Rigorous computation of class group and unit group

Koen de Boer[1] **Alice Pellet-Mary**[2] Benjamin Wesolowski[2]

[1] CWI and Leiden university [2] CNRS and Bordeaux university

Lfant seminar, Bordeaux

Main result

We describe an algorithm that computes the class group and unit group

- in any number field K (with discriminant Δ_K and degree n)
- runs in expected subexponential time $L_{\Delta_K}(1/2) + L_{n^n}(2/3)$
  (and polynomial in the residue ρ_K of the Dedekind zeta function at 1)
- is provably correct (assuming ERH)

History

Notation: $L_x(\alpha) = \exp\left(O(\log(x)^\alpha \cdot \log\log(x)^{1-\alpha})\right)$

	Number fields	Complexity	Non heuristic
[HM89]	quadratic imaginary	$L_{\Delta_K}(1/2)$	✓

(all algorithms assume ERH)

[HM89] Hafner, McCurley. A rigorous subexponential algorithm for computation of class groups. Journal of the American mathematical society.

History

Notation: $L_x(\alpha) = \exp\left(O(\log(x)^\alpha \cdot \log\log(x)^{1-\alpha})\right)$

	Number fields	Complexity	Non heuristic
[HM89]	quadratic imaginary	$L_{\Delta_K}(1/2)$	✓
[Buc88]	fixed degree n	$L_{\Delta_K}(1/2)$	✗

(all algorithms assume ERH)

[Buc88] Buchmann. A subexponential algorithm for the determination of class groups and regulators of algebraic number fields. Séminaire de théorie des nombres.

History

Notation: $L_x(\alpha) = \exp\left(O(\log(x)^\alpha \cdot \log\log(x)^{1-\alpha})\right)$

	Number fields	Complexity	Non heuristic
[HM89]	quadratic imaginary	$L_{\Delta_K}(1/2)$	✓
[Buc88]	fixed degree n	$L_{\Delta_K}(1/2)$	✗
[BF14]	arbitrary degree n	$L_{\Delta_K}(2/3)$	✗

(all algorithms assume ERH)

[BF14] Biasse, Fieker. Subexponential class group and unit group computation in large degree number fields. LMS Journal of Computation and Mathematics.

History

Notation: $L_x(\alpha) = \exp\left(O(\log(x)^\alpha \cdot \log\log(x)^{1-\alpha})\right)$

	Number fields	Complexity	Non heuristic
[HM89]	quadratic imaginary	$L_{\Delta_K}(1/2)$	✓
[Buc88]	fixed degree n	$L_{\Delta_K}(1/2)$	✗
[BF14]	arbitrary degree n	$L_{\Delta_K}(2/3)$	✗
[BF14,Gel17] [BEF+17]	specific defining polynomial	as small as $L_{\Delta_K}(1/3)$	✗

(all algorithms assume ERH)

[Gel17] Gélin. Class group computations in number fields and applications to cryptology. PhD thesis.

[BEF+17] Biasse, Espitau, Fouque, Gélin, Kirchner. Computing generator in cyclotomic integer rings. Eurocrypt.

History

Notation: $L_x(\alpha) = \exp\left(O(\log(x)^\alpha \cdot \log\log(x)^{1-\alpha})\right)$

	Number fields	Complexity	Non heuristic
[HM89]	quadratic imaginary	$L_{\Delta_K}(1/2)$	✓
[Buc88]	fixed degree n	$L_{\Delta_K}(1/2)$	✗
[BF14]	arbitrary degree n	$L_{\Delta_K}(2/3)$	✗
[BF14,Gel17] [BEF+17]	specific defining polynomial	as small as $L_{\Delta_K}(1/3)$	✗
[BS16]	arbitrary degree n	quantum poly	✓

(all algorithms assume ERH)

[BS16] Biasse, Song. A polynomial time quantum algorithm for computing class groups and solving the principal ideal problem in arbitrary degree number fields. SODA.

History

Notation: $L_x(\alpha) = \exp\left(O(\log(x)^\alpha \cdot \log\log(x)^{1-\alpha})\right)$

	Number fields	Complexity	Non heuristic
[HM89]	quadratic imaginary	$L_{\Delta_K}(1/2)$	✓
[Buc88]	fixed degree n	$L_{\Delta_K}(1/2)$	✗
[BF14]	arbitrary degree n	$L_{\Delta_K}(2/3)$	✗
[BF14,Gel17] [BEF+17]	specific defining polynomial	as small as $L_{\Delta_K}(1/3)$	✗
[BS16]	arbitrary degree n	quantum poly	✓
This work	arbitrary degree n	$\rho_K(L_{\Delta_K}(1/2) + L_{n^n}(2/3))$	✓

(all algorithms assume ERH)

[BS16] Biasse, Song. A polynomial time quantum algorithm for computing class groups and solving the principal ideal problem in arbitrary degree number fields. SODA.

S-units

Notations:

- $\mathcal{S}$ is a finite set of prime ideals of $\mathcal{O}_K$
- $\mathrm{Log} : K \to \mathbb{R}^n$ is the logarithmic embedding
  ($\mathrm{Log}(x) = (\log|\sigma_1(x)|, \cdots, \log|\sigma_n(x)|)$, with σ_i the complex embeddings of K)

S-units

Notations:

- $\mathcal{S}$ is a finite set of prime ideals of $\mathcal{O}_K$
- $\mathrm{Log} : K \to \mathbb{R}^n$ is the logarithmic embedding
  ($\mathrm{Log}(x) = (\log|\sigma_1(x)|, \cdots, \log|\sigma_n(x)|)$, with σ_i the complex embeddings of K)

Definition

The Log-$\mathcal{S}$-unit lattice is

$$\Lambda_{\mathcal{S}} := \left\{ \left(\mathrm{Log}(x), (-n_{\mathfrak{p}})_{\mathfrak{p}\in\mathcal{S}} \right) \,\middle|\, x\mathcal{O}_K = \prod_{\mathfrak{p}\in\mathcal{S}} \mathfrak{p}^{n_{\mathfrak{p}}} \right\} \subset \mathbb{R}^n \times \mathbb{Z}^{|\mathcal{S}|}$$

S-units

Notations:

- $\mathcal{S}$ is a finite set of prime ideals of $\mathcal{O}_K$
- $\mathrm{Log} : K \to \mathbb{R}^n$ is the logarithmic embedding
  ($\mathrm{Log}(x) = (\log|\sigma_1(x)|, \cdots, \log|\sigma_n(x)|)$, with σ_i the complex embeddings of K)

Definition

The Log-$\mathcal{S}$-unit lattice is

$$\Lambda_{\mathcal{S}} := \left\{ \left(\mathrm{Log}(x), (-n_{\mathfrak{p}})_{\mathfrak{p}\in\mathcal{S}}\right) \,\middle|\, x\mathcal{O}_K = \prod_{\mathfrak{p}\in\mathcal{S}} \mathfrak{p}^{n_{\mathfrak{p}}} \right\} \subset \mathbb{R}^n \times \mathbb{Z}^{|\mathcal{S}|}$$

$$\boxed{\text{Computing } \mathcal{S}\text{-units} = \text{computing a basis of } \Lambda_{\mathcal{S}}}$$

Theorem and applications

Theorem

Assuming ERH, there is a probabilistic algorithm which computes $\Lambda_{\mathcal{S}}$ in expected time polynomial in its input length, in ρ_K, in $L_{\Delta_K}(1/2)$ and in $L_{n^n}(2/3)$.

Reminder: ρ_K is the residue at 1 of ζ_K

Theorem and applications

Theorem

Assuming ERH, there is a probabilistic algorithm which computes $\Lambda_{\mathcal{S}}$ in expected time polynomial in its input length, in ρ_K, in $L_{\Delta_K}(1/2)$ and in $L_{n^n}(2/3)$.

Reminder: ρ_K is the residue at 1 of ζ_K

Applications: we can also compute

- unit group $(\mathcal{S} = \emptyset)$
- class-group ($\mathcal{S}$ generates Cl_K)
- generators of principal ideals
- class-group discrete logarithms

Outline of the talk

Computing a vector of $\Lambda_{\mathcal{S}}$

Definition: $\mathcal{S} = \{\text{prime } \mathfrak{p} \,|\, \mathcal{N}(\mathfrak{p}) \leq B\}$ (for some B to be determined)

Computing a vector of $\Lambda_{\mathcal{S}}$

Definition: $\mathcal{S} = \{\text{prime } \mathfrak{p} \,|\, \mathcal{N}(\mathfrak{p}) \le B\}$ (for some B to be determined)

Algorithm `SampleVector`

1: **repeat**
2: Sample random $x \in \mathcal{O}_K$
3: **until** $x\mathcal{O}_K = \prod_{\mathfrak{p}\in\mathcal{S}} \mathfrak{p}^{n_\mathfrak{p}}$ for some $n_\mathfrak{p} \in \mathbb{Z}$
4: **return** $\big(\mathrm{Log}(x), (-n_\mathfrak{p})_\mathfrak{p}\big)$

Computing a vector of $\Lambda_{\mathcal{S}}$

Definition: $\mathcal{S} = \{\text{prime } \mathfrak{p} \,|\, \mathcal{N}(\mathfrak{p}) \leq B\}$ (for some B to be determined)

Algorithm `SampleVector`

1: **repeat**
2: Sample random $x \in \mathcal{O}_K$
3: **until** $x\mathcal{O}_K = \prod_{\mathfrak{p}\in\mathcal{S}} \mathfrak{p}^{n_\mathfrak{p}}$ for some $n_\mathfrak{p} \in \mathbb{Z}$
4: **return** $\big(\mathrm{Log}(x), (-n_\mathfrak{p})_\mathfrak{p}\big)$

Correctness: ✓

Computing a vector of $\Lambda_{\mathcal{S}}$

Definition: $\mathcal{S} = \{\text{prime } \mathfrak{p} \,|\, \mathcal{N}(\mathfrak{p}) \leq B\}$ (for some B to be determined)

Algorithm `SampleVector`

1: **repeat**
2: Sample random $x \in \mathcal{O}_K$
3: **until** $x\mathcal{O}_K = \prod_{\mathfrak{p}\in\mathcal{S}} \mathfrak{p}^{n_\mathfrak{p}}$ for some $n_\mathfrak{p} \in \mathbb{Z}$
4: **return** $\big(\mathrm{Log}(x), (-n_\mathfrak{p})_\mathfrak{p}\big)$

Correctness: ✓

Complexity: $O\Big(T_{\text{sample}} \cdot p_{\text{smooth}}^{-1} \cdot |\mathcal{S}|\Big)$

- T_{sample}: time to sample x
- p_{smooth}: probability that $x\mathcal{O}_K$ is smooth
- $|\mathcal{S}| = O(B)$: time to test smoothness

Computing a vector of $\Lambda_{\mathcal{S}}$

Definition: $\mathcal{S} = \{\text{prime } \mathfrak{p} \,|\, \mathcal{N}(\mathfrak{p}) \leq B\}$ (for some B to be determined)

Algorithm `SampleVector`

1: **repeat**
2: Sample random $x \in \mathcal{O}_K$
3: **until** $x\mathcal{O}_K = \prod_{\mathfrak{p} \in \mathcal{S}} \mathfrak{p}^{n_\mathfrak{p}}$ for some $n_\mathfrak{p} \in \mathbb{Z}$
4: **return** $\big(\mathrm{Log}(x), (-n_\mathfrak{p})_\mathfrak{p}\big)$

Correctness: ✓

Complexity: $O\Big(T_{\text{sample}} \cdot p_{\text{smooth}}^{-1} \cdot |\mathcal{S}|\Big)$

- T_{sample}: time to sample x
- p_{smooth}: probability that $x\mathcal{O}_K$ is smooth
- $|\mathcal{S}| = O(B)$: time to test smoothness

How to sample x? – Buchmann

Buchmann:

- sample random ideal $I = \prod_{\mathfrak{p} \in \mathcal{S}} \mathfrak{p}^{m_\mathfrak{p}}$
- $x = \texttt{Shortest\_Vector}(I)$

How to sample x? – Buchmann

Buchmann:

- sample random ideal $I = \prod_{\mathfrak{p} \in \mathcal{S}} \mathfrak{p}^{m_\mathfrak{p}}$
- $x = \texttt{Shortest\_Vector}(I)$

$$x\mathcal{O}_K \text{ smooth} \Leftrightarrow xI^{-1} \text{ smooth}$$

($\Rightarrow$ the smaller $\mathcal{N}(xI^{-1})$ the better)

How to sample x? – Buchmann

Buchmann:

- sample random ideal $I = \prod_{\mathfrak{p} \in \mathcal{S}} \mathfrak{p}^{m_\mathfrak{p}}$
- $x = \texttt{Shortest\_Vector}(I)$

$$x\mathcal{O}_K \text{ smooth} \Leftrightarrow xI^{-1} \text{ smooth}$$
$$(\Rightarrow \text{the smaller } \mathcal{N}(xI^{-1}) \text{ the better})$$

Complexity: $2^{O(n)}$ (for $\texttt{Shortest\_Vector}(I)$)

How to sample x? – Buchmann

Buchmann:

- sample random ideal $I = \prod_{\mathfrak{p} \in \mathcal{S}} \mathfrak{p}^{m_\mathfrak{p}}$
- $x = \texttt{Shortest\_Vector}(I)$

$x\mathcal{O}_K$ smooth $\Leftrightarrow$ xI^{-1} smooth

($\Rightarrow$ the smaller $\mathcal{N}(xI^{-1})$ the better)

Complexity: $2^{O(n)}$ (for $\texttt{Shortest\_Vector}(I)$)

Subexponential only for fixed n

How to sample x? – Biasse-Fieker

Biasse-Fieker:

- sample random ideal $I = \prod_{\mathfrak{p} \in \mathcal{S}} \mathfrak{p}^{m_{\mathfrak{p}}}$
- $x = \mathrm{BKZ}_\beta(I)$ (blocksize β)

How to sample x? – Biasse-Fieker

Biasse-Fieker:

- sample random ideal $I = \prod_{\mathfrak{p} \in \mathcal{S}} \mathfrak{p}^{m_{\mathfrak{p}}}$
- $x = \mathrm{BKZ}_\beta(I)$ (blocksize β)

Complexity: $2^{O(\beta)}$ (can be subexponential)

How to sample x? – Biasse-Fieker

Biasse-Fieker:

- sample random ideal $I = \prod_{\mathfrak{p} \in \mathcal{S}} \mathfrak{p}^{m_\mathfrak{p}}$
- $x = \mathrm{BKZ}_\beta(I)$ (blocksize β)

Complexity: $2^{O(\beta)}$ (can be subexponential)

Shortness: $\|x\|_\infty \leq n^{n/\beta} \cdot \Delta_K^{1/(2n)} \cdot \mathcal{N}(I)^{1/n}$

How to sample x? – Biasse-Fieker

Biasse-Fieker:

- sample random ideal $I = \prod_{\mathfrak{p} \in \mathcal{S}} \mathfrak{p}^{m_\mathfrak{p}}$
- $x = \mathrm{BKZ}_\beta(I)$ (blocksize β)

Complexity: $2^{O(\beta)}$ (can be subexponential)

Shortness: $\|x\|_\infty \le n^{n/\beta} \cdot \Delta_K^{1/(2n)} \cdot \mathcal{N}(I)^{1/n}$

$$\boxed{\Rightarrow \quad \mathcal{N}(xI^{-1}) \le n^{n^2/\beta} \cdot \sqrt{\Delta_K}}$$

Intermediate summary

$\mathcal{S} := \{\text{prime } \mathfrak{p} \,|\, \mathcal{N}(\mathfrak{p}) \leq B\}$

Algorithm `SampleVector`

1: **repeat**
2: Sample random $I = \prod_{\mathfrak{p} \in \mathcal{S}} \mathfrak{p}^{m_\mathfrak{p}}$
3: $x \leftarrow \mathrm{BKZ}_\beta(I)$
4: **until** $xI^{-1} = \prod_{\mathfrak{p} \in \mathcal{S}} \mathfrak{p}^{n_\mathfrak{p}}$ for some $n_\mathfrak{p} \in \mathbb{Z}$
5: **return** $\big(\mathrm{Log}(x), (-n_\mathfrak{p} - m_\mathfrak{p})_\mathfrak{p}\big)$

Intermediate summary

$\mathcal{S} := \{\text{prime } \mathfrak{p} \,|\, \mathcal{N}(\mathfrak{p}) \leq B\}$

Algorithm `SampleVector`

1: **repeat**
2: Sample random $I = \prod_{\mathfrak{p}\in\mathcal{S}} \mathfrak{p}^{m_\mathfrak{p}}$
3: $x \leftarrow \mathrm{BKZ}_\beta(I)$
4: **until** $xI^{-1} = \prod_{\mathfrak{p}\in\mathcal{S}} \mathfrak{p}^{n_\mathfrak{p}}$ for some $n_\mathfrak{p} \in \mathbb{Z}$
5: **return** $\left(\mathrm{Log}(x), (-n_\mathfrak{p} - m_\mathfrak{p})_\mathfrak{p}\right)$

Complexity:

$$O\Big(T_{\text{sample}} \cdot p_{\text{smooth}}^{-1} \cdot |\mathcal{S}|\Big) \;=\; 2^{O(\beta)} \cdot B \cdot p_{\text{smooth}}^{-1}$$

Intermediate summary

$\mathcal{S} := \{\text{prime } \mathfrak{p} \mid \mathcal{N}(\mathfrak{p}) \leq B\}$

Algorithm `SampleVector`

1: **repeat**
2: Sample random $I = \prod_{\mathfrak{p}\in\mathcal{S}} \mathfrak{p}^{m_\mathfrak{p}}$
3: $x \leftarrow \mathrm{BKZ}_\beta(I)$
4: **until** $xI^{-1} = \prod_{\mathfrak{p}\in\mathcal{S}} \mathfrak{p}^{n_\mathfrak{p}}$ for some $n_\mathfrak{p} \in \mathbb{Z}$
5: **return** $\big(\mathrm{Log}(x), (-n_\mathfrak{p} - m_\mathfrak{p})_\mathfrak{p}\big)$

Complexity:

$$O\Big(T_{\text{sample}} \cdot p_{\text{smooth}}^{-1} \cdot |\mathcal{S}|\Big) \;=\; 2^{O(\beta)} \cdot B \cdot p_{\text{smooth}}^{-1}$$

Smoothness probability

Notation: $p_{y,B}$ = proba that a uniformly random ideal of norm $\leq y$ is B-smooth

Smoothness probability

Notation: $p_{y,B}$ = proba that a uniformly random ideal of norm $\leq y$ is B-smooth

Asymptotically: $p_{y,B} \sim \rho(u) \approx u^{-u}$ (ρ Dickman function, $u = \log y / \log B$)

Smoothness probability

Notation: $p_{y,B}$ = proba that a uniformly random ideal of norm $\leq y$ is B-smooth

Asymptotically: $p_{y,B} \sim \rho(u) \approx u^{-u}$ (ρ Dickman function, $u = \log y / \log B$)

⚠ K fixed and B, u tending to infinity

Smoothness probability

Notation: $p_{y,B}$ = proba that a uniformly random ideal of norm $\leq y$ is B-smooth

Asymptotically: $p_{y,B} \sim \rho(u) \approx u^{-u}$ (ρ Dickman function, $u = \log y / \log B$)

⚠ K fixed and B, u tending to infinity

Heuristic: $p_{\text{smooth}} \approx u^{-u}$ where $u = \frac{\log \mathcal{N}(xI^{-1})}{\log B}$

Smoothness probability

Notation: $p_{y,B}$ = proba that a uniformly random ideal of norm $\leq y$ is B-smooth

Asymptotically: $p_{y,B} \sim \rho(u) \approx u^{-u}$ (ρ Dickman function, $u = \log y / \log B$)

⚠ K fixed and B, u tending to infinity

Heuristic: $p_{\text{smooth}} \approx u^{-u}$ where $u = \frac{\log \mathcal{N}(xI^{-1})}{\log B}$

2 assumptions hidden:

- the provable asymptotic bounds require huge B to be effective (roughly $B \gtrsim 2^{2^n}$)
- xI^{-1} is not a random ideal of bounded norm

Sampling one vector – summary

Algorithm `SampleVector`

1: **repeat**
2: Sample random $I = \prod_{\mathfrak{p}\in\mathcal{S}} \mathfrak{p}^{m_\mathfrak{p}}$
3: $x \leftarrow \mathrm{BKZ}_\beta(I)$
4: **until** $xI^{-1} = \prod_{\mathfrak{p}\in\mathcal{S}} \mathfrak{p}^{n_\mathfrak{p}}$ for some $n_\mathfrak{p} \in \mathbb{Z}$
5: **return** $\big(\mathrm{Log}(x), (-n_\mathfrak{p} - m_\mathfrak{p})_\mathfrak{p}\big)$

Complexity (heuristic):

$$2^{O(\beta)} \cdot B \cdot u^u \qquad \text{with } u = \frac{\log \mathcal{N}(xI^{-1})}{\log B} \leq \frac{n^2 \log n/\beta + \log|\Delta_K|/2}{\log B}$$

Sampling one vector – summary

Algorithm `SampleVector`

1: **repeat**
2: Sample random $I = \prod_{\mathfrak{p} \in \mathcal{S}} \mathfrak{p}^{m_\mathfrak{p}}$
3: $x \leftarrow \mathrm{BKZ}_\beta(I)$
4: **until** $xI^{-1} = \prod_{\mathfrak{p} \in \mathcal{S}} \mathfrak{p}^{n_\mathfrak{p}}$ for some $n_\mathfrak{p} \in \mathbb{Z}$
5: **return** $\big(\mathrm{Log}(x), (-n_\mathfrak{p} - m_\mathfrak{p})_\mathfrak{p}\big)$

Complexity (heuristic):

$$2^{O(\beta)} \cdot B \cdot u^u \qquad \text{with } u = \frac{\log \mathcal{N}(xI^{-1})}{\log B} \leq \frac{n^2 \log n/\beta + \log|\Delta_K|/2}{\log B}$$

Instantiating:

- $\beta = n^{2/3}$
- $B = L_{\Delta_K}(1/2) + L_{n^n}(2/3)$

Total complexity: $L_{\Delta_K}(1/2) + L_{n^n}(2/3)$

Computing the full lattice $\Lambda_{\mathcal{S}}$

Remark: One can efficiently approximate $\det(\Lambda_{\mathcal{S}}) = \mathrm{Reg}_K \cdot h_K$
($\mathcal{S}$ generates the class-group)

Computing the full lattice $\Lambda_{\mathcal{S}}$

Remark: One can efficiently approximate $\det(\Lambda_{\mathcal{S}}) = \mathrm{Reg}_K \cdot h_K$
($\mathcal{S}$ generates the class-group)

Algorithm `ComputeSUnits`

1: **repeat**
2: $\quad \vec{z}_i \leftarrow$ `SampleVector()`
3: **until** $\mathcal{L}((\vec{z}_i)_i)$ is a lattice with the desired rank and det
4: Compute a basis B of $\mathcal{L}((\vec{z}_i)_i)$ (linear algebra)
5: **return** B

Computing the full lattice $\Lambda_{\mathcal{S}}$

Remark: One can efficiently approximate $\det(\Lambda_{\mathcal{S}}) = \mathrm{Reg}_K \cdot h_K$
($\mathcal{S}$ generates the class-group)

Algorithm `ComputeSUnits`

1: **repeat**
2: $\quad \vec{z}_i \leftarrow$ `SampleVector()`
3: **until** $\mathcal{L}((\vec{z}_i)_i)$ is a lattice with the desired rank and det
4: Compute a basis B of $\mathcal{L}((\vec{z}_i)_i)$ (linear algebra)
5: **return** B

Correctness: ✓

Computing the full lattice $\Lambda_{\mathcal{S}}$

Remark: One can efficiently approximate $\det(\Lambda_{\mathcal{S}}) = \mathrm{Reg}_K \cdot h_K$

($\mathcal{S}$ generates the class-group)

Algorithm `ComputeSUnits`

```
1: repeat
2:     z⃗_i ← SampleVector()
3: until L((z⃗_i)_i) is a lattice with the desired rank and det
4: Compute a basis B of L((z⃗_i)_i)     (linear algebra)
5: return B
```

Correctness: ✓

Heuristic: $O(B)$ vectors from `SampleVector()` generate $\Lambda_{\mathcal{S}}$ with good probability $\quad (\mathrm{rk}(\Lambda_{\mathcal{S}}) = O(B))$

Computing the full lattice $\Lambda_{\mathcal{S}}$

Remark: One can efficiently approximate $\det(\Lambda_{\mathcal{S}}) = \mathrm{Reg}_K \cdot h_K$
($\mathcal{S}$ generates the class-group)

Algorithm `ComputeSUnits`

1: **repeat**
2: $\quad \vec{z}_i \leftarrow$ `SampleVector()`
3: **until** $\mathcal{L}((\vec{z}_i)_i)$ is a lattice with the desired rank and det
4: Compute a basis B of $\mathcal{L}((\vec{z}_i)_i)$ (linear algebra)
5: **return** B

Correctness: ✓

Heuristic: $O(B)$ vectors from `SampleVector()` generate $\Lambda_{\mathcal{S}}$ with good probability $(\mathrm{rk}(\Lambda_{\mathcal{S}}) = O(B))$

Complexity: $\mathrm{poly}(B) = L_{\Delta_K}(1/2) + L_{n^n}(2/3)$

Heuristics – summary

Heuristic 1: $p_{\text{smooth}} \approx u^{-u}$ where $u = \frac{\log \mathcal{N}(xI^{-1})}{\log B}$

1.1. the asymptotic bounds hold even for smallish B's

1.2. xI^{-1} behaves like a uniform ideal of bounded norm

Heuristics – summary

Heuristic 1: $p_{\text{smooth}} \approx u^{-u}$ where $u = \frac{\log \mathcal{N}(xI^{-1})}{\log B}$

1.1. the asymptotic bounds hold even for smallish B's

1.2. xI^{-1} behaves like a uniform ideal of bounded norm

Heuristic 2: $O(B)$ vectors from `SampleVector()` generate $\Lambda_{\mathcal{S}}$ with good probability

Outline of the talk

1. Heuristic algorithms
2. Removing the first heuristic
3. Removing the second heuristic

Provable sampling in ideals [BDPW22]

Algorithm `SampleInIdeal`

1: sample random $I = \prod_{\mathfrak{p} \in \mathcal{S}} \mathfrak{p}^{m_\mathfrak{p}}$
2: $B \leftarrow BKZ_\beta(I)$ (B reduced basis of I)
3: sample random $x \in I$ (using small basis B)
4: **return** (x, I)

[BDPW22] de Boer, Ducas, Pellet-Mary, Wesolowski. Sampling ideals in a class: smooth, near-prime or otherwise.

Provable sampling in ideals [BDPW22]

Algorithm `SampleInIdeal`

1: sample random $I = \prod_{\mathfrak{p} \in \mathcal{S}} \mathfrak{p}^{m_{\mathfrak{p}}}$
2: $B \leftarrow BKZ_{\beta}(I)$ (B reduced basis of I)
3: sample random $x \in I$ (using small basis B)
4: **return** (x, I)

Theorem (ERH) [BDPW22]

For any infinite set $\mathcal{T}$ of ideals, it holds that

$$\Pr_{(x,I) \leftarrow \texttt{SampleInIdeal}} \left(xI^{-1} \in \mathcal{T} \right) \geq \frac{\delta_{\mathcal{T}}[n^{n^2/\beta} \cdot \sqrt{\Delta_K}]}{3}.$$

Definition: $\delta_{\mathcal{T}}[y] \approx \frac{|\{\mathfrak{a} \in \mathcal{T} \,|\, \mathcal{N}(\mathfrak{a}) \leq y\}|}{|\{\mathfrak{a} \text{ ideal} \,|\, \mathcal{N}(\mathfrak{a}) \leq y\}|}$ (density of $\mathcal{T}$ at y)

[BDPW22] de Boer, Ducas, Pellet-Mary, Wesolowski. Sampling ideals in a class: smooth, near-prime or otherwise.

Heuristic 1.2

Heuristic 1.2: xI^{-1} behaves like a uniform ideal of bounded norm

Can be proven using previous slide up to

- changing slightly the sampling procedure
- decreasing by 3 the success probability

Heuristic 1.1

Heuristic 1.1: $\delta_{\mathcal{S}}[y] \approx u^{-u}$ even for small B's $\quad (u = \log y / \log B)$

Heuristic 1.1

Heuristic 1.1: $\delta_{\mathcal{S}}[y] \approx u^{-u}$ even for small B's $\quad (u = \log y / \log B)$

- We could not prove it, but we proved

Lemma

For any $B \geq \Omega((n + \log \Delta_K)^3)$,

$$\delta_{\mathcal{S}}[y] \gtrsim u^{-u} \cdot \rho_K^{-1}.$$

Heuristic 1.1

Heuristic 1.1: $\delta_{\mathcal{S}}[y] \approx u^{-u}$ even for small B's $\quad (u = \log y / \log B)$

- We could not prove it, but we proved

Lemma

For any $B \geq \Omega((n + \log \Delta_K)^3)$,

$$\delta_{\mathcal{S}}[y] \gtrsim u^{-u} \cdot \rho_K^{-1}.$$

Discussion:

- for cyclotomic fields, $\rho_K = \mathrm{poly}(n)$

Heuristic 1.1

Heuristic 1.1: $\delta_{\mathcal{S}}[y] \approx u^{-u}$ even for small B's $\quad (u = \log y / \log B)$

- We could not prove it, but we proved

Lemma

For any $B \geq \Omega((n + \log \Delta_K)^3)$,

$$\delta_{\mathcal{S}}[y] \gtrsim u^{-u} \cdot \rho_K^{-1}.$$

Discussion:

- for cyclotomic fields, $\rho_K = \mathrm{poly}(n)$
- what about other fields?

Heuristic 1.1

Heuristic 1.1: $\delta_{\mathcal{S}}[y] \approx u^{-u}$ even for small B's $\quad (u = \log y / \log B)$

- We could not prove it, but we proved

Lemma

For any $B \geq \Omega((n + \log \Delta_K)^3)$,

$$\delta_{\mathcal{S}}[y] \gtrsim u^{-u} \cdot \rho_K^{-1}.$$

Discussion:

- for cyclotomic fields, $\rho_K = \mathrm{poly}(n)$
- what about other fields?
- if the bound is tight, this impacts also the heuristic algorithm

Heuristic 1 – summary

One can prove heuristic 1 up to

- changing slightly the sampling procedure (same asymptotic complexity)
- dividing p_{smooth} by ρ_K

Heuristic 1 – summary

One can prove heuristic 1 up to

- changing slightly the sampling procedure (same asymptotic complexity)
- dividing p_{smooth} by ρ_K

Theorem (ERH)

There is an algorithm `SampleVector` that computes $\vec{v} \in \Lambda_{\mathcal{S}}$ in time

$$\rho_K \cdot \Big(L_{\Delta_K}(1/2) + L_{n^n}(2/3) \Big).$$

Outline of the talk

1. Heuristic algorithms
2. Removing the first heuristic
3. Removing the second heuristic

Variation on `SampleVector`

Reminder

There is an algorithm `SampleVector` that

- computes $\vec{v} \in \Lambda_{\mathcal{S}}$
- in time

$$\rho_K \cdot \Big(L_{\Delta_K}(1/2) + L_{n^n}(2/3) \Big).$$

Variation on SampleVector

Reminder

There is an algorithm SampleVector that

- computes $x \in K$ and $(n_{\mathfrak{p}})_{\mathfrak{p}} \in \mathbb{Z}^{|\mathcal{S}|}$ such that $x\mathcal{O}_K = \prod_{\mathfrak{p} \in \mathcal{S}} \mathfrak{p}^{n_{\mathfrak{p}}}$
- in time

$$\rho_K \cdot \Big(L_{\Delta_K}(1/2) + L_{n^n}(2/3) \Big).$$

Variation on `SampleVector`

Reminder

There is an algorithm `SampleVector` that

- takes as input an ideal I
- computes $x \in K$ and $(n_{\mathfrak{p}})_{\mathfrak{p}} \in \mathbb{Z}^{|\mathcal{S}|}$ such that $x\mathcal{O}_K = I \cdot \prod_{\mathfrak{p} \in \mathcal{S}} \mathfrak{p}^{n_{\mathfrak{p}}}$
- in time

$$\rho_K \cdot \Big(L_{\Delta_K}(1/2) + L_{n^n}(2/3) \Big).$$

Variation on SampleVector

Reminder

There is an algorithm SampleVector that

- takes as input an ideal I
- computes $x \in K$ and $(n_\mathfrak{p})_\mathfrak{p} \in \mathbb{Z}^{|\mathcal{S}|}$ such that $x\mathcal{O}_K = I \cdot \prod_{\mathfrak{p} \in \mathcal{S}} \mathfrak{p}^{n_\mathfrak{p}}$
- in time
$$\rho_K \cdot \Big(L_{\Delta_K}(1/2) + L_{n^n}(2/3) \Big).$$

From now on, we use SampleVector in a black-box way

Heuristic 2 – main idea

Algorithm `RandomVector`

1: sample random $\vec{v} := \big(\operatorname{Log} x, (-n_{\mathfrak{p}})\big)$
2: define $I := x \cdot \prod_{\mathfrak{p} \in \mathcal{S}} \mathfrak{p}^{-n_{\mathfrak{p}}}$
3: $\vec{w} := \big(\operatorname{Log} y, (-m_{\mathfrak{p}})_{\mathfrak{p}}\big) \leftarrow$ `SampleVector(I)` $\quad (y\mathcal{O}_K = I \cdot \prod \mathfrak{p}^{m_{\mathfrak{p}}})$
4: **return** $\vec{w} - \vec{v}$

Heuristic 2 – main idea

Algorithm `RandomVector`

1: sample random $\vec{v} := \big(\operatorname{Log} x, (-n_{\mathfrak{p}})\big)$
2: define $I := x \cdot \prod_{\mathfrak{p} \in \mathcal{S}} \mathfrak{p}^{-n_{\mathfrak{p}}}$
3: $\vec{w} := \big(\operatorname{Log} y, (-m_{\mathfrak{p}})_{\mathfrak{p}}\big) \leftarrow$ `SampleVector(I)` $\quad (y\mathcal{O}_K = I \cdot \prod \mathfrak{p}^{m_{\mathfrak{p}}})$
4: **return** $\vec{w} - \vec{v}$

Correctness: $yx^{-1} \cdot \mathcal{O}_K = \prod_{\mathfrak{p}} \mathfrak{p}^{m_{\mathfrak{p}} - n_{\mathfrak{p}}} \quad \Rightarrow \quad \vec{w} - \vec{v} \in \Lambda_{\mathcal{S}}$

Heuristic 2 – main idea

Algorithm `RandomVector`

1: sample random $\vec{v} := \big(\operatorname{Log} x, (-n_{\mathfrak{p}})\big)$
2: define $I := x \cdot \prod_{\mathfrak{p} \in \mathcal{S}} \mathfrak{p}^{-n_{\mathfrak{p}}}$
3: $\vec{w} := \big(\operatorname{Log} y, (-m_{\mathfrak{p}})_{\mathfrak{p}}\big) \leftarrow$ `SampleVector(I)` $\quad (y\mathcal{O}_K = I \cdot \prod \mathfrak{p}^{m_{\mathfrak{p}}})$
4: **return** $\vec{w} - \vec{v}$

Correctness: $yx^{-1} \cdot \mathcal{O}_K = \prod_{\mathfrak{p}} \mathfrak{p}^{m_{\mathfrak{p}} - n_{\mathfrak{p}}} \quad \Rightarrow \quad \vec{w} - \vec{v} \in \Lambda_{\mathcal{S}}$

Intuition: $\vec{w} - \vec{v}$ is random in $\Lambda_{\mathcal{S}}$ (and independent from other vectors obtained so far) because `SampleVector` cannot guess $\vec{v}$ from I.

More details

Reminder:

- $\vec{v} = \big(\operatorname{Log} x, (-n_\mathfrak{p})_\mathfrak{p}\big)$
- $I = x \cdot \prod_\mathfrak{p} \mathfrak{p}^{-n_\mathfrak{p}}$
- $\vec{w} = \texttt{SampleVector(I)}$

More details

Reminder:

- $\vec{v} = \big(\operatorname{Log} x, (-n_{\mathfrak{p}})_{\mathfrak{p}}\big)$
- $I = x \cdot \prod_{\mathfrak{p}} \mathfrak{p}^{-n_{\mathfrak{p}}}$
- $\vec{w} = \texttt{SampleVector(I)}$

Observation: $x \prod_{\mathfrak{p}} \mathfrak{p}^{-n_{\mathfrak{p}}} = x' \prod_{\mathfrak{p}} \mathfrak{p}^{-n'_{\mathfrak{p}}} \Leftrightarrow \vec{v} = \vec{v}' \bmod \Lambda_{\mathcal{S}}$

More details

Reminder:

- $\vec{v} = \big(\operatorname{Log} x, (-n_{\mathfrak{p}})_{\mathfrak{p}}\big)$
- $I = x \cdot \prod_{\mathfrak{p}} \mathfrak{p}^{-n_{\mathfrak{p}}}$
- $\vec{w} = \texttt{SampleVector(I)}$

Observation: $x \prod_{\mathfrak{p}} \mathfrak{p}^{-n_{\mathfrak{p}}} = x' \prod_{\mathfrak{p}} \mathfrak{p}^{-n'_{\mathfrak{p}}} \Leftrightarrow \vec{v} = \vec{v}' \bmod \Lambda_{\mathcal{S}}$

$\Rightarrow$ I only depends on $\vec{v} + \Lambda_{\mathcal{S}}$
$\Rightarrow$ $\vec{w}$ only depends on $\vec{v} + \Lambda_{\mathcal{S}}$
(provided we have a canonical representation for I)

Distribution of $\vec{v}$

$$\vec{w} = f(\vec{v} + \Lambda_{\mathcal{S}}) \qquad \text{(f might be randomized)}$$

Distribution of $\vec{v}$

$$\boxed{\vec{w} = f(\vec{v} + \Lambda_{\mathcal{S}}) \qquad \text{(f might be randomized)}}$$

Consequence: ($D_{\Lambda,\sigma,\vec{c}}$ discrete gaussian distribution over Λ, center $\vec{c}$, deviation σ)

- $\vec{v} \leftarrow D_\sigma$
- $\vec{w} \leftarrow f(\vec{v} + \Lambda_{\mathcal{S}})$
- return $\vec{z} := \vec{v} - \vec{w}$

$\Longleftrightarrow$

(if σ large enough)

- $\vec{v}' + \Lambda_{\mathcal{S}} \leftarrow D_\sigma \bmod \Lambda_{\mathcal{S}}$
- $\vec{w} \leftarrow f(\vec{v}' + \Lambda_{\mathcal{S}})$
- $\vec{v} \leftarrow D_{\vec{v}' + \Lambda_{\mathcal{S}}, \sigma}$
- return $\vec{z} := \vec{v} - \vec{w}$

Distribution of $\vec{v}$

$$\vec{w} = f(\vec{v} + \Lambda_{\mathcal{S}}) \qquad (f \text{ might be randomized})$$

Consequence: ($D_{\Lambda,\sigma,\vec{c}}$ discrete gaussian distribution over Λ, center $\vec{c}$, deviation σ)

- $\vec{v} \leftarrow D_\sigma$
- $\vec{w} \leftarrow f(\vec{v} + \Lambda_{\mathcal{S}})$
- return $\vec{z} := \vec{v} - \vec{w}$

$\Longleftrightarrow$ (if σ large enough)

- $\vec{v}' + \Lambda_{\mathcal{S}} \leftarrow D_\sigma \bmod \Lambda_{\mathcal{S}}$
- $\vec{w} \leftarrow f(\vec{v}' + \Lambda_{\mathcal{S}})$
- $\vec{v} \leftarrow D_{\vec{v}'+\Lambda_{\mathcal{S}},\sigma}$
- return $\vec{z} := \vec{v} - \vec{w}$

$$\vec{v} - \vec{w} \sim D_{\Lambda_{\mathcal{S}},\sigma,\vec{c}} \quad (\text{for some random center } \vec{c})$$

Distribution of $\vec{v}$

$$\vec{w} = f(\vec{v} + \Lambda_{\mathcal{S}}) \qquad (f \text{ might be randomized})$$

Consequence: ($D_{\Lambda,\sigma,\vec{c}}$ discrete gaussian distribution over Λ, center $\vec{c}$, deviation σ)

- $\vec{v} \leftarrow D_\sigma$
- $\vec{w} \leftarrow f(\vec{v} + \Lambda_{\mathcal{S}})$
- return $\vec{z} := \vec{v} - \vec{w}$

$\iff$ (if σ large enough)

- $\vec{v}' + \Lambda_{\mathcal{S}} \leftarrow D_\sigma \bmod \Lambda_{\mathcal{S}}$
- $\vec{w} \leftarrow f(\vec{v}' + \Lambda_{\mathcal{S}})$
- $\vec{v} \leftarrow D_{\vec{v}'+\Lambda_{\mathcal{S}},\sigma}$
- return $\vec{z} := \vec{v} - \vec{w}$

$$\vec{v} - \vec{w} \sim D_{\Lambda_{\mathcal{S}},\sigma,\vec{c}} \qquad (\text{for some random center } \vec{c})$$

Lemma: $O(B)$ samples from $D_{\Lambda_{\mathcal{S}},\sigma,\vec{c}}$ generate $\Lambda_{\mathcal{S}}$ with high probability

Heuristic 2 – summary

Algorithm `ComputeSUnits`

1: **repeat**
2: sample $\vec{v} := \big(\operatorname{Log}(x), (-n_{\mathfrak{p}})\big) \leftarrow D_\sigma$
3: $\vec{w} \leftarrow \texttt{SampleVector}(x \cdot \prod_{\mathfrak{p}\in\mathcal{S}} \mathfrak{p}^{-n_{\mathfrak{p}}})$
4: $\vec{z}_i := \vec{v} - \vec{w}$
5: **until** $\mathcal{L}\big((\vec{z}_i)_i\big)$ has good rank and volume

Heuristic 2 – summary

Algorithm `ComputeSUnits`

1: **repeat**
2: sample $\vec{v} := \big(\mathrm{Log}(x), (-n_{\mathfrak{p}})\big) \leftarrow D_\sigma$
3: $\vec{w} \leftarrow \texttt{SampleVector}(x \cdot \prod_{\mathfrak{p}\in\mathcal{S}} \mathfrak{p}^{-n_{\mathfrak{p}}})$
4: $\vec{z}_i := \vec{v} - \vec{w}$
5: **until** $\mathcal{L}\big((\vec{z}_i)_i\big)$ has good rank and volume

Theorem (ERH)

If `SampleVector` is correct, then `ComputeSUnits` computes a basis of $\Lambda_{\mathcal{S}}$ in time $T(\texttt{SampleVector}) \cdot \mathrm{poly}(B)$.

Conclusion

Summary:

- remove both heuristics of Biasse-Fieker algorithm (under ERH)
- algorithm is slightly modified
- the run time is increased by a factor ρ_K

Conclusion

Summary:

- remove both heuristics of Biasse-Fieker algorithm (under ERH)
- algorithm is slightly modified
- the run time is increased by a factor ρ_K

Open question: is $\delta_{\mathcal{S}}[y] \approx u^{-u}$ or $\delta_{\mathcal{S}}[y] \approx \rho_K^{-1} \cdot u^{-u}$?

(**Reminder:** $\delta_{\mathcal{S}}[y]$ = density of B-smooth ideals of norm $\leq y$)

- can we improve our runtime?
- or are the runtime of the heuristic algorithms too optimistic?

Conclusion

Summary:

- remove both heuristics of Biasse-Fieker algorithm (under ERH)
- algorithm is slightly modified
- the run time is increased by a factor ρ_K

Open question: is $\delta_{\mathcal{S}}[y] \approx u^{-u}$ or $\delta_{\mathcal{S}}[y] \approx \rho_K^{-1} \cdot u^{-u}$?

(**Reminder:** $\delta_{\mathcal{S}}[y]$ = density of B-smooth ideals of norm $\leq y$)

- can we improve our runtime?
- or are the runtime of the heuristic algorithms too optimistic?

Questions?