Formal Verification of Bounds for the LLL Basis Reduction Algorithm

Maximilian Haslbeck and René Thiemann

University of Innsbruck, Austria

Abstract. The LLL basis reduction algorithm was the first polynomial-time algorithm to compute a reduced basis of a given lattice, and hence also a short vector in the lattice. It thereby approximates an NP-hard problem where the approximation quality solely depends on the dimension of the lattice, but not the lattice itself. The algorithm has several applications in number theory, computer algebra and cryptography.
In a recent paper, we presented the first formal soundness proof of the LLL algorithm. However, this proof did not include a formal statement of its complexity. Therefore, in this paper we provide two formal statements on the polynomial runtime. First, we formally prove a polynomial bound on the number of arithmetic operations. And second, we show that the numbers during the execution stay polynomial in size, so that each arithmetic operation can be performed in polynomial time.

1 Introduction

The LLL basis reduction algorithm by Lenstra, Lenstra and Lovász [5] is a remarkable algorithm with numerous applications. There even exists a 500-page book solely about the LLL algorithm [7]. It lists applications in number theory and cryptology, and also contains the best known polynomial factorization algorithm that is used in today's computer algebra systems.

The LLL algorithm plays an important role in finding short vectors in lattices: Given some list of linearly independent integer vectors $f_0, \ldots, f_{m-1} \in \mathbb{Z}^n$, the corresponding lattice L is the set of integer linear combinations of the f_i; and the shortest vector problem is to find some non-zero element in L which has the minimum norm.

Example 1. Consider $f_1 = (1, 1\,894\,885\,908, 0)$, $f_2 = (0, 1, 1\,894\,885\,908)$, and $f_3 = (0, 0, 2\,147\,483\,648)$. The lattice of f_1, f_2, f_3 has a shortest vector $(-3, 17, 4)$. It is the linear combination $(-3, 17, 4) = -3f_1 + 5\,684\,657\,741 f_2 + 5\,015\,999\,938 f_3$.

Whereas finding a shortest vector is NP-hard [6], the LLL algorithm is a polynomial time algorithm for approximating a shortest vector: The algorithm is parametric by some $\alpha > \frac{4}{3}$ and computes a *short vector*, i.e., a vector whose norm is at most $\alpha^{\frac{m-1}{2}}$ times as large than the norm of any shortest vector.

In recent work we developed the first mechanized soundness proof of the LLL algorithm [3]. It is proven in Isabelle/HOL [8] and states the functional correctness. To be more precise, there exists a function *reduce_basis* which implements

the LLL algorithm; it is proven that the result of *reduce_basis* is indeed reduced; and it is proven that the first vector of a reduced basis is short. However, there is no *formal* statement about the complexity of *reduce_basis*.

To this end, we extend the existing formalization in two important directions:

- We define an extended version of *reduce_basis* – *reduce_basis_cost* – which in addition to the result also returns the number of required arithmetic operations. We then show that the results of *reduce_basis_cost* and *reduce_basis* are identical. And more importantly, we formally verify the following polynomial upper bound on the number of arithmetic operations. Here, A is the maximum squared norm of the input vectors, and *Log* is the logarithm of A w.r.t. the basis $\frac{4\alpha}{4+\alpha}$. Note that *Log* is polynomially bounded in the size of the binary representation of the input vectors. The reason is that the basis of the logarithm is larger than 1 because $\alpha > \frac{4}{3}$.

  lemma *reduce_basis_cost_expanded:*
  assumes $A =$ *max_list* (*map* (*nat o sq_norm*) *fs*)
  and *Log_A* $=$ *nat* $\lceil$ *log* $(4 \cdot$ *of_rat* $\alpha)$ / $(4 +$ *of_rat* $\alpha))$ $A\rceil$
  shows *cost* (*reduce_basis_cost fs*) $\leq$
  $\big(4 \cdot m^2 + 3 \cdot m + (4 \cdot m^2 + 12 \cdot m) \cdot (1 + 2 \cdot m \cdot$ *Log_A*$)\big) \cdot n \cdot$ *arith_cost*

- Since the cost of an arithmetic operation – *arith_cost* – clearly depends on the size of the numbers, we also formally prove size bounds on the numbers that occur during the execution of the algorithm.

  lemma *combined_size_bound:* **assumes** ...
  and $A =$ *max_list* (*map* (*nat o sq_norm*) *fs*)
  and $M =$ *Max* $\{$*abs* (*fs ! i $ j*) $\mid$ *i j. i* $<$ *m* $\wedge$ *j* $<$ *n*$\}$
  and *x* $\in$ ... `(* description of numbers during run of algorithm *)`
  and *quotient_of x* $=$ (*num, denom*)
  and *number* $\in \{$*num, denom*$\}$
  shows *log 2* $|$*number*$|$ $\leq$ *2* $\cdot$ *m* $\cdot$ *log 2 A* $+$ *m* $+$ *log 2 m*
  and *log 2* $|$*number*$|$ $\leq$ *4* $\cdot$ *m* $\cdot$ *log 2* $(M \cdot n) + m +$ *log 2 m*

  Here, *x* is some number that occurs during the execution of the algorithm, *number* is the numerator or denominator of *x*, and the lemma states that the size of *number* is roughly $\mathcal{O}(m \cdot \log_2(M \cdot n))$ where M is the maximum absolute value that occurs in the input *fs*.

We first recall the existing formalization in Section 2, then present details on the formal bound on the number of arithmetic operation in Section 3 and finally illustrate the bounds on the size of the numbers in Section 4.

Our formal proofs are available in the development version of the AFP, theories `LLL_Complexity.thy` and `LLL_Number_Bounds.thy` [2]. They are based on definitions and proofs from a textbook on computer algebra [11, Chapter 16].

2 The Formalized LLL Algorithm

In this section we briefly recapitulate the existing Isabelle/HOL formalization of the LLL algorithm.

One ingredient of the LLL algorithm is the Gram–Schmidt orthogonalization (GSO) procedure. It takes a list of linearly independent vectors $f_0, \ldots, f_{m-1}$ from $\mathbb{R}^n$ or $\mathbb{Q}^n$ as input, and returns an orthogonal basis $g_0, \ldots, g_{m-1}$ for the space that is spanned by the input vectors. In this case, we also write that $g_0, \ldots, g_{m-1}$ is the *GSO* of $f_0, \ldots, f_{m-1}$. This procedure has already been formalized in Isabelle as a function *gram_schmidt* when proving the existence of Jordan normal forms [10].

The formalization of the LLL algorithm follows the pseudo-code presented in Algorithm 1. Here, $\lfloor x \rceil = \lfloor x + \frac{1}{2} \rfloor$ is the integer nearest to x; $u \bullet v$ is the inner product of vectors u and v, and $\|u\|^2 = u \bullet u$ is the squared Euclidean norm of u. Moreover, the function μ is defined as follows where f and g always refer to the current values of f and g in Algorithm 1.

$$\mu_{i,j} := \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } j > i \\ \frac{f_i \bullet g_j}{\|g_j\|^2} & \text{if } j < i \end{cases} \tag{1}$$

Algorithm 1: The LLL basis reduction algorithm, verified version

Input: A list of linearly independent vectors $f_0, \ldots, f_{m-1} \in \mathbb{Z}^n$ and $\alpha > \frac{4}{3}$
Output: A basis that generates the same lattice as $f_0, \ldots, f_{m-1}$ and is reduced w.r.t. α

1 $i := 0$; $g_0, \ldots, g_{m-1} :=$ *gram_schmidt* $f_0, \ldots, f_{m-1}$
2 **while** $i < m$ **do**
3 **for** $j = i - 1, \ldots, 0$ **do**
4 $f_i := f_i - \lfloor \mu_{i,j} \rceil \cdot f_j$
5 **if** $i > 0 \wedge \|g_{i-1}\|^2 > \alpha \cdot \|g_i\|^2$ **then**
6 $g'_{i-1} := g_i + \mu_{i,i-1} \cdot g_{i-1}$
7 $g'_i := g_{i-1} - \frac{f_{i-1} \bullet g'_{i-1}}{\|g'_{i-1}\|^2} \cdot g'_{i-1}$
8 $(i, f_{i-1}, f_i, g_{i-1}, g_i) := (i - 1, f_i, f_{i-1}, g'_{i-1}, g'_i)$
  else
9 $i := i + 1$
10 **return** $f_0, \ldots, f_{m-1}$

Algorithm 1 has been formalized in Isabelle as follows where the presentation in this paper hides some refinements. Throughout the paper, we are presenting Isabelle source which reside in a context that fixes the approximation factor α, the dimensions n and m, and the basis fs_{init} of the initial lattice L.

```
type_synonym state = nat × int vec list × rat vec list

definition basis_reduction_step :: state ⇒ state where
  basis_reduction_step = ...              (* implementation of lines 3−9 *)

partial_function (tailrec) basis_reduction_main :: state ⇒ state where
  basis_reduction_main state = (case state of (i,fs,gs) ⇒
    if i < m
    then basis_reduction_main (basis_reduction_step state)
    else state)

definition reduce_basis :: int vec list ⇒ int vec list where
  reduce_basis fs = (let gs = gram_schmidt (RAT fs) in
    case basis_reduction_main (i, fs, gs) of (_, fs', _) ⇒ fs')
```

- We define an Isabelle type *state*, which is just a triple (*i, fs, gs*) which stores the index i, the lattice basis $f_0, \ldots, f_{m-1}$ as a list of vectors *fs*, and the GSO $g_0, \ldots, g_{m-1}$ as list of vectors *gs*.
  Note that the triple representation is a simplified version of the actual formalization. In reality, we also store the norms of each g_i and split the lists *fs* and *gs* at position i so that accessing the i-th element can be done in constant time.
- The body of the while-loop, i.e., lines 3–9 is modelled by a function *basis_reduction_step* where we omit further details.
- The while-loop itself is modelled as partial function *basis_reduction_main*. Note that it is not terminating, since the termination depends on valid inputs, e.g., it will not terminate for $\alpha = 0$. Putting these assumptions into the context might be possible for proving properties of the LLL algorithm, but will prevent code-generation.
- The full algorithm is available as function *reduce_basis* which starts the loop after computing the initial GSO, and then just returns the final integer basis $f_0, \ldots, f_{m-1}$. Here, *RAT* converts a list of integer vectors into rational vectors.

The key correctness property of the LLL algorithm is present in the following lemma:

```
lemma basis_reduction_step:
  assumes LLL_invariant (i, fs, gs) and i < m
    and basis_reduction_step α (i, fs, gs) = (i', fs', gs')
  shows LLL_invariant (i', fs', gs')
    and LLL_measure (i', fs', gs') < LLL_measure (i, fs, gs)
```

It states that the invariant is preserved in the while-loop and that in each iteration the measure decreases. In the upcoming definition of the invariant, *lin_indpt_list* is a predicate expressing linear independence. The predicate *reduced* k *gs* (μ (*RAT fs*)) demands that the first k vectors of *fs* and *gs* are reduced, i.e., in particular $\mu_{i,j} \leq \frac{1}{2}$ for all $j < i < k$ where $\mu_{i,j}$ refers to the current values of *fs* and *gs*.

definition *LLL_invariant* (*i, fs, gs*) = (
gram_schmidt (*RAT fs*) = *gs* ∧
lin_indpt_list (*RAT fs*) ∧
lattice_of fs = *lattice_of fs_init* ∧
length fs = *m* ∧
reduced i gs (μ (*RAT fs*)) ∧
i ≤ *m*)

Based on the lemma *basis_reduction_step*, one can easily prove crucial properties of the LLL algorithm where *A* is the maximum squared norm of the initial lattice basis fs_{init}.

1. The resulting basis is reduced and is a basis for the same lattice as the initial basis.
2. The algorithm terminates, since the *LLL_measure* is decreasing in each iteration.
3. The number of loop iterations is bounded by *LLL_measure* (*i, fs, gs*) when invoking the algorithm on inputs (*i, fs, gs*), so *reduce_basis* requires at most *LLL_measure* (*i, fs, gs*) many iterations.
4. *LLL_measure* (*i, fs, gs*) ≤ $m + 2 \cdot m \cdot m \cdot$ *log* ($\frac{4 \cdot \alpha}{4+\alpha}$) *A*

These properties have all been stated and proven in the ITP paper [3]. Moreover, since every while-loop iteration requires $\mathcal{O}(m \cdot n)$ many arithmetic operations, one can derive a total bound of $\mathcal{O}(m^3 \cdot n \cdot \log A)$ arithmetic operations.

However, this has not been done formally, moreover there is no proven bound on the values of f_i, g_i and $\mu_{i,j}$.

3 A Formally Verified Bound on the Number of Arithmetic Operations

In this section we provide details on how lemma *reduce_basis_cost_expanded* was proven.

The first step to be able to reason about the number of arithmetic operations is to extend the whole algorithm by annotating and collecting costs. To this end, we use following cost model.

– We only count arithmetic operations, namely those that occur in vector operations.

– We use a parameter *arith_cost* for the cost of an arithmetic operations. It does not distinguish between rational and integer operations.
– An inner product and squared-norm calculation has costs $2 \cdot n \cdot$ *arith_cost* since there are n multiplications and $n-1$ additions to be performed. Vector addition and scalar multiplication has a cost of $n \cdot$ *arith_cost*.

To integrate this model formally, we use a simple approach. It has the advantage that it was very easy to integrate on top of the existing formalization: the formalization of this section requires only 450 lines and was finished within two days.

– We use a type *'a cost* = *'a* × *nat* to represent a result of type *'a* in combination with a cost for computing the result.
– For every Isabelle function *f :: 'a ⇒ 'b* that is used to define the LLL algorithm, we define a corresponding extended function *f_cost :: 'a ⇒ 'b cost*. These extended functions use pattern matching to access the costs of sub-algorithms, and then return a pair where all costs are summed up.
– In order to state correctness, we define two selectors *cost :: 'a cost ⇒ nat* and *result :: 'a cost ⇒ 'a*. Then soundness of *f_cost* is split into two properties. The first one states that the result is correct: *result* (*f_cost x*) = *f x*, and the second one provides a cost bound *cost* (*f_cost x*) ≤ ....

We did not use state monads – which accumulate the costs – to model the functions *f_cost*. The reason is that we would then always have to break the monad abstraction in order to formally prove the cost bounds.

We illustrate our approach using two example functions where *basis_reduction_swap_cost* implements lines 6–8 of Algorithm 1, and *basis_reduction_main_cost* is an annotated version of *basis_reduction_main* as it is defined in Section 2.

```
fun basis_reduction_swap_cost (i,fs,gs) = (
 case compute_mu_cost fs gs i (i−1) of                (* compute μ_{i,i−1} *)
  (mu,c1) ⇒ let                                  (* c1: costs of sub−routine *)
  gi = ...;                                          (* extract g_i from gs *)
  gim1 = ...;                                      (* extract g_{i−1} from gs *)
  gim1' = gi + mu ·_v gim1;                         (* count 2n operations *)
  ...;
  c2 = (2 + ... ) · n · arith_cost                 (* c2: local costs in let *)
  in ((i−1, fs', gs'), c1 + c2)                                (* sum up costs *)

function basis_reduction_main_cost state = (case state of (i,fs,gs) ⇒
  if i < m ∧ LLL_invariant state                    (* new: enforce invariant *)
  then case basis_reduction_step_cost state of
   (state1,c1) ⇒                          (* c1: cost for one loop iteration *)
    case basis_reduction_main_cost state1 of
```

```
      (state2,c2) ⇒                          (* c2: cost for recursion *)
       (state2, c1 + c2)                                  (* sum costs *)
  else (state,0))                                          (* 0 costs *)
```

The function *basis_reduction_swap_cost* is the usual case: one invokes sub-algorithms recursively and extracts their costs by pattern matching on pairs (here: *c1*), one does some local operations and manually annotates the costs for them (here: *c2*), and finally returns the pair of the computed result and the total cost.

Of course, one should validate the definitions, i.e., carefully inspect whether the formal cost definitions really match the intended cost model. However, verification of the desired soundness properties is then just a simple exercise.

lemma *basis_reduction_swap_cost:*
result (*basis_reduction_swap_cost state*) = *basis_reduction_swap state*
cost (*basis_reduction_swap_cost state*) ≤ *12* · *n* · *arith_cost*

The function *basis_reduction_main_cost* is a bit more interesting. Note that the corresponding function *basis_reduction_main* was defined as a *tail-recursive* **partial_function**. However, *basis_reduction_main_cost* is no longer tail-recursive since it has to adjust the costs from *c2* to *c1* + *c2*. As a consequence, *basis_reduction_main_cost* cannot be defined via **partial_function**, and instead we use the **function** command which requires a termination proof.

Recall that it depends on the input arguments whether the main loop is terminating or not. Therefore, to actually ensure termination we add the condition that the state must satisfy the invariant. Then we prove termination by a decrease of *LLL_measure* by using lemma *basis_reduction_step*. In order to get rid of the second precondition of the lemma – *basis_reduction_step* α (*i, fs, gs*) = (*i′, fs′, gs′*) – we utilize the soundness lemma for *basis_reduction_step_cost* which includes the property *result* (*basis_reduction_step_cost state*) = *basis_reduction_step state*.

Although the reasoning for *basis_reduction_main_cost* is a bit more complicated than the one for *basis_reduction_swap_cost*, the final statement is similar easy. In the statement, *body_cost* and *num_loops* are constants which encapsulate arithmetic expressions for the bounds.

lemma *basis_reduction_main_cost:* **assumes** *LLL_invariant state*
shows *result* (*basis_reduction_main_cost state*) = *basis_reduction_main state*
and *cost* (*basis_reduction_main_cost state*) ≤ *body_cost* · *num_loops*

In the same way we also define the final function *reduce_basis_cost* and prove its soundness and cost bound, e.g. the lemma *reduce_basis_cost_expanded* from the introduction. As final remark in this section, we do not want to hide that this lemma – our first main result – has some (obvious) assumptions that have not been mentioned in the introduction: $\alpha > \frac{4}{3}$, *m* ≠ *0*, *length fs* = *m*, and *lin_indpt_list* (*RAT fs*).

4 Bounds on the Numbers in the LLL Algorithm

Whereas the previous section provides a formally verified upper bound on the number of arithmetic operations, in this section we consider the costs of each individual arithmetic operation. To be more precise, we formally derive bounds on the f_i, g_i, and $\mu_{i,j}$. Whereas the bounds for g_i will be valid throughout the whole execution of the algorithm, the bounds for the f_i depend on whether we are inside the or outside the for-loop in lines 3–4: within the for-loop the value of the $\|f_i\|$ can get slightly larger than outside the loop.

To formally verify bounds on the numbers, we first define a stronger invariant which includes the conditions *f_bound outside fs* and *g_bound gs*.

```
definition f_bound outside k fs = (∀ i < m. ‖fs ! i‖² ≤
  (if outside ∨ k ≠ i then A · m else 4^(m−1) · A^m · m²))
definition g_bound gs = (∀ i < m. ‖gs ! i‖² ≤ A)
definition LLL_bound_invariant outside (i,fs,gs) =
  (LLL_invariant i fs ∧ f_bound outside i fs ∧ g_bound gs)
```

Note that *LLL_bound_invariant* does not enforce a bound on the $\mu_{i,j}$, since such a bound can be derived from the bounds on *fs* and *gs*. Here, we use the *Gramian determinant* which is the determinant of the *Gramian matrix* M. Given a set of vectors $v_1, \ldots, v_n$ the entries of M are $M_{ij} = v_i \bullet v_j$. There are multiple ways to define and describe the Gramian matrix and determinant in Isabelle:

```
definition Gramian_matrix fs k = (let M = mat k n (λ(i, j). (fs ! i) $ j) in M · M^T

definition Gramian_determinant fs k = det (Gramian_matrix fs k)

lemma assumes k < m
  shows Gramian_matrix fs k = mat k k (λ(i, j). fs ! i • fs ! j)

lemma Gramian_determinant:
  assumes LLL_invariant (i, fs, gs) and k ≤ m
  shows Gramian_determinant fs k = (∏ j<k. ‖gs ! j‖²)
```

Apart from our use case, the Gramian determinant also appears when proving the termination of the LLL algorithm, since it is used to define *LLL_measure*.

With the definition of the Gramian determinant and bounds on the vectors in *fs* we can derive bounds on all $\mu_{i,j}$ where $j < i$:

```
lemma mu_bound_Gramian_determinant:
  assumes j < i and i < m
  shows (μ fs i j)² ≤ Gramian_determinant fs j · ‖fs ! i‖²
```

The proof of this fact is rather straightforward and follows closely the one from [11, Chapter 16]. The proof uses Cauchy's inequality ($\|u \bullet v\|^2 \leq \|u\|^2 \cdot \|v\|^2$) which we had to show for our vector library.

Bounds on the Gramian determinants can be directly derived from the lemma *Gramian_determinant* and *g_bound gs*:

lemma *Gramian_determinant_bound:*
assumes *LLL_invariant* (*i, fs, gs*) **and** *g_bound gs* **and** $k < m$
shows *Gramian_determinant fs k* $\leq A^k$

Note that *f_bound outside fs* clearly gives a bound on the sizes of the values in each vector f_i. However, the situation is less clear for *g_bound gs*: even if $\|g_i\|^2$ is bounded, this does not imply a bound on the numerators and denominators of each number. The same is true for all $\mu_{i,j}$.

To derive bounds on the numerators and denominators appearing in the LLL algorithm, Cramer's lemma becomes important. Although this lemma is already available in the Isabelle distribution, there are two obstacles before we can use it in our proof.

lemma *cramer_lemma_distribution:* **fixes** *A :: real^'n^'n*
shows *det* (*replace_col_hma A* (*A* $\cdot_v$ *x*) *k*) = *x* $ *k* · *det A*

The first problem is that Cramer's lemma is available in HOL-analysis which uses Harrison's technique to represent vector dimensions via type variables [4]. In contrast, the whole LLL algorithm is formalized using the matrix- and vector-library of the AFP-entry [9] where the dimension is a natural number.

To solve this first problem, we mainly use a recent connection that permits to transfer theorems between the two matrix libraries [1, Section 4]. In our application, it just requires one additional transfer-rule to establish a link between the two constants in the two libraries that replace a row by a matrix. This transfer-rule is easy to prove, and afterwards Cramer's lemma can be transferred immediately: The following Isabelle source contains the full proof for lemma *cramer_lemma_real*.

lemma *HMA_M_replace_col[transfer_rule]:*
(*HMA_M* ===> *HMA_V* ===> *HMA_I* ===> *HMA_M*)
replace_col replace_col_hma `(* simple proof, not displayed *)`

lemma *cramer_lemma_real:* **fixes** *A :: real mat*
assumes *A* ∈ *carrier_mat n n* **and** *x* ∈ *carrier_vec n* **and** $k < n$
shows *det* (*replace_col A* (*A* $\cdot_v$ *x*) *k*) = *x* $ *k* · *det A*
using *assms cramer_lemma_distribution[untransferred, cancel_card_constraint]*
by *auto*

The second problem is that Cramer's lemma in HOL-analysis is only available for real-valued vectors and matrices, but we need it for rational numbers. Here, we use existing homomorphism lemmas for determinants and matrix multiplication, to obtain Cramer's lemma for rational matrices. The resulting lemma is exactly like *cramer_lemma_real*, except that *A :: rat mat*.

The g_i are defined recursively with $g_i = f_i - \sum_{j<i} \mu_{ij} g_j$ with the definition of μ_{ij} from 1. We can then prove that there exists a definition of g_i which is not based on the other vectors from *gs* and only on vectors from *fs*. We label the parameters in these new equations with λ_{ij} and attain the definition $g_i = f_i - \sum_{j<i} \lambda_{ij} f_j$. Each g_i is orthogonal to every f_l with $l < i$ and therefore $f_l \bullet g_i = f_l \bullet f_i - \sum_{j<i} \lambda_{ij}(f_l \bullet f_j) = 0$. So the λ_{ij} form a solution to a system of linear equations:

$$\underbrace{\begin{pmatrix} f_1 \bullet f_1 & \dots & f_1 \bullet f_{i-1} \\ \vdots & \ddots & \vdots \\ f_{i-1} \bullet f_1 & \dots & f_{i-1} \bullet f_{i-1} \end{pmatrix}}_{=M=\textit{Gramian\_determinant fs i}} \cdot \underbrace{\begin{pmatrix} \lambda_{i1} \\ \vdots \\ \lambda_{i(i-1)} \end{pmatrix}}_{=L} = \underbrace{\begin{pmatrix} f_1 \bullet f_i \\ \vdots \\ f_{i-1} \bullet f_i \end{pmatrix}}_{=F}$$

The coefficient matrix M on the left hand side where $M_{ij} = f_i \bullet f_j$ is exactly the Gramian matrix of *fs* and *i*. With Cramer's lemma we deduce

$$\begin{aligned} \lambda_{ij} \cdot \textit{Gramian\_determinant fs i} &= L \; \$ \; j \cdot \textit{det } M \\ &= \textit{det (replace\_col } M \; (M \cdot_v L) \; j) \\ &= \textit{det (replace\_col } M \; F \; j) \end{aligned}$$

The matrix *replace_col M F j* only has inner products of the vectors in *fs* as entries and these are of course integers. Then the determinant is also an integer and $\lambda_{ij} \cdot$ *Gramian_determinant fs i* $\in \mathbb{Z}$. Unfolding the definition of g_i where $g_i = f_i - \sum_{j<i} \lambda_{ij} f_j$ in *Gramian_determinant fs i* $\cdot_v$ g_i leaves us with sums and differences only consisting of integers. We have therefore proven the following lemma:

lemma
assumes *LLL_invariant* (*k*, *gs*, *fs*)
and $i < m$ **and** $j < n$
shows (*Gramian_determinant fs i* $\cdot_v$ *gs ! i*) *$ j* $\in \mathbb{Z}$

In the following we define $d_j =$ *Gramian_determinant fs j*. Considering lemma *Gramian_determinant* we have $\frac{d_{j+1}}{d_j} = \|g_j\|^2$ and with the previous lemma we deduce and state in Isabelle:

$$d_{j+1}\mu_{i,j} = d_{j+1}\frac{f_i \bullet g_j}{\|g_j\|^2} = d_{j+1}\frac{f_i \bullet g_j}{d_{j+1}/d_j} = d_j(f_i \bullet g_j) = f_i \bullet (d_j \cdot_v g_j) \in \mathbb{Z}$$

lemma
assumes *LLL_invariant* (*k*, *gs*, *fs*)
and $i < m$ **and** $j < n$
shows *Gramian_determinant fs* (*Suc j*) $\cdot\ \mu$ *fs i j* $\in \mathbb{Z}$

These lemmas now allow us to derive bounds on the numerators and denominators of all g_i and $\mu_{i,j}$. As the Gramian determinants are integers and as they are a multiple of the denominators of an g_i or an $\mu_{i,j}$, the denominators of the g_i and $\mu_{i,j}$ must be bounded by the Gramian determinants. In combination with upper bounds for the rational numbers, we can easiliy derive an upper bound on the numerator. We can state these properties in Isabelle:

lemma *quotient_of_bounds:*
fixes *i::int*
assumes *quotient_of r* = (*num*, *denom*) **and** $i \cdot r \in \mathbb{Z}$ **and** $0 < i$ **and** $|r| \leq b$
shows $|num| \leq i \cdot b$ **and** $denom \leq i$

lemma *LLL_invariant_g_bound:*
assumes *LLL_bound_invariant outside* (*k*, *fs*, *gs*) **and** $i < m$ **and** $j < n$
and *quotient_of* (*gs ! i $ j*) = (*num*, *denom*)
shows $|num| \leq A^m$ **and** $|denom| \leq A^m$

lemma *LLL_invariant_mu_bound:*
assumes *LLL_bound_invariant outside* (*k*, *fs*, *gs*) **and** $i < m$ **and** $j < n$
and *quotient_of* (μ *fs i j*) = (*num*, *denom*)
shows $|num| \leq 2^m \cdot m \cdot A^{2m}$ **and** $|denom| \leq A^m$

Based on these bounds on the numerators and denominator, it was an easy task to derive the size-bounds on the numerators and denominators that are stated in the introduction in Lemma *combined_size_bound*.

At this point we have proven that we can infer polynomial size bounds from *LLL_bound_invariant*. Moreover, the Isabelle sources also contain formal proofs for the one remaining task, namely to show that *LLL_bound_invariant* is indeed an invariant that is maintained throughout the execution of the algorithm: during the execution of the for-loop in lines 3–4, *LLL_bound_invariant False* (*i*, *fs*, *gs*) is satisfied; and outside the for-loop, *LLL_bound_invariant True* (*i*, *fs*, *gs*) is satisfied.

Acknowledgments

This research was supported by the Austrian Science Fund (FWF) project Y757. The authors are listed in alphabetical order regardless of individual contributions or seniority.

References

1. J. Divasón, S. Joosten, O. Kunčar, R. Thiemann, and A. Yamada. Efficient certification of complexity proofs: Formalizing the Perron–Frobenius theorem (invited talk paper). In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, pages 2–13. ACM, 2018.
2. J. Divasón, M. Haslbeck, S. Joosten, R. Thiemann, and A. Yamada. A verified LLL algorithm. *Archive of Formal Proofs*, Feb. 2018. `http://isa-afp.org/entries/LLL_Basis_Reduction.html`, Formal proof development.
3. J. Divasón, S. Joosten, R. Thiemann, and A. Yamada. A formalization of the LLL basis reduction algorithm. In *ITP 2018*, LNCS, 2018. To appear.
4. J. Harrison. The HOL light theory of Euclidean space. *J. Autom. Reasoning*, 50(2):173–190, 2013.
5. A. K. Lenstra, H. W. Lenstra, and L. Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261:515–534, 1982.
6. D. Micciancio. The shortest vector in a lattice is hard to approximate to within some constant. *SIAM J. Comput.*, 30(6):2008–2035, 2000.
7. P. Q. Nguyen and B. Vallée, editors. *The LLL Algorithm – Survey and Applications*. Information Security and Cryptography. Springer, 2010.
8. T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
9. R. Thiemann and A. Yamada. Matrices, Jordan normal forms, and spectral radius theory. *Archive of Formal Proofs*, Aug. 2015.
10. R. Thiemann and A. Yamada. Formalizing Jordan normal forms in Isabelle/HOL. In *CPP 2016*, pages 88–99. ACM, 2016.
11. J. von zur Gathen and J. Gerhard. *Modern computer algebra (3rd ed.)*. Cambridge University Press, 2013.