Libfact

A C library to compute generalised factorial functions and co.

2020
October

This paper presents a tiny C library builded over Pari/GP to compute with many objects related to generalized factorial functions of subsets of number fields. The library focuses on two kind of subset : finite subsets of algebraic integers and the whole ring of algebraic integers of a number field.

Libfact provides functions to compute three different kind of orderings and their invariants as well as the associated generalized factorial functions. The ultimate goal is to provide functions to compute, when they exist, so-called regular basis for various modules of integer-valued polynomials.

A section is dedicated to the particular case of quadratic number fields and putative simultaneous ordering of their ring of algebraic integers.

A script to install in a gp session all the functions presented below is provided with the package. Whenever default arguments are present in the gp-installed version, a corresponding prototype is provided in the function description of this documentation. The script also install help sections to consult within gp.

Contents

1 Orderings and factorial ideals of finite sets in $\mathbb{Z}_K$

This section presents functions to compute different kind of orderings of finite sets of algebraic integers in a number field generalizing regular $\mathfrak{p}$-orderings introduced by Bhargava in [Bha97] and the associeted generalized factorial functions.

Beware that the notion of ordering here is more general than just permuting the elements of a sequence in an appropriate way.

Through all this section :

- a *set* (often `S`) is a vector of distinct algebraic integers (the order does not matter)
- a *sequence* is a vector of algebraic integers (not neccessarily distinct) where the order matter, i.e permuting in anyway the sequence lead to a different sequence
- *ordering* a set means transforming a set into a sequence of elements of the set. The resulting sequence can be larger than the original set and there can be repetitions. This is the case for r-removed orderings for example.
- the sequence obtained after ordering a set is called an *ordering* or the *ordered set*.

1.1 Orderings of finite sets in $\mathbb{Z}_K$

Functions to compute three type of orderings of a finite set of algebraic integers in $\mathbb{Z}_K$, namely :

- $\mathfrak{p}$-ordering
- r-removed $\mathfrak{p}$-ordering
- $\mathfrak{p}$-ordering of order h

The precise and original definitions can be found in [Bha97] and [Bha09]. The first type is actually a particular case of both the second and third one for $r = 0$ and $h = \infty$ respictively, but it is convenient to have simpler dedicated functions.

There are two different things you can be interested in related to these orderings.

The first one is the ordering itself, meaning the sequence of elements of the set that realizes the ordering. For this purpose, the library provides the three functions `pord`, `rpord` and `opord` (`r` for `r`emoved and `o` for `o`rder). They return the requested ordered set and accept a wide range of arguments to tune the request and ask for additional related objects.

The second one is the ordering's invariants.For this purpose, the library provides the three function `pord_e`, `rpord_e` and `opord_e` (just append `_e` for `e`xponent). They return an increasing vector of integers which are the exponents of the invariant sequence associated to the ordering. This three functions should be preferred when only the invariants are needed since they are really faster in most cases.

The six functions `pord`, `pord_e`, `rpord`, `rpord_e`, `opord` and `opord_e` use shuffling (see [Joh10]) for maximum efficiency. It is a simple divide-and-conquer strategy : split the working set in different modular classes, order the resulting sets separately and recompose an ordering of the original set by shuffling the different orderings in an appropriate way. Anyway, be aware that r-removed $\mathfrak{p}$-orderings are much slower to compute since they involve taking a minimum over both the working set and all subsets of a certain cardinality.

There is usually many way to order a set, but the functions `pord`, `rpord` and `opord` do not provide facilities to ask for a specific one, except for the first element in the case of `pord` and `opord`, so you should consider the returned ordered set as a random one.

If you are interested in computing all orderings of a set or all orderings with specific properties (like starting by a choosen subset for example), see the miscellaneous functions in 1.1.4 and the discussion in the to-do list.

1.1.1 $\mathfrak{p}$-ordering of finite sets

`GEN pord_e(GEN nf, GEN pr, GEN S, long trunc)`

Return in a vector the first `trunc-1` invariant exponents of any `pr`-ordering of the set `S`. If `trunc = -1`, it is set to `#S`.

The gp prototype is `pord_e(nf, pr, S, {trunc = -1})`

Examples

```
? K = nfinit(x^2 + 1);
? pr = idealprimedec(K, 2)[1]
? S = [1, 3, x, x + 2, -3*x, 3*x + 8, -2*x + 3, 3*x - 7, 8*x, 2*x + 3, 34*x + 2];
? pord_e(K, pr, S, #S)
% = [0, 1, 1, 3, 4, 4, 7, 8, 11, 13]
```

`GEN pord(GEN nf, GEN pr, GEN S, long first, long trunc, GEN *ex, GEN *inv)`

Return in a vector beginning by `S[first]` the first `trunc` elements of a `pr`-ordering of the set `S`. If `trunc = -1`, it is set to `#S`. Set to `*ex` the extraction small vector and to `*inv` the invariant vector as it would be returned by `pord_e(nf, pr, S, trunc)`.

The gp prototype is `pord(nf, pr, S, {first = 1}, {trunc = -1}, {&ex = NULL}, {&inv = NULL})`

Examples

```
? K = nfinit(x^2 + 1)
? pr = idealprimedec(K, 2)[1]
? S = [1, 3, x, x + 2, -3*x, 3*x + 8, -2*x + 3, 3*x - 7, 8*x, 2*x + 3, 34*x + 2];

? pord(K, pr, S)    /* just a pr-ordering */
% = [1, 3*x - 7, x, 8*x, 3, x + 2, 34*x + 2, 3*x + 8, -2*x + 3, -3*x, 2*x + 3];

? pord(K, pr, S, 7)    /* beginning by -2*x + 3 */
% = [-2*x + 3, 3*x - 7, x, 8*x, 3, x + 2, 34*x + 2, 3*x + 8, 1, -3*x, 2*x + 3]
```

```
? pord(K, pr, S, 7, 4)    /* only the first 4 elements */
% = [-2*x + 3, 3*x - 7, x, 8*x]

? pord(K, pr, S, 7, 4, &ex); ex    /* request the extraction vector */
% = Vecsmall([7, 4, 3, 5])

? pord(K, pr, S, 7, 4, &ex, &inv); inv    /* request the invariants */
% = [0, 1, 1]
```

`int ispord(GEN nf, GEN pr, GEN S, long trunc, GEN *i)`

Return 1 if the sequence of the first `trunc` elements of `S` is the beginning of a `pr`-ordering of the set `S`, 0 otherwise. If `trunc = -1`, it is set to `#S`. Set to `*i` the index of the first element of `S` responsible for failure.

The gp prototype is `ispord(nf, pr, S, {trunc = -1}, {&i = NULL})`

Examples

```
? K = nfinit(x^2 + 1);
? pr = idealprimedec(K,2)[1];
? S = [3, x, 1 + x, 3*x + 1, 2*x - 2, 4*x, -3*x - 1];

? ispord(K, pr, S)
% = 0    /* S is not a pr ordering */

? ispord(K, pr, S,, &i); i
% i = 2    /* because of x */
```

`GEN pord_get_e(GEN nf, GEN pr, GEN po, long trunc)`

Return in a vector the first `trunc - 1` invariant exponents of the `pr`-ordering `po`. If `trunc = -1`, it is set to `#po`.

1.1.2 r-removed $\mathfrak{p}$-orderings of finite sets

`GEN rpord_e(GEN nf, GEN pr, GEN S, long r, long trunc)`

Return in a vector the first `trunc-1` invariant exponents of any `r`-removed `pr`-ordering of the set `S`. If `trunc = -1`, it is set to `#S*(r+1)`.

The gp prototype is `rpord_e(nf, pr, S, r, {trunc = -1})`

Examples

```
? K = nfinit(x^3 + 2);
? pr = idealprimedec(K, 2)[1];
? S = [1, 3, x, x + 2, -3*x, 3*x + 8, -2*x + 3, 3*x - 7, 8*x, 2*x + 3];
? for(r=1, 3, print(rpord_e(K, pr, S, r)))    /* r = 1, 2, 3 */
[0, 0, 0, 1, 1, 2, 2, 5, 5]
[0, 0, 0, 0, 0, 1, 1, 2, 2]
[0, 0, 0, 0, 0, 0, 0, 1, 1]

? rpord_e(K, pr, S, 0) == pord_e(K, pr, S)    /* r = 0 is equivalent to a pr-ordering */
% = 1
```

`GEN rpord(GEN nf, GEN pr, GEN S, long r, long trunc, GEN *ex, GEN *inv)`

Return in a vector the first `trunc` elements of a `r`-removed `pr`-ordering of the set `S`. If `trunc = -1`, it is set to `#S*(r+1)`. Set to `*ex` the extraction small vector and to `*inv` the invariant vector as it would be returned by `rpord_e(nf, pr, S, r, trunc)`.

The gp prototype is `rpord(nf, pr, S, r, {trunc = -1}, {&ex = NULL}, {&inv = NULL})`

Examples

```
? K = nfinit(x^3 + 2);
? pr = idealprimedec(K, 2)[1];
? S = [1, 3, x, x + 2, -3*x, 3*x + 8, -2*x + 3, 3*x - 7, 8*x, 2*x + 3];

? for(r=1, 3, print(rpord(K, pr, S, r, 6)))    /* r = 1, 2, 3 */
[1, 3, x, x + 2, 3*x - 7, 8*x]
[1, 3, -2*x + 3, x, x + 2, -3*x]
[1, 3, -2*x + 3, 3*x - 7, x, x + 2]

? rpord(K, pr, S, 1,, &ex, &E); print(ex); print(E);
Vecsmall([1, 2, 3, 4, 8, 9, 8, 9, 2, 4])    /* note that there is repetitions */
[0, 0, 0, 1, 1, 2, 2, 5, 5]
```

`int isrpord(GEN nf, GEN pr, GEN S, long r, long trunc, GEN *i)`

Return 1 if the sequence of the first `trunc` elements of `S` is the beginning of a `r`-removed `pr`-ordering of the set `S`, 0 otherwise. If `trunc = -1`, it is set to `#S*(r+1)`. Set to `*i` the index of the first element of `S` responsible for failure.

The gp prototype is `isrpord(nf, pr, S, r, {trunc = -1}, {&i = NULL})`

Examples

```
? K = nfinit(x^3 + 2);
? pr = idealprimedec(K, 2)[1];
? S = [1, 3, x, x + 2, -3*x, 3*x + 8, -2*x + 3, 3*x - 7, 8*x, 2*x + 3];

? for(r=0, 4, print(isrpord(K, pr, S, r)))    /* r = 0,1,2,3,4 */
0
```

```
0
0
1
1

? isrpord(K, pr, S, 2,, &i)
% = 0
? isrpord(K, pr, S, 2, i)
% = 1
```

GEN rpord_get_e(GEN nf, GEN pr, GEN rpo, long r, long trunc)

Return in a vector the first `trunc - 1` invariant exponents of the `r`-removed `pr`-ordering `rpo`. If `trunc = -1`, it is set to `#rpo`.

1.1.3 $\mathfrak{p}$-orderings of order h of finite sets

GEN opord_e(GEN nf, GEN pr, GEN S, long h, long trunc)

Return in a vector the first `trunc-1` invariant exponents of any `pr`-ordering of order `h` of the set `S`. If `trunc = -1`, it is set to `#S`.

The gp prototype is `opord_e(nf, pr, S, h, {trunc = -1})`

Examples

```
? K = nfinit(x^2 - 2);
? pr = idealprimedec(K, 2)[1];
? S = [[1, 1]~, [1, 2]~, [-3, 1]~, [-1, 0]~, [2, 2]~, [0, 2]~];

? for(h=1, 4, print(opord_e(K, pr, S, h)))    /* h = 1,2,3,4 */
[0, 1, 1, 2, 3]
[0, 1, 2, 3, 4]
[0, 1, 3, 3, 5]
[0, 1, 3, 3, 6]

? opord_e(K, pr, S, 10000) == pord_e(K, pr, S)
% = 1    /* when h is large enough, a pr-ordering of order h is just a pr-ordering */
```

GEN opord(GEN nf, GEN pr, GEN S, long h, long first, long trunc, GEN *ex, GEN *inv)

Return in a vector beginning by `S[first]` the first `trunc` elements of a `pr`-ordering of order `h` of the set `S`. If `trunc = -1`, it is set to `#S`. Set to `*ex` the extraction small vector and to `*inv` the vector of invariants as it would be returned by `opord_e(nf,npr, S, r, trunc)`.

The gp prototype is `opord(nf, pr, S, h, {first=1}, {trunc = -1}, {&ex = NULL}, {&inv = NULL})`

Examples

```
? K = nfinit(x^2 - 2);
? pr = idealprimedec(K, 2)[1];
? S = [[1, 1]~, [1, 2]~, [-3, 1]~, [-1, 0]~, [2, 2]~, [0, 2]~];

? for(h=1, 3, print(opord(K, pr, S, h, 3, 5)))    /* h = 1,2,3 */
[[-3, 1]~, [1, 2]~, [1, 1]~, [-1, 0]~, [2, 2]~]
[[-3, 1]~, [1, 2]~, [2, 2]~, [-1, 0]~, [1, 1]~]
[[-3, 1]~, [1, 2]~, [2, 2]~, [0, 2]~, [-1, 0]~]

? opord(K, pr, S, 3,,, &ex, &inv); print(ex); print(inv);
Vecsmall([1, 2, 5, 6, 4, 3])
[0, 1, 3, 3, 5]
```

`int isopord(GEN nf, GEN pr, GEN S, long h, long trunc, GEN *i)`

Return 1 if the sequence of the first `trunc` elements of `S` is the beginning of a `pr`-ordering of order `h` of the set `S`, 0 otherwise. If `trunc = -1`, it is set to `#S`. Set to `*i` the index of the first element of `S` responsible for failure.

The gp prototype is `isopord(nf, pr, S, h, {trunc = -1}, {&i = NULL})`

Examples

```
? K = nfinit(x^2 - 2);
? pr = idealprimedec(K, 2)[1];
? S = [[1, 1]~, [1, 2]~, [-3, 1]~, [-1, 0]~, [2, 2]~, [0, 2]~];

? for(h=1, 3, print(isopord(K, pr, S, h)))    /* h = 1,2,3 */
1
1
0

? ispord(K, pr, opord(K, pr, S, 1000))
1
```

`GEN opord_get_e(GEN nf, GEN pr, GEN opo, long h, long trunc)`

Return in a vector the first `trunc - 1` invariant exponents of the `pr`-ordering of order `h` `opo`. If `trunc = -1`, it is set to `#opo`.

1.1.4 Miscellaneous

This section present a fonction to compute all $\mathfrak{p}$-orderings of a finite set and functions to deal with simultaneous orderings.

The function `allpord` allows to compute all $\mathfrak{p}$-orderings beginning by a choosen subsequence. There are serious limitations on doing this. For example, for any $m \in \mathbb{Z}$, there exist sets of cardinal m with 2^m $\mathfrak{p}$-orderings.

If you are interested only in the number of $\mathfrak{p}$-orderings of a finit set and not the orderings themself, there are bounds and exact formula in certain cases. See [Joh09] for example.

`GEN allpord(GEN nf, GEN pr, GEN S, GEN SS, long trunc, GEN *ex)`

Return in a vector of vectors all sequences of `trunc` elements beginning by the subsequence `SS` which are the beginning of a `pr`-orderings of the set `S`. Set to `*ex` all the extraction small vectors. The argument `SS` can also be a `t_INT` i which is interpreted as if `SS` = `[S[i]]`.

The gp prototype is `allpord(nf, pr, S, SS, {trunc=-1}, {&ex=NULL})`

Examples

```
? K = nfinit(x^2 + 1);
? pr = idealprimedec(K, 2)[1];
S = [[4, 0]~, [1, -1]~, [1, 1]~, [2, 2]~, [2,-4]~, [0,7]~, [1,-2]~, [1,-67]~, [-5,2]~]

? #allpord(K, pr, S, [])    /* number of pr-ordering of S */
% = 11904

? v = vector(#S);
? for(i=1, #S, v[i] = #allpord(K, pr, S, i)); print(v);
[912, 1048, 900, 912, 1280, 2112, 1920, 900, 1920]

? for(i=1, #S, v[i] = #allpord(K, pr, S, S[1..i])); print(v);
[912, 1048, 900, 912, 1280, 2112, 1920, 900, 1920]

? SS = vecextract(S, [2,6,1]);
? #allpord(K, pr, S, SS)
% = 48
? for(i=1, #S, v[i] = #allpord(K, pr, S, SS, i)); print(v);
[0, 0, 1, 6, 22, 56, 92, 88, 48]

? #allpord(K, pr, S, 1)
% = 912
? for(i=1, #S, v[i] = #allpord(K, pr, S, 1, i)); print(v);
[0, 3, 21, 101, 343, 884, 1538, 1644, 912]
```

A simultaneous ordering of a set is a $\mathfrak{p}$-ordering for all $\mathfrak{p}$ simultaneously. Such orderings do not always exist. The function below will return one if possible.

`GEN simulord(GEN nf, GEN S, long trunc, GEN *ex)`

Return in a vector the first `trunc` elements of a possible simultaneous ordering (or Newton sequence) of the set `S`. Set to `*ex` the extraction small vector. If no such sequence exists, return an empty vector.

The gp prototype is `simulord(nf, S, {trunc = -1}, {&ex = NULL})`

Examples

```
? K = nfinit(x^2 + 1);
    ? S = [4, x + 1, -x + 1, 2*x + 2];
    ? simulord(K, S, 3, &ex)
    % = [x + 1, 2*x + 2, 4]

    ? A = [3, x + 2, -2*x, 3*x + 1]
    ? simulord(K, A)
    % = []    /* there is no simultaneous ordering of A */
```

`GEN issimulord(GEN nf, GEN S, long trunc, GEN *i)`

Return 1 if the sequence of the first `trunc` elements of `S` could be the beginning of a simultaneous ordering of the set `S`, 0 otherwise. Set to `*i` the index of the first element responsible for failure.

The gp prototype is `issimulord(nf, S, {trunc = -1}, {&i = NULL})`

Examples

```
    ? K = nfinit(x^2 - 5);
    ? S = [4, x + 1, -x + 1, -2, 2*x + 2];

    ? issimulord(K, S,, &i)
    % = 0    /* S is not a simultaneous ordering */
    ? i
    % = 4    /* S[5] = -2 is responsible */
    ? issimulord(K, S, 3)
    % = 1
```

`GEN allsimulord(GEN nf, GEN S, GEN SS, long trunc, GEN *ex)`

Return in a vector of vectors all sequences of `trunc` elements beginning by the subsequence `SS` which could be the beginning of a simultaneous ordering of the set `S`. In particular, `allsimulord(nf, S, SS, #S)` will return all simultaneous ordering os `S` beginning by `SS`. Set to `*ex` all the extraction small vectors. The argument `SS` can also be a `t_INT` i which is interpreted as if `SS` = `[S[i]]`.

The gp prototype is `allsimulord(nf, S, SS, {trunc=-1}, {&ex=NULL})`

Examples

```
    ? S = [4, x + 1, -x + 1, -2, 2*x + 2];

    ? #allsimulord(K, S, [])
    % = 4    /* there is 4 simultaneous ordering of S */
```

```
? #allsimulord(K, S, 2)
% = 2    /* two of them start by x + 1 */
? #allsimulord(K, S, 5)
% = 2    /* and the two others start by 2*x + 2 */
```

1.2 Factorial ideals of finite sets in $\mathbb{Z}_K$

Functions to compute factorial ideals of finite sets associated with the three type of orderings handled by the library.

If you are interested in all such ideals up to a certain bound n, it is better to use the dedicated functions (ending by `_vec`) because they compute the needed prime ideal products and invariants once and for all which make them significantly faster than building the ideals one by one.

If you are interested in the norm of a factorial ideal, there are also dedicated functions that should be preferred over building the ideal and calling `idealnorm`.

1.2.1 Regular factorial ideals of finite sets

Regular factorial ideals of finite sets are not zero up to `#S-1`.

`GEN sfact(GEN nf, GEN S, long k)`

Return the `k`-th factorial ideal of the set `S`.

Examples

```
? K = nfinit(x^3 + 1);
? S = [3, x, x+1, x^2 + x, -3*x + 2, x^2 + 2*x + 1];
? sfact(K, S, 4)
% = [1290 708 338]  /* the 4-th factorial ideal in HNF */
    [   0   6   2]
    [   0   0   2]
```

`GEN sfact_vec(GEN nf, GEN S, long n)`

Return in a vector the first `n` factorial ideals of the set of algebraic integers `S`. If `n=-1`, it is set to `#S-1`. This function is faster than building such a vector by incremental calls to the function `sfact`.

The gp prototype is `sfact_vec(nf, S, {n=-1})`

Examples

```
? K = nfinit(x^3 + 1);
? S = [3, x, x+1, x^2 + x, -3*x + 2, x^2 + 2*x+ 1];

? for(i=1, #S, sfact(K, S, i) );
time = 20 ms.
? sfact_vec(K, S, #S);
time = 7 ms.    /* faster */
```

`GEN sfactnorm(GEN nf, GEN S, long k)`

Return the norm of the `k`-th factorial ideal of the set `S`.

`GEN sfactnorm_vec(GEN nf, GEN S, long n)`

Return in a vector the norms of the first `n` factorial ideals of the set `S`. If `n=-1`, it is set to `#S-1`. This function is faster than building the vector by incremental calls to `sfactnorm`.

The gp prototype is `sfactnorm_vec(nf, S, {n=-1})`

Examples

```
? K = nfinit(polcyclo(5));
? S = [1, 2, 3, 4, x, 2*x, x^3 + 9, x^3 + x, 78]

? for(i=1, 8, sfactnorm(K,S,i))
time = 109 ms.

? sfactnorm_vec(K, S, 8);
time = 18 ms.    /* much faster */
```

`GEN sfactexp(GEN nf, GEN S, GEN x)`

Return the `S`-exponential of `x`.

1.2.2 `r`-removed factorial ideals of finite sets

The r-removed factorial ideals of a finite set are not zero up to `k = ((r+1)*#S)-1`.

`GEN sremfact(GEN nf, GEN S, long r, long k)`

Return the `k`-th `r`-removed factorial ideal of the set `S`.

Examples

```
? K = nfinit(polcyclo(5));
? S = [1, 2, 3, 4, x, 2*x, x^3 + 9, x^3 + x, 78];

? sremfact(K, S, 2, 18)    /* r = 2 */
[44 8 28 32]
[ 0 4  0  0]
[ 0 0  4  0]
[ 0 0  0  4]

? sremfact(K, S, 2, (2+1)*#S - 1)
[12855002631049216 12832915842478080 5122269399859200 8704793309683712]
[                0              4096                0                0]
[                0                 0             4096                0]
[                0                 0                0             4096]
```

`GEN sremfact_vec(GEN nf, GEN S, long r, long n)`

Return in a vector the first `n` `r`-removed factorial ideals of the set `S`. If `n = -1`, it is set to `((r+1)*#S)-1`. Faster than building the vector by incremental calls to `sremfact`.

The gp prototype is `sremfact_vec(nf, S, r, {n=-1})`

Examples

```
? K = nfinit(x^3 + 2);
? S = [[1, 0, 1]~, [2, 0, 2]~, [0, -3, 1]~, [2, -2, 3]~, [1, 2, -3]~, [5, 0, 0]~];

? for(i=1,(2*#S - 1), sremfact(K, S, 1, i))
time = 71 ms.

? sremfact_vec(K, S, 1);
time = 13 ms.    /* faster */
```

`GEN sremfactnorm(GEN nf, GEN S, long r, long k)`

Return the norm of the `k`-th `r`-removed factorial ideal of the set `S`.

`GEN sremfactnorm_vec(GEN nf, GEN S, long r, long n)`

Return in a vector the norms of the n `first` `r`-removed factorial ideals of the set `S`. If `n = -1`, it is set to `((r+1)*#S)-1`. Faster than building the vector by incremental calls to `sremfactnorm`.

The gp prototype is `sremfactnorm_vec(nf, S, r, {n=-1})`

Examples

```
? K = nfinit(x^3 + 2);
? S = [[1, 0, 1]~, [2, 0, 2]~, [0, -3, 1]~, [2, -2, 3]~, [1, 2, -3]~, [5, 0, 0]~];

? for(i=1,(2*#S - 1), sremfactnorm(K, S, 1, i))
time = 64 ms.

? sremfactnorm_vec(K, S, 1);
time = 14 ms.    /* faster */
```

1.2.3 Factorial ideals of modulus M of finite sets

A modulus is a two column matrix with prime ideals in the first column and integers in the second, as it is returned by `idealfactor`. In particular, any algebraic integer represents a modulus.

Like regular factorial ideals of finite sets, factorial ideals of modulus M of finite sets are not zero up to `#S - 1`.

`GEN sfactmod(GEN nf, GEN S, GEN M, long k)`

Return the `k`-th factorial ideal of modulus `M` of the set `S`.

Examples

```
? K = nfinit(x^3 + 2);
? S = [[1, 0, 1]~, [2, 0, 2]~, [0, -3, 1]~, [2, -2, 3]~, [1, 2, -3]~, [5, 0, 0]~];
? M = idealfactor(K, [-1,3,0]~);    /* some modulus */

? sfactmod(K, S, M, 4)
% = [25 3 16]
    [ 0 1  0]
    [ 0 0  1]
```

`GEN sfactmod_vec(GEN nf, GEN S, GEN M, long n)`

Return in a vector the `n` first factorial ideals of modulus `M` of the set `S`. If `n = -1`, it is set to `#S - 1`. Faster than building the vector by incremental calls to `sfactmod`.

The gp prototype is `sfactmod_vec(nf, S, M, {n=-1})`

Examples

```
? K = nfinit(polcyclo(13));
? S = [1, 7, x^6, x^7 + 3*x, x^11 + x^10 + 1, 12*x^3 + 2, x^8 + 1, x^4 + 3*x];
? M = idealfactor(K, 3);    /* some modulus */

? for(i=1, #S-1, sfactmod(K, S, M, i))
time = 38 ms.

? sfatcmod_vec(K, S, M)
time = 24 ms.    /* faster */
```

`GEN sfactmodnorm(GEN nf, GEN S, GEN M, long k)`

Return the norm of the `k`-th factorial ideal of modulus `M` of the set `S`.

`GEN sfactmodnorm_vec(GEN nf, GEN S, GEN M, long n)`

Return in a vector the norms of the first `n` factorial ideals of modulus `M` of the set `S`. If `n = -1`, it is set to `#S - 1`. Faster than building the vector by incremental calls to `sfactmodnorm`.

The gp prototype is `sfactmodnorm_vec(nf, S, M, {n = -1})`

Examples

```
? K = nfinit(polcyclo(13));
? S = [1, 7, x^6, x^7 + 3*x, x^11 + x^10 + 1, 12*x^3 + 2, x^8 + 1, x^4 + 3*x];
? M = idealfactor(K, 3);    /* some modulus */
? for(i=1, #S-1, sfactmodnorm(K, S, M, i))
time = 16 ms.

? sfactmodnorm_vec(K, S, M);
time = 2 ms.    /* much faster */
```

2 Orderings and factorial ideals of $\mathbb{Z}_K$

Functions to compute three type of ordering of the whole ring of algebraic integer of a number field, namely :

- $\mathfrak{p}$-ordering
- r-removed $\mathfrak{p}$-ordering
- $\mathfrak{p}$-ordering of order h

The precise and original definitions can be found in [Bha97] and [Bha09]. The first type is actually a particular case of both the second and third one for $r = 0$ and $h = \infty$ respictively, but it is convenient to have simpler dedicated functions.

To build the three precited kind of orderings and compute their invariant sequence, it is needed to take a minimum over an infinite set. But exact formula in the local case and good behavior under localisation ([CCS97]) permit to compute directly the invariant sequence. This justify the following definition : a finite ordering (of any three kind) of length n of $\mathbb{Z}_K$ is a sequence of length n of algebraic integers sharing the same invariants as the sequence of the first n elements of any ordering of $\mathbb{Z}_K$.

There are two different things you can be interested in related to these orderings.

The first one is the ordering itself, meaning the sequence of algebraic integers that realizes the ordering. For this purpose, the library provides the three functions `zkpord`, `zkrpord` and `zkopord` (`r` for removed and `o` for order). They return an ordering of $\mathbb{Z}_K$ of requested length.

The second one is the ordering's invariant. For this purpose, the library provides the three functions `legf_vec`, `rlegf_vec` and `olegf_vec`. They return an increasing vector of integers which are the exponents of the truncated invariant sequence associated to any ordering of $\mathbb{Z}_K$.

Consequently, the invariant's vector of the ordering returned by any of `zkpord`, `zkrpord` and `zkopord` will always equal the corresponding vector returned by `legf_vec`, `rlegf_vec` and `olegf_vec` respectively:

```
? K = nfinit(x^2 + 1);
? pr = idealprimedec(K, 2)[1]
? q = idealnorm(K, pr);

? S = zkpord(K, pr, 10);
? pord_e(K, pr, S) == legf_vec(q, 9)
% = 1

? S = zkrpord(K, pr, 2, 10);
? rpord_e(K, pr, S, 2) == rlegf_vec(q, 2, 9)
% = 1

? S = zkopord(K, pr, 1, 10);
? opord_e(K, pr, S, 1) == olegf_vec(q, 1, 9)
% = 1
```

2.1 Orderings of $\mathbb{Z}_K$

2.1.1 Regular $\mathfrak{p}$-ordering of $\mathbb{Z}_K$

`GEN legf(GEN q, GEN n)`

Generalised Legendre formula. If $v_q(n)$ is the exponent of the highest power of $q \geq 2$ dividing n, the function computes $w_q(n) = \sum_{i=1}^{n} v_q(i)$. If `q` is prime, this is just the `q` valuation of `n!`.

`GEN legf_vec(GEN q, GEN n)`

Return the vector `[legf(q, k)]`, $1 \leq k \leq n$.

`GEN zkpord(GEN nf, GEN pr, long n)`

Return a `pr`-ordering of length `n` of `nf`. Actually, the ordering will be a strong `pr`-ordering, but if you intend to build one, `strongpord` (see below) is more convenient. The argument `pr` can be a single prime ideal or a vector of prime ideals. In this case, the function returns a vector of vectors which are $\mathfrak{p}$-ordering for each corresponding prime.

Examples

```
? K = nfinit(x^3 - 6*x^2 + 1);
? pr = idealprimedec(K, 5)[1];
? v = zkpord(K, pr, 10)
% = [0, 1, 2, 3, 4, [0, 0, 1]~, [1, 0, 1]~, [2, 0, 1]~, [3, 0, 1]~, [4, 0, 1]~]
? ispord(K, pr, v)
% = 1
? isstrongpord(K, pr, v)
% = 1
```

A strong ɪ-ordering (also called *very well distributed and ordered sequence* is a ɪ-ordering statisfying a condition of regularity for the distribution of its elements in the different classes modulo $\mathfrak{p}^k$ for all k. See [CCS97].

`GEN strongpord(GEN nf, GEN pr, long n)`

Return a strong `pr`-ordering of length `n` of `nf`. The argument `pr` can be a single prime ideal or a vector of prime ideals in which case the returned sequence will be a strong p-ordering for every prime ideal p in the vector `pr`.

Examples

```
? K = nfinit(x^2 + 3);
? pr = idealprimedec(K,2)[1];    /* a single prime */
? S = strongpord(K, pr, 5)
% = [0, [-1, 0]~, [0, -1]~, [-1, -1]~, [-2, 0]~]
? ispord(K, pr, S)
% = 1
```

```
? pr = idealfactor(K, 2*3*5)[,1];    /* a vector of primes */
? S = strongpord(K, pr, 5)
% = [0, [1, 0]~, [-8, -5]~, [13, 5]~, [14, 20]~]
? isstrongpord(K, pr[1], S)
% = 1
? isstrongpord(K, pr[2], S)
% = 1
```

`int isstrongpord(GEN nf, GEN pr, GEN S)`

Return 1 if the sequence `S` is a strong `pr`-ordering of length #`S`, 0 otherwise. The argument `pr` can be a single prime ideal or a vector of prime ideals in which case the function return 1 if `S` is a strong p-ordering for every prime ideal p in the vector `pr`.

2.1.2 r-removed $\mathfrak{p}$-orderings of $\mathbb{Z}_K$

`GEN rlegf(GEN q, GEN n, long r)`

Analogous of Legendre formula for `r`-removed $\mathfrak{p}$-ordering.
The function return $\texttt{legf}(q,n) - \texttt{legf}(q, \left\lfloor \frac{n}{q^k} \right\rfloor) - kr$ where $k = \left\lfloor \frac{\log \frac{n}{r}}{\log q} \right\rfloor$.

`GEN rlegf_vec(GEN q, long r, GEN n)`

Return the vector `[rlegf(q,r,k)]`, $1 \le k \le n$.

`GEN zkrpord(GEN nf, GEN pr, long r, long n)`

Return a `r`-removed `pr`-ordering of length `n` of `nf`.

2.1.3 $\mathfrak{p}$-orderings of order h of $\mathbb{Z}_K$

`GEN olegf(GEN q, GEN n, GEN h)`

Analogous of Legendre formula for $\mathfrak{p}$-orderings of order `h`.
The function return $\texttt{legf}(q,n) - \texttt{legf}(q, \left\lfloor \frac{n}{q^h} \right\rfloor)$.

`GEN olegf_vec(GEN q, GEN h, long n)`

Return the vector `[olegf(q, k, h)]`, $1 \le k \le n$.

`GEN zkopord(GEN nf, GEN pr, long h, long n)`

Return a `pr`-ordering of order `h` of length `n` of `nf`.

2.2 Factorial ideals of $\mathbb{Z}_K$

Functions to compute factorial ideals of $\mathbb{Z}_K$ associated with the three type of orderings handled by the library.

If you are interested in all such ideals up to a certain bound n, it is better to use the dedicated functions (ending by `_vec`) because they compute the needed prime ideal products and invariants once and for all which make them significantly faster than building the ideals one by one.

If you are interested in the norm of a factorial ideal, there are also dedicated functions that should be preferred over building the ideal and calling `idealnorm`.

2.2.1 Regular factorial ideals of $\mathbb{Z}_K$

`GEN zkfact(GEN nf, long k)`

Return the `k`-th factorial ideal of `nf`.

Examples

```
? K = nfinit(x^2 + 5);
? zkfact(K, 3);
% = [6 3]
    [0 3]
```

`GEN zkfact_vec(GEN nf, long n)`

Return in a vector the `n` first factorial ideals of `nf`. This function is faster than building the vector by incremental calls to the function `zkfact`.

Examples

```
? K = nfinit(x^2 + 5);

? for(i=1, 100, zkfact(K,i))
time = 114 ms.

? zkfact_vec(K, 100);
time = 55 ms.    /* faster */
```

`GEN zkfactnorm(GEN nf, long k)`

Return the norm of the `k`-th factorial ideal of `nf`. This function is faster than calling `idealnorm(nf, zkfact(nf, k))`.

Examples

```
? K = nfinit(x^2 + 1);

? for(i=1, 200, idealnorm(K, zkfact(K, i)));
time = 185 ms.
```

```
? for(i=1, 200, zkfactnorm(K, i));
time = 107 ms.    /* faster */
```

`GEN zkfactnorm_vec(GEN nf, long n)`

Return in a vector the norm of the `n` first factorial ideals of `nf`. Faster than calling `zkfact_vec` and then computing the norms.

2.2.2 r-removed factorial ideals of $\mathbb{Z}_K$

`GEN zkremfact(GEN nf, long r, long k)`

Return the `k`-th `r-removed` factorial ideal of `nf`.

Examples

```
? K = nfinit(x^2 + 1);
? for(r=0, 4, print(zkremfact(K, r, 20)))    /* r = 0,1,2,3,4 */
[636480000, 0; 0, 636480000]
[48000, 0; 0, 48000]
[1600, 800; 0, 800]
[160, 80; 0, 80]
[16, 8; 0, 8]
```

`GEN zkremfact_vec(GEN nf, long r, long n)`

Return in a vector the `n` `r`-removed factorial ideals of `nf`. Faster than building the vector by incremental calls to `zkremfact`.

Examples

```
? K = nfinit(polcyclo(15));

? for(i=1, 100, zkremfact(K, 2, i))
% = time = 470 ms.

? zkremfatc_vec(K, 2, 100);
% = time = 44 ms.    /* faster */
```

`GEN zkremfactnorm(GEN nf, long r, long k)`

Return the norm of the `k`-th `r`-removed factorial ideal of `nf`.

`GEN zkremfactnorm_vec(GEN nf, long r, long n)`

Return in a vector the norms of the first `n` `r`-removed factorial ideals of `nf`. Faster than building the vector by incremental calls to `zkremfactnorm`.

2.2.3 Factorial ideals of modulus M of $\mathbb{Z}_K$

`GEN zkfactmod(GEN nf, GEN modulus, long k)`

Return the `k`-th factorial ideal of modulus `M` of `nf`.

Examples

```
? K = nfinit(x^4 + 2*x + 2);
? M = idealfactor(K, 2*3*5);    /* some modulus */
? zkfactmod(K, M, 10)
% = [100 16 32 36]
    [  0  4  0  0]
    [  0  0  4  0]
    [  0  0  0  4]
```

`GEN zkfactmod_vec(GEN nf, GEN modulus, long n)`

Return in a vector the `n` first factorial ideals of modulus `M` of `nf`. Faster the building the vector by incremental calls to `zkfactmod`.

Examples

```
? K = nfinit(x^4 + 2*x + 2);
? M = idealfactor(K, 2*3*5);    /* some modulus */

? for(i=1, 1000, zkfactmod(K, M, i))
time = 5,667 ms.

? zkfactmod_vec(K, M, 1000);
time = 433 ms.    /* much faster */
```

`GEN zkfactmodnorm(GEN nf, GEN modulus, long k)`

Return the norm of the `k`-th factorial ideal of modulus `M` of `nf`.

`GEN zkfactmodnorm_vec(GEN nf, GEN modulus, long n)`

Return in a vector the norms of the first `n` factorial ideals of modulus `M` of `nf`. Faster than building the vector by incremental calls to `zkfactmodnorm`.

Examples

```
? K = nfinit(x^4 + 2*x + 2);
? M = idealfactor(K, 2*3*5);    /* some modulus */

? for(i=1, 1000, zkfactmodnorm(K, M, i))
time = 42 ms.

? zkfactmodnorm_vec(K, M, 1000);
time = 33 ms.    /* faster */
```

2.3 Almost strong simultaneous ordering of $\mathbb{Z}_K$

The existence of simultaneous oderings of the ring of integers of a number field is an open question, but it is believed and conjectured that they do not exist. In [CC18, Theorem 4.6], the authors present an inductive procedure to build a sequence of algebraic integers in any number field that is really close to be a simultaneous ordering, so close that they call such a sequence an almost strong simultaneous ordering. The function `zkalmostsso` implements this procedure as it appears in the precited paper. A particular use of the function is to compute n-universal sets of $\mathbb{Z}_K$ (see [CC18, Definition 1.1]). Unfortunatly, the recursive use of the chinese remainder theorem makes the function impraticable for large n.

`GEN zkalmostsso(GEN nf, long n, GEN a0, GEN ipr`

Return an almost strong simultaneous ordering of length `n` starting by `a0`, i.e a sequence of length `n` of algebraic integers in `nf` satisfying the two following property:

1. for every prime ideal *pr*, the sequence obtained by slicing at most one element (depending on *pr*) is a strong *pr*-ordering of length `n-1` (or `n` if no slice happened)

2. every subsequence of $k+2$ consecutive terms of the sequence is a k-universal set of zk.

The argument `ipr` (for **i**nitial **pr**imes) can be a single prime ideal or a vector (`t_VEC` or `t_COL`) of prime ideals (possibly empty), those one for which `a0` might have to be sliced to satisfy the first property.

In particular, the following returns a n-universal set of zk : `zkalmostsso(nf, n + 2)`

The gp prototype is `zkalmostsso(nf, n, {a0 = 0}, {ipr = NULL})`

Examples

```
? K = nfinit(x^2 + 1);

? zkalmostsso(K, 5, [1,1]~)     /* length 5, starting by 1 + i */
% [[1, 1]~, [2, 1]~, [0, 0]~, [-1, 1]~, [0, 1]~]

? pr = idealfatcor(K, 2*3)[,1];    /* some prime ideals */
? zkalmostsso(K, 5,, pr)
% =  [0, 1, [2, 0]~, [0, 1]~, [1, 1]~]
```

The following return a 4-universal set of $\mathbb{Z}[i]$.

```
? A = zkalmostsso(K, 6); 0 */
% = [0, 1, 3, [2, 0]~, [0, 1]~, [4, 10]~]
```

Every polynomial in $\mathbb{Q}(i)[X]$ of degree at most 4 which is integer valued on A is integer valued on the whole $\mathbb{Z}[i]$.

`int zkissimulord(GEN nf, GEN S)`

Return 1 if the sequence `S` is a simultaneous ordering of length $\#$`S` $- 1$ of `nf`, 0 otherwise.

Examples

```
? K = nfinit(x^2 + 1);
? S = [0, 1, x, x +1];
? zkissimulord(K, S)
% = 1
```

Since `S` is a simultaneous ordering, we can compute a generator for the first $\#$`S` $- 1$ factorial ideals of `nf`.

```
? idealhnf(K, vdiffprod_i(K, S, 4)) == zkfact(K,3)
% = 1
```

3 Regular basis for modules of integer valued polynomials

To orderings and generalised factorial functions of $\mathbb{Z}_K$ are naturally associated subrings of the ring of integer-valued polynomial of a number field ([Bha09], [BCY09]). If we restrain those rings to polynomials of degree at most n, we obtain zk-modules, and the present section provides functions to compute regular basis for those modules : `zkregbasis`, `zkremregbasis` and `zkmodregbasis`. Being a *regular* basis means that the basis is composed of polynomials of each degree from 0 to n.

The considered modules do not always have a regular basis, and the library provides three functions to know if they do : `ispolyaupto`, `ispolyaupto_rem` and `ispolyaupto_mod`.

3.1 Regular basis for integer valued polynomials

`GEN zkfactpol(GEN nf, long k, const char *s, long cmode)`

Return a polynomial *pol* of degree `k` in $zk[X]$ such that $pol(zk)$ generates the `k`-th factorial ideal of `nf`. The variable name is set to `s`. The flag `cmode` tunes the returned polynomial coefficients : 0 for `t_POLMOD`, 1 for `t_POL`, 2 for `t_COL`.

The gp prototype is `zkfactpol(nf, k, s, {cmode = 1})`

Examples

```
? K = nfinit(x^2 + 1);
? P = zkfactpol(K, 4, "t")
% = t^4 + (2*x + 2)*t^3 + 3*x*t^2 + (x - 1)*t
```

Let's check on few examples that P indeed takes value in the 4-th factorial ideal of $\mathbb{Z}[i]$ when evaluated on integers.

```
? idealfactor(K, zkfact(K, 4))
% = [[2, [1, 1]~, 2, 1, [1, -1; 1, 1]] 3]

? t = variable(P);
? y = subst(P, t, 1 + x));
? idealfactor(K, y)
% = [  [2, [1, 1]~, 2, 1, [1, -1; 1, 1]] 3]    /* 3 >= 3, ok  */
    [[5, [2, 1]~, 1, 1, [-2, -1; 1, -2]] 1]

? y = subst(P, t, 2*x -3)
? idealfactor(K, y)
% = [  [2, [1, 1]~, 2, 1, [1, -1; 1, 1]] 4]    /* 4 >= 3, ok */
    [ [5, [-2, 1]~, 1, 1, [2, -1; 1, 2]] 1]
    [[5, [2, 1]~, 1, 1, [-2, -1; 1, -2]] 1]
```

`GEN zkfactpol_vec(GEN nf, long n, const char *s, long cmode)`

Return a vector v of length n+1 such that $v[i]$ = `zkfactpol(nf,i-1,s,cmode)`. This function is used by `zkregbasis` to build a regular basis if possible.

The gp prototype is `zkfactpol_vec(nf, n, s, {cmode = 1})`

`int ispolyaupto(GEN bnf, long n)`

Return 1 if all products of prime ideals of equal norm up to `n` are principal, 0 otherwise. This is the case if and only if the first `n` factorial ideals of `nf` are principal. This function can be used to test if `zkregbasis` or `zkregbasis_fermat` are callable for a given value of `n`.

Be aware that this function do not check if `nf` is a Polya field. Indeed, even if the function return true some factorial ideals > `n` may not be principal. In the quadratic case, the function `qispolya` does so.

Examples

```
? K = bnfinit(x^2 - 221);
? ispolyaupto(K, 3)
% = 1    /* zkregbasis(K, 3) is legit */
? ispolyaupto(K, 4)
% = 0    /* zkregbasis(K, 4) is not legit*/
```

`GEN zkregbasis(GEN bnf, long n, const char *s, long cmode)`

Return in a vector v of length n+1 a regular basis for the zk-module Int(n,X). It is the module of all integer valued polynomials in `bnf`[X] of degree at most `n`.

Being a *regular* basis means that $\deg(v[i]) = i - 1$ for $1 \leq i \leq n + 1$.

For such a basis to exist, it is **mandatory** that all factorial ideals of `bnf` up to `n` are principal and this can be checked with the function `ispolyaupto`.

If the later condition is not met, the behavior is undefined. The flag `cmode` tunes the returned polynomial coefficients : 0 for `t_POLMOD`, 1 for `t_POL`, 2 for `t_COL`.

The gp prototype is `zkregbasis(bnf, n, s, {cmode = 1})`

Examples

```
? K = bnfinit(x^2 + 1);
? ispolyaupto(K, 2)    /* is zkregbasis callable ? */
% = 1

? B = zkregbasis(K, 2, "t")[3]
% = (-1/2*x + 1/2)*t^2 + (-1/2*x + 1/2)*t
```

The polynomial $B \in \mathbb{Q}(i)[X]$ is integer valued, for example :

```
? t = variable(P);
? nfalgtobasis(K, subst(B, t, 2*x + 3)))
% = [11, 3]~
```

`GEN zkregbasis_fermat(GEN bnf, long n, const char *s, long cmode)`

Exactely the same as `zkregbasis` except that the basis is built using Fermat binomials instead of calling `zkfactpol_vec`. As a consequence, the function is slower and the basis can differ from the one returned by `zkregbasis`.

The gp prototype is `zkregbasis_fermat(bnf, n, s, {cmode = 1})`

Examples

```
? K = bnfinit(x^2 + 1);

? B1 = zkregbasis(K, 40, "t");
% = time = 1,045 ms.

? B2 = zkregbasis_fermat(K, 40, "t")
% = time = 16,190 ms.    /* much slower */
```

3.2 Regular basis for integr valued polynomials with divided differences

`GEN zkremfactpol(GEN nf, long r, long k, const char* s, long cmod)`

Return a polynomial *pol* of degree `k` in $zk[X]$ such that $pol(zk)$ generates the `k`-th `r`-removed factorial ideal of `nf`. The variable name is set to `s`. The flag `cmode` tunes the returned polynomial coefficients : 0 for `t_POLMOD`, 1 for `t_POL`, 2 for `t_COL`.

The gp prototype is `zkremfactpol(nf, r, k, s, {cmode = 1})`

Examples

```
? K = nfinit(x^3 -x + 2);
? P = zkremfactpol(K, 1, 5, "t")
% = t^5 + (x^2 + x + 2)*t^4 + (2*x^2 + 2*x + 1)*t^3 + (x^2 + x)*t^2
```

Let's check on few integers that P indeed takes value in the 5-th 1-removed factorial ideal of K :

```
? idealfactor(K, zkremfact(K, 1, 5))
% =
[[2, [0, 1, 0]~, 1, 1, [0, -2, 0; 0, 0, -2; 1, 0, -1]] 1]
[[2, [1, 1, 0]~, 2, 1, [1, -1, -2; 1, 1, -2; 1, 1, 0]] 1]

? t = variable(P);
? idealfactor(K, subst(P, t, x^2 regbasis_mod- 2*x + 1))
% =
[      [2, [0, 1, 0]~, 1, 1, [0, -2, 0; 0, 0, -2; 1, 0, -1]] 2] /* 2 >= 1, ok */
[      [2, [1, 1, 0]~, 2, 1, [1, -1, -2; 1, 1, -2; 1, 1, 0]] 5] /* 5 >= 1, ok */
[[11, [3, 1, 0]~, 1, 1, [-2, -5, 6; -3, -2, -2; 1, -3, -3]] 1]
```

```
? idealfactor(K, subst(P, t, -2))
% =
[[2, [0, 1, 0]~, 1, 1, [0, -2, 0; 0, 0, -2; 1, 0, -1]] 4] /* 4 >= 1, ok */
[[2, [1, 1, 0]~, 2, 1, [1, -1, -2; 1, 1, -2; 1, 1, 0]] 5] /* 5 >= 1, ok */
```

`GEN zkremfactpol_vec(GEN nf, long r, long n, const char *s, long cmode)`

Return a vector v of length `n+1` such that $v[i]$ = `zkremfactpol(nf,r,i-1,s,cmode)`. This function is used by `zkremregbasis` to build a regular basis if possible.

The gp prototype is `zkremfactpol_vec(nf, r, n, s, {cmode = 1})`

`GEN ispolyaupto_rem(GEN nf, long r, long n)`

Return 1 if the first `n` `r`-removed factorial ideals of `nf` are principal, 0 otherwise. This function can be used to check if `zkremregbasis` is callable.

`GEN nfX_divdiff(GEN nf, GEN pol, long k, GEN *vars)`

Return the `k`-th divided difference of the polynomial `pol` $\in$ `nf`$[X]$. The returned polynomial is in `nf`$[x_0, x_1, \ldots, x_n]$. If not `NULL`, set to `*vars` the vector of variables $[x_0, \ldots, x_k]$.

The gp prototype is `nfX_divdiff(nf, pol, k, {&vars = NULL})`

Examples

```
? K = nfinit(x^2 + 1);
? pol = zkfactpol(K, 3, "t", 0)
% = Mod(1, x^2 + 1)*t^3 + Mod(x + 2, x^2 + 1)*t^2 + Mod(x + 1, x^2 + 1)*t

? dd = nfX_divdiff(K, pol, 2, &v)
% = Mod(1, x^2 + 1)*x2 + (Mod(1, x^2 + 1)*x1 + (Mod(1, x^2 + 1)*x0 + Mod(x + 2, x^2 + 1)
? v
% = [x0, x1, x2]

? substvec(dd, v, [1+x, 2*x -1, 3])
% = Mod(4*x + 5, x^2 + 1)
```

`GEN zkremregbasis(GEN bnf, long r, long n, const char *s, long cmode)`

Return in a vector v of length `n+1` a regular basis for the zk-module Int(n, r, X).

It is the module of all integer-valued polynomials of `bnf`[X] of degree at most `n` such that their `r` first divided differences are also integer-valued.

Being a regular basis means that $\deg(v[i]) = i - 1$ for $1 \leq i \leq n + 1$.

For such a basis to exist, it is **mandatory** that the `n` first `r`-removed factorial ideals of `bnf` are principal and this can be checked with the function `ispolyaupto_rem`.

If the later condition is not met, the behavior is undefined. The flag `cmode` tunes the returned polynomial coefficients : 0 for `t_POLMOD`, 1 for `t_POL`, 2 for `t_COL`.

The gp prototype is `zkremregbasis(bnf, r, n, s, {cmode = 1})`

Examples

```
? K = bnfinit(x^2 + 7);
? ispolyaupto_rem(K, r, 4)    /* is zkremregbasis callable ? */
% = 1

/* integer valued polynomial who's 1st divided difference is integer valued */

? pol = zkremregbasis(K, 1, 4, "t", 0)[5]
% = Mod(-1/2*x + 1/2, x^2 + 1)*t^4 + Mod(-x + 1, x^2 + 1)*t^3 \
    + Mod(-1/2*x + 1/2, x^2 + 1)*t^2

/* compute the 1st divided difference and evaluate on integers */

? dd1 = nfX_divdiff(K.nf, pol, 1, &v);
? substvec(dd1, v, [2*x + 3, 1 + x]);
? nfalgtobasis(K, %)
% = [53, 59]~    /* integer */
```

3.3 Regular basis for integer valued polynomials with modulus

`GEN zkfactmodpol(GEN nf, GEN M, long k, const char *s, long cmode)`

Return a polynomial *pol* of degree `k` in $zk[X]$ such that $pol(zk)$ generates the `k`-th factorial ideal of modulus `M` of `nf`. The variable name is set to `s`. The flag `cmode` tunes the returned polynomial coefficients : 0 for `t_POLMOD`, 1 for `t_POL`, 2 for `t_COL`.

The gp prototype is `zkfactmodpol(nf, M, l, s, {cmode = 1})`

Examples

```
? K = nfinit(x^2 + 1);
? M = idealfactor(K, (3*x+1)^3)    /* some modulus */
? P = zkfactmodpol(K, M, 4, "t");
? t = variable(P);

? idealfactor(K, zkfactmod(K, M, 4))
% = [[2, [1, 1]~, 2, 1, [1, -1; 1, 1]] 3]

? idealfactor(K, subst(P, t, 2*x + 1))
? idealfactor(K, subst(P,t,-x+2))
% =
[      [2, [1, 1]~, 2, 1, [1, -1; 1, 1]] 3]    /* 3 >= 3, ok */
[     [5, [-2, 1]~, 1, 1, [2, -1; 1, 2]] 2]
[   [13, [5, 1]~, 1, 1, [-5, -1; 1, -5]] 1]
[[277, [-60, 1]~, 1, 1, [60, -1; 1, 60]] 1]
```

`GEN zkfactmodpol_vec(GEN nf, GEN M, long n, const char *s, long cmode)`

Return a vector v of length `n+1` such that $v[i]$ = `zkfactmodpol(nf,M,i-1,s,cmode)`. This function is used by `zkmodregbasis` to build a regular basis if possible.

The gp prototype is `zkfactmodpol_vec(nf, M, n, s, {cmode = 1})`

`GEN ispolyaupto_mod(GEN bnf, GEN M, long n)`

Return 1 if the first `n` factorial ideals of modulus `M` of `bnf` ar principal, 0 otherwise. This function can be used to check if `zkmodregbasis` is callable.

`GEN zkmodregbasis(GEN bnf, GEN M, long n, const char *s, long cmode)`

Return in a vector v of length `n+1` a regular basis for the zk-module Int(n, M, X).

It is the module of all integer-valued polynomials *pol* of `bnf`[X] of degree at most `n` such that if I_M is the ideal represented by the modulus `M` and $m \in I_M$, then $pol(mX + s) \in zk[X]$ for all $s \in zk$.

Being a regular basis means that $\deg(v[i]) = i - 1$ for $1 \leq i \leq n + 1$.

For such a basis to exist, it is **mandatory** that the `n` first factorial ideals of modulus `M` of `bnf` are principal and this can be checked with the function `ispolyaupto_mod`.

If the later condition is not met, the behavior is undefined. The flag `cmode` tunes the returned polynomial coefficients : 0 for `t_POLMOD`, 1 for `t_POL`, 2 for `t_COL`.

The gp prototype is `zkmodregbasis(nf, M, n, s, {cmode = 1})`

Examples

```
? K = bnfinit(x^2 + 1);
? M = idealfactor(K, x-3)    /* some modulus */
? ispolyaupto_mod(K, M, 2)    /* is zkmodregbasis callable ? */
% = 1

? B = zkmodregbasis(K, M, 2, "t", 0);
? P = B[3]    /* not in Z[i][X] */
% = Mod(-1/2*x + 1/2, x^2 + 1)*t^2 + Mod(1/2*x - 1/2, x^2 + 1)*t

? t = variable(P);
? m = (x-3)*(2*x-1);    /* some element in M */

? subst(P, t, m*t + x)    /* in Z[i][X] */
% = Mod(17*x - 31, x^2 + 1)*t^2 + Mod(-2*x + 11, x^2 + 1)*t + Mod(-1, x^2 + 1)
```

4 Simultaneous $\mathfrak{p}$-ordering in quadratic number fields

4.1 Factorial ideals of quadratic number fields

The functions in this section are specialisations to the quadratic case of their general counterpart presented in the previous sections. They are really faster and should be preferred when computing in quadratic number fields. The trick is that since the kronecker symbol tells everything about how primes split it is often possible to compute the result without actually decomposing primes or multilpying ideals, which save a lot of time.

`GEN qfact(GEN nf, long k)`

Return the `k`-th factorial ideal of the quadratic number field `nf`. This function is faster than calling `zkfact`.

Examples

```
    ? K = nfinit(x^2 + 3)
    ? for(i = 1, 1000, qfact(K, i))
    time = 680 ms.

    ? for(i = 1, 1000, zkfact(K, i))
    time = 2,620 ms.  /* much slower */
```

`GEN qfact_vec(GEN nf, long n)`

Return in a vector the first `n` factorial ideals of the quadratic number field `nf`.

`GEN qfactnorm(GEN nf, long k)`

Return the norm of the `k`-th factorial ideal of the quadratic number field `nf`. This function is faster than calling `idealnorm(nf,qfact(nf,k))` or `zkfactnorm(nf,k)`.

Examples

```
    ? K = nfinit(x^2 - 5)
    ? for(i = 1, 1000, qfactnorm(K, i))
    time = 259 ms.

    ? for(i = 1, 1000, zkfactnorm(K, i))
    time = 1,067 ms.  /* much slower */
```

`GEN qfactnorm_vec(GEN nf, long n)`

Returns in a vector the norms of the first `n` factorial ideals of the quadratic number field `nf`.

`int qispolya(GEN nf)`

Returns 1 if the quadratic number field `nf` is a Polya number field, 0 otherwise. If it is the case, `zkregbasis` is callable without restriction.

4.2 Simultaneous ordering of quadratic number fields

This section deals with simultaneous ordering in quadratic number fields and is closely related to [AC11]. The function prefix *qrso* stands for *quadratic real simultaneous ordering.*

`GEN qallsimulord(GEN nf, long n)`

Return in a vector of vectors all basal (i.e starting by `[0,1]`) simultaneous ordering of length `n` of the quadratic number field `nf` $= \mathbb{Q}(\sqrt{d})$.

To keep with the notation in [AC11], the length of the sequence $\{a_0, a_1, \ldots, a_n\}$ is n, so there are $n+1$ elements in a sequence of length n.

Let denote m_d the least non-splitting prime in $\mathbb{Q}(\sqrt{d})$ as returned by `qfirstnonsplit` and n_d the maximal length of a basal simultaneous ordering.

Now be aware that :

- if $d \neq 1 \pmod 8$, then $m_d = 2$ and $n_d = 1 = m_d - 1$ except for d = -3, -1, 2, 3, 5. For this 5 exceptions, the maximum lengths are 4, 3, 4, 5, 6 respectively, and there are 24, 4, 64, 16, 672 such orderings respectively again. You can check all this with the function !
- if $d = 1 \pmod 8$, then n_d is superior or equal to $m_d - 1$ and is exactely $m_d - 1$ in the imaginary case (for any d) and in the real case for d *large enough.* For $d = 17$, the maximum length is 4, there are 16 orderings and none of them is contained in $\mathbb{Z}$ (check it). Since $m_{17} = 3$, $d = 17$ provides an exception to the rule $n_d = m_d - 1$. If you run the function with $d > 17$ and $d = 1 \pmod 8$, you will observe that the 'rule' $n_d = m_d - 1$ seems always satisfied and that all found orderings are contained in $\mathbb{Z}$. The function `qrso_testfirstnonsplit` searches for orderings of length m_d such that the first $m_d - 1$ elements are contained in $\mathbb{Z}$.

Examples

```
? K = nfinit(x^2 + 3);  /* -3 is one of the exceptions */
? #qallsimulord(K,4)
% = 24
? #qallsimulord(K,5)
% = 0

? K = nfinit(x^2 - 17);  /* 17 do not follow the 'rule' */
? #qallsimulord(K,4)
% = 16
? x = qallsimulord(K,4)[1]
% = [0, 1, [-2, -1]~, [3, 1]~, [-5, -2]~]  /* not contained in Z */

? K = nfinit(x^2 - 97)  /* 97 */
? qfirstnonsplit(97)
% = 5
```

```
? #qallsimulord(K, 4)
% = 8
? #qallsimulord(K, 5)
% = 0  /* the 'rule' is respected */
? qallsimulord(K, 4)[1]
% = [0, 1, [-1, 0]~, [-2, 0]~, [-3, 0]~]  /* contained in Z */
```

`GEN qrso_testfirstnonsplit(GEN d)`

The argument `d` is a positive squarefree integer and represents the real quadratic number field $nf = \mathbb{Q}(\sqrt{d})$. Let m_d be the least prime who does not split in nf as returned by `qfirstnonsplit`. This function will test efficiently if there exist a basal simultaneous ordering of length m_d in $\mathbb{Q}(\sqrt{d})$ such that the first $m_d - 1$ terms are contained in $\mathbb{Z}$ and return in a vector the candidates if any, the empty vector otherwise. It is expected that there is no such sequence, except for the 3 real quadratic fields $\mathbb{Q}(\sqrt{d})$, $d = 2, 3, 5$. The function `qrso_search` searches for such a sequence for $d = 1 \pmod 8$ in a choosen range.

Examples

```
/* three exceptions */

? qrso_testfirstnonsplit(2)
% = [[-1, -1]~, [-1, 1]~, [0, -1]~, [0, 1]~, [1, -1]~, [1, 1]~, [2, -1]~, [2, 1]~]
? qrso_testfirstnonsplit(3)
% = [[-1, -1]~, [-1, 1]~, [2, -1]~, [2, 1]~]
? qrso_testfirstnonsplit(5)
% = [[-1, -1]~, [0, -1]~, [0, 1]~, [1, -1]~, [1, 1]~, [2, 1]~]
```

`int qrso_search(GEN first, GEN upto, int verbose, GEN *found)`

This function looks for a real quadratic number field $\mathbb{Q}(\sqrt{d})$, for d running from `first` to `upto` and $d = 1 \pmod 8$, such that there exist a sequence of length superior or equal to the value $m_d =$ `qfirstnonsplit(d)` with the first $m_d - 1$ terms contained in $\mathbb{Z}$.

It is known that such sequences do not exist for d large enough, so this function searches for a potential exception in range [`first`..`upto`]. The function returns 1 if some exception is found and set to `*found` the value of d, 0 otherwise. Setting `verbose` to 1 will print informations about the search on standart output.

The gp prototype is `qrso_search(first, upto, {verbose = 0}, {&found = NULL})`

Examples

```
? qrso_search(2, 10^6)
% = 0;  /* nothing in range [2..10^6]  */

% qrso_search(10^6, 10^7)
% = 0  /* nothing in range [10^6..10^7 ] */
```

`GEN qrso_bound( GEN A, GEN lambda, GEN M)`

Warning : this function is not smart and kind of impraticable. It is here for testing purpose only.

Return a bound $B > 0$ depending on $0 <$ `A` < 1 and $0 <$ `lambda` $< \frac{1}{4}$ such that for $d > B$ any basal simultaneous ordering in $\mathbb{Q}(\sqrt{d})$ of length $< d^\lambda$ is contained in $\mathbb{Z}$. The argument `M` is a 3×2 matrix of `t_INT`s giving values for internal parameters. The precise meaning is related to [AC11], and is as follow :

- each column of `M` is of the form `[first, delta]` where `first` is the initial value for a bruteforce loop while `delta` is the increment.
- the columns of `M` corrsepond to parameters for the computation of `B2`, `N0` and `B3` respectively, keeping with the notations in [AC11].

The function will print the values corresponding to `B2`, `N0` and `B3`.

Examples

```
? M = [[1,100]~,[1,1]~,[1,1]~];
? qrso_bound(0.68, 2/11, M)
B2 : 724401
N0 : 25
B3 : 34153
% = 724401
```

5 Miscellaneous

5.1 Useful functions

Few internal functions used by the library that can be useful.

`GEN vdiffprod(GEN nf, GEN v, GEN x)`

Return the product of differences of `x` with components of the vector `v`.

`GEN vdiffprod_i(GEN nf, GEN v, long i)`

Equivalent to `vdiffprod(nf, v[1..i-1], v[i])`.

`GEN vdiffs(GEN nf, GEN v)`

Return the vector `[vdiffprod_i(nf, v, i)]`, $2 \leq i \leq \#$`v`.

`GEN volume(GEN nf, GEN v)`

Return the volume of the vector `v`, i.e the product of all distinct pairs of elements of `v`. Volume is defined up to ± 1.

`GEN volume_i(GEN nf, GEN v, long i)`

Equivalent to `volume(nf, vec_shorten(S, i))`.

`GEN volume2(GEN nf, GEN v)`

This is just the square of `volume`, sometimes preferred to `volume`.

`GEN qfirstnonsplit(GEN nf)`

Returns the first prime number who does not split in the quadratic number field `nf`. The argument `nf` can also be a fundamental discriminant or a squarefree integer.

`GEN idealmaxlist(GEN nf, long n)`

Same as `ideallist` but for maximal ideals.

`GEN idealmaxprod(GEN nf, GEN p, long k)`

Return the product of all maximal ideals of norm equal to p^k.

`GEN idealmaxprodlist(GEN nf, long n)`

Return the vector v such that $v[i]$ equals the product of all maximal ideals of norm i, $1 \leq i \leq n$.

`GEN qfunorm(GEN nf)`

Return the norm of the fundamental unit of the quadratic number field `nf`. The argument `nf` can also be a fundamental discriminant or a squarefree integer.

5.2 gp interface

This section list in alphabetical order the prototypes of the functions installed by the gp script libfact.gp. A help section is accessible for each function inside gp by the usual help command.

```
allpord (nf, pr, S, SS, {&trunc=-1}, {&ex=NULL})

allsimulord (nf, S, SS, {trunc=-1}, {&ex=NULL})

idealmaxlist (nf, n)

idealmaxprod (nf, p, k)

idealmaxprodlist (nf, n)

isopord (nf, pr, S, h, {trunc=-1})

ispolyaupto (nf, n)

ispolyaupto_mod (nf, M, n)

ispolyaupto_rem (nf, r, n)

ispord (nf, pr, S, {trunc=-1}, {&i=NULL})

isrpord (nf, pr, S, r, {trunc=-1}, {&i=NULL})

issimulord (nf, S, {trunc=-1}, {&i=NULL})

isstrongpord (nf, pr ,S)

legf (q, n)

legf_vec (q, n)

nfX_divdiff (nf, pol, k, {&vars=NULL})

olegf (q, n, h)

olegf_vec (q, h, n)

opord (nf, pr, S, h, {first=1}, {trunc=-1}, {&ex=NULL}, {&inv=NULL})

opord_e (K, pr, S, h, {trunc=-1})

pord (nf, pr, S, {first=1}, {trunc=-1}, {&ex=NULL}, {&inv=NULL})

pord_e (nf, pr, S, {trunc=-1})

qallsimulord (nf, n)

qfact (nf, k)

qfactnorm (nf, k)

qfactnorm_vec (nf,n)

qfact_vec (nf,n)
```

```
qfirstnonsplit (nf)
qfunorm (nf)
qispolya (nf)
qrso_bound (A, lambda, d, delta)
qrso_lambda (A, lamba, {d = 7}, {delta = 1})
qrso_search (first, upto, {verbose = 0}, {&found = NULL})
qrso_testfirstnonsplit (d)
rlegf (q, n, r)
rlegf_vec (q, r, n)
rpord_e (nf, pr, S, r, {trunc=-1})
rpord (nf, pr, S, r, {trunc=-1}, {&ex=NULL}, {&inv=NULL})
simulord (nf, S, {trunc=-1}, {&ex=NULL})
sfact (nf, S, k)
sfactexp (nf, S, x)
sfactmod (nf, S, M, k)
sfactmodnorm (nf, S, M, k)
sfactmodnorm_vec (nf, S, M, {n=-1})
sfactmod_vec (nf, S, M, {n=-1})
sfactnorm (nf, S, k)
sfactnorm_vec (nf, S, {n=-1})
sfact_vec (nf, S, {n=-1})
sremfact (nf, S, r, k)
sremfactnorm (nf, S, r, k)
sremfactnorm_vec (nf, S, r, {n=-1})
sremfact_vec (nf, S, r, {n=-1})
strongpord (nf, pr, n)
vdiffprod (nf, v, x)
vdiffprod_i (nf, v, i)
vdiffs (nf, v)
volume (nf, v)
volume_i (nf, v, i)
```

```
volume2 (nf, v)

zkalmostsso (nf, n, {a0 = 0}, {ipr = []})

zkissimulord (nf, S)

zkfact (nf, k)

zkfactmod (nf, M, k)

zkfactmodnorm (nf, M, k)

zkfactmodnorm_vec (nf, M, n)

zkfactmodpol (nf, M, k, s, {cmode=1})

zkfactmodpol_vec (nf, M, n, s, {cmode=1})

zkfactmod_vec (nf, M, n)

zkfactnorm (nf, k)

zkfactnorm_vec (nf, n)

zkfactpol (nf, k, s, {cmode=1})

zkfactpol_vec (nf, n, s, {cmode=1})

zkfact_vec (nf, n)

zkmodregbasis (nf, M, n, s, {cmode=1})

zkopord (nf, pr, h, n)

zkpord(nf, pr, n)

zkregbasis (bnf, n, s, {cmode=1})

zkregbasis_fermat (bnf, n, s, {cmode=1})

zkremfact (nf, r, k)

zkremfactnorm (nf, r, k)

zkremfactnorm_vec (nf, r, n)

zkremfactpol (nf, r, k, s, {cmode=1})

zkremfactpol_vec (nf, r, n, s, {cmode=1})

zkremfact_vec (nf, r, n)

zkremregbasis (nf, r, n, s, {cmode=1})

zkrpord (nf, S, r, n)
```

5.3 Todo

- The library does not provide functions to compute with the special case of $\mathbb{Z}$. This is because Pari/GP is not needed for this, any good arithmetic library should do, and because such functions probably already exist. For completeness anyway, it would be nice to include them. See [Joh10].

- In [Ada18], the author introduced another type of ordering which is a natural candidate for new functions in the library.

- The functions `allpord` and `allsimulord` are a mess : they do not proceed the right way. They iterate over permutations with a rustic hashtable system to eliminate most of them. The right way would be to build the orderings using shuffling. This is a lot of (simple) combinatorial work, but a lot anyway. Also, it would be nice to provide functions to compute only the cardinality of sets returned by `allpord` and co. See ([Joh09]). Finally, the library lacks functions like `allrpord`, `allopord` and so on for the other types of ordering. More generally, the whole source file misc.c should be refreshed.

- The `rpord` family functions can be optimized. More precisely, it is not needed to iterate over all subsets of cardinality (n-r) to compute the required minimum, some subsets can easily be avoided.

References

[Ada18] David Adam. “Polynômes à valeurs entières sur des ensembles invariants par rotation”. In: *Journal of Number Theory* 186 (2018), pp. 417–438. ISSN: 0022-314X. DOI: https://doi.org/10.1016/j.jnt.2017.10.011. URL: http://www.sciencedirect.com/science/article/pii/S0022314X17304031.

[AC11] David Adam and Paul-Jean Cahen. “Newtonian and Schinzel quadratic fields”. In: *Journal of Pure and Applied Algebra* 215 (Aug. 2011), pp. 1902–1918. DOI: 10.1016/j.jpaa.2010.11.003.

[Bha97] Manjul Bhargava. “P-orderings and polynomial functions on arbitrary subsets of Dedekind rings.” In: *Journal für die reine und angewandte Mathematik* 490 (1997), pp. 101–128. URL: http://eudml.org/doc/153942.

[Bha00] Manjul Bhargava. “The Factorial Function and Generalizations”. In: *The American Mathematical Monthly* 107.9 (2000), pp. 783–799. ISSN: 00029890, 19300972. URL: http://www.jstor.org/stable/2695734.

[Bha09] Manjul Bhargava. “On P-orderings, rings of integer-valued polynomials, and ultrametric analysis”. English (US). In: *Journal of the American Mathematical Society* 22.4 (Oct. 2009), pp. 963–993. ISSN: 0894-0347. DOI: 10.1090/S0894-0347-09-00638-9.

[BCY09] Manjul Bhargava, Paul-Jean Cahen, and Julie Yeramian. “Finite generation properties for various rings of integer-valued polynomials”. In: *Journal of Algebra* 322.4 (2009), pp. 1129–1150. ISSN: 0021-8693. DOI: https://doi.org/10.1016/j.jalgebra.2009.04.017. URL: http://www.sciencedirect.com/science/article/pii/S0021869309002221.

[CCS97] P.J. Cahen, J.L. Chabert, and American Mathematical Society. “Integer-valued Polynomials”. In: American Mathematical Society Translations (1997). URL: https://books.google.fr/books?id=OdLxBwAAQBAJ.

[CC18] Paul-Jean Cahen and Jean-Luc Chabert. “Test sets for polynomials: n -universal subsets and Newton sequences”. In: *Journal of Algebra* 502 (Feb. 2018). DOI: 10.1016/j.jalgebra.2018.01.020.

[Joh09] Keith Johnson. “P-Orderings of Finite Subsets of Dedekind Domains”. In: *J. Algebraic Comb.* 30.2 (Sept. 2009), pp. 233–253. ISSN: 0925-9899. DOI: 10.1007/s10801-008-0157-9. URL: https://doi.org/10.1007/s10801-008-0157-9.

[Joh10] Keith Johnson. “Computing r-removed P-orderings and P-orderings of order h”. en. In: *Actes des rencontres du CIRM* 2.2 (2010), pp. 33–40. DOI: 10.5802/acirm.31. URL: acirm.centre-mersenne.org/item/ACIRM_2010__2_2_33_0/.

[Woo03] Melanie Wood. “P-orderings: A metric viewpoint and the non-existence of simultaneous orderings”. In: *Journal of Number Theory - J NUMBER THEOR* 99 (Mar. 2003), pp. 36–56. DOI: 10.1016/S0022-314X(02)00056-2.