# redlib: Reduction types of curves

Tim Dokchitser
School of Mathematics
University of Bristol
timdok@gmail.com

v3.1 — November 7, 2024

Reduction types of curves over discrete valuation rings in Magma
Combinatorics of reduction types in Magma, Python and JavaScript
https://people.maths.bris.ac.uk/~matyd/redlib/

# 1 Introduction

The `redlib` library is a collection of routines for working with

- Combinatorics of special fibres of minimal regular models with normal crossings, their dual graphs and reduction types (Magma, Python, JavaScript),
- General discrete valuation rings (Magma),
- Reduction types of general curves over DVRs (Magma).

The Magma version implements computing reduction types for

- $\Delta_v$-regular curves (see [Do1])
- Hyperelliptic curves of any genus in residue characteristic $\neq 2$ (Muselli's algorithm [Mu]).

All three versions implement conversion between dual graphs of special fibres, reduction types and their labels, and implement drawing reduction types and their associated shapes in TeX. Magma version also implements drawing special fibres in TeX.

To install the library, unpack it into a working directory, and use

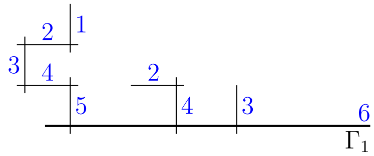
| | | |
|---|---|---|
| `AttachSpec("redlib.spec");` | Magma | see ex-redlib.m |
| `import redtype` | Python | see ex-redlib.py |
| `import redlib from './redtype.ts';` | standalone JavaScript | see ex-redlib.js |
| `<script src="redtype.js"></script>` | JavaScript in html | |

This library accompanies the paper [Do2] on the classification of reduction types. We now describe the functionality in Magma. See §10 for the python version and §11 for the JavaScript version.

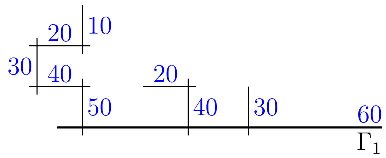
## 1.1 Examples: reduction types and labels

**Example** (Type II* elliptic curve).

```
> R:=ReductionType("II*");      // Kodaira-Neron type II*
> G:=DualGraph(R);
> TeX(G);
```



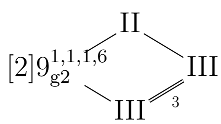

```
> Sprint(G,"Magma");
DualGraph([6,5,4,3,2,1,4,2,3], [0,0,0,0,0,0,0,0,0],
  [[1,2],[1,7],[1,9],[2,3],[3,4],[4,5],[5,6],[7,8]])
> R:=ReductionType("[10]II*");  // same II*, but now with multiplicity 10
> TeX(DualGraph(R));
```



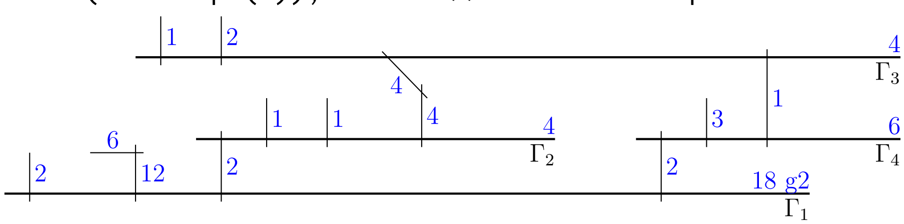

```
> Genus(R);                         // any such type has chi=0 and genus 1
1
```

**Example** (Reduction type in large genus).

```
> R:=ReductionType("III=(3)III-II-{2-2}18g2^2,2,2,12-c1");
> Genus(R);                     // Genus of the generic fibre
58
> TeX(R);                       // Reduction type as a graph
```

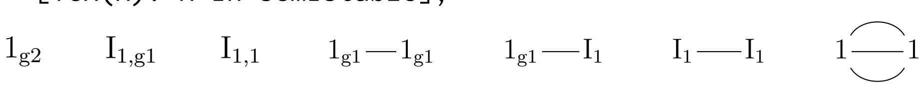



```
> TeX(DualGraph(R));            // associated special fibre
```



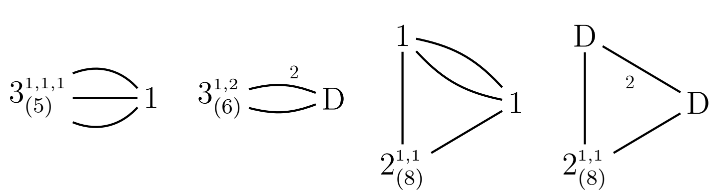

**Example** (All reduction types in a given genus).

```
> #ReductionTypes(2);                           // 104 reduction type families for g=2
104
> semistable:=ReductionTypes(2: semistable);  //   of which 7 are semistable
> [TeX(R): R in semistable];
```




**Example** (Reduction types of a given shape).

```
> L:=[D[1]: D in Shapes(3) | D[2] in [5..8]];      // Genus 3 shapes
> &cat [TeX(S): S in L];                           // with 5..8 reduction types in them
```



```
> ReductionTypes(L[2]);         // Labels of reduction types in the second one
[III*-{2-2}-D,I1*-=D,I0*-=D,III--{2-2}D,II*-{4-2}-D,II--{2-2}D]
> PrincipalTypes(-3,[1,2]);     // and 6 principal types that can be a leftmost vertex
[I0*-{1}=,I1*-{1}=,III-{1}-{2},III*-{3}-{2},II-{1}-{2},II*-{5}-{4}]
```

## 1.2 Examples: Muselli's algorithm for hyperelliptic curves

**Example** (Hypereliptic curves over $\mathbb{Q}$).

```
> R<x>:=PolynomialRing(Q);
> C:=HyperellipticCurve(x^9+10);    // C/Q: y^2=x^9+10
> ReductionType(C,3);    // bad
12^1,5,6-{5-2}IV*
> ReductionType(C,5);    // bad
18^1,8,9
> ReductionType(C,7);    // good
1g4
> ReductionType(C,2);    // uses Delta_v-regular models (see below)
18^1,8,9
```

**Example** (Genus 2 curves over $\mathbb{Q}_p$).

```
> K:=pAdicField(3,20);                          // work over Q_3
> R<x>:=PolynomialRing(K);
> ReductionType(HyperellipticCurve(x^3+3));   // y^2=x^3+3 (elliptic, same as Kodaira)
II
> R:=ReductionType(HyperellipticCurve(x^6+3*x^3+9));
> R;                                            // y^2=x^6+3x^3+9 (genus 2)
T=(3)T
> nu,page:=NamikawaUeno(R);
> nu;             // Namikawa-Ueno type name in genus 2
III$_{3}$
> page;           // and page in their paper to avoid ambiguities
184
> ReductionType(HyperellipticCurve(x^9+3));   // y^2=x^9+3  (genus 4)
18^1,8,9
> ReductionType(HyperellipticCurve(x^81+3));  // y^2=x^81+3 (genus 40)
162^1,80,81
```

**Example** (Hypereliptic curves over number fields).

```
> R<x>:=PolynomialRing(Q);
> C:=HyperellipticCurve(x^5+3);    // y^2=x^5+3 at p=3
> ReductionType(C,3);              // bad reduction over Q
10^1,4,5
> K<r5>:=NumberField(x^5-3);
> CK:=BaseChange(C,K);
> PK:=ideal<Integers(K)|r5>;
> ReductionType(CK,PK);             // nearly good over Q(3^(1/5))
2^1,1,1,1,1,1
> L<r10>:=NumberField(x^10-3);
> CL:=BaseChange(C,L);
> PL:=ideal<Integers(L)|r10>;
```

```
> ReductionType(CL,PL);                // good over Q(3^(1/10))
1g2
```

**Example** (Hypereliptic curves over extensions of $p$-adics).

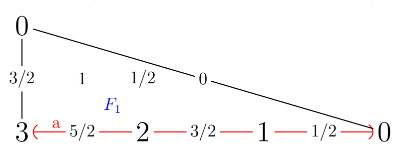
```
> K:=pAdicField(3,20);                 // same example as above but much faster, because
> R<x>:=PolynomialRing(K);             //   no need to compute rings of integers
>                                      //   in number fields, which is slow
> C:=HyperellipticCurve(x^5+3);        // y^2 = x^5+3 over Q3
> ReductionType(C);                    // bad reduction over Q3
10^1,4,5
> L1:=ext<K|x^5-3>;
> ReductionType(BaseChange(C,L1));     // nearly good over Q3(3^(1/5))
2^1,1,1,1,1,1
> L2:=ext<K|x^10-3>;
> ReductionType(BaseChange(C,L2));     // good over Q3(3^(1/10))
1g2
```

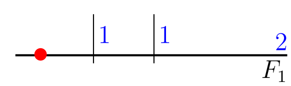
**Example** (Hypereliptic curves over $\mathbb{F}_3(t)$ at $t=0$).

```
> K<t>:=RationalFunctionField(GF(3));
> R<x>:=PolynomialRing(K);
> C1:=HyperellipticCurve(x^5+t);       // y^2=x^5+t over F_3(t)
> Model(C1,t);
Muselli model of x^5+t=0 at 3 of type 10^1,4,5
```

**Example** (Higher degree MacLane valuations).

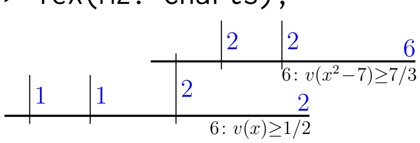
```
> R<x,y>:=PolynomialRing(Q,2);
> p:=7;
> f:=y^2 - (x^2-p)^3 - p^7;
> M1:=DeltaRegularModel(f,DVR(Q,p));     // not Delta_v-regular in any model
> TeX(M1: Delta, Charts);
```





| | | | |
|---|---|---|---|
| $F_1$ | $x = X^{-1}YZ$ | $X = y^{-2}p^3$ | $6Y^6 + 3XY^4 + 4X^2Y^2 + X^3 + X^2 = 0$ |
| | $y = X^{-2}Z^3$ | $Y = xy^{-1}p$ | $Z^2 = 0$ |
| | $p = X^{-1}Z^2$ | $Z = y^{-1}p^2$ | |
| $a$ | $L = 1$ | $r = [1]^3$ | |

```
> M2:=Model(f,p);                        // default Model uses Muselli's algorithm
> TeX(M2: Charts);                       // and higher degree MacLane valuations
```

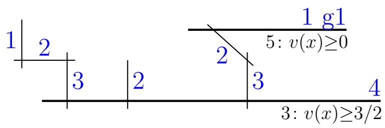



| $\mathfrak{s}$ | $v$ | $\lvert\mathfrak{s}\rvert$ | $d_v$ | $b_v$ | $e_v$ | $\nu_v$ | $n_v$ | $m_v$ | $t_v$ | $p_v$ | $s_v$ | $\gamma_v$ | $p_v^0$ | $s_v^0$ | $\gamma_v^0$ | $u_v$ | $g$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\mathfrak{s}_1$ | $v(x^2-7)\geq 7/3$ | 6 | 2 | 3 | 6 | 7 | 2 | 6 | 3 | 1 | 7/6 | 1 | 2 | $-7/6$ | 1 | 1 | 0 |
| $\mathfrak{s}_2$ | $v(x)\geq 1/2$ | 6 | 1 | 2 | 2 | 3 | 2 | 2 | 6 | 2 | 1/2 | 1 | 2 | $-1$ | 2 | 2 | 0 |

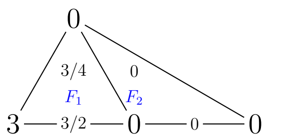
## 1.3 Examples: $\Delta_v$-regular models for plane curves

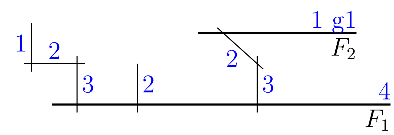
**Example** (Curve from [Do1, Table 1 (i)]).

```
> R<x,y>:=PolynomialRing(Q,2);
> p:=3;
> f:=x*y^2-x^4-x^2-p^3;
> M:=Model(f,3);                   // by default uses Muselli's algorithm, as
> TeX(M);                          // it is hyperelliptic and p<>2
```



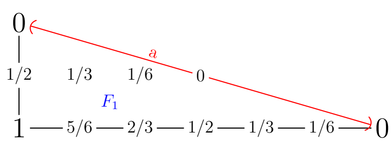

```
> M:=Model(f,3: model:="delta");   // force Magma to use Delta_v-regular machinery
> TeX(M: Delta);                   //   and show Newton polygon as well
```

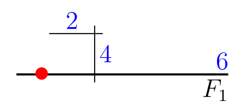



**Example** (Model of $y^2 = x^6 + 2$ over $\mathbb{Q}_2$).

```
> K:=pAdicField(2,30);             // no Muselli's algoritm when p=2, so Model will
> R<x>:=PolynomialRing(K);         // attempt to use Delta_v-regular models
> C1:=HyperellipticCurve(x^6+2);   // given equation is not Delta_v-regular
> TeX(Model(C1): Delta);           // with a singularity along the y^2,y*x^3,x^6 segment
```

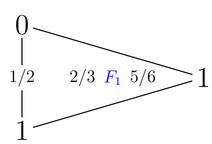


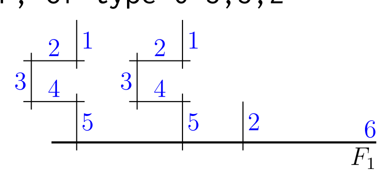




The reduced polynomial $t^2 + 1$ has a double root, so we shift it to 0 with $y \mapsto y + x^3$

```
> C2:=Transformation(C1,1,x^3);
> TeX(Model(C2): RedType, Delta);  // this is now Delta_v-regular, of type 6^5,5,2
```

Type $6^{5,5,2}$

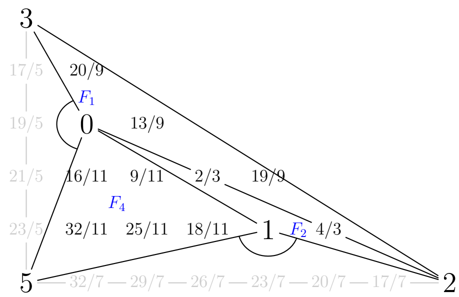


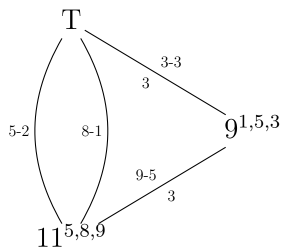

```
> NamikawaUeno(ReductionType(C2)); // or V* in Namikawa-Ueno
V$^*$ 156
```

**Example** (Large genus example).

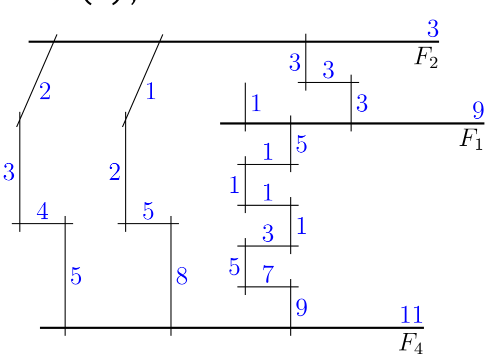
```
> R<x,y>:=PolynomialRing(Q,2);
> p:=13;
> f:=p^3*y^5 + p^2*x^7 + p^5 + p*x^4*y + x*y^3;
> M:=Model(f,p);                 // This is Delta_v regular as seen from the picture
> DeltaTeX(M);                   // (nothing in red that indicates singularities)
```




```
> IsSingular(M);
false
> R:=ReductionType(M); R;      // Associated reduction type
11^5,8,9-{9-5}(3)9^1,5,3-{3-3}(3)T-{1-8}-{2-5}c1
> Genus(R);                    // and genus (=number of interior pts in Newton polygon)
12
> TeX(R: scale:=2);            // Reduction type as a graph
```




```
> TeX(M);                      // and picture of the special fibre
```



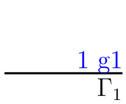

## 1.4 Examples: Classification in genus 1 and 2

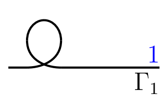
**Example** (Elliptic curves). Reduction types of elliptic curves come in 10 families, called Kodaira types. They are accessed like this:

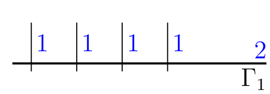
```
> E:=ReductionTypes(1: elliptic); E;
1g1, I1, I0*, I1*, IV, IV*, III, III*, II, II*
```

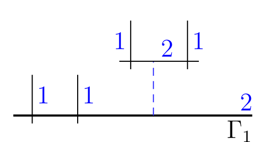
Define a helper function to TeX a dual graph of a reduction type given by a label, and generate their special fibres in tikz. Note that In, In* ($n \geq 1$) are families, with link chains of varying possible lengths, while the others do not allow for any variation.

```
> t:=func<s|TeX(DualGraph(ReductionType(s)): xscale:=0.75)*" \\hfill ">;
> t("1g1"), t("I1"), t("I0*"), t("I1*");
```

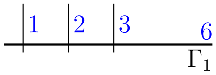


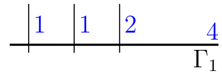


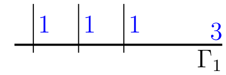




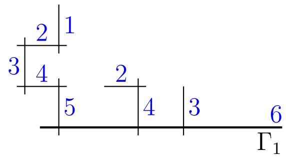
```
> t("II"), t("III"), t("IV");
```

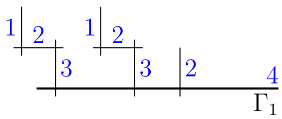


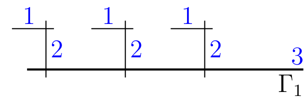





```
> t("II*"), t("III*"), t("IV*");
```

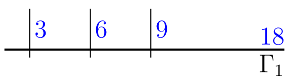







**Example** (Genus 1 curves). Genus 1 curves have reduction types [d]K where K is one of the Kodaira types above, and $d \geq 1$ any multiple. For example,

```
> R:=ReductionType("[3]II");
> Genus(R);
1
> TeX(DualGraph(R));     // 3 x Type II
```




**Example** (Genus 2 curves). Reduction types of genus 2 come in 104 families, classified by Namikawa–Ueno. Here is how to construct all of them by labels. Write K for one of the 10 Kodaira types

```
> ReductionTypes(1: elliptic);
1g1, I1, I0*, I1*, IV, IV*, III, III*, II, II*
```

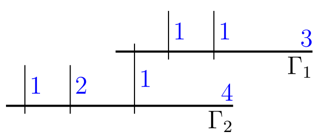
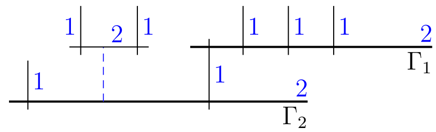
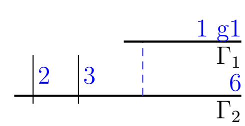
and define again a helper function to TeX a dual graph of a reduction type given by a label

```
> t:=func<s|TeX(DualGraph(ReductionType(s)): xscale:=0.75)*" \\hfill ">;
```

Genus 2 classification (104 in total):

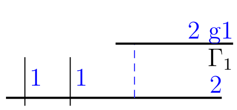
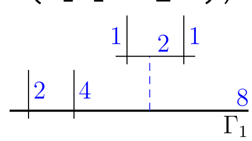
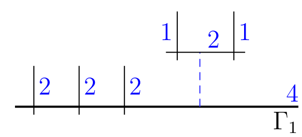
1. The 55 types of the form K1-K2 where K1, K2 are any of the 10 Kodaira types. For example, IV-III, I1*-I0*, 1g1-II*, etc.

   ```
   > t("IV-III"), t("I1*-I0*"), t("1g1-II");
   ```

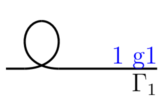
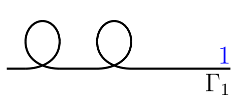
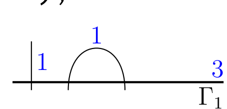
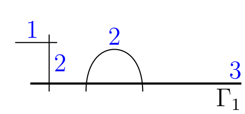
2. The 10 types of the form [2]K_D where K is one of the 10 Kodaira types. There is a unique way to attach a D-link in a minimal way to [2]K in every case. For example, [2]IV_D, [2]I1*_D, etc.

   ```
   > t("[2]1g1_D"), t("[2]III_D"), t("[2]I0*_D");
   ```

3. The 8 types K_n obtained by adding a loop to every Kodaira type except II, II*. For II, II* all the outgoing open chains have different initial multiplicities, so this is not possible, but it is possible for all the others, again in a unique minimal way. For example, 1g1_1, IV_0, IV*_-1, etc.

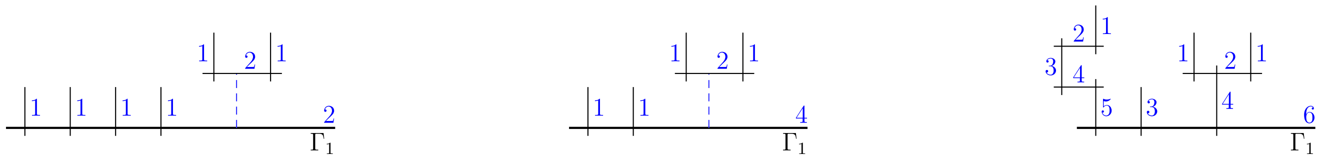
   ```
   > t("1g1_1"), t("I1_1"), t("IV_0"), t("IV*_-1");
   ```

4. The 6 types K_D obtained by adding a D-link to a Kodaira type whose principal component has

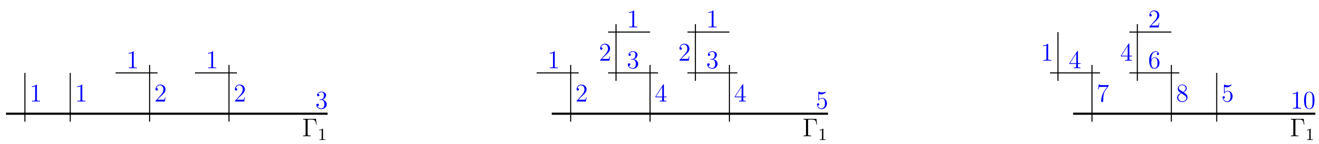
even multiplicity, namely I0*, I1*, III, III*, II, II*. For example, I0*_D, III_D, II*_D, etc.

```
> t("I0*_D"), t("III_D"), t("II*_D");
```



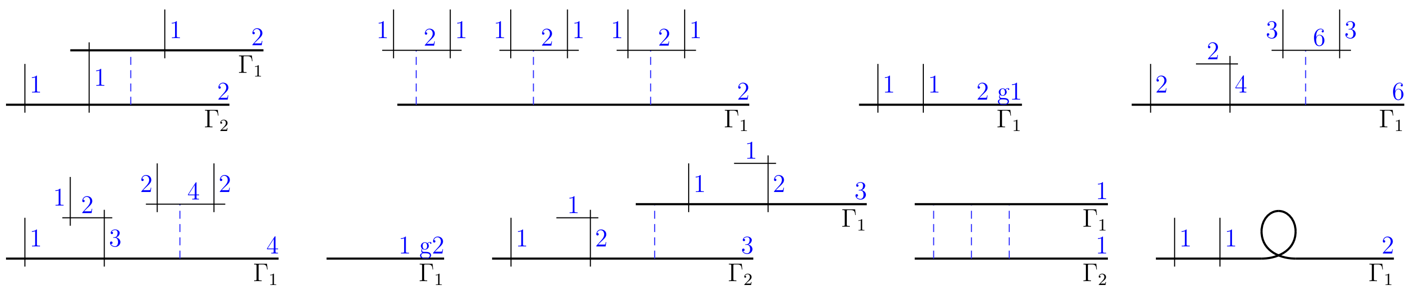
5. The 16 types from cores with $\chi = -2$, consisting of one principal component of genus 0 and multiplicity $m$, and open chains with initial multiplicities $d_1, ..., d_k \in \mathbb{Z}/m\mathbb{Z}$ and $\sum d_i = 0$.

```
> Cores(-2);
[ 2^1,1,1,1,1,1, 3^1,1,2,2, 4^1,3,2,2, 5^1,1,3, 5^1,2,2, 5^2,4,4, 5^3,3,4, 6^1,1,4,
  6^2,4,3,3, 6^5,5,2, 8^1,3,4, 8^5,7,4, 10^1,4,5, 10^3,2,5, 10^7,8,5, 10^9,6,5 ]
> t("3^1,1,2,2"), t("5^2,4,4"), t("10^7,8,5");
```



6. The 9 leftover types D–=D, [2]_D,D,D, Dg1, [2]T_{6}D, 4^1,3_D, 1g2, T=T, 1–––1, D_{2-2}:

```
> left:=["D-=D","[2]_D,D,D","Dg1","[2]T_{6}D","4^1,3_D","1g2","T=T","1---1","D_{2-2}"];
> [t(R): R in left]);
```



# 2 Reduction types (redtype.m)

```
type RedCore
```

```
type RedChain
```

```
type RedPrin
```

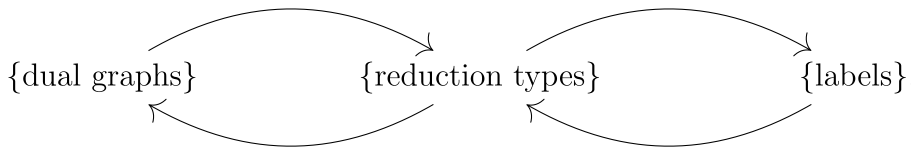
```
type RedShape
```

```
type RedType
```

The library redtype.m implements the combinatorics of reduction types, in particular

- Arithmetic of open and link sequences that controls the shapes of chains of $\mathbb{P}^1$s in special fibres of minimal regular normal crossing models,
- Methods for reduction types (RedType), their cores (RedCore), link chains (RedChain) and shapes (RedShape),
- Canonical labels for reduction types,
- Reduction types and their labels in TeX,

- Conversion between dual graphs, reduction type, and their labels:




**Example** (Reduction types, labels and dual graphs).

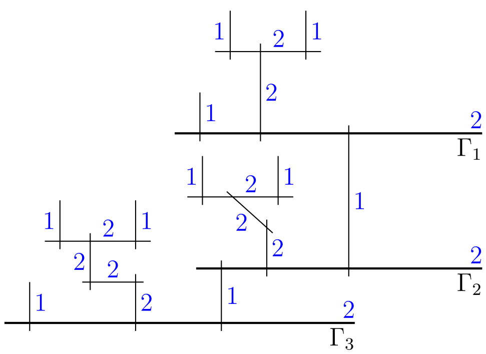
```
> R:=ReductionType("I2*-I3*-I4*");
> Label(R);              // Plain label
I2*-I3*-I4*
> Label(R: tex);         // TeX label
```

$I_2^*{-}I_3^*{-}I_4^*$

```
> TeX(R);                // Reduction type as a graph
```

$I_2^* — I_3^* — I_4^*$

```
> TeX(DualGraph(R));     // Associated dual graph, in TeX
```




This is a large dual graph on 22 components, all of multiplicity 1 or 2, and all of genus 0. Taking the associated reduction type gives back R:

```
> G:=DualGraph([2,2,2,1,1,2,1,1,2,1,2,1,1,2,2,1,2,1,1,2,2,2],
  [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
  [[1,4],[1,9],[1,10],[2,10],[2,14],[2,16],[3,5],[3,16],[3,20],
  [6,7],[6,8],[6,9],[11,12],[11,13],[11,15],[14,15],
  [17,18],[17,19],[17,22],[20,21],[21,22]]);
> ReductionType(G);
I2*-I3*-I4*
```

## 2.1 Open and link chains

A reduction type is a graph that has principal types as vertices (like $I_2^*$, $I_3^*$, $I_4^*$ above) and link chains as edges. Principal types encode principal components together with open chains, loops and D-links. The three functions that control multiplicities of open and link chains, and their depths are as follows:

```
intrinsic OpenSequence(m::RngIntElt, d::RngIntElt: includem:=true) ->
  SeqEnum[RngIntElt]
```

```
Unique open sequence of type (m,d) for integers m>=1 and 1<d<m. It is of the form
   [m,d,...,gcd(m,d)]
with every three consecutive terms d_(i-1), d_i, d_(i+1) satisfying
   d_(i-1) + d_(i+1) = d_i * (integer > 1).
If includem:=false, exclude the starting point m from the sequence.
```

**Example** (OpenSequence).

```
> OpenSequence(6,5);
[ 6, 5, 4, 3, 2, 1 ]
> OpenSequence(13,8);
[ 13, 8, 3, 1 ]
```

```
intrinsic LinkSequence(m1::RngIntElt, d1::RngIntElt, m2::RngIntElt,
  dk::RngIntElt, n::RngIntElt: includem:=true) -> SeqEnum[RngIntElt]
```

```
Unique link sequence of type m1(d1-dk-n)m2, that is of the form [m1,d1,...,dk,m2] with n+1 terms
equal to gcd(m1,d1)=gcd(m2,dk) and satisfying the chain condition: for every three consecutive terms
   d_(i-1), d_i, d_(i+1)
we have
   d_(i-1) + d_(i+1) = d_i * (integer > 1).
If includem:=false, exclude the endpoints m1,m2 from the sequence.
```

**Example** (LinkSequence).

```
> LinkSequence(3,2,3,2,-1);
[ 3, 2, 3 ]
> LinkSequence(3,2,3,2,0);
[ 3, 2, 1, 2, 3 ]
> LinkSequence(3,2,3,2,1);
[ 3, 2, 1, 1, 2, 3 ]
```

```
intrinsic MinimalLinkDepth(m1::RngIntElt, d1::RngIntElt, m2::RngIntElt,
  dk::RngIntElt) -> RngIntElt
```

```
Minimal depth of a link chain m1=d0,d1,d2,...,dk,m2=d(k+1) of P1s between principal components
of multiplicity m1, m2 and initial link multiplicities d1,dk. The depth is defined as
   -1 + number of times gcd(d1,...,dk) appears in the sequence.
For example, 5,4,3,2,1 is a valid link sequence, and MinimalLinkDepth(5,4,1,2) = -1 + 1 = 0.
```

**Example**. Example from the description of the intrinsic:

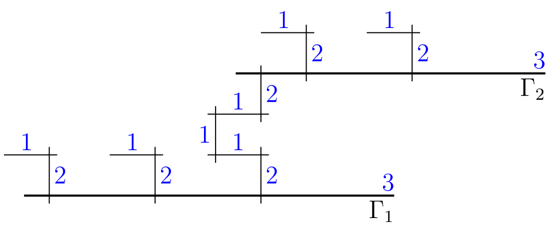
```
> MinimalLinkDepth(5,4,1,2);
0
```

For another example, the minimal $n$ in the Kodaira type $\mathrm{I}_{\mathrm{n}}^*$ is 1. Here the chain links two components of multiplicity 2, and the initial multiplicities are 2 on both sides as well:

```
> MinimalLinkDepth(2,2,2,2);
1
```

Here is an example of a reduction type with a link chain between two components of multiplicity 3 and outgoing multiplicities 2 on both sides:

```
> R:=ReductionType("IV*-(2)IV*");
> TeX(DualGraph(R));
```



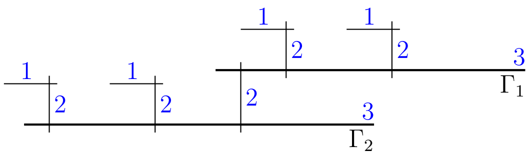

The link chain has gcd=GCD(3,2)=1 and

$$\text{depth} = -1 + \#1\text{'s}(=\text{gcd}) \text{ in the sequence } 3,2,1,1,1,2,3 = 2$$

This is the depth specified in round brackets in IV*-(2)IV*

```
> MinimalLinkDepth(3,2,3,2);        // Minimal possible depth for such a chain = -1
-1
> R1:=ReductionType("IV*-IV*");     // used by default when no expicit depth is specified
```

```
> R2:=ReductionType("IV*-(-1)IV*");
> assert R1 eq R2;
> TeX(DualGraph(R1));
```




The next two functions are used in Label to determine the ordering of chains (including loops and D-links), and default multiplicities which are not printed in labels.

```
intrinsic SortLinks(m::RngIntElt, O::SeqEnum) -> SeqEnum
```

```
Sort a sequence of multiplicities O by gcd with m, then by o. This is how open and loose
multiplicities are sorted in reduction types.
```

**Example** (Ordering open multiplicities in reduction types).

```
> SortLinks(6,[1,2,3,3,4,5]);     // sort links in O by gcd(o,m), then by o mod m
[ 1, 5, 2, 4, 3, 3 ]
```

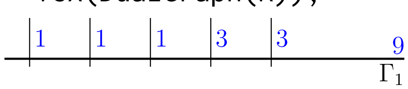
```
intrinsic DefaultMultiplicities(m1::RngIntElt, o1::SeqEnum, m2::RngIntElt,
  o2::SeqEnum, loop::BoolElt) -> RngIntElt, RngIntElt
```

```
Default edge multiplicities d1,d2 for a component with multiplicity m1, available outgoing
multiplicities o1, and one with m2,o2. Parameter loop:boolean specifies whether it is a loop or a
link between two different principal components
```

**Example** (DefaultMultiplicities). Let us illustrate what happens when we take a principal component $9^{1,1,1,3,3}$ and add five default loops of depth 2,2,1,2,3, to get a reduction type $9^{1,1,1,3,3}_{2,2,1,2,3}$. How do default loops decide which initial multiplicities to take?

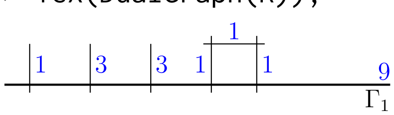
We start with a component of multiplicity $m = 9$ and open multiplicities $\mathcal{O} = \{1, 1, 1, 3, 3\}$.

```
> R:=ReductionType("9^1,1,1,3,3");
> TeX(DualGraph(R));
```




We can add a loop to it linking two 1's of depth 2 by

```
> R:=ReductionType("9^1,1,1,3,3_{1-1}2");
> TeX(DualGraph(R));
```

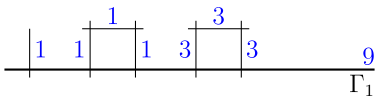



In this case, {1-1} does not need to be specified because this is the minimal pair of possible multiplicities in $\mathcal{O}$, as sorted by SortLinks:

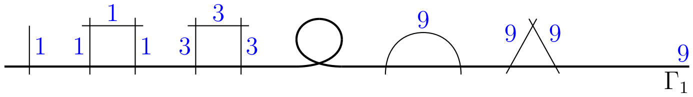
```
> DefaultMultiplicities(9,[1,1,1,3,3],9,[1,1,1,3,3],true);
1 1
> assert R eq ReductionType("9^1,1,1,3,3_2");
```

After adding the loop, $\{1, 3, 3\}$ are left as potential outgoing multiplicities, so the next default loop links 3 and 3. Note that $1, 3$ is not a valid pair because $\gcd(1, 9) \neq \gcd(3, 9)$.

```
> DefaultMultiplicities(9,[1,3,3],9,[1,3,3],true);
3 3
> R2:=ReductionType("9^1,1,1,3,3_2,2");       // 2 loops, use 1-1 and 3-3
> TeX(DualGraph(R2));
```



```
> DefaultMultiplicities(9,[1],9,[1],true);
9 9
> R3:=ReductionType("9^1,1,1,3,3_2,2,1,2,3"); // no pairs left -> next three loops
> TeX(DualGraph(R3));                         //    use (m,m)=(9,9)
```



```
> assert R3 eq ReductionType("9^1,1,1,3,3_{1-1}2,{3-3}2,{9-9}1,{9-9}2,{9-9}3");
```

## 2.2 Principal component core (RedCore)

```
type RedCore
```

A core is a pair $(m, O)$ with 'principal multiplicity' $m \geq 1$ and 'outgoing multiplicities' $O = \{o_1, o_2, ...\}$ that add up to a multiple of $m$, and such that $\gcd(m, O) = 1$. It is implemented as the following type:

```
declare type RedCore;
declare attributes RedCore:
  m,          // main component multiplicity
  O,          // outgoing multiplicities in Z/mZ with GCD(m,O)=1, sorted with SortLinks
  chi;        // Euler characteristic m*(2-#O) + sum_{o in O} GCD(m,o), even <=2
```

```
intrinsic Core(m::RngIntElt, O::SeqEnum) -> RedCore
```

Create a new core from principal multiplicity m and outgoing multiplicities O.

```
intrinsic Print(C::RedCore, level::MonStgElt)
```

Print a principal component core through its label.

**Example** (Create and print a principal component core $(m, O)$).

```
> Core(8,[1,3,4]);     // Typical core - multiplicities add up to a multiple of m
8^1,3,4
> Core(8,[9,3,4]);     // Same core, as they are in Z/mZ
8^1,3,4
```

This is how cores are printed, with the exception of 7 cores of $\chi = 0$ (see below) that come from Kodaira types and two additional special ones D and T:

```
> Core(6,[1,2,3]);                    // from a Kodaira type
II
> [Core(2,[1,1]),Core(3,[1,2])];      // two special ones
[D,T]
```

## 2.3 Basic invariants and printing

```
intrinsic Multiplicity(C::RedCore) -> RngIntElt
```

Principal multiplicity m of a reduction type core.

```
intrinsic OpenMultiplicities(C::RedCore) -> SeqEnum
```

Outgoing multiplicities O of a reduction type core, sorted with SortLinks

```
intrinsic Chi(C::RedCore) -> RngIntElt
```

Euler characteristic of a reduction type core (m,O), chi = m(2-|O|) + sum_(o in O) gcd(o,m)

```
intrinsic Label(C::RedCore: tex:=false) -> MonStgElt
```

Label of a reduction type core, for printing (or TeX if tex:=true)

```
intrinsic TeX(C::RedCore) -> MonStgElt
```

Print a reduction type core in TeX.

**Example** (Core labels and invariants).

```
> C:=Core(2,[1,1,1,1]);
> Multiplicity(C);        // Principal multiplicity m
2
> OpenMultiplicities(C);  // Outgoing multiplicities O
[ 1, 1, 1, 1 ]
> Chi(C);                 // Euler characteristic
0
> Label(C);               // Plain label
I0*
> TeX(C);                 // TeX label
```

$I_0^*$

```
> C: Magma;               // How it can be defined
Core(2,[1,1,1,1])
```

```
intrinsic Cores(chi::RngIntElt: mbound:="all", sort:=true) -> SeqEnum
```

Returns all reduction type cores (m,O) with given Euler characteristic chi<=2. When chi=2 there are infinitely many, so a bound on m must be given

**Example** (Cores).

```
> Cores(0);                              // I0*,IV,IV*,III,III*,II,II* (7 of them)
I0*, IV, IV*, III, III*, II, II*
> [#Cores(i): i in [0..-10 by -2]];      // 7, 16, 43, 65, 64, ...
[ 7, 16, 43, 65, 64, 193 ]
```

## 2.4 Link chains (RedChain)

Link chains between principal components fall into three classes: loops on a principal type, D-link on a principal type, and chains between principal types that link two of their loose edge endpoints. All of these are implemented as type RedChain that carries class=cLoop, cD or cLoose, and keeps track of all the invariants.

```
declare type RedChain;         // Link chain: loop, D-link or linking two loose edge
declare attributes RedChain:   //   endpoints of two distinct principal components
  class,    // cLoop, cD, cLoose - must be assigned
            // all other attributes may be false if unassigned
  index,    // unique identifier, eventually index in a global array of edges
  Si,Sj,    // principal types S[i], S[j] between which the edge is going
  mi,mj,    // principal multiplicities of the components S[i], S[j]
  di,dj,    // outgoing multiplicities of the link chain, so that it is mi,di,...,dj,mj
  depth,    // original depth, used for sorting
```

```
  depthstr; // string for printing, by default Sprint(depth), but could me "m", "n", etc.
```

```
type RedChain
```

```
intrinsic Link(class::RngIntElt, mi::RngIntElt, di::RngIntElt, mj::Any, dj::Any:
  depth:=false, Si:=false, Sj:=false, index:=false) -> RedChain
```

```
Return a link chain of a given class and specified invaraints:
  class   = cLoop (loop), cD (D-link) or cLoose (link chain between different principal types)
  Si      = originating principal type S_i (by default unspecified (Si:=false))
  mi, di  = principal multiplicity of S_i and outgoing multiplicity of the chain from S_i
  Sj      = target principal type S_j (by default unspecified (Sj:=false))
  mj, dj  = principal multiplicity of S_j and outgoing multiplicity of the chain from S_j
    so that the chain of P1s has multiplicities [mi,di,...,dj,mj]
  depth   = depth of the chain (by default minimal (depth:=false))
  index   = index in the list of link chains of a reduction type to which the chain belongs
            (by default unspecified (index:=false))
```

```
intrinsic Print(c::RedChain, level::MonStgElt)
```

```
Print a chain c like 'class mi,di - (depth) mj,dj', together with indices of Si, Sj and c if
assigned
```

**Example** (Some link chains, with no principal types specified).

```
> cLoop, cD, cLoose := Explode([1,2,3]);
> Link(cLoop,2,1,2,1);              // loop
loop 2,1 -(0) 2,1
> Link(cD,2,2,false,false);         // D-link
D-link 2,2 -(1) 2,2
> Link(cLoose,2,2,false,false);     // to another (yet unspecified) principal type
loose 2,2 -(false) false,false
```

## 2.5 Invariants and depth

```
intrinsic Class(c::RedChain) -> RngIntElt
```

```
Class of a RedChain - cLoop, cD or cLoose depending on the type of the chain
```

```
intrinsic GCD(c::RedChain) -> RngIntElt
```

```
GCD of all elements in the chain (=GCD(mi,di)=GCD(mj,dj))
```

```
intrinsic Index(c::RedChain) -> RngIntElt
```

```
Index of the chain c used for ordering chains in a reduction type, and sorting in label.
```

```
intrinsic DepthString(c::RedChain) -> MonStgElt
```

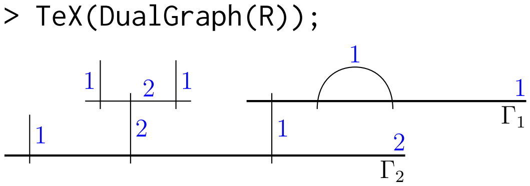
```
String set by SetDepths how c is printed, e.g. "1" or "n"
```

```
intrinsic SetDepthString(c::RedChain, depth::Any)
```

```
Set how c is printed, e.g. "1" or "n"
```

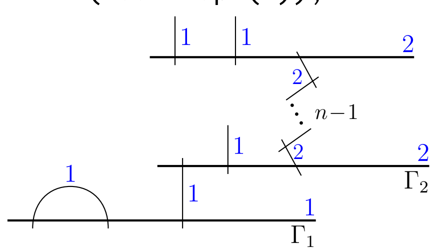
**Example** (Invariants of link chains). Take a genus 2 reduction type $\mathrm{I}_2 \frac{}{1} \mathrm{I}*_2$ whose special fibre consists of Kodaira types $\mathrm{I}_2$ (loop of $\mathbb{P}^1$s) and $\mathrm{I}_2^*$ linked by a chain of $\mathbb{P}^1$s of multiplicity 1.

```
> R:=ReductionType("I2-(1)I2*");
> TeX(DualGraph(R));
```



There are two principal types R!!1=$I_2$ and R!!2=$I_2^*$, with a loop on R!!1 (class cLoop=1), a link chain between them (class cLoose=3), and a D-link on R!!2 (class cD=2) This is the order in which they are printed in the label.

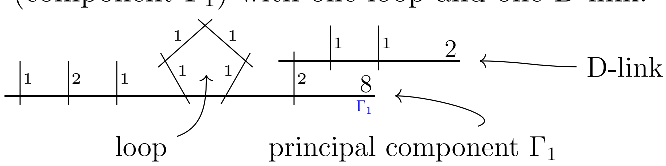
```
> [R!!1,R!!2];                     // two principal types R!!1 and R!!2
[I2-{1},I2*-{1}]
> c1,c2,c3:=Explode(LinkChains(R)); c1,c2,c3;
[1] loop c1 1,1 -(2) c1 1,1
[2] loose c1 1,1 -(1) c2 2,1
[3] D-link c2 2,2 -(2) 2,2
> Class(c3);                       // cLoop=1, *cD=2*, cLoose=3
2
> GCD(c3);                         // GCD of the chain multiplicities [2,2,2]
2
> Index(c3);                       // index in the reduction type
3
> SetDepthString(c3, "n");         // change how its depth is printed in labels
> c3;                              //   and drawn in dual graphs of reduction types
[3] D-link c2 2,2 -(n) 2,2
> Label(R);
I2-(1)In*
> TeX(DualGraph(R));
```




## 2.6 Principal component types (RedPrin)

```
type RedPrin
```

The classification of special fibre of mrnc models is based on principal types. For curves of genus $\geq 2$ such a type is a principal component with $\chi < 0$, together with its open chains, loops, chains to principal component with $\chi = 0$ (called D-links) and a tally of link chains to other principal components with $\chi < 0$, called loose links. For example, the following reduction type has only principal type (component $\Gamma_1$) with one loop and one D-link:




A principal type is implemented as the following Magma type.

```
declare type RedPrin;                 // (m,g,O,Lloops,LD,lloose)
declare attributes RedPrin:
  m,        // principal multiplicity
  g,        // genus
  C,        // chains: open, loops, D-links or loose from S
```

```
O,        // outgoing multiplicities for open chains
L,        // outgoing multiplicities from all other chains
gcd,      // gcd(m,O,L)
core,     // core of type RedCore (divide by gcd)
chi;      // Euler characteristic =chi(m,g,O,L)
```

## 2.7 Creation functions

```
intrinsic PrincipalType(m::RngIntElt, g::RngIntElt, O::SeqEnum, Lloops::SeqEnum,
  LD::SeqEnum, Lloose::SeqEnum: index:=0) -> RedPrin
```

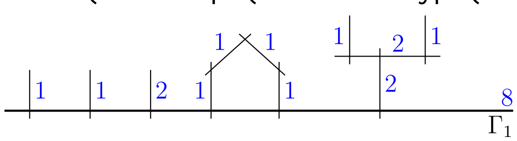
Create a new principal type from its primary invariants, and check integral self-intersection.

**Example**. We construct the principal type from example above. It has $m = 8$, $g = 0$, open multiplicities 1,1,2, loop $1-1$ of depth 3, a D-link with outgoing multiplicity 2 of depth 1, and no loose chains (so that it is a reduction type in itself).

```
> S:=PrincipalType(8,0,[1,1,2],[[1,1,3]],[[2,1]],[]);
```

We print S in a format that can be evaluated back (S: Magma), print its label (by printing S or Label(S)) and draw its dual graph.

```
> S:Magma;
PrincipalType(8,0,[1,1,2],[[1,1,3]],[[2,1]],[])
> S;
8^1,1,1,1,2,2_3,1D
> TeX(DualGraph(ReductionType("8^1,1,1,1,2,2_3,1D")));
```




We can generate all principal types S a given Euler characteristic Chi(S), or restrict to those with a given core or a given sequence of gcd's of outgoing multiplicities of all loose chains. The latter are used to generate all reduction types in given genus through their shapes (see RedShape), where such types placed at the vertices‘.

```
intrinsic PrincipalTypes(chi::RngIntElt, C::RedCore: withlgcds:=false,
  sorted:=true) -> SeqEnum[RedPrin], SeqEnum[SeqEnum[RngIntElt]]
```

Find all possible principal types S with a given core C and Euler characteristic chi. Return a sequence of them.
If withlgcds:=true, also return a sequence lgcds representing all possible LGCD(S).

```
intrinsic PrincipalTypes(chi::RngIntElt: semistable:=false, withlgcds:=false,
  sorted:=true) -> SeqEnum, SeqEnum
```

Find all possible principal types S with a given Euler characteristic chi. Return a sequence of them.
If withlgcds:=true, also return a sequence lgcds representing all possible LGCD(S).

```
intrinsic PrincipalTypes(chi::RngIntElt, lgcd::SeqEnum: semistable:=false,
  withlgcds:=false, sorted:=true) -> SeqEnum
```

All possible principal types with a given Euler characteristic chi and GCDs of loose multiplicities.
If withlgcds:=true, also returns [lgcd] as a second parameter (like all other PrincipalTypes instances).

**Example** (Generating principal types). Geneate principal types of Euler characteristic $\chi = -1, -2, -3, -4$

```
> [#PrincipalTypes(-n): n in [1..4]];      // 13, 83, 75, 277, 176, 591, ...
[ 13, 83, 75, 277 ]
```

Generate those with $\chi = -1$ and one loose chain of multiplicity 1

```
> assert #PrincipalTypes(-1,[1]) eq 10;    // Table 1_10^1 in the classification paper
```

Principal types with core $\chi = -1$ and core IV

```
> PrincipalTypes(-2,Core(3,[1,1,1]));
IV_0, IV-{1}-{1}, [2]IV_D, [2]IV-{2}
```

**Example** (Principal type with given $\chi$ and gcds of loose links).

```
> S:=PrincipalType(4,0,[1,2],[],[],[1]);
> S;              // Kodaira type with one loose link
III-{1}
> Chi(S);         // with chi(S)  = -1
-1
> LGCD(S);        // and  LGCD(S) = [1]
[ 1 ]
> PrincipalTypes(Chi(S),LGCD(S));    // all principal types with these parameters
[1g1-{1},I1-{1},I0*-{1},I1*-{1},IV-{1},IV*-{2},III-{1},III*-{3},II-{1},II*-{5}]
```

## 2.8 Invariants of principal types

```
intrinsic Multiplicity(S::RedPrin) -> RngIntElt
```

Principal multiplicity m of a principal type

```
intrinsic GeometricGenus(S::RedPrin) -> RngIntElt
```

Geometric genus g of a principal type S=(m,g,O,...)

```
intrinsic Index(S::RedPrin) -> RngIntElt
```

Index of the principal component in a reduction type, 0 if freestanding

```
intrinsic Chains(S::RedPrin: class:=0) -> SeqEnum[RedChain]
```

Sequence of chains of type RedChain originating in S. By default, all (loops, D-links, loose) are returned, unless class is specified.

```
intrinsic OpenMultiplicities(S::RedPrin) -> SeqEnum[RngIntElt]
```

Sequence of open multiplicities S`O of a principal type, sorted

```
intrinsic LooseMultiplicities(S::RedPrin) -> SeqEnum[RngIntElt]
```

Sequence of loose multiplicities of a principal type, sorted

```
intrinsic LinkMultiplicities(S::RedPrin) -> SeqEnum[RngIntElt]
```

Sequence of link multiplicities S`L of a principal type, sorted as in label

```
intrinsic Loops(S::RedPrin) -> SeqEnum[RedChain]
```

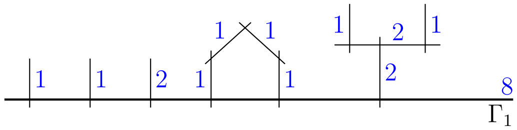
Sequence of chains in S representing loops (class cLoop)

```
intrinsic DLinks(S::RedPrin) -> SeqEnum[RedChain]
```

Sequence of chains in S representing D-links (class cD)

**Example** (Invariants of principal types).

```
> S:=PrincipalType(8,0,[1,1,2],[[1,1,3]],[[2,1]],[]);     // Example above
> TeX(DualGraph(ReductionType([S])));
```




```
> Multiplicity(S);          // Principal component multiplicity
8
> GeometricGenus(S);        // Geometric genus of the principal component
0
> OpenMultiplicities(S);    // Open chain initial multiplicities O=[1,1,2]
[ 1, 1, 2 ]
> Loops(S);                 // Loops (of type RedChain)
[[1] loop c1 8,1 -(3) c1 8,1]
> DLinks(S);                // D-Links (of type RedChain)
[[2] D-link c1 8,2 -(1) 2,2]
> LooseMultiplicities(S);   // Loose link multiplicities
[]
> LinkMultiplicities(S);    // All initial link multiplicities (loops, D-links, loose)
[ 1, 1, 2 ]
```

```
intrinsic GCD(S::RedPrin) -> RngIntElt
```

Return GCD(m,O,L) for a principal type

```
intrinsic Core(S::RedPrin) -> RedCore
```

Core of a principal type - no genus, all non-zero link multiplicities put to 0, and gcd(m,O)=1

```
intrinsic Chi(S::RedPrin) -> RngIntElt
```

Euler characteristic chi of a principal type (m,g,O,Lloops,LD,Lloose), chi = m(2-2g-|O|-|L|) + sum_(o in O) gcd(o,m), where L consists of all the link multiplicities in Lloops (2 from each), LD (1 from each), Lloose (1 from each)

```
intrinsic LGCD(S::RedPrin) -> SeqEnum[RngIntElt]
```

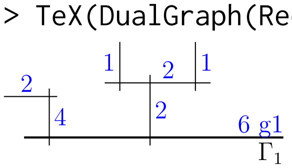
Outgoing link pattern of a principal type = multiset of GCDs of loose edges with m.

**Example** (GCD). Define a principal component type by its primary invariants: $m = 6$, $g = 1$, open multiplicities $\mathcal{O} = \{4\}$, no loops, one D-link with initial multiplicity 2 and length 1, and no loose links:

```
> S:=PrincipalType(6,1,[4],[],[[2,1]],[]);
> GCD(S);                 // its GCD(m,O,L)=GCD(4,[2],[2])=2
2
> Core(S);                // divide by GCD, unlink all chains
T
> S;                      // these are seen as [2] and T in the name
[2]Tg1_1D
```

Note, however, that S is not a multiple of 2 of another principal component type because its D-link is primitive. In other words, the special fibre has odd multiplicity components.

```
> TeX(DualGraph(ReductionType("[2]Tg1_1D")));
```



## 2.9 RedPrin: Weight and comparison

```
intrinsic Weight(S::RedPrin) -> SeqEnum[RngIntElt]
```

Sequence [chi,m,-g,#loose,#Ds,#loops,#O,O,loops,Ds,loose] that determines the weight of a principal type, and characterises it uniquely.

```
intrinsic PrincipalType(w::SeqEnum[RngIntElt]) -> RedPrin
```

Create a principal type S from its weight sequence w (=Weight(S)).

**Example** (Weight).

```
> S:=PrincipalType(8,0,[4,2],[[1,1,1]],[[2,1]],[6]);   // create principal type
> w:=Weight(S);             // its weight encodes chi,m,g,... and characterises it
> w;
[ -26, 8, 0, 1, 1, 1, 2, 2, 4, 1, 1, 1, 2, 1, 6 ]
> PrincipalType(w): Magma;  // so that the component can be reconstructed
PrincipalType(8,0,[2,4],[[1,1,1]],[[2,1]],[6])
```

```
intrinsic 'eq'(S1::RedPrin, S2::RedPrin) -> BoolElt
```

Compare two principal types by their weight

```
intrinsic 'lt'(S1::RedPrin, S2::RedPrin) -> BoolElt
```

Compare two principal types by their weight

```
intrinsic 'le'(S1::RedPrin, S2::RedPrin) -> BoolElt
```

Compare two principal types by their weight

```
intrinsic 'gt'(S1::RedPrin, S2::RedPrin) -> BoolElt
```

Compare two principal types by their weight

```
intrinsic 'ge'(S1::RedPrin, S2::RedPrin) -> BoolElt
```

Compare two principal types by their weight

```
intrinsic Sort(S::SeqEnum[RedPrin]) -> SeqEnum[RedPrin]
```

Sort principal types by their weight

```
intrinsic Sort(~S::SeqEnum[RedPrin])
```

Sort principal types by their weight

**Example** (Sorting principal types by Weight in increasing order).

```
> L := PrincipalTypes(-2,[4]) cat PrincipalTypes(-2,[2,2]);
> [Weight(S): S in L];
[[-2,4,0,1,0,0,2,1,3,4], [-2,4,0,1,1,0,1,2,2,0,4], [-2,2,0,2,0,0,2,1,1,2,2],
  [-2,2,0,2,1,0,0,2,1,2,2]]
> Sort(L);
[D==, [2]_D==, 4^1,3=, [2]D_D=]
```

## 2.10 Printing

```
intrinsic Label(S::RedPrin: tex:=false, loose:=false, wrap:=true,
  returnpieces:=false) -> MonStgElt
```

```
Ascii Label or TeX label of a principal type.
Setting tex:=true prints the tex label, in \redtype... format by default, unless wrap:=false.
Setting loose:=true prints outgoing loose edges as well (standalone principal type).
```

**Example** (Labels without and with loose link chains.). The former are used for printing reduction types (where loose link chains form edges) and the latter are standalone, and define the type uniquely.

```
> [Label(S): S in PrincipalTypes(-1)];
[ 1g1, I1, 1, I0*, I1*, D, IV, T, IV*, III, III*, II, II* ]
> [Sprint(S): S in PrincipalTypes(-1)];
[ 1g1-{1}, I1-{1}, 1-{1}-{1}-{1}, I0*-{1}, I1*-{1}, D-{1}=, IV-{1}, T=, IV*-{2}, III-{1},
  III*-{3}, II-{1}, II*-{5} ]
```

```
intrinsic Print(S::RedPrin, level::MonStgElt)
```

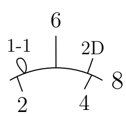
```
Print a principal type as an ascii label or as an evaluatable Magma string (when level="Magma").
```

```
intrinsic TeX(S::RedPrin: length:="35pt", label:=false, standalone:=false) ->
  MonStgElt
```

```
TeX a principal type as a tikz arc with outer and inner lines, loops and Ds.
label:=true puts its label underneath
standalone:=true wraps it in \tikz
```

**Example** (TeX). We define a principal type starting from a core $8^{1,1,2,2,4,6}$, keeping $g = 0$, and declaring $\mathcal{O} = \{2, 4\}$ to be open multiplicities, linking 1,1 one loop of depth 1, using one 2 for a D-link of depth 1, and leaving one 6 as a loose multiplicity.

```
> S:=PrincipalType(8,0,[2,4],[[1,1,1]],[[2,1]],[6]);
> TeX(S: standalone);  // how it appears in the tables (wrapped in \tikz{...})
```

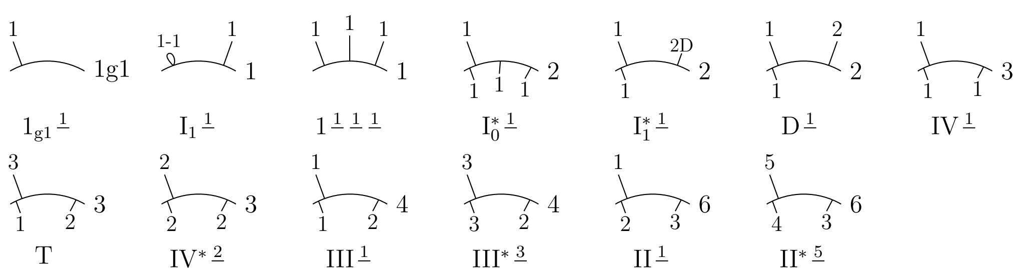



```
intrinsic TeX(T::SeqEnum[RedPrin]: width:=10, scale:=0.8, sort:=true,
  label:=false, length:="35pt", yshift:="default") -> MonStgElt
```

```
TeX a list of principal types as a rectangular table in a tikz picture.
  label:=true puts principal type label underneath.
  sort:=true sorts the types by Weight first, in decreasing order.
  yshift:="default" changes y by 2 (with label) / 1.2 (without label) after every row
  width:=10 puts 10 principal types in every row
  scale:=0.8 controls tikz picture global scale
```

**Example** (TeX table of principal types).

```
> list:=PrincipalTypes(-1); // All 13 principal types with chi=-1, sorted
> TeX(list: label, width:=7, yshift:=2.2);    //    (10 Kodaira + 3 'exotic')
```



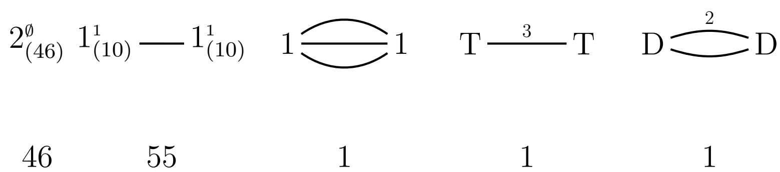

## 2.11 Shapes (RedShape)

```
type RedShape
```

A reduction type a graph whose vertices are principal types (type RedPrin) and edges are link chains. They fall naturally into 'shapes', where every vertex only remembers the Euler characteristic $\chi$ of the type, and edge the gcd of the chain. Thus, the problem of finding all reduction types in a given genus (see ReductionTypes) reduces to that of finding the possible shapes (see Shapes) and filling in shape components with given $\chi$ and gcds of loose edges (see PrincipalTypes).

**Example**. Here is how this works in genus 2. The 104 families of reduction types break into five possible shapes, with all but three types in the first two shape (46 and 55 types, respectively):

```
> L:=Shapes(2);
> &cat [TeX(D[1]: shapelabel:=Sprint(D[2])): D in L];
```




A shape is represented by a Magma type RedShape with the following invariants:

```
declare type RedShape;
declare attributes RedShape:
  G,       // Underlying undirected graph with vertices labelled by [chi]
           //   and edges by [lgcd1,lgcd2,...] (gcds are sorted)
  V,       // Vertex set of G
  E,       // Edge set of G
  D,       // Double graph: vertex for every vertex of G, and for every edge
           //   of G except simple edges with lgcd=[1]. Edges are unlabelled,
           //   and D determines the shape up to isomorphism.
  label;   // Label based on minimum path, determines the shape up to isomorphism.
```

## 2.12 Printing and TeX

```
intrinsic Print(S::RedShape, level::MonStgElt)
```

Print a shape as Shape(vertices,edges) so that the shape can be reconstructed. Vertices are '-chi' of principal types, and edges are of the form [from_vertex,to_vertex,gcd1,gcd2,...] with gcd_i the gcd's of the link chains between principal types

**Example** (Printing a shape).

```
> Shape(ReductionType("IV-IV-IV"));    // 3 vertices with chi=-1,-2,-1 and 2 edges
Shape([1,2,1],[[1,2,1],[2,3,1]])
> Shape(ReductionType("1---1"));       // 2 vertices with chi=-1,-1 and a triple edge
Shape([1,1],[[1,2,1,1,1]])
```

```
intrinsic TeX(S::RedShape: scale:=1.5, center:=false, shapelabel:="",
  complabel:="default", boundingbox:=false) -> MonStgElt, FldReElt, FldReElt,
  FldReElt, FldReElt
```

Tikz picture for a shape S of a reduction graph, or, if boundingbox:=true, returns S,x1,y1,x2,y2, where the last four define the bounding box.

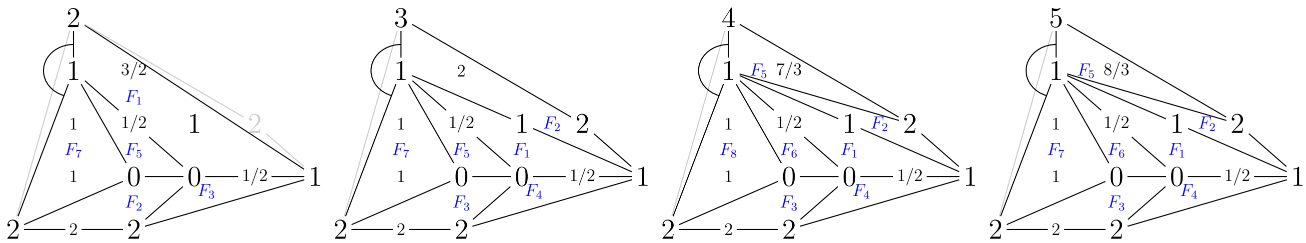
**Example** (Reduction types in a family of curves). We look at curves $p^n xy^4 = x^2(1+x)y + pxy(x^4 + x^2y+y^2) + p^2(1+x^2+x^4y^2)$ for $p = 7$ and $n \geq 3$.

```
> _<x,y>:=PolynomialRing(Q,2);
> p:=7;
```

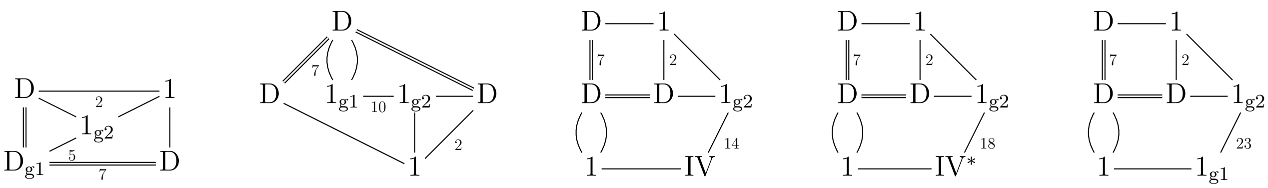
```
> f:=func<n| p^n*x*y^4=x^2*(1+x)*y+p*x*y*(x^4+x^2*y+y^2)+p^2*(1+x^2+x^4*y^2) >;
> M:=func<n| Model(f(n),p) >;                    // Model
> R:=func<n| ReductionType(M(n)) >;              //   and Reduction type as a function of n
```

The curves are $\Delta_v$-regular and the shape of $\Delta_v$ is unchanged as long as $n > 3$, with only the height of one vertex being affected. For $n \leq 3$ some of the faces merge:

```
> [DeltaTeX(M(n)): n in [2..5]];
```

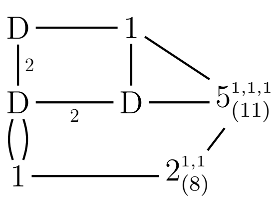


```
> [TeX(R(n)): n in [2..6]];
```



For $n > 3$ the shape of the reduction type remains the same:

```
> TeX(Shape(R(6)));
```



## 2.13 Construction and isomorphism testing

```
intrinsic Shape(V::SeqEnum[RngIntElt], E::SeqEnum[SeqEnum[RngIntElt]]) ->
  RedShape
```

```
  Constructs a graph shape from the data V,E as in shapes*.txt data files:
     V = sequence of -chi's for individual components
     E = list of edges v_i->v_j of the form [i,j,edgegcd1,edgegcd2,...]
```

```
intrinsic IsIsomorphic(S1::RedShape, S2::RedShape) -> BoolElt
```

```
  Check whether two shapes are isomorphic via their double graphs
```

**Example** (Shape isomorphism testing).

```
> S1:=Shape([1,2,3],[[1,2,3],[2,3,1],[1,3,2]]);
> S2:=Shape([2,3,1],[[1,2,1],[2,3,2],[1,3,3]]);   // rotate the graph
> assert IsIsomorphic(S1,S2);
> S3:=Shape(VertexLabels(S1),EdgeLabels(S1));     // reconstruct S1 from labels
> assert IsIsomorphic(S1,S3);
```

## 2.14 Primary invariants

```
intrinsic Graph(S::RedShape) -> GrphUnd
```

Labelled underlying graph G of the shape

```
intrinsic DoubleGraph(S::RedShape) -> GrphUnd
```

Vertex-labelled double graph D of the shape, used for isomorphism testing

```
intrinsic Vertices(S::RedShape) -> SetIndx
```

Vertices of the underlying graph Graph(S), as an indexed set

```
intrinsic Edges(S::RedShape) -> SetIndx
```

Edges of the underlying graph Graph(S), an an indexed set

```
intrinsic Chi(S::RedShape, v::GrphVert) -> RngIntElt
```

Euler characteristic chi(v_i)<=0 of ith vertex of the graph G in a shape S

```
intrinsic LGCDs(S::RedShape, v::GrphVert) -> RngIntElt
```

LGCDs of a vertex v that together with chi determine the vertex type (chi, lgcds)

```
intrinsic Chi(S::RedShape) -> RngIntElt
```

Total Euler characteristic of a graph shape chi<=0, sum over chi's of vertices

```
intrinsic VertexLabels(S::RedShape) -> SeqEnum
```

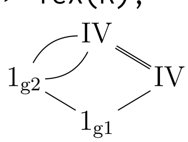
Sequence of -chi's for individual components of the shape S so that
S=Shape(VertexLabels(S),EdgeLabels(S))

```
intrinsic EdgeLabels(S::RedShape) -> SeqEnum
```

List of edges v_i->v_j of the form [i,j,edgegcd] so that S=Shape(VertexLabels(S),EdgeLabels(S))

**Example** (Graph, DoubleGraph and primary invariants for shapes). Under the hood of shapes of reduction types are their labelled graphs and associated 'double' graphs. As an example, take the following reduction type:

```
> R:=ReductionType("1g2--IV=IV-1g1-c1");
> TeX(R);
```

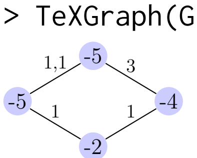



There are four principal types, and they become vertices of Shape(R) whose labels are their Euler characteristics $-5, -2, -4, -5$. The edges are labelled with GCDs of the link chain between the types. For example:

— the link chain 1g2-1g1 of gcd 1 becomes the label "1",
— the link chain IV=IV of gcd 3 becomes "3",
— the two chains 1g2–IV of gcd 1 become "1,1"

on the corresponding edges.

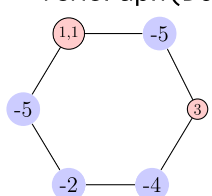
```
> S:=Shape(R); S;
Shape([5,2,4,5],[[1,2,1],[1,4,1,1],[2,3,1],[3,4,3]])
> TeXGraph(Graph(S): scale:=1);
```



```
> Vertices(S);          // Indexed set of vertices of Graph(S)
{@ 1, 2, 3, 4 @}
> Edges(S);             //   and edges {@ {from_vertex, to_vertex}, ... @}
{@ {1, 2}, {1, 4}, {2, 3}, {3, 4} @}
> VertexLabels(S);      // [-chi] for each type
[5,2,4,5]
> EdgeLabels(S);        // [ [from_vertex, to_vertex, gcd1, gcd2, ...], ...]
[[1,2,1],[1,4,1,1],[2,3,1],[3,4,3]]
```

Both Magma's IsIsomorphic for graphs and MinimumWeightPaths are implemented for graphs with labelled vertices but not edges. To use them for shapes, the underlying graphs are converted to graphs with only labelled vertices. This is done simply by introducing a new vertex on every edge which carries the corresponding edge label. For compactness, if the label is "1" (most common case), we don't introduce the vertex at all. This is called the double graph of the shape:

```
> blue:="circle,scale=0.7,inner sep=2pt,fill=blue!20";      // former vertices
> red:="circle,draw,scale=0.5,inner sep=2pt, fill=red!20";  // former edges
> bluered:=func<v|&+Label(v) le 0 select blue else red>;
>
> TeXGraph(DoubleGraph(S): scale:=1, vertexnodestyle:=bluered);
```




These are used in isomorphism testing for shapes, and to construct minimal paths.

```
intrinsic WeightIsSmaller(new::SeqEnum, best::SeqEnum) -> MonStgElt
```

```
Compares two sequences of integers, and returns "<", ">", "l", "s", "=":
  <=smaller   : new has smaller weight than best
  >=greater   : new has greater weight
  l=longer    : new and best coincide until #best, and new is longer
  s=shorter   : new and best coincide until #new,  and new is shorter
  ==identical : new=best
```

```
intrinsic MinimumWeightPaths(D::GrphUnd) -> SeqEnum, SeqEnum
```

```
Minimum weight paths for a labelled undirected graph (e.g. double graph underlying shape)
returns W=bestweight [<index, v_label, jump>,...] (characterizes D up to isomorphism)
  and I=list of possible vertex index sequences
For example for a rectangular loop G with all vertex chis=-1 and edges as follows
  V:=[1,1,1,1]; E:=[[1,2,1],[2,3,1],[3,4,2],[1,4,1,1]]; S:=Shape(V,E);
the double graph D has 6 vertices and 6 edges in a loop, and here minimum weight W is
  W = [<0,[-1],false>,<0,[-1],false>,<0,[-1],false>,<0,[1,1],false>,<0,[-1],false>,
       <0,[2],false>,<1,[-1],true>]
The unique trail T[1] (generally Aut D-torsor) is D.3->D.2->D.1->...->D.3, encoded
  T = [[3,2,1,6,4,5,3]]
```

```
intrinsic Label(G::GrphUnd) -> MonStgElt
```

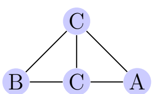
```
Graph label based on a minimum weight path, determines G up to isomorphism
```

```
intrinsic MinimumWeightPaths(S::RedShape) -> SeqEnum, SeqEnum
```

```
  Minimum weight paths for a shape, computed through its double graph and refers
  to its vertices and edges.
  Returns W=bestweight [<index, v_label, jump>,...] (characterizes D up to isomorphism)
    and I=list of possible vertex index sequences
  For example for a rectangular loop G with all vertex chis=-1 and edges as follows
    V:=[1,1,1,1]; E:=[[1,2,1],[2,3,1],[3,4,2],[1,4,1,1]]; S:=Shape(V,E);
  the double graph D has 6 vertices and 6 edges in a loop, and here minimum weight W is
    W = [<0,[-1],false>,<0,[-1],false>,<0,[-1],false>,<0,[1,1],false>,<0,[-1],false>,
         <0,[2],false>,<1,[-1],true>]
  The unique trail T[1] (generally Aut D-torsor) is D.3->D.2->D.1->...->D.3, encoded
    T = [[3,2,1,6,4,5,3]]
```

**Example** (MinimumWeightPaths).

```
> G:=Graph<4|{{1,2},{2,3},{3,4},{4,1},{1,3}}>;     // labelled graph on four vertices
> AssignLabels(Vertices(G),["C","B","C","A"]);    // v1(C), v2(B), v3(C), v4(A)
> TeXGraph(G);
```



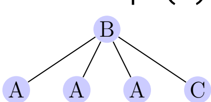

```
> P,a:=MinimumWeightPaths(G);
```

All shortest paths start and end in a C vertex (Eulerian path), and the minimal path is C–A–C–B–1–3. Note that C–A–C–1–B–2 is also a valid path, but it is not minimal. By our convention, vertex labels (B) precede used vertex indices (1) in the lexicographic ordering used to define the minimal path.

```
> P;
[<0,"C",false>,<0,"A",false>,<0,"C",false>,<0,"B",false>,<1,"C",false>,<3,"C",true>]
> Label(G);              // Graph label derived from minimal path
C-A-C-B-c1-c3
```

Here is another graph on five vertices, this time not Eulerian:

```
> G:=Graph<5|{{2,1},{2,3},{2,4},{2,5}}>;
> AssignLabels(Vertices(G),["A","B","A","A","C"]);
> TeXGraph(G);
```




```
> SetVerbose("redlib",0);
> P,a:=MinimumWeightPaths(G);         // Minimal path is A-B-A&A-2-C
> P;
[<0,"A",false>,<0,"B",false>,<0,"A",true>,<0,"A",false>,<2,"B",false>,<0,"C",true>]
```

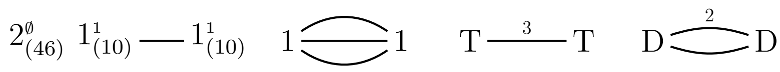
There are 6 ways to trace this path, and they form an Aut($G$)=$S_3$-torsor. The first one is

$$v_1 \mapsto v_2 \mapsto v_3 \mapsto v_4 \mapsto v_2 \mapsto v_5$$

```
> a;
[[1,2,3,4,2,5],[1,2,4,3,2,5],[3,2,1,4,2,5],[3,2,4,1,2,5],[4,2,3,1,2,5],[4,2,1,3,2,5]]
> GroupName(AutomorphismGroup(G));
S3
> Label(G);              // Graph label derived from minimal path
A-B-A&A-c2-C
```

**Example** (Shapes). Here is a table of all genus 2 shapes, with numbers of reduction types for each one:

```
> L:=Shapes(2);
> &cat [TeX(D[1]: shapelabel:=Sprint(D[2])): D in L];
```




46 55 1 1 1

The total is 104, the number of genus 2 reduction types families.

## 2.15 Reduction Types (RedType)

Now we come to reduction types, implemented through the following type RedType:

```
declare type RedType;
declare attributes RedType:
  C,       // array of principal types of type RedPrin, ordered in label order
           // either one with chi=0 (for g=1) or all with chi<0.
  L,       // all link chains, sorted as for label, of type SeqEnum[RedLink]
  weight,  // weight used for comparison and sorting
  shape,   // shape of R of type RedShape
  bestweight,     // e.g. [<0,{*-1*},true>,<0,{*-2*},true>,<0,{*-1*},false>,...
                  // constructed with MinimumWeightPaths, used in canonical label
  besttrail;      // e.g. [1,2,3,4,1,3] tracing vertices with repetitions.
```

They can be constructed in a variety of ways:

| | |
|---|---|
| `ReductionType(m,g,O,L)` | Construct from a sequence of components (including all principal ones), their multiplicities m, genera g, outgoing multiplicities of open chains O, and link chains L beween them, e.g.<br>`ReductionType([1],[0],[[]],[[1,1,0,0,3]]);` (Type $I_3$) |
| `ReductionTypes(g)` | All reduction types in genus g. Can restrict to just semistable ones and/or ask for their count instead of actual the types, e.g.<br>`ReductionTypes(2);` (all 104 genus 2 types)<br>`ReductionTypes(2: countonly);` (only count them)<br>`ReductionTypes(2: semistable);` (7 semistable ones) |
| `ReductionType(label)` | Construct from a canonical label, e.g.<br>`ReductionType("I3");` |
| `ReductionType(G)` | Construct from a dual graph, e.g.<br>`ReductionType(DualGraph([1],[1],[]));` (good elliptic curve) |
| `ReductionTypes(S)` | Reduction types with a given shape, e.g.<br>`ReductionTypes(Shape([2],[]));` (46 of the genus 2 types) |

Conversely, from a reduction type we can construct its dual graph (`DualGraph`) and a canonical label `Label`), and these functions are also described in this section. Finally, there are functions to draw reduction types and their dual graphs in TeX (`TeX`).

```
type RedType
```

```
intrinsic Print(R::RedType, level::MonStgElt)
```

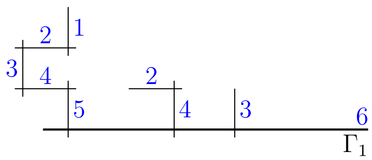
Print a reduction type through its Label.

```
intrinsic ReductionType(m::SeqEnum[RngIntElt], g::SeqEnum[RngIntElt],
  O::SeqEnum[SeqEnum], L::SeqEnum[SeqEnum]) -> RedType
```

```
Construct a reduction type from a sequence of components, their invariants, and chains of P1s:
  m = sequence of multiplicities of components c_1,...,c_k
  g = sequence of their geometric genera
  O = outgoing multiplicities of open chains, one sequence for each component
  L = link chains, of the form
      [[i,j,di,dj,n],...] - link chain from c_i to c_j with multiplicities m[i],di,...,dj,m[j], of
      depth n
      n can be omitted, and chain data [i,j,di,dj] is interpreted as having minimal possible depth.
```

**Example** (Type II*).

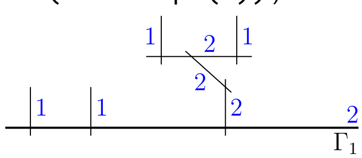
```
> m:=[6];            // multiplicities of starting components
> g:=[0];            // their geometric genera
> O:=[[3,4,5]];      // outgoing multiplicities of open chains from each of them
> L:=[];             // link chains
> R:=ReductionType(m,g,O,L);
> R, TeX(DualGraph(R));
```




II*

**Example** (Type I3*).

```
> m:=[2,2];             // multiplicities of starting components Gamma_1, Gamma_2
> g:=[0,0];             // their geometric genera
> O:=[[1,1],[1,1]];     // outgoing multiplicities of open chains from each of them
> L:=[[1,2, 2,2, 3]];   // link chains [[i,j, di,dj ,optional depth],...]
> R:=ReductionType(m,g,O,L);
> R, TeX(DualGraph(R));
```




$I_3^*$

```
intrinsic ReductionTypes(g::RngIntElt: semistable:=false, countonly:=false,
  elliptic:=false) -> SeqEnum[RedType]
```

```
All reduction types in genus g<=6 or their count (if countonly:=true; faster).
semistable:=true restricts to semistable types, elliptic:=true (when g=1) to Kodaira types of
elliptic curves.
```

**Example**.

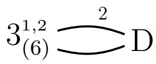
```
> ReductionTypes(1: elliptic);                // 10 Kodaira types of elliptic curves
[1g1,I1,I0*,I1*,IV,IV*,III,III*,II,II*]
> ReductionTypes(2: countonly);               // Genus 2 count
104
> ReductionTypes(3: semistable, countonly);   // Genus 3 semistable count
42
```

```
intrinsic ReductionTypes(S::RedShape: countonly:=false, semistable:=false) ->
  SeqEnum[RedType]
```

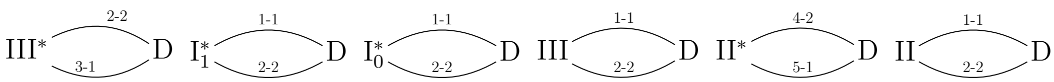
```
Sequence of reduction types with a given shape. If countonly=true, only count their number
```

**Example** (Reduction types with a given shape). There are 1901 reduction types in genus 3, in 35 different shapes. Here is one of the more ‘exotic’ ones, with 6 types in it. It has two vertices with $\chi = -3$ and $\chi = -1$ and two edges between them, with gcd 1 and 2.

```
> S:=Shape([3,1],[[1,2,1,2]]);
> TeX(S);
```




```
> L:=ReductionTypes(S); L;
[
III*-{2-2}-D,
I1*-=D,
I0*-=D,
III--{2-2}D,
II*-{4-2}-D,
II--{2-2}D
]
> &cat [TeX(R: scale:=1.5, forcesups): R in L];
```




## 2.16 Arithmetic invariants

```
intrinsic Chi(R::RedType) -> RngIntElt
```

Total Euler characteristic of R

```
intrinsic Genus(R::RedType) -> RngIntElt
```

Total genus of R

**Example**.

```
> R:=ReductionType("III=(3)III-{2-2}II-{6-12}18g2^6,12");
> Label(R);      // Canonical label
[6]Tg2-{12-6}II-{2-2}III=(3)III
> Genus(R);      // Total genus
43
```

```
intrinsic IsGood(R::RedType) -> BoolElt
```

true if comes from a curve wih good reduction

```
intrinsic IsSemistable(R::RedType) -> BoolElt
```

true if comes from a curve with semistable reduction (all (principal) components of an mrnc model have multiplicity 1)

```
intrinsic IsSemistableTotallyToric(R::RedType) -> BoolElt
```

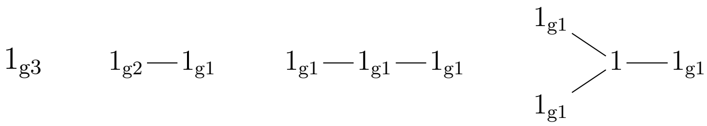
true if comes from a curve with semistable totally toric reduction (semistable with no positive genus components)

```
intrinsic IsSemistableTotallyAbelian(R::RedType) -> BoolElt
```

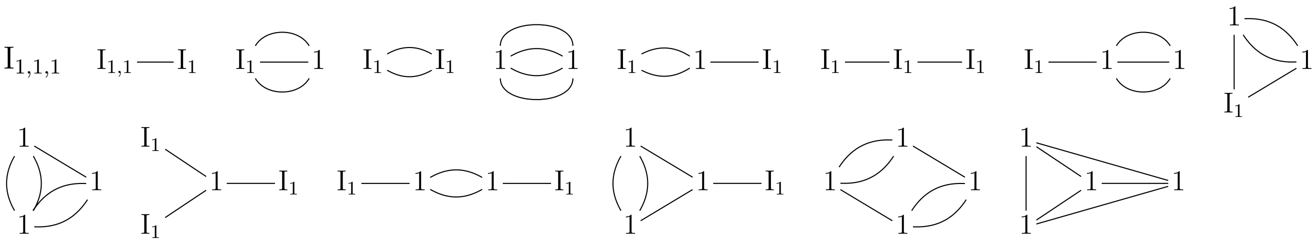
true if comes from a curve with semistable totally abelian reduction (semistable with no loops in the dual graph)

**Example** (Semistable reduction types).

```
> semi:=ReductionTypes(3: semistable);                // genus 3, semistable,
> ab:=[R: R in semi | IsSemistableTotallyAbelian(R)];  //   totally abelian reduction
> [TeX(R): R in ab];
```




```
> tor:=[R: R in semi | IsSemistableTotallyToric(R)];
> #tor;                                               //   totally toric reduction
15
> [TeX(R): R in tor];
```



Count semistable reduction types in genus 2,3,4,5

```
> [ReductionTypes(n: semistable, countonly): n in [2..5]];   // OEIS A174224
[ 7, 42, 379, 4555 ]
```

```
intrinsic TamagawaNumber(R::RedType) -> RngIntElt
```

Tamagawa number of the curve with a given reduction type, over an algebraically closed residue field.

**Example** (Tamagawa numbers for elliptic curves).

```
> for R in ReductionTypes(1: elliptic) do Label(R),TamagawaNumber(R); end for;
1g1 1
I1 1
I0* 4
I1* 4
IV 3
IV* 3
III 2
III* 2
II 1
II* 1
```

## 2.17 Invariants of individual principal components and chains

```
intrinsic PrincipalTypes(R::RedType) -> SeqEnum[RedPrin]
```

Principal types (vertices) R of the reduction type R

```
intrinsic PrincipalType(R::RedType, i::RngIntElt) -> RedPrin
```

Principal type number i in the reduction type R, same as R!!i

```
intrinsic LinkChains(R::RedType) -> SeqEnum[RedLink]
```

Return all the link chains in R, including loops and D-links, as a sequence SeqEnum[RedLink], sorted as in label

```
intrinsic LooseChains(R::RedType) -> SeqEnum[RedLink]
```

Return all the link chains in R between different principal components, as a sequence SeqEnum[RedLink], sorted as in label

```
intrinsic Multiplicities(R::RedType) -> SeqEnum
```

Sequence of multiplicities of principal types

```
intrinsic Genera(R::RedType) -> SeqEnum
```

Sequence of geometric genera of principal types

```
intrinsic GCD(R::RedType) -> RngIntElt
```

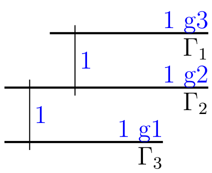
GCD detecting non-primitive types

```
intrinsic Shape(R::RedType) -> RedShape
```

The shape of the reduction type R. Every principal type is a vertex that only remembers its Euler characteristic, and every edge only remembers the gcd of the corresponding link chain

**Example** (Principal types and chains). Take a reduction type that consists of smooth curves of genus 3, 2 and 1, connected with two chains of $\mathbb{P}^1$s of depth 2.

```
> R:=ReductionType("1g3-(2)1g2-(2)1g1");
> TeX(DualGraph(R));
```




This is how we access the three principal types, their primary invariants, and the chains. Both the principal types and the chains are ordered as in the canonical label.

```
> R!!1, R!!2, R!!3;      // individual principal types, same as PrincipalTypes(R)
1g3-{1}
1g2-{1}-{1}
1g1-{1}
> Genera(R);             // geometric genus g of each principal type
[ 3, 2, 1 ]
> Multiplicities(R);     // multiplicity m of each principal type
[ 1, 1, 1 ]
> LinkChains(R);         // all chains between them (including loops and D-links)
[
[1] loose c1 1,1 -(2) c2 1,1,
[2] loose c2 1,1 -(2) c3 1,1
]
```

## 2.18 Comparison

```
intrinsic Weight(R::RedType) -> SeqEnum[RngIntElt]
```

Weight of a reduction type, used for comparison and sorting

```
intrinsic 'eq'(R1::RedType, R2::RedType) -> BoolElt
```

Compare two reduction types by their weight

```
intrinsic 'lt'(R1::RedType, R2::RedType) -> BoolElt
```

Compare two reduction types by their weight

```
intrinsic 'gt'(R1::RedType, R2::RedType) -> BoolElt
```

Compare two reduction types by their weight

```
intrinsic 'le'(R1::RedType, R2::RedType) -> BoolElt
```

Compare two reduction types by their weight

```
intrinsic 'ge'(R1::RedType, R2::RedType) -> BoolElt
```

Compare two reduction types by their weight

```
intrinsic Sort(S::SeqEnum[RedType]) -> SeqEnum[RedType]
```

Sort reduction types by their weight

```
intrinsic Sort(~S::SeqEnum[RedType])
```

Sort reduction types by their weight

**Example** (Sorted reduction types in genus 1 and 2).

```
> Sort(ReductionTypes(1: elliptic));
1g1, I1, I0*, I1*, IV, IV*, III, III*, II, II*
> Sort(ReductionTypes(2));
1g2, I1g1, I1,1, Dg1, [2]g1_D, 2^1,1,1,1,1,1, I0*_0, D_{2-2}, I0*_D, I1*_0, [2]_1,D,
  I1*_D, [2]_D,D,D, 3^1,1,2,2, IV_0, IV*_-1, 4^1,3,2,2, III_0, III*_-1, III_D, 4^1,3_D,
  III*_D, [2]I0*_D, [2]I1*_D, 5^1,1,3, 5^1,2,2, 5^2,4,4, 5^3,3,4, 6^1,1,4, 6^5,5,2,
  6^2,4,3,3, II_D, [2]IV_D, [2]T_{6}D, [2]IV*_D, II*_D, 8^1,3,4, 8^5,7,4, [2]III_D,
  [2]III*_D, 10^1,4,5, 10^3,2,5, 10^7,8,5, 10^9,6,5, [2]II_D, [2]II*_D, 1g1-1g1, 1g1-I1,
  1g1-I0*, 1g1-I1*, 1g1-IV, 1g1-IV*, 1g1-III, 1g1-III*, 1g1-II, 1g1-II*, I1-I1, I1-I0*,
  I1-I1*, I1-IV, I1-IV*, I1-III, I1-III*, I1-II, I1-II*, I0*-I0*, I0*-I1*, I0*-IV,
  I0*-IV*, I0*-III, I0*-III*, I0*-II, I0*-II*, I1*-I1*, I1*-IV, I1*-IV*, I1*-III,
  I1*-III*, I1*-II, I1*-II*, IV-IV, IV-IV*, IV-III, IV-III*, IV-II, IV-II*, IV*-IV*,
  IV*-III, IV*-III*, IV*-II, IV*-II*, III-III, III-III*, III-II, III-II*, III*-III*,
  III*-II, III*-II*, II-II, II-II*, II*-II*, T=T, D-=D, 1---1
```

## 2.19 Reduction types, labels, and dual graphs

```
intrinsic ReductionType(G::GrphDual) -> RedType
```

Create a reduction type from a full dual mrnc graph or return false if G is singular

```
intrinsic DualGraph(R::RedType: compnames:="default") -> GrphDual
```

Full dual graph from a reduction type, possibly with variable length edges

```
intrinsic Label(R::RedType: tex:=false, html:=false, wrap:=true,
  forcesubs:=false, forcesups:=false, depths:="default") -> MonStgElt
```

```
Return canonical string label of a reduction type.
     tex:=true       gives a TeX-friendly label (\redtype ...)
     html:=true      gives a HTML-friendly label (<span...>...</span>)
     wrap:=false     keeps the format above but removes \redtype wrapping
     forcesubs:=true forces lengths of chains and loops to be always printed (usually in round
     brackets)
     forcesups:=true forces outgoing chain multiplicities to be always printed (in curly brackets).
```

```
intrinsic Family(R::RedType) -> MonStgElt
```

Reduction type with minimal chain lengths in the same family

```
intrinsic ReductionType(S::MonStgElt) -> RedType
```

Construct a reduction type from a string label.

**Example** (Plain and TeX labels for reduction types).

```
> R:=ReductionType("IIg1_1-(3)III-(4)IV");
> Label(R);                 // plain text label
IIg1_1-(3)III-(4)IV
> R2:=ReductionType(Label(R));
> assert R eq R2;           // can be used to reconstruct the type
> Family(R);                // family (reduction type with minimal depths)
IIg1_1-III-IV
> Label(R: tex);            // print label in TeX, wrap in \redtype{...} macro
```

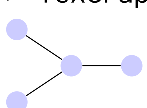
$\mathrm{II}_{g1,1}\,{}_{3}\,\mathrm{III}\,{}_{4}\,\mathrm{IV}$

```
> Label(R: html);           // print label in HTML, wrap in redtype span
II<sub>g1,1</sub><span class='edg'><sup> </sup><sub>3</sub></span>III<span
  class='edg'><sup> </sup><sub>4</sub></span>IV
> R!!1;                     // first principal type as a standalone type
IIg1_1-{1}
> Label(R!!1);              // first principal type: label in R
IIg1_1
> Label(R!!1: tex);         // first principal type: TeX label
```

$\mathrm{II}_{g1,1}$

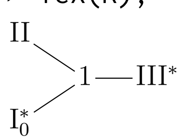
**Example** (Canonical label in detail). Take a graph $G$ on 4 vertices

```
> G:=Graph<4|{{1,2},{1,3},{1,4}}>;
> TeXGraph(G: labels:="none");
```

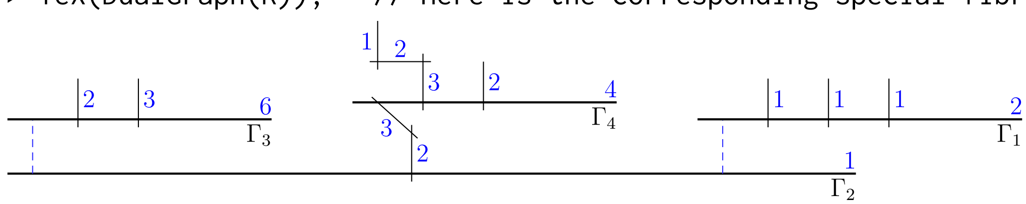


Place a component of multiplicity 1 at the root and II, III*, $\mathrm{I}_0^*$ at the three leaves. Link each leaf to the root with a chain of multiplicity 1. This gives a reduction type that occurs for genus 3 curves:

```
> R:=ReductionType("1-II&c1-III*&c1-I0*");    // First component is the root,
> TeX(R);                                     //   the other three are leaves
```




```
> TeX(DualGraph(R));   // Here is the corresponding special fibre
```




How is the following canonical label chosen among all possible labels?

```
> R;
I0*-1-II&III*-c2
```

Each principal component is a principal type (as there are no loops or D-links), and its primary invariants are its Euler characteristic $\chi$ and a multiset lgcd of gcd's of outgoing (loose) link chains

```
> [R!!i: i in [1..#R]];
[I0*-{1},1-{1}-{1}-{1},II-{1},III*-{3}]
> [Chi(R!!i): i in [1..#R]];      // add up to 2-2*genus, so genus=3
[ -1, -1, -1, -1 ]
> [LGCD(R!!i): i in [1..#R]];
[[1],[1,1,1],[1],[1]]
```

All four leaves have $\chi = -2$, lgcd=[1] and the root $\chi = 1$, lgcd=$[1, 1, 1]$

```
> PrincipalTypes(-1,[1]);        // 10 such (II-, III-, IV-, ...) drawn $1^1_{(10)}$
[1g1-{1},I1-{1},I0*-{1},I1*-{1},IV-{1},IV*-{2},III-{1},III*-{3},II-{1},II*-{5}]
> PrincipalTypes(-1,[1,1,1]);    // unique one of this type, drawn as 1
[1-{1}-{1}-{1}]
```

Together they form a shape graph $S$ as follows:

```
> S:=Shape(R);
> TeX(S: scale:=1);
```

$1^1_{(10)}$

$\rangle 1 — 1^1_{(10)}$

$1^1_{(10)}$

The vertices and edges of $S$ are assigned weights. Vertex weights are $\chi$'s, edge weights are lgcd's

```
> [Label(v): v in Vertices(S)];
[[-1],[-1],[-1],[-1]]
> [Label(e): e in Edges(S)];
[[1],[1],[1]]
```

Then the shortest path is found using MinimumWeightPaths. It is v-v-v&v-2 (v=new vertex with $\chi = -1$, -=edge, &=jump). Note that by convention actual edges are preferred to jumps, and going to a new vertex preferred to revisiting an old one. Also vertices with smaller $\chi$ come first, if possible, as they have smaller labels.

v-v-v&v-2 $<$ v-v&v-2-v (jumps are larger than edge marks)
v-v-v&v-2 $<$ v-v-v&2-v (repeated vertex indices are larger than vertex marks)

```
> P,T:=MinimumWeightPaths(S);
> P;      // v-v-v&v-2
[<0,[-1],false>,<0,[-1],false>,<0,[-1],true>,<0,[-1],false>,<2,[-1],true>]
```

This path can be used to construct the graph, and determines it up to isomorphism. There are $|\operatorname{Aut} S| = 6$ ways to trail $S$ in accordance with this path, and as far the shape is concerned, they are completely identical.

```
> T;
[[1,2,3,4,2],[1,2,4,3,2],[3,2,1,4,2],[3,2,4,1,2],[4,2,3,1,2],[4,2,1,3,2]]
```

This gives six possible labels for our reduction type that all traverse the shape according to path $P$:

```
> t:=[Label(R!!i): i in [1..#R]];
> [Sprintf("%o-%o-%o&%o-c2",t[c[1]],t[c[2]],t[c[3]],t[c[4]]): c in T];
I0*-1-II&III*-c2 I0*-1-III*&II-c2 II-1-I0*&III*-c2 II-1-III*&I0*-c2 III*-1-II&I0*-c2
  III*-1-I0*&II-c2
```

Now we assign weights to vertices and edges that characterise the actual shape components (rather than just their $\chi$) and link chains (rather than just their lgcd)

```
> Weight(R!!1), Weight(R!!2), Weight(R!!3), Weight(R!!4);
[ -1, 2, 0, 1, 0, 0, 3, 1, 1, 1, 1 ] [ -1, 1, 0, 3, 0, 0, 0, 1, 1, 1 ] [ -1, 6, 0, 1, 0,
  0, 2, 2, 3, 1 ] [ -1, 4, 0, 1, 0, 0, 2, 3, 2, 3 ]
> EdgesWeight(R,2,1);    // weight of the 1-II link chain
[ 1, 1, 0 ]
> EdgesWeight(R,2,3);    // weight of the 1-I0* link chain
[ 1, 1, 0 ]
> EdgesWeight(R,2,4);    // weight of the 1-III* link chain
[ 1, 3, 0 ]
```

The component weight Weight(R!!i) starts with $(\chi, -m, -g, ...)$ so when all components have the same $\chi$ like in this example, the ones with large multiplicity $m$ have smaller weight. Because m(II)=6, m(III*)=4, m(I0*)=2, the trails $T[1]$ and $T[2]$ are preferred to the other four. They both start with a component II, then an edge II-1 and a component 1. After that they differ in that $T[1]$ traverses an edge 1-I0* and $T[2]$ an edge 1-III*. Because the edge weight is smaller for $T[1]$, this is the minimal path, and it determines the label for $R$:

```
> R;
I0*-1-II&III*-c2
```

**Example** (Labels of individual principal types).

```
> R:=ReductionType("II-III-IV");
> [Label(R!!i): i in [1..#R]];
[ IV, III, II ]
```

```
intrinsic LabelRegex(R::RedType: magma:=true) -> MonStgElt
```

```
Returns a regular expression that recognises reduction types in the same family as R and captures
the corresponding edge depths. For example,
  LabelRegex(ReductionType("Dg1_1"));
returns ^Dg1_([0-9]+)$, which is a regular expression that matches Dg1_n for any n>=0 and returns n
in the captured group. Flag magma:=true makes the returned regex compatible with Magma's Regexp
function (which is old V8) but may have brackets around the returned integers. Setting magma:=false
makes it compatible with all recent regex implementations, and only returns pure integers in
captured groups.
```

**Example**.

```
> R:=ReductionType("III-II");
> re:=LabelRegex(R); re;
^III-([(][0-9]+[)])?II$
```

This regex matches III-II or III-(2)II which are in the correct format, but not II-2III which is not

```
> ok,_,B:=Regexp(re,"III-II"); ok, B;            // Yes
true []
> ok,_,B:=Regexp(re,"III-(2)II"); ok, B;         // Yes
true [ (2) ]
> Regexp(re,"III-2II");                          // No
false
```

B contains the captured lengths, possibly in brackets (as above), and [eval b: b in B] gives them as integers. The reason for the brackets is that Magma uses old (V8) regex format that does not support non-capturing groups. Calling

```
> LabelRegex(R: magma:=false);
^III-(?:[(]([0-9]+)[)])?II$
```

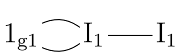
returns a newer regex format (supported in python, javascript etc.) that has the same behaviour but just captures integer lengths.

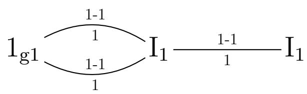
```
intrinsic TeX(R::RedType: forcesups:=false, forcesubs:=false, scale:=0.8,
  xscale:=1, yscale:=1, oneline:=false) -> MonStgElt
```

```
TikZ representation of a reduction type, as a graph with PrincipalTypes (principal components with
chi>0) as vertices, and edges for link chains.
oneline:=true removes line breaks.
forcesups:=true and/or forcesubs:=true shows edge decorations (outgoing multiplicities and/or chain
depths) even when they are default.
```

**Example** (TeX for reduction types).

```
> R:=ReductionType("1g1--I1-I1");
> TeX(R), TeX(R: forcesups, forcesubs, scale:=1.5);
```



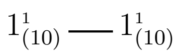




**Example** (Degenerations of two elliptic curves meeting at a point).

```
> S:=Shape(ReductionType("1g1-1g1")); // Two elliptic curves meeting at a point (genus 2)
```

The corresponding shape is a graph v-v with two vertices with $\chi = -1$ and one edge of gcd 1

```
> TeX(S);
```




```
> PrincipalTypes(-1,[1]);                // There are 10 possibilities for such a vertex,
[1g1-{1},I1-{1},I0*-{1},I1*-{1},IV-{1},IV*-{2},III-{1},III*-{3},II-{1},II*-{5}]
>                                         // one for each Kodaira type
> ReductionTypes(S: countonly);           // and Binomial(10,2) such types in total
55
> ReductionTypes(S)[[1..10]];             // first 10 of these
[I1*-III*,I1*-II,III-II,III*-II,I1-II*,1g1-II*,1g1-I1,I1*-II*,IV-II*,III-III*]
```

## 2.20 Variable depths in Label and DualGraph

Reduction types belong to the same family if they are the same apart except that the depths of chains of $\mathbb{P}^1$s may differ. This section describes functions to print labels and draw dual graphs of families of reduction types with variable depths.

```
intrinsic SetDepths(~R::RedType, depth::UserProgram)
```

```
Set depths for DualGraph and Label to be determined by depth function.
  depth has to be of the form
    function depth(e::RedLink) -> integer/string
  to show how the depth in the edge is to be printed
  For example,
    f(e) = e`depth                              [ original as in SetDepths(R,true) ]
    f(e) = MinimalLinkDepth(e`mi,e`di,e`mj,e`dj)  [ minimal  as in SetDepths(R,false) ]
    f(e) = Sprintf("n_%o",e`index)              [ "n_1","n_2",...]
```

```
intrinsic SetDepths(~R::RedType, S::SeqEnum)
```

```
Set depths for DualGraph and Label to a sequence, e.g. S=["m","n","2"]
```

```
intrinsic SetVariableDepths(~R::RedType)
```

```
Set depths for DualGraph and Label to i->"n_i"
```

```
intrinsic SetOriginalDepths(~R::RedType)
```

```
Remove depths set by SetDepths, so that original ones are printed by Label and other functions
```

```
intrinsic SetMinimalDepths(~R::RedType)
```

```
Set depths to minimal ones in the family (MinimalLinkDepth = -1,0 or 1) for every edge
```

```
intrinsic GetDepths(R::RedType) -> SeqEnum
```

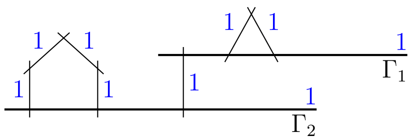
```
Return depths (string sequence) set by SetDepths or originals if not changed from defaults
```

**Example** (Setting variable depths for drawing families).

```
> R:=ReductionType("I3-(2)I5");
> Label(R: tex);
```

$\mathrm{I}_3 \frac{}{2} \mathrm{I}_5$

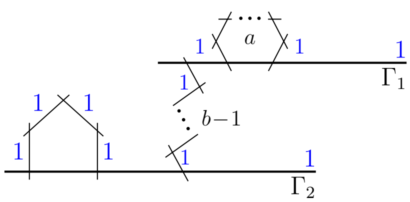
```
> TeX(DualGraph(R));
```




```
> SetDepths(~R,["a","b","5"]);    // Make two of the three chains variable depth
> Label(R: tex);
```

$\mathrm{I}a \frac{}{b} \mathrm{I}_5$

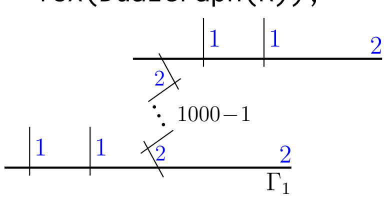
```
> TeX(DualGraph(R));
```




```
> SetOriginalDepths(~R);
> R;
I3-(2)I5
```

**Example** ($\mathrm{I}^*_{1000}$). This can also be used to draw types with large depths:

```
> R:=ReductionType("I1*");
> SetDepths(~R,["1000"]);
> TeX(DualGraph(R));
```




## 2.21 Namikawa-Ueno conversion in genus 2

```
intrinsic NamikawaUeno(R::RedType: pottype:="all", depths:="original",
  warnings:=true) -> MonStgElt, RngIntElt
```

```
returns Namikawa-Ueno reduction type pair nutype, page if unique,
  or false, [<pottype,guess,page>,...] if there are several depending
  on the potential semistable type (I,II,III,...,VII)
```

**Example**.

```
> R:=ReductionType("5^1,1,3");
```

```
> NamikawaUeno(R);
IX-2 157
> R:=ReductionType("[2]_1,D");        // several possible types
> NamikawaUeno(R);
false [ <"VII", "2I$_{1}$-1", 181>, <"IV", "II$^*_{1-1}$", 184> ]
> NamikawaUeno(R: pottype:="VII");    // specify Liu's potential semistable type
2I$_{1}$-1 181
```

# 3 General discrete valuation rings (dvr.m)

The file provides basic support for fields with a valuation and DVRs. Type RngDVR incorporates a base field $K$, residue field $k$, valuation $v\colon K \to \mathbb{Z}$, uniformizer $\pi$, reduction map $O_v \to k$ and its section (lifting map) $k \to O_v$.

```
type RngDVR
```

There is a variety of creation functions of the form DVR(field) and DVR(field, prime) to get DVRs from the rational, number fields, p-adics, function fields etc., as well as the function BaseDVR that gives an underlying DVR for an object over a field.

Basic invariants Field, Valuation, ResidueField, Characteristic, ResidueCharacteristic, Uniformizer can be accessed separately, or at once with an Eltseq function.

There is basic functionality for valuations of roots, Newton polygons and residual polynomials for a polynomial over a DVR.

## 3.1 Basic type functions: IsCoercible, in, Print

```
intrinsic Print(D::RngDVR, level::MonStgElt)
```

Print a RngDVR.

## 3.2 Creation functions

```
intrinsic DVR(K::FldRat, p::RngIntElt) -> RngDVR
```

Construct a DVR of type RngDVR(K,k,v,red,lift,pi) for K=Q, p=prime number

```
intrinsic DVR(Z::RngInt, p::RngIntElt) -> RngDVR
```

Construct a DVR of type RngDVR(K,k,v,red,lift,pi) for O=Z, p=prime number

```
intrinsic DVR(K::FldNum, p::RngOrdIdl) -> RngDVR
```

Construct a DVR of type RngDVR(K,k,v,red,lift,pi) for K=number field, p=prime ideal

```
intrinsic DVR(K::FldNum, p::PlcNumElt) -> RngDVR
```

Construct a DVR of type RngDVR(K,k,v,red,lift,pi) for K=number field, p=place

```
intrinsic DVR(O::RngOrd, p::RngOrdIdl) -> RngDVR
```

Construct a DVR of type RngDVR(K,k,v,red,lift,pi) for O=integer ring of a number field, p=prime ideal

```
intrinsic DVR(K::FldPad) -> RngDVR
```

Construct a DVR of type RngDVR(K,k,v,red,lift,pi) for K=p-adic field

```
intrinsic DVR(O::RngPad) -> RngDVR
```

Construct a DVR of type RngDVR(K,k,v,red,lift,pi) for O=integers in a p-adic field

```
intrinsic DVR(K::FldFunRat, p::FldFunRatUElt) -> RngDVR
```

Construct a DVR of type RngDVR(K,k,v,red,lift,pi) for a rational function field in one variable and element p

```
intrinsic Extend(D::RngDVR, n::RngIntElt) -> RngDVR
```

Make an unramified degree n extension of a DVR of type RngDVR. This assumes that the residue field k is finite, and the base field K is p-adic or a number field

**Example** (3-adic valuation on $\mathbb{Q}$).

```
> D:=DVR(Integers(),3);
> D;
DVR K=Rational Field p=3
> Field(D);
Rational Field
```

```
intrinsic BaseDVR(X::Any, P::Any) -> RngDVR
```

Guess an underlying DVR from an object X over some field K at a place p: the object could be a curve, polynomial, polynomial equation lhs=rhs, for example

```
intrinsic BaseDVR(X::Any) -> RngDVR
```

Guess an underlying DVR from an object X over some field K that has a canonical valuation: the object could be a curve, polynomial, polynomial equation lhs=rhs, for example

## 3.3 Basic invariants

```
intrinsic Eltseq(D::RngDVR) -> .,.,.,.,.,.
```

return 6 basic invariants K,k,v,red,lift,pi of a RngDVR

```
intrinsic Field(D::RngDVR) -> Fld
```

Base field of fractions K for a DVR

```
intrinsic Valuation(D::RngDVR) -> Map
```

Underlying discrete valuation v for a DVR

```
intrinsic ResidueField(D::RngDVR) -> Fld, Map, Map
```

Residue field k for a DVR, reduction map and the lifting map

```
intrinsic Characteristic(D::RngDVR) -> RngIntElt
```

Characteristic of the field of fractions K for a DVR

```
intrinsic ResidueCharacteristic(D::RngDVR) -> RngIntElt
```

Characteristic of the residue field k for a DVR

```
intrinsic Uniformizer(D::RngDVR) -> RngElt
```

Uniformizer pi for a DVR

```
intrinsic UniformizingElement(D::RngDVR) -> RngElt
```

Uniformizer pi for a DVR

**Example**.

```
> D:=DVR(Rationals(),2);       // 2-adic valuation on Q
> D;
DVR K=Rational Field p=2
> K:=Field(D);
> v:=Valuation(D);
> pi:=Uniformizer(D);
> k,red,lift:=ResidueField(D);
> pi^v(K!100);                 // Compute v_2(100)
4
> lift(k!100);                 // Lift 100 from GF(2) to Q
0
```

### 3.4 Newton polygons

```
intrinsic ValuationsOfRoots(f::RngUPolElt, D::RngDVR) -> SeqEnum
```

Valuations of roots of f defined over a RngDVR or its field of fractions

**Example.**

```
> Q:=Rationals();
> R<x>:=PolynomialRing(Q);
> ValuationsOfRoots(x^5+x,DVR(Q,2));
[ <Infinity, 1>, <0, 4> ]
```

```
intrinsic NewtonPolygon(f::RngUPolElt, D::RngDVR) -> NwtnPgon
```

Newton polygon of f with respect to a RngDVR

```
intrinsic ResidualPolynomials(f::RngUPolElt, D::RngDVR) -> SeqEnum, SeqEnum,
  SeqEnum, SeqEnum
```

Residual polynomials, Vertices of the (lower) Newton polygon N, slopes(N), lengths(N)

**Example.**

```
> Q:=Rationals();
> R<x>:=PolynomialRing(Q);
> D:=DVR(Q,2);
> f:=(x^2-2)*(x^3-2)*x;    // 3 segments
> N:=NewtonPolygon(f,D);
> N;
Newton Polygon of x^6 + 2*x^4 + 2*x^3 + 4*x over Rational Field at 2
> Slopes(N);
[ -1/2, -1/3 ]
> respoly,vert,slopes,lengths:=ResidualPolynomials(f,D);
> slopes;        // slopes of 3 segments
[* Infinity, 1/2, 1/3 *]
> lengths;       // number of roots in each
[ 1, 2, 3 ]
> respoly;       // reduced residual polynomials for each
[x,x + 1,x + 1]
> vert;          // vertices of the newton polygon
[ <0, 2>, <1, 2>, <3, 1>, <6, 0> ]
```

# 4 MacLane valuations over a DVR (maclane.m)

The file provides MacLane valuations on $K[x]$, where $K$ is a field with a discrete valuation. This is implemented as a type MacV. As in MacLane's paper, such a valuation $v$ is constructed inductively from the Gauss valuation $v_0$ on $K[x]$ with repeated assignments $v(g_i) = \lambda_i$ for some key polynomials $g_i$ and rationals $\lambda_i$.

See S. MacLane, *A construction for absolute values in polynomial rings*, Trans. Amer. Math. Soc. 40 (1936), no. 3, 363–395.

```
type MacV
```

## 4.1 Basic type functions

```
intrinsic Print(v::MacV, level::MonStgElt)
```

```
Print a MacLane valuation v
```

## 4.2 Creation functions

```
intrinsic MacLaneValuation(D::RngDVR, g::SeqEnum, lambda::SeqEnum) -> MacV
```

```
Create a MacLane valuation from its primary invariants: key polynomials g_i and rationals lambda_i,
so that v(g_i)=lambda_i
```

```
intrinsic GaussValuation(D::RngDVR) -> MacV
```

```
Gauss valuation on K[x] for K a field with a valuation specified with D of type RngDVR
```

```
intrinsic MacLaneValuation(D::RngDVR, t::SeqEnum[Tup]) -> MacV
```

```
Create a MacLane valuation from its primary invariants: key polynomials g_i and rationals lambda_i,
so that v(g_i)=lambda_i. The invariants a given as a sequence t of tuples [<g_i,lambda_i>]
```

**Example.**

```
> R<x>:=PolynomialRing(Q);
> D:=DVR(Q,3);
> v:=MacLaneValuation(D,[<x,1/2>,<x^2-3,1>]);
> v;
[x->1/2,x^2 - 3->1]
> TeX(v);
```

$v(x^2-3)\geq 1$

## 4.3 Basic invariants

```
intrinsic Length(v::MacV) -> RngIntElt
```

```
Length n of the MacLane valuation (number of the defining key polynomials g_1,...,g_n)
```

```
intrinsic Center(v::MacV) -> RngUPolElt
```

```
Center of the MacLane valuation (last g_n in the list g_1,...,g_n of key polynomials)
```

```
intrinsic Degree(v::MacV) -> RngIntElt
```

```
Degree of the MacLane valuation (degree of the last defining polynomial g_n=Center(v))
```

```
intrinsic Radius(v::MacV) -> FldRatElt
```

Radius of the MacLane valuation (last lambda in the list of key polynomial assignments v(g_i)=lambda_i)

```
intrinsic IsGauss(v::MacV) -> BoolElt
```

True if v is the Gauss valuation

```
intrinsic Extends(v2::MacV, v1::MacV) -> BoolElt
```

True if v2 extends v1 as a MacLane valuation

```
intrinsic Truncate(v::MacV, n::RngIntElt) -> MacV
```

Truncate a MacLane valuation to a smaller n <= Length(v)

```
intrinsic Truncate(v::MacV) -> MacV
```

Truncate a MacLane valuation to n-1 where n is Length(v)

```
intrinsic ChangeSlope(v::MacV, s::FldRatElt) -> MacV
```

Copy valuation with the last slope lambda_n changed to s

```
intrinsic RamificationDegree(v::MacV) -> RngIntElt
```

Ramification degree of a MacLane valuation over the Gauss valuation

```
intrinsic Monomial(v::MacV, s::FldRatElt) -> RngUPolElt
```

Canonical monomial in the key polynomials of v of a given rational valuation s, constructed inductively

```
intrinsic MacData(v::MacV) -> SeqEnum
```

List of tuples [<g_i,lambda_i>] that define a given MacLane valuation

**Example.**

```
> R<x>:=PolynomialRing(Q);
> D:=DVR(Q,3);
> v:=MacLaneValuation(D,[<x,1/2>,<x^2-3,1>]);
> RamificationDegree(v);
2
> Extends(v,GaussValuation(D));
true
> MacData(v);
[<x, 1/2>,<x^2 - 3, 1>]
> Monomial(v,3/2);
3*x
```

## 4.4 Newton polygons

```
intrinsic Expand(f::RngUPolElt, g::RngUPolElt) -> SeqEnum
```

Expand f in powers of g and return the sequence of coefficients, which are polynomials of degree < deg g

**Example.**

```
> R<x>:=PolynomialRing(Q);
> Expand((x^2-2)^3+(x^2-2)+x,x^2-2);
[x,1,0,1]
```

```
intrinsic Valuation(f::RngUPolElt, v::MacV: n:="Full") -> Tup
```

```
Valuation of a polynomial f with respect to a MacLane valuation v, computed inductively
using the expansion of f in key polynomials of v
```

```
intrinsic Valuation(f::FldFunRatUElt, v::MacV: n:="Full") -> Tup
```

```
Valuation of a rational function f with respect to a MacLane valuation v
```

```
intrinsic NewtonPolygon(f::RngUPolElt, v::MacV) -> SeqEnum
```

```
Compute the slopes of the Newton polygon of a polynomial f with respect to a MacLane valuation v
and relevant monomials (not reduced to the residue field). Returns a list of tuples
  [* <valuation, ramification degree, unreduced coefficients>, ... *]
valuation may be Infinity()
```

**Example.**

```
> R<x>:=PolynomialRing(Q);
> D:=DVR(Q,3);
> v:=MacLaneValuation(D,[<x,1/2>,<x^2-3,1>]);
> Valuation(x*(x^2-3),v);
3/2 2
> NewtonPolygon(x*(x^2-3),v);
[*
<Infinity, 1, [
x
]>
*]
```

```
intrinsic Distance(v,w::MacV) -> FldRatElt
```

```
Valuation distance between v and w. The valuations are viewed as defining discoids.
This function is symmetric, and d(v,v)=lambda_n/deg g_n
```

**Example.**

```
> R<x>:=PolynomialRing(Q);
> D:=DVR(Q,3);
> v2:=MacLaneValuation(D,[<x,1/2>,<x^2-3,1>]);
> v1:=Truncate(v2);
> v0:=GaussValuation(D);
> Distance(v0,v2);
0
> Distance(v1,v2);
1/2
> Distance(v2,v2);
1/2
```

## 4.5 Printing in TeX

```
intrinsic TeX(v::MacV) -> MonStgElt
```

```
Print a MacLane valuation in TeX in diskoid form, as v(Center)>=radius. This is used for cluster
names
```

# 5 Muselli-MacLane rational clusters (mclusters.m)

The file provides MacLane valuations on $K[x]$, where $K$ is a field with a discrete valuation. This is implemented as a type MacV. As in MacLane's paper, such a valuation $v$ is constructed inductively from the Gauss valuation $v_0$ on $K[x]$ with repeated assignments $v(g_i) = \lambda_i$ for some key polynomials $g_i$ and rationals $\lambda_i$.

See S. Muselli, *Regular models of hyperelliptic curves*, Indag. Math. (2023).

In the examples below we set

```
Q:=Rationals();
R<x>:=PolynomialRing(Q);
```

The package defines two types: rational MacLane-Muselli clusters (ClM) and the associated cluster pictures:

```
type ClM
```

```
type ClPicM
```

## 5.1 Basic type functions for clusters (ClM)

```
intrinsic Print(s::ClM, level::MonStgElt)
```

Print a MM cluster

## 5.2 Basic cluster invariants (ClM)

```
intrinsic Degree(s::ClM) -> RngIntElt
```

Degree of a MM cluster = degree of the defining valuation Valuation(s)

```
intrinsic Valuation(s::ClM) -> RngIntElt
```

Valuation that cuts out the cluster

```
intrinsic ClusterPicture(s::ClM) -> ClPicM
```

Cluster picture in which the cluster s lives

```
intrinsic Index(s::ClM) -> RngIntElt
```

Index of the cluster in the cluster picture

## 5.3 Equality and children

```
intrinsic 'eq'(s1::ClM, s2::ClM) -> BoolElt
```

Equality testing for MM clusters in the same cluster picture

```
intrinsic IsProperSubset(s::ClM, p::ClM) -> BoolElt
```

True if s is properly contained in p for MM clusters

```
intrinsic Children(s::ClM) -> SeqEnum
```

Proper children of a MM cluster

```
intrinsic ParentCluster(s::ClM) -> ClM
```

Parent of a MM cluster

```
intrinsic RootClusters(s::ClM) -> SeqEnum
```

List of root cluster valuations contained in s

## 5.4 Basic type functions for cluster pictures (ClPicM)

```
intrinsic Print(Sigma::ClPicM, level::MonStgElt)
```

Print a MM cluster picture

## 5.5 Basic invariants for cluster pictures (ClPicM)

```
intrinsic Genus(Sigma::ClPicM) -> RngIntElt
```

Genus of a MM cluster picture

```
intrinsic BaseField(Sigma::ClPicM) -> Fld
```

Original base field K for a MM cluster picture

```
intrinsic ResidueField(Sigma::ClPicM) -> Fld
```

Residue field k of the base field K over which a MM cluster picture is defined

```
intrinsic FieldOfDefinition(Sigma::ClPicM) -> Fld
```

Unramified extension F of the base field K of a MM cluster picture over which the centers are defined

```
intrinsic Clusters(Sigma::ClPicM) -> SeqEnum[ClM]
```

List of all clusters (of type ClM) that form the cluster picture

```
intrinsic RootClusters(~D::RngDVR, ~f::RngUPolElt, ~S)
```

Sequence of root (improper) clusters; they correspond to factors of f over the completion of K (unramified closure).
May need to extend D and base change f along the way, if unramfied extension is necessary.

**Example.**

```
> f:=x^6+9;                        // hyperelliptic curve C/Q3: y^2=f(x)
> Qp:=pAdicField(3,20);            //   here f(x)=x^6+9=(x^3+3i)(x^3-3i)
> D:=DVR(Qp);                      // and RootClusters extends Q3 to Q3(i)
> RootClusters(~D,~f,~S);
> D;
DVR K=Unramified extension defined by the polynomial x^2 + 2*x + 2 over 3-adic field mod
  3^20
> S;
[
[x->1/3,x^3+(-2*r1-2)*3->2],
[x->1/3,x^3+(-r1-1)*3->20]
]
```

## 5.6 Creation functions for cluster pictures

```
intrinsic ClusterPicture(f::RngUPolElt, D::RngDVR) -> ClPicM
```

```
MM cluster picture for a hyperelliptic curve y^2=f(x) over a RngDVR (res char<>2)
```

**Example.**

```
> D:=DVR(Q,3);
> TeX(ClusterPicture(x^3+3,D));                    // One cluster of size 3
```

| $\mathfrak{s}$ | $v$ | $\lvert\mathfrak{s}\rvert$ | $d_v$ | $b_v$ | $e_v$ | $\nu_v$ | $n_v$ | $m_v$ | $t_v$ | $p_v$ | $s_v$ | $\gamma_v$ | $p_v^0$ | $s_v^0$ | $\gamma_v^0$ | $u_v$ | $g$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\mathfrak{s}_1$ | $v(x)\geq 1/3$ | 3 | 1 | 3 | 3 | 1 | 1 | 6 | 3 | 1 | 1/6 | 1 | 2 | $-1/6$ | 2 | 2 | 0 |

```
> TeX(ClusterPicture((x^3+3)*(x-1)*(x-2),D));    // Two nested clusters
```

| $\mathfrak{s}$ | $v$ | $\lvert\mathfrak{s}\rvert$ | $d_v$ | $b_v$ | $e_v$ | $\nu_v$ | $n_v$ | $m_v$ | $t_v$ | $p_v$ | $s_v$ | $\gamma_v$ | $p_v^0$ | $s_v^0$ | $\gamma_v^0$ | $u_v$ | $g$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\mathfrak{s}_1$ | $v(x)\geq 1/3$ | 3 | 1 | 3 | 3 | 1 | 1 | 6 | 3 | 1 | 1/6 | 1 | 2 | $-1/6$ | 2 | 2 | 0 |
| $\mathfrak{s}_2$ | $v(x)\geq 0$ | 5 | 1 | 1 | 1 | 0 | 2 | 1 | 5 | 1 | 0 | 1 | 2 | 0 | 1 | 3 | 1 |

```
> TeX(ClusterPicture((x^2-3)^3+x^7,D));          // Non-rational cluster
```

| $\mathfrak{s}$ | $v$ | $\lvert\mathfrak{s}\rvert$ | $d_v$ | $b_v$ | $e_v$ | $\nu_v$ | $n_v$ | $m_v$ | $t_v$ | $p_v$ | $s_v$ | $\gamma_v$ | $p_v^0$ | $s_v^0$ | $\gamma_v^0$ | $u_v$ | $g$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\mathfrak{s}_1$ | $v(x^2-3)\geq 7/6$ | 6 | 2 | 3 | 6 | 7/2 | 1 | 12 | 3 | 1 | 7/12 | 1 | 2 | $-7/12$ | 2 | 2 | 0 |
| $\mathfrak{s}_2$ | $v(x)\geq 1/2$ | 6 | 1 | 2 | 2 | 3 | 2 | 2 | 6 | 2 | 1/2 | 1 | 2 | $-1$ | 2 | 2 | 0 |
| $\mathfrak{s}_3$ | $v(x)\geq 0$ | 7 | 1 | 1 | 1 | 0 | 2 | 1 | 7 | 1 | 0 | 1 | 2 | 0 | 1 | 1 | 0 |

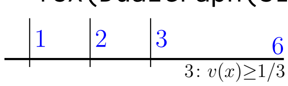
## 5.7 Dual graph from a cluster picture and associated model (Muselli's theorem)

```
intrinsic DualGraph(Sigma::ClPicM: check:=true, contract:=true,
  texsettings:="default") -> GrphDual
```

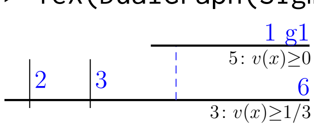
```
Dual graph of a cluster picture (Muselli's Theorem);
  check: test multiplicities;
  contract: contract components to get minimal r.n.c. model;
```

**Example.**

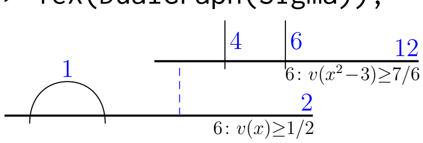
```
> D:=DVR(Q,3);
> Sigma:=ClusterPicture(x^3+3,D);                    // One cluster of size 3
> TeX(DualGraph(Sigma));
```




```
> Sigma:=ClusterPicture((x^3+3)*(x-1)*(x-2),D);     // Two nested clusters
> TeX(DualGraph(Sigma));
```




```
> Sigma:=ClusterPicture((x^2-3)^3+x^7,D);           // Non-rational cluster
> TeX(DualGraph(Sigma));
```




```
intrinsic TeX(Sigma::ClPicM) -> MonStgElt
```

```
list of clusters as an TeX array
```

```
intrinsic MuselliModel(f::RngUPolElt, D::RngDVR: Style:=[]) -> CrvModel
```

```
Maclane-Muselli model of a hyperelliptic curve
```

**Example.**

```
> D:=DVR(Q,3);
> M:=MuselliModel(x^3+3,D);                  // One cluster of size 3
> ReductionType(M);
II
> M:=MuselliModel((x^3+3)*(x-1)*(x-2),D);    // Two nested clusters
> ReductionType(M);
1g1-II
> M:=MuselliModel((x^2-3)^3+x^7,D);          // Non-rational cluster
> ReductionType(M);
D_0-[2]II
```

# 6 Model wrapping functions (model.m)

```
type CrvModel
```

## 6.1 Basic type functions

```
intrinsic Print(C::CrvModel, level::MonStgElt)
```

Print a curve model

## 6.2 Invariants

```
intrinsic DualGraph(C::CrvModel) -> GrphDual
```

Dual graph of a curve model

```
intrinsic ReductionType(C::CrvModel) -> RedType
```

Reduction graph of a curve model, or false if singular

```
intrinsic IsSingular(C::CrvModel) -> RedType
```

true if failed to find a regular model (neither hyperelliptic nor Delta_v-regular)

```
intrinsic Genus(C::CrvModel) -> RngIntElt
```

Genus of the generic fibre of a model

```
intrinsic IsGood(C::CrvModel) -> BoolElt
```

true if comes from a curve wih good reduction

```
intrinsic IsSemistable(C::CrvModel) -> BoolElt
```

true if comes from a curve with semistable reduction

```
intrinsic IsSemistableTotallyToric(C::CrvModel) -> BoolElt
```

true if comes from a curve with semistable totally toric reduction

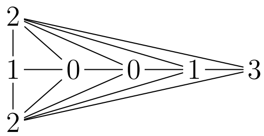
```
intrinsic IsSemistableTotallyAbelian(C::CrvModel) -> BoolElt
```

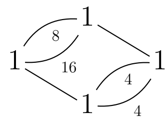
true if comes from a curve with semistable totally abelian reduction

**Example** (Totally toric hyperelliptic curves in any residue characteristic (IsSemistableTotallyToric):).

```
> U<p>:=RationalFunctionField(GF(2));        // work over F_2(t) at t=0
```

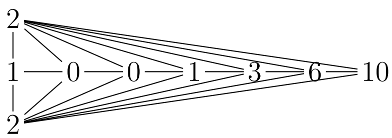
```
> R<x,y>:=PolynomialRing(U,2);
> style:=[["ContractFaces","false"],["FaceNames","false"]];    // less clutter
> f:=p^2*y^2+p^2+y*(x+p)*(x+1)*(p*x+1)*(p^2*x+1); // break a Newton polygon into length 1
> M:=Model(f,p: Style:=style); // pieces to get totally toric reduction
> DeltaTeX(M), TeX(ReductionType(M)), "Genus", Genus(M);
```

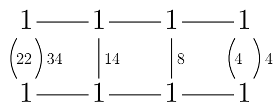



Genus 3

```
> f:=p^2*y^2+p^2+y*(x+p)*(x+1)*(p*x+1)*(p^2*x+1)*(p^3*x+1)*(p^4*x+1); // same in genus 5
> M:=Model(f,p: Style:=style);
> DeltaTeX(M), TeX(ReductionType(M)), "Genus", Genus(M);
```




Genus 5

```
> IsGood(M);                        // no, not 1g5
false
> IsSemistable(M);                  // yes, all components have multiplicity 1
true
> IsSemistableTotallyToric(M);   // yes, semistable with no positive genus components
true
```

```
intrinsic TeX(M::CrvModel: Charts:=false, Equation:=false, Delta:=false,
  RedType:=false, texsettings:=[]) -> MonStgElt
```

Full TeX description of a model of a curve

## 6.3 Model and ReductionType wrappers

```
intrinsic Model(X::Any, P::Any: model:="default", Style:=[]) -> CrvModel
```

Minimal regular with normal crossings model of a curve X at P.
Parameter model controls the default algorithm, and can be "default",
"delta" (use Delta-regular machinery) or "clusters" (use Muselli-Maclane clusters
for hyperelliptic curves in odd residue characteristic)
A univariate polynomial is interpreted as defining a hyperelliptic curve

```
intrinsic Model(X::Any: model:="default", Style:=[]) -> CrvModel
```

Minimal regular with normal crossings model of a curve X at P.
Parameter model controls the default algorithm, and can be "default",
"delta" (use Delta-regular machinery) or "clusters" (use Muselli-Maclane clusters
for hyperelliptic curves in odd residue characteristic)
A univariate polynomial is interpreted as defining a hyperelliptic curve

```
intrinsic ReductionType(X::Any, P::Any) -> RedType
```

Reduction type of X at P

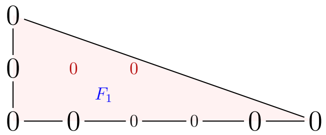
```
intrinsic ReductionType(X::Any) -> RedType
```

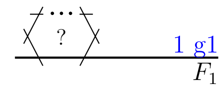
Reduction type of X at the default valuation of its base field

**Example** (See [Do1, Table 1 (v),(viii),(ix)]).

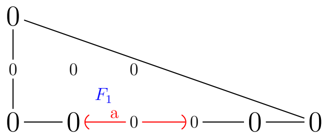
```
> R<x,y>:=PolynomialRing(Q,2);
> eqn:=(y-1)^2=(x-1)*(x-2)*(x-3)^2*(x-4)+5^4;   // Example (v)
> M:=Model(eqn,5: model:="delta");
```

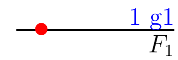
```
> TeX(M: Delta);
```





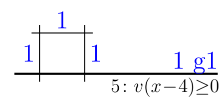

```
> eqn:=y^2=(x-1)*(x-2)*(x-3)^2*(x-4)+5^4;        // Example (viii)
> M:=Model(eqn,5: model:="delta");
> TeX(M: Delta);
```



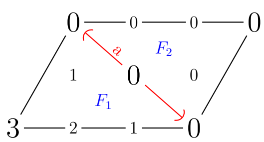


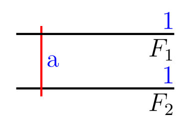

```
> M:=Model(eqn,5);        // Actual model for those two, computed with Muselli
> Label(ReductionType(M): tex),TeX(M);
```

$\mathrm{I}_{4,\mathrm{g1}}$



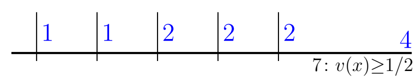

```
> eqn:=x^4*y^2=x*(y-x)^2+5^3;                        // Example (ix)
> M:=Model(eqn,5: model:="delta");
> TeX(M: Delta);
```






```
> M:=Model(eqn,5);        // Actual model, computed with Muselli
> Label(ReductionType(M): tex),TeX(M);
```

$4^{1,1,2,2,2}$




# 7 $\Delta_v$-regular models (delta.m)

## 7.1 Main function

```
intrinsic DeltaRegularModel(f::RngMPolElt, D::RngDVR: Style:=[]) -> CrvModel
```

Delta-regular model for a curve C given by f=0 (main function)

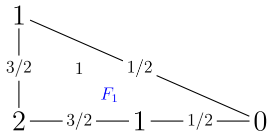
## 7.2 TeX for $\Delta_v$

```
intrinsic DeltaTeX(C::CrvModel: xscale:=0.8, yscale:=0.7) -> MonStgElt
```

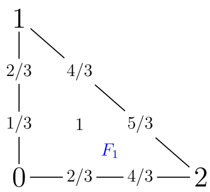
Newton polytope and v-faces in TikZ

**Example.**

```
> R<x,y>:=PolynomialRing(Q,2); // 2 exceptional shapes that give deficient genus 1 curves
> p:=37;
> f:=p*y^2+x^4+p*x^2+p^2;             // 2g1
> C:=DeltaRegularModel(f,DVR(Q,p));
> DeltaTeX(C);
```




```
> f:=p*y^3 + p^2*x^3 + 1;            // 3g1
> C:=DeltaRegularModel(f,DVR(Q,p));
> DeltaTeX(C);
```




```
intrinsic EquationTeX(C::CrvModel) -> MonStgElt
```

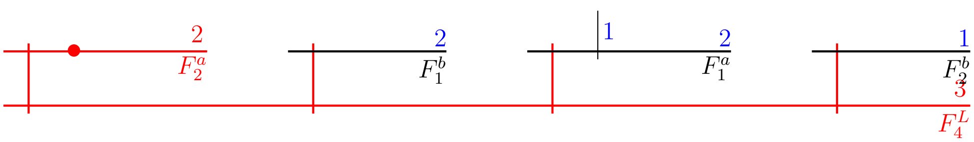
Original defining equation in TeX

**Example** (Taken from [Do1, Ex 3.18]).

```
> R<x,y>:=PolynomialRing(Q,2);  // Example from Poonen-Silverberg-Stoll paper at p=2
> f:=-2*x^3*y-2*x^3+6*x^2*y+3*x*y^3-9*x*y^2+3*x*y-x+3*y^3-y;
> C:=Model(f,2);
> EquationTeX(C);
```

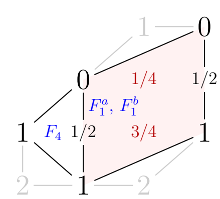
$(3x+3)y^3 - 9xy^2 + (-2x^3 + 6x^2 + 3x - 1)y - 2x^3 - x$

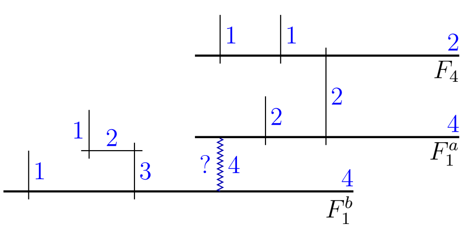
```
> TeX(C);
```




```
> f2:=Evaluate(f,[x+1,x*y+1]);  // Better model
> C2:=Model(f2,2);
> TeX(C2: Equation, Delta);     // All in one call
```

$f = (3x^4+6x^3)y^3+9x^2y^2+(-2x^4+6x)y-4x^3-6x^2-4x$ at $p=2$




## 7.3 Charts and transformation matrices

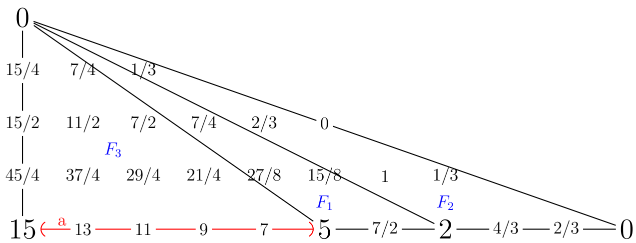
```
intrinsic ChartsTeX(C::CrvModel) -> MonStgElt
```

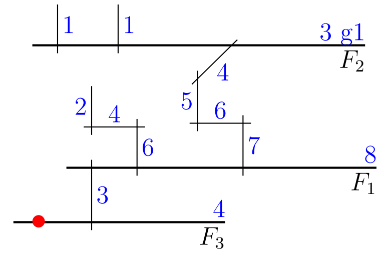
Charts for components in TeX for a curve model

**Example** (TeX, DeltaTeX, ChartsTeX for $\Delta_v$-regular model).

```
> R<x,y>:=PolynomialRing(Q,2);
> p:=5;
```

```
> f:=x^10+y^4+p^2*x^7+p^5*x^5+p^15;
> M:=Model(f,p);                // ChartsTeX also shows the root of
> DeltaTeX(M),TeX(M);           // the singular point on the leftmost edge
```

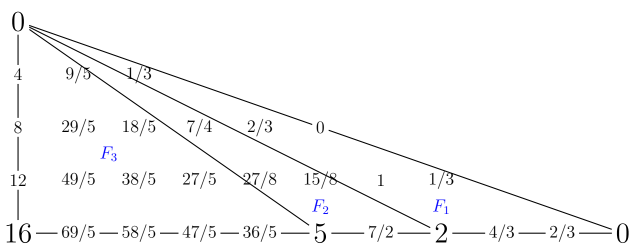


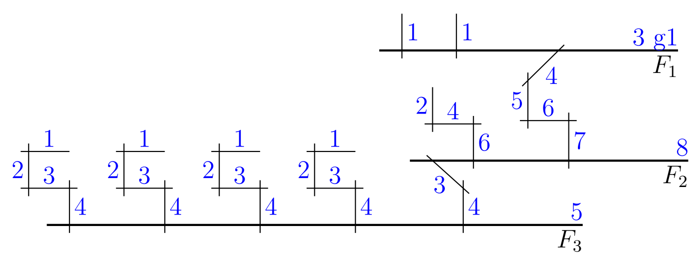




```
> ChartsTeX(M);                 // alternatively TeX(M: Delta, Charts) does the same
```

| | | | |
|---|---|---|---|
| $F_1$ | $x = XY^{10}Z^{12}$ | $X = x^{-7}y^4p^{-2}$ | $Y + X^3 + X^2 = 0$ |
| | $y = X^2Y^{21}Z^{25}$ | $Y = x^{-16}y^8p^{-1}$ | $Z^8 = 0$ |
| | $p = Y^7Z^8$ | $Z = x^{14}y^{-7}p$ | |
| $F_2$ | $x = X^{-1}Z^2$ | $X = x^{-5}y^2$ | $X^3Y^2 + X^2 + 1 = 0$ |
| | $y = X^{-2}Z^5$ | $Y = x^6y^{-3}p$ | $Z^3 = 0$ |
| | $p = YZ^3$ | $Z = x^{-2}y$ | |
| $F_3$ | $x = X^6YZ^8$ | $X = y^{-4}p^{15}$ | $XY^5 + X + 1 = 0$ |
| | $y = X^{11}Z^{15}$ | $Y = xp^{-2}$ | $Z^4 = 0$ |
| | $p = X^3Z^4$ | $Z = y^3p^{-11}$ | |
| $a$ | $L = 1$ | $r = [4]^5$ | |

```
> f2:=Evaluate(f,[x+4*p^2,y]);      // Shift it along the singular edge
> M2:=Model(f2,p);                  // to try to resolve singularity
> texsettings:=[["dualgraph.root","3"],["dualgraph.scale","0.9"]]; // put F3 at the bottom
> TeX(M2: Delta, texsettings:=texsettings);
```




# 8 Drawing planar graphs (planar.m)

## 8.1 Main functions

```
intrinsic StandardGraphCoordinates(G::GrphUnd: attempts:=10) -> SeqEnum,
  SeqEnum, SeqEnum
```

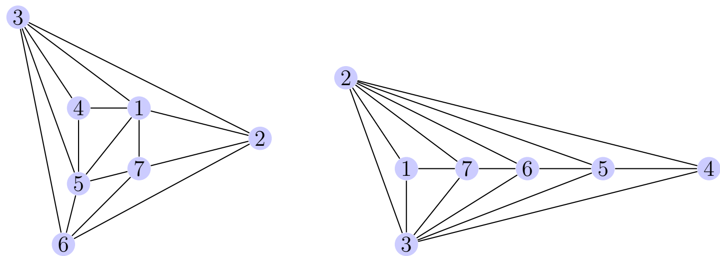
Tries to embed a graph in the plane with the least number of edge self-intersections. For planar graphs on at most 7 vertices and a few others, use a built-in database. Returns x=[x1,x2,...], y=[y1,y1,...] - x,y-coordinates for every vertex in VertexSet(G), and suggested vertex labels

```
intrinsic TeXGraph(G::GrphUnd: x:="default", y:="default", labels:="default",
  scale:=0.8, xscale:=1, yscale:=1, vertexlabel:="default",
  edgelabel:="default", vertexnodestyle:="default", edgenodestyle:="default",
  edgestyle:="default") -> MonStgElt
```

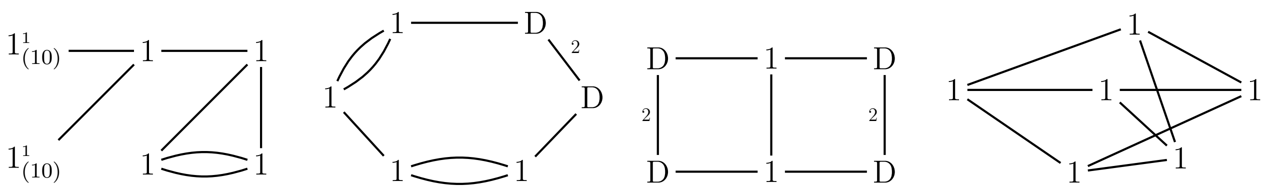
Simple function to draw a small planar graph in tikz. Labels can be a sequence of strings (or "none", or "default" -> 1,2,3,... unless G is labelled) to draw vertices. This function is not used in the core of the package, and is just here to illustrate StandardGraphCoordinates used for drawing shapes and reduction types

**Example** (Drawing planar graphs).

```
> D:=PlanarGraphDatabase(7);          // assuming database is installed
> G1:=Graph(D,#D-2);                  // draw three most complex planar graphs
> G2:=Graph(D,#D-1);                  // on 7 vertices
> G3:=Graph(D,#D);
> TeXGraph(G1),TeXGraph(G2),TeXGraph(G3);
```




```
> shapes:=[S[1]: S in Shapes(4) | #S[1] eq 6][[4,20,28,30]];
> &cat [TeX(S): S in shapes];         // This is used when drawing shapes
```



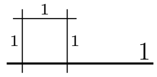
```
> IsPlanar(Graph(shapes[4]));
false
```

# 9 Special fibres or mrnc models (dualgraph.m)

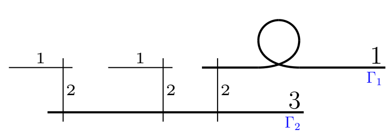
```
type GrphDualVert
```

```
type GrphDual
```

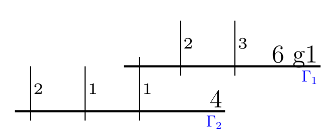
A dual graph is a combinatorial representation of the special fibre of a model with normal crossings. It is a multigraph whose vertices are components $\Gamma_i$, and an edge corresponds to an intersection point of two components. Every component $\Gamma$ has **multiplicity** $m = m_\Gamma$ and geometric **genus** $g = g_\Gamma$. Here are three examples of dual graphs, and their associated reduction types; we always indicate the multiplicity of a component (as an integer), and only indicate the genus when it is positive (as g followed by an integer).



Type $\mathrm{I}_4$ (genus 1)



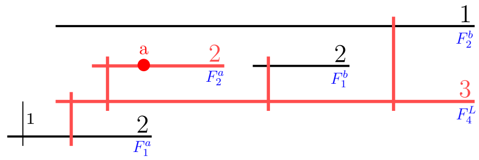

Type $\mathrm{I}_1-\mathrm{IV}*$ (genus 2)

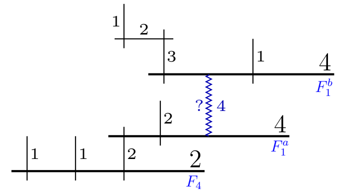



Type $\mathrm{II}_{\mathrm{g}1}-\mathrm{III}$ (genus 8).

A component is **principal** if it meets the rest of the special fibre in at least 3 points (with loops on a component counting twice), or has $g > 0$. The first example has no principal components, and the other two have two each, $\Gamma_1$ and $\Gamma_2$.

This module dualgraph.m provides a data type (**GrphDual**) for representing dual graphs and their manupulation and invariants. Sometimes, when working with models, it is desirable to store and draw incomplete or singular dual graphs, such as these (see [Do1, Ex 3.18]):





Such dual graphs are supported as well.

```
type GrphDual:
  V,       // associative array: name -> vertex of type GrphDualVert
           //   one for each component, not necessarily principal
  G,       // underlying abstract multigraph of all components
           // vertex labels come from v`name
  P,       // principal components (sequence of names)
  C,       // chains of P1s - one for includetexname=false one for =true
           // [<"1","1",[2,3,2]>,<"1","2",[]>,...] initialised by ChainsOfP1s
  specialchains,  // singular and other special chains, and those of variable length
                  //  in the format <c1, c2, singular, linestyle, endlinestyle,
                  //     labelstyle, margins, P1length, multiplicities>
  texsettings;    // [["name","setting"],...] settings overwriting defaults in settings.m
```

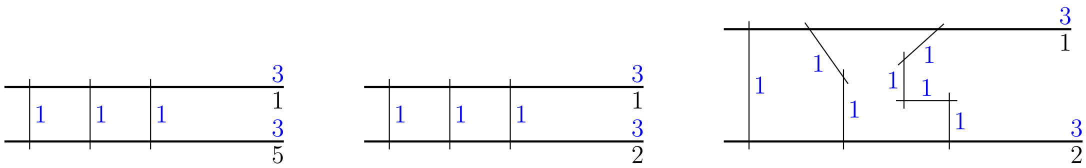
## 9.1 Default construction

```
intrinsic DualGraph(m::SeqEnum[RngIntElt], g::SeqEnum[RngIntElt],
  E::SeqEnum[SeqEnum]: comptexnames:="%o", texsettings:=[]) -> GrphDual
```

```
Construct a dual graph from a sequence of n multiplicities of components, sequence of n genera of
components and sequences of edges. Each edge is either
  [i,j]          - intersection point between component #i and component #j (1<=i,j<=n)
  [i,0,d1,d2,...] - open chain from component #i (1<=i<=n)
  [i,j,d1,d2,...] - link chain from component #i to component #j (1<=i,j<=n)
This can be used to reconstruct a dual graph printed with Sprint(G,"Magma").
comptexnames determines the names of principal components in TeX (v`texname), and each component for
which texname<>""
is considered principal when drawing dual graphs. The options are
  comptexnames::MonStgElt - string such as "c%o" which assigns names for principal components (and
  only those)
    among those specified by m_i, g_i
  comptexnames::SeqEnum      - sequence of strings for all components specified by m_i, g_i
  comptexnames::UserFunction - function i->string that defines such a sequence.
```

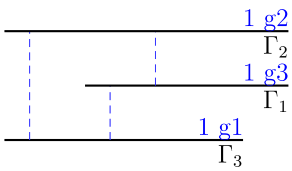
**Example** (Constructing a dual graph).

```
> m := [3,1,1,1,3];         // All components and intersection points
> g := [0,0,0,0,0];
> E := [[1,2],[1,3],[1,4],[2,5],[3,5],[4,5]];
> G1:= DualGraph(m,g,E);
> m := [3,3];               // Principal components and chains (same graph)
> g := [0,0];
> E := [[1,2,1],[1,2,1],[1,2,1]];
> G2:= DualGraph(m,g,E);
> m := [3,3];
> g := [0,0];               // Principal components, different chains
> E := [[1,2,1],[1,2,1,1],[1,2,1,1,1,1]];
> G3:= DualGraph(m,g,E);
> TeX(G1), TeX(G2), TeX(G3);
```




**Example** (Printing dual graph as a string and reconstructing it).

```
> R:=ReductionType("1g1-1g2-1g3-c1");
> G:=DualGraph(R);                       // Triangular dual graph on 3 vertices and 3 edges
> TeX(G);
```




```
> Sprint(G,"Magma");                      // Printed as DualGraph(m,g,E)
DualGraph([1,1,1], [3,2,1], [[1,2],[1,3],[2,3]])
> G2:=eval Sprint(G,"Magma");             // and reconstructed back
> Sprint(G2,"Magma");
DualGraph([1,1,1], [3,2,1], [[1,2],[1,3],[2,3]])
```

## 9.2 Step by step construction

```
intrinsic DualGraph(: texsettings:=[]) -> GrphDual
```

Create an empty dual graph. Assumes components and chains will be added later.

```
intrinsic AddComponent(~G::GrphDual, c::MonStgElt, genus::RngIntElt,
  mult::RngIntElt: texname:=c, singular:=false)
```

Add a vertex to a dual graph corresponding to a component with a given name c, genus, multiplicity and optional texname.
If singular:=true, the whole graph is marked as singular (no associated reduction type) and the component is drawn in red.

```
intrinsic AddComponent(~G::GrphDual, ~c::MonStgElt, genus::RngIntElt,
  mult::RngIntElt: texname:=c, singular:=false)
```

Add a vertex to a dual graph corresponding to a component, with given genus and multiplicity.
If singular:=true, the whole graph is marked as singular (no associated reduction type) and the component is drawn in red.
Sets and returns component name in c if c="".

```
intrinsic AddComponent(~G::GrphDual, genus::RngIntElt, mult::RngIntElt)
```

Add a no-named vertex to a dual graph corresponding to a component with a genus and multiplicity

```
intrinsic AddChain(~G::GrphDual, c1::MonStgElt, c2::MonStgElt,
  mults::SeqEnum[RngIntElt])
```

Add a chain of P1s with multiplicities (possibly empty) between components c1 and c2

```
intrinsic AddMinimalLinkChain(~G::GrphDual, c1::MonStgElt, c2::MonStgElt,
  d::RngIntElt, a::FldRatElt, b::FldRatElt: family:=false)
```

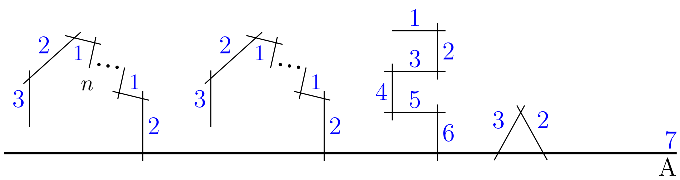
Add a chain of P1s between c1 and c2 (open-ended if c2="") with multiplicities d times denominators of minimal continued fractions from a to b.
family:=true or family:="$n$" shows multiplicity d components as variable chains of a given length (none or $n$).

```
intrinsic AddMinimalOpenChain(~G::GrphDual, c::MonStgElt, d::RngIntElt,
  a::FldRatElt)
```

```
Add an open-ended chain of P1s from c with multiplicities d times denominator
  of minimal continued fractions from a to an integer Floor(d*a-1)/d.
```

**Example** (Hand-crafted dual graphs with variable length chains).

```
> G:=DualGraph();
> AddComponent(~G,"A",0,7);
> AddMinimalOpenChain(~G,"A",1,6/7);          // open
> AddMinimalLinkChain(~G,"A","A",1,5/7,3/7);   // link
> assert IsConnected(G) and not IsSingular(G);
> AddMinimalLinkChain(~G,"A","A",1,5/7,3/7: family:="$n$");
> AddMinimalLinkChain(~G,"A","A",1,5/7,3/7: family);
> TeX(G);
```




```
intrinsic AddSingularPoint(~G::GrphDual, c::MonStgElt, point::MonStgElt)
```

```
Add a standard singular point point = "redbullet" or "bluenode" on a component c of a dual graph
```

```
intrinsic AddSingularChain(~G::GrphDual, c1::MonStgElt, c2::MonStgElt:
  singular:=true, mults:=[""], linestyle:="default", endlinestyle:="default",
  labelstyle:="default", linemargins:="default", P1linelength:="default")
```

```
Add a singular chain in given tikz style with multiplicities mults (sequence of integers or strings)
    between c1 and c2; use c2="0" for an open chain; default style="red"
```

```
intrinsic AddVariableChain(~G::GrphDual, c1::MonStgElt, c2::MonStgElt,
  mults::List)
```

```
Add a chain where some parts have variable length, e.g. [* 1,2,<3,"$n$">,<4,"$m$">,3,2,1 *]
```

**Example** (Hand-crafted dual graphs with all possible decorations).

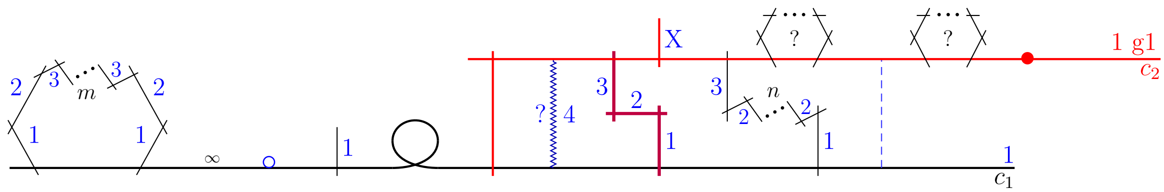
```
> G:=DualGraph();
> AddComponent(~G,"1",0,1: texname:="$c_1$"); // name,genus,multiplicity [+ component
  name]
> AddComponent(~G,"2",1,1: texname:="$c_2$", singular);  // singular component (red)
> AddSingularPoint(~G,"2","bluenode");          // singular points (standard)
> AddSingularPoint(~G,"2","bluenode");          //   node of unknown length
> AddSingularPoint(~G,"2","redbullet");         //   red bullet singular point
> AddSpecialPoint(~G,"1","blue,inner sep=0pt,above=-1pt","$\\circ$");     // singular pt
> AddSpecialPoint(~G,"1","above,scale=0.5","$\\infty$": singular:=false); // non-sing pt
> AddChain(~G,"1","1",[]);                      // self-chain of length 0 (node)
> AddChain(~G,"1","2",[]);                      // chain of length 0 (dashed)
> AddChain(~G,"1","0",[1]);                     // open chain
> AddSingularChain(~G,"1","2");                 // singular chain (red line)
> AddSingularChain(~G,"2","0": mults:=["X"]);  // singular open chain
```

Add a “zigzag” style chain of unknown length and multiplicity 4

```
> AddSingularChain(~G,"1","2": mults:=["$\\hspace{-11pt}?\\ \\ 4$"],
  linestyle:="snake=zigzag,segment length=2,segment amplitude=1,blue!70!black");
```

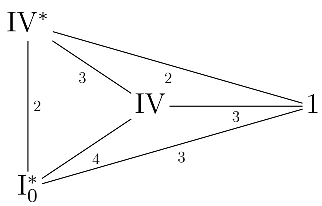
Add a custom purple chain with multiplicties 1,2,3

```
> AddSingularChain(~G,"1","2": mults:=[1,2,3], linestyle:="shorten <=-3pt,shorten >=-3pt,
  very thick, purple");
> AddVariableChain(~G,"1","2",[* 1,<2,"$n$">,3*]);          // variable length
> AddVariableChain(~G,"1","1",[* 1,2,<3,"$m$">,2,1 *]); // self-chain of variable length
> TeX(G);
```

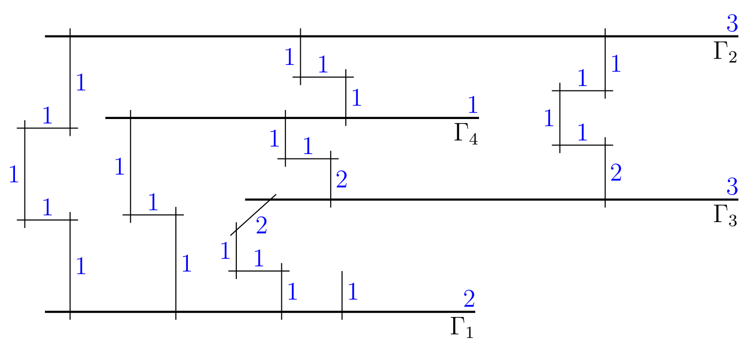



**Example** (K4). TeX for dual graphs is limited to small planar graphs, and K4 is more or less the most complex one that it can draw. Here is a reduction type like that:

```
> R:=ReductionType("1-(3)IV-(3)IV*-(2)I0*-(3)c1-(2)c3&c2-(4)c4");
> TeX(R: scale:=1.5);
```




```
> TeX(DualGraph(R));
```



## 9.3 Arithmetic invariants of dual graphs

```
intrinsic IsSingular(G::GrphDual) -> BoolElt
```

```
Check if G has any singular components or points, or special chains. If yes, no self-intersections
will be checked components contracted (so MakeMRNC does nothing).
```

```
intrinsic IsConnected(G::GrphDual) -> BoolElt
```

```
True if underlying graph is connected.
```

```
intrinsic HasIntegralSelfIntersections(G::GrphDual) -> BoolElt
```

```
Are all component self-intersections integers
```

```
intrinsic AbelianDimension(G::GrphDual) -> RngIntElt
```

```
Sum of genera of components)
```

```
intrinsic ToricDimension(G::GrphDual) -> RngIntElt
```

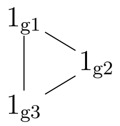
```
Number of loops in the dual graph
```

```
intrinsic IntersectionMatrix(G::GrphDual) -> AlgMatElt
```

```
Intersection matrix for a dual graph, whose entries are pairwise intersection numbers of the
components.
```

**Example**. Here is the dual graph of the reduction type $1_{g3}-1_{g2}-1_{g1}-c_1$, consisting of three components genus 1,2,3, all of multiplicity 1, connected in a triangle.

```
> G := DualGraph([1,1,1],[1,2,3],[[1,2],[2,3],[3,1]]);
> assert not IsSingular(G);                    // Has no singular points or components
> assert IsConnected(G);                       // Check the dual graph is connected
> assert HasIntegralSelfIntersections(G);      //   and every component c has c.c in Z
> AbelianDimension(G);                         // genera 1+2+3 => 6
6
> ToricDimension(G);                           // 1 loop       => 1
1
> TeX(ReductionType(G));
```




```
> IntersectionMatrix(G);                       // Intersection(G,v,w) for v,w components
[-2  1  1]
[ 1 -2  1]
[ 1  1 -2]
```

## 9.4 Contracting components to get a mrnc model

```
intrinsic AddEdge(~G::GrphDual, c1::MonStgElt, c2::MonStgElt)
```

```
Add an edge between two components in a dual graph
```

```
intrinsic ContractComponent(~G::GrphDual, c::MonStgElt: checks:=true)
```

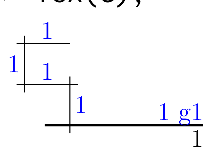
```
Contract a component in the dual graph, assuming it meets one or two components, and has genus 0
```

```
intrinsic MakeMRNC(~G::GrphDual)
```

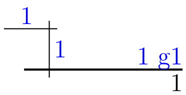
```
Contract all genus 0 components of self-intersection -1, resulting in a minimal model with normal
crossings
```

**Example** (Contracting components).

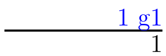
```
> G := DualGraph([1,1],[1,0],[[1,2,1,1,1]]);    // Not a minimal rnc model
> TeX(G);
```




```
> Components(G);
[ 1, 2, c3, c4, c5 ]
> ContractComponent(~G,"2");                      // Remove the last component
> ContractComponent(~G,"c5");                     //   and then the one before that
> TeX(G);
```




```
> Components(G);
[ 1, c3, c4 ]
> MakeMRNC(~G);                                    // Contract the rest of the chain
> TeX(G);
```




## 9.5 Invariants of individual vertices (components)

```
intrinsic Components(G: GrphDual) -> SeqEnum[MonStgElt]
```

Names of all components of G, e.g. "1","2","c3","c4","c5"

```
intrinsic HasComponent(G::GrphDual, c::MonStgElt) -> BoolElt, MonStgElt
```

True if the dual graph has a component with a given c, in which case also return its index

```
intrinsic AddAlias(~G::GrphDual, c::MonStgElt, alias:MonStgElt)
```

Add alias to a component c, e.g "2+" for "2"

```
intrinsic Genus(G::GrphDual, c::MonStgElt) -> RngIntElt
```

Genus of a component in a dual graph

```
intrinsic Multiplicity(G::GrphDual, c::MonStgElt) -> RngIntElt
```

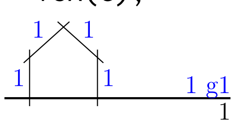
Multiplicity of a component in a dual graph

```
intrinsic Intersection(G::GrphDual, c1::MonStgElt, c2::MonStgElt) -> FldRatElt
```

Compute intersection of two components in a dual graph, or self-intersection if c1=c2

**Example** (Cycle of 5 components).

```
> G:=DualGraph([1],[1],[[1,1,1,1,1,1]]);
> TeX(G);
```




```
> C:=Components(G); C;
[ 1, c2, c3, c4, c5 ]
> assert HasComponent(G,"1");
> AddAlias(~G,"1","main");
> assert HasComponent(G,"main");
> Multiplicity(G,"main");
1
> Genus(G,"main");
1
> Matrix([[Intersection(G,v,w): v in C]: w in C]);
[-2  1  0  0  1]
[ 1 -2  1  0  0]
[ 0  1 -2  1  0]
[ 0  0  1 -2  1]
```

```
[ 1  0  0  1 -2]
```

## 9.6 Principal components and chains of $\mathbb{P}^1$s

```
intrinsic Neighbours(G::GrphDual, c::MonStgElt) -> SeqEnum[MonStgElt]
```

```
Neighbour vertices of a component, one for every edge (and two for every loop)
```

```
intrinsic PrincipalComponents(G::GrphDual) -> SeqEnum
```

```
Return a list of indices of principal components.
A vertex is a principal component if either its genus is greater than 0
or it has 3 or more incident edges (counting loops twice).
In the exceptional case [d]I_n one component is declared principal.
```

```
intrinsic ChainsOfP1s(G::GrphDual) -> SeqEnum
```

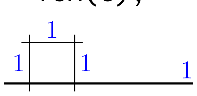
```
Sequence of tuples [<v0,v1,[chain multiplicities]>]
    for chains of P1s between principal components
```

**Example** (Cycle of 5 components). We take the same cycle graph as above, on 5 components.

```
> G:=DualGraph([1],[1],[[1,1,1,1,1,1]]);
> Components(G);        // Names of all components
[ 1, c2, c3, c4, c5 ]
> Neighbours(G,"c2");   // Neighbouring components, one for every edge out of c2
[ c3, 1 ]
> ChainsOfP1s(G);       // Chains of P1s between principal components
[
<"1", "1", [ 1, 1, 1, 1 ]>
]
```

**Example** (Exceptional case $[d]\mathrm{I}_n$). In the exceptional case $\mathrm{I}_n$ (genus 1) and its multiples, one (arbitrary) component is declared principal, so that such a reduction type falls into the general framework.

```
> G:=DualGraph(ReductionType("I4"));
> TeX(G);
```




```
> Components(G);
[ 1, 2, 3, 4 ]
> PrincipalComponents(G);    // One component pretends to be principal
[ 3 ]
> ChainsOfP1s(G);            //   and has a chain to itself
[
<"3", "3", [ 1, 1, 1 ]>
]
```

# 10 Reduction types in python (redtype.py)

The library redtype.py implements the combinatorics of reduction types, in particular

- Arithmetic of open and link sequences that controls the shapes of chains of $\mathbb{P}^1$s in special fibres of minimal regular normal crossing models,
- Methods for reduction types (RedType), their cores (RedCore), link chains (RedChain) and shapes (RedShape),

- Canonical labels for reduction types,
- Reduction types and their labels in TeX,
- Conversion between dual graphs, reduction type, and their labels:

$$\{\text{dual graphs}\} \rightleftarrows \{\text{reduction types}\} \rightleftarrows \{\text{labels}\}.$$

**Example** (Reduction types, labels and dual graphs).

```
> R = ReductionType("I2*-1g1-IV")
> print(R.Label())              # Canonical plain label
I2*-1g1-IV
> print(R.Label(tex=True))      # TeX label
```

$\mathrm{I}_2^*\text{-}1_{\mathrm{g}1}\text{-}\mathrm{IV}$

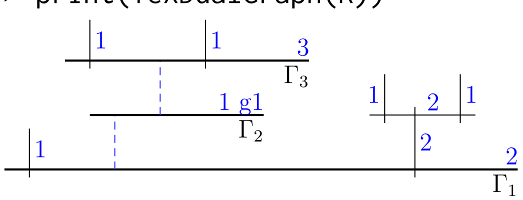
```
> print(R.TeX())                # Reduction type as a graph
```

$\mathrm{I}_2^*$ — $1_{\mathrm{g}1}$ — $\mathrm{IV}$

```
> print(R.DualGraph())          # Associated dual graph
DualGraph([2,1,3,1,1,1,2,1,1,2], [0,1,0,0,0,0,0,0,0,0],
  [[1,10],[1,2],[1,4],[2,3],[3,5],[3,6],[7,10],[7,8],[7,9]])
```

This is a dual graph on 10 components, of multiplicity 1, 2 and 3, and genus 0 and 1, and here is the picture of the corresponding special fibre. Principal components are thick horizontal lines marked with $\Gamma_1$, $\Gamma_2$, $\Gamma_3$, all other components are $\mathbb{P}^1$s, and dashed line indicate principal components meeting at a point.

```
> print(TeXDualGraph(R))
```




Taking the associated reduction type gives back R:

```
> G = DualGraph([3,1,2,1,1,1,2,1,1,2], [0,1,0,0,0,0,0,0,0,0],
  [[1,2],[1,4],[1,5],[2,3],[3,6],[3,10],[7,8],[7,9],[7,10]])
> print(G.ReductionType())
I2*-1g1-IV
```

```
def DeterminantBareiss(M)
```

Bareiss' algorithm to compute Det(M) in a stable way

## 10.1 Open and link chains

A reduction type is a graph that has principal types as vertices (like IV, 1g1, $\mathrm{I}_2^*$ above) and link chains as edges. Principal types encode principal components together with open chains, loops and D-links. The three functions that control multiplicities of open and link chains, and their depths are as follows:

```
def OpenSequence(m: int, d: int, includem=True) -> List[int]
```

```
Unique open sequence of type (m,d) for integers m>=1 and 1<d<m. It is of the form
   [m,d,...,gcd(m,d)]
with every three consecutive terms d_(i-1), d_i, d_(i+1) satisfying
   d_(i-1) + d_(i+1) = d_i * (integer > 1).
If includem=False, exclude the starting point m from the sequence.}
```

**Example** (OpenSequence).

```
> print(OpenSequence(6, 5))
[6, 5, 4, 3, 2, 1]
> print(OpenSequence(13, 8))
[13, 8, 3, 1]
```

```
def LinkSequence(m1: int, d1: int, m2: int, dk: int, n: int, includem=True) ->
  List[int]
```

```
Unique link sequence of type m1(d1-dk-n)m2, that is of the form [m1,d1,...,dk,m2] with n+1 terms
equal to gcd(m1,d1)=gcd(m2,dk) and satisfying the chain condition: for every three consecutive terms
   d_(i-1), d_i, d_(i+1)
we have
   d_(i-1) + d_(i+1) = d_i * (integer > 1).
If includem=False, exclude the endpoints m1,m2 from the sequence.
```

**Example** (LinkSequence).

```
> print(LinkSequence(3, 2, 3, 2, -1))
[3, 2, 3]
> print(LinkSequence(3, 2, 3, 2, 0))
[3, 2, 1, 2, 3]
> print(LinkSequence(3, 2, 3, 2, 1))
[3, 2, 1, 1, 2, 3]
```

```
def MinimalLinkDepth(m1: int, d1: int, m2: int, dk: int) -> int
```

```
Minimal depth of a link sequence between principal components of multiplicities m1 and m2 with
initial links d1 and dk.
Minimal depth of a chain d1,d2,...,dk of P1s between principal component of multiplicity m1, m2 and
initial link multiplicities d1,dk. The depth is defined as -1 + number of times GCD(d1,...,dk)
appears in the sequence.
For example, 5,4,3,2,1 is a valid link sequence, and MinimalLinkDepth(5,4,1,2) = -1 + 1 = 0.
```

**Example**. Example for MinimalLinkDepth from the description of the function:

```
> print(MinimalLinkDepth(5,4,1,2))
0
```

For another example, the minimal $n$ in the Kodaira type $\mathrm{I}_{\mathrm{n}}^{*}$ is 1. Here the chain links two components of multiplicity 2, and the initial multiplicities are 2 on both sides as well:

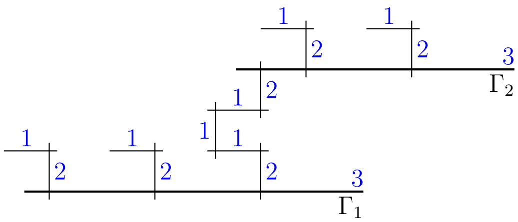
```
> print(MinimalLinkDepth(2,2,2,2))
1
```

Here is an example of a reduction type with a link chain between two components of multiplicity 3 and outgoing multiplicities 2 on both sides:

```
> R = ReductionType("IV*-(2)IV*")
```

Here is what its dual graph looks like:

```
> print(TeXDualGraph(R))
```




The link chain has gcd=GCD(3,2)=1 and

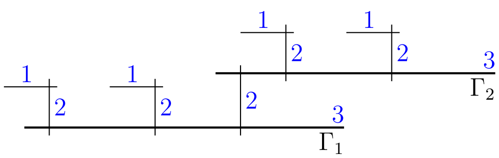
$$\text{depth} = -1 + \#1\text{'s}(=\text{gcd}) \text{ in the sequence } 3, 2, 1, 1, 1, 2, 3 = 2$$

This is the depth specified in round brackets in IV*-(2)IV*

```
> print(MinimalLinkDepth(3,2,3,2))   # Minimal possible depth for such a chain = -1
-1
> R1 = ReductionType("IV*-IV*")      # used by default when no expicit depth is specified
> R2 = ReductionType("IV*-(-1)IV*")
> assert R1==R2
```

Here is what its dual graph looks like:

```
> print(TeXDualGraph(R1))
```




The next two functions are used in Label to determine the ordering of chains (including loops and D-links), and default multiplicities which are not printed in labels.

```
def SortLinks(m, O)
```

Sort a sequence of multiplicities O by gcd with m, then by o. This is how open and loose multiplicities are sorted in reduction types.

**Example** (Ordering open multiplicities in reduction types).

```
> m = 6                        # principal component multiplicity
> O = [1,2,3,3,4,5]            # initial multiplicities for outgoing open chains
> SortLinks(6, O)              # sort them first by gcd(o,m), then by o mod m
> print(O)
[1, 5, 2, 4, 3, 3]
```

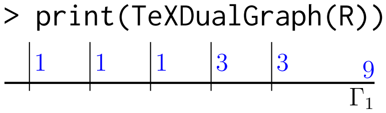
```
def DefaultMultiplicities(m1, o1, m2, o2, loop)
```

Default edge multiplicities for a component with given multiplicities and outgoing options.
Default edge multiplicities d1, d2 for a component with multiplicity m1, available outgoing multiplicities o1, and one with m2, o2.
loop: boolean specifies whether it is a loop or a link between two different principal components.

**Example** (DefaultMultiplicities). Let us illustrate what happens when we take a principal component $9^{1,1,1,3,3}$ and add five default loops of depth 2,2,1,2,3, to get a reduction type $9^{1,1,1,3,3}_{2,2,1,2,3}$. How do default loops decide which initial multiplicities to take?

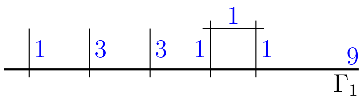
We start with a component of multiplicity $m = 9$ and open multiplicities $\mathcal{O} = \{1, 1, 1, 3, 3\}$.

```
> R = ReductionType("9^1,1,1,3,3")
> print(TeXDualGraph(R))
```




We can add a loop to it linking two 1's of depth 2 by

```
> R = ReductionType("9^1,1,1,3,3_{1-1}2")
> print(TeXDualGraph(R))
```

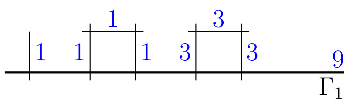



In this case, {1-1} does not need to be specified because this is the minimal pair of possible multiplicities in $\mathcal{O}$, as sorted by SortLinks:

```
> print(DefaultMultiplicities(9,[1,1,1,3,3],9,[1,1,1,3,3],True))
(1, 1)
> assert R == ReductionType("9^1,1,1,3,3_2")
```

After adding the loop, $\{1, 3, 3\}$ are left as potential outgoing multiplicities, so the next default loop links 3 and 3. Note that $1, 3$ is not a valid pair because $\gcd(1, 9) \neq \gcd(3, 9)$.

```
> print(DefaultMultiplicities(9,[1,3,3],9,[1,3,3],True))
(3, 3)
> R2 = ReductionType("9^1,1,1,3,3_2,2")       # 2 loops, use 1-1 and 3-3
> print(TeXDualGraph(R2))
```

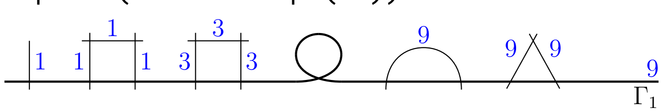



There are no pairs left, so the next three loops use $(m, m) = (9, 9)$

```
> print(DefaultMultiplicities(9,[1],9,[1],True))
(9, 9)
> R3 = ReductionType("9^1,1,1,3,3_2,2,1,2,3")
> assert R3 == ReductionType("9^1,1,1,3,3_{1-1}2,{3-3}2,{9-9}1,{9-9}2,{9-9}3")
```

This is what its dual graph looks like:

```
> print(TeXDualGraph(R3))
```




## 10.2 Principal component core (RedCore)

A core is a pair $(m, O)$ with 'principal multiplicity' $m \geq 1$ and 'outgoing multiplicities' $O = \{o_1, o_2, ...\}$ that add up to a multiple of $m$, and such that $\gcd(m, O) = 1$. It is implemented as the following type:

```
def Core(m: int, O: list[int]) -> 'RedCore'
```

Core of a principal component defined by multiplicity m and list O.

**Example** (Create and print a principal component core $(m, O)$).

```
> print(Core(8,[1,3,4]))     # Typical core - multiplicities add up to a multiple of m
8^1,3,4
> print(Core(8,[9,3,4]))     # Same core, as they are in Z/mZ
8^1,3,4
```

This is how cores are printed, with the exception of 7 cores of $\chi = 0$ (see below) that come from Kodaira types and two additional special ones D and T:

```
> print(Core(6,[1,2,3]))                   # from a Kodaira type
II
> print([Core(2,[1,1]),Core(3,[1,2])])     # two special ones
```

```
[D, T]
```

## 10.3 Basic invariants and printing

```
class RedCore
```

```
def definition(self)
```

Returns a string representation of a core in the form 'Core(m,O)'.

```
def Multiplicity(self)
```

Returns the principal multiplicity m of the principal component.

```
def Multiplicities(self)
```

Returns the list of outgoing chain multiplicities O, sorted with SortLinks.

```
def Chi(self)
```

Euler characteristic of a reduction type core (m,O), chi = m(2-|O|) + sum_(o in O) gcd(o,m)

```
def Label(self, tex=False)
```

Label of a reduction type core, for printing (or TeX if tex=True)

```
def TeX(self)
```

Returns the core label in TeX, same as Label with TeX=True.

**Example** (Core labels and invariants).

```
> C=Core(2,[1,1,1,1])
> print(C.Label())              # Plain label
I0*
> print(C.TeX())                # TeX label
```

$I_0^*$

```
> print(C.definition())         # How it can be defined
Core(2,[1,1,1,1])
> print(C.Multiplicity())       # Principal multiplicity m
2
> print(C.Multiplicities())     # Outgoing multiplicities O
[1, 1, 1, 1]
> print(C.Chi())                # Euler characteristic
0
```

```
def Cores(chi, mbound="all", sort=True)
```

Returns all cores (m,O) with given Euler characteristic chi<=2. When chi=2 there are infinitely many, so a bound on m must be given.

**Example** (Cores).

```
> print(Cores(2, mbound=4))         # Chi=2 (infinitely many), with bound for m
[1, D, T, 4^1,3]
> print(Cores(0))                   # 7 cores I0* ,IV, IV*, III, III*, II, II*
[I0*, IV, IV*, III, III*, II, II*]
> print([len(Cores(i)) for i in (0,-2,-4,-6,-8)])  # 7, 16, 43, 65, 64, ...
[7, 16, 43, 65, 64]
```

## 10.4 Link chains (RedChain)

Link chains between principal components fall into three classes: loops on a principal type, D-link on a principal type, and chains between principal types that link two of their loose edge endpoints. All of these are implemented in the class RedChain that carries class=cLoop, cD or cLoose, and keeps track of all the invariants.

```
def Link(Class, mi, di, mj=False, dj=False, depth=False, Si=False, Sj=False,
  index=False) -> 'RedChain'
```

`Define a RedChain from its invariants`

**Example** (Some link chains, with no principal types specified).

```
> print(Link(cLoop,2,1,2,1))    # loop
loop 2,1 -(0) 2,1
> print(Link(cD,2,2))           # D-link
D-link 2,2 -(1) 2,2
> print(Link(cLoose,2,2))       # to another (yet unspecified) principal type
loose 2,2 -(False) False,False
```

## 10.5 Invariants and depth

```
class RedChain
```

```
def GCD(self)
```

`GCD of all elements in the chain (=GCD(mi,di)=GCD(mj,di)).`

```
def Index(self)
```

`Index of the RedChain, used for distinguishing between chains.`

```
def SetDepth(self, n)
```

`Set the depth and depth string of the RedChain.`

```
def SetMinimalDepth(self)
```

`Set the depth of the RedChain to the minimal possible value.`

```
def DepthString(self)
```

`Return the string representation of the RedChain's depth.`

```
def SetDepthString(self, depth)
```

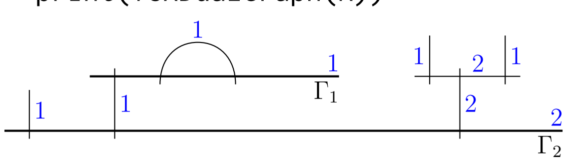
`Set how the depth is printed (e.g., "1" or "n").`

**Example** (Invariants of link chains). Take a genus 2 reduction type $\mathrm{I}_2 \overline{\phantom{x}}_1 \mathrm{I}{*}_2$ whose special fibre consists of Kodaira types $\mathrm{I}_2$ (loop of $\mathbb{P}^1$s) and $\mathrm{I}_2^*$ linked by a chain of $\mathbb{P}^1$s of multiplicity 1.

```
> R = ReductionType("I2-(1)I2*");
```

This is what its special fibre looks like:

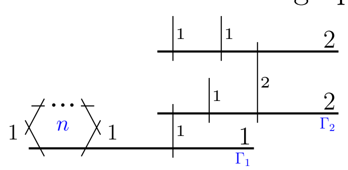
```
> print(TeXDualGraph(R))
```




There are two principal types R[1]=$\mathrm{I}_2$ and R[2]=$\mathrm{I}_2^*$, with a loop on R[1] (class cLoop=1), a link chain between them (class cLoose=3), and a D-link on R[2] (class cD=2) This is the order in which they are printed in the label.

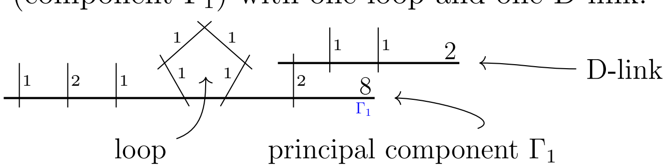
```
> print([R[1],R[2]])                # two principal types R[1] and R[2]
[I2-{1}, I2*-{1}]
> c1,c2,c3 = R.LinkChains()
> print(c1)
[1] loop c1 1,1 -(2) c1 1,1
> print(c2)
[2] loose c1 1,1 -(1) c2 2,1
> print(c3)
[3] D-link c2 2,2 -(2) 2,2
> print(c3.Class)                   # cLoop=1, *cD=2*, cLoose=3
2
> print(c3.GCD())                   # GCD of the chain multiplicities [2,2,2]
2
> print(c3.Index())                 # index in the reduction type
3
> c3.SetDepthString("n")            # change how its depth is printed in labels
> print(c3)                         #   and drawn in dual graphs of reduction types
[3] D-link c2 2,2 -(n) 2,2
> print(R.Label())
I2-(1)In*
```

This is what its dual graph looks like:



## 10.6 Principal components (RedPrin)

The classification of special fibre of mrnc models is based on principal types. For curves of genus $\geq 2$ such a type is a principal component with $\chi < 0$, together with its open chains, loops, chains to principal component with $\chi = 0$ (called D-links) and a tally of link chains to other principal components with $\chi < 0$, called loose links. For example, the following reduction type has only principal type (component $\Gamma_1$) with one loop and one D-link:




A principal type is implemented as the following python class.

```
def PrincipalType(m, g, O, Lloops, LD, Lloose, index=0)
```

```
Create a new principal type from its primary invariants:
m         multiplicity of the principal component, e.g. 8
g         geometric genus of the principal component, e.g. 0
O         outgoing multiplicities for open chains, e.g. 1,1,2
Lloops    list of loops [[di,dj,depth],...], e.g. [[1,1,3]]
LD        list of D-links [[di,depth],...], e.g. [[2,1]]  (m and all d_i must be even)
Lloose    list of loose multiplicities, e.g. [8]
```

**Example** (Construction). We construct the principal type from example above. It has $m = 8$, $g = 0$, open multiplicities 1,1,2, loop $1-1$ of depth 3, a D-link with outgoing multiplicity 2 of depth 1, and no loose chains (so that it is a reduction type in itself).

```
> S = PrincipalType(8,0,[1,1,2],[[1,1,3]],[[2,1]],[])
```

```
class RedPrin
```

```
def Multiplicity(self)
```

Principal multiplicity m of a principal type

```
def GeometricGenus(self)
```

Geometric genus g of a principal type S=(m,g,O,...)

```
def Index(self)
```

Index of the principal component in a reduction type, 0 if freestanding

```
def Chains(self, Class=0)
```

Sequence of chains of type RedChain originating in S. By default, all (loops, D-links, loose) are returned, unless class is specified.

```
def OpenMultiplicities(self)
```

Sequence of open multiplicities S`O of a principal type, sorted

```
def LinkMultiplicities(self)
```

Sequence of link multiplicities S`L of a principal type, sorted as in label

```
def Loops(self)
```

Sequence of chains in S representing loops (class cLoop)

```
def DLinks(self)
```

Sequence of chains in S representing D-links (class cD)

```
def LooseChains(self)
```

Sequence of loose chains of a principal type, sorted

```
def LooseMultiplicities(self)
```

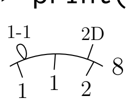
Sequence of loose multiplicities of a principal type, sorted

```
def definition(self) -> str
```

Returns a string representation of a principal type in the form of the PrincipalType constructor.

**Example** (Invariants). We continue with the principal type above. It has $m = 8$, $g = 0$, open multiplicities 1,1,2, loop $1-1$ of depth 3, a D-link with outgoing multiplicity 2 of depth 1, and no loose chains (so that it is a reduction type in itself).

```
> S = PrincipalType(8,0,[1,1,2],[[1,1,3]],[[2,1]],[])
> print(S)
8^1,1,1,1,2,2_3,1D
> print(S.TeX(standalone=True))     # How it appears in the tables
```



```
> print(S.Multiplicity())          # Principal component multiplicity
8
> print(S.GeometricGenus())        # Geometric genus of the principal component
0
> print(S.OpenMultiplicities())    # Open chain initial multiplicities O=[1,1,2]
[1, 1, 2]
> print(S.Loops())                 # Loops (of type RedChain)
[loop c0 8,1 -(3) c0 8,1]
> print(S.DLinks())                # D-Links (of type RedChain)
[D-link c0 8,2 -(1) 2,2]
> print(S.LooseMultiplicities())   # Loose link multiplicities
[]
> print(S.LinkMultiplicities()) # All initial link multiplicities (loops, D-links, loose)
[1, 1, 2]
> print(S.definition())            # evaluatable string to reconstruct S
PrincipalType(8,0,[1,1,2],[[1,1,3]],[[2,1]],[])
```

```
def GCD(self)
```

Return GCD(m,O,L) for a principal type

```
def Core(self)
```

Core of a principal type - no genus, all non-zero link multiplicities put to 0, and gcd(m,O)=1

```
def Chi(self)
```

Euler characteristic chi of a principal type (m,g,O,Lloops,LD,Lloose), chi = m(2-2g-|O|-|L|) + sum_(o in O) gcd(o,m), where L consists of all the link multiplicities in Lloops (2 from each), LD (1 from each), Lloose (1 from each)

```
def LGCD(self)
```

Outgoing link pattern of a principal type = multiset of GCDs of loose edges with m.

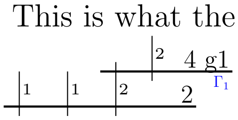
**Example** (GCD). Define a principal type by its primary invariants: $m = 4$, $g = 1$, open multiplicities $\mathcal{O} = [2]$, no loops, one D-link with initial multiplicity 2 and length 1, and no loose links

```
> S = PrincipalType(4, 1, [2], [], [[2, 1]], [])
> print(S.GCD())              # its GCD(m,O,L)=GCD(4,[2],[2])=2
2
> print(S)                    # which is seen as [2] in its name
[2]Dg1_1D
```

Note, however, it is not a multiple of 2 of another principal component type because its D-link is primitive. The special fibre is not a multiple of 2.

```
> print(ReductionType("[2]Dg1_1D").DualGraph().Multiplicities())
[4, 2, 2, 1, 1, 2]
```

This is what the special fibre looks like:




```
def Weight(self) -> list[int]
```

Sequence [chi,m,-g,#loose,#Ds,#loops,#O,O,loops,Ds,loose] that determines the weight of a principal type, and characterises it uniquely.

```
def __eq__(self, other)
```

```
Compare two principal types by their weight.
```

```
def __lt__(self, other)
```

```
Compare two principal types by their weight.
```

```
def __le__(self, other)
```

```
Compare two principal types by their weight.
```

```
def __gt__(self, other)
```

```
Compare two principal types by their weight.
```

```
def __ge__(self, other)
```

```
Compare two principal types by their weight.
```

**Example** (Sorting principal types by Weight in increasing order).

```
> L = PrincipalTypes(-2,[4]) + PrincipalTypes(-2,[2,2])
> print([S.Weight() for S in L]);
[[-2, 4, 0, 1, 0, 0, 2, 1, 3, 4], [-2, 4, 0, 1, 1, 0, 1, 2, 2, 0, 4], [-2, 2, 0, 2, 0, 0,
  2, 1, 1, 2, 2], [-2, 2, 0, 2, 1, 0, 0, 2, 1, 2, 2]]
> print(sorted(L,key=lambda S: S.Weight()))
[D==, [2]_D==, 4^1,3=, [2]D_D=]
```

```
def Label(self, tex=False, loose=False, wrap=False, returnpieces=False) -> str
```

```
Ascii Label or TeX label of a principal type.
Setting tex=True prints the tex label, in \redtype{...} format by default, unless wrap=False
Setting loose=True prints outgoing loose edges as well (standalone principal type).
```

```
def TeX(self, length="35pt", label=False, standalone=False)
```

```
TeX a principal type as a TikZ arc with outer and inner lines, loops, and Ds.
label=True puts its label underneath.
standalone=True wraps it in \tikz.
```

```
def PrincipalTypes(chi: int, arg=None, semistable=False, withlgcds=False,
  sort=True) -> Tuple[List[RedPrin], List[List[int]]]
```

```
Principal types with a given Euler characteristic chi, and optional restrictions.
Returns list of types, or (list of types, discovered GCDs of loose chains) if withlgcds=True.
Can be used as either:
  PrincipalTypes(chi)          - all
  PrincipalTypes(chi,C)        - with a given core C
  PrincipalTypes(chi,LGCDs)    - with a given sequence of loose chain lgcds
In all three cases can restrict to semistable types, setting semistable=True
```

**Example**.

```
> comps, lgcds = PrincipalTypes(-1, withlgcds = True)
> print(len(comps))                       # all principal types with chi=-1
13
> print(lgcds)                            # their possible edge gcds (see RedShape)
[[1, 1, 1], [1], [1, 2], [3]]
> print(PrincipalTypes(-1,[1,2]))         # select those with edge gcds = [1,2]
[D-{1}=]
> print([len(PrincipalTypes(-n)) for n in range(1,8+1)])   # all with chi=-1,...
[13, 83, 75, 277, 176, 591, 352, 1068]
```

```
def PrincipalTypeFromWeight(w: list[int]) -> RedPrin
```

```
Create a principal type S from its weight sequence w (=Weight(S)).
```

**Example**.

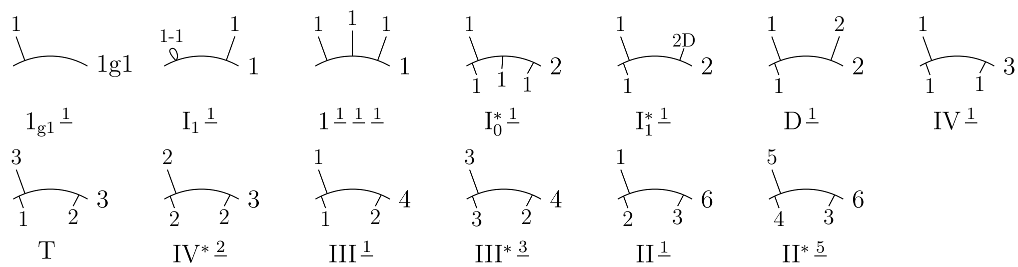
```
> S = PrincipalType(8,0,[4,2],[[1,1,1]],[[2,1]],[6])  # Create a principal type
> w = S.Weight()                                      # weight encodes chi, m, g etc.
> print(w)                                            #   and characterizes S
[-26, 8, 0, 1, 1, 1, 2, 2, 4, 1, 1, 1, 2, 1, 6]
> print(PrincipalTypeFromWeight(w).definition())      # Reconstruct S from the weight
PrincipalType(8,0,[2,4],[[1,1,1]],[[2,1]],[6])
```

```
def PrincipalTypesTeX(T, width=10, scale=0.8, sort=True, label=False,
  length="35pt", yshift="default")
```

```
TeX a list of principal types as a rectangular table in a TikZ picture.
label=True puts the principal type label underneath.
sort=True sorts the types by Weight first, in increasing order.
yshift controls the y-axis shift after every row, based on label presence.
width controls the number of principal types per row.
scale controls the TikZ picture global scale.
```

**Example** (TeX for principal types). Here are the 13 principal types with chi=-1 (10 Kodaira + 3 'exotic')

```
> L = PrincipalTypes(-1)
> print(PrincipalTypesTeX(L, label=True, width=7, yshift=2.2))
```

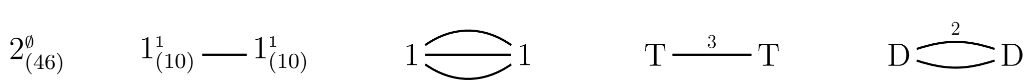


## 10.7 RedShape

A reduction type a graph whose vertices are principal types (type RedPrin) and edges are link chains. They fall naturally into 'shapes', where every vertex only remembers the Euler characteristic $\chi$ of the type, and edge the gcd of the chain. Thus, the problem of finding all reduction types in a given genus (see ReductionTypes) reduces to that of finding the possible shapes (see Shapes) and filling in shape components with given $\chi$ and gcds of loose edges (see PrincipalTypes).

**Example** (Table of all genus 2 shapes, with numbers of principal type combinations.). Here is how this works in genus 2. The 104 families of reduction types break into five possible shapes, with all but three types in the first two shape (46 and 55 types, respectively):

```
> L = Shapes(2)
> print("\\qquad ".join([D[0].TeX(shapelabel=D[1]) for D in L]))
```



| | | | | |
|---|---|---|---|---|
| 46 | 55 | 1 | 1 | 1 |

```
class RedShape
```

```
def TeX(self, scale=1.5, center=False, shapelabel="", complabel="default",
  boundingbox=False)
```

Tikz a shape of a reduction graph, and, if required the bounding box x1, y1, x2, y2.

```
def Graph(self)
```

Returns the underlying undirected graph G of the shape.

```
def __len__(self)
```

Returns the number of vertices in the graph G underlying the shape.

```
def Vertices(self)
```

Returns the vertex set of G as a graph.

```
def Edges(self)
```

Returns the edge set of G as a graph.

```
def DoubleGraph(self)
```

Returns the vertex-labelled double graph D of the shape.

```
def Chi(self, v=None)
```

Returns the Euler characteristic chi(v) <= 0 of the vertex v, or total Euler characteristic if v=None

```
def LGCDs(self, v)
```

Returns the LGCDs of a vertex v that together with chi determine the vertex type (chi, lgcds).

```
def VertexLabels(self)
```

Returns a sequence of -chi's for individual components of the shape S.

```
def EdgeLabels(self)
```

Returns a list of edges v_i -> v_j of the form [i, j, edgegcd].

```
def Shape(V: list[int], E: list[list[int]]) -> RedShape
```

Constructs a graph shape from the vertex data V and list of edges with multiplicities E.
The format is as in shapes*.txt data files:
V = sequence of -chi's for individual components
E = list of edges v_i->v_j of the form [i,j,edgegcd1,edgegcd2,...]

**Example** (Printing a shape).

```
> print(ReductionType("IV-IV-IV").Shape())  # 3 vertices with chi=-1,-2,-1 and 2 edges
Shape([1,2,1],[[1,2,1],[2,3,1]])
> print(ReductionType("1---1").Shape())     # 2 vertices with chi=-1,-1 and a triple edge
Shape([1,1],[[1,2,1,1,1]])
```

```
def IsIsomorphic(S1: RedShape, S2: RedShape) -> bool
```

Check whether two shapes are isomorphic via their double graphs

**Example** (Shape isomorphism testing).

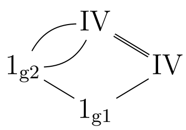
```
> S1 = Shape([1, 2, 3], [[1, 2, 3], [2, 3, 1], [1, 3, 2]])
> S2 = Shape([2, 3, 1], [[1, 2, 1], [2, 3, 2], [1, 3, 3]])  # rotate the graph
> assert IsIsomorphic(S1, S2)
> S3 = Shape(S1.VertexLabels(),S1.EdgeLabels())             # reconstruct S1
> assert IsIsomorphic(S1, S3)
```

```
def Shapes(genus, filemask="data/shapes{}.txt")
```

Returns all shapes in a given genus, assuming they were downloaded in data/

**Example** (Graph, DoubleGraph and primary invariants for shapes). Under the hood of shapes of reduction types are their labelled graphs and associated 'double' graphs. As an example, take the following reduction type:

```
> R=ReductionType("1g2--IV=IV-1g1-c1")
> print(R.TeX())
```

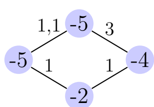



There are four principal types, and they become vertices of R.Shape() whose labels are their Euler characteristics $-5, -2, -4, -5$. The edges are labelled with GCDs of the link chain between the types. For example:

— the link chain 1g2-1g1 of gcd 1 becomes the label "1",
— the link chain IV=IV of gcd 3 becomes "3",
— the two chains 1g2–IV of gcd 1 become "1,1"

on the corresponding edges.

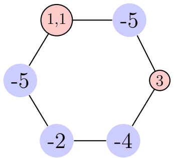
```
> S=R.Shape()
> print(S)
Shape([5,2,4,5],[[1,2,1],[1,4,1,1],[2,3,1],[3,4,3]])
> print(TeXGraph(S.Graph()))
```




```
> print(S.Vertices())          # Indexed set of vertices of S.Graph(), numbered from 1
[1, 2, 3, 4]
> print(S.Edges())             #   and edges [ (from_vertex, to_vertex), ... ]
[(1, 2), (1, 4), (2, 3), (3, 4)]
> print(S.VertexLabels())      # [-chi] for each type
[5, 2, 4, 5]
> print(S.EdgeLabels())        # [ [from_vertex, to_vertex, gcd1, gcd2, ...], ...]
[[1, 2, 1], [1, 4, 1, 1], [2, 3, 1], [3, 4, 3]]
```

MinimumWeightPaths is implemented in python for graphs with labelled vertices but not edges. To use them for shapes, the underlying graphs are converted to graphs with only labelled vertices. This is done simply by introducing a new vertex on every edge which carries the corresponding edge label. For compactness, if the label is "1" (most common case), we don't introduce the vertex at all. This is called the double graph of the shape:

```
> blue = "circle,scale=0.7,inner sep=2pt,fill=blue!20"       # former vertices
> red  = "circle,draw,scale=0.5,inner sep=2pt, fill=red!20"  # former edges
> D    = S.DoubleGraph()
> bluered = lambda v: blue if sum(GetLabel(D,v)) <= 0 else red
> print(TeXGraph(D, scale:=1, vertexnodestyle=bluered))
```




These are used in isomorphism testing for shapes, and to construct minimal paths.

## 10.8 Labelled graphs and minimum paths

```
def Graph(vertices, edges=[])
```

```
Construct a graph from vertices (or their number) and edges, numbered from 1
For example Graph(3,[[1,2],[2,3]]) or Graph([3,4,5],[[3,4],[4,5]])
```

```
def IsLabelled(G, v)
```

```
Determines if vertex v in graph G has an associated label.
```

```
def IsLabelled(G)
```

```
Checks if all vertices in graph G have an assigned label.
```

```
def GetLabel(G, x)
```

```
Retrieves the label of a vertex or edge x from graph G.
```

```
def GetLabels(G)
```

```
Returns a list of labels assigned to the vertices of graph G.
```

```
def AssignLabel(G, v, label)
```

```
Assign a label to the vertex v in graph G.
```

```
def AssignLabels(G, labels)
```

```
Assigns labels to the vertices of graph G based on the provided list of labels.
```

```
def DeleteLabels(G)
```

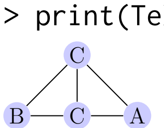
```
Deletes the labels from all vertices in the graph G if they exist.
```

```
def MinimumWeightPaths(D)
```

```
Determines minimum weight paths in a connected labelled undirected graph, returning weights and
possible vertex index sequences.
Minimum weight paths for a labelled undirected graph (e.g. double graph underlying shape)
returns W=bestweight [<index, v_label, jump>,...] (characterizes D up to isomorphism)
  and I=list of possible vertex index sequences
For example for a rectangular loop G with all vertex chis=1 and edges as follows
  V:=[1,1,1,1]; E:=[[1,2,1],[2,3,1],[3,4,2],[1,4,1,1]]; S:=Shape(V,E);
the double graph D has 6 vertices and 6 edges in a loop, and here minimum weight W is
  W = [<0,[-1],False>,<0,[-1],False>,<0,[-1],False>,<0,[1,1],False>,<0,[-1],False>,
       <0,[2],False>,<1,[-1],True>]
The unique trail T[1] (generally Aut D-torsor) is D.3->D.2->D.1->...->D.3, encoded
  T = [[3,2,1,6,4,5,3]]
```

**Example** (Minimum weight paths).

```
> G = Graph(4,[(1,2),(2,3),(3,4),(4,1),(1,3)])
> AssignLabels(G, ["C", "B", "C", "A"])
> print(TeXGraph(G))
```




Now we calculate minimum weight paths:

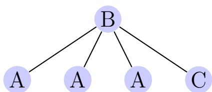
```
> P, a = MinimumWeightPaths(G)
```

Print the minimal path and the trails, both from one odd degree vertex to the other one:

```
> print("P:", P)
P: [(0, 'C', False), (0, 'A', False), (0, 'C', False), (0, 'B', False), (1, 'C', False),
  (3, 'C', True)]
> print("a:", a)
a: [[1, 4, 3, 2, 1, 3], [3, 4, 1, 2, 3, 1]]
```

Here is another graph on five vertices, this time not Eulerian

```
> G = Graph(5,[(2,1),(2,3),(2,4),(2,5)])
> AssignLabels(G, ["A", "B", "A", "A", "C"])
> print(TeXGraph(G))
```




Calculate minimum weight path, which is A-B-A, A-2-C (where 2 is 'second vertex on the path')

```
> P, a = MinimumWeightPaths(G)
```

Print the minimal path

```
> print("P:", P)
P: [(0, 'A', False), (0, 'B', False), (0, 'A', True), (0, 'A', False), (2, 'B', False),
  (0, 'C', True)]
```

There are 6 ways to trace this path, and they form an Aut(G)=S3-torsor. The first one is

```
> print(f"One trail out of {len(a)} is {a[0]}")
One trail out of 6 is [1, 2, 3, 4, 2, 5]
```

```
def GraphLabel(G, full=False, usevertexlabels=True)
```

```
Generate a graph label based on a minimum weight path, determines G up to isomorphism.
The label is constructed by iterating through the minimum weight path and formatting
the vertices and edges with labels, if present.
If full=True, returns also P, T from MinimumWeightPaths(G) for vertex recoding
```

```
def StandardGraphCoordinates(G)
```

```
Vertex coordinate lists x,y for planar drawing
```

```
def TeXGraph(G, x="default", y="default", labels="default", scale=0.8, xscale=1,
  yscale=1, vertexlabel="default", edgelabel="default",
  vertexnodestyle="default", edgenodestyle="default", edgestyle="default")
```

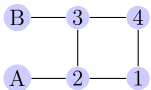
```
Generate TikZ code for drawing a small planar graph.
Parameters:
- G: An connected undirected networkx graph.
- x, y: Coordinates of vertices.
- labels: Vertex labels ("none", "default", or a list of strings).
- scale: Overall scaling factor for the graph.
- xscale, yscale: Scaling factors for x and y dimensions.
- vertexlabel, edgelabel: Functions or strings for labeling vertices/edges.
- vertexnodestyle, edgenodestyle, edgestyle: Functions or strings defining styles for nodes/edges.

Returns:
- TikZ code as a string.
```

```
def GraphFromEdgesString(edgesString)
```

```
Construct a graph from a string encoding edges such as "1-2-3-4, A-B, C-D", assigning the vertex
labels to the corresponding strings.
```

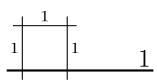
**Example.**

```
> G = GraphFromEdgesString("1-2-3-4-1, 2-A, 3-B")
> print(GraphLabel(G))
[2]-[1]-[4]-[3]-[B]&[A]-1-4
> print(TeXGraph(G))
```

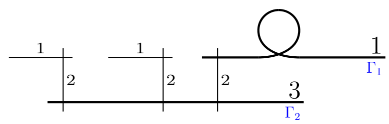



## 10.9 Dual graphs (GrphDual)

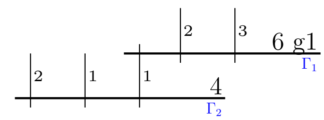
A dual graph is a combinatorial representation of the special fibre of a model with normal crossings. It is a multigraph whose vertices are components $\Gamma_i$, and an edge corresponds to an intersection point of two components. Every component $\Gamma$ has **multiplicity** $m = m_\Gamma$ and geometric **genus** $g = g_\Gamma$. Here are three examples of dual graphs, and their associated reduction types; we always indicate the multiplicity of a component (as an integer), and only indicate the genus when it is positive (as g followed by an integer).




Type $\mathrm{I}_4$ (genus 1)




Type $\mathrm{I}_1$ – $\mathrm{IV}*$ (genus 2)




Type $\mathrm{II}_{\mathrm{g1}}$ – III (genus 8).

A component is **principal** if it meets the rest of the special fibre in at least 3 points (with loops on a component counting twice), or has $g > 0$. The first example has no principal components, and the other two have two each, $\Gamma_1$ and $\Gamma_2$.

This section provides a class (**GrphDual**) for representing dual graphs and their manupulation and invariants.

## 10.10 Default construction

```
def DualGraph(m: List[int], g: List[int], edges: List[List[int]], comptexnames =
  "default") -> 'GrphDual'
```

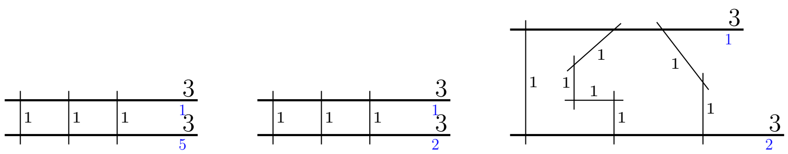
```
Construct a dual graph (GrphDual) from multiplicities and genera of vertices, and edges of the
underlying graph.
Parameters:
m: List of multiplicities for each provided component
g: List of genera for each provided component
edges: List of edges in the form
  [i,j]          - intersection point between component #i and component #j (1<=i,j<=n)
  [i,0,d1,d2,...] - open chain from component #i (1<=i<=n)
  [i,j,d1,d2,...] - link chain from component #i to component #j (1<=i,j<=n)
comptexnames (optional): 'default', function to name components, or a list of names for components.
```

**Example** (Constructing a dual graph).

```
> m  = [3,1,1,1,3]                             # multiplicities of c1,c2,c3,c4,c5
> g  = [0,0,0,0,0]                             # genera of c1,c2,c3,c4,c5
> E  = [[1,2],[1,3],[1,4],[2,5],[3,5],[4,5]]   # edges c1-c2,... as 2-tuples or lists
> G1 = DualGraph(m,g,E)
> print(G1)
DualGraph([3,1,1,1,3], [0,0,0,0,0], [[1,2],[1,3],[1,4],[2,5],[3,5],[4,5]])
> m  = [3,3]              # Principal components and chains (same graph)
> g  = [0,0]
> E  = [[1,2,1],[1,2,1],[1,2,1]]
```

```
> G2 = DualGraph(m,g,E)
> print(G2)
DualGraph([3,3,1,1,1], [0,0,0,0,0], [[1,3],[1,4],[1,5],[2,3],[2,4],[2,5]])
> m  = [3,3]
> g  = [0,0]              # Principal components, different chains
> E  = [[1,2,1],[1,2,1,1],[1,2,1,1,1,1]]
> G3 = DualGraph(m,g,E)
> print(G3)
DualGraph([3,3,1,1,1,1,1,1,1], [0,0,0,0,0,0,0,0,0],
  [[1,3],[1,4],[1,6],[2,3],[2,5],[2,9],[4,5],[6,7],[7,8],[8,9]])
```

This is what the three special fibres look like (with component names in blue):




**Example** (Printing dual graph as a string and reconstructing it).

```
> R = ReductionType("1g1-1g2-1g3-c1")
> G = R.DualGraph();        # Triangular dual graph on 3 vertices and 3 edges
> print(G)
DualGraph([1,1,1], [3,2,1], [[1,2],[1,3],[2,3]])
> G2 = eval(str(G))         # and reconstructed back
> print(G2)
DualGraph([1,1,1], [3,2,1], [[1,2],[1,3],[2,3]])
```

## 10.11 Step by step construction

```
class GrphDual
```

```
def __init__(self)
```

```
Initialize an empty dual graph
```

```
def AddComponent(self, name: str, genus: int, multiplicity: int, texname=None)
```

```
Adds a component (vertex) to the graph with attributes m, g, and optional texname.
Returns name of the added component (which is given by name if <>None, <>"")
```

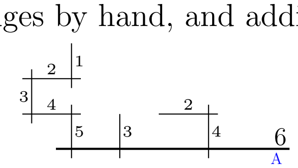
```
def AddEdge(self, node1, node2)
```

```
Adds an edge between two components (vertices) in the graph.
```

```
def AddChain(self, c1: str, c2: Union[str, None], mults: List[int])
```

```
Adds a chain of P1s with multiplicities between c1 and c2. Adds as many vertices as
there are multiplicities in 'mults', and links them in a chain starting at c1 and
ending at c2 (if c2 is provided, else it's an open chain).
```

**Example** (Type II* reduction). This is how we can construct the dual graph of the type II* elliptic curve, creating some components and edges by hand, and adding the rest as open chains.




```
> G = GrphDual()
```

```
> c1 = G.AddComponent("A", genus=0, multiplicity=6)   # Called 'A', multiplicity 6
> c2 = G.AddComponent("", genus=0, multiplicity=3)    # default name ('c2')
> G.AddEdge(c1,c2)              # Link the two (shortest chain)
> G.AddChain(c1,None,[4,2])     # The other two chains
> G.AddChain(c1,None,[5,4,3,2,1])
> print(G.Components())
['A', 'c2', 'c3', 'c4', 'c5', 'c6', 'c7', 'c8', 'c9']
> print(G.ReductionType())
II*
```

## 10.12 Global methods and arithmetic invariants

```
def Graph(self) -> nx.Graph
```

Returns the underlying graph.

```
def Components(self) -> list
```

Returns the list of components (vertices) of the dual graph.

```
def IsConnected(self)
```

True if underlying graph is connected

```
def HasIntegralSelfIntersections(self)
```

Are all component self-intersections integers

```
def AbelianDimension(self)
```

Sum of genera of components

```
def ToricDimension(self)
```

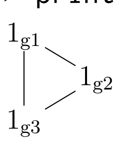
Number of loops in the dual graph

```
def IntersectionMatrix(self)
```

Intersection matrix for a dual graph, whose entries are pairwise intersection numbers of the components.

**Example**. Here is the dual graph of the reduction type $1_{g3}-1_{g2}-1_{g1}-c_1$, consisting of three components genus 1,2,3, all of multiplicity 1, connected in a triangle.

```
> G = DualGraph([1,1,1],[1,2,3],[[1,2],[2,3],[3,1]])
> assert G.IsConnected()                       # Check the dual graph is connected
> assert G.HasIntegralSelfIntersections()     #   and every component c has c.c in Z
> print(G.AbelianDimension())                  # genera 1+2+3 => 6
6
> print(G.ToricDimension())                    # 1 loop       => 1
1
> print(G.ReductionType().TeX())
```




```
> print(G.IntersectionMatrix())                # Intersection(G,v,w) for v,w components
[[-2, 1, 1], [1, -2, 1], [1, 1, -2]]
```

```
def PrincipalComponents(self)
```

```
Return a list of indices of principal components.
A vertex is a principal component if either its genus is greater than 0
or it has 3 or more incident edges (counting loops twice).
In the exceptional case [d]I_n one component is declared principal.
```

```
def ChainsOfP1s(self)
```

```
Returns a sequence of tuples [(v1,v2,[chain multiplicities]),...] for chains of P1s between
principal components, and v2=None for open chains
```

```
def ReductionType(self)
```

```
Reduction type corresponding to the dual graph
```

## 10.13 Contracting components to get a mrnc model

```
def ContractComponent(self, c, checks=True)
```

```
Contract a component in the dual graph, assuming it meets one or two components, and has genus 0.
```

```
def MakeMRNC(self)
```

```
Repeatedly contract all genus 0 components of self-intersection -1, resulting in a minimal model
with normal crossings.
```

```
def Check(self)
```

```
Check that the graph is connected and self-intersections are integers.
```

**Example** (Contracting components).

```
> G = DualGraph([1,1],[1,0],[[1,2,1,1,1]])  # Not a minimal rnc model
> print(G.Components(),[G.Intersection(v,v) for v in G.Components()])
['1', '2', 'c3', 'c4', 'c5'] [-1, -1, -2, -2, -2]
> G.ContractComponent("2")   # Remove the last component
> G.ContractComponent("c5")  #   and then the one before that
> print(G.Components())
['1', 'c3', 'c4']
> print(G)
DualGraph([1,1,1], [1,0,0], [[1,2],[2,3]])
> G.MakeMRNC()               # Contract the rest of the chain
> print(G.Components())
['1']
> print(G)
DualGraph([1], [1], [])
> print(G.ReductionType())   # Associated reduction type
1g1
```

## 10.14 Invariants of individual vertices

```
def HasComponent(self, c)
```

```
Test whether the graph has a component named c
```

```
def Multiplicity(self, c)
```

```
Returns the multiplicity m of vertex c from the graph.
```

```
def Multiplicities(self) -> list
```

Returns the list of multiplicities of components.

```
def Genus(self, c)
```

Returns the geometric genus g of vertex c from the graph.

```
def Genera(self) -> list
```

Returns the list of geometric genera of components.

```
def Neighbours(self, c)
```

List of incident vertices, with each loop contributing the vertex itself twice

```
def Intersection(self, c1, c2)
```

Compute the intersection number between components c1 and c2 (or self-intersection if c1=c2).

**Example** (Cycle of 5 components).

```
> G = DualGraph([1], [1], [[1,1,1,1,1,1]])
> C = G.Components()
> print(C)
['1', 'c2', 'c3', 'c4', 'c5']
> assert G.HasComponent("c2")
> print(G.Multiplicity("c2"))
1
> print(G.Genus("c2"))
0
> print([[G.Intersection(v, w) for v in C] for w in C])    # = G.IntersectionMatrix()
-2 1 0 0 1
1 -2 1 0 0
0 1 -2 1 0
0 0 1 -2 1
1 0 0 1 -2
```

## 10.15 Reduction Types (RedType)

Now we come to reduction types, implemented through the class RedType. They can be constructed in a variety of ways:

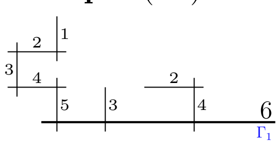
| | |
|---|---|
| `ReductionType(m,g,O,L)` | Construct from a sequence of components (including all principal ones), their multiplicities m, genera g, outgoing multiplicities of open chains O, and link chains L beween them, e.g.<br>`ReductionType([1],[0],[[]],[[1,1,0,0,3]])` (Type $I_3$) |
| `ReductionTypes(g)` | All reduction types in genus g. Can restrict to just semistable ones and/or ask for their count instead of actual the types, e.g.<br>`ReductionTypes(2)` (all 104 genus 2 types)<br>`ReductionTypes(2, countonly=True)` (only count them)<br>`ReductionTypes(2, semistable=True)` (7 semistable ones) |
| `ReductionType(label)` | Construct from a canonical label, e.g.<br>`ReductionType("I3")` |
| `ReductionType(G)` | Construct from a dual graph, e.g.<br>`ReductionType(DualGraph([1],[1],[]))` (good elliptic curve) |
| `ReductionTypes(S)` | Reduction types with a given shape, e.g.<br>`ReductionTypes(Shape([2],[]))` (46 of the genus 2 types) |

Conversely, from a reduction type we can construct its dual graph (`R.DualGraph()`) and a canonical label `R.Label()`), and these functions are also described in this section. Finally, there are functions to draw reduction types in TeX (`R.TeX()`).

```
def ReductionType(*args) -> 'RedType'
```

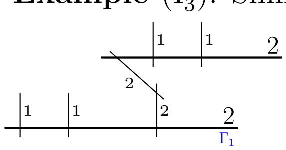
```
Reduction type from either:
ReductionType(label: Str)    reduction type from a label, e.g. "I3"
ReductionType(G: GrphDual)   reduction type from a dual graph
ReductionType(m, g, O, L) reduction type from sequence of components, their invariants, and chains
of P1s:
  m = sequence of multiplicities of components c_1,...,c_k
  g = sequence of their geometric genera
  O = outgoing multiplicities of open chains, one sequence for each component
  L = link chains, of the form
      [[i,j,di,dj,n],...] - link chain from c_i to c_j with multiplicities m[i],di,...,dj,m[j], of
      depth n
      n can be omitted, and chain data [i,j,di,dj] is interpreted as having minimal possible depth.
```

**Example** ($II^*$). We construct Kodaira type II∗ as a reduction type




```
> m = [6]          # multiplicity of one starting component Gamma_1
> g = [0]          # their geometric genera
> O = [[3, 4, 5]]  # outgoing multiplicities of open chains from each of them
> L = []           # link chains
> R = ReductionType(m, g, O, L)
> print(R.Label())
II*
> assert R == ReductionType("II*")    # same type from label
```

**Example** ($I_3^*$). Similarly, we construct Kodaira type $I_3^*$ as a reduction type




```
> m = [2, 2]                # multiplicities of starting components Gamma_1, Gamma_2
> g = [0, 0]                # their geometric genera
```

```
> O = [[1, 1], [1, 1]]   # outgoing multiplicities of open chains from each of them
> L = [[1, 2, 2, 2, 3]]  # link chains [[i,j, di,dj ,optional depth],...]
> R = ReductionType(m, g, O, L)
> print(R.Label())
I3*
> assert R == ReductionType("I3*")    # same type from label
```

```
def ReductionTypes(arg, *args, **kwargs)
```

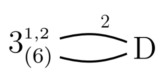
```
ReductionTypes(g: int, [countonly=False, semistable=False, elliptic=False])
  All reduction types in genus g<=6 or their count (if countonly=True; faster).
  semistable=True restricts to semistable types, elliptic=True (when g=1) to Kodaira types of
  elliptic curves.
ReductionTypes(S: RedShape, [countonly=False, semistable=False])
  Sequence of reduction types with a given shape S, again semistable if necessary, and/or their
  count
  If countonly=True, only return the number of types (faster).
returns a sequence of RedType's or an integer if countonly=True
```

**Example** (Reduction types in a given genus). Here are all reduction types for elliptic curves (10 Kodaira types), the count for genus 2 (104 Namikawa-Ueno types) and the count for semistable types in genus 3.

```
> print(ReductionTypes(1, elliptic=True))
[1g1, I1, I0*, I1*, IV, IV*, III, III*, II, II*]
> print(ReductionTypes(2, countonly=True))
104
> print(ReductionTypes(3, semistable=True, countonly=True))
42
```

**Example** (Reduction types with a given shape). There are 1901 reduction types in genus 3, in 35 different shapes. Here is one of the more 'exotic' ones, with 6 types in it. It has two vertices with $\chi = -3$ and $\chi = -1$ and two edges between them, with gcd 1 and 2.

```
> S = Shape([3, 1], [[1, 2, 1, 2]])
> print(S.TeX())
```




```
> L = ReductionTypes(S)
> print(L)
[I0*-=D, I1*-=D, III--{2-2}D, III*-{2-2}-D, II--{2-2}D, II*-{4-2}-D]
> print("\\qquad".join(R.TeX(scale=1.5, forcesups=True) for R in L))
```


```
class RedType
```

```
def Chi(self)
```

```
Total Euler characteristic of R
```

```
def Genus(self)
```

```
Total genus of R
```

**Example**.

```
> R = ReductionType("III=(3)III-{2-2}II-{6-12}18g2^6,12")
> print(R.Label())      # Canonical label
[6]Tg2-{12-6}II-{2-2}III=(3)III
```

```
> print(R.Genus())      # Total genus
43
```

```
def IsGood(self)
```

True if comes from a curve with good reduction

```
def IsSemistable(self)
```

True if comes from a curve with semistable reduction (all (principal) components of an mrnc model have multiplicity 1)

```
def IsSemistableTotallyToric(self)
```

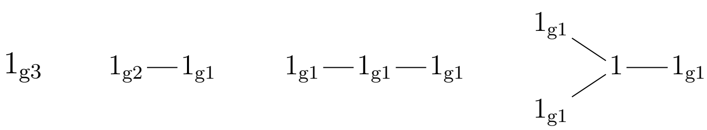
True if comes from a curve with semistable totally toric reduction (semistable with no positive genus components)

```
def IsSemistableTotallyAbelian(self)
```

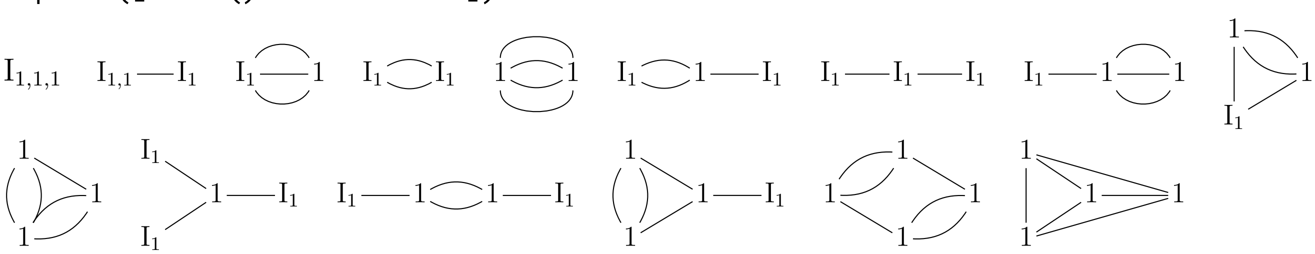
True if comes from a curve with semistable totally abelian reduction (semistable with no loops in the dual graph)

**Example** (Semistable reduction types).

```
> semi = ReductionTypes(3, semistable=True)                 # genus 3, semistable,
> ab = [R for R in semi if R.IsSemistableTotallyAbelian()]  #   totally abelian reduction
> print([R.TeX() for R in ab])
```



```
> tor = [R for R in semi if R.IsSemistableTotallyToric()]
> print([R.TeX() for R in tor])
```



Count semistable reduction types in genus 2,3,4,5 (OEIS A174224)

```
> print([ReductionTypes(n, semistable=True, countonly=True) for n in [2,3,4,5]])
[7, 42, 379, 4555]
```

```
def TamagawaNumber(self)
```

Tamagawa number of the curve with a given reduction type, over an algebraically closed residue field

**Example** (Tamagawa numbers for reduction types of elliptic curves).

```
> for R in ReductionTypes(1, elliptic=True): print(R, R.TamagawaNumber())
1g1 1
I1 1
I0* 4
I1* 4
IV 3
IV* 3
III 2
III* 2
II 1
```

```
II* 1
```

## 10.16 Invariants of individual principal components and chains

```
def PrincipalTypes(self)
```

Principal types (vertices) of the reduction type

```
def __len__(self)
```

Number of principal types in reduction type

```
def __getitem__(self, i)
```

Principal type number i in the reduction type, accessed as R[i] (numbered from i=1)

```
def LinkChains(self)
```

Return all the link chains in the reduction type

```
def LooseChains(self) -> list
```

Return all the link chains in R between different principal components, sorted as in label.

```
def Multiplicities(self)
```

Sequence of multiplicities of principal types

```
def Genera(self)
```

Sequence of geometric genera of principal types

```
def GCD(self)
```

GCD detecting non-primitive types

```
def Shape(self)
```

The shape of the reduction type.

**Example** (Principal types and chains). Take a reduction type that consists of smooth curves of genus 3, 2 and 1, connected with two chains of $\mathbb{P}^1$s of depth 2.

```
> R = ReductionType("1g3-(2)1g2-(2)1g1")
> print(R.TeX())
```

$1_{g3} \frac{}{\;2\;} 1_{g2} \frac{}{\;2\;} 1_{g1}$

This is how we access the three principal types, their primary invariants, and the chains.

```
> print(R[1], R[2], R[3])   # individual principal types, same as R.PrincipalTypes()
1g3-{1} 1g2-{1}-{1} 1g1-{1}
> print(R.Genera())         # geometric genus g of each principal type
[3, 2, 1]
> print(R.Multiplicities()) # multiplicity m of each principal type
[1, 1, 1]
> print(R.LinkChains())     # all chains between them (including loops and D-links)
[[1] loose c1 1,1 -(2) c2 1,1, [2] loose c2 1,1 -(2) c3 1,1]
```

## 10.17 Comparison

```
def Weight(self) -> list[int]
```

Weight of a reduction type, used for comparison and sorting

**Example.**

```
> R1 = ReductionType("I1g1")
> print(R1.Weight())
[1, 0, -2, 1, -1, 0, 0, 1, 0, 1, 1, 1, 4, 73, 49, 103, 49]
> R2 = ReductionType("Dg1")
> print(R2.Weight())
[1, 0, -2, 2, -1, 0, 0, 0, 2, 1, 1, 3, 68, 103, 49]
> print(R1<R2)       # I1g1<Dg1 so it precedes it in tables
True
```

```
def __eq__(self, other)
```

Determines if two principal types are equal based on their weight.

```
def __lt__(self, other)
```

Compares two reduction types by their weight.

```
def __gt__(self, other)
```

Compares two reduction types by their weight.

```
def __le__(self, other)
```

Compares two reduction types by their weight.

```
def __ge__(self, other)
```

Compares two reduction types by their weight.

```
def Sort(seq)
```

Sorts a sequence of reduction types in ascending order based on their weight.

**Example** (Sorted reduction types in genus 1 and 2).

```
> L = ReductionTypes(1, elliptic=True)
> RedType.Sort(L)
> print(L)
[1g1, I1, I0*, I1*, IV, IV*, III, III*, II, II*]
> L = ReductionTypes(2)
> RedType.Sort(L)
> print(L)
[1g2, I1g1, I1,1, Dg1, [2]g1_D, 2^1,1,1,1,1,1, I0*_0, D_{2-2}, I0*_D, I1*_0, [2]_1,D,
  I1*_D, [2]_D,D,D, 3^1,1,2,2, IV_0, IV*_-1, 4^1,3,2,2, III_0, III*_-1, III_D, 4^1,3_D,
  III*_D, [2]I0*_D, [2]I1*_D, 5^1,1,3, 5^1,2,2, 5^2,4,4, 5^3,3,4, 6^1,1,4, 6^5,5,2,
  6^2,4,3,3, II_D, [2]IV_D, [2]T_{6}D, [2]IV*_D, II*_D, 8^1,3,4, 8^5,7,4, [2]III_D,
  [2]III*_D, 10^1,4,5, 10^3,2,5, 10^7,8,5, 10^9,6,5, [2]II_D, [2]II*_D, 1g1-1g1, 1g1-I1,
  1g1-I0*, 1g1-I1*, 1g1-IV, 1g1-IV*, 1g1-III, 1g1-III*, 1g1-II, 1g1-II*, I1-I1, I1-I0*,
  I1-I1*, I1-IV, I1-IV*, I1-III, I1-III*, I1-II, I1-II*, I0*-I0*, I0*-I1*, I0*-IV,
  I0*-IV*, I0*-III, I0*-III*, I0*-II, I0*-II*, I1*-I1*, I1*-IV, I1*-IV*, I1*-III,
  I1*-III*, I1*-II, I1*-II*, IV-IV, IV-IV*, IV-III, IV-III*, IV-II, IV-II*, IV*-IV*,
  IV*-III, IV*-III*, IV*-II, IV*-II*, III-III, III-III*, III-II, III-II*, III*-III*,
  III*-II, III*-II*, II-II, II-II*, II*-II*, T=T, D-=D, 1---1]
```

## 10.18 Reduction types, labels, and dual graphs

```
def DualGraph(self, compnames="default")
```

```
Full dual graph from a reduction type, possibly with variable length edges, and optional names of
components.
Returns: GrphDual   The constructed dual graph.
```

```
def TeXLabel(self, forcesubs=False, forcesups=False, wrap=True)
```

```
TeX label of a reduction type used with the \redtype macro
```

```
def Label(self, tex=False, html=False, wrap=True, forcesubs=False,
  forcesups=False, depths="default")
```

```
Return canonical string label of a reduction type.
  tex=True        gives a TeX-friendly label (\redtype{...})
  html=True       gives a HTML-friendly label (<span class='redtype'>...</span>)
  wrap=False      keeps the format above but removes \redtype / <span> wrapping
  forcesubs=True  forces depths of chains & loops to be always printed (usually in round brackets)
  forcesups=True  forces outgoing chain multiplicities to be always printed (in curly brackets).
depths can be "default", "original", "minimal", or a custom sequence.
```

```
def Family(self) -> str
```

```
Returns the reduction type label with minimal chain lengths in the same family.
```

**Example** (Plain and TeX labels for reduction types).

```
> R = ReductionType("IIg1_1-(3)III-(4)IV")
> print(R.Label())            # plain text label
IIg1_1-(3)III-(4)IV
> R2 = ReductionType(R.Label())
> assert R == R2              # can be used to reconstruct the type
> print(R.Family())           # family (reduction type with minimal depths)
IIg1_1-III-IV
> print(R.Label(tex=True))    # label in TeX, wrapped in \redtype{...} macro
```

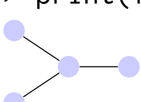
$\mathrm{II}_{g1,1}\,{}_{\overline{3}}\mathrm{III}_{\overline{4}}\mathrm{IV}$

```
> print(R[1])                 # first principal type as a standalone type
IIg1_1-{1}
> print(R.TeX())              # reduction type as a graph in TeX
```

$\mathrm{II}_{g1,1}\,\frac{}{3}\,\mathrm{III}\frac{\quad}{4}\mathrm{IV}$

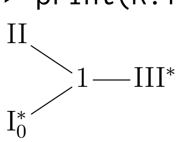
**Example** (Canonical label in detail). Take a graph $G$ on 4 vertices

```
> G = Graph(4,[[1,2],[1,3],[1,4]])
> print(TeXGraph(G, labels="none"))
```



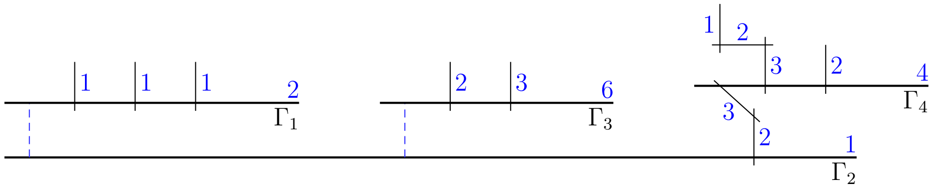
Place a component of multiplicity 1 at the root and II, III*, $\mathrm{I}_0^*$ at the three leaves. Link each leaf to the root with a chain of multiplicity 1. This gives a reduction type that occurs for genus 3 curves:

```
> R = ReductionType("1-II&c1-III*&c1-I0*")    # First component is the root,
> print(R.TeX())                               #   the other three are leaves
```




Here is the corresponding special fibre

```
> print(TeXDualGraph(R))
```




How is the following canonical label chosen among all possible labels?

```
> print(R)
I0*-1-II&III*-c_2
```

Each principal component is a principal type (as there are no loops or D-links), and its primary invariants are its Euler characteristic $\chi$ and a multiset lgcd of gcd's of outgoing (loose) link chains

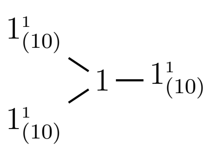
```
> print([S for S in R])
[I0*-{1}, 1-{1}-{1}-{1}, II-{1}, III*-{3}]
> print([S.Chi() for S in R])    # add up to 2-2*genus, so genus=3
[-1, -1, -1, -1]
> print([S.LGCD() for S in R])
[[1], [1, 1, 1], [1], [1]]
```

All four leaves have $\chi = -2$, lgcd=[1] and the root $\chi = 1$, lgcd=[1, 1, 1]

```
> print(PrincipalTypes(-1,[1]))        # 10 such (II-, III-, IV-, ...) drawn $1^1_{(10)}$
[1g1-{1}, I1-{1}, I0*-{1}, I1*-{1}, IV-{1}, IV*-{2}, III-{1}, III*-{3}, II-{1}, II*-{5}]
> print(PrincipalTypes(-1,[1,1,1]))    # unique one of this type, drawn as 1
[1-{1}-{1}-{1}]
```

Together they form a shape graph $S$ as follows:

```
> S = R.Shape()
> print(S.TeX(scale = 1))
```



The vertices and edges of $S$ are assigned weights. Vertex weights are $\chi$'s, edge weights are lgcd's

```
> print([GetLabel(S.Graph(),v) for v in S.Vertices()])
[[-1], [-1], [-1], [-1]]
> print([GetLabel(S.Graph(),e) for e in S.Edges()])
[[1], [1], [1]]
```

Then the shortest path is found using MinimumWeightPaths. It is v-v-v&v-2 (v=new vertex with $\chi = -1$, -=edge, &=jump). Note that by convention actual edges are preferred to jumps, and going to a new vertex preferred to revisiting an old one. Also vertices with smaller $\chi$ come first, if possible, as they have smaller labels.

v-v-v&v-2 $<$ v-v&v-2-v (jumps are larger than edge marks)
v-v-v&v-2 $<$ v-v-v&2-v (repeated vertex indices are larger than vertex marks)

```
> P,T = MinimumWeightPaths(S)
> print(P)      # v-v-v&v-2
[(0, [-1], False), (0, [-1], False), (0, [-1], True), (0, [-1], False), (2, [-1], True)]
```

This path can be used to construct the graph, and determines it up to isomorphism. There are $|\operatorname{Aut} S| = 6$ ways to trail $S$ in accordance with this path, and as far the shape is concerned, they are completely identical.

```
> print(T)
[[1, 2, 3, 4, 2], [1, 2, 4, 3, 2], [3, 2, 4, 1, 2], [3, 2, 1, 4, 2], [4, 2, 3, 1, 2], [4,
  2, 1, 3, 2]]
```

This gives six possible labels for our reduction type that all traverse the shape according to path $P$:

```
> l = lambda i: R[i].Label()
> print([f"{l(c[0])}-{l(c[1])}-{l(c[2])}&{l(c[3])}-c2" for c in T])
['I0*-1-II&III*-c2', 'I0*-1-III*&II-c2', 'II-1-III*&I0*-c2', 'II-1-I0*&III*-c2',
  'III*-1-II&I0*-c2', 'III*-1-I0*&II-c2']
```

Now we assign weights to vertices and edges that characterise the actual shape components (rather than just their $\chi$) and link chains (rather than just their lgcd)

```
> print([S.Weight() for S in R])
[[-1, 2, 0, 1, 0, 0, 3, 1, 1, 1, 1], [-1, 1, 0, 3, 0, 0, 0, 1, 1, 1], [-1, 6, 0, 1, 0, 0,
  2, 2, 3, 1], [-1, 4, 0, 1, 0, 0, 2, 3, 2, 3]]
> print(R.EdgesWeight(2,1))    # weight of the 1-II link chain
[1, 1, 0]
> print(R.EdgesWeight(2,3))    # weight of the 1-I0* link chain
[1, 1, 0]
> print(R.EdgesWeight(2,4))    # weight of the 1-III* link chain
[1, 3, 0]
```

The component weight R[i].Weight() starts with $(\chi, -m, -g, ...)$ so when all components have the same $\chi$ like in this example, the ones with large multiplicity $m$ have smaller weight. Because m(II)=6, m(III*)=4, m(I0*)=2, the trails $T[1]$ and $T[2]$ are preferred to the other four. They both start with a component II, then an edge II-1 and a component 1. After that they differ in that $T[1]$ traverses an edge 1-I0* and $T[2]$ an edge 1-III*. Because the edge weight is smaller for $T[1]$, this is the minimal path, and it determines the label for $R$:

```
> print(R)
I0*-1-II&III*-c_2
```

**Example** (Labels of individual principal types).

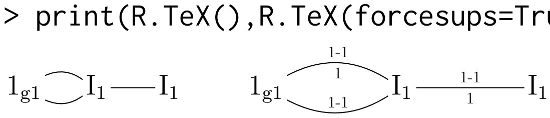
```
> R = ReductionType("II-III-IV")
> print([S.Label() for S in R])                    # As part of R
['IV', 'III', 'II']
> print([S.Label(loose=True) for S in R])          # As standalone principal types
['IV-{1}', 'III-{1}-{1}', 'II-{1}']
```

```
def TeX(self, forcesups=False, forcesubs=False, scale=0.8, xscale=1, yscale=1,
  oneline=False)
```

```
TikZ representation of a reduction type, as a graph with PrincipalTypes (principal components with
chi>0) as vertices, and edges for link chains.
oneline:=true removes line breaks.
forcesups:=true and/or forcesubs:=true shows edge decorations (outgoing multiplicities and/or chain
depths) even when they are default.
```

**Example** (TeX for reduction types).

```
> R = ReductionType("1g1--I1-I1")
> print(R.TeX(),R.TeX(forcesups=True, forcesubs=True, scale=1.5))
```



**Example** (Degenerations of two elliptic curves meeting at a point).

```
> S=ReductionType("1g1-1g1").Shape()    # Two elliptic curves meeting at a point (genus 2)
```

The corresponding shape is a graph v-v with two vertices with $\chi = -1$ and one edge of gcd 1

```
> print(S.TeX())
```

$1^{1}_{(10)}$ —— $1^{1}_{(10)}$

```
> print(PrincipalTypes(-1,[1]))              # There are 10 possibilities for such
[1g1-{1}, I1-{1}, I0*-{1}, I1*-{1}, IV-{1}, IV*-{2}, III-{1}, III*-{3}, II-{1}, II*-{5}]
>                                            #   a vertex, one for each Kodaira type
> print(ReductionTypes(S, countonly=True))  # and Binomial(10,2) such types in total
55
> print(ReductionTypes(S)[:10])              # first 10 of these
[1g1-1g1, 1g1-I1, 1g1-I0*, 1g1-I1*, 1g1-IV, 1g1-IV*, 1g1-III, 1g1-III*, 1g1-II, 1g1-II*]
```

## 10.19 Variable depths in Label

```
def SetDepths(self, depth)
```

```
Set depths for DualGraph and Label based on either a function or a sequence.
If `depth` is a function, it should be of the form:
  depth(e: RedChain) -> int/str
For example:
  lambda e: e.depth  # Original depths
  lambda e: MinimalLinkDepth(e.mi, e.di, e.mj, e.dj)  # Minimal depths
  lambda e: f"n_{e.index}"  # Custom string-based depth

If `depth` is a sequence, its length must match the number of link chains in the reduction type.

Raises:
  ValueError: If `depth` is neither a function nor a sequence or if the sequence length doesn't
  match.
```

```
def SetVariableDepths(self)
```

```
Set depths for DualGraph and Label to a variable depth format like 'n_i'.
```

```
def SetOriginalDepths(self)
```

```
Remove custom depths and reset to original depths for printing in Label and other functions.
```

```
def SetMinimalDepths(self)
```

```
Set depths to minimal ones in the family for each edge.
```

```
def GetDepths(self)
```

```
Return the current depths (string sequence) set by SetDepths or the original ones if not changed.
Returns:
  list: A list of depth strings for each link chain.
```

**Example** (Setting variable depths for drawing families).

```
> R = ReductionType("I3-(2)I5")
> print(R.Label(tex=True))
```

$\mathrm{I}_3\frac{}{2}\mathrm{I}_5$

```
> R.SetDepths(["a", "b", "5"])    # Make two of the three chains variable depth
> print(R.Label(tex=True))
```

$\mathrm{I}a\frac{}{b}\mathrm{I}_5$

```
> R.SetOriginalDepths()
> print(R.Label(tex=True))
```

$\mathrm{I}_3 \underset{2}{-} \mathrm{I}_5$

# 11 Reduction types in JavaScript (redtype.js)

The library redtype.js implements the combinatorics of reduction types, in particular

- Arithmetic of open and link sequences that controls the shapes of chains of $\mathbb{P}^1$s in special fibres of minimal regular normal crossing models,
- Methods for reduction types (RedType), their cores (RedCore), link chains (RedChain) and shapes (RedShape),
- Canonical labels for reduction types,
- Reduction types and their labels in TeX,
- Conversion between dual graphs, reduction type, and their labels:

$$\{\text{dual graphs}\} \rightleftarrows \{\text{reduction types}\} \rightleftarrows \{\text{labels}\}.$$

**Example** (Reduction types, labels and dual graphs).

```
> var R = ReductionType("I2*-1g1-IV");
> console.log(R.Label());             // Canonical plain label
I2*-1g1-IV
> console.log(R.Label({tex: true}));  // TeX label
```

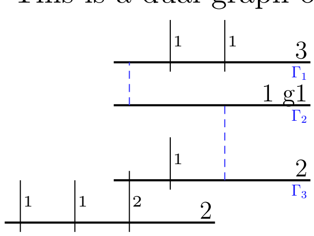
$\mathrm{I}*_2 .1_{\mathrm{g}1}.\mathrm{IV}$

```
> console.log(R.TeX());               // Reduction type as a graph
```

$\mathrm{I}_2^* — 1_{\mathrm{g}1} — \mathrm{IV}$

```
> console.log(R.DualGraph())         // Associated dual graph
DualGraph([2,1,3,1,1,1,2,1,1,2], [0,1,0,0,0,0,0,0,0,0],
  [[1,2],[1,4],[1,10],[2,3],[3,5],[3,6],[7,8],[7,9],[7,10]])
```

This is a dual graph on 10 components, of multiplicity 1, 2 and 3, and genus 0 and 1:




Taking the associated reduction type gives back R:

```
> var G = DualGraph([3,1,2,1,1,1,2,1,1,2], [0,1,0,0,0,0,0,0,0,0],
  [[1,2],[1,4],[1,5],[2,3],[3,6],[3,10],[7,8],[7,9],[7,10]]);
> console.log(G.ReductionType().Label());
I2*-1g1-IV
```

## 11.1 Open and link chains

A reduction type is a graph that has principal types as vertices (like IV, 1g1, $I_2^*$ above) and link chains as edges. Principal types encode principal components together with open chains, loops and D-links. The three functions that control multiplicities of open and link chains, and their depths are as follows:

```
function OpenSequence(m, d, includem = true)
```

**Example** (OpenSequence).

```
> console.log(OpenSequence(6, 5));
[ 6, 5, 4, 3, 2, 1 ]
> console.log(OpenSequence(13, 8));
[ 13, 8, 3, 1 ]
```

```
function LinkSequence(m1, d1, m2, dk, n, includem = true)
```

```
Unique link sequence of type m1(d1-dk-n)m2, that is of the form [m1,d1,...,dk,m2] with n+1 terms
equal to gcd(m1,d1)=gcd(m2,dk) and satisfying the chain condition: for every three consecutive terms
   d_(i-1), d_i, d_(i+1)
we have
   d_(i-1) + d_(i+1) = d_i * (integer > 1).
If includem = false, exclude the endpoints m1,m2 from the sequence.
```

**Example** (LinkSequence).

```
> console.log(LinkSequence(3, 2, 3, 2, -1));
[ 3, 2, 3 ]
> console.log(LinkSequence(3, 2, 3, 2, 0));
[ 3, 2, 1, 2, 3 ]
> console.log(LinkSequence(3, 2, 3, 2, 1));
[ 3, 2, 1, 1, 2, 3 ]
```

```
function MinimalLinkDepth(m1, d1, m2, dk)
```

```
Minimal depth of a link sequence between principal components of multiplicities m1 and m2 with
initial links d1 and dk.
Minimal depth of a chain d1,d2,...,dk of P1s between principal component of multiplicity m1, m2 and
initial link multiplicities d1,dk. The depth is defined as -1 + number of times GCD(d1,...,dk)
appears in the sequence.
For example, 5,4,3,2,1 is a valid link sequence, and MinimalLinkDepth(5,4,1,2) = -1 + 1 = 0.
```

**Example**. Example for MinimalLinkDepth from the description of the function:

```
> console.log(MinimalLinkDepth(5,4,1,2))
0
```

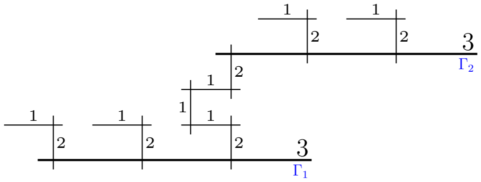
For another example, the minimal $n$ in the Kodaira type $I_n^*$ is 1. Here the chain links two components of multiplicity 2, and the initial multiplicities are 2 on both sides as well:

```
> console.log(MinimalLinkDepth(2,2,2,2))
1
```

Here is an example of a reduction type with a link chain between two components of multiplicity 3 and outgoing multiplicities 2 on both sides:

```
> var R = ReductionType("IV*-(2)IV*")
```

Here is what its dual graph looks like:



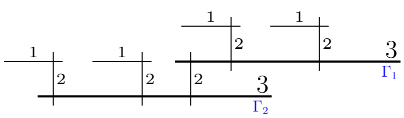
The link chain has gcd=GCD(3,2)=1 and

$$\text{depth} = -1 + \#1\text{'s}(=\text{gcd}) \text{ in the sequence } 3,2,1,1,1,2,3 = 2$$

This is the depth specified in round brackets in IV*-(2)IV*

```
> console.log(MinimalLinkDepth(3,2,3,2))   // Minimal possible depth for such a chain = -1
-1
> var R1 = ReductionType("IV*-IV*") // used by default when no expicit depth is specified
> var R2 = ReductionType("IV*-(-1)IV*")
> console.assert(R1.equals(R2))
Assertion failed
```

Here is what its dual graph looks like:



The next two functions are used in Label to determine the ordering of chains (including loops and D-links), and default multiplicities which are not printed in labels.

```
function SortLinks(m, O)
```

```
Sort a multiset of multiplicities O by GCD with m, then by O. This is how open and free
multiplicities are sorted in reduction types.
```

**Example** (Ordering open multiplicities in reduction types).

```
> var m = 6                      // principal component multiplicity
> var O = [1,2,3,3,4,5]          // initial multiplicities for outgoing open chains
> SortLinks(6, O)                // sort them first by gcd(o,m), then by o mod m
> console.log(O)
[ 1, 5, 2, 4, 3, 3 ]
```

```
function DefaultMultiplicities(m1, o1, m2, o2, loop)
```

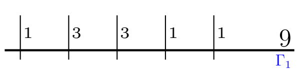
```
Intrinsic
function DefaultMultiplicities(m1, o1, m2, o2, loop)
Default edge multiplicities d1, d2 for a component with multiplicity m1, available outgoing
multiplicities o1, and one with m2, o2.
loop: boolean specifies whether it is a loop or a link between two different principal components.
```

**Example** (DefaultMultiplicities). Let us illustrate what happens when we take a principal component $9^{1,1,1,3,3}$ and add five default loops of depth 2,2,1,2,3, to get a reduction type $9^{1,1,1,3,3}_{2,2,1,2,3}$. How do default loops decide which initial multiplicities to take?

We start with a component of multiplicity $m = 9$ and open multiplicities $\mathcal{O} = \{1,1,1,3,3\}$.

```
> var R = ReductionType("9^1,1,1,3,3");
```

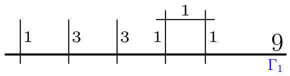
This is what its dual graph looks like:



We can add a loop to it linking two 1's of depth 2 by

```
> R = ReductionType("9^1,1,1,3,3_{1-1}2");
```

This is what its dual graph looks like:



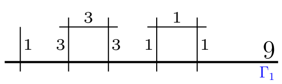

In this case, {1-1} does not need to be specified because this is the minimal pair of possible multiplicities in $\mathcal{O}$, as sorted by SortLinks:

```
> console.log(DefaultMultiplicities(9,[1,1,1,3,3],9,[1,1,1,3,3],true));
[ 1, 1 ]
> console.assert(R.equals(ReductionType("9^1,1,1,3,3_2")));
Assertion failed
```

After adding the loop, $\{1,3,3\}$ are left as potential outgoing multiplicities, so the next default loop links 3 and 3. Note that $1,3$ is not a valid pair because $\gcd(1,9) \neq \gcd(3,9)$.

```
> console.log(DefaultMultiplicities(9,[1,3,3],9,[1,3,3],true));
[ 3, 3 ]
> var R2 = ReductionType("9^1,1,1,3,3_2,2");       // 2 loops, use 1-1 and 3-3
```

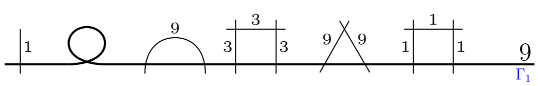
This is what its dual graph looks like:




There are no pairs left, so the next three loops use $(m,m) = (9,9)$

```
> console.log(DefaultMultiplicities(9,[1],9,[1],true));
[ 9, 9 ]
> var R3 = ReductionType("9^1,1,1,3,3_2,2,1,2,3");
> var R4 = ReductionType("9^1,1,1,3,3_{1-1}2,{3-3}2,{9-9}1,{9-9}2,{9-9}3");
> console.assert(R3.equals(R4));
Assertion failed
```

This is what its dual graph looks like:




## 11.2 Principal component core (RedCore)

A core is a pair $(m, O)$ with 'principal multiplicity' $m \geq 1$ and 'outgoing multiplicities' $O = \{o_1, o_2, ...\}$ that add up to a multiple of $m$, and such that $\gcd(m, O) = 1$. It is implemented as the following type:

```
function Core(m,O)
```

```
Core of a principal component defined by multiplicity m and list O.
```

**Example** (Create and print a principal component core $(m, O)$).

```
> console.log(Core(8,[1,3,4]).toString());   // Typical core; note 1+3+4=0 mod m=8
8^1,3,4
> console.log(Core(8,[9,3,4]).toString());   // Same core, as they are in Z/mZ
8^1,3,4
```

This is how cores are printed, with the exception of 7 cores of $\chi = 0$ (see below) that come from Kodaira types and two additional special ones D and T:

```
> console.log(Core(6,[1,2,3]).toString());                 // from a Kodaira type
II
> console.log([Core(2,[1,1]),Core(3,[1,2])].join(', '));   // two special ones
```

D, T

## 11.3 Basic invariants and printing

```
class RedCore
```

```
RedCore.definition()
```

Returns a string representation of a core in the form 'Core(m,O)'.

```
RedCore.Multiplicity()
```

Returns the principal multiplicity m of the principal component.

```
RedCore.Multiplicities()
```

Returns the list of outgoing chain multiplicities O, sorted with SortLinks.

```
RedCore.Chi()
```

Euler characteristic of a reduction type core (m,O), chi = m(2-|O|) + sum_(o in O) gcd(o,m)

```
RedCore.Label(tex = false)
```

Label of a reduction type core, for printing (or TeX if tex=True)

```
RedCore.TeX()
```

Returns the core label in TeX, same as Label with TeX=true.

**Example** (Core labels and invariants).

```
> let C=Core(2,[1,1,1,1])
> console.log(C.Label());                // Plain label
I0*
> console.log(C.TeX());                  // TeX label
```

$I_0^*$

```
> console.log(C.definition());           // How it can be defined
Core(2,1,1,1,1)
> console.log(C.Multiplicity());         // Principal multiplicity m
2
> console.log(C.Multiplicities());       // Outgoing multiplicities O
[ 1, 1, 1, 1 ]
> console.log(C.Chi());                  // Euler characteristic
0
```

```
function Cores(chi, {mbound="all", sort=true} = {})
```

Returns all cores (m,O) with given Euler characteristic chi<=2. When chi=2 there are infinitely many, so a bound on m must be given.

**Example** (Cores).

```
> let C = Cores(-2, {mbound: 4})
> console.log(C.join(', '))
2^1,1,1,1,1,1, 3^1,1,2,2, 4^1,3,2,2
> C = Cores(0)
> console.log(C.join(', '))
I0*, IV, IV*, III, III*, II, II*
> console.log([0,-2,-4,-6,-8].map(i=>Cores(i).length));  // [7, 16, 43, 65, 64, ...]
```

```
[ 7, 16, 43, 65, 64 ]
```

## 11.4 Link chains (RedChain)

Link chains between principal components fall into three classes: loops on a principal type, D-link on a principal type, and chains between principal types that link two of their loose edge endpoints. All of these are implemented in the class RedChain that carries class=cLoop, cD or cLoose, and keeps track of all the invariants.

**Example** (Link chains, with no principal types specified).

```
> console.log(Link(cLoop, 2, 1, 2, 1).toString());  // Loop
loop 2,1 -(0) 2,1
> console.log(Link(cD, 2, 2).toString());           // D-Link
D-link 2,2 -(1) 2,2
> console.log(Link(cLoose, 2, 2).toString());       // to another principal type
loose 2,2 -(false) false,false
```

## 11.5 Invariants and depth

```
class RedChain
```

```
RedChain.GCD()
```

GCD of all elements in the chain (=GCD(mi,di)=GCD(mj,di))

```
RedChain.Index()
```

Index of the RedChain, used for distinguishing between chains

```
RedChain.DepthString()
```

Return the string representation of the RedChain's depth

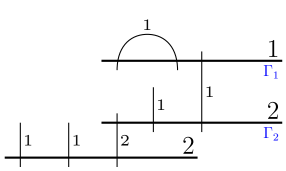
```
RedChain.SetDepthString(depth)
```

Set how the depth is printed (e.g., "1" or "n")

**Example** (Invariants of link chains). Take a genus 2 reduction type $\mathrm{I}_2 \frac{}{1} \mathrm{I}{*}_2$ whose special fibre consists of Kodaira types $\mathrm{I}_2$ (loop of $\mathbb{P}^1$s) and $\mathrm{I}_2^*$ linked by a chain of $\mathbb{P}^1$s of multiplicity 1.

```
> var R = new ReductionType("I2-(1)I2*");
```

This is what its dual graph looks like:

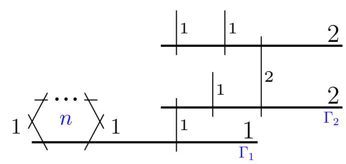



There are two principal types R[1]=$\mathrm{I}_2$ and R[2]=$\mathrm{I}_2^*$, with a loop on I[1] (class cLoop=1), a link chain between them (class cLoose=3), and a D-link on I[2] (class cD=2) This is the order in which they are printed in the label.

```
> console.log([R[1],R[2]].join(' '));    // two principal types R[1] and R[2]
I2-{1} I2*-{1}
> var [c1,c2,c3] = R.LinkChains();
> console.log(c1.toString());
[1] loop c1 1,1 -(2) c1 1,1
```

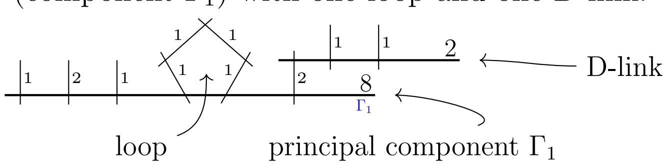
```
> console.log(c2.toString());
[2] loose c1 1,1 -(1) c2 2,1
> console.log(c3.toString());
[3] D-link c2 2,2 -(2) 2,2
> console.log(c3.Class);                    // cLoop=1, *cD=2*, cLoose=3
2
> console.log(c3.GCD());                    // GCD of the chain multiplicities [2,2,2]
2
> console.log(c3.Index());                  // index in the reduction type
3
> c3.SetDepthString("n");                   // change how its depth is printed in labels
> console.log(c3.toString());              //   and drawn in dual graphs of reduction types
[3] D-link c2 2,2 -(n) 2,2
> console.log(R.Label());
I2-(1)In*
```

This is what its dual graph looks like:



## 11.6 Principal components (RedPrin)

The classification of special fibre of mrnc models is based on principal types. For curves of genus $\geq 2$ such a type is a principal component with $\chi < 0$, together with its open chains, loops, chains to principal component with $\chi = 0$ (called D-links) and a tally of link chains to other principal components with $\chi < 0$, called loose links. For example, the following reduction type has only principal type (component $\Gamma_1$) with one loop and one D-link:




A principal type is implemented as the following javascript class.

```
function PrincipalType(m, g, O, Lloops, LD, Lloose, index = 0)
```

```
Create a new principal type from its primary invariants:
m         multiplicity of the principal component, e.g. 8
g         geometric genus of the principal component, e.g. 0
O         outgoing multiplicities for open chains, e.g. 1,1,2
Lloops    list of loops [[di,dj,depth],...], e.g. [[1,1,3]]
LD        list of D-links [[di,depth],...], e.g. [[2,1]]  (m and all d_i must be even)
Lloose    list of loose multiplicities, e.g. [8]
```

**Example** (Construction). We construct the principal type from example above. It has $m = 8$, $g = 0$, open multiplicities 1,1,2, loop $1 - 1$ of depth 3, a D-link with outgoing multiplicity 2 of depth 1, and no loose chains (so that it is a reduction type in itself).

```
> const S = PrincipalType(8,0,[1,1,2],[[1,1,3]],[[2,1]],[]);
```

```
class RedPrin
```

```
RedPrin.function order(e)
```

`RedPrin.Multiplicity()`

Principal multiplicity m of a principal type

`RedPrin.GeometricGenus()`

Geometric genus g of a principal type S = (m, g, O, ...)

`RedPrin.Index()`

Index of the principal component in a reduction type, 0 if freestanding

`RedPrin.Chains(Class = 0)`

Sequence of chains of type RedChain originating in S. By default, all (loops, D-links, loose) are returned,
unless a specific chain class is specified.

`RedPrin.OpenMultiplicities()`

Sequence of open multiplicities S.O of a principal type, sorted

`RedPrin.LinkMultiplicities()`

Sequence of link multiplicities S.L of a principal type, sorted as in label

`RedPrin.Loops()`

Sequence of chains in S representing loops (class cLoop)

`RedPrin.DLinks()`

Sequence of chains in S representing D-links (class cD)

`RedPrin.LooseChains()`

Sequence of loose chains of a principal type, sorted

`RedPrin.LooseMultiplicities()`

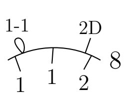
Sequence of loose multiplicities of a principal type, sorted

`RedPrin.definition()`

Returns a string representation of the principal type object in the form of the PrincipalType constructor.

**Example** (Invariants). We continue with the principal type above. It has $m = 8$, $g = 0$, open multiplicities 1,1,2, loop $1-1$ of depth 3, a D-link with outgoing multiplicity 2 of depth 1, and no loose chains (so that it is a reduction type in itself).

```
> const S = PrincipalType(8,0,[1,1,2],[[1,1,3]],[[2,1]],[]);
> console.log(S.toString());
8^1,1,1,1,2,2_3,1D
> console.log(S.TeX({standalone: true})); // How it appears in the tables
```




```
> console.log(S.Multiplicity());          // Principal component multiplicity
8
> console.log(S.GeometricGenus());        // Geometric genus of the principal component
0
> console.log(S.OpenMultiplicities());    // Open chain initial multiplicities O=[1,1,2]
```

```
[ 1, 1, 2 ]
> console.log(S.Loops().toString());      // Loops (of type RedChain)
loop c0 8,1 -(3) c0 8,1
> console.log(S.DLinks().toString());     // D-Links (of type RedChain)
D-link c0 8,2 -(1) 2,2
> console.log(S.LooseMultiplicities());   // Loose link multiplicities
[]
> console.log(S.LinkMultiplicities());    // All initial link multiplicities
[ 1, 1, 2 ]
> console.log(S.definition());            // evaluatable string to reconstruct S
PrincipalType(8,0,[1,1,2],[[1,1,3]],[[2,1]],[])
```

```
RedPrin.GCD()
```

Return GCD(m, O, L) for a principal type

```
RedPrin.Core()
```

Core of a principal type - no genus, all non-zero link multiplicities put to 0, and gcd(m, O) = 1

```
RedPrin.Chi()
```

Euler characteristic chi of a principal type (m, g, O, Lloops, LD, Lloose).
chi = m(2-2g-|O|-|L|) + sum_(o in O) gcd(o, m), where L consists of all the link multiplicities in Lloops (2 from each), LD (1 from each), Lloose (1 from each).

```
RedPrin.LGCD()
```

Outgoing link pattern of a principal type = multiset of GCDs of loose edges with m.

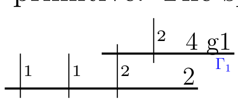
```
RedPrin.Copy(index = false)
```

Make a copy of a principal type.

**Example** (GCD). Define a principal type by its primary invariants: $m = 4$, $g = 1$, open multiplicities $\mathcal{O} = [2]$, no loops, one D-link with initial multiplicity 2 and length 1, and no loose links

```
> const S = PrincipalType(4, 1, [2], [], [[2, 1]], []);
> console.log(S.GCD());             // its GCD(m,O,L)=GCD(4,[2],[2])=2
2
> console.log(S.toString());        // which is seen as [2] in its name
[2]Dg1_1D
```

Note, however, it is not a multiple of 2 of another principal component type because its D-link is primitive. The special fibre is not a multiple of 2. This is what the special fibre looks like:




```
RedPrin.Weight()
```

Sequence [chi,m,-g,#loose,#Ds,#loops,#O,O,loops,Ds,loose] that determines the weight of a principal type, and characterises it uniquely.

```
RedPrin.equals(other)
```

Compare two principal types by their weight.

```
RedPrin.lessThan(other)
```

Compare two principal types by their weight.

```
RedPrin.lessThanOrEqual(other)
```

```
Compare two principal types by their weight.
```

```
RedPrin.greaterThan(other)
```

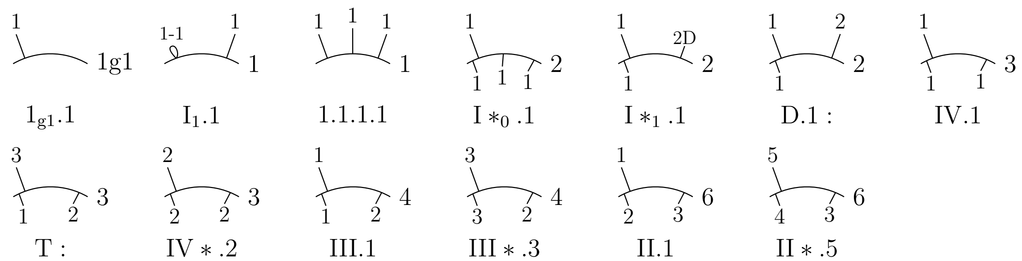
```
Compare two principal types by their weight.
```

```
RedPrin.greaterThanOrEqual(other)
```

```
Compare two principal types by their weight.
```

**Example** (TeX for principal components). Here are the 13 principal types with chi=-1 (10 Kodaira + 3 'exotic')

```
> const L = PrincipalTypes(-1);
> console.log(PrincipalTypesTeX(L, { label: true, width: 7, yshift: 2.2 }));
```




```
Label(options={})
```

```
Ascii Label or TeX label of a principal type.
Setting tex:=true prints the tex label, in \redtype{...} format by default, unless plain:=true.
Setting loose:=true prints outgoing loose edges as well (standalone principal type).
```

```
TeX(options = {})
```

```
TeX a principal type as a TikZ arc with outer and inner lines, loops, and Ds, with options:
  length     [="35pt"] determines arc length
  label      [=false]  if true puts its label underneath.
  standalone [=false]  if true wraps it in \tikz.
```

```
function PrincipalTypeFromWeight(w)
```

```
Create a principal type S from its weight sequence w (=Weight(S)).
```

**Example**.

```
> S = new PrincipalType(8,0,[4,2],[[1,1,1]],[[2,1]],[6]);  // Create a principal type
> var w = S.Weight(); // weight encodes chi, m, g etc.
> console.log(w);                                          //   and characterizes S
[ -26, 8, -0, 1, 1, 1, 2, 2, 4, 1, 1, 1, 2, 1, 6 ]
> console.log(PrincipalTypeFromWeight(w).definition()); // Reconstruct S from the weight
PrincipalType(8,0,[2,4],[[1,1,1]],[[2,1]],[6])
```

```
function PrincipalTypes(chi, arg, {semistable=false, sort=true, withlgcds=false}
  = {})
```

```
Principal types with a given Euler characteristic chi, and optional restrictions.
Returns (list of types, discovered GCDs of loose chains). Can be used as either:
  PrincipalTypes(chi)          - all
  PrincipalTypes(chi,C)        - with a given core C
  PrincipalTypes(chi,LGCDs)    - with a given sequence of loose chain lgcds
In all three cases can restrict to semistable types, setting semistable=True
```

**Example** (Printing principal types).

```
> let comps = PrincipalTypes(-1,[1]);
> console.log(comps.join(", "));
1g1-{1}, I1-{1}, I0*-{1}, I1*-{1}, IV-{1}, IV*-{2}, III-{1}, III*-{3}, II-{1}, II*-{5}
```

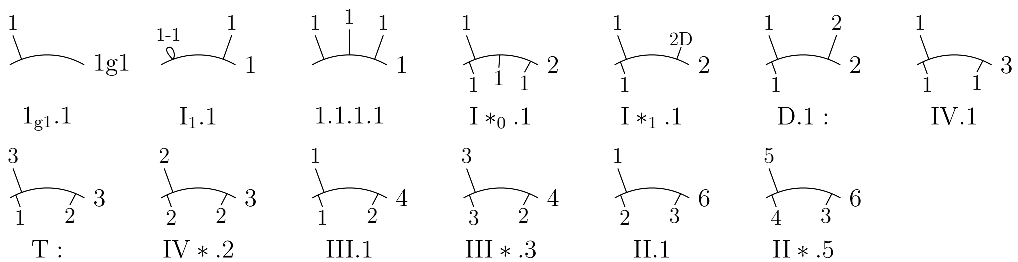
```
> comps = PrincipalTypes(-2,[1,1]);
> console.log(comps.join(", "));
1g1-{1}-{1}, I1-{1}-{1}, I0*-{1}-{1}, I1*-{1}-{1}, IV-{1}-{1}, IV*-{2}-{2}, III-{1}-{1},
  III*-{3}-{3}
> comps = PrincipalTypes(-2,[2]);
> console.log(comps.join(", "));
[2]g1=, I0*=, D_0=, [2]_1=, I1*=, [2]_D,D=, III-{2}, III*-{2}, [2]I0*-{2}, [2]I1*-{2},
  II-{2}, [2]IV-{2}, [2]IV*-{4}, II*-{4}, [2]III-{2}, [2]III*-{6}, [2]II-{2}, [2]II*-{10}
```

```
function PrincipalTypesTeX(T, options = {})
```

```
TeX a list of principal types T as a rectangular table in a TikZ picture, with options:
  label  [=false]     puts the principal type label underneath.
  sort   [=true]      sorts the types by Weight first, in increasing order.
  yshift [="default"] controls the y-axis shift after every row, based on label presence.
  width  [=10]        controls the number of principal types per row.
  length [="35pt"]    controls the length of each arc.
  scale  [=0.8]       controls the TikZ picture global scale.
```

**Example** (TeX for principal components). Take all 13 principal types with chi=-1 (10 Kodaira + 3 'exotic'), and draw them as a TeX table of width 7

```
> let L = PrincipalTypes(-1)
> console.log(PrincipalTypesTeX(L, {label: true, width: 7, yshift: 2.2}))
```




## 11.7 Basic labelled undirected graphs (Graph)

This section provides a basic implementation of labelled undirected graphs, offering core functionality for graph manipulation in javascript. It allows the user to construct graphs using a set of vertices and edges, and supports key operations such as adding and removing vertices and edges, checking for the existence of specific vertices or edges, and retrieving or modifying vertex labels.

Graph traversal and connectivity are handled through `BFS` (breadth-first search), `ConnectedComponents`, which is used later for connectivity testing, and `MinimumWeightPaths`. The latter is used to generate a canonical label for a vertex-labelled graph that can be used for isomorphism testing (`IsIsomorphic`).

The library also supports generating subgraphs from a subset of edges (`EdgeSubgraph`), and copying the entire graph (`Copy`).

Finally, we have visualization functions `TeXGraph` and `SVGGraph` to draw graphs in TikZ and HTML.

```
class Graph
```

```
Graph.constructor(vertexSet = [], edgeSet = [])
```

```
Initialize the graph with a set of vertices and edges. vertexSet can be an integer (number of
vertices) -> [1,2,3,...]
EdgesSet should be a list of edges e.g. [[1,2],[2,3],[3,4]] with vertices from vertexSet
```

```
Graph.AddVertex(vertex, label = undefined)
```

```
Add a vertex with an optional label. If the vertex already exists, update its label.
```

`Graph.AddEdge(vertex1, vertex2)`

Add an edge between two vertices (both vertices must exist).

`Graph.RemoveVertex(v)`

Remove a vertex v from the graph, together with its incident edges

`Graph.HasVertex(vertex)`

Check if a vertex exists in the graph.

`Graph.GetLabel(vertex)`

Get the label of a vertex. Returns undefined if the vertex doesn't exist.

`Graph.SetLabel(vertex, label)`

Set the label for a specific vertex. Raises an error if the vertex doesn't exist.

`Graph.GetLabels()`

Get all labels in the graph.

`Graph.SetLabels(labels)`

Set labels for all vertices. Raises an error if the number of labels doesn't match the number of vertices.

`Graph.RemoveLabels()`

Remove labels from graph vertices

`Graph.HasEdge(vertex1, vertex2)`

Check if an edge exists between two vertices. No loops are allowed.

`Graph.Vertices()`

Return the set of vertices as an array.

`Graph.Edges(v = undefined)`

If v is undefined, return all edges as an array of arrays of length 2.
If v is defined, check it is a vertex, and return all edges where v is one of the vertices.

`Graph.Neighbours(vertex)`

Get all neighbours of a given vertex. This returns an array of adjacent vertices, and loops contribute twice.

`Graph.BFS(startVertex)`

Perform BFS starting from the given vertex and return the connected component as an array.

`Graph.ConnectedComponents()`

Find all connected components in the graph using BFS. Return as an array of arrays of vertices.

`Graph.RemoveEdge(vertex1, vertex2)`

Remove an edge from the graph

`Graph.EdgeSubgraph(edgeSet)`

Returns a new Graph object containing only the specified edges

`Graph.Degree(vertex)`

Returns the degree of a vertex (number of indident edges)

```
Graph.Copy()
```

Copy a graph

```
Graph.Label(options = {})
```

Generate a graph label based on a minimum weight path, determines G up to isomorphism. The label is constructed by iterating through the minimum weight path and formatting the vertices and edges with labels, if present.

```
Graph.IsIsomorphic(other)
```

Test whether are two graphs are isomorphic, through their labels

**Example** (Graph usage).

```
> const graph = new Graph();
> graph.AddVertex(1, "A");
> graph.AddVertex(4, "B");
> graph.AddVertex(6, "C");
> graph.AddEdge(1, 4);
> graph.AddEdge(4, 6);
> console.log(graph.HasVertex(1));     // true
true
> console.log(graph.GetLabel(4));      // "B"
B
> console.log(graph.HasEdge(1, 4));    // true
true
> console.log(graph.HasEdge(1, 6));    // false
false
```

**Example** (Graph usage).

```
> const graph = new Graph();
> graph.AddVertex(1, "A");
> graph.AddVertex(4, "B");
> graph.AddVertex(6, "C");
>
> graph.AddEdge(1, 4);
> graph.AddEdge(4, 6);
>
> console.log(graph.Vertices());  // [1, 4, 6]
[ 1, 4, 6 ]
> console.log(graph.Edges());    // [[1, 4], [4, 6]]
[ [ 1, 4 ], [ 4, 6 ] ]
> console.log(graph.HasEdge(4, 1)); // true (order doesn't matter)
true
>
> const graph2 = new Graph([1,2,3],[[1,2],[2,3]]);     // Same graph defined differently
> graph2.SetLabels(["C","B","A"]);
>
> console.log(graph.IsIsomorphic(graph2));
true
```

**Example** (Connected components).

```
> const graph = new Graph([1, 2, 3, 4, 5], [[1, 2], [2, 3], [4, 5]]);
> const components = graph.ConnectedComponents();
> console.log(components);  // Example output: [[1, 2, 3], [4, 5]]
[ [ 1, 2, 3 ], [ 4, 5 ] ]
```

```
function MinimumWeightPaths(D)
```

```
Determines minimum weight paths in a connected labelled undirected graph, returning weights and
possible vertex index sequences.
Minimum weight paths for a labelled undirected graph (e.g. double graph underlying shape)
returns W=bestweight [<index, v_label, jump>,...] (characterizes D up to isomorphism)
  and I=list of possible vertex index sequences
For example for a rectangular loop G with all vertex chis=1 and edges as follows
  V:=[1,1,1,1]; E:=[[1,2,1],[2,3,1],[3,4,2],[1,4,1,1]]; S:=Shape(V,E);
the double graph D has 6 vertices and 6 edges in a loop, and here minimum weight W is
  W = [<0,[-1],false>,<0,[-1],false>,<0,[-1],false>,<0,[1,1],false>,<0,[-1],false>,
       <0,[2],false>,<1,[-1],true>]
The unique trail T[1] (generally Aut D-torsor) is D.3->D.2->D.1->...->D.3, encoded
  T = [[3,2,1,6,4,5,3]]
```

**Example** (A-B-C-c1).

```
> const G = new Graph();
> G.AddVertex(1);
> G.AddVertex(2);
> G.AddVertex(3);
> G.AddEdge(1, 2);
> G.AddEdge(2, 3);
> G.AddEdge(3, 1);
> G.SetLabels(["A", "A", "A"]);
> const [P, a] = MinimumWeightPaths(G, false);
> console.log("P:", P);
P: [ [ 0, "A", false ], [ 0, "A", false ], [ 0, "A", false ], [ 1, "A", true ] ]
> console.log("a:", a);
a: [ [ 1, 2, 3, 1 ], [ 1, 3, 2, 1 ], [ 2, 1, 3, 2 ], [ 2, 3, 1, 2 ], [ 3, 2, 1, 3 ], [ 3,
  1, 2, 3 ] ]
```

**Example** (MinimumWeightPaths).

```
> const G = new Graph();
> G.AddVertex(1, "C");
> G.AddVertex(2, "B");
> G.AddVertex(3, "C");
> G.AddVertex(4, "A");
> G.AddEdge(1, 2);
> G.AddEdge(2, 3);
> G.AddEdge(3, 4);
> G.AddEdge(4, 1);
> G.AddEdge(1, 3);
```

Calculate minimum weight paths

```
> const [P, a] = MinimumWeightPaths(G, false);
```

Print the minimal path

```
> console.log("P:", P);
P: [ [ 0, "C", false ], [ 0, "A", false ], [ 0, "C", false ], [ 0, "B", false ], [ 1, "C",
  false ], [ 3, "C", true ] ]
> console.log("a:", a);
```

```
a: [ [ 1, 4, 3, 2, 1, 3 ], [ 3, 4, 1, 2, 3, 1 ] ]
> console.log("G = ", G.Label());
G =  C-A-C-B-c1-c3
```

Example 2: Another graph on five vertices, not Eulerian

```
> const G2 = new Graph();
> G2.AddVertex(1, "A");
> G2.AddVertex(2, "B");
> G2.AddVertex(3, "A");
> G2.AddVertex(4, "A");
> G2.AddVertex(5, "C");
> G2.AddEdge(2, 1);
> G2.AddEdge(2, 3);
> G2.AddEdge(2, 4);
> G2.AddEdge(2, 5);
```

Calculate minimum weight paths

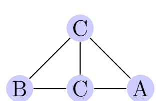
```
> const [P2, a2] = MinimumWeightPaths(G2, false);
```

Print the minimal path

```
> console.log("P2:", P2);
P2: [ [ 0, "A", false ], [ 0, "B", false ], [ 0, "A", true ], [ 0, "A", false ], [ 2, "B",
  false ], [ 0, "C", true ] ]
> console.log("a2:", a2);
a2: [ [ 1, 2, 3, 4, 2, 5 ], [ 1, 2, 4, 3, 2, 5 ], [ 3, 2, 1, 4, 2, 5 ], [ 3, 2, 4, 1, 2, 5
  ], [ 4, 2, 1, 3, 2, 5 ], [ 4, 2, 3, 1, 2, 5 ] ]
> console.log("G2 = ", G2.Label());
G2 =  A-B-A&A-c2-C
```

**Example** (Minimum weight paths).

```
> var G = new Graph(4, [[1, 2], [2, 3], [3, 4], [4, 1], [1, 3]]);
> G.SetLabels(["C", "B", "C", "A"]);
> console.log(TeXGraph(G));
```




Now we calculate minimum weight paths:

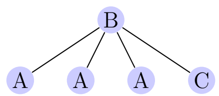
```
> let [P, a] = MinimumWeightPaths(G);
```

Print the minimal path and the trails, both from one odd degree vertex to the other one:

```
> console.log("P:", P);
P: [ [ 0, "C", false ], [ 0, "A", false ], [ 0, "C", false ], [ 0, "B", false ], [ 1, "C",
  false ], [ 3, "C", true ] ]
> console.log("a:", a);
a: [ [ 1, 4, 3, 2, 1, 3 ], [ 3, 4, 1, 2, 3, 1 ] ]
```

Here is another graph on five vertices, this time not Eulerian

```
> G = new Graph(5, [[2, 1], [2, 3], [2, 4], [2, 5]]);
> G.SetLabels(["A", "B", "A", "A", "C"]);
> console.log(TeXGraph(G));
```




Calculate minimum weight path, which is A-B-A, A-2-C (where 2 is 'second vertex on the path')

```
> [P, a] = MinimumWeightPaths(G);
```

Print the minimal path

```
> console.log("P:", P);
P: [ [ 0, "A", false ], [ 0, "B", false ], [ 0, "A", true ], [ 0, "A", false ], [ 2, "B",
  false ], [ 0, "C", true ] ]
```

There are 6 ways to trace this path, and they form an Aut(G)=S3-torsor. The first one is

```
> console.log(`One trail out of ${a.length} is ${a[0]}`);
One trail out of 6 is 1,2,3,4,2,5
```

```
function StandardGraphCoordinates(G)
```

```
Returns vertex coordinate lists x, y for planar drawing of a graph G
```

```
function TeXGraph(G, options = {})
```

```
Draw a graph in TikZ, preferably planar. Options:
  x = "default",           // X-coordinates for vertices
  y = "default",           // Y-coordinates for vertices
  labels = "default",      // Labels for vertices (sequence or "default")
  scale = 0.8,             // Global scale for the TikZ picture
  xscale = 1,              // Scale factor in x direction
  yscale = 1,              // Scale factor in y direction
  vertexlabel = "default", // Labeling function for vertices (or "default")
  edgelabel = "default",   // Labeling function for edges (or "default")
  vertexnodestyle = "default", // Style for vertices
  edgenodestyle = "default",   // Style for edge labels
  edgestyle = "default"        // Style for edges
```

```
function SVGGraph(G, options = {})
```

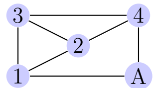
```
Draw a graph in SVG, preferably planar. Options:
  x = "default",            // x-coordinates for vertices
  y = "default",            // y-coordinates for vertices
  labels = "default",       // Labels for vertices (sequence or "default")
  scale = 0.8,              // Global scale for the TikZ picture
  xscale = 100,             // Scale factor in x direction
  yscale = 100,             // Scale factor in y direction
  innersep = (labels?1:3),  // Inner separation space for vertices in pixels
  nodeRadius = 10,          // Vertex radius
  padding    = 12,          // Vertex radius + eps for padding at the edges
Labels can be a sequence of strings (or None, or "default" -> 1, 2, 3) to draw vertices.
```

```
function GraphFromEdgesString(edgesString)
```

```
Construct a graph from a string encoding edges such as "1-2-3-4, A-B, C-D", assigning the vertex
labels to the corresponding strings.
```

**Example.**

```
> const G   = GraphFromEdgesString("1-2-3-4, 1-3, 2-4-A-1")
> console.log(G.Label())
1-2-3-4-A-c1-c3&c2-c4
> console.log(TeXGraph(G))
```




```
> const svg = SVGGraph(G)           // for use in HTML files
```

## 11.8 RedShape

A reduction type a graph whose vertices are principal types (type RedPrin) and edges are link chains. They fall naturally into 'shapes', where every vertex only remembers the Euler characteristic $\chi$ of the type, and edge the gcd of the chain. Thus, the problem of finding all reduction types in a given genus (see ReductionTypes) reduces to that of finding the possible shapes (see Shapes) and filling in shape components with given $\chi$ and gcds of loose edges (see PrincipalTypes).

```
class RedShape
```

```
RedShape.Graph()
```

Returns the underlying undirected graph G of the shape.

```
RedShape.DoubleGraph()
```

Returns the vertex-labelled double graph D of the shape.

```
RedShape.Vertices()
```

Returns the vertex set of G as a graph.

```
RedShape.Edges()
```

Returns the edges of G as a graph

```
RedShape.NumVertices()
```

Returns the number of vertices in the graph G underlying the shape.

```
RedShape.Chi(v)
```

Returns the Euler characteristic chi(v) <= 0 of the vertex v.

```
RedShape.LGCDs(v)
```

Returns the LGCDs of a vertex v that together with chi determine the vertex type (chi, lgcds).

```
RedShape.TotalChi()
```

Returns the total Euler characteristic of a graph shape chi <= 0, sum over chi's of vertices.

```
RedShape.VertexLabels()
```

Returns a sequence of -chi's for individual components of the shape S.

```
RedShape.EdgeLabels()
```

Returns a list of edges v_i -> v_j of the form [i, j, edgegcd].

```
RedShape.toString()
```

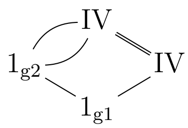
Print in the form Shape(V,E) so as to be evaluatable

```
RedShape.TeX(options = {})
```

Tikz a shape of a reduction graph, and, if required, the bounding box x1, y1, x2, y2.

**Example** (Graph, DoubleGraph and primary invariants for shapes). Under the hood of shapes of reduction types are their labelled graphs and associated 'double' graphs. As an example, take the following reduction type:

```
> const R = ReductionType("1g2--IV=IV-1g1-c1");
> console.log(R.TeX());
```

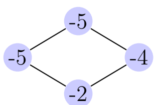



There are four principal types, and they become vertices of R.Shape() whose labels are their Euler characteristics $-5, -2, -4, -5$. The edges are labelled with GCDs of the link chain between the types. For example:

— the link chain 1g2-1g1 of gcd 1 becomes the label "1",
— the link chain IV=IV of gcd 3 becomes "3",
— the two chains 1g2–IV of gcd 1 become "1,1"

on the corresponding edges.

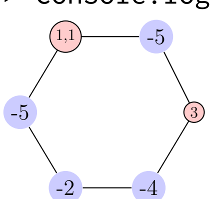
```
> const S = R.Shape();
> console.log(S.toString());
Shape([5,2,4,5],[1,2,1,1,4,1,1,2,3,1,3,4,3])
> console.log(TeXGraph(S.Graph()));
```




```
> console.log(S.Vertices());      // Indexed (from 1) set of vertices of S.Graph()
[ 1, 2, 3, 4 ]
> console.log(S.Edges());         //   and edges [ (from_vertex, to_vertex), ... ]
[ [ 1, 2 ], [ 1, 4 ], [ 2, 3 ], [ 3, 4 ] ]
> console.log(S.VertexLabels());  // [-chi] for each type
[ 5, 2, 4, 5 ]
> console.log(S.EdgeLabels());    // [ [from_vertex, to_vertex, gcd1, gcd2, ...], ...]
[ [ 1, 2, 1 ], [ 1, 4, 1, 1 ], [ 2, 3, 1 ], [ 3, 4, 3 ] ]
```

MinimumWeightPaths is implemented in python for graphs with labelled vertices but not edges. To use them for shapes, the underlying graphs are converted to graphs with only labelled vertices. This is done simply by introducing a new vertex on every edge which carries the corresponding edge label. For compactness, if the label is "1" (most common case), we don't introduce the vertex at all. This is called the double graph of the shape:

```
> const blue = "circle,scale=0.7,inner sep=2pt,fill=blue!20";       // former vertices
> const red = "circle,draw,scale=0.5,inner sep=2pt, fill=red!20";  // former edges
> const D = S.DoubleGraph();
> const bluered = v => (D.GetLabel(v)[0] <= 0 ? blue : red);
> console.log(TeXGraph(D, { scale: 1, vertexnodestyle: bluered }));
```




These are used in isomorphism testing for shapes, and to construct minimal paths.

```
function Shape(V, E)
```

```
Constructs a graph shape from the data V, E as described in shapes*.txt data files:
V = sequence of chi's for individual components
E = list of edges v_i->v_j of the form [i,j,edgegcd1,edgegcd2,...]
```

**Example**.

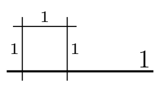
```
> const shape = Shape([1, 2, 3], [[1, 2, 3], [2, 3, 1], [1, 3, 2]])
> console.log(shape.G.Vertices());  // Vertex set of graph G
[ 1, 2, 3 ]
> console.log(shape.G.Edges());     // Edge set of graph G
[ [ 1, 2 ], [ 2, 3 ], [ 1, 3 ] ]
> console.log(shape.D.Vertices());  // Vertex set of graph D
[ 1, 2, 3, 4, 5 ]
> console.log(shape.D.Edges());     // Edge set of graph D
[ [ 1, 4 ], [ 2, 4 ], [ 2, 3 ], [ 1, 5 ], [ 3, 5 ] ]
```

```
function Shapes(genus)
```

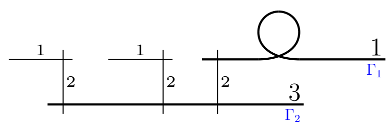
```
Returns all shapes {shape:..., count:...} in a given genus g=2, 3 or 4
```

## 11.9 Dual graphs (GrphDual)

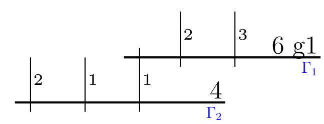
A dual graph is a combinatorial representation of the special fibre of a model with normal crossings. It is a multigraph whose vertices are components $\Gamma_i$, and an edge corresponds to an intersection point of two components. Every component $\Gamma$ has **multiplicity** $m = m_\Gamma$ and geometric **genus** $g = g_\Gamma$. Here are three examples of dual graphs, and their associated reduction types; we always indicate the multiplicity of a component (as an integer), and only indicate the genus when it is positive (as g followed by an integer).



Type $I_4$ (genus 1)



Type $I_1$ −IV∗ (genus 2)



Type $II_{g1}$ −III (genus 8).

A component is **principal** if it meets the rest of the special fibre in at least 3 points (with loops on a component counting twice), or has $g > 0$. The first example has no principal components, and the other two have two each, $\Gamma_1$ and $\Gamma_2$.

This section provides a class (**GrphDual**) for representing dual graphs and their manupulation and invariants.

## 11.10 Default construction

```
function DualGraph(m, g, edges, comptexnames = "default")
```

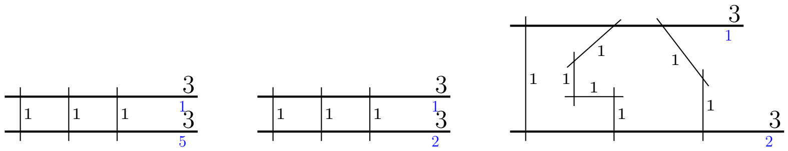
```
Parameters:
m: List of multiplicities for each provided component
g: List of genera for each provided component
edges: List of edges in the form
  [i,j]          - intersection point between component #i and component #j (1<=i,j<=n)
  [i,0,d1,d2,...] - open chain from component #i (1<=i<=n)
  [i,j,d1,d2,...] - link chain from component #i to component #j (1<=i,j<=n)
comptexnames (optional): 'default', function to name components, or a list of names for components.
```

**Example** (Constructing a dual graph).

```
> let m  = [3,1,1,1,3];                          // multiplicities of c1,c2,c3,c4,c5
> let g  = [0,0,0,0,0];                          // genera of c1,c2,c3,c4,c5
> let E  = [[1,2],[1,3],[1,4],[2,5],[3,5],[4,5]];   // edges c1-c2,...
> const G1 = DualGraph(m,g,E);
> console.log(G1.toString());
DualGraph([3,1,1,1,3], [0,0,0,0,0], [[1,2],[1,3],[1,4],[2,5],[3,5],[4,5]])
```

```
> m = [3,3];              // Principal components and chains (same graph)
> g = [0,0];
> E = [[1,2,1],[1,2,1],[1,2,1]];
> const G2 = DualGraph(m,g,E);
> console.log(G2.toString());
DualGraph([3,3,1,1,1], [0,0,0,0,0], [[1,3],[1,4],[1,5],[3,2],[4,2],[5,2]])
> m = [3,3];
> g = [0,0];              // Principal components, different chains
> E = [[1,2,1],[1,2,1,1],[1,2,1,1,1,1]];
> const G3 = DualGraph(m,g,E);
> console.log(G3.toString());
DualGraph([3,3,1,1,1,1,1,1,1], [0,0,0,0,0,0,0,0,0],
  [[1,3],[1,4],[1,6],[3,2],[4,5],[5,2],[6,7],[7,8],[8,9],[9,2]])
```

This is what the three special fibres look like (with component names in blue):



**Example** (Printing dual graph as a string and reconstructing it).

```
> const R = ReductionType("1g1-1g2-1g3-c1");
> const G = R.DualGraph();        // Triangular dual graph on 3 vertices and 3 edges
> console.log(G.toString());
DualGraph([1,1,1], [3,2,1], [[1,2],[1,3],[2,3]])
> const G2 = eval(G.toString());   // and reconstructed back
> console.log(G2.toString());
DualGraph([1,1,1], [3,2,1], [[1,2],[1,3],[2,3]])
```

## 11.11 Step by step construction

```
class GrphDual
```

```
GrphDual.constructor()
```

```
Initialize an empty dual graph
```

```
GrphDual.AddComponent(name, genus, multiplicity, texname = null)
```

```
Adds a component (vertex) to the graph with attributes m, g, and optional texname.
Returns name of the added component (which is given by name if <>None, <>"")
```

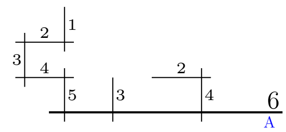
```
GrphDual.AddEdge(node1, node2)
```

```
Adds an edge between two components (vertices) in the graph.
```

```
GrphDual.AddChain(c1, c2, mults)
```

```
Adds a chain of P1s with multiplicities between c1 and c2. Adds as many vertices as
there are multiplicities in 'mults', and links them in a chain starting at c1 and
ending at c2 (if c2 is provided, else it's an open chain).
```

**Example** (Type II* reduction). This is how we can construct the dual graph of the type II* elliptic curve, creating some components and edges by hand, and adding the rest as open chains.




```
> var G = new GrphDual();
> var c1 = G.AddComponent("A", 0, 6);   // Called 'A', multiplicity 6
> var c2 = G.AddComponent("", 0, 3);    // default name ('c2')
> G.AddEdge(c1, c2);                    // Link the two (shortest chain)
> G.AddChain(c1, null, [4, 2]);         // The other two chains
> G.AddChain(c1, null, [5, 4, 3, 2, 1]);
> console.log(G.Components());
[ "A", "c2", "c3", "c4", "c5", "c6", "c7", "c8", "c9" ]
> console.log(G.ReductionType().Label());
II*
```

## 11.12 Global methods and arithmetic invariants

`GrphDual.Graph()`

Returns the underlying graph.

`GrphDual.Components()`

Returns the list of vertices of the underlying graph.

`GrphDual.IsConnected()`

Check that the dual graph is connected

`GrphDual.HasIntegralSelfIntersections()`

Are all component self-intersections integers

`GrphDual.AbelianDimension()`

Sum of genera of components

`GrphDual.ToricDimension()`

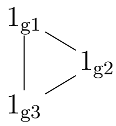
Number of loops in the dual graph

`GrphDual.IntersectionMatrix()`

Intersection matrix for a dual graph, whose entries are pairwise intersection numbers of the components.

**Example**. Here is the dual graph of the reduction type $1_{g3}-1_{g2}-1_{g1}-c_1$, consisting of three components genus 1,2,3, all of multiplicity 1, connected in a triangle.

```
> var G = DualGraph([1,1,1],[1,2,3],[[1,2],[2,3],[3,1]]);
> console.assert(G.IsConnected()); // Check the dual graph is connected
> console.assert(G.HasIntegralSelfIntersections()); // and every component c has c.c in Z
> console.log(G.AbelianDimension());                 // genera 1+2+3 => 6
6
> console.log(G.ToricDimension());                   // 1 loop       => 1
1
> console.log(G.ReductionType().TeX());
```




```
> console.log(G.IntersectionMatrix()); // Intersection(G,v,w) for v,w components
[ [ -2, 1, 1 ], [ 1, -2, 1 ], [ 1, 1, -2 ] ]
```

```
GrphDual.PrincipalComponents()
```

```
Return a list of indices of principal components.
A vertex is a principal component if either its genus is greater than 0
or it has 3 or more incident edges (counting loops twice).
In the exceptional case [d]I_n one component is declared principal.
```

```
GrphDual.ChainsOfP1s()
```

```
Returns a sequence of tuples [<v0,v1,[chain multiplicities]>] for chains of P1s between principal
components, and v1=None for open chains
```

```
GrphDual.ReductionType()
```

```
Reduction type from a dual graph
```

## 11.13 Contracting components to get a mrnc model

```
GrphDual.ContractComponent(c, checks=true)
```

```
Contract a component in the dual graph, assuming it meets one or two components, and has genus 0.
```

```
GrphDual.MakeMRNC()
```

```
Repeatedly contract all genus 0 components of self-intersection -1, resulting in a minimal model
with normal crossings.
```

```
GrphDual.Check()
```

```
Check that the graph is connected and self-intersections are integers.
```

**Example** (Contracting components).

```
> let G = DualGraph([1,1],[1,0],[[1,2,1,1,1]]);  // Not a minimal rnc model
> console.log(G.Components(), G.Components().map(v => G.Intersection(v,v)));
[ "1", "2", "c3", "c4", "c5" ] [ -1, -1, -2, -2, -2 ]
> G.ContractComponent("2");   // Remove the last component
> G.ContractComponent("c5");  //   and then the one before that
> console.log(G.Components());
[ "1", "c3", "c4" ]
> console.log(G.toString());
DualGraph([1,1,1], [1,0,0], [[1,2],[2,3]])
> G.MakeMRNC();               // Contract the rest of the chain
> console.log(G.Components());
[ "1" ]
> console.log(G.toString());
DualGraph([1], [1], [])
> console.log(G.ReductionType().Label());   // Associated reduction type
1g1
```

## 11.14 Invariants of individual vertices

```
GrphDual.HasComponent(c)
```

Test whether the graph has a component named c

```
GrphDual.Multiplicity(v)
```

Multiplicity m of the vertex

```
GrphDual.Multiplicities()
```

Returns the list of multiplicities of all the vertices.

```
GrphDual.Genus(v)
```

Genus g of the vertex

```
GrphDual.Genera()
```

Returns the list of geometric genera of all the vertices.

```
GrphDual.Neighbours(i)
```

List of incident vertices, with each loop contributing the vertex itself twice

```
GrphDual.Intersection(c1, c2)
```

Compute the intersection number between components c1 and c2 (or self-intersection if c1=c2).

```
GrphDual.TeXName(v)
```

TeXName assigned to a vertex v

**Example** (Cycle of 5 components).

```
> let G = DualGraph([1], [1], [[1,1,1,1,1,1]]);
> let C = G.Components();
> console.log(C);
[ "1", "c2", "c3", "c4", "c5" ]
> console.assert(G.HasComponent("c2"));
> console.log(G.Multiplicity("c2"));
1
> console.log(G.Genus("c2"));
0
> console.log(G.IntersectionMatrix());
-2 1 0 0 1
1 -2 1 0 0
0 1 -2 1 0
0 0 1 -2 1
1 0 0 1 -2
```

## 11.15 Reduction types (RedType)

Now we come to reduction types, implemented through the class `RedType`. They can be constructed in a variety of ways:

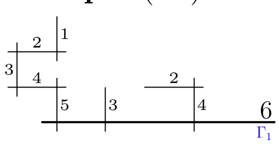
| | |
|---|---|
| `ReductionType(m,g,O,L)` | Construct from a sequence of components (including all principal ones), their multiplicities m, genera g, outgoing multiplicities of open chains O, and link chains L beween them, e.g.<br>`ReductionType([1],[0],[[]],[[1,1,0,0,3]])` (Type $I_3$) |
| `ReductionTypes(g)` | All reduction types in genus g. Can restrict to just semistable ones and/or ask for their count instead of actual the types, e.g.<br>`ReductionTypes(2)` (all 104 genus 2 types)<br>`ReductionTypes(2, countonly=True)` (only count them)<br>`ReductionTypes(2, semistable=True)` (7 semistable ones) |
| `ReductionType(label)` | Construct from a canonical label, e.g.<br>`ReductionType("I3")` |
| `ReductionType(G)` | Construct from a dual graph, e.g.<br>`ReductionType(DualGraph([1],[1],[]))` (good elliptic curve) |
| `ReductionTypes(S)` | Reduction types with a given shape, e.g.<br>`ReductionTypes(Shape([2],[]))` (46 of the genus 2 types) |

Conversely, from a reduction type we can construct its dual graph (`R.DualGraph()`) and a canonical label `R.Label()`), and these functions are also described in this section. Finally, there are functions to draw reduction types in TeX (`R.TeX()`).

```
function ReductionType(...args)
```

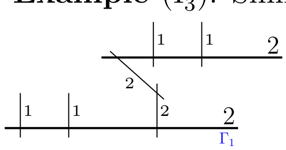
```
Reduction type from either:
ReductionType(label: Str)    reduction type from a label, e.g. "I3"
ReductionType(G: GrphDual)   reduction type from a dual graph
ReductionType(m, g, O, L) reduction type from sequence of components, their invariants, and chains
of P1s:
  m = sequence of multiplicities of components c_1,...,c_k
  g = sequence of their geometric genera
  O = outgoing multiplicities of open chains, one sequence for each component
  L = link chains, of the form
      [[i,j,di,dj,n],...] - link chain from c_i to c_j with multiplicities m[i],di,...,dj,m[j], of
      depth n
      n can be omitted, and chain data [i,j,di,dj] is interpreted as having minimal possible depth.
```

**Example** ($II^*$). We construct Kodaira type II∗ as a reduction type



```
> const m = [6];          // multiplicity of one starting component Gamma_1
> const g = [0];          // their geometric genera
> const O = [[3, 4, 5]];  // outgoing multiplicities of open chains from each of them
> const L = [];           // link chains
> const R = ReductionType(m, g, O, L);
> console.log(R.Label());
II*
> console.assert(R.equals(ReductionType("II*")));    // same type from label
Assertion failed
```

**Example** ($I_3^*$). Similarly, we construct Kodaira type $I_3^*$ as a reduction type



```
> const m = [2, 2];            // multiplicities of starting components Gamma_1, Gamma_2
```

```
> const g = [0, 0];             // their geometric genera
> const O = [[1, 1], [1, 1]]; // outgoing multiplicities of open chains from each of them
> const L = [[1, 2, 2, 2, 3]];  // link chains [[i,j, di,dj ,optional depth],...]
> const R = ReductionType(m, g, O, L);
> console.log(R.Label());
I3*
> console.assert(R.equals(ReductionType("I3*")));    // same type from label
Assertion failed
```

```
function ReductionTypes(arg, options = {})
```

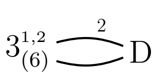
```
ReductionTypes(arg, { countonly=false, semistable=false, elliptic=false })
  - All reduction types in genus g <= 6 or their count (if countonly=true; faster).
  - semistable=true restricts to semistable types.
  - elliptic=true (when g=1) restricts to Kodaira types of elliptic curves.

ReductionTypes(S, { countonly=false, semistable=false })
  - Sequence of reduction types with a given shape S, semistable if necessary.
  - If countonly=true, only return the number of types (faster).

Returns a sequence of RedType's or an integer if countonly=true.
```

**Example** (Reduction types in a given genus). Here are all reduction types for elliptic curves (10 Kodaira types), the count for genus 2 (104 Namikawa-Ueno types) and the count for semistable types in genus 3.

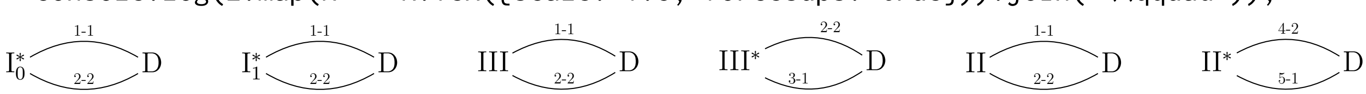
```
> console.log(ReductionTypes(1, {elliptic: true}).map(R => R.Label()));
[ "1g1", "I1", "I0*", "I1*", "IV", "IV*", "III", "III*", "II", "II*" ]
> console.log(ReductionTypes(2, {countonly: true}));
104
> console.log(ReductionTypes(3, {semistable: true, countonly: true}));
42
```

**Example** (Reduction types with a given shape). There are 1901 reduction types in genus 3, in 35 different shapes. Here is one of the more 'exotic' ones, with 6 types in it. It has two vertices with $\chi = -3$ and $\chi = -1$ and two edges between them, with gcd 1 and 2.

```
> const S = Shape([3, 1], [[1, 2, 1, 2]]);
> console.log(S.TeX());
```




```
> const L = ReductionTypes(S);
> console.log(L.map(R => R.Label()));
[ "I0*-=D", "I1*-=D", "III--{2-2}D", "III*-{2-2}-D", "II--{2-2}D", "II*-{4-2}-D" ]
> console.log(L.map(R => R.TeX({scale: 1.5, forcesups: true})).join("\\qquad"));
```




```
class RedType
```

```
RedType.get(target, prop)
```

```
RedType.Chi()
```

```
Total Euler characteristic of R
```

```
RedType.Genus()
```

```
Total genus of R
```

**Example**.

```
> R = new ReductionType("III=(3)III-{2-2}II-{6-12}18g2^6,12");
> console.log(R.Label());      // Canonical label
[6]Tg2-{12-6}II-{2-2}III=(3)III
> console.log(R.Genus());      // Total genus
43
```

```
RedType.IsGood()
```

true if comes from a curve with good reduction

```
RedType.IsSemistable()
```

true if comes from a curve with semistable reduction (all (principal) components of an mrnc model have multiplicity 1)

```
RedType.IsSemistableTotallyToric()
```

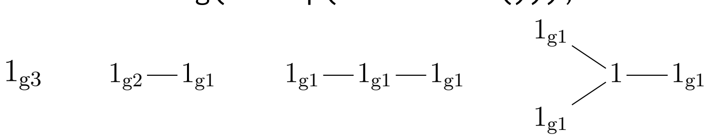
true if comes from a curve with semistable totally toric reduction (semistable with no positive genus components)

```
RedType.IsSemistableTotallyAbelian()
```

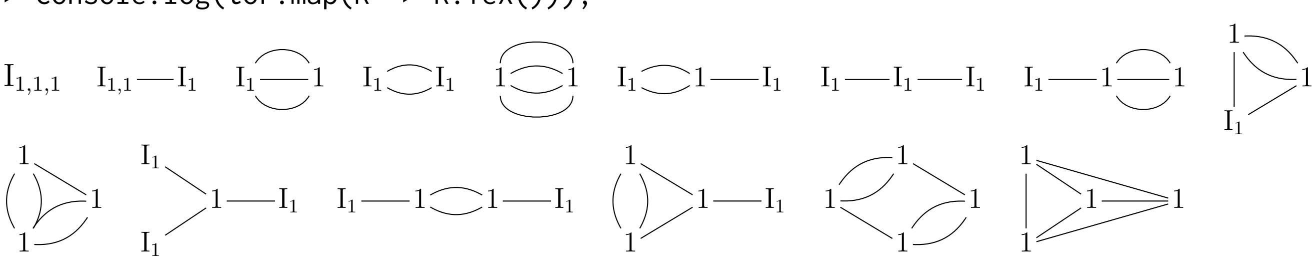
true if comes from a curve with semistable totally abelian reduction (semistable with no loops in the dual graph)

**Example** (Semistable reduction types).

```
> let semi = ReductionTypes(3, {semistable: true}); // genus 3, semistable,
> console.log(semi.map(R => R.Label()).join(" "));
1g3 I1g2 I1g1,1 I1,1,1 1g2-1g1 1g2-I1 I1g1-1g1 I1g1-I1 I1,1-1g1 I1,1-I1 1g1---1 I1---1
  1g1--1g1 I1--I1 1g1--I1 1----1 1g1--1-1g1 1g1--1-I1 I1--1-1g1 I1--1-I1 1g1-1g1-1g1
  1g1-I1-1g1 1g1-1g1-I1 1g1-I1-I1 I1-1g1-I1 I1-I1-I1 1g1-1---1 I1-1---1 1g1-1--1-c1
  I1-1--1-c1 1--1-1--c1 1g1-1-1g1&1g1-c2 1g1-1-1g1&I1-c2 1g1-1-I1&I1-c2 I1-1-I1&I1-c2
  1g1-1--1-1g1 1g1-1--1-I1 I1-1--1-I1 1g1-1-1--1-c2 I1-1-1--1-c2 1-1--1-1--c1
  1-1-1-1-c1-c3&c2-c4
> let ab = semi.filter(R => R.IsSemistableTotallyAbelian()); // totally abelian reduction
> console.log(ab.map(R => R.TeX()));
```



```
> let tor = semi.filter(R => R.IsSemistableTotallyToric());
> console.log(tor.map(R => R.TeX()));
```



Count semistable reduction types in genus 2,3,4,... (OEIS A174224)

```
> console.log([2,3,4].map(n => ReductionTypes(n, {semistable: true, countonly: true})));
[ 7, 42, 379 ]
```

```
RedType.TamagawaNumber()
```

Tamagawa number of the curve with a given reduction type, over an algebraically closed residue field

**Example** (Tamagawa numbers for reduction types of elliptic curves).

```
> var E = ReductionTypes(1, {elliptic: true});
> for (const R of E) {console.log(R.Label(), R.TamagawaNumber());}
1g1 1
I1 1
I0* 4
I1* 4
IV 3
IV* 3
III 2
III* 2
II 1
II* 1
```

## 11.16 Invariants of individual principal components and chains

`RedType.PrincipalTypes()`

Principal types (vertices) of the reduction type

`RedType.length()`

Number of principal types in a reduction type

`RedType.getItem(i)`

Principal type number i in the reduction type, accessed as R[i] (numbered from i=1)

`RedType.LinkChains()`

Return all the link chains in the reduction type

`RedType.LooseChains()`

Return all the link chains in R between different principal components, sorted as in label.

`RedType.Multiplicities()`

Sequence of multiplicities of principal types

`RedType.Genera()`

Sequence of geometric genera of principal types

`RedType.GCD()`

GCD detecting non-primitive types

`RedType.Shape()`

The shape of the reduction type.

**Example** (Principal types and chains). Take a reduction type that consists of smooth curves of genus 3, 2 and 1, connected with two chains of $\mathbb{P}^1$s of depth 2.

```
> var R = ReductionType("1g3-(2)1g2-(2)1g1");
> console.log(R.TeX());
```

$1_{g3} \frac{}{\ 2\ } 1_{g2} \frac{}{\ 2\ } 1_{g1}$

This is how we access the three principal types, their primary invariants, and the chains. Individual principal types can be accessed as R[i], and all of them as R.PrincipalTypes()

```
> console.log(R[1].Label(), R[2].Label(), R[3].Label());
1g3 1g2 1g1
> console.log(R.Genera());          // geometric genus g of each principal type
[ 3, 2, 1 ]
> console.log(R.Multiplicities()); // multiplicity m of each principal type
[ 1, 1, 1 ]
> console.log(R.LinkChains().join(", "));     // chains, including loops and D-links
[1] loose c1 1,1 -(2) c2 1,1, [2] loose c2 1,1 -(2) c3 1,1
```

`RedType.Weight()`

Weight of a reduction type, used for comparison and sorting

**Example.**

```
> R1 = ReductionType("I1g1")
> console.log(R1.Weight());
[ 1, 0, -2, 1, -1, 0, 0, 1, 0, 1, 1, 1, 4, 73, 49, 103, 49 ]
> R2 = ReductionType("Dg1")
> console.log(R2.Weight());
[ 1, 0, -2, 2, -1, 0, 0, 0, 2, 1, 1, 3, 68, 103, 49 ]
> console.log(R1.lessThan(R2));     // I1g1<Dg1 so it precedes it in tables
true
```

`RedType.equals(other)`

Equality comparison based on label.

`RedType.lessThan(other)`

Less than comparison based on weight.

`RedType.greaterThan(other)`

Greater than comparison based on weight.

`RedType.lessThanOrEqual(other)`

Less than or equal to comparison based on weight.

`RedType.greaterThanOrEqual(other)`

Greater than or equal to comparison based on weight.

**Example** (Sorted reduction types in genus 1 and 2).

```
> var L = ReductionTypes(1, {elliptic: true});
> RedType.Sort(L);
> console.log(L.map(R => R.Label()).join(", "));
1g1, I1, I0*, I1*, IV, IV*, III, III*, II, II*
> L = ReductionTypes(2);
> RedType.Sort(L);
> console.log(L.map(R => R.Label()).join(", "));
1g2, I1g1, I1,1, Dg1, [2]g1_D, 2^1,1,1,1,1,1, I0*_0, D_{2-2}, I0*_D, I1*_0, [2]_1,D,
  I1*_D, [2]_D,D,D, 3^1,1,2,2, IV_0, IV*_-1, 4^1,3,2,2, III_0, III*_-1, III_D, 4^1,3_D,
  III*_D, [2]I0*_D, [2]I1*_D, 5^1,1,3, 5^1,2,2, 5^2,4,4, 5^3,3,4, 6^1,1,4, 6^5,5,2,
  6^2,4,3,3, II_D, [2]IV_D, [2]T_{6}D, [2]IV*_D, II*_D, 8^1,3,4, 8^5,7,4, [2]III_D,
  [2]III*_D, 10^1,4,5, 10^3,2,5, 10^7,8,5, 10^9,6,5, [2]II_D, [2]II*_D, 1g1-1g1, 1g1-I1,
  1g1-I0*, 1g1-I1*, 1g1-IV, 1g1-IV*, 1g1-III, 1g1-III*, 1g1-II, 1g1-II*, I1-I1, I1-I0*,
```

```
I1-I1*, I1-IV, I1-IV*, I1-III, I1-III*, I1-II, I1-II*, I0*-I0*, I0*-I1*, I0*-IV,
I0*-IV*, I0*-III, I0*-III*, I0*-II, I0*-II*, I1*-I1*, I1*-IV, I1*-IV*, I1*-III,
I1*-III*, I1*-II, I1*-II*, IV-IV, IV-IV*, IV-III, IV-III*, IV-II, IV-II*, IV*-IV*,
IV*-III, IV*-III*, IV*-II, IV*-II*, III-III, III-III*, III-II, III-II*, III*-III*,
III*-II, III*-II*, II-II, II-II*, II*-II*, T=T, D-=D, 1---1
```

## 11.17 Reduction types, labels, and dual graphs

```
RedType.DualGraph({compnames="default"} = {})
```

```
Full dual graph from a reduction type, possibly with variable length edges,
and optional names of components.
Returns: GrphDual - The constructed dual graph.
```

```
RedType.Label(options = {})
```

```
Return canonical string label of a reduction type.
  tex:=true        gives a TeX-friendly label (\redtype{...})
  html:=true       gives a HTML-friendly label (<span class='redtype'>...</span>)
  wrap:=false      keeps the format above but removes \redtype / <span> wrapping
  forcesubs:=true  forces depths of chains & loops to be always printed (usually in round brackets)
  forcesups:=true  forces outgoing chain multiplicities to be always printed (in curly brackets).
depths can be "default", "original", "minimal", or a custom sequence.
```

```
RedType.Family()
```

```
Returns the reduction type label with minimal chain lengths in the same family.
```

**Example** (Plain and TeX labels for reduction types).

```
> var R = ReductionType("IIg1_1-(3)III-(4)IV");
> console.log(R.Label());             // plain text label
IIg1_1-(3)III-(4)IV
> var R2 = ReductionType(R.Label());
> console.assert(R.equals(R2));       // can be used to reconstruct the type
Assertion failed
> console.log(R.Family());            // family (reduction type with minimal depths)
IIg1_1-III-IV
> console.log(R.Label({tex: true})); // label in TeX
```

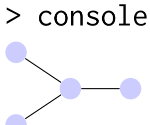
$\mathrm{II}_{\mathrm{g}1,1}.(3)\mathrm{III}.(4)\mathrm{IV}$

```
> console.log(R[1].toString());       // first principal type as a standalone type
IIg1_1-{1}
> console.log(R.TeX());               // reduction type as a graph in TeX
```

$\mathrm{II}_{\mathrm{g}1,1}\overline{\phantom{x}}_{3}\mathrm{III}\overline{\phantom{xx}}_{4}\mathrm{IV}$

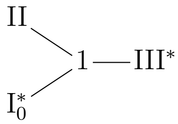
**Example** (Canonical label in detail). Take a graph $G$ on 4 vertices

```
> var G = new Graph(4,[[1,2],[1,3],[1,4]]);
> console.log(TeXGraph(G, {labels: "none"}));
```

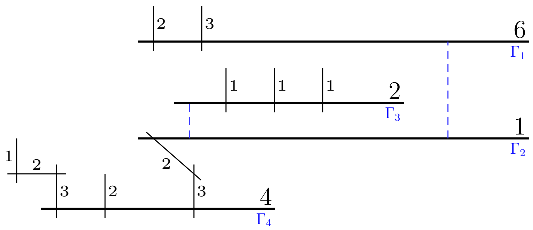


Place a component of multiplicity 1 at the root and II, III$^*$, $\mathrm{I}_0^*$ at the three leaves. Link each leaf to the root with a chain of multiplicity 1. This gives a reduction type that occurs for genus 3 curves:

```
> var R = ReductionType("1-II&c1-III*&c1-I0*");    // First component is the root,
> console.log(R.TeX());                             //   the other three are leaves
```




Here is the corresponding special fibre




How is the following canonical label chosen among all possible labels?

```
> console.log(R.Label());
I0*-1-II&III*-c2
```

Each principal component is a principal type (as there are no loops or D-links), and its primary invariants are its Euler characteristic $\chi$ and a multiset lgcd of gcd's of outgoing (loose) link chains

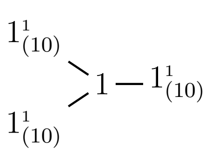
```
> let Prin = R.PrincipalTypes();
> console.log(Prin.map(S => S.toString()));
[ "I0*-{1}", "1-{1}-{1}-{1}", "II-{1}", "III*-{3}" ]
> console.log(Prin.map(S => S.Chi()));    // add up to 2-2*genus, so genus=3
[ -1, -1, -1, -1 ]
> console.log(Prin.map(S => S.LGCD()));
[ [ 1 ], [ 1, 1, 1 ], [ 1 ], [ 1 ] ]
```

All four leaves have $\chi = -2$, lgcd=[1] and the root $\chi = 1$, lgcd=$[1, 1, 1]$. There are 10 types of the former kind (II-, III-, IV-, ...), drawn as $1^1_{(10)}$ in shapes, and one of the root kind, drawn as 1.

```
> console.log(PrincipalTypes(-1,[1]).toString());
1g1-{1},I1-{1},I0*-{1},I1*-{1},IV-{1},IV*-{2},III-{1},III*-{3},II-{1},II*-{5}
> console.log(PrincipalTypes(-1,[1,1,1]).toString());
1-{1}-{1}-{1}
```

Together they form a shape graph $S$ as follows:

```
> var S = R.Shape();
> console.log(S.TeX({scale: 1}));
```




The vertices and edges of $S$ are assigned weights. Vertex weights are $\chi$'s, edge weights are lgcd's

```
> console.log(S.VertexLabels());
[ 1, 1, 1, 1 ]
> console.log(S.EdgeLabels());
[ [ 1, 2, 1 ], [ 2, 3, 1 ], [ 2, 4, 1 ] ]
```

Then the shortest path is found using MinimumWeightPaths. It is v-v-v&v-2 (v=new vertex with $\chi = -1$, -=edge, &=jump). Note that by convention actual edges are preferred to jumps, and going to a new vertex preferred to revisiting an old one. Also vertices with smaller $\chi$ come first, if possible, as they have smaller labels.

v-v-v&v-2 < v-v&v-2-v (jumps are larger than edge marks)
v-v-v&v-2 < v-v-v&2-v (repeated vertex indices are larger than vertex marks)

```
> var [P, T] = MinimumWeightPaths(S);
> console.log(P);      // v-v-v&v-2
[ [ 0, [ -1 ], false ], [ 0, [ -1 ], false ], [ 0, [ -1 ], true ], [ 0, [ -1 ], false ], [
  2, [ -1 ], true ] ]
```

This path can be used to construct the graph, and determines it up to isomorphism. There are $|\operatorname{Aut} S| = 6$ ways to trail $S$ in accordance with this path, and as far the shape is concerned, they are completely identical.

```
> console.log(T);
[ [ 1, 2, 3, 4, 2 ], [ 1, 2, 4, 3, 2 ], [ 3, 2, 1, 4, 2 ], [ 3, 2, 4, 1, 2 ], [ 4, 2, 1,
  3, 2 ], [ 4, 2, 3, 1, 2 ] ]
```

This gives six possible labels for our reduction type that all traverse the shape according to path $P$:

```
> var l = (i) => R[i].Label();
> console.log(T.map(c => `${l(c[0])}-${l(c[1])}-${l(c[2])}&${l(c[3])}-c2`));
[ "I0*-1-II&III*-c2", "I0*-1-III*&II-c2", "II-1-I0*&III*-c2", "II-1-III*&I0*-c2",
  "III*-1-I0*&II-c2", "III*-1-II&I0*-c2" ]
```

Now we assign weights to vertices and edges that characterise the actual shape components (rather than just their $\chi$) and link chains (rather than just their lgcd)

```
> console.log(R.PrincipalTypes().map(S => S.Weight()));
[ [ -1, 2, -0, 1, 0, 0, 3, 1, 1, 1, 1 ], [ -1, 1, -0, 3, 0, 0, 0, 1, 1, 1 ], [ -1, 6, -0,
  1, 0, 0, 2, 2, 3, 1 ], [ -1, 4, -0, 1, 0, 0, 2, 3, 2, 3 ] ]
> console.log(R.EdgesWeight(2,1));    // weight of the 1-II link chain
[ 1, 1, 0 ]
> console.log(R.EdgesWeight(2,3));    // weight of the 1-I0* link chain
[ 1, 1, 0 ]
> console.log(R.EdgesWeight(2,4));    // weight of the 1-III* link chain
[ 1, 3, 0 ]
```

The component weight Weight(R[i]) starts with $(\chi, -m, -g, ...)$ so when all components have the same $\chi$ like in this example, the ones with large multiplicity $m$ have smaller weight. Because m(II)=6, m(III*)=4, m(I0*)=2, the trails $T[1]$ and $T[2]$ are preferred to the other four. They both start with a component II, then an edge II-1 and a component 1. After that they differ in that $T[1]$ traverses an edge 1-I0* and $T[2]$ an edge 1-III*. Because the edge weight is smaller for $T[1]$, this is the minimal path, and it determines the label for $R$:

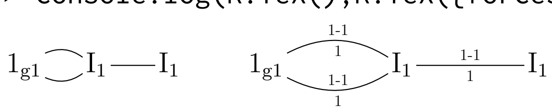
```
> console.log(R.Label());
I0*-1-II&III*-c2
```

```
RedType.TeX(options = {})
```

```
TikZ representation of a reduction type, as a graph with PrincipalTypes (principal components with
chi>0) as vertices, and edges for link chains.
oneline:=true removes line breaks.
forcesups:=true and/or forcesubs:=true shows edge decorations (outgoing multiplicities and/or chain
depths) even when they are default.
```

**Example** (TeX for reduction types).

```
> R = new ReductionType("1g1--I1-I1");
> console.log(R.TeX(),R.TeX({forcesups: true, forcesubs: true, scale: 1.5}));
```



**Example** (Degenerations of two elliptic curves meeting at a point).

```
> const S = ReductionType("1g1-1g1").Shape(); // Two elliptic curves meeting at a point
  (genus 2)
```

The corresponding shape is a graph v-v with two vertices with $\chi = -1$ and one edge of gcd 1

```
> console.log(S.TeX());
```

$1^1_{(10)}$ —— $1^1_{(10)}$

There are 10 possibilities for such a vertex, one for each Kodaira type, and Binomial(10,2)=55 such types in total

```
> console.log(PrincipalTypes(-1,[1]).join(", "));
1g1-{1}, I1-{1}, I0*-{1}, I1*-{1}, IV-{1}, IV*-{2}, III-{1}, III*-{3}, II-{1}, II*-{5}
> console.log(ReductionTypes(S, {countonly: true}));
55
```

```
RedType.SetDepths(depth)
```

```
Set depths for DualGraph and Label based on either a function or a sequence.
If `depth` is a function, it should be of the form:
  depth(e: RedChain) -> int/str
For example:
  e => e.depth  // Original depths
  e => MinimalLinkDepth(e.mi, e.di, e.mj, e.dj)  // Minimal depths
  e => `n_${e.index}`  // Custom string-based depth

If `depth` is a sequence, its length must match the number of link chains in the reduction type.

Raises:
  Error: If `depth` is neither a function nor a sequence or if the sequence length doesn't match.
```

```
RedType.SetVariableDepths()
```

```
Set depths for DualGraph and Label to a variable depth format like 'n_i'.
```

```
RedType.SetOriginalDepths()
```

```
Remove custom depths and reset to original depths for printing in Label and other functions.
```

```
RedType.SetMinimalDepths()
```

```
Set depths to minimal ones in the family for each edge.
```

```
RedType.GetDepths()
```

```
Return the current depths (string sequence) set by SetDepths or the original ones if not changed.
```

**Example** (Setting variable depths for drawing families).

```
> var R = new ReductionType("I3-(2)I5");
> console.log(R.Label({tex: true}));
```

$\mathrm{I}_3.(2)\mathrm{I}_5$

```
> R.SetDepths(["a", "b", "5"]);    // Make two of the three chains variable depth
> console.log(R.Label({tex: true}));
```

$\mathrm{I}_a.(b)\mathrm{I}_5$

```
> R.SetOriginalDepths();
> console.log(R.Label({tex: true}));
```

$\mathrm{I}_3.(2)\mathrm{I}_5$

## 12 References


[Do1] T. Dokchitser, Models of curves over discrete valuation rings, Duke Math. J. 170, no. 11 (2021), 2519–2574.

[Do2] T. Dokchitser, Classification of reduction types of curves, in preparation.

[Liu] Q. Liu, Modèles entiers des courbes hyperelliptiques sur un corps de valuation discrète, Trans. Amer. Math. Soc. 348 no. 11 (1996), 4577–4610.

[Mu] S. Muselli, Regular models of hyperelliptic curves, arXiv:2206.10420, to appear in Indag. Math.

[NU] Y. Namikawa, K. Ueno, The Complete Classification of Fibres in Pencils of Curves of Genus Two, Manuscripta Math. 9 (1973), 143–186.

[Ta] J. Tate, Algorithm for determining the type of a singular fiber in an elliptic pencil, in: Modular Functions of One Variable IV, Lect. Notes in Math. 476, B. J. Birch and W. Kuyk, eds., Springer-Verlag, Berlin, 1975, 33–52.