order make the system run out of precision, which results in the value "undefined".

Module ideal.g

2000/11/4, original by Claus Fieker, extended by Nils Bruin

In order to be able to use Z as an order and ZIdeals as proper Ideals, one has to wrap some of the routines for orders, ideals and elts. The implementations for Z are mostly trivial.

The wrapped routines have an X added in front of their name. This package evolved on an "as needed" basis, so it is not complete and error checking need not be very good either.

The routines themselves are not documented. They are simple enough to be self-explanatory and the user interface is that of the wrapped routine.

The following generic functions are overridden by a new implementation: Valuation, Factor a "mod" field is added to _ZIdealOperations

Module latcalc.g

1999/8/28, Nils Bruin

Some routines to work with subspaces of Z^r. Lattices are represented as rowspans of matrices over Z.

Function LatHermite

Returns upper row hermite normal form (i.e. a canonical basis of a lattice)

SYNTAX. `M2:=LatHermite(M1);`

`Mat  M1,M2`

Function LatSum

Returns the joint span of two lattices

SYNTAX. `M3:=LatSum(M1,M2);`

`Mat  M1,M2,M3`

Function LatSectExt

Returns the intersection of two lattices in relative bases to the original bases.

SYNTAX. `L:=LatSectExt(M1,M2);`

`List  L` list of two matrices [I1,I2]

EXPLANATION. Compute intersection of 2 lattices and return representations with respect to the bases for M1 and M2. in the following way: Stack M1 and -M2 and compute the kernel M3 of this matrix. The first Rows(M1) columns span the linear combinations of M1 that are also expressible with respect to M2 (thus, the intersection). The same for the last Rows(M2) columns of M3.

Function LatContainsVec

Tests if a vector is in the span of a lattice

SYNTAX. `B:=LatContainsVec(M,L);`

`Boolean`	`B`	
`Mat`	`M`	
`List`	`L`	vector

Function LatSect

Returns intersection of two lattices

SYNTAX. `M3:=LatSect(M1,M2);`

`Mat  M1,M2,M3`

Function LatRelBasis

Express sublattice relative to another lattice.

SYNTAX. `M3:=LatRelBasis(M1,M2);`

`Mat`	`M3`	matrix such that M3*M2=M1
`Mat`	`M1`	sublattice of M2
`Mat`	`M2`	lattice containing M1

EXPLANATION. M2 should be of maximal rank in ambient space (i.e. invertible over Q)

Function LatChinRem

Returns a representative satisfying two congruences modulo lattices

SYNTAX. `v3:=LatChinRem(v1,M1,v2,M2)`

`Boolean | List`	`v3`	lifted representative or false
`List`	`v1,v2`	lists representing vectors
`Mat`	`M1,M2`	matrices giving congruence lattices

EXPLANATION. The returned vector should be interpreted mod LatSect(M1,M2). false designates that v1 mod M1 and v2 mod M2 do not lift to one vector, which is equivalent with the fact that v1-v2 does not lie inside LatSum(M1,M2).

Function LatUniqRep

Returns a list without double representatives modulo a lattice

SYNTAX. `L2:=LatUniqRep(M,L1);`

`List`	`L2`	list of vectors, unique mod M
`Mat`	`M`	lattice
`List`	`L1`	list of vectors to choose L2 from

Function MatDecompose

Given a matrix over an order O, returns Degree(O) matrices over Q that form the decomposition of M over the Z-basis of O.

SYNTAX. `L:=MatDecompose(M)`

`List`	`L`	List of matrices over Z
`Mat`	`M`	Matrix over some order O

Module matalg.g

1999/8/22, Nils Bruin

This module implements some routines to work with vector spaces, represented as row spaces of matrices. Note that, while matrices over Z and other rings that are not fields are usually accepted, but not all routines will return sensical answers.

Function MatRing

Determines the ring over which a matrix is defined.

SYNTAX. `R:=MatRing(M);`

`Ring`	`R`	Ring of definition of M
`Mat`	`M`	

EXPLANATION. Although this routine might seem quite straightforward, it is not, and it needs quite some operations to be failsafe for matrices over Z and Q and for matrices with no rows or columns. Therefore, try to avoid using this routine in those situations.

Function MatOrth

Returns the orthogonal space

SYNTAX. `B:=MatOrth(A);`

`Mat A,B`

EXPLANATION. Implemented as the kernel of the transpose.

Function MatUnion

Returns the union of two spaces

SYNTAX. `C:=MatUnion(A,B);`

`Mat`	`A,B`	Matrices defined over same ring
`Mat`	`C`	Rows of A and B stacked together.

Function MatBasis

Returns a matrix with a minimal number of rows with the same row space

SYNTAX. `A:=MatBasis(B);`

`Mat A,B`

EXPLANATION. implemented as the MathOrth of the MatOrth.

Function MatIntersect

Returns intersection of row spaces.

SYNTAX. `C:=MatIntersect(A,B);`

```
Mat  A,B,C
```

EXPLANATION. implemented as MatOrth of the union of the MatOrths

Function MatCatDiag

Constructs the "direct sum" of two matrices.

SYNTAX. `C:=MatCatDiag`

```
Mat  A,B,C
```

EXPLANATION. put A and B on the diagonal and fills the rest with zeroes.

Function MatIsZero

Determines whether A is a zero matrix

SYNTAX. `B:=MatIsZero(A);`

```
Boolean  B
Mat      A
```

Module place.g

1999/9/3, Nils Bruin

This module implements a new data structure, a *Place*. It is a data structure mainly suited for representing the multiplicative group of a completion of a number field mod squares. There are also some routines to facilitate finite precision p-adic computations.

The data structure is implemented as a "rec" with the following data fields.

`String`	`Type`	Constant "Place"
`Rec`	`operations`	Constant _PlcOps
`Elt`	`unif`	a uniformising element
`List`	`filtr`	a list with stored powers of the prime ideal
`Integer`	`val4p1`	Valuation(filtr[1],4)+1
`Integer`	`primebelow`	Characteristic of residue field
`Integer`	`ramdeg`	Degree of ramification of prime ideal

This module depends on the global variable F2:=FF(2) and sets this if unset. This module depends on matalg.g

Function IdealRepBounds

Returns a MultiCounter bounds vector for enumerating residue classes mod an ideal, relative to the basis of the order of definition of the ideal.

SYNTAX. `B:=IdealRepBounds(I);`

`List of integers`	`B`	The bound vector
`Ideal`	`I`	the ideal

EXPLANATION. Returns the diagonal of the Hermite Normal Form of I.

Function PlaceOrder

Returns order of definition of a place.

SYNTAX. `O:=PlaceOrder(plc);`

`Order`	`O`	order of definition
`Place`	`plc`	the place

Function PlaceInit

Initialises a Place data structure

SYNTAX. `Plc:=PlaceInit(I);`

`Place`	`Plc`	The new place
`Ideal`	`I`	The prime ideal at the place.

EXPLANATION. PlaceInit has a memory, so two calls with equal I return identical places.

Function IsPlace

Type indicator for Place data structures.

SYNTAX. `B:=IsPlace(obj);`

`Boolean`	`B`	Result
`Object`	`obj`	the object of which the type is to be determined

Function PlaceCompare

Equality test for Places

SYNTAX. `B:=PlaceCompare(plc1,plc2);`

`Boolean`	`B`	result
`Place`	`plc1,plc2`	the places to be compared

EXPLANATION. Places are considered equal if the prime ideals they are based on test as equal. This routine is used for "plc1=plc2".

Function PlaceMod

Reduction at a Place

SYNTAX. `r:=PlaceMod(obj,plc);`

`object`	`r`	reduced object
`object`	`obj`	object to be reduced
`Place`	`plc`	place at which to do reduction

EXPLANATION. This routine tries to give a generic behaviour for "reduction". If obj is a record with a operations field containing a "placemod" field, then obj.operations.placemod(obj,plc) is returned. For an Elt or an Int, EltToFFE is used. For lists, PlaceMod is applied to the elements. Otherwise, "obj mod plc.filtr[1]" is tried. The latter is only a fudge to give some meaning. "mod Place" should give a result over the residue field, while "mod Ideal" usually gives a result in the same ring. This routine is used for "obj mod plc".

Function PlaceGetIdeal

Get a power of the prime ideal associatied to a place

SYNTAX. `I:=PlaceGetIdeal(plc,n);`

`Ideal`	`I`	returned ideal
`Place`	`plc`	place
`Integer`	`n`	power of ideal to be returned

EXPLANATION. If called twice with the same parameters, this routine is guaranteed to return *identical* ideals. Any precomputed data, like that produced by RayResidueRing, will carry through. This is important for computations that rely on the choice of basis. If this does not make sense to you: Just use this routine if you need a power of the prime deal belonging to a place.

Function PlacePrime

Get underlying prime ideal of a place

SYNTAX. `I:=PlacePrime(plc);`

`Ideal`	`I`	prime ideal
`Place`	`plc`	place at prime.

EXPLANATION. Equivalent to place.filtr[1]. The latter is implementation dependent, though.

Function PlaceVal

Valuation of an element at a place

SYNTAX. `n:=PlaceVal(plc,elt);`

`Integer`	`n`	valuation
`Place`	`plc`	Place at which to take valuation
`Elt`	`elt`	object to take valuation of.

EXPLANATION. This routine is basically equivalent to Valuation(PlacePrime(place),elt), except that if elt=0, the value of the preset global variable PlaceValZeroVal is returned instead of the non-numerical "infinity" that Valuation returns. This variable is preset to 32767, which is usually big enough to pass for "infinity" in practice.

Function PlaceSqrRep

Representation of an element of a localised number field mod squares

SYNTAX. `L:=PlaceSqrRep(plc,elt);`

`List`	`L`	List of integers in $\{0,1\}$
`Place`	`plc`	Place at which to take representative mod squares
`Elt`	`elt`	Element to represent.

EXPLANATION. Let K be a number field, K_p its localisation and K_p^* the multiplicative group. Then this routine returns a vector of the F_2-vector space $K_p^*/(K_p^*)^2$. The first coordinate is the parity of the valuation. The rest depends on choice of basis (well, at characteristic 2 places). The choice of basis is guaranteed to be the same for different calls with the same place, but not for calls with non-identical but equal places (i.e. choices may vary across different runs)

Function PlaceSqrRepGens

Returns the choice of basis for PlaceSqrRep

SYNTAX. `L:=PlaceSqrRepGens(plc);`

`List`	`L`	List of elements
`Place`	`plc`	Place to give generators for.

EXPLANATION. Returns a lift of a basis for $K_p^*/(K_p^*)^2$ (elements of O_K). This is the basis used by PlaceSqrRep.

Function InfSqrRep

Square representation at infinite places.

SYNTAX. `L:=InfSqrRep(i,elt);`

`List`	`L`	[0] or [1]
`Int`	`i`	i-th archimedean embedding
`Elt`	`elt`	element to represent

EXPLANATION. Returns [1] if i is a real embedding and elt is negative (i.e. not a square) under that embedding. Returns [0] otherwise. (also for complex embeddings) The routine is mainly to accompany PlaceSqrRep.

Function PlaceIsSqr

Tests if an element is a square locally.

SYNTAX. `B:=PlaceIsSqr(plc,elt);`

`Boolean`	`B`	result
`Place`	`plc`	Place to localise at
`Elt`	`elt`	element to consider

EXPLANATION. This routine considers 0 to be a square. For other values, it tests if PlaceSqrRep returns a zero vector.

Function IsLocSqr

Determines if a p-adic approximation is a square.

SYNTAX. `n:=IsLocSqr(plc,l,elt);`

`Int`	`n`	-1, 0 or 1
`Place`	`plc`	Place to localise at
`Int`	`l`	level of approximation
`Elt`	`elt`	approximation.

EXPLANATION. elt is an approximation to an element with the same image mod PlaceGetIdeal(plc,l). If all such elements are squares, then $n = 1$. If all such elements are non-squares, then $n = -1$. Otherwise, $n = 0$ is returned.

elt=0 is special. The result returned depends on ILS_zero_is_precise. If this variable is false (this is the default), then $n = 0$ is returned. This corresponds to the explanation above. However, in some situations, it is useful to consider elt=0 (as input! not just mod PlaceGetIdeal(plc,l)) as precise, rather than just as an approximation. Then $n = 1$ should be answered. This happens if ILS_zero_is_precise is set to true.

Function IntIsLocSqr

Determines if an approximation to a p-adic number is a square.

SYNTAX. `n:=IntIsLocSquare(p,l,a);`

`Int`	`n`	in $\{-1, 0, 1\}$
`Int`	`p`	prime number
`Int`	`l`	level of approximation
`Rational`	`a`	approximation

EXPLANATION. This routine mimics IsLocSqr for rational numbers. See that routine for more information. ILS_zero_is_precise has the same function here.

Function FFEToInt

Lifts an element of a primitive finite field to an integer

SYNTAX. `n:=FFEToInt(e);`

`Int`	`n`	the lift
`FFElt`	`e`	element of a primitive finite field

Function SelectPrimPlaces

Returns a list of places of inertia degree 1 of a given order, avoiding a given discriminant.

SYNTAX. `L:=SelectPrimPlaces(O,n,D);`

`List of places`	`L`	list of length n of primitive places of O
`Order`	`O`	order in which to look
`Int`	`n`	number of places to select
`Elt`	`D`	element required to be a unit at selected places

EXPLANATION. This routine finds the n smallest primitive places of O such that D has valuation 0 there. This is exactly what is tested, so the requirement that D is an element can be relaxed to the requirement that Valuation(p,D) should be well defined for prime ideals p of O.

Function PlaceSupport

Returns a list of places at which the given object is not a unit.

SYNTAX. `L:=PlaceSupport(elt);`
`L:=PlaceSupport(I);`

`List`	`L`	list of places
`Elt`	`elt`	
`Ideal`	`I`	

Function PlacePointNormal

Normalises a padic projective point

SYNTAX. `L:=PlacePointNormal(plc,M);`

`List`	`L`	normalised list of Elts
`Place`	`plc`	place at which to work
`List`	`M`	list of Elts

EXPLANATION. Does a scalar multiplication on M such that the minimum valuation of the resulting list is exactly 0. Note that even after normalisation, representations of the same projective point might still differ by a unit scalar factor.

Function PlaceClassRepP

Returns a representation of a place in $Pic^0/p\,Pic^0$

SYNTAX. `L:=PlaceClassRepP(p,plc);`

`List`	`L`	vector of numbers mod p
`Int`	`p`	prime number
`Place`	`plc`	

Function PlaceClassExtendP

Extends an S-basis of places such that $Pic^0/p\,Pic^0$ is generated.

SYNTAX. `S2:=PlaceClassExtendP(p,S1);`

`List`	`S2`	list of places
`Int`	`p`	prime
`List`	`S1`	list of places

Function SquareBasis

Computes a basis of K(S,2).

SYNTAX. `B:=SquareBasis(S [,"fast"]);`

`List`	`S`	list of Places
`List`	`B`	list of Elts

EXPLANATION. This routine computes the torsion unit, a 2-maximal set of independent units and uniformisers of the given places and returns them as B. It checks if the class number of the order is odd and it is an error if it is not. Even class number may be supported by future versions.

If the option "fast" is included, some checks for P-maximality are not done.

Function PlacesSqrRep

PlaceSqrRep at several places simultaneously, together with InfSqrReps.

SYNTAX. `L:=PlacesSqrRep(P,elt);`

`List`	`L`	list of $\{0,1\}$
`List`	`P`	list of places
`Elt`	`elt`	element to represent

EXPLANATION. returns the concatenation of InfSqrRep(i,elt) and PlaceSqrRep(plc,elt) for i running through the real embeddings and plc running through the places in P.

Function PlaceListInjSqr

Returns a list of places to facilitate checking relations mod squares over a fixed basis.

SYNTAX. `L:=PlaceListInjSqr(M);`

`List`	`L`	list of Places
`List`	`M`	list of Elts, independent mod squares.

EXPLANATION. Computes a list of places such that PlacesSqrRep gives a set of independent vectors if applied to M.

Function EltIsSquare

Determines if an element of a ring is square.

SYNTAX. `B:=EltIsSquare(obj);`

`Boolean`	`B`	result
`Ring element`	`obj`	object to be examined.

EXPLANATION. Checks if the polynomial $x^2 - obj$ is irreducible. Therefore, obj should not be a polynomial.

Function PolyRoots

Returns the roots (with multiplicity) of a polynomial over its ring of definition

SYNTAX. `L:=PolyRoots(P [,"nomult"]);`

`List`	`L`	list of roots
`Poly`	`P`	

EXPLANATION. If the flag "nomult" is passed, then roots are returned without multiplicity.

Function PlacePolyRoots

returns approximations to p-adic roots of a polynomial.

SYNTAX. `L:=PlacePolyRoots(plc,f,prec)`

`List`	`L`
`Place`	`plc`
`Poly`	`f`
`Int`	`prec`

EXPLANATION. No fancy newton approximation, but does work regardless of ramification. Only returns *integral* roots.

Function PlacePolyFactor

returns approximation to p-adic factorisation of a polynomial

SYNTAX. `L:=PlacePolyFactor(plc,f,prec)`

`List`	`L`	list [[f1,1],...[fn,1]]
`Place`	`plc`	
`Poly`	`f`	
`Int`	`prec`	

EXPLANATION. Current implementation only works if PlacePolyRoots gives enough information on how f factors. That means that f should be seperable and is of degree ¡= 3 or has at least PolyDeg(f)-2 p-adic roots. Furthermore, prec should be high enough to distinguish separate roots. If prec¿PlaceVal(plc,PolyDisc(f)), things should be safe. It is an error if f is not monic with locally integral coefficients.

Function PlaceDescend

Descends a place to a suborder.

SYNTAX. `Plc2:=PlaceDescend(Plc1,O);`

`Place`	`Plc1,Plc2`
`Order`	`O`

EXPLANATION. Kash must be able to lift ideals from O to the ring of definition of Plc1.

Module pntsrch.g

2000/1/21, Nils Bruin

This module implements several routines for searching for solutions of polynomial equations, both local and global. One of the main features is HypEllFindX, which interfaces to an external, high performance sieving program to search for solutions to equations of the form $Y^2 = F(X)$ for X, Y in some order.

This module requires place.g

Function PolyDim

Returns number of variables of multivariate polynomial P.

SYNTAX. `d:=PolyDim(P);`

`Int`	`d`	
`Poly`	`P`	multivariate polynomial

Function PolyVars

returns a list of variables of multivariate polynomial P

SYNTAX. `L:=PolyVars(P);`

`List`	`L`	List of polynomial variables
`Poly`	`P`	multivariate polynomial

EXPLANATION. The returned variables are ordered with increasing dimension. The returned variables are actually polynomials of polynomial algebras of increasing dimension.

Function Meval

Evaluate multivariate polynomial in a point

SYNTAX. `obj:=Meval(P,L);`

`object`	`obj`	resulting value
`Poly`	`P`	multivariate polynomial
`List`	`L`	list of values for variables

EXPLANATION. If Length(L)<PolyDim(P) then the result still has the unassigned variables in it. It is an error if Length(L)>PolyDim(P).

Function PolyDecomp

Decompose a multivariate polynomial into abstract sparse representation

SYNTAX. `L:=PolyDecomp(P);`

`Poly`	`P`	multivariate polynomial
`List`	`L`	List of [coeff, exponent list]

EXPLANATION. The returned value is a list of pairs [cf, exponent list], where cf is a non-zero coefficient of the monomial in P with the exponents given in the list. The order of the exponent vector is compatible with the other multivariate polynomial routines, i.e. outermost variable last.

The returned representation is called "abstract" because the variables are not explicitly mentioned - only the exponents.

Function PolyRecomp

Evaluate abstract sparse multivariate polynomial in a point.

SYNTAX. `P:=PolyRecomp(L,V);`

`List`	`L`	list of [coeff,exponent list]
`List`	`V`	list of specialisation values
`obj`	`P`	resulting object

EXPLANATION. This routine basically is an Meval for abstract sparse polynomials. V is assumed to be of sufficient length. Extra elements at the end are ignored. (this is quite contrary to the specifications of Meval. This is because in abstract representation there is no ambiguity about which variables are part of the base ring and which are to be considered polynomial variables). Obviously, by evaluating an abs. sparse polynomial in a vector of polynomial variables, we obtain a standard kash-representation of the polynomial.

Note that, as in other evaluation routines, it is left to KASH to determine how multiplications, additions and exponentiations are to be carried out.

Function PolyCoefDecomp

Decomposes a multivariate polynomial over an order with respect to its basis.

SYNTAX. `L:=PolyCoefDecomp(P,V);`

`Poly`	`P`	Multivariate Poly over an order
`List`	`V`	variable list
`List`	`L`	list of multivariate polynomials in V

EXPLANATION. Given a polynomial P over an order O, splits the coefficients with respect to the basis of O (using EltToList) and returns a list of polynomials with these split coefficients and the variables in V. V is not required to be a variable list, so this routine could be used as some twisted evaluation routine. However, it is NOT the same as EltToList(EvalM(P,[Elt(O,v)])). it would be if v in Z, though.

Also, note that PolyCoefDecomp(P,PolyVars(P)) will usually NOT have the desired effect. Although in value, the result will be correct, the returned polynomials will only be defined over O, not the base ring of O. This is because the variables of P are considered to be polynomials over O.

Function HasNoLocalPoint

Test if projective variety has local points.

SYNTAX. `B:=HasNoLocalPoint(P,L,N)`

`Boolean | String`	`B`	result or "unknown"
`Place`	`P`	Place to localise at
`List`	`L`	List of multivariate homogeneous polynomials
`Int`	`N`	Maximum level to look

EXPLANATION. Current implementation only tries to prove there are no points (returns true). In the current implementation, false is never returned. The used algorithm just tries to systematically find points that let all polynomials vanish to increasingly higher valuations at P, with maximum level N. Therefore, if "true" is returned for some value of N=N0, the run-time will be independent of N for $N > N0$.

If L is a polynomial instead of a list, then it is assumed that it is homogeneous and describes a hypersurface, i.e., it is automatically made into a list [L].

Function HypEllFindLocRatX

tries to find a rational X such that $F(X)$ is a square locally

SYNTAX. `L:=HypEllFindLocRatX(P,F)`

`List`	`L`	List describing found coordinates
`Place`	`P`	Place to localise at
`Poly`	`F`	polynomial over an order

EXPLANATION. Tries to find rational values for X such that F(X) is a local square at P. Returns a list in the form [...,[u,e],...], such that u mod p^e is a suitable environment for X. If e<0, then 1/X should be in u mod p^{-e} routine fails if poly has a zero in Q_p that is not in Q.

If ILS_zero_is_precise=true then zeroes in Q are handled correctly, because in FLRX_rec, only exact values are fed to IsLocSqr, which will then consider 0 to be a square regardless of level. This is set.

Function HypEllFindLocX

Try to find local solutions to $Y^2 = F(X)$

SYNTAX. `L:=HypEllFindLocX(P,F[,true])`

`List`	`L`	List describing found coordinates
`Place`	`P`	Place to localise at
`Poly`	`F`	polynomial over an order

EXPLANATION. Tries to find values for X such that poly(X) is a local square at plc. Returns a list in the form [...,[u,e],...], such that u mod p^e is a suitable environment for X. If e<0, then 1/X should be in u mod p^{-e} routine fails if poly has a zero in Q_p that is not in Q.

If ILS_zero_is_precise=true then zeroes in Q are handled correctly, because in FLRX_rec, only exact values are fed to IsLocSqr, which will then consider 0 to be a square regardless of level.

If a third parameter "true" is passed, then only one point is returned, if found.

This routine is especially optimised for equations of the form $Y^2 = f_4X^4 + f_2X^2 + f_0$.

Function HypEllFindX

Search for global solutions to $Y^2 = F(X)$

SYNTAX. `L:=HypEllFindX(F,B,N);`

`List`	`L`	list of found x-coordinates
`Poly`	`F`	polynomial over an order
`Int`	`B`	search bound
`Int`	`N`	number of filters to employ

EXPLANATION. This routine interfaces to an external high performance sieving routine. The order of definition of F is taken. Basically all linear combinations of the basis of that order are taken, with coefficients bounded by $-B, \ldots, B$. We look at N primitive places first to speed up search. N=50 generally gives good results. The global variable _FLTcommand should contain the proper string to invoke the "filter" program with and _FLTin and _FLTout should contain the names of files that can be used for communication.

Function HypEllLocRatXTest

Tests if a set of polynomials take a square value (locally) somewhere in a point in the corresponding Q_p.

SYNTAX. `B:=HypEllLocRatXTest(plc,L);`

`Boolean`	`B`	true is a value has been found; false otherwise.
`Place`	`plc`	
`List`	`L`	list of polynomials

EXPLANATION. The algorithm used is an adaptation of HypEllFindLocRatX.

Function FilterTwists

Returns a list of deltas such that HypEllLocRatXTest(plc,[delta*R,delta*Q]) returns true at all places above PolyResultant(R,Q).

SYNTAX. `L:=FilterTwists(R,Q);`

`List`	`L`	List of values for delta
`P`	`Poly:  R,Q`	

EXPLANATION. The values for delta that are tried come from SquareBasis on 2*PolyResultant Currently, infinite places are not considered. (but in fact they should be)

Function HypEllSize

Returns number of points on a hyperelliptic curve over a finite field.

SYNTAX. `N:=HypEllSize(P);`

`Int`	`N`	number of points
`Poly`	`P`	polynomial over finite field

EXPLANATION. The hyperelliptic curve considered is given by $Y^2 = P(X)$. If the degree of P is odd, 1 point at infinity is supposed. Otherwise, if the leading coefficient of P is a square, 2 points at infinity are counted, otherwise 0. This corresponds to the number of points on the smooth model. (if P is square free)

Function HypEllJacSize

returns number of points on the jacobian of the hyperelliptic curve

SYNTAX. `N:=HypEllJacSize(P);`

`Int`	`N`	number of points
`Poly`	`P`	polynomial of finite field

EXPLANATION. only implemented for genus 1 and 2 at the moment.

Function HypEllRev

Reverses a hyperelliptic curve (i.e. applies the substitution x-¿1/x) and returns a new integral model)

SYNTAX. `Q:=HypEllRev(P)`

`Poly  P,Q`

EXPLANATION. Takes care of odd degree polynomials in the proper way.

Module powser.g

2000/1/31, Nils Bruin

Implementation of rudimentary power series support. Power series rings are based on a polynomial ring and have a precision associated to them. They are implemented as "floating precision", so that precision is the maximum difference in degree between highest and lowest non-zero coefficient.

An example: if precision is 2, then $1+x^2$ and 1 are represented by the same power series, but x^3 and x^3+x^4 are distinguishable.

Elements have a precision associated to them as well, which does not have to be the maximum precision of the ring.

All computations are done as precise as possible, unless truncation intermediate results require a precision higher than is allowed in the ambient ring. In any case, the precision of the returned answer is correct.

PowSerRing is a record with the following fields:

`string`	`Type`	"PowSerRing"
`Poly`	`T`	degree 1 monomial of base PolyAlg
`ring`	`R`	coefficient ring of PolyAlg
`int`	`maxprec`	maximum precision

PowSer is a record with the following field:

`string`	`Type`	"PowSer"
`rec`	`operations`	methods record
`PowSerRing`	`PowSerRing`	ambient ring
`int`	`prec`	precision
`Poly`	`P`	value
`int`	`e`	shift

So, is S is a PowSer with R=S.PowSerRing then the value is $R.T^{S.e} * S.P + O(R.T^{S.prec+S.e})$ Normally, the constant term of S.P should be non-zero. There is an obvious amiguity in representing the zero-series.

Function IsPowSer

Tests whether an object is a PowSerRing

SYNTAX. `B:=IsPowSer(X)`

```
object   X
boolean  B
```

Function PowSerRing

Constructs a power series ring

SYNTAX.
```
R:=PowSerRing(A [,prec]);
R:=PowSerRing(S);
```

`PowSerRing`	`R`	
`PolyAlg`	`A`	base polynomial algebra
`PowSer`	`S`	series to give ambient ring of
`int`	`prec`	maximum precision

EXPLANATION. prec should be at least 1. The second form simply returns the ring that S is a member of.

Function PolyShift

Removes trailing zero-coefficients from a polynomial

SYNTAX. `L:=PolyShift(P)`

`List`	`L`	[n,Q]
`Poly`	`P,Q`	
`int`	`n`	

EXPLANATION. Let x be the variable of P. Then $x^n Q = P$. If P is nonzero, then the constant coefficient of Q is non-zero. If $P = 0$ then $n = 0$ (and $Q = P = 0$).

Function IsPowSerRing

Tests whether an object is a PowSerRing

SYNTAX. `B:=IsPowSerRing(X)`

`object`	`X`
`boolean`	`B`

Function PowSerNormalise

Scales a power series to standard representation

SYNTAX. `S2:=PowSerNormalise(S1)`

`PowSer`	`S1,S2`

EXPLANATION. If $S1$ is nonzero, then $S2.P$ has a non-zero constant coefficient. If $S1$ is zero, then $S2.prec = 0$. Effective precision is preserved, i.e. $S1.e + S1.prec = S2.e + S2.prec$.

This choice makes sure that $S2.e$ indicates the order of vanishing at 0.

Function PowSer

Returns a power series

SYNTAX.
```
S:=PowSer(R,P);
S:=PowSer(R,L);
S:=PowSer(R,Q);
```

`PowSer`	`S`	
`PowSerRing`	`R`	
`Poly`	`P`	
`List`	`L`	[e,P,prec]
`Qfe`	`Q`	

EXPLANATION. A normalised power series is returned. The answer is of the highest precision allowed by both the input and the precision of R.

Function PowSerPrint

Prints a PowSer

SYNTAX. `PowSerPrint(S);`

```
PowSer  S
```

Function PowSerAdd

Adds two power series

SYNTAX. `S3:=PowSerAdd(S1,S2);`

```
PowSer  S1,S2,S3
```

EXPLANATION. S1 and S2 should be in the same PowSerRing. If one of the arguments is not a PowSer, a rudimentary cast is attempted through Kash's automatic polynomial cast.

Function PowSerMul

Multiplies two power series

SYNTAX. `S3:=PowSerMul(S1,S2);`

```
PowSer  S1,S2,S3
```

EXPLANATION. S1 and S2 should be in the same PowSerRing. If one of the arguments is not a PowSer, a rudimentary cast is attempted through Kash's automatic polynomial cast.

Function PolyInvert

Inverts a polynomial

SYNTAX. `Q:=PolyInvert(P,n)`

```
Poly  P,Q
int   n
```

EXPLANATION. Computes the polynomial Q of degree $n-1$ such that $PQ = 1+O(x^n)$. The constant coefficient of P should be invertible in the coefficient ring of P. (i.e. no automatic cast from Z to Q)

Function PowSerDiv

Divides two power series

SYNTAX. `S3:=PowSerDiv(S1,S2);`

```
PowSer  S1,S2,S3
```

EXPLANATION. S1 and S2 should be in the same PowSerRing. If one of the arguments is not a PowSer, a rudimentary cast is attempted through Kash's automatic polynomial cast.

Function PowSerTrunc

Truncates a PowSer to a polynomial or a rational function.

SYNTAX. `P:=PowSerTrunc(S [, n]);`

`Poly | Qfe`	`P`	
`PowSer`	`S`	
`int`	`n`	precision

EXPLANATION. If no n is supplied, truncation is done on the precision available. If n is present, truncation mod $O(x^n)$ is performed. It is an error if S has insufficient precision.

Function PowSerSub

Subtracts two power series

SYNTAX. `S3:=PowSerSub(S1,S2);`

```
PowSer  S1,S2,S3
```

EXPLANATION. S1 and S2 should be in the same PowSerRing. If one of the arguments is not a PowSer, a rudimentary cast is attempted through Kash's automatic polynomial cast.

Function PowSerPow

Computes a power of a PowSer by repeated squaring.

SYNTAX. `S2:=PowSerPow(S1,n);`

```
PowSer  S1,S2
int     n
```

EXPLANATION. If $n = 0$, 1 with maxprec is returned. For $n < 0$, $1/PowSerPow(S1, -n)$ is returned.

Function PowSerNewton

Performs Newton approximation for a power series solution of a polynomial equation.

SYNTAX. `S:=PowSerNewton(F,S0)`

`P`	`PowSer: S, S0 -`	
`Poly`	`F`	equation

EXPLANATION. Poly should be of trancendance degree 1 over the polynomial algebra that S is based on. $S0$ should be an approximation of a root to sufficient precision to distinguish it from other roots. The returned series S lets F vanish to at least the precision of $S0$. If F is ramified at T, this is NOT the same as that S is a root up to that presision.

Function PowSerO

returns a 0 power series of given precision.

SYNTAX. `S:=PowSerO(R,n);`

`PowSer`	`S`
`PowSerRing`	`R`
`int`	`n`

EXPLANATION. returns $0 + O(R.T^n)$.

Function PowSerRingPolyAlg

Returns underlying PolyAlg of a PowSerRing

SYNTAX. `A:=PowSerRingPolyAlg(R)`

`PolyAlg`	`A`
`PowSerRing`	`R`

Function PowSerDeriv

Formally differentiates a PowSer.

SYNTAX. `S2:=PowSerDeriv(S1);`

`PowSer  S1,S2`

Function PowSerInt

Formally integrate a PowSer.

SYNTAX. `S2:=PowSerInt(S1);`

`PowSer  S1,s2`

EXPLANATION. S1 should have valuation >= 0

Function PowSerVal

returns valuation of PowSer, i.e. lowest degree of nonzero coefficient.

SYNTAX. `n:=PowSerVal(S);`

```
PowSer  S
int     n
```

EXPLANATION. If S is zero, then the precision is returned, since that may be the lowest term that actually occurs. Therefore, PowSerVal is the degree to which S is sure to vanish at 0.

Function PowSerEval

Computes a specialisation of a PowSer.

SYNTAX. `V:=PowSerEval(S,t);`

```
object  V,t
PowSer  S
```

EXPLANATION. equivalent with casting to a rational function and substituting the variable with the supplied value.

Function PowSerCoef

Returns specified coefficient from PowSer

SYNTAX. `C:=PowSerCoef(S,n);`

```
ring element  C
PowSer        S
int           n
```

EXPLANATION. Returns the coefficient of $S.T^n$ in S. If n exceeds the precision of S then the string "undetermined" is returned.

Function PowSerMapInv