格子の最短ベクトル問題に対する離散的考察と並列計算アルゴリズム

柏原賢二 kashiwa@idea.c.u-tokyo.ac.jp

東京大学大学院総合文化研究科広域システム科学系

概要

格子の最短ベクトル問題は、短い格子ベクトルをいかに効率的に見つけるかという問題である。これまでは格子ベクトルの生成時に直交基底ベクトルに対する係数が一様に分布するというランダム仮定をもとに短い格子ベクトルが見つかる確率を推測してきた。ランダム仮定は連続的なモデルであり、実際には短いベクトルは離散的に分布しているので同じ格子ベクトルが複数回得られることがある。この論文では実験により格子ベクトルの重複について検証する。また基底の評価関数を用いた並列環境における効率的な基底簡約の方法を提案する。

キーワード: 格子基底簡約, 並列計算, Gram-Charlier 展開, 効率的アルゴリズム

1 はじめに

この論文では格子の最短ベクトル問題 (Shortest Vector Problem, SVP 問題) について、大規模並列計算機を利用した環境における効率的なアルゴリズムについて研究する。ユークリッド空間において、整数成分を持つ正則行列 (基底行列と呼ぶ) が与えられると、その行ベクトル (基底ベクトルと呼ぶ) たちの整数倍を足し合わせて作られる点を全部集めることで格子が生成される。格子の最短ベクトル問題とは、基底が与えられたときに、基底によって張られる格子点のうち、原点以外で原点にもっとも近い点を見つける問題である。この問題の難しさは公開鍵暗号システムである格子暗号システムの安全性の根拠になっている。格子暗号システムは、量子アルゴリズムでも効率的な解法が見つかっていない耐量子暗号として知られている。

SVP 問題に対する多くの効率的なアルゴリズムでは、最短ベクトル問題を基底簡約問題に帰着させて解いている。基底簡約問題とは、与えられた格子の基底に対して、同じ格子を生成する、より簡約された基底を求める問題である。基底簡約とは、基底をより「小さい」基底に基底変換することである。「小さい」の意味はあとで定義される。基底は簡約されるされることにより、短い格子ベクトルが見つけやすくなる。

1.1 SVP Challenge と近似的 SVP 問題

SVP Challenge は、ドイツのダルムシュタット工科大学によって運営されている最短ベクトル問題をインターネット上で公開しているチャレンジサイト [7] である。各次元ごとの基底行列が問題として公開されていて、与えられた基準よりも短くて、すでに提出されたものより短い格子ベクトルを見つけた人がエントリーできる。基本的にランダムに格子を作っているので、問題作成者であるサイトの運営者ももっとも短いベクトルを知らない。最短の格子ベクトルという厳密解を見つけるよりも、なるべく短時間で条件に合う長さのベクトルを見つけることが目標であるという近似的な SVP 問題を考えることになる。SVP Challenge において登録できる長さの基準は、最短ベクトルの長さの推測値よりも、1.05 倍以下であるというものである。

基底簡約による SVP のアルゴリズムでは、基底を使って短い格子ベクトルを生成するフェーズと、得られた短いベクトルを使って基底簡約を行うフェーズを繰り返すことになる。ベクトル生成アルゴリズムにおいて、基底行列自体が乱数生成器のようになっていて、格子ベクトルはランダムに生成するとみなすことができる。連続的なモデルを考えると生成ベクトルの長さは確率分布に従って分布することになる。決められた長さ以下の格子ベクトルが見つかるまでの時間の期待値をより短くするようなアルゴリズムを作るのが目標になる。SVP Challenge にエントリーしようというのがそもそものこの研究を始めた動機のひとつであった。また、格子暗号システムに対するチャレンジにおいても必ずしも最短ベクトル問題の厳密解を確実に求める必要はなく、近似的な SVP 問題を考えればよいことが多い。よって、本研究では、近似的 SVP 問題を考えることにする。

1.2 これまでの状況

私の所属した研究グループも 2013 年から SVP Challenge に取り組み [10, 24]、当時、利用していた基底簡約アルゴリズムは基本的には Schnorr らの RSR アルゴリズム [22] を独自に改良したものであり、ベクトル生成には、enumeration を利用していた。並列化してスーパーコンピュータ上に実装することにより、2018 年までは SVP Challenge において、当時のもっとも高次元 (152 次元など) の記録をエントリーしていた。しかし、最近は、sieving というアルゴリズムを利用した Ducas らのプログラム [9, 1, 8] がもっとも高い次元の記録 (180 次元) をエントリーしている。Ducas らの開発した G6K と呼ばれる基底簡約プログラムは、基底簡約の枠組みの中に射影空間 (直交補空間) での sieving を取り入れてた非常に効率的なものである。Laarhoven [17] らも類似のアイディアの研究をおこなっている。sieving は、既知のベクトル集合の和や差を考えることによって短い格子ベクトルを探していく手法である。sieving は大量のメモリを必要とするために高い次元で行うことは従来は困難であったが、射影空間という低い次元で sieving を行うことにより射影空間で短いベクトルを見つけて、それを全体空間に Babai lifting で拡張することにより、短い格子ベクトルを探した。

われわれのグループが SVP Challenge の記録を出していたころのプログラムは、C++ で記述した完全に独自のプログラムで、格子計算の分野で著名な FPLLL [25] のライブラリも使っていなかった。Ducas らの開発した G6K に SVP Challenge のトップの記録を塗り替えられた 2018 年よりあとは、彼らの論文を参考に、しばらく独自実装の sieving のアルゴリズムを試していたが、なかなか効率がよいものが作れずに、われわれは完全独自開発を諦めた。そして、公開されている G6K のソースプログラムを利用して、それに基底簡約のやり方をわれわれの方法に近づけたりプロセス並列化の取り入れるなどして、大幅に書き換えることによりアルゴリズムの改善に取り組むことにした。そのプログラムの概要を提案アルゴリズムとしてこの論文に記載する。

1.3 提案プログラムと基底簡約時のベクトルの選択

基底簡約問題に帰着させて最短ベクトル問題を考える場合は、基底簡約を何回も繰り返すことにより、基底をよいものにすることが目標となる。すなわち、現在の基底を利用して、大量の格子ベクトルを生成する部分 (格子ベクトルのサンプリングとも呼ばれる) と、その生成されたベクトルのうち基底簡約に利用可能な格子ベクトルを取り出して、その格子ベクトルのうちのどれかを選択して基底簡約を行うということを繰り返す。よって基底簡約アルゴリズムはベクトル生成をどのようにして行うのかと、サンプリングされたどの短い格子ベクトルを使って、基底簡約を行うのかという部分に分けることができる。G6K においては、実行速度が重要な sieving 部分や Babai lifting などは C++ で実装し、ユーザーインターフェースや、基底簡約に関する部分は Python や Cython で作られている。Ducas らのプログラムである G6K は GitHub で GPL v2 ライセンスで公開されている [26]。

われわれは G6K を元に、私たちのもともと作っていた基底簡約のアルゴリズムに近いように改造した。元になっている G6K のソースプログラムは 2021 年の初めごろのものである。その後も G6K は GPU を利用するなど高速化の進化を続けている [8]。ここで現在のわれわれのプログラムの概要を紹介する。C++ と Python と Cython を使うというプログラムの枠組みは基本的に G6K のものを利用している。Cython は Python から C++ のルーチンを呼ぶインターフェースとして利用している。また、基底簡約部分を中心に適宜、FPLLL のライブラリも利用している。われわれのアルゴリズムにおいてはベクトル生成部分は、G6K による sieving の計算ルーチンをほぼ、そのまま利用している。G6K は、いくつかの sieving の方式を選択できるが、われわれは並列化された triple sieve を利用して計算を行なっている。triple sieve は、格子ベクトルの 3 つ組を足したり引いたりして短いベクトルを探す手法で、保存しているベクトル数に比較して、多くの範囲のベクトルを生成することが可能である。triple sieve により射影された直交補空間で sieving を行い、それで得られた短いベクトルを Babai Lifting で全体空間に拡張している。どのベクトルを基底簡約に用いるかという基底簡約部分と、プロセス並列化の部分は、独自に新しく作っている。基本的に Python を用いて記述されている。これは、基底簡約部分は、私たちのグループがいままで RSR に近い手法で SVP に取り組んでいた方法を踏襲している。簡約に利用できる複数のベクトル候補のうち、評価関数を用いて、もっともよいベクトルを見つけるというものである。そのた

めに、基底行列の評価関数を導入し、その評価関数がなるべく下がる方向にいくようにベクトルを選択する。基底の評価関数は、直交基底ベクトルの長さに index ごとの重みをつけて足した数値で表現される。評価関数は、その基底がその時点でどのくらい短いベクトルを生成する能力があるかの指標と必ずしも一致しないし、基底簡約の進み具合を表す、直交基底ベクトルの長さの列の全順序とも一致しない。その 2 つの指標の間ともいえる量となっている。その時点での短いベクトルを生成する能力だけでなく、将来的に短いベクトルを生成する能力がある基底に早く到達できるかを重視している。

index の先頭から、与えられた個数の基底ベクトルたちすべてに直交するベクトルで作る直交補空間において、格子ベクトルをその空間に射影して作った格子を考える。基底簡約は、射影した格子の中で直交基底ベクトルよりも短い格子ベクトルを探すことで行われる。基底簡約のフェーズにおいて、簡約に利用できる格子ベクトルが見つかったらとにかく簡約するのではなく、もっとも理想とする基底にいかに早くたどり着くかのルートを選んで簡約していくというのがわれわれのアプローチの特徴である。そのために、われわれのアルゴリズムでは、見つかった短いベクトルを次元ごとに、もともとの座標系の整数座標の形式でなるべく長い期間、保持することを行なっている。利用されなかった短いベクトルデータは各基底簡約後に捨てられるのではなく、基底簡約のタイミングを超えて保持される。G6K においては、基底行列の係数表現により格子ベクトルを表していたので、基底変換を超えて、短い格子ベクトルのデータを十分に保持できなかった。これが G6K を改造しようと思った動機のひとつであった。基底簡約につかう格子ベクトルを、基底の評価関数をなるべく良くするように選ぶような、ベクトル選択のための評価のシステムも作った。

1.4 連続モデルの限界とベクトルの重複について

本研究で構築する基底簡約アルゴリズムの目標は、ゴールとして与えられた長さ以下の格子ベクトルをなるべく短時間で見つけることができるようなプログラムを作ることである。これまでの生成ベクトルの分布の研究は、連続的なモデルを使って、基本的にひとつの基底から作られる重複のないベクトルを前提にしてきた。基底が固定であれば、ランダム仮定のような連続的なモデルを用いて、ベクトル生成時において、その基底から生成される格子ベクトルの長さの分布を推測することができる [18, 19]。固定された基底を用いて実際に生成した格子ベクトルの長さの分布は、連続的な仮定でのモデルによく合っていることが知られている。連続的なモデルではすべて異なるベクトルが生成されることになるが、実際の格子ベクトルの空間は離散的であり、基底簡約プログラムにおいて基底は時間とともに変化していき、その結果、生成される格子ベクトルは過去にも生成されたベクトルと同じ座標であることがある。これを生成ベクトルの重複と呼ぶことにする。いくら短い格子ベクトルがたくさん生成されても、それが過去に得られたものと同じベクトルであると、短いベクトルを探索する効率が落ちることになる。

2 つの基底から生成するベクトルが重なりやすいときに、基底が近いと考えることにする。この論文ではどういうときに基底が近いかを考察する。短い格子ベクトルを見つけるためには、プログラムで生成される格子ベクトルの長さの分布をよいものにする必要がある。基底簡約によって、生

成するベクトルの分布がよいものになるような基底に変えていく。基底をよくするために直交基底ベクトルの長さの自乗の和を下げようとすると、前の index のほうから徐々に短い基底ベクトルを固定していくことになる。すると、新しい基底と過去の基底の、先頭からの基底ベクトルが共通になってしまって、基底が似てくる。すると生成可能な格子ベクトルの空間が重なり、ベクトルの重複が増えてしまう可能性がある。基底をよくすることのメリットとデメリットの具合を探るのがこの論文のテーマの一つである。

1.5 並列化について

困難な問題に対する計算機による高速な求解には、並列計算は欠かせない。並列計算には、大きく分けて、一般に高速にアクセスできるメモリを CPU 間で共有しているスレッド並列と、比較的低速なアクセスのファイルストレージ (分散ファイルシステム) を使った情報共有や MPI などを利用したプロセス間通信をせざるをえないプロセス並列のプログラムに分けられる。簡約計算の並列化に関する既存の研究は、理論的なものかあるいはスレッド並列によるものが多かった [12, 6, 27, 2, 15]。また、GPU を使った並列計算の研究も行われている [13]。プロセス並列計算を用いた基底簡約の研究としては、[5] がある。

われわれのプログラムは並列計算の部分は、MPI を用いたプロセス並列で計算している。それは、CPU 間でメモリを共有していないコンピュータ環境を想定して作っている。Ducas らの G6K は sieving においてベクトルの和と差を計算するときなどに、スレッド並列の並列処理を利用している。スレッド並列なので、CPU 間でメモリを共有できるような環境を想定している。メモリ中にベクトル情報を保持して sieving を行うので、高次元の sieving を行う際には、巨大なメモリが必要となる。大きなメモリを使って、高次元の sieving を行うと非常に効率的に短い格子ベクトルが見つけられることが知られている。われわれが現在利用している並列計算機環境では、ノードごとにメモリの空間が独立していて、スレッド並列には必ずしも向いていない。このような環境はある程度、一般的だと思われる。われわれは、プロセス並列を用いたプログラムを作成した。並列ノード間でベクトル情報を共有するのではなく、基底情報をある程度、ゆるく共有することによって、並列処理を実現した。ここで、「ゆるく共有する」の意味は、並列ノード間で、常に同じ基底で計算するのではなく、よい基底が別のプロセスで得られたときに限り、基底を置き換えるというようなことを指している。ゆるく共有することで、基底ベクトルが似てくるのを防いでいる。基底情報の共有には、共有ストレージを用いて行っている。MPI の仕組みは、プログラムの起動時のみに用いていてプロセス間通信は行っていない。もともと G6K の triple sieve などに組み込まれているスレッド並列は利用可能である。

プロセス並列を用いた並列計算の概要を述べる。各プロセスはそれぞれ基底をもっている。プログラム起動時、すべてのプロセスは同じ基底でスタートするが、利用しているベクトル生成ルーチンは乱数を用いているので、異なるプロセスは、異なる計算を行い、次第に基底がばらばらになっていく。よい基底を得たプロセスが、そのプロセスで保持している基底情報を、BEST BASIS として共有ファイルシステムにあるファイルに書き込む。BEST BASIS は常にたかだかひとつの基

底しか保持できないとする。ここでよい基底とは直交基底ベクトルの長さの列の全順序の意味において、簡約がより進んでいるという点と、基底の評価関数において、よりよい値であるという 2 つの面から評価する。どちらの観点でも同時に優れていると認められるときに、既存の BEST BASIS を置き換える。

1.6 論文概要まとめ

まとめると、本論文には以下の 3 つの柱がある。

1. 基底簡約において利用できるベクトルをやみくもに選ぶのではなく、複数の候補からもっとも目標とする基底に効率よく辿り着けるようなルートを実現できるようなベクトルを選ぶ。そのために基底の評価関数を導入する。
2. メモリの少ないような計算機環境でも効率的に計算できるような並列アルゴリズムの構築を目指す。そのために、基底の評価関数を用いたプロセス並列化のアルゴリズムを提案する。
3. ベクトルの重複を通して、基底の近さについて考察する。基底を長期間簡約することにより、直交基底ベクトルの長さの自乗和が小さい基底が得られて、短いベクトルを見つける能力が上がっていくのか、それとも基底が似てきてベクトル重複が増えて効果が落ちるのかを観察する。

2 基底簡約についての一般論と数学的記法

2.1 格子の最短ベクトル問題と基底簡約アルゴリズム

整数成分からなる正則な N 次正方行列 $\mathbf{B} = (\mathbf{b}_0, \ldots, \mathbf{b}_{N-1})$ を考える。基底行列の列ベクトルを**基底ベクトル**と呼ぶことにする。基底行列の基底ベクトルには、順番がついていることになるが、その順番を表す数字を index と呼ぶことにする。index 全体の集合を $I = \{0 \leq i < N | i \in \mathbb{Z}\}$ と書く。基底ベクトルたちの整数結合全体で作られる点の集合 $\mathcal{L} = \{\sum_{i=0}^{N-1} u_i \mathbf{b}_i | u_i \in \mathbb{Z}\} \subseteq \mathbb{Z}^N$ を格子と呼ぶ。格子の点を表す位置ベクトルを格子ベクトルと呼ぶ。原点以外で原点にもっとも近い格子点を表すベクトルを最短ベクトルと呼ぶことにする。基底行列が与えられたときに、最短ベクトルを問題が最短ベクトル問題 (Shortest Vector Problem, SVP) である。Gaussian Heuristic と呼ばれる方法で格子の次元 N と格子の単位体積 $\det(\mathcal{L})$(=基底の行列式の絶対値) から最短ベクトルの長さが推測できて、$\frac{\Gamma(N/2+1)^{1/N}}{\sqrt{\pi}} \det(\mathcal{L})^{1/N}$ と書ける。ここで Γ はガンマ関数である。

2.2 直交基底ベクトルと直交補空間

空間を直交基底ベクトルにより分解することで、長さ (の自乗) を考えるときに直交成分ごとに考えればよくなる。基底行列 $\mathbf{B}$ に対して、以下のようなベクトル (以下、直交基底ベクトルと呼ぶ) の列 $(\mathbf{b}_k^*)_{k \in I}$ を考えることができる。これは、以下のような性質を満たすベクトルである。

1. N 個すべてのベクトルが直交している。すなわち異なる $i, j \in I$ に対して、$\langle \mathbf{b}_i^*, \mathbf{b}_j^* \rangle = 0$。
2. 任意の $k \in I$ に対して、基底行列の列ベクトル $\mathbf{b}_0, \ldots, \mathbf{b}_k$ とベクトル $\mathbf{b}_0^*, \ldots, \mathbf{b}_k^*$ で張られる線形空間が同じである。
3. 任意の $k \in I$ に対して、$|\mathbf{b}_k^*|^2 = \langle \mathbf{b}_k, \mathbf{b}_k^* \rangle$ が成り立つ。

基底行列 $\mathbf{B}$ に対して、グラムシュミット直交化のアルゴリズムにより、上の条件を満たすような N 個の直交基底ベクトル $(\mathbf{b}_k^*)_{k \in I}$ を計算することができる。直交基底ベクトルは、通常は非整数成分を持つので格子ベクトルではないことに注意する。直交基底ベクトルを列ベクトルとして持つ行列を直交基底行列 $\mathbf{B}^*$ と呼ぶ。

直交基底ベクトルの列 $(\mathbf{b}_k^*)_{k \in I}$ に対応して、直交基底ベクトルの長さの列 $(|\mathbf{b}_k^*|)_{k \in I}$ を考えることができる。基底行列の簡約の度合いを対応する直交基底ベクトルの長さの列を見て判定することになる。実際のプログラムでは、直交基底ベクトルの長さの列よりも、直交基底ベクトルの長さの自乗の列のほうが扱いやすいので、それぞれの長さを自乗した値の列 $(|\mathbf{b}_i^*|^2)_{i \in I}$ を考えている。これを基底の shape と呼ぶことにする。なお、直交基底ベクトルの列が与えられても、それに対応する基底行列は一意ではない。size reduction という基底変換で、自乗長さ列は変化しないのでそれ以上 size reduction ができないような唯一の size reduced な基底に対応させることができる。基底の size reduction とは、k 番目の基底ベクトル $\mathbf{b}_k$ を $i < k$ となる $\mathbf{b}_i$ の整数倍を足したもの $\mathbf{b}_k + u\mathbf{b}_i$ に置き換えることで、$-0.5 < \frac{\langle \mathbf{b}_k, \mathbf{b}_i^* \rangle}{\langle \mathbf{b}_i^*, \mathbf{b}_i^* \rangle} \leq 0.5$ となるように変換することである。この操作により、同じ格子 $\mathcal{L}$ を生成する上に、対応する自乗長さ列 $(|\mathbf{b}_i^*|^2)_i$ も不変であることに注意する。

次のように、基底行列とその index k に対して、k より前のすべての基底ベクトルに直交する点全体のなす空間に、全体空間を正射影した空間を考えることができる。

定義 2.1. 基底行列 $\mathbf{B}$ に対して、k-直交補空間とは、*0* 番から $k-1$ 番目までの基底ベクトルたちにすべて直交する点全体のなす空間 $\{\mathbf{x} \in \mathbb{R}^N | \langle \mathbf{x}, \mathbf{b}_m^* \rangle = 0 \ for \ m = 0, ..., k-1\}$ である。

命題 2.2. 基底行列 $\mathbf{B}$ に対して、格子 $\mathcal{L}$ から k-直交補空間への正射影する線形演算 π_k^B は、以下のようになる。

$$\pi_k^B : \mathcal{L} \to \pi_k^B(\mathcal{L}) \ s.t. \ \pi_k^B(\mathbf{v}) = \mathbf{v} - \sum_{m=0}^{k-1} (\frac{\langle \mathbf{v}, \mathbf{b}_m^* \rangle}{|\mathbf{b}_m^*|^2}) \mathbf{b}_m^* = \sum_{m=k}^{N-1} (\frac{\langle \mathbf{v}, \mathbf{b}_m^* \rangle}{|\mathbf{b}_m^*|^2}) \mathbf{b}_m^*.$$

また、$|\pi_k^B(\mathbf{v})|^2 = \sum_{m=k}^{N-1} (\frac{\langle \mathbf{v}, \mathbf{b}_m^* \rangle}{|\mathbf{b}_m^*|})^2$ が成り立つ。

$\mathbf{v}$ の直交空間への分解 $\mathbf{v} = \sum_{m=0}^{N-1} \frac{\langle \mathbf{v}, \mathbf{b}_m^* \rangle}{|\mathbf{b}_m^*|^2} \mathbf{b}_m^*$ に注意する。自乗長さのほうの関係式は、直交するベクトルの内積がゼロであることに注意して計算すればわかる。

実際にプログラムの中では、上のように後ろのほうの index の射影の長さの自乗を足すことで、直交補空間での長さの計算している。ただし、k-直交補空間での格子ベクトル $\mathbf{v}$ の長さの 2 乗の値 $|\pi_k^B(\mathbf{v})|^{\mathbf{2}}$ は、基底行列のうち、0 番目から $k-1$ 番目の基底ベクトルで決定され、それ以降の基底ベクトルにはよらないことに注意する。たとえば、基底変換で、index k 以降の基底ベクトル $(\mathbf{b}_i)_{i \geq k}$ が変わっても k-直交補空間での長さは不変になるということである。

格子 $\mathcal{L}$ の点を正射影 π_k^B によって k-直交補空間に射影することができ、また格子になる。射影した格子点全体は、$\pi_k^B(\mathcal{L})$ と書ける。$\mathbf{b}_k^* = \pi_k^B(\mathbf{b}_k)$ が成り立つ。k-直交補空間における、格子ベクトル $\mathbf{v} \in \mathcal{L}$ の長さは、$\pi_k^B(\mathcal{L})$ に射影したベクトル $\pi_k^B(\mathbf{v})$ の長さ、すなわち、$\mathbf{v}$ を 0 番目から $k-1$ 番目までの基底ベクトルに直交する成分のみの長さ $|\pi_k^B(\mathbf{v})|$ で考える。$\mathbf{v} = \sum_{i \in I} \nu_i \mathbf{b}_k^*$ と書いたときに、$|\pi_k^B(\mathbf{v})|^2 = \sum_{i=k}^{N-1} |\nu_i|^2 |\mathbf{b}_i^*|^2$ となる。

2.3 基底簡約と簡約順序

基底変換とは、$\mathcal{L}$ のある基底行列 $\mathbf{B}$ を同じ格子 $\mathcal{L}$ を張る、別の基底行列 $\mathbf{B}'$ に変換することである。$\mathbf{B}$ と $\mathbf{B}'$ は、整数成分のユニモジュラー行列 $\mathbf{U}$ を掛けたものになっている。$\mathbf{B} = \mathbf{U}\mathbf{B}'$ で $\det(\mathbf{U}) = \pm 1$ である。ここでの基底簡約とは、以下のような簡約順序で小さいものに基底変換するものとする。

基底行列全体に関して (正確には、直交基底ベクトルの長さの列、すなわち shape に関して)、以下のような辞書式順序を考える。先頭の index から直交基底ベクトルの長さを辞書式順序で比較して順序を決める。つまり前のほうの直交ベクトルの長さが同じであれば、はじめて長さが異なる index において直交基底ベクトルの長さを比べて、大小を定義する。LLL や BKZ などの基底簡約アルゴリズムは、この辞書式順序を小さくする方向で基底変換を行なっている。すなわち、基底変換前の基底行列よりも、基底変換後の基底行列のほうが必ず、辞書式順序において小さくなっている。ここでは、このように辞書式順序が前のものに基底変換することを基底簡約するということにする。

基底が与えられれば自乗長さ列が唯一定まるが、逆に自乗長さ列が与えられても、それに対応する基底は複数ありうる。size reduction アルゴリズムで標準化を行うことにより、ただ一つの標準的な基底行列に対応させることができる。なお、格子 $\mathcal{L}$ の任意の基底を固定すると、辞書式順序においてそれよりも小さい ($\mathcal{L}$ のある基底に対応する) 自乗長さ列は、有限個であることに注意する。最小となるような自乗長さの列も唯一、存在する。その自乗長さ列に対応する基底は HKZ 基底と呼ばれる。

2.4 k 番目の直交補空間での最短ベクトル問題と基底簡約

基底行列 $\mathbf{B}$ に対して、格子ベクトル $\mathbf{v} \in \mathcal{L}$ が k 番目の基底ベクトルを基底簡約出来るとは、k-直交補空間において、$\mathbf{v}$ のほうが k 番目の基底ベクトルよりも短い場合 $|\pi_k^B(\mathbf{v})| < |\mathbf{b}_k^*|$ をいうことにする。$\mathbf{v}$ が index k において基底簡約できるかは、k 番目の直交基底ベクトルの長さの自乗 $|\mathbf{b}_k^*|^2$ と、$\mathbf{v}$ を k 番目以降の直交基底ベクトルに射影したベクトルの長さの自乗の和 $|\pi_k^B(\mathbf{v})|^2 = \sum_{m=k}^{N-1} (\frac{\langle \mathbf{v}, \mathbf{b}_m^* \rangle}{|\mathbf{b}_m^*|})^2$ を比べることによって判定出来る。$\mathbf{v} = \sum_{i \in I} u_i \mathbf{b}_i$ のように基底ベクトルの整数結合で書けているとすると、この k-直交補空間でのベクトル $\mathbf{v}$ の自乗長さ $|\pi_k^B(\mathbf{v})|^2$

は、

$$|\pi_k^B(\mathbf{v})|^2 = \sum_{m=k}^{N-1}(\frac{\langle \mathbf{v}, \mathbf{b}_m^* \rangle}{|\mathbf{b}_m^*|})^2 = \sum_{m=k}^{N-1}\left(\sum_{i=0}^{N-1} u_i \frac{\langle \mathbf{b}_i, \mathbf{b}_m^* \rangle}{|\mathbf{b}_m^*|}\right)^2 = \sum_{m=k}^{N-1}\left(\sum_{i=m}^{N-1} u_i \frac{\langle \mathbf{b}_i, \mathbf{b}_m^* \rangle}{|\mathbf{b}_m^*|}\right)^2$$

となる。$i < m$ のときに、$\langle \mathbf{b}_i, \mathbf{b}_m^* \rangle = 0$ に注意する。

k 番目の直交補空間の最短ベクトル問題とは、$\mathcal{L}$ を k-直交補空間に射影したときにできる点全体 $\pi_k^B(\mathcal{L})$ のうち、原点以外でもっとも (あるいは、なるべく) 原点に近い点を求めているということになる。一度に $\mathcal{L}$ での最短ベクトルを見つけるのが次元が高い場合は難しいので、直交補空間に次元を落とした問題を考えることで、部分的に少しずつ基底行列を短くしているととらえることができる。後ろの index の直交基底ベクトルが短くなると、前のほうの index の直交基底ベクトルも更新可能なベクトルがより見つけやすくなることをのちに見る。

プログラムにおける実際の基底簡約は以下のように行われる。格子ベクトル $\mathbf{v} \in \mathcal{L}$ と基底行列 $\mathbf{B}$ に対して、$\mathbf{v}$ により長さを短く更新出来る index k が複数ある場合には、そのうち、一番小さい k を格子ベクトル $\mathbf{v}$ の挿入 index と呼ぶことにする。挿入 index が k のベクトル $\mathbf{v}$(を $\mathbf{Z}^N$ 座標で表したもの) を、基底行列の $k-1$ 行目と k 行目の間に行ベクトルとして挟み込んで、$N \times (N+1)$ の行列を作る。しかし、この行列は正方行列ではなく、従属関係が存在する。この従属性を取り除くために LLL アルゴリズムを利用する。LLL アルゴリズムで用いられる swap と size reduction は、行列の張る格子を変化させない。LLL アルゴリズムについては、[14] などを参照せよ。LLL アルゴリズムの適用すると、N 個の線形独立な格子ベクトルとゼロベクトルが得られる。ゼロベクトルを取り除くことで、正則な $N \times N$ の行列となり、$\mathcal{L}$ の基底行列を得ることができる。以上がここで考える基底簡約の方法である。この基底簡約を繰返し行なうことで、基底を良くして、最短ベクトルを見つけやすくすることを目指す。基底簡約後の基底は、LLL 簡約条件を満たすことに注意する。まとめると、格子ベクトル $\mathbf{v}$ と基底行列 $\mathbf{B}$ に対して、$\mathbf{v}$ が挿入 index k を持つときに、基底行列に適用することで、k 番目の shape の値を更新した新たな基底行列を得ることができる。

LLL 簡約条件を満たす基底行列 $\mathbf{B}$ が与えられたとして、ベクトル $\mathbf{v}$ を基底行列の挿入 index である k 行目に挿入して、LLL アルゴリズム適用後の直交基底ベクトルの自乗長さの列がどうなるかを考える。まず、0 番目から $k-1$ 番目までの直交基底ベクトル $\mathbf{b}_i^*$ は不変である。なぜなら、与えられた基底行列 $\mathbf{B}$ は、0 番目から $k-1$ 番目までは LLL 基底簡約アルゴリズムで安定であり、$\mathbf{v}$ の挿入 index が k であることから $|\pi_{k-1}^B(\mathbf{b}_{k-1})| = |\mathbf{b}_{k-1}^*| < |\pi_{k-1}^B(\mathbf{v})|$ となり、index $(k-1)$ と index k の間での swap が起こらない。よって、このベクトル $\mathbf{v}$ は、0 番目から $k-1$ 番目の直交基底ベクトルを置き換えることができない。ベクトル $\mathbf{v} \in \mathcal{L}$ を挿入 index k に挿入することにより、k 番目の直交基底ベクトルは短くなる。LLL 基底簡約後の index k の基底ベクトルは、挿入されたベクトル $\mathbf{v}$ になるので、index k の直交基底ベクトルの自乗長さは、$\mathbf{v}$ の k-直交補空間での自乗長さ $|\pi_k^B(\mathbf{v})|^2$ に置き換わる。

ある程度、基底が簡約されていると、基底の自乗長さの列、すなわち shape $(|\mathbf{b}_i^*|^2)_{i \in I}$ は後ろの index にいけばいくほど、一般的には、値が小さくなっていく傾向にある。これは直交補空間の次元が下がるためにより短いベクトルが探しやすくなっているからである。shape の時間的な変化を

みることで、基底簡約によって、どの index が更新されたかを見ることができる。

図 1 は基底簡約前後でどのように shape が変わるかの例を示したものである。index 10 において基底簡約した例になっている。index 10 では、簡約前の shape を表した青いグラフから簡約後のオレンジのグラフで値が減っているが、その後ろの index では長くなったり短くなったりしている。このときの次元は 140 次元のものであるが、前の index の shape が重要なので前の 40 次元だけを取り出している。後ろのほうの shape は、考えている格子の低いので LLL アルゴリズムなどで比較的簡単に最短値近くまで短くすることができる。

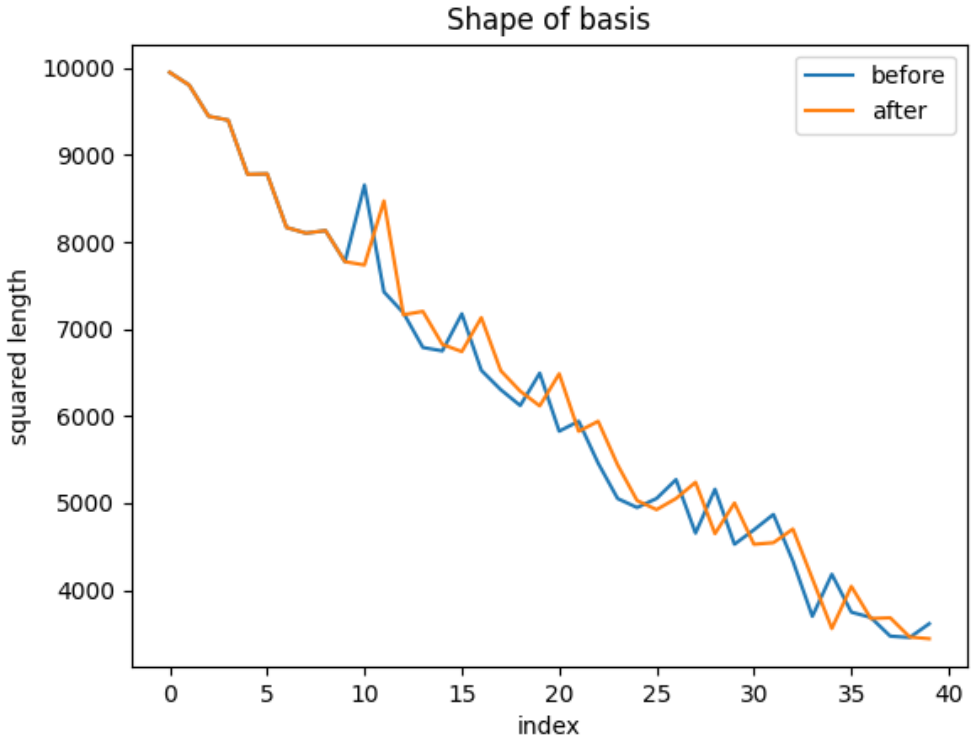

図 1　shapes before and after transformation at index 10

index k での簡約後に $k+1$ 番目以降の直交基底ベクトル $\mathbf{b}_i^*$ は、長くなったり短くなったりだが、もともとある程度簡約されて短かった場合は、長くなることのほうが多い。

各格子ベクトルの全体空間における自乗長さは、各直交成分の自乗長さの列の和 $\sum_{m=k}^{N-1}(\frac{\langle \mathbf{v}, \mathbf{b}_m^* \rangle}{|\mathbf{b}_m^*|})^2$ であることに注意する。すなわち、短い格子ベクトルを得たい場合は、各 index ごとに成分を短くすればよい。その index の直交成分の長さを短くするためには、その成分の直交基底ベクトルの自乗長さ $|\mathbf{b}_m^*|^2$ を短くすることによって、確率分布的に短いベクトルが多くなる。よって、Babai lifting を行う index の集合 I_L に関する直交基底ベクトルの長さの自乗和 $\sum_{i \in I_L} |\mathbf{b}_i^*|^2$ を短くすることが基底簡約の目標となる。上に書いたように、各成分を独立に小さくするのは困難であるので、この研究では後に見るように基底の評価関数を参考に基底簡約を行う。

3 ベクトル生成

3.1 自然数表現と Babai lifting

基底簡約できるような短い格子ベクトルを見つけるためには、基底から新しい格子ベクトルを大量に作る必要がある。基底行列の列ベクトルである基底ベクトルに対する整数係数を決めることで新しい格子ベクトルが整数結合 $\mathbf{v} = \sum_{i \in I} u_i \mathbf{b}_i$ として得られることになる。個々の格子ベクトルをなるべく短く作るときは、後ろの index の係数から設定していく。直交基底ベクトルはお互いに直交しているので、ベクトル $\mathbf{v}$ の index k での直交補空間に射影した自乗長さ

$$|\pi_k^B(\mathbf{v})|^2 = \sum_{m=k}^{N-1} \left(\frac{\langle \mathbf{v}, \mathbf{b}_m^* \rangle}{|\mathbf{b}_m^*|} \right)^2 = \sum_{m=k}^{N-1} \left(\sum_{i=m}^{N-1} u_i \frac{\langle \mathbf{b}_i, \mathbf{b}_m^* \rangle}{|\mathbf{b}_m^*|} \right)^2$$

は、index $(k+1)$ での直交補空間でのベクトルの自乗長さ $\sum_{m=k+1}^{N-1} (\frac{\langle \mathbf{v}, \mathbf{b}_m^* \rangle}{|\mathbf{b}_m^*|})^2$ に、新しい index k での直交成分の自乗長さ

$$\left(\frac{\langle \mathbf{v}, \mathbf{b}_k^* \rangle}{|\mathbf{b}_k^*|} \right)^2 = \left(\sum_{i=k}^{N-1} u_i \frac{\langle \mathbf{b}_i, \mathbf{b}_k^* \rangle}{|\mathbf{b}_k^*|} \right)^2 = \left(u_k + \sum_{i=k+1}^{N-1} u_i \frac{\langle \mathbf{b}_i, \mathbf{b}_k^* \rangle}{|\mathbf{b}_k^*|^2} \right)^2 |\mathbf{b}_k^*|^2$$

を足すことにより、逐次的に計算できる。index k より大きな u_i はすでに与えられているとしたときに整数結合の係数 u_k をこの値が一番小さくなるような整数値に選ぶことでもっとも短くなる。すなわち、$0.5 < \nu_k \leq 0.5$ になるように選ぶことができる。このように直交補空間での長さが短くなるように係数を選んでいくのが Babai lifting である [4]。Babai lifting でベクトルの次元を 1 次元ずつ上げていくことを最後まで繰り返すと、全体空間のベクトルが得られる。短い格子ベクトルの探索するための係数決定アルゴリズムとしてみると、Babai lifting では計算は分岐せずに一本道に進む、各時点でもっとも短くなるように係数を選んでいく greedy な係数決定アルゴリズムと言える。

同様に $-1 < \nu_k \leq 0.5$ または、$0.5 < \nu_k \leq 1$ の範囲に入るような整数 u_k が唯一存在する。そのように u_k を選ぶことで直交補空間での長さが 2 番目に短いベクトルが得られる。それ以降も同様に $(a+1)$ 番目に短いベクトルとして、$-a/2 - 0.5 < \nu_k \leq -a/2$ または、$a/2 < \nu_k \leq a/2 + 0.5$ を満たすような u_k の値が唯一取れる。このときに、この index の係数に a という自然数のタグをつける。すなわち、係数に直交成分が短いほうから $0, 1, \ldots$ と自然数によるタグをつけていくことができる。最後尾の index からこのようなことを行えば、N 次元の自然数 $\mathbb{Z}^N$ と格子点 $\mathcal{L}$ を全単射で対応付けることができる。格子ベクトルの自然数表現と呼ぶことにする。格子において、自然数表現が与えられれば、対応する格子ベクトルを計算することができるし、格子ベクトルが与えられればそれに対応する自然数表現を計算することができる。Babai lifting により生成したベクトルのその成分の自然数表現は 0 ということになる。

3.2 ベクトル生成アルゴリズム

ベクトル生成の主なアルゴリズムとしては、sieving と enumeration がある [16]。sieving は、既知のベクトルの集合を考えて、その中でベクトルの組の和や差を計算することで新しい短い格子ベクトルを探っていく方法である。長いベクトルを新たに得られた短いベクトルに置き換えるなどして、短いベクトルの集合が得る。sieving を行うベクトルの空間が高次元であると、ベクトル集合を徐々に短くするのに必要なベクトルの数が膨大になる [21]。そこで、G6K などでは、線形空間全体で sieving を行うのではなく、index が後ろのほうの直交補空間で sieving を行って、短いベクトルが得られたあとは、Babai lifting で全体空間に拡張する方法が用いられている。適当な次元の直交補空間で sieving を行うことにより、次元を下げて必要なメモリの量を減らしている。

enumeration は、$\mathbf{v} = \sum_{i \in I} u_i \mathbf{b}_i$ を作る時に、後ろの index から考えて、その index の整数係数 u_i を射影ベクトルが短くなるものからいくつか分岐させていくような探索木を考えることで、全体空間におけるベクトルたちを作っていく方法である。後ろの index から計算していく直交補空間での自乗長さ $|\pi_k^B(\mathbf{v})|^2 = \sum_{m=k}^{N-1} (\frac{\langle \mathbf{v}, \mathbf{b}_m^* \rangle}{|\mathbf{b}_m^*|})^2$ は index が進むにつれて (k が小さくなるにつれて) 単調増加になる。係数 u_k の決定において、最短ベクトル問題のその時点の射影長さが目標とするベクトルの長さを超えてしまうものは、その先の計算を考える必要はない。最短ベクトルを必ず求められるような厳密解法を考える場合は、目標値より長くならないような整数係数すべての範囲を考えることになる。近似的な解法を考えている場合は、より効率を上げるために、短くなる可能性が高いものだけに絞って、分岐する整数係数を適当な範囲に制限してもよい。また、高次元においては、後ろのほうの 40 次元ぐらいは enumeration を使って、探索の分岐を行い、前のほうの係数の決定は、分岐せずに Babai lifting で一直線に進んでいく方法がとられる。sieving を利用する以前のわれわれのプログラムではこの方法を用いていた。enumration の枝刈りについては、[11] などでも研究されている。

われわれの現在のプログラムでは、enumeration ではなく、sieving を用いてベクトル生成を行っている。直交補空間で sieving を行い、必要に応じて Babai lifting で全体空間に拡張している。基本的には、enumeration も sieving も直交補空間において、短いベクトルを生成するという意味では変わらないが、高次元においては、enumeration よりも sieving のほうが効率的であることが知られている。sieving におけるベクトル生成では sieving に対応する index の直交基底ベクトルは使っていない。よって、その部分の直交基底ベクトルを基底簡約で短くする必要がない。ベクトル生成アルゴリズムとして enumaration を用いた場合には、すべての index において、直交基底ベクトルの長さを短くする必要があるが、sieving を用いると、Babai lifting 部分の基底の部分のみが基底の簡約の対象となるので、効率的に基底簡約ができる。

Babai lifting を行う index $I_L = \{i \in I | 0 \leq i < L\}$ を Babai index、$I_S = I - I_L$ を sieving index と呼ぶことにする。ここで L は Babai Lifting を行う index の幅を表す。sieving を使った場合の、基底簡約アルゴリズムにおける探索点は、I_L の基底ベクトルの列 $(\mathbf{b}_i | i \in I_L)$ となる。この探索点をよくしていくとこにより、ゴールに近づく。特に直交基底ベクトルの自乗長さの列

($|\mathbf{b}_i^*|^2 | i \in I_L$) が重要になる。sieving を考える直交補空間は、I_L の基底ベクトルで決定される。sieving のアルゴリズムを固定した場合、sieving においてどのような分布のベクトルが得られるかは、I_L の基底ベクトルで決定されることになる。

3.3 ランダム仮定

Babai lifting において、格子ベクトルを $\mathbf{v} = \sum_{i \in I} \nu_i \mathbf{b}_i^*$ と書いた時の $\nu_i (i \in I_L)$ は、-0.5 から 0.5 までの範囲でどこになるかはまったく情報がなく、一様にランダムに分布する。この範囲で一様ランダムで、index 間で確率的に独立に分布すると仮定して確率計算をするのがランダム仮定である。このランダム仮定のもと、Babai lifting 部分の長さの分布を計算することができる。ランダム仮定についてはたとえば、[18] を参照せよ。

ランダム仮定は、格子のベクトルが空間に偏りなく存在していることと、設定された範囲に存在する格子ベクトルは、その矩形の範囲で、得られる確率が同一であるということが前提になっている。基底が与えられて、sieving 部分の射影ベクトルが一つ与えられたとき、その先の Babai lifting がどうなるかを考える。Babai lifting により限定された矩形範囲 $\{\sum_{i \in I_L} \nu_i \mathbf{b}_i^* | - 0.5 < \nu_i \leq 0.5 (i \in I_L)\}$ において格子点が均等に分布していて、それらは等しい確率で得られる。Babai lifting により限定された範囲で等確率ではあるが、実際には、sieving 部分の射影ベクトルが与えられたときに、Babai lifting で対応させた格子ベクトルは一つである。どの程度の密度で分布しているは、格子の行列式から推測できる。

ランダム仮定は連続的なモデルであり、固定された基底に関してベクトル生成した場合を主に考えている。実際には格子ベクトルの空間は離散的であり、格子を固定して基底を変えて何回もベクトル生成しても、連続的な分布が得られるわけではない。ある長さ以下の格子ベクトルは有限個なので、特に短い部分では離散性が際立ってくる。

この論文では、基底簡約アルゴリズムにおいて、ランダム仮定は、ベクトルの重複を除いて成り立っていると仮定して話を進めていく。重複した分だけ効率が落ちるが、それを除ければ確率分布に従って、短い格子ベクトルが得られるということである。この仮定自体が正しいかどうかは別に検証していく必要がある。ゴールベクトルを得る確率を正確に測るのは難しいが、ゴールベクトルよりも少し長いベクトルであれば観察しやすい。ゴールベクトルを得る確率を上げるには、それよりも少し長いが短いベクトルを得る確率が高くすればよいという前提は妥当なものだと思われる。われわれはやや短いベクトルを得る頻度を観察し、アルゴリズムを、やや短いベクトルを得る頻度が高くなるようにパラメータを調整している。

3.4 ベクトルの長さの分布

格子ベクトル $\mathbf{v} = \sum_{m=0}^{N-1} \nu_m \mathbf{b}_m^*$ の自乗長さ $|\mathbf{v}|^2$ は、Babai lifting 部分の直交成分の長さ $\sum_{m=0}^{L-1} \nu_m^2 |\mathbf{b}_m^*|^2$ と、sieving 部分の直交成分の自乗長さ $\sum_{m=L}^{N-1} \nu_m^2 |\mathbf{b}_m^*|^2$ の和になる。短いベクトルを得るにはどちらも短くする必要がある。ここでは、ベクトル生成時の各ベクトルの Babai

lifting 部分の直交成分の自乗長さ $\sum_{m=0}^{L-1}(\frac{\langle \mathbf{v}, \mathbf{b}_m^* \rangle}{|\mathbf{b}_m^*|})^2 = \sum_{m=0}^{L-1} \nu_m^2 |\mathbf{b}_m^*|^2$ の分布を考える。ランダム仮定により ν_i は -0.5 から 0.5 の範囲を一様分布し、成分ごとに独立に振る舞うと仮定することができる。この部分だけを考えると矩形領域となる。次元が高いと正規分布に近づくが、実際には有限の次元なので、確率を正しく計算するには、Gram-Charlier 展開を用いて計算する必要がある [3, 19]。長さの限定で定義できる確率事象 $\{\mathbf{v} \in \mathcal{L} | \ |\mathbf{v}| < l\}$ を考えている場合には矩形領域と球との共通部分を考えることになる。基底簡約にまたがって複数の基底を考える場合は、それらの先頭 index からの共通する基底ベクトルにより考えている矩形範囲が絞られる。基底が固定されている場合は、実際の生成分布に近い値が計算で求めることができるが、本研究では実際の簡約アルゴリズムでの計算中の生成ベクトルの長さに注目しており、基底が変化するので Gram-Charlier 分布との比較は省略した。

短い格子ベクトルが見つかる確率を上げるような方向を意図して基底簡約をするのは難しい。直交基底ベクトルの自乗長さの和 $\sum_{m=0}^{L-1} |\mathbf{b}_m^*|^2$ を指標に用いるとわかりやすい。つまり短い格子ベクトルを得るために、Babai lifting 部分の直交基底ベクトルの自乗長さの和を下げることを基底簡約の最終的な目標に簡約計算を行っていく。前のほうだけではなく、これらの index 全体にわたって短くする必要がある。基底の直交ベクトルの自乗長さの和を下げるということは生成するベクトルの自乗長さの分布の平均を下げるということになる。長さの分布の平均値を下げたいわけではなくて、短い格子ベクトルが得られる確率を上げたいのであるが、あとで見るように実験的にはそれほど変わらないので、平均を下げることを目標にすることにする。shape の形の面から別の言い方をすれば、直交基底ベクトルの長さを前のほうの index だけでなく、後ろの方の index まで全体的に短くすることが目標になる。

Babai lifting を行う部分の直交基底ベクトルの自乗長さの和と得られる短い格子ベクトルの個数に相関がみられることを実験結果として示す。図 2 は、基底の直交基底ベクトルの Babai lifting index 部分の自乗長さの和によって、Gaussian Heuristic で推測される最短自乗長さの 1.6 倍以下の自乗長さを持つような短いベクトルが単位時間当たりにどのくらいの個数が得られるかを表している。横軸が Babai lifting 部分の直交基底ベクトルの長さの自乗の和で、縦軸はその基底からどのくらいの頻度で短いベクトルが得られるかを示している。自乗和が小さいほど、短いベクトルが得られる頻度が高くなっていることを表している。基底簡約の計算を行いながら記録しており、出現した基底は横軸の中央付近のものが多く、左右の端では平均を考える基底の数が少ないので、グラフが上下に揺れている。なお、このグラフではベクトルの重複は考慮せずに短いベクトルの数を数えている。このグラフの実験では、格子の次元は 140 次元で、sieving 部分の index の次元は 80 次元の場合の例を用いている。

全体空間の格子ベクトルの長さの分布ではなく、途中の index の直交基底ベクトルをより短いもので置き換える問題を考える場合も同様に、その index より後ろの直交基底ベクトルの長さの自乗和を短いほど、短い格子ベクトルが得られる確率は上がる。よって基底簡約においては、Babai lifting が行われる index の直交基底ベクトルを短くすることが目標となる。

実際に得られる格子ベクトルの自乗長さは、sieving 部分の自乗長さと Babai lifting 部分の自乗長さを足し合わせたものになる。Babai lifting 部分は矩形領域に均等に分布するにしても、sieving

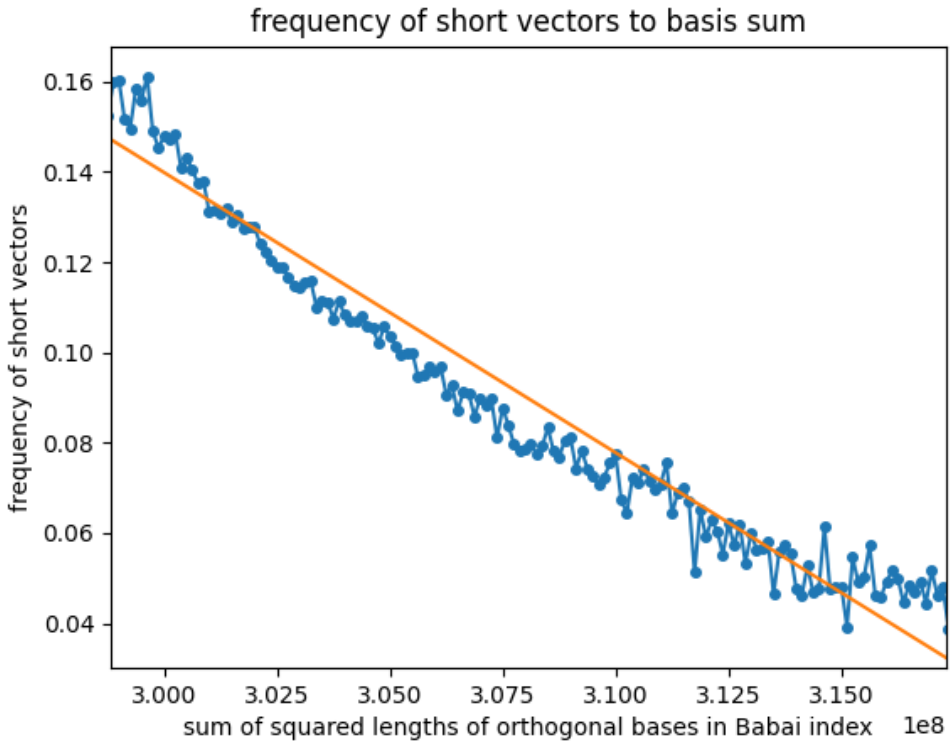

図 2　基底の自乗和ごとの短いベクトルが得られる頻度

部分は空間に均等に分布するわけでもなく、長さによって、分布に偏りがある。ここで考えるモデルでは、おおまかに原点を中心とした球内に均等に分布するとする。このとき、全空間での格子ベクトルの範囲を考えた場合は、sieving 部分と Babai lifting 部分の直交部分空間にわけて考えると、sieving 部分は球状の領域に一様分布、Babai lifting 部分は、矩形上の領域に一様分布して、それぞれが独立に振る舞うと仮定することになる。全体空間における長さの分布は、それぞれの分布の畳み込みで計算することができる。

3.5　ベクトルの重複と基底間の距離

ベクトル生成の連続的なモデルでは確率分布に従って格子ベクトルの長さの分布が得られ、同じ長さのベクトルが再び得られることはないことになる。しかし、実際の格子は離散的であり、コンピュータ上での基底簡約計算では、すでに得られたものと同じ格子ベクトルが得られることがある。これを生成ベクトルの重複と呼ぶことにする。たとえば、基底簡約プログラム実行中に直交補空間においてその時点でもっとも短いベクトルを記録していると、もっとも短いベクトルと同一のベクトルが複数回得られることがよく観察される。短いベクトルの探索において、ベクトルを生成時に同じベクトルが得られると効率が下がることになる。

同じベクトルを生成しやすい 2 つの基底は、距離が近いとみなすことにする。2 つの基底が近い

ほど、ベクトルの生成可能範囲の共通部分が多くなり、ベクトルの重複が増える。2 つの基底間の距離には、共通に持つ基底ベクトルの数が関係してくる。特に、先頭 index から何個の基底ベクトルが同じであるかの数は重要である。共通に持つ基底ベクトルの数は、一方の基底から何回基底簡約を行なって、もう一方の基底が得られたかの回数に関係する。基底簡約を何回も行うことにより、2 つの基底で共通に持つ基底ベクトルが減ってきて、距離が離れていく。

近い基底が生じるケースには、簡約の前後の時間的な関係で生じる場合と、並列計算に関係して基底の複製により近くなる場合がある。まず簡約の前後に関して、近い場合を考える。基底簡約アルゴリズムでは基底を少しずつ簡約していくが簡約前と簡約後で前のほうの index の基底ベクトルは共通になっている。たとえば、index k が挿入 index であるような基底簡約を行ったときには、簡約前と簡約後の基底で、$(\mathbf{b}_i)_{i<k}$ の部分の基底ベクトルが共通である。基底簡約アルゴリズムにおいて基底を簡約していく過程で先頭の基底ベクトルが固定されていくが、固定されるベクトルが増えるほど、このときの基底の前のほうの基底ベクトルが共通になっていく。前のほうの基底ベクトルが固定されているときに Babai lifting アルゴリズムで作られる格子ベクトルは限られている。Babai lifting 部分の index が多いとさらにベクトルが限定されて、見つけたい短い格子ベクトルが、その固定された基底ベクトルのもとで制限された範囲にはひとつもない可能性もある。

並列計算に関係して基底が近くなる場合を考える。複数のプロセスを走らせて、それぞれが基底を持っているとする。プロセス間の協力関係のために、基底簡約の結果をプロセス間で共有すると、それにより、基底が近くなることがある。並列計算においては、プロセス数に比例して、目的の短いベクトルが見つかりやすくなってほしいが、重複があると期待通りの結果が得られれなくなる。

基底簡約 1 回ごとに新たな基底ベクトルが 1 つ挿入されるので、基本的には、基底簡約の回数だけ基底間の距離が離れていくことになる。2 つの基底の前からの index で、k 個の基底が共通だとすると、残りの Babai lifting 部分の次元は、$L-k-1$ となる。この個数だけ基底が異なる基底ベクトルの個数の上限値になる。

連続的なモデルと離散的なモデルの関係について述べる。格子ベクトルが十分ある状況では連続的なモデルがよく当てはまる。しかし、とても短いベクトルが (前のほうの基底ベクトルを固定するなどして) 現在探索している範囲に存在するかどうかという状況では、連続的なモデルよりも離散的に考えるほうがよい。ランダム仮定と同様に格子点が空間内に一様に分布しているという仮定の元に、球と矩形の共通部分の体積を考えることで探索範囲にベクトルが何個存在するかを議論することができる。

なぜベクトルの重複が起こるかをモデル化して考える。index の全体 I が前半の Babai lifting 部分 I_L と、sieving 部分 I_S に分かれているとする。$I = I_L \cup I_S$。基底が与えられると、全体空間が、Babai lifting 部分と sieving 部分の 2 つの直交空間に分解されることになる。単純化して考えるために、sieving により、sieving 空間中の与えられた長さ l のベクトルが全部求まるとする。すると得られるベクトルは、sieving 部分の球内に存在するベクトルをそれぞれ Babai lifting したものになる。最初に与えられた基底 B と少し異なる基底 B' に変わったとすると、最初に与えられた基底で生成される格子ベクトル $\mathbf{v} \in \mathcal{L}$ は、新しい基底での自然数表現を考えると、Babai lifting

部分ですべてゼロになるとは限らない。ただし、とても短いベクトルではすべてゼロになる可能性が高い。ベクトルの自乗長さは、Babai lifting 部分と sieving 部分の自乗長さ $\pi_L^B(\mathbf{v})$ の和なので、新しい基底で考えてもそれほど極端には増減しないことになる。新しい基底の sieving 空間においても、長さが l 以下であれば、新しい基底の sieving においても、$\mathbf{v}$ が探索範囲に含まれることになり、再び見つかってしまい、重複が起こる。

基底が与えられたときに、seiving 部分で長さ l 以下で、Babai lifting して得られるベクトルを生成可能なベクトルと考える。すると、2 つの基底が似ているとは、生成可能なベクトルがどれだけ重なるかと定義することができる。格子ベクトル $\mathbf{v} \in \mathcal{L}$ が与えられたときに、基底が与えられると、Babai lifting 部分と sieving 部分に直交分解することができる。Babai lifting 部分において、$\mathbf{v}$ の自然数表現がすべてゼロで、sieving 部分で長さが設定された長さ l 以下の場合に生成可能なベクトルの範囲に入っていることになる。2 つの基底、B_1 と B_2 により、それぞれ、ともに Babai lifting index での自然数表現がすべてゼロの場合を考える。このときに、sieving 部分の長さが B_1 と B_2 でどのくらい異なるかを調べることで、2 つの基底の類似度を測ることができる。

Babai lifting のみで考えると、ランダム仮定により、生成可能な空間に一様に格子ベクトルが生じるように思ってしまうが、基底簡約しながら格子ベクトルを生成していくと、見つかりやすいベクトルと見つかりやすいベクトルがあることがわかる。見つかりやすいベクトルは何回も重複して見つかる。これは、sieving の空間で球形の範囲で一様に分布したとしても、基底簡約により、球形の範囲に対応する格子ベクトルが変化してしまうからである。Babai lifting 部分の直交補空間と、sieving 部分の直交補空間に全体空間を分解したときに、sieving 部分の直交補空間が短いベクトルほど、重複しやすいことになる。基底簡約しても、sieving 部分の直交補空間が短いベクトルは、またその直交補空間での長さが短くなって再び見つかりやすい。

全体空間における格子ベクトルだけでなく、直交補空間における格子ベクトルにおいても重複が起こる。次元が低い分、全体空間よりも直交補空間のほうが重複が起こりやすくなる。また、その次元の最短ベクトルを見つけるのに必要以上の sieving 計算を行うと重複が起こりやすくなる。

3.6 基底間の距離の実験

基底簡約によって、どのくらい基底が異なってくるのか、重複するベクトルを生成しなくなるのかを実験により示す。

Babai lifting 部分の index の幅を L とする。基底が与えられたときに、全体空間 $\mathbb{R}^N$ を Babai lifting 部分 ($\mathbf{b}_0$ から $\mathbf{b}_{L-1}$ で張られる線形空間) と sieving 部分 $\pi_L^B(\mathbb{R}^N)$ に直交分解することができる。よって、基底とベクトルが与えられると、ベクトルの sieving 部分の自乗長さを計算することができる。140 次元の格子に対して、基底簡約プログラムを動かし、最短推測の自乗長さの 1.3 倍以内の長さのベクトルの座標はすべて保存した。ここでは 600 本の格子ベクトルを作った。この保存した短い格子ベクトルの集合を使って、2 つの基底間の類似を調べる。それらの保存している短いベクトル $\mathbf{v}$ に対して、基底が与えられたときに、sieving 部分の自乗長さ $|\pi_L^B(\mathbf{v})|^2$ を計算できる。この自乗長さは基底に依存することに注意する。Babai lifting 部分の係数の絶対値がすべ

て 0.5 以下で、sieving 部分の自乗長さが十分短いときに、sieving と Babai lifting を行うことでそのベクトルが見つかる範囲に入っていることになる。2 つの基底が与えられたとき、どちらの基底の sieving でも見つかる場合は、sieving 部分の長さが共に短いことになる。よって、2 つの基底に対して、sieving 部分の自乗長さの差の絶対値を見ることで 2 つの基底の距離を評価することができる。たくさんの短いベクトルに対して、sieving 部分の長さの変化を調べて、自乗長さの差の合計をグラフにしてみた (図 3)。without small sieving のラベルのグラフは、一つの基底から基底簡約して別の基底を作り、また基底簡約してというようにとなりあう基底同士は 1 回の基底簡約で作られるものである。最初の基底から基底簡約するたびに sieving index 上の空間における射影長さが一番最初の基底のものとどのくらい離れていくかをグラフにしたものである。値が大きいほど、基底が離れているということになる。ここでは、140 次元の格子に対して、格子が 140 次元で、Babai lifting 部分が 60 次元で、sieving 部分が 80 次元の基底簡約プログラムにより、基底の列を作成した。グラフより、基底簡約の回数が増えるほど、sieving 部分の射影長さが離れていくことがわかる。それに伴いベクトルの重複も減っていく。

with 1 small sieve というグラフは、80 次元の基底簡約の間に 70 次元の基底簡約を 1 回入れて後ろのほうの index に限定して基底簡約をした場合である。80 次元の sieving と 70 次元の sieving は末尾の index を揃えるのではなく、先頭の index を揃えて sieving を行なっている。よって、70 次元の sieving で基底簡約が起こった場合は、80 次元の Babai index の基底ベクトルが簡約されるので、sieving の空間に影響が出ることになる。グラフにおいては、大きな sieving と小さな sieving の組みで 1 回の transformation と数えている。with 2 small sieves というグラフは 80 次元の基底簡約の間に 70 次元の基底簡約を 2 回いれた場合である。この実験において、間にいれる 70 次元の基底簡約の回数を増やした場合、基底簡約の回数が少ないと多少、small sieve を増やすことで距離が増えていることが観察されるが、何回か基底簡約をおこなっていると、あまり基底の距離が離れていくスピードには影響がないことがわかる。ここでの実験は、直接重複ベクトルの数を数えたものではないが、実際の基底簡約では 3 回も基底簡約すると、ほとんどベクトルの重複は観測されなくなっている。

4 Gram-Charlier 展開による確率計算

基底が与えられたときに Babai lifting で作ったベクトルの Babai index 上でのベクトルの自乗長さについて考える。この Babai index 上での射影自乗長さは、ランダム仮定の元、Gram-Charlier 展開 [20] によって分布が計算できる。

探索領域 $P \subseteq \mathbb{R}^N$ は、凸集合で、$P \cap \mathcal{L}$ がアルゴリズムによって生成されるベクトルの集合となるようなものとする。ベクトルは探索空間 P の中を一様ランダムに分布するとする。Gaussian heuristic により、P に含まれる格子ベクトルの数は (P の体積)/ (格子の行列式) となる。ゴールベクトルとは、ゼロベクトルを除く長さが l 以下のベクトルのこととする。われわれは、P の中でどのくらいの割合がゴールベクトルになるのか推測する。そのためには、B_l を原点中心の半径 l の球として、$P \cap B_l$ の体積を計算する必要がある。

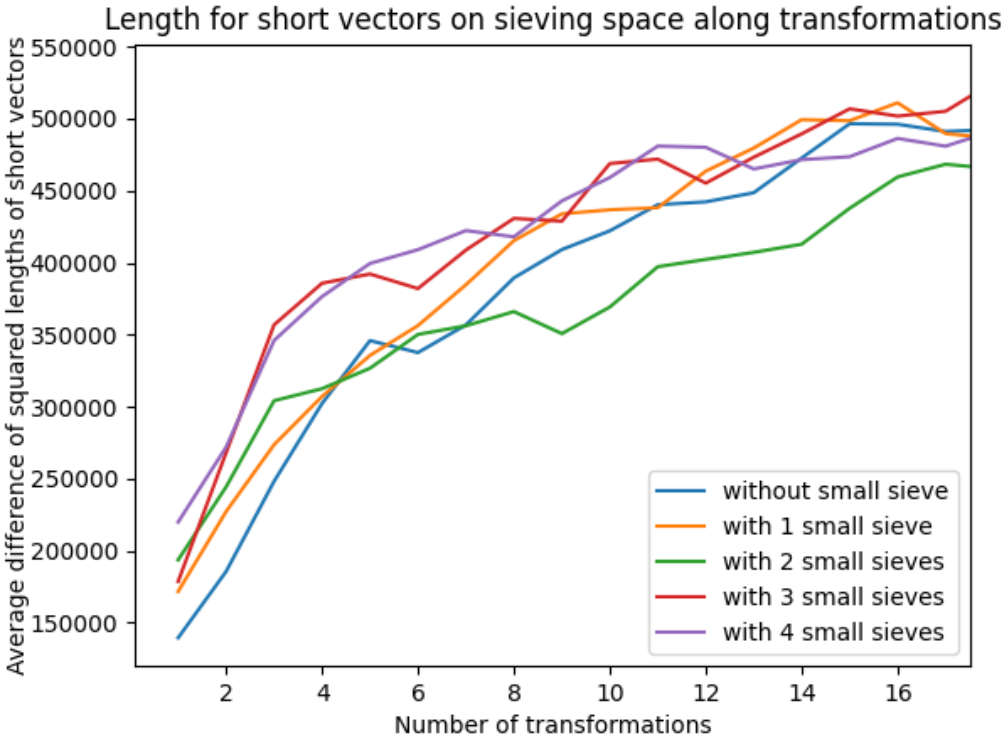

図 3　基底簡約の回数と sieving index における射影長さの差の平均

命題 4.1. 探索空間 $P \subseteq \mathbb{R}^N$ をボレル集合とする。X を P 上に一様ランダムに分布するベクトルの自乗長さを表す確率変数だとする。すると、 $P \cap B_l$ は、$Vol(P \cap B_l) = \Pr(X < l^2)\, Vol(P)$ となる。ただし、B_l は、原点中心の半径 l の球とする。*Gaussian heuristic* のもとで、$|P \cap B_l \cap \mathcal{L}|$ の期待値は、$\Pr(X < l^2)|P \cap \mathcal{L}|$ となる。

Proof. 分布は一様なので、確率は体積に比例する。つまり、$\Pr(X < l^2) = \mathrm{Vol}(P \cap B_l)/\mathrm{Vol}(P)$ となる。Gaussian heuristic により、$|P \cap B_l \cap \mathcal{L}| = \mathrm{Vol}(P \cap B_l)/\det(\mathcal{L}) = \Pr(X < l^2)\mathrm{Vol}(P)/\det(\mathcal{L}) = \Pr(X < l^2)|P \cap \mathcal{L}|$ となる。 □

よって、ゴール球 B_l と探索空間 P の体積がわかれば、探索空間 P においてゴールベクトルが見つかる確率もわかることになる。球の中でもゼロベクトルはゴールベクトルではないので、その部分は誤差になる。探索空間とゴール球に含まれる格子ベクトルの個数の期待値が 1 よりもとても小さい場合には全体の個数との比はゴールベクトルが含まれる確率と解釈することができる。

4.1　探索領域が矩形領域の場合

ランダム仮定のもと、Babai lifting により作ったベクトルの Babai index 成分の射影長さは、矩形領域に一様に分布することになる。それがゴールベクトルになるかどうかは、原点を中心とした

球と矩形領域の共通部分を計算することによって得られる。

まず、 単純化のため、探索領域 P が線分の積で書けるような矩形領域である場合を考える。RSR アルゴリズムにおけるベクトル生成の探索空間などがこのような矩形領域とみなすことができる。線分の方向は、直交基底ベクトル方向である $\mathbf{b}_i^*$ となる。たとえば、Schnorr の RSR アルゴリズムにおけるサンプリングの探索領域は矩形領域となる。矩形領域とゴール球の共通部分に関しては、Fast Inverse Laplace Transform による方法が Aono and Nguyen [3] で扱われている。ここでは、Matsuda, Teruya and Kashiwabara [20] に沿って、Gram-Charlier A series の展開を用いた方法を紹介する。Gram-Charlier 展開により、自乗長さの分布に関する確率密度関数や、累積分布関数 $P(X < l^2)$ などが計算できる。また、球と矩形領域の共通部分の体積も計算できる。ゴールベクトルを得る確率に探索空間にあるベクトルの個数をかけることで、探索によりゴールベクトルの得られる個数の期待値を得ることができる (命題 4.1)。

Gram-Charlier 展開による確率密度関数や累積分布関数 $P(X < l^2)$ の計算は、高次のモーメントを用いて計算する。格子ベクトル $\mathbf{v}$ の自乗長さ $|\mathbf{v}|^2 = \sum_{i \in I} {\nu_i}^2 |\mathbf{b}_i^*|^2$ は、各直交基底ベクトル方向への射影自乗長さ ${\nu_i}^2|\mathbf{b}_i^*|^2$ の和になる。 index $i \in I$ に対する射影自乗長さ ${\nu_i}^2|\mathbf{b}_i^*|^2$ は、ランダム仮定により、独立ランダムに分布することに注意する。

Gram-Charlier 展開による累積分布関数は以下のように計算できる。

$$F(x) = \int_{\infty}^{x} \frac{e^{-\frac{u^2}{2}}}{\sqrt{2\pi}} du - \left(\sum_{r=3}^{\infty} c_r H_{r-1}(x) \right) \frac{e^{-\frac{x^2}{2}}}{\sqrt{2\pi}}. \tag{1}$$

よって、値の計算のために係数 c_r を計算する必要がある。Gram-Charlier 展開 (1) は正規化されているので、ゴールの基準値 l を代入するときには、$(X - \kappa_1)/\sqrt{\kappa_2}$ に従って正規化する必要がある。$H_r(x)$ は、r 次の確率論の Hermite 関数である。(1) は無限級数になるので、 実際には有限項のみを計算した近似値になる。(1) の級数は、密度関数の台が有界の場合には、収束することが知られている。

積分の区間を矩形探索区間に設定することで、index i に対する r 次のモーメント $\mu_{r,i}$ を計算することができる。

$$\mu_{r,i} = \frac{|\mathbf{b}_i^*|^{2r}}{\beta_i - \alpha_i} \int_{\alpha_i}^{\beta_i} \zeta_i^{2r} d\zeta = |\mathbf{b}_i^*|^{2r} \frac{{\beta_i}^{2r+1} - {\alpha_i}^{2r+1}}{(2r+1)(\beta_i - \alpha_i)}. \tag{2}$$

そして、index i の r 次のキュムラント $\kappa_{r,i}$ も関係 (3) を使って計算できる。

$$\kappa_n = \mu_n - \sum_{m=1}^{n-1} \binom{n-1}{m-1} \kappa_m \mu_{n-m}. \tag{3}$$

矩形領域の辺方向は、直交基底ベクトル $\mathbf{b}_i^*$ で規定されるので、index i の自然数表現の動く範囲が $0, 1, \ldots, a$ のときは、index i の積分区間は $(-\frac{1}{2}(a+1), \alpha_i, \frac{1}{2}(a+1)]$ となる。

矩形の探索空間においては、一様ランダムにベクトルが分布すると仮定しているので、ベクトルの成分ごとの自乗長さも index ごとに独立に分布する。よって、ベクトル全体の自乗長さの r 次のキュムラントは index ごとの r 次のキュムラントの和になる。

モーメント μ_r とキュムラント κ_r の関係 (3) を再び使うことで、r 次モーメント μ_r が計算できる。

エルミート関数は、以下のようになる。

$$H_n(x) = n! \sum_{m=0}^{\lfloor \frac{n}{2} \rfloor} \frac{(-1)^m}{m!(n-2m)!} \frac{x^{n-2m}}{2^m}. \tag{4}$$

よって、$\int f(x)H_n(x)dx = n!c_n$ と、r 次モーメント $\mu_r = \int f(x)x^r dx$ を使うと Gram-Charlier 展開の係数 c_r は、

$$c_r = \sum_{m=0}^{\lfloor \frac{r}{2} \rfloor} \frac{(-1)^m}{m!(r-2m)!2^m} \int f(x)x^{r-2m}dx = \sum_{m=0}^{\lfloor \frac{r}{2} \rfloor} \frac{(-1)^m}{m!(r-2m)!2^m} \mu_{r-2m}. \tag{5}$$

よって、これにより、累積分布関数 (1) を計算することができる。

命題 4.1 より、ゴールベクトルの長さの基準値を l_g として、探索空間が P のときに、ゴールベクトルの個数 $|B_l \cap P \cap \mathcal{L}|$ の期待値は、$|P \cap \mathcal{L}| \times P(X < l_g^2)$ となる。

4.2 sieving を組み合わせた探索領域での確率分布

基底に対するゴールベクトルが得られる確率の傾向を掴むのには矩形領域を使った期待値計算でも可能であるが、実際のアルゴリズムに近い sieving と Babai lifting を組み合わせた場合の確率の計算を考えてみる。基底を固定して、全体の空間が前半の index $I_L = \{i \in I | i < L\}$ の空間と後半の $(I - I_L)$ の index の空間に直交分解されるとする。後半の index では sieving を使ってベクトルが生成される。計算の単純化のため、長さが u 以下のベクトルが一定割合で均等に一様に得られるとする。そして、I_L 上の矩形領域と I_S 上の探索球の部分の自乗長さも独立に分布するとする。

この場合の探索空間は、$\{(\sum_{i \in I_L} \nu_i \boldsymbol{b}_i^*) + \boldsymbol{x}| - 0.5 \leq \nu_i < 0.5, \boldsymbol{x} \in \pi_L^B(\mathbb{R}^N), |\boldsymbol{x}| < u\}$ となる。この探索空間における累積分布関数 (1) を Gram-Charlier 展開によって計算する。

I_S 上の自乗長さのモーメントを計算することで、矩形領域と同様な方法で、キュムラントを計算することができる。$i \in I_L$ の r 次のキュムラントを $\kappa_{r,i}$ として、I_S の r 次のキュムラントを κ'_r とする。分布の独立性よりキュムラントを足し合わせたものが全体のキュムラントになるので、全体空間のキュムラントは $(\sum_{i \in I_L} \kappa_{r,i}) + \kappa'_r$ となり、関係 (3) と (5) 式より、累積分布関数が計算できる。

命題 4.2. q 次元空間中の半径 u の探索球に一様に分布するベクトルの自乗長さの分布に対する r 次のモーメント μ'_r は、$\frac{q}{q+2r}u^r$ となる。

Proof. 長さが u 以下のベクトルの数は $\frac{\Gamma(q/2+1)}{D\sqrt{\pi}}u^q$ となる。D は、球の体積をベクトルの総個数で割った値とする。$U=u^2$ とおくと、これは、$\frac{\Gamma(q/2+1)}{D\sqrt{\pi}}U^{q/2}.$ となる。$U^r f(U)$ の積分を $U=u^2$ に対して考えることで r 次モーメント μ'_r は $\int_0^u f(U)U^r dU$ となる。u 以下のベクトルの数を U で微分することにより、$\frac{q\Gamma(q/2+1)}{2D\sqrt{\pi}}U^{q/2-1}$ が得られる。よって、確率密度関数は、以下のようになる。

$$\begin{aligned} f(U) &= \frac{q\Gamma(q/2+1)}{2\sqrt{\pi}}U^{q/2-1} \Big/ \left(\int_0^u \frac{q\Gamma(q/2+1)}{2\sqrt{\pi}}W^{q/2-1}dW\right) \\ &= \frac{U^{q/2-1}}{\int_0^u W^{q/2-1}dW} \end{aligned}$$

よって、自乗長さの r 次のモーメント μ'_r は、

$$\mu'_r = \left(\int_0^u U^{q/2-1+r}dU\right) \Big/ \left(\int_0^u U^{q/2-1}dU\right) = u^r \frac{q}{q+2r}$$

となる。 □

今は、$q=N-L$ であり、sieving がある長さ以下のベクトルを一様に生成すると仮定すると、I_S 上の sieving のベクトルの部分のキュムラントは、この r 次モーメントから計算することができる。

5 本研究が提案するアルゴリズム

5.1 簡約アルゴリズムの全体の流れ

われわれの開発したプロセス並列計算による基底簡約プログラムの概要を述べる。われわれの基底簡約プログラムの各並列プロセスは、基底をひとつだけ持つ。プロセス並列の基底簡約プログラムを動かす際は、MPI を使って複数プロセスが同時に起動される。プロセス間の情報の共有に MPI 通信は用いずに、MPI の仕組みは、プログラムの起動時のみに用いている。それぞれのプロセスは引数で指定されるなどした同一の基底から計算がスタートするが、ベクトル生成計算には乱数も利用されるので、時間がたつにつれ、プロセスごとに徐々に基底が異なってくる。 各プロセスは同一のアルゴリズムで計算を行い、特別な役割のプロセスは存在しない。並列プロセス間ではメモリを共有していなので、並列プロセス間の情報交換は、分散ファイルシステムに保存した基底ファイルを通して行う。並列環境での協調計算の詳細に関しては、後の節で扱うことにして、まずは各プロセスの動作を記述する。

われわれの取り組んでいる SVP Challenge の問題では、Gaussian Heuristic による最短推測長さの 1.05 倍がゴールベクトルの長さの基準となる。基底の簡約を繰り返すことにより基底をよりよいものに変換していくことにより、 なるべく短い時間でゴール条件を満たす格子ベクトルを見つけるプログラムを作りたい。全体空間において、目標となる長さ以下の格子ベクトルが見つかったときには、成功として基底簡約アルゴリズムは終了となる。目標とする長さ以下のベクトルが見つかる前に、現在の基底からベクトル生成をしても一つも基底の更新条件を満たすようなベクトルが見つからなければ、失敗として終了ということになる。

プログラム実行中では簡約順序で小さくなる方向にのみ基底変換を行うことにする。このような基底変換をこの論文では基底簡約と呼んでいる。格子の空間は離散的であり、各直交補空間でのベクトルはいくらでも短くなるわけではなく、最短となるものが存在する。よって、長い期間、簡約プログラムを動かし続けていれば、いつかは簡約条件を満たすような格子ベクトルが見つからなくなる基底に到達する。どの index の直交補空間においても設定された割合以上短くするような直交基底ベクトルが見つからなくなったら、設定された更新条件を自動的に緩和して、適用を保留されていたベクトルを使って基底簡約を行う仕組みを実装している。このように、現実的な時間では更新ベクトルが見つからずにプログラムが終了するということがないようにプログラムが組まれている。また、そのプロセス単体で更新ベクトルが見つからなくても、BEST BASIS を使った協調計算の仕組みで別の並列プロセスの基底に置き換えられて、計算が継続することもある。

仮にゴールとなる格子ベクトルが見つからないなどの理由でプログラムが終了した場合も、ファイルとして保存されている少し前の基底などを入力としてプロセスを手動で再開するような運用も可能である。プログラムの実行中は、簡約順序が小さくなる方向にしか基底が変わっていかないように作っているが、簡約順序が戻った基底とともにプロセスを再起動することで、計算を継続してゴールベクトルを探すことも時には有効である。基底を過去のものに戻したとしても、過去と同じ計算が行われるわけではなく、すこしの時間で過去とまったく違う計算の状態になると考えられる。

簡約アルゴリズム全体としては、sieving によるベクトル生成と、そこで見つけたベクトルを使っての基底簡約を繰り返すことになる。BKZ アルゴリズム [22] や RSR アルゴリズム [23] などサンプリングを利用した伝統的な基底簡約アルゴリズムは、そのときどきで簡約のターゲットとなる (直交基底ベクトルの長さを短くしたい)index を定めて、そのターゲットとなる index を移動させていくという方式が多かった。われわれは簡約のターゲットになる index を限定せずに、前のほうの直交補空間と後ろのほうの直交補空間が簡約できるかを同時に探索し、ひとつの基底に対し、ある程度の数の格子ベクトルを生成し、簡約可能なベクトルの候補をたくさん生成したあと、そのうち、もっともよいと判断したベクトルと index を使って基底簡約を行う。ベクトル生成には、G6K の pump up という手法を使っている。見つかった基底簡約に使える格子ベクトルは、各 index(に対応する直交補空間) ごとに一番短いものひとつにかぎりメモリに保持している。ベクトル生成時に best vector として変数に記録して、それを基底簡約のときに利用する。

[全体アルゴリズムの流れ]

入力：基底

主要パラメータ：Babai index の幅 L

---

1. 初期設定
2. ループ (終了条件を満たすまで繰り返す):
3. 　　ベクトル生成
4. 　　基底簡約処理

5. 並列協調処理

今回のアルゴリズムは、各 index の直交補空間ごとにもっとも短いベクトルを保持し、各ベクトル生成 (pump up のセット) ごとに 1 回だけ基底簡約を行っている。

プログラムでは、基本的なアルゴリズムに加えて、効率を上げるためのギミックをいろいろと取り入れているが、全体のアルゴリズムに関係するものを中心に 3 つ紹介する。

1 つ目は index ごとに対応する直交補空間で一番短いベクトル best vector だけでなく、2 番目に短いベクトル second vector も保持するという工夫である。index i の best vector を使って挿入 index として直交基底ベクトルが更新されたときに、新しい index $(i+1)$ の直交補空間での best vector として、index i の 2 番目に短いベクトルが使えないかをチェックする仕組みを実装している。index i の直交補空間で短いベクトルは、index $(i+1)$ の直交補空間でも短いからである。

2 つ目のギミックは、短い格子ベクトルを保存しておいて、前のほうの基底ベクトルを一気に短くする仕組みである。プログラムでは全体空間で特別に短いベクトルは、ファイルに保存しており、それらのベクトルを使って、sieving を挟まずに連続適用することで、一気に前のほうの直交基底ベクトルを短くするような仕組みを実装しており、手動で動かすことができる。更新割合条件を緩めて再起動した際に利用できる。前のほうの直交基底ベクトルは、ひとつずつ短くしていると一時的に評価関数が悪くなったりして時間がかかるので、このような仕組みを用意している。ある index で短いベクトルは、その少しあとの index の直交補空間に射影してもまずまず短いベクトルとして使えるということを利用している。今回のプログラムでは、単純な仕組みにするために、基本的に 1 回の pump up につき 1 回の基底簡約しか行っていない。このギミックでは全体空間で短いベクトルのみで複数ベクトルによる基底簡約を利用しているが、一般化すると各 index ごとの直交補空間で複数の短いベクトルを保持して、各 sieving ごとに複数回の基底簡約を行うことも考えられる。

3 つ目のギミックは、高い次元の pump up と低い次元の pump up を組み合わせて交互に行うというものである。高い次元の pump up を行うことで、前の index の短いベクトルを見つけやすくなるが、後ろの index の直交基底ベクトルを短くするためには、何回も基底簡約が必要なので、低い次元の pump up のほうが効率的である。また、高い次元の pump up と、低い次元で pump up を交互に行うことで、高い次元の pump up に用いる基底の類似を減らすことでベクトルの重複を減らすことができる。G6K では pump up に加えて、次元を下げながら sieving を行う pump down という処理を行っている。pump down は、この提案プログラムでは利用していないが、高い次元と低い次元の pump up を交互に行うことで同様の効果を発揮している。

このあとは、簡単のために、ギミックを取り入れていない基本的なアルゴリズムに沿って説明していくことにする。

簡約状況に合わせて、sieving を行う次元を細かく調整することも有効である。低い次元の sieving は時間がかからず、高い次元の sieving は、時間はかかるが短いベクトルが見つかる。基底を簡約して良くするには、直交補空間で短いベクトルを見つければよいので、低い次元の sieving

でよい。多くの基底ベクトルを簡約するためにも短い時間で sieving を行う必要がある。全体空間での短いベクトルを見つけるためには、高次元の sieving のほうが効率的である。よって、基底簡約の序盤は、低い次元で sieving を行い、細かく基底簡約を行うとよい。基底が良くなってきた基底簡約の終盤は、高い次元で sieving を行うとよい。現在のプログラムでは sieving の次元の調整は完全には自動化されておらず、手動で再起動することで調整している。

5.2 提案プログラムにおけるベクトル生成

提案プログラムでは、G6K の pump up というアルゴリズムを利用してベクトルを生成している。sieving を考える部分空間の index の範囲を、左端の index を小さくしていく方向に徐々に広げて、sieving を行う次元を上げていきながら、基底簡約せずに同一の基底を使って sieving を何回か繰り返すものである。なお、sieving を行う index の右端は、index の最後尾に固定している。pump up の最初で乱数を利用して初期ベクトルを作りベクトルのデータベースに入れる。これが 1 回目の sieving の入力ベクトルとなる。1 回目の sieving で得られたベクトルデータベースを Babai lifting することで、すこし高い次元の 2 回目の sieving の入力ベクトルとして活用する。このように sieving したベクトルを次の sieving の入力ベクトルとするために Babai lifting で次元を上げている。G6K においては、pump up と pump down を交互に繰り返すがここでは pump up だけを利用している。pump up とその入力ベクトルを作る処理に関しては、G6K のプログラムをほぼそのまま利用している。

基底簡約プログラムでは、基底を改善するために直交補空間における短いベクトルを見つけるということと、ゴールベクトルを見つけるために全体空間で短いベクトルを見つけるということを同時に行なっている。基底を改善するために最初は比較的、次元の低い sieving でこまめに基底簡約するのが効果的である。基底がよくなったあとに全体空間で短いベクトルを見つけるためには、高次元で時間をかけて sieving を行うことが効果的である。pump up における sieving の最高次元を高くするとメモリも大量に必要になる上、計算時間もかかるが、次元を上げれば上げるほど効率的に全体空間における短いベクトルが得られることが知られている [1]。しかし、われわれの実行環境ではメモリ利用の制限やジョブごとの実行時間の制限などでそれほど sieving の次元が上げられないので、その前提での効率化を目指している。この論文では Babai index の幅を L で表しており、基本的に固定している。$N - L$ が pump up の際の最高次元ということになる。しかし、設定された更新条件では更新するベクトルが見つからなくったときなどに一時的にそのプロセスの pump up の次元を上げるようなギミックもプログラムには実装している。

[pump up の流れ]

パラメータ: pump up を行う際の (スタートの次元、一回に次元を上げる幅、最高次元)。

入力兼出力：best vector

---

1. 最初の sieving の初期ベクトルを乱数を利用して作りベクトルデータベースに入れる。

2. pump up ループ (設定された次元まで sieving の次元をすこしずつ上げながら繰り返す):
3. 設定された次元の直交補空間で sieving を行う。短いベクトルが best vector に記録される。
4. 次の次元の sieving の入力ベクトルとするために、データベースのベクトルを Babai lifting で次元を拡張する。

pump up においては、その間は基底変換が行われないことにより、Babai lifting する部分の index の基底ベクトルがほぼ共通なので、低い次元での sieving で得られたベクトルと高い次元での sieving で得られたベクトルが比較的重複しやすい。

sieving の方式は、G6K の triple sieve をそのまま利用している。sieving の行う際のスレッド並列の仕組みも利用できる。triple sieve は、2 つのベクトルの和や差で比較的短いベクトルが見つかった場合に、そのベクトルの第 3 のベクトルとの和や差を計算している。そして、データベースに保存するほど短いベクトルではなくても、Babai lifting したら全体では短いベクトルが見つかるかもしれないので、データベースに保存する長さの基準と、Babai lifting index で短いかの計算を進めるかどうかの基準が違っている。後者をこの論文では Babai lifting 打ち切り長さの基準と呼んでいる。G6K の論文では sieving をしながら、全体で短い格子ベクトルを探していく手法を on the fly と呼んでいる。

pump up を行うことにより見つけた sieving 空間 $\pi_L^B(\mathbb{R}^N)$ において短いベクトルに対して、Babai lifting の index 上において、1 次元ずつ Babai lifting を行う。各ベクトルは Babai lifting で index を一つづつ前に動かしながらベクトルの係数と長さを計算していく。各 index に対応する直交補空間ごとに設定した基準の長さを超えたらその先の計算は短いベクトルが見つかる見込みが少ないとして、そのベクトルの計算はそこでストップする。その次元の直交補空間で、基底簡約可能なベクトル (直交基底ベクトルよりも短いベクトル) でいままで見つかったなかでもっとも短いものが見つかった場合は、best vector と呼ぶ変数に保存する。計算の打ち切りなしに最後の次元まで Babai lifting が行われた場合にゴールベクトルの基準を満たすようなベクトルかどうかチェックする。best vector の値は、sieving や pump up を超えて保持されていて、基底簡約に利用される。基底簡約後においても、変化がない index に関しては保持される。best vector は、基底変換を超えて保持するために、基底に対する係数の座標ではなく、もともとのユークリッド空間 $\mathbb{Z}^N$ の座標に変換されて記録される。基底簡約のためにベクトルを基底に挿入する際にも整数座標で保存されていた方が都合がよい。よって、best vector に保存されるベクトルは、打ち切りの基準と関係なしに最後の index まで Babai lifting して係数を求め、そこから整数座標に変換する。best vector は、index ごとに整数座標とその index の直交補空間に射影したときの自乗長さの組みとして記録される。

[各 sieving の処理の流れ]

入力兼出力：データベース内のベクトル

入力兼出力：各 Babai index ごとの best vector

パラメータ：sieving に関する各種パラメータ
パラメータ：各 Babai index ごとの打ち切りの長さの上限

---

1. 設定された次元の直交補空間でベクトルデータベースの sieving を行う。
2. sieving 中に基準以下の短いベクトルが見つかった場合は、以下のループを行う:
3. ひとつ前の index に進んで Babai lifting で長さを調べる。
4. 長さがその index の上限に達していたらこのベクトルの処理を終える。
5. その index でいままでより短い格子ベクトルであれば best vector に座標を記録。
6. 最後の index までいったら全体空間での長さでゴールベクトルか判定してこのベクトルの処理を終える。

5.3 基底の評価関数

基底が辞書式順序で小さければ小さいほどよいという考えで基底簡約を行なっているわけでないところがわれわれの簡約アルゴリズムの特徴のひとつである。短い格子ベクトルを見つけやすくするために、たとえば、直交基底ベクトルの自乗長さの和 $\sum_{i\in I_L} |\mathbf{b}_i|^2$ を下げるのが目標の一つになる。しかし、この自乗長さの和を単調に下げる方向に基底簡約を繰り返すことは困難である。前のほうの基底ベクトルを簡約することにより、一時的に自乗和が上がった状態を経由する必要がある。最適な基底簡約のルートを誘導するために基底の評価関数を導入する。基底の評価関数は $\sum_{i\in I_L} \Theta^i |\mathbf{b}_i^*|^2$ のようにパラメータ Θ を使って表される。ここで $0 < \Theta \leq 1$ となるがプログラムでは $\Theta = 0.85$ で計算している。評価関数の値が小さいほど望ましい基底ということになる。直交基底ベクトルの自乗長さに Θ の i 乗を掛けることにより、前の index の直交基底ベクトルの長さを短くすることを重視していることを表している。Θ を 1 に近づけるほど、自乗和を下げることを重視していることになり、0 に近づけるほど、簡約順序で小さくすることを重視していることになる。直交基底ベクトルの長さの自乗の和の値は、今現在、この基底を使って短いベクトルを生成する能力に関係しているが、評価関数の値は、将来的に望ましい直交基底ベクトルにどのくらい近いかということも考慮に入れた指標になっている。基底簡約により、直交基底ベクトルの長さの自乗の和を単調に減らすことは困難なので、評価関数をよくする方向に沿って基底簡約することにより、最終的には直交基底ベクトルの長さの自乗和が小さい基底を得ることが目標になる。

5.4 適用ベクトルの選択

基底簡約のフェーズにおいては、基底簡約の対象となる各 index $i \in I_L$ に対して、i-直交補空間における、もっとも短い格子ベクトルを探索し、現状の直交基底ベクトルの長さ $|\mathbf{b}_i^*|$ よりも短いものが見つかれば、そのベクトルを使って index i で基底簡約を行なう。基底行列と、格子ベクトル $\mathbf{v} \in \mathcal{L}$ が与えられたときに、原理的には、ある index i で $|\mathbf{b}_i^*|^2 > |\pi_i^B(\mathbf{v})|^2$ となるときに基底簡約できる。われわれの実際のアルゴリズムでは、挿入したい index k をひとつ決めて、そこの長さを

更新できるベクトルを探すのではなく、短いであろうベクトルの集合を候補を生成したのちに、そのベクトルによって値を更新できる index を考えている。

基底簡約において、直交基底ベクトルがすこしだけ短くなっても嬉しくない。簡約順序としては小さくなり、前に進むが、基底の自乗和が下がった状態という目標とする地点には逆に遠くなる。われわれの基底簡約アルゴリズムでは更新割合条件を満たした best vector のうち、もっとも評価関数がよいものが基底簡約に利用されることになる。具体的にどのベクトルがそのときの基底簡約に利用されるかは、以下のように決定される。

更新幅 $|\mathbf{b}_i^*|^2 - |\pi_i^B(\mathbf{v})|^2$ が大きいベクトル $\mathbf{v}$ と index i の組みを基底簡約に使いたい。なお、i は必ずしも挿入 index とは限らない。よって、基底簡約するベクトルとして採用された場合は、挿入 index に挿入されるので、基底簡約により、index i よりも前の直交基底ベクトルが短くなることもありうる。各 index ごとにそれまで見つかった中でもっとも更新幅が大きい一つを best vector としてメモリに保持している。基底簡約に使われる格子ベクトルは best vector の中から選ばれる。なお、best vector は基底簡約を超えて保存されるが、基底簡約により直交補空間が変化した時は、その index の best vector はクリアされる。つまり、index i を挿入 index とする基底簡約が行われた場合は、それ以降の index の best vector はクリアされることになる。

index が小さいところの基底ベクトルが更新された場合は、評価関数が上昇しやすい。基底簡約によりその index 以降の直交基底基底ベクトルの長さが微妙に長くなる傾向にあるからであり、前のほうの index での基底簡約の場合は対象範囲が広いからである。よって、前のほうの index の更新の条件は少し厳しくしたい。このプログラムでは、簡約対象の index の直交基底ベクトルの自乗長さが index ごとに設定された割合以上短くならない更新は採用されないようにしている。たとえば、その index で設定された割合が 10% であれば、$|\mathbf{b}_i^*|^2 \times 0.9 > |\pi_i^B(\mathbf{v})|^2$ の場合しか $\mathbf{v}$ が基底簡約に使われるベクトルの候補にならない。この割合は index ごとに決まっていて、先頭の index だと 20% に設定されていて、Babai index の末尾の index だと 0% であり、途中の index の更新割合は線形補間で計算される。この条件をベクトルの更新条件と呼ぶことにする。この更新条件を満たすことが基底簡約に使われるベクトルの必要条件になる。このような制限を設ける理由は、前のほうの index の基底ベクトルをある程度、固定していかないと後ろの index の直交基底ベクトルがなかなか短くなってくれず、Babai lifting の行われる部分の index 全体として、直交基底ベクトルが短くなっていかないからともいえる。前のほうの index の直交基底ベクトルは、限界まで短くせずにちょっと長いままであっても、後ろの index の直交基底ベクトルを短くしさえすれば、目標とする短い格子ベクトルを見つけるのに十分なよい基底が得られるだろうということでもある。前の index の直交基底ベクトルを短くするのは大変だが、後ろの直交基底ベクトルを短くするのは、比較的簡単であり、基底簡約しても評価関数が悪化するというデメリットも少ないので、後ろの index から短くしていこうということである。更新するベクトルがなくなった場合は、適宜、一時的に基準が下げられる。

基本的には、後ろのほうの index の直交基底ベクトルを優先して短くするか、前のほうの index の直交基底ベクトルを優先して短くするかということになるが、どのパラメータが最適であるかは、基底簡約の目標によって異なる。基本的にはとても短いベクトルを得たい場合には前のほうの

基底ベクトルも短くする必要があるので、更新条件の割合を小さくする必要がある。基底簡約の序盤と終盤で変えてもよいのであれば、基底簡約の序盤は、先頭の index の更新割合を 25% ぐらいに大きくして、基底簡約終盤は 10% ぐらいにするなどしてもよい。

更新割合条件を満たす best vector のうち、基底簡約にどの index の best vector が選ばれるかは、ベクトルを点数化して、その点数でもっともよいものを選んでいる。点数化には、2 通りの評価を切り替えて使えるようにした。pair index 法と正範囲 index 法と呼ぶことにする。pair index 法では index i だけでなく、index $(i+1)$ の更新幅 $|\mathbf{b}_{i+1}^*|^2 - |\pi_{i+1}^B(\mathbf{v})|^2$ も考慮にいれて評価している。index $(i+1)$ での更新幅は 1/2 の重みで計算にいれて、(index i の更新幅)+(index $(i+1)$ の更新幅)/2 で、index ごとの best vector に点数をつけている。index $(i+1)$ の直交基底ベクトルの長さの基底簡約後の本当の更新幅は、$|\mathbf{b}_{i+1}^*|^2 - |\pi_{i+1}^B(\mathbf{v})|^2$ になるわけでないことに注意する。

正範囲 index 法は、$|\mathbf{b}_i^*|^2 > |\pi_i^B(\mathbf{v})|^2$ となる隣接する一塊の index の区間を考えて、その部分の更新幅、$|\mathbf{b}_i^*|^2 - |\pi_i^B(\mathbf{v})|^2$ を全部足し合わせて、評価値とするものである。区間は、$|\mathbf{b}_i^*|^2 > |\pi_i^B(\mathbf{v})|^2$ となる index i のうち、連続している範囲を考えるので、場所によって長くなったり短くなったりする。best vector を考えている index から右にどこまで $|\mathbf{b}_i^*|^2 - |\pi_i^B(\mathbf{v})|^2$ が正か、左にどこまで正かを調べて index の範囲を決定する。その範囲の更新幅を足し合わせて評価値を決定する。これにより、既存の直交基底となるべく直交しているベクトルを選ぶことができる。

実験により、2 つの評価方法は、それほど顕著な差は現れなかった。どちらの方法も複数の大きな更新を持つベクトルを優先的に選んでいるので、方針に大きな差はない。今後は正範囲 index 法のほうを基本的に採用していく。ベクトルの評価関数に加えて、同じベクトル更新条件を組み合わせて基底簡約しているので、更新条件に沿った shape ができることになる。なお、更新条件のシステムがなくても、正範囲 index 法の場合は、後ろの index から優先して簡約を行なってくれることが観察された。

このようなベクトルの選択の方法は、基底の評価値がなるべく良くなるようなベクトルを選択したいということからきている。基底の評価関数の計算では前のほうの長さのほうが重みが大きいが、簡約により後ろの直交基底ベクトルがちょっとずつ長くなってしまうので、前のほうの index の基底簡約は更新幅の大きいときしか採用したくない。適用されるベクトルは、簡約後の基底の評価値を計算して選んでいるわけではないので、基底簡約後に必ずしも基底の評価値が改善されるとは限らない。実際に基底簡約してみて評価値が下がる場合だけを採用する方法も考えられるが、今回のプログラムではそこまでは行っていない。index i の基底簡約により index i に $|\pi_i^B(\mathbf{v})|$ の長さのベクトルが挿入されてそのまま index i の基底ベクトルになり、もともと index i の基底ベクトルが index $(i+1)$ に移動し、もともと index $(i+1)$ の基底ベクトルが index $(i+2)$ に移動するというのがもっとも想定されるケースである。そのときの基底簡約後の基底の直交基底ベクトルの自乗長さは簡単に計算できる。たとえば、簡約後の index $(i+1)$ の直交基底ベクトルの自乗長さは、$|\mathbf{b}_i^*|^2$ から、$\mathbf{b}_i$ を $\mathbf{v}^*$ 方向に射影したベクトルの自乗長さ $\frac{\langle \mathbf{b}_i, \mathbf{v}^* \rangle^2}{|\mathbf{v}^*|^2}$ を引いたもの $|\mathbf{b}_i^*|^2 - \frac{\langle \mathbf{b}_i, \mathbf{v}^* \rangle^2}{|\mathbf{v}^*|^2}$ になる。ここで、$\mathbf{v}^*$ は、挿入されるベクトル $\mathbf{v}$ を i-直交補空間に射影したベクトル $\pi_i^B(\mathbf{v})$ であり、簡約後の基底の index i の直交基底ベクトルとして想定されるものである。

LLL の swapping においては、swapping が行われた 2 つの index の直交基底ベクトルの自乗長さの合計は下がることが知られているので、更新幅の大きい index i と挿入 index の間の部分だけを考えると、$|\mathbf{b}_i^*|^2 - |\pi_i^B(\mathbf{v})|^2$ のだけの自乗和の低下は保証されることになる。

[基底簡約のアルゴリズムの流れ]

入力兼出力：基底、各 index ごとの best vector

主要パラメータ：ベクトル更新条件の割合、ベクトル評価方法の選択

---

1. 更新割合条件を満たす best vector のうち、もっとも点数のよい index を選定
2. そのベクトルを使って、基底簡約
3. 挿入 index 以降の index の best vector をクリア

なお、実験に用いたプログラムでは、前のほうの index において、なるべく更新幅の大きな基底簡約を行うためにさらなる工夫を取り入れている。基底簡約のタイミングで毎回すべての index を簡約対象にするのではなく、たとえば、4 回中 3 回は index 45 以降の部分の best vector に制限してそこから選んでいる。3 回に 1 回だけすべての index から選んでいる。

5.5 並列協調計算の仕組み

われわれの基底簡約プログラムにおいては、目標長さ以下の格子ベクトルであるゴールベクトルを見つけることと、ゴールベクトルが見つかりやすいように基底を改善していくということを同時に行なっていく。基底の改善の指標としては、直交基底ベクトルの自乗長さに重みをかけて足しあわせて計算した基底の評価値を用いる。ゴールベクトルを見つけることに関しては、並列数に応じて、試行も増えるので並列計算の効果が出やすい。基底をよくすることに関しては、途中の index の直交基底ベクトルを簡約する必要があるが、考える格子の次元が低くなるので、ベクトルの重なりが起きやすい。よって、複数の並列プロセスで同じ簡約結果が得られることもあり、比較的、並列計算の効果が出にくいと考えることができる。

プロセス間の計算の協調関係には、どのプロセスからも見える分散ファイルシステムにたかだかひとつ基底を記録できる BEST BASIS と呼ぶファイルを通して行う。各プロセスは、BEST BASIS が記録されている場合は、ファイルから読み込んで、メモリ内に保持する基底とファイルに記録されている BEST BASIS を比べる。評価値も BEST BASIS のほうがよく、簡約順序も進んでいる場合は、メモリに保持する基底を BEST BASIS でまるごと置き換える。逆に、評価値が BEST BASIS より、メモリ内の基底のほうがよく、簡約もメモリ内の基底のほうが進んでいる場合は、BEST BASIS をメモリ内の基底で上書きする。メモリ内の基底が置き換えられた際は、best vector については、直交補空間が変わってしまった index に関しては、クリアしている。なお、BEST BASIS がファイル上にない場合は、現在の基底を BEST BASIS としてファイルに書き込むことになる。

このシステムにおいては、基底の簡約が進んでいるが、基底の評価値が悪くなってしまった場

合は、一時的に協調計算から外れてしまうことになる。そのようなものが増えると効率が落ちるので、BEST BASIS の他にもう一つ FRONT BASIS というものを設けている。FRONT BASIS も BEST BASIS と同じくひとつ基底を記録させられることができ、BEST BASIS よりも簡約が進んだ基底に関して、BEST BASIS と同じルールで各プロセスのメモリ内の基底と読み書きを行う。それにより、BEST BASIS の付近での協調関係に加えて、FRONT BASIS の周りでも協調が起こることになる。プログラムにおいては FRONTBASIS と BEST BASIS は一定の距離をとるような仕組みを作っていて、FRONT BASIS が BEST BASIS に近づいて追いついた場合はどうするかなどの詳細の説明は省略する。

[各プロセスの並列協調計算の流れ]

入力と出力：基底、ファイル上の BEST BASIS、FRONT BASIS。

固定パラメータ：基底の評価値の重み Θ

---

1. BEST BASIS をファイルから読み込む。
2. BEST BASIS よりもメモリの基底のほうが評価値がよく、簡約順序も進んでいれば、BEST BASIS に書き出して終了。
3. BEST BASIS のほうがメモリの基底に比べて、評価値がよく、簡約順序も進んでいれば、基底を BEST BASIS に置き換えて終了。
4. メモリの基底のほうが BEST BASIS に比べて、簡約順序も進んでいれば、FRONT BASIS で同じことを行う。

BEST BASIS に加えて FRONT BASIS をもうけた意図としては、あるプロセスよりも基底の評価値と簡約順序の両方で劣っているプロセスを少し減らしたいということでもある。理想的にはそのような優劣が完全についたペアをなくすような仕組みを設けることもできるが、今回はそこまでの仕組みは実装してない。

基底簡約の道のりにおいて、序盤では基底を改善していくことが重要で、終盤ではゴールベクトルを見つけることが重要である。並列プロセス間の協力関係が必要なのは主に序盤から中盤にかけてである。終盤は更新するベクトルがなくならないように計算が続くことが重要である。更新ベクトル情報をプロセス間で共有して、前のほうの index の直交基底ベクトルをもっと積極的に一致させるという方法を以前のわれわれのプログラムでは採用していたこともあった。その場合は、プロセス間の協力関係が密接になり、より簡約の速度が加速するが、終盤でも簡約の勢いが衰えないために更新するベクトルが比較的早く枯渇してしまうということが観察された。簡約が進み過ぎても更新するベクトルが見つからなくなって、計算がストップしてしまう。また、同じ基底で計算するよりはばらばらな基底で計算した方がベクトルの重複も少なくなると思われる。複数のプロセスで基底を置き換えながら協力して簡約することは基底の評価値を下げるフェーズでは非常に役に立つが、すでに十分評価値が下がった状態では、それ以上、評価値を下げることよりは、更新するベクトルを見つけて簡約を続けることが重要になる。現在のプログラムでは、基底による情報のやり取

りを利用した、より緩い協力方法が採用されている。

このように緩く共有する理由としては、基底簡約においてどのルートを辿るのかが有効なのかはっきりわからないので、プロセスごとに違う道筋で基底簡約して、結果を残したほうが残っていくという面も期待している。簡約順序では遅れているが評価値の良いプロセスが前のほうの index の短いベクトルを見つけるのが早いか、簡約が進んでいるが評価値が悪いプロセスがどんどん評価値を下げていって、評価値で追いつくのかどちらが早いか競争する形になる。ずっと計算がばらばらに行われるというわけではなく、簡約が進んでいるプロセスの評価値がだんだん下がっていくなどして、計算が合流することが観察される。また、Babai lifting の行われる index の基底ベクトルをばらばらにすることで、限定されない多様なベクトルを生成する効果を期待している。前のほうの Babai lifting の index の基底ベクトルが固定されていると生成するベクトルの範囲が限定されることが知られている。

図 4　BEST BASIS と FRONT BASIS

6　実験

6.1　実験環境

主な実験環境はパソコンとスーパーコンピュータである。時間は、すべて実時間で計測している。プロセス再スタートからの秒数が基本となっている。

パソコン環境では、M1 プロセッサ搭載の Mac mini や iMac を Intel 互換モードで実行している。Intel 互換モードを使っているのは、G6K のコンパイルの都合である。主に 140 次元の SVP

Challenge の問題を用いた。末尾の index から主に 82 次元の sieving を行い、その前の index は、Babai lifting を行うことにより短いベクトルを探す。並列数は、基本的に 4 つを MPI を利用して立ち上げている。メモリは 8GB 搭載しているが、プログラム実行中の Mac のモニタリングでは Python プロセスは 4 つ全部のプロセスが 82 次元の sieving を行なっているときは、2GB ほどであった。メモリプレッシャーは基本的に上限まではいっておらず、スワップメモリも基本的に利用していないことが、OS のモニタリングより観察された。

iMac を使った計算では、もっとも早い回では、9 日間、プログラムを連続して走らせることで、SVP Challenge 基準のゴールベクトルを見つけることができた。4 つのプロセスを並行して走らせていた。このようにすぐにゴールベクトルが見つかることもあれば、すでに簡約が進んでいる基底からスタートして何週間か走らせても、ゴールベクトルが見つからないこともあった。

スーパーコンピュータ環境では、東京大学によって運用されている Oakbridge CX を用いている。1 つのノードに 2 つ CPU(Intel Xeon Platinum 8280) が載って 8 つのノードを用いて並列計算を行った。ノードあたり 206GB のメモリが搭載されている。ノードを跨いではメモリが共有できないので、4 ノードで同時に計算するために、MPI を使って 4 つのプロセスで並列計算をしている。ジョブの計算時間の上限は 48 時間になっていて、時間が来るとプロセスが強制終了する。終了するたびに保存された基底を使って再起動を繰り返した。時間的な制約のために、大きな次元の sieving が困難になっている。1 つのノードの中ではもともと G6K の sieving 計算に実装されているスレッド並列のアルゴリズムを利用している。

基底簡約がどのくらい有効であるかどうかは、Gaussian Heuristic から推測される最短自乗長さの 1.6 倍のベクトルの自乗長さをログファイルに記録して、時間当たりどのくらいの頻度で見つけることができるかを指標にしている。

スーパーコンピュータを用いて論文のための実験を行いながら、SVP Challenge の 156 次元のゴールベクトルも探していた。sieving の次元は 115 次元などである。プログラムを変更しながらの基底簡約になったが、1 年以上にわたる基底簡約の結果、SVP Challenge の基準である推測値の 1.05 倍以内のベクトルは論文執筆中になんとか出力することができた。ただし、SVP Challenge には 156 次元のもっと短いベクトルがエントリされていたので、登録はできなかった。その後、158 次元の計算を始めて、現在も計算中である。

6.2 基底簡約の計算の道のり

基底簡約アルゴリズムでは、基底を少しずつ簡約しながら基準以下の長さの短い格子ベクトルを探していくことになる。説明の便宜として基底簡約の道のりを序盤と中盤と終盤に分けて考えることにする。プログラムはどの段階でも対応できるような同一アルゴリズムで動くように設計する。

基底簡約の序盤は前のほうの index も後ろのほうの index も直交基底ベクトルの長さが十分に短くなっていない段階である。基底簡約の序盤は基底の直交基底ベクトルの自乗長さの和も大きく、あまり短いベクトルが見つからないが、基底簡約されることで徐々に短いベクトルが見つかる頻度が上がってくる。ベクトルの評価関数に従って、まずは後ろのほうから直交基底ベクトルを短

くしていくことになる。前のほうの index は大きな改善があった場合に簡約されるという感じになる。並列プロセス間で協力しながら評価値もほぼ単調に改善されていく様子が観察される。序盤は、長いベクトルが多いので、sieving を行う次元を低くて、短い時間感覚で、基底簡約していくのが効率的である。

基底簡約の中盤は、大まかには全体的に直交基底ベクトルの長さが短くなってきている段階である。後ろの index の直交基底ベクトルの長さもそれなりに短いので、前のほうの index における基底簡約が起こると一時的に基底の評価値は上がるが、上がった部分は後ろの index であり比較的簡約しやすいのでしばらくするともっと低い評価値に到達する。

基底簡約の終盤は、前のほうの index からそれ以上短くならない長さに到達してくる段階とする。この段階では基底の自乗和を下げるだけではなく、基底簡約できるベクトルを見つけて計算を継続することや、重複ベクトルがあまり生じないようにすることにも気を配る必要がある。基底もそこそこよくなり、それ以上は短いベクトルを見つけられる頻度はそれほど変わらなくなるが、SVP Challenge のようなとても短い格子ベクトルを見つけることが目標の場合には、簡単にはゴールベクトルが見つからないので、ある程度の計算の継続が必要である。sieving を行う次元としては、メモリが許す最大限の次元で計算する必要がある。メモリが大量にあるマシンであれば、sieving の次元を上げることで一回の sieving の時間を増やして効率的に短いベクトルを探すこともできるが、利用できるメモリに制限がある場合は、細かい基底簡約を繰り返すことにより、徐々に基底をよいものにしていくことで、短いベクトルを探すことになる。

6.3 ベクトルの重複に関する実験

生成したベクトルが重複しているかどうかの判定には、そのベクトルの全体空間 $\mathcal{L} \subseteq \mathbb{R}^N$ での自乗長さが等しくかつ第 1 座標の絶対値が一致するかで判定している。異なるベクトルがたまたま自乗長さが等しく、第 1 座標も等しい場合も同一と判定されてしまうが、そのようなケースはごく稀だと考えられる。格子ベクトルの全体空間での自乗長さも第 1 座標も整数値であることに注意する。基底簡約プログラムのベクトル生成部分では、ベクトルの射影長さは浮動小数点数で計算されるが、誤差が出てしまうのでベクトルの一致の判定には使えない。ベクトルの全体の自乗長さは、整数座標値からあらためて再計算している。基底簡約プログラムの実行中に全体空間において、推測される最短自乗長さの 1.6 倍以下の長さの格子ベクトルが見つかった時にその自乗長さと第 1 座標の絶対値を計算ログとしてファイルに出力している。そして、同じ自乗長さで第一座標も同じデータが複数個見つかったら、そのベクトルは重複ベクトルと判断する。最短推測自乗長さの 1.6 倍以内の短いベクトルが重複しているということは、ゴール条件を満たすベクトルなど見つけたい範囲のもっと短いベクトルも重複しているだろうという仮定のもとに研究している。これは推測できない範囲では格子ベクトルは一様ランダムに分布しているとの考えであるランダム仮定に合致している。

どの程度のベクトルが重複するのか理論的に推測するのは困難である。ここではベクトルの生成方法として sieving を用いているので、基底が固定されて、1 回の sieving をおこなっている間は、

重複する格子ベクトルはデータとして登録されないようにすることで、重複は起こらないようにすることもできる。よって、異なる基底間でのベクトル生成時の重複が問題になる。基底が近いほど、重複の度合いが大きいと考えられる。ここでは、実際の基底簡約プログラムの実行ログの解析からベクトルの重複の様子について観察する。また、1 つの pump up 中は基底変換なしに同じ基底で次元を上げながら sieving を何回も行うことになる。このときの sieving 間では、基底が同じことからそれなりの割合で重複ベクトルが生じる。このときの重複は今回は数えないことにする。この論文におけるすべての実験は、すべて異なる pump up 間、すなわち異なる基底間でのベクトルの重複のみをカウントしている。

また、基底簡約なしで何回も同じ基底で pump up を繰り返すということをしてみると、重複率は高くなるが、すべてのベクトルが重複しているわけではない。格子の次元が 140 次元で pump up の次元が 90 次元で 1 日プログラムを走らせてみると、グラフなどは省略するが、8 割程度のベクトルは、過去に得られたベクトルと同じベクトルだったが、それでも 2 割程度は新規のベクトルが得られた。基底簡約プログラムでは pump up をして、基底簡約して、pump up をして、基底簡約してを繰り返すことになるが、基底簡約する分、より重複は減ることになる。

次に基底簡約プログラムを動かして実験してみた結果を書く。図 5 は、ベクトルの生成時間の間隔と重複との関係である。重複していたベクトル同士の生成時間の差が x 軸で、縦軸が重複の頻度 (時間当たりの重複の回数) である。両方とも対数目盛りになっている。一定時間経つと、ベクトルの重複はほとんどなくなる。離れた時間での重複はあまり気にしなくてもよいことがわかる。図 6 は、基底簡約プログラム実行中に、重複ベクトルが生じたペアについて、それぞれを生成した基底間の先頭から基底ベクトルを比べて何個目で初めて異なるかをグラフにしたものである。基底ベクトルの記録は先頭から 40 index までしか行なっていないので、40 個すべてが一致している場合が、index 40 のところに数えられている。これのグラフによっても、離れている基底間でのベクトルの重複がほとんど生じていないことがわかる。

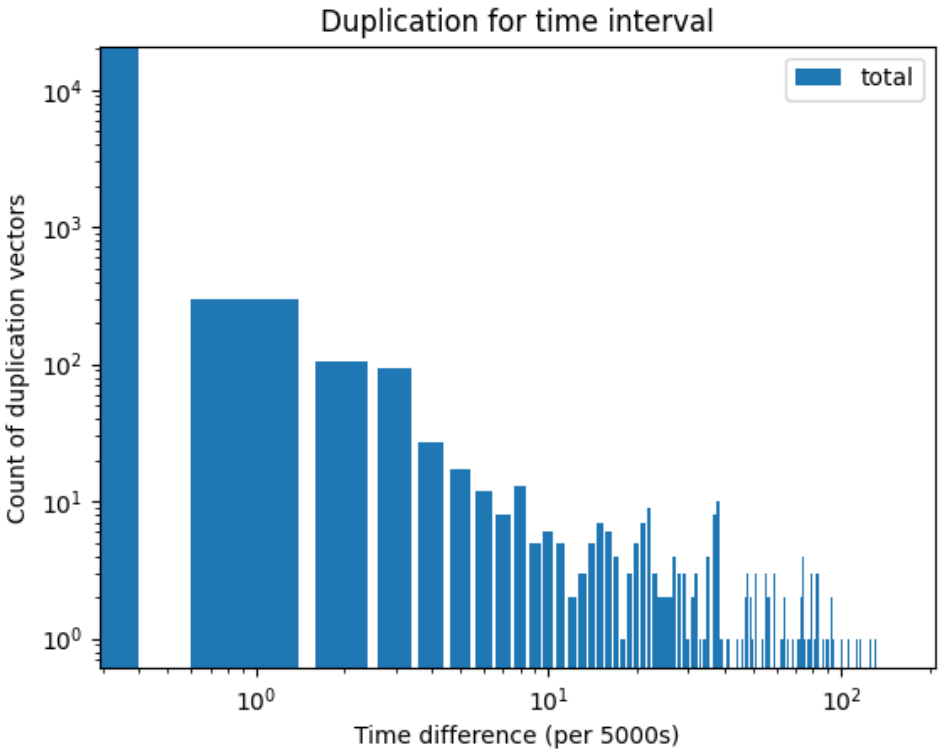

図 5　重複ベクトルの時間間隔

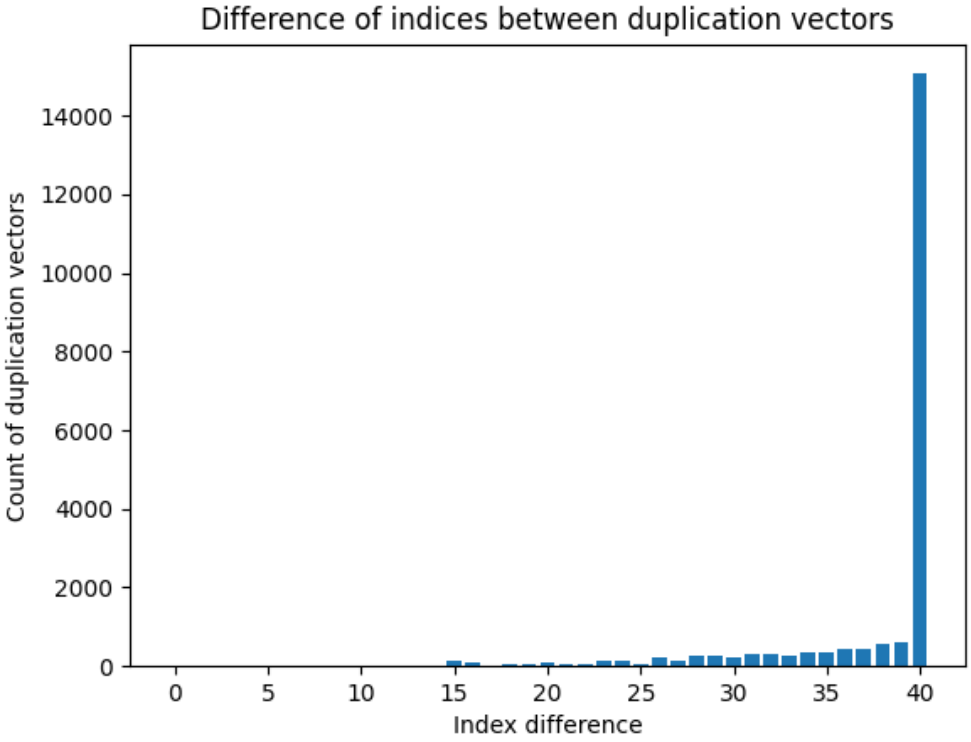

図 6　重複ベクトルの元になった基底間の初めて異なる基底ベクトルの index の場所

図 7 は、ベクトルの長さごとの重複率である。横軸が記録された短い格子ベクトルの自乗長さであり、縦軸が、ベクトルの自乗長さが短いほうから 2000 個ごとにまとめて、そのうち重複しているベクトルの数を数えたものである。重複しているベクトルはまとめて 1 つと数えている。短いベクトルのほうが体積あたりの密度がすこし多い分、重複も多くなっているが、それほどは変わりがない。

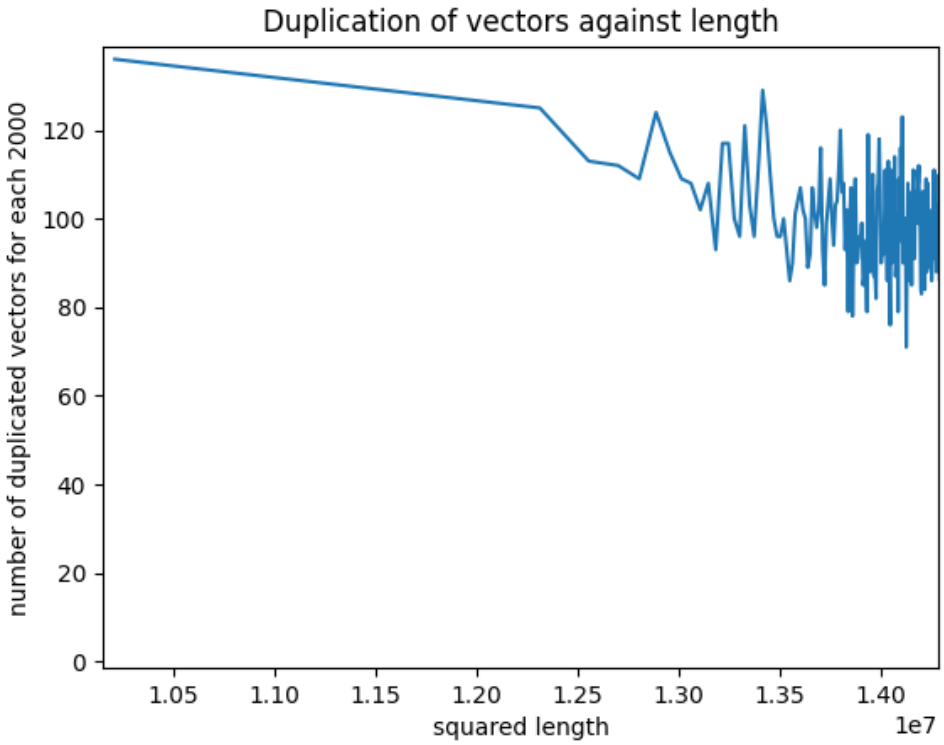

図 7　ベクトルの長さごとの重複率

図 8 は、ベクトルの長さごとに、得られたベクトルの密度の程度をグラフにしたものである。中心から等体積ごとに区切って基底簡約プログラムで得られた格子ベクトルの個数を数えている。このグラフより短いベクトルのほうが得られる個数が多いことがわかる。

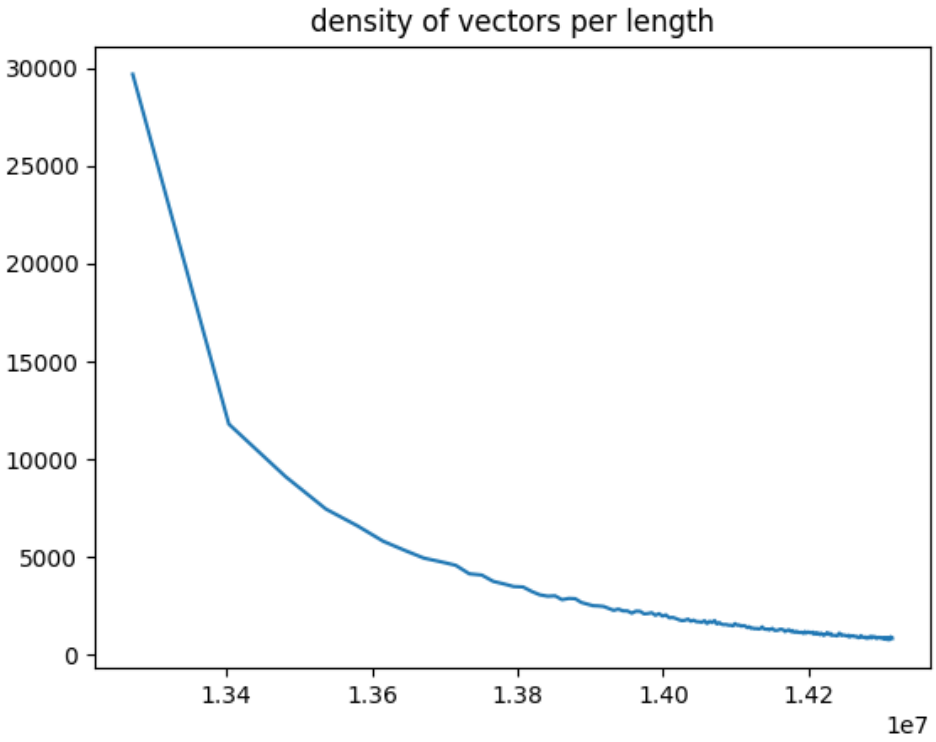

図 8　ベクトルの長さごとの密度

ベクトルの重複についてまとめる。事前に思ったほどは、生成ベクトルの重複が多くなかった。比較的後ろのほうの index の基底ベクトルしか変化しなくなる基底簡約の終盤ではもっとベクトルが重複するかと思われたが、実験では、意外と重複は増えなかった。少し、基底が異なるだけで、ベクトルの重複は極端に減ることがわかった。

6.4　並列計算に関する実験

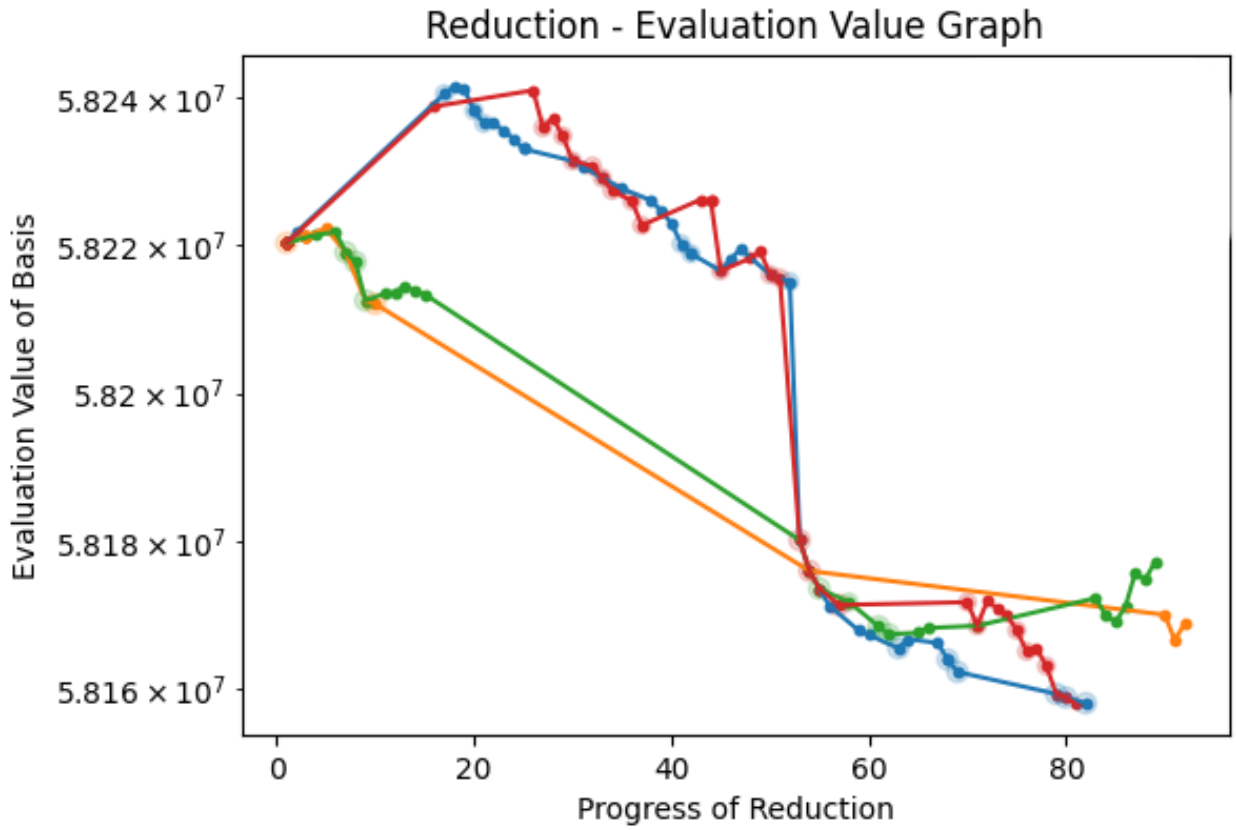

図 9　プロセス間で協力して評価値を下げていく様子

図 9 のグラフは、140 次元で 4 つのプロセスでの実験の簡約終盤のある 1 日の計算をグラフにしたものである。4 つのグラフが 4 つのプロセスを表している。各色に割り当てられた番号は、ランダムに割り当てられたプロセスの番号である。番号は識別のためであり簡約には利用していない。各点が一組の pump up のセットに対応している。横軸は、そのときの出力された基底の shape を簡約順序でソートしてプロットしたものである。縦軸は基底の評価値になっていて、値が小さいほどよい基底ということになる。評価関数 $\sum_i \Theta^i |\mathbf{b}_i^*|^2$ のパラメータ Θ は、0.85 で計算している。このパラメータの値が 1 に近いとあまり協力が行われず、0 に近いと簡約重視でプロセス間の協調が進む方向となる。各プロセスの基底は簡約が進む方向にしか変化しないので、それぞれのグラフは右側にしか伸びないが、横軸は時間と一致しているわけではないことに注意する。よってグラフの傾きの比較などは意味がない。グラフの序盤の黄緑とオレンジの重なりは BEST BASIS を介した協力関係、赤と青グラフの重なりは、FRONT BASIS を利用した協調関係になっている。このグラフの中盤ですべてのグラフの点が一致したところは、赤のプロセスが、前のほうの index を使った大きな更新を行なって、BEST BASIS よりも評価値のよい基底になったために、他のプロセス

もそれに置き換えられたことを表している。各基底簡約では、直接的には、基底の評価値を更新ベクトルの選択に利用していないので、必ずしも基底の評価値の下がる方向のみに動いているわけではない。BEST BASIS の基底の置き換えの処理で評価値を利用している。全体のグラフをみると、評価値が下がっていく方向に動いていって、それに伴い、基底の直交基底ベクトルの長さの自乗和も下がってきている。それにより、短い格子ベクトルが見つかりやすくなっている。前のほうの index の基底が簡約されることにより、評価関数の値はときには大きく上がって、時には大きく下がる。値が上がった場合も、後ろの index の直交基底ベクトルの長さが少し長くなって、簡約しやすくなっていることにより、計算を継続していけばいずれは下がることになる。

Θ の値が幾つが妥当なのかということは、状況によって多少異なる。この論文ではその点については詳しく扱ってないが、効率的な基底簡約の本質に関わる部分である。基底を簡約していくときに、後ろの index の直交基底ベクトルの長さから小さくしていくのか、前のほうを優先して簡約して行ったほうがよいのかということになる。目標となる長さがどのくらい短いものであるかということにも依存する。

6.5 基底簡約終盤での重複の実験

基底簡約の序盤の計算など、評価値が下がっていく局面では、BEST BASIS の協力関係は非常にうまく機能して、各プロセスで協力して評価値を下げていくことが観察できる。評価値がなかなか下がらなくなっていく局面では、各プロセスは、それほど協調せずに計算が行われ、プロセス間の生成ベクトルの重複が避けられ、計算も長く継続できる。すでに前のほうの index に最短値に近い基底ベクトルが見つかっているような最も終盤の計算では、前のほうの index の基底ベクトルがほとんど簡約されず、実質的に低次元の基底簡約の計算の実験をしていることとそれほど変わらないことになる。前のほうの基底ベクトルが簡約されないことにより、それぞれの基底が近くなって、重複ベクトルが生じる可能性が増えることになる。並列計算のときに、終盤の計算をどのように行ったらよいかというのは基底簡約を使った最短ベクトル探索計算にとって重要なので、計算例とともに考察する。

パソコン上での 140 次元の格子に対する終盤の基底簡約の基底簡約の様子を例にとって説明する。SVP Challenge の問題基底からスタートして 3 ヶ月ほど動かした計算したあとの終盤における 1 週間程度の様子である。すでに先頭の 15 個ぐらいの index の基底ベクトルはすでに十分に短くなっていて、基底簡約できるようなベクトルが見つかりにくい状態になっている。比較的前の index において基底簡約が起こったときに、評価関数の値が一時的に上がるが、しばらく計算するとまた落ちていくことが観察される。あるときは、BEST BASIS や FRONT BASIS で協力関係しながら値が下がっていったり、あるときはそれぞればらばらにグラフが進んでいくが、これは BEST BASIS の置き換えのルールに従って、自律的に計算した結果である。

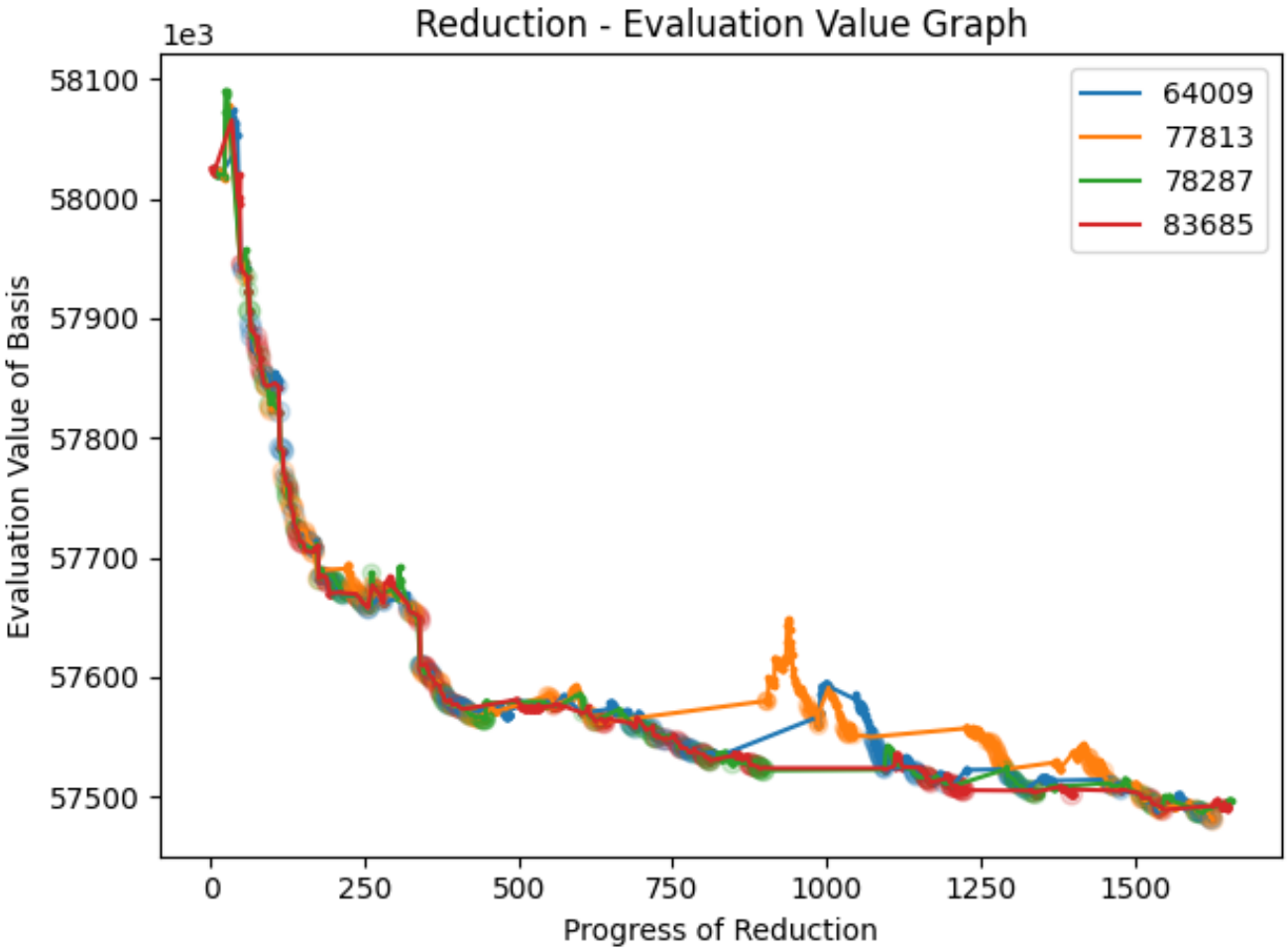

図 10　終盤にプロセス間で協力して評価値を下げていく様子

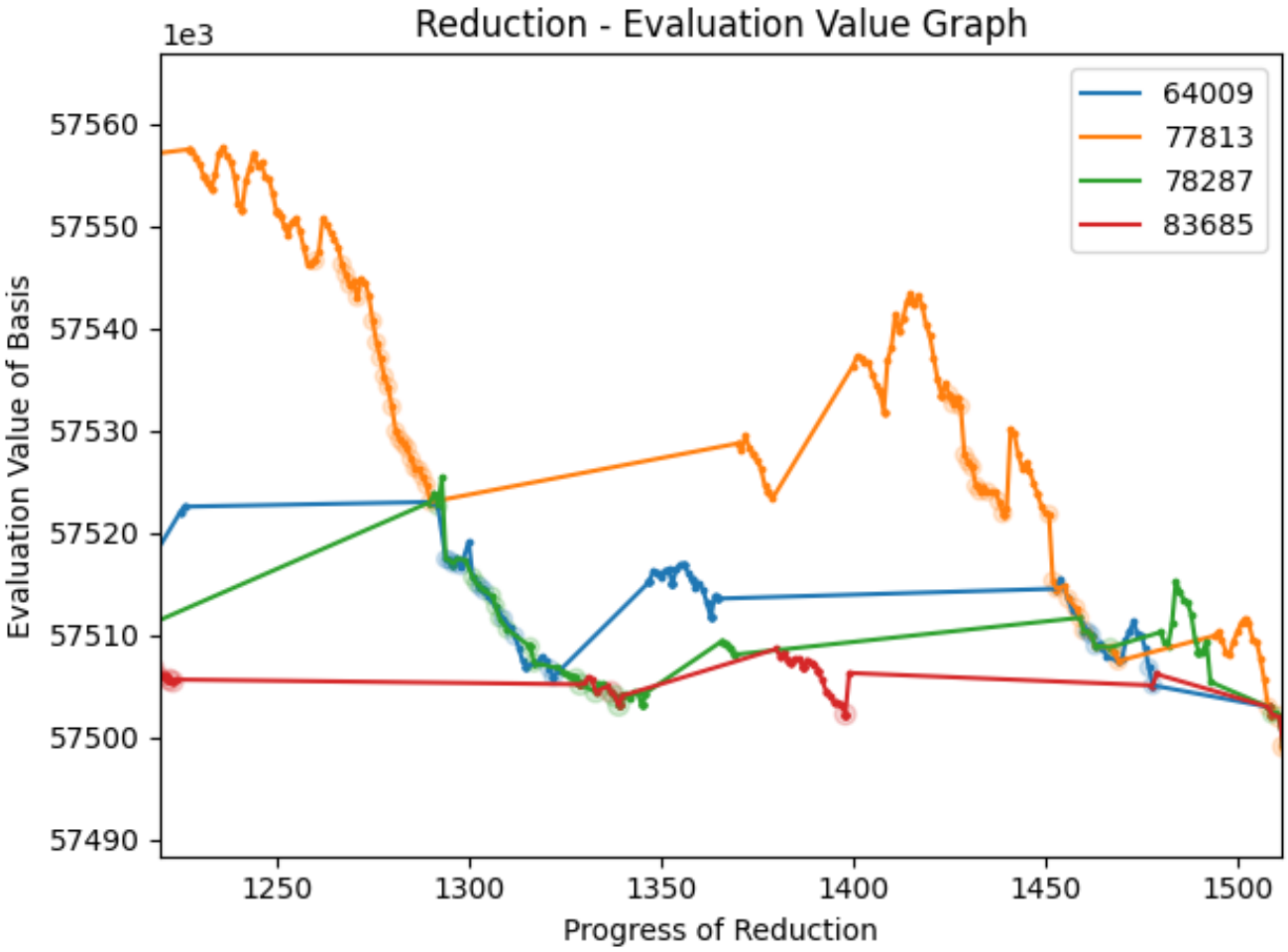

図 11　プロセス間で協力して評価値を下げていく様子の拡大図

基底の簡約順序に対する評価値の変化のグラフも示す (図 10)。凡例の数字は、ランダムに振ったプロセス番号である。図 11 のほうは、図 10 の横軸の目盛りである基底の簡約順序で 1300 番目あたりのグラフを拡大したものである。グラフ上のすこし滲んだ点が BEST BASIS や FRONT BASIS を表している。BEST BASIS よりもやや上にある滲んだ点が FRONT BASIS を表している。2 つのプロセス間で簡約順序と評価値の両方の基準で優劣がついた場合に BEST BASIS の基底とメモリの基底の間でコピーが起こっている。FRONT BASIS も同様である。

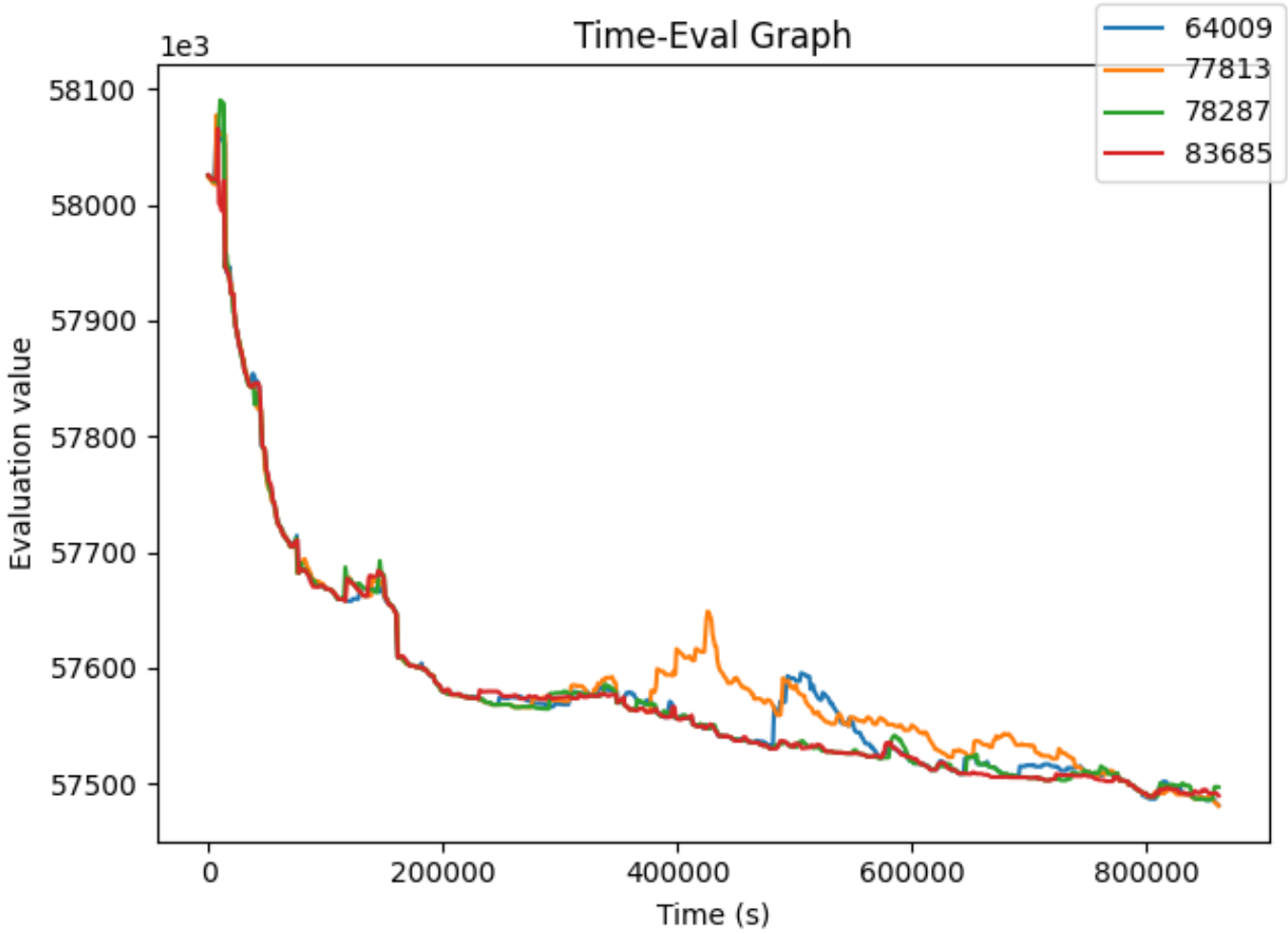

図 12　経過時間に対する評価値の変化

図 13 に時間経過にそった基底の自乗和 $\sum_{m=0}^{L-1} |\mathbf{b}_m^*|^2$ の変化のグラフも示す。縦軸は、Babai lifting の対象となる index の直交基底ベクトルの長さの自乗和である。グラフの最後のほうは、前のほうの index があまり更新しなくなってきたので和が下がってきている。このときは、短い格子ベクトルの生成数も上がるが、ベクトルの重複も増えるのでそれほど効率が改善しているわけでもない。このグラフの実験した時間よりも広い範囲で、もう少し長期的にみると徐々に効率が上がってきていることが観察されている。

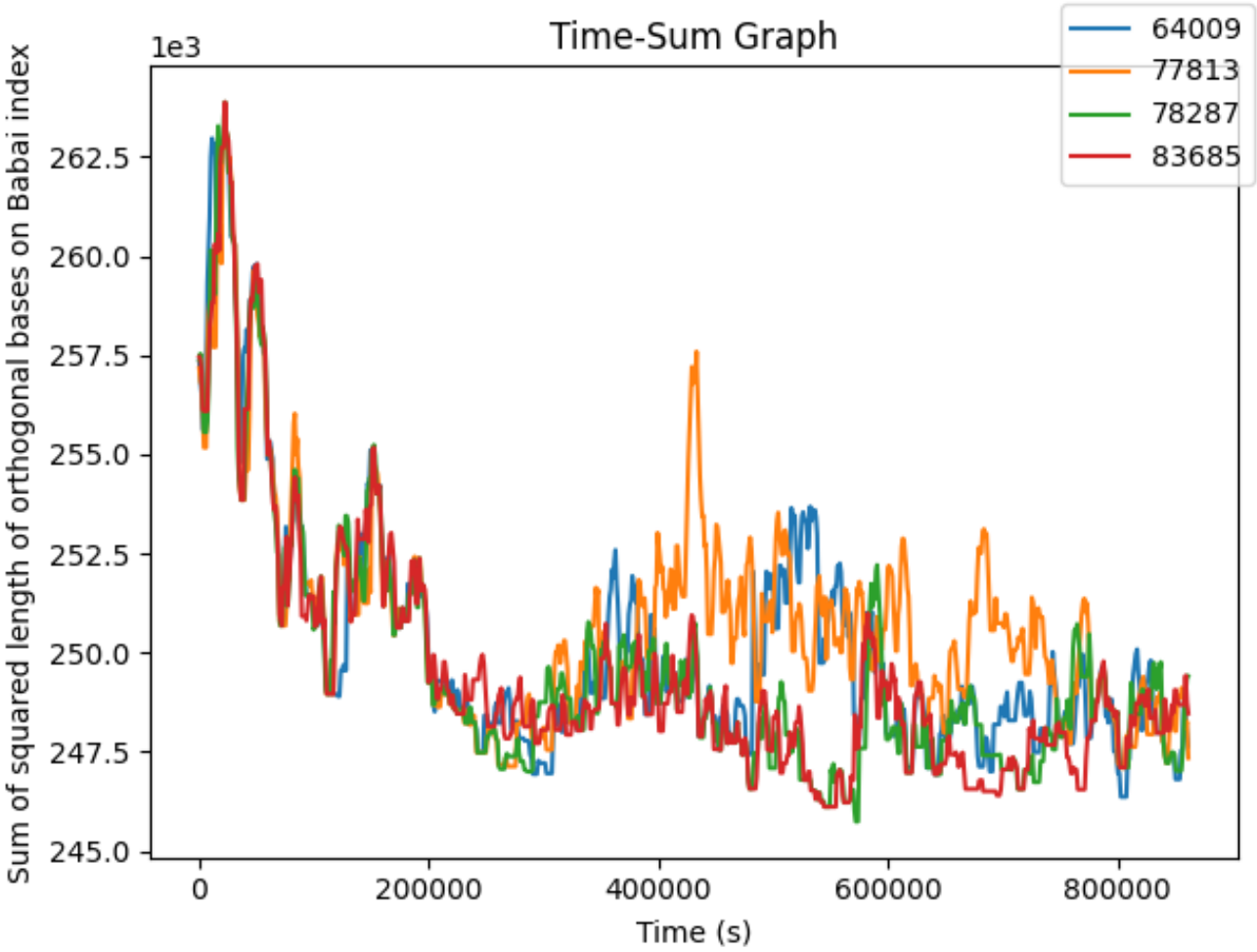

図 13　経過時間に対する自乗和の変化

図 14 にプロセスごとにどの index で基底簡約が行われたかのグラフも示す。縦軸が基底簡約の挿入 index であり、横軸が経過時間 (単位は秒) である。重複のグラフと比べてみることで、後ろの index で基底簡約されたときのほうがやや重複が多くなっていることが確認できる。

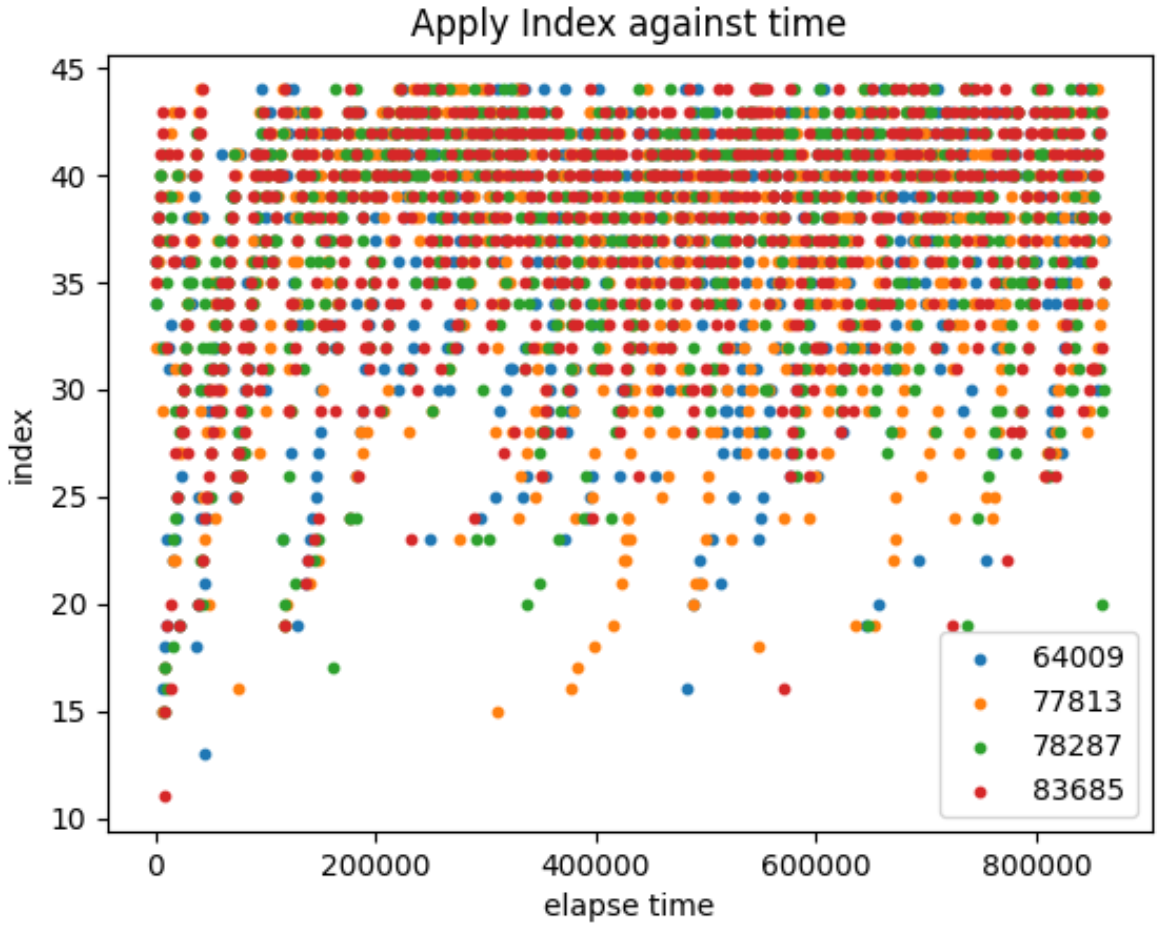

図 14　基底簡約が行われた index

図 15 に時間ごとに記録されたベクトルの数がどのように変化したかのグラフも示す。重複しているベクトルは 1 つと数えて、最初に記録されたところにいれている。ベクトルは sieving 終了後にまとめて記録されるためにグラフが細かく上下してしまうので、グラフを少し平均化して見やすくしている。基底の自乗和の減少に伴って、短いベクトルの数が増えている。今回は長い基底簡約の一部の期間だけ取り出したグラフであるが、もっと長い期間で比べると自乗和を下げることで、得られるベクトルの数が何倍も違ってくる。

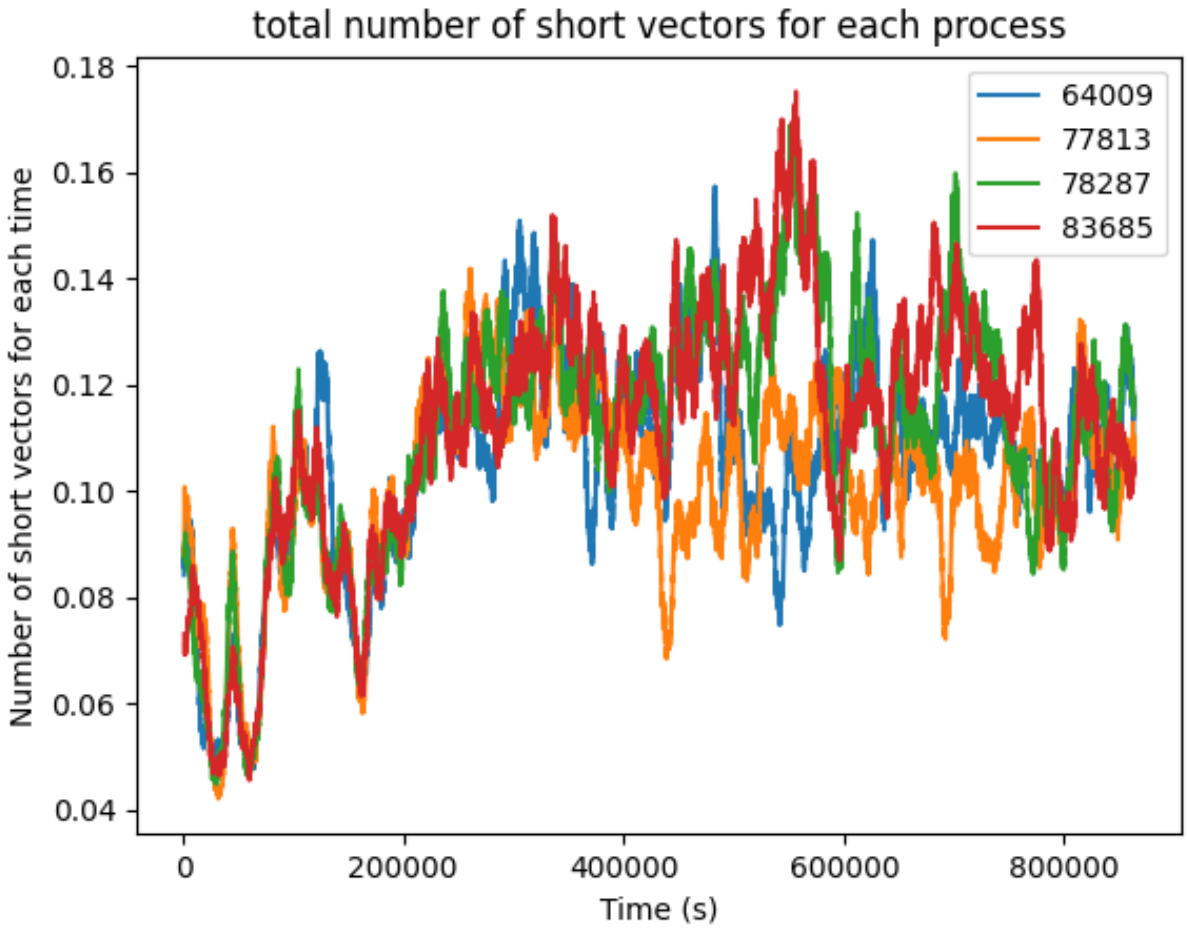

図 15　経過時間に対する、重複を除いた得られたベクトルの数

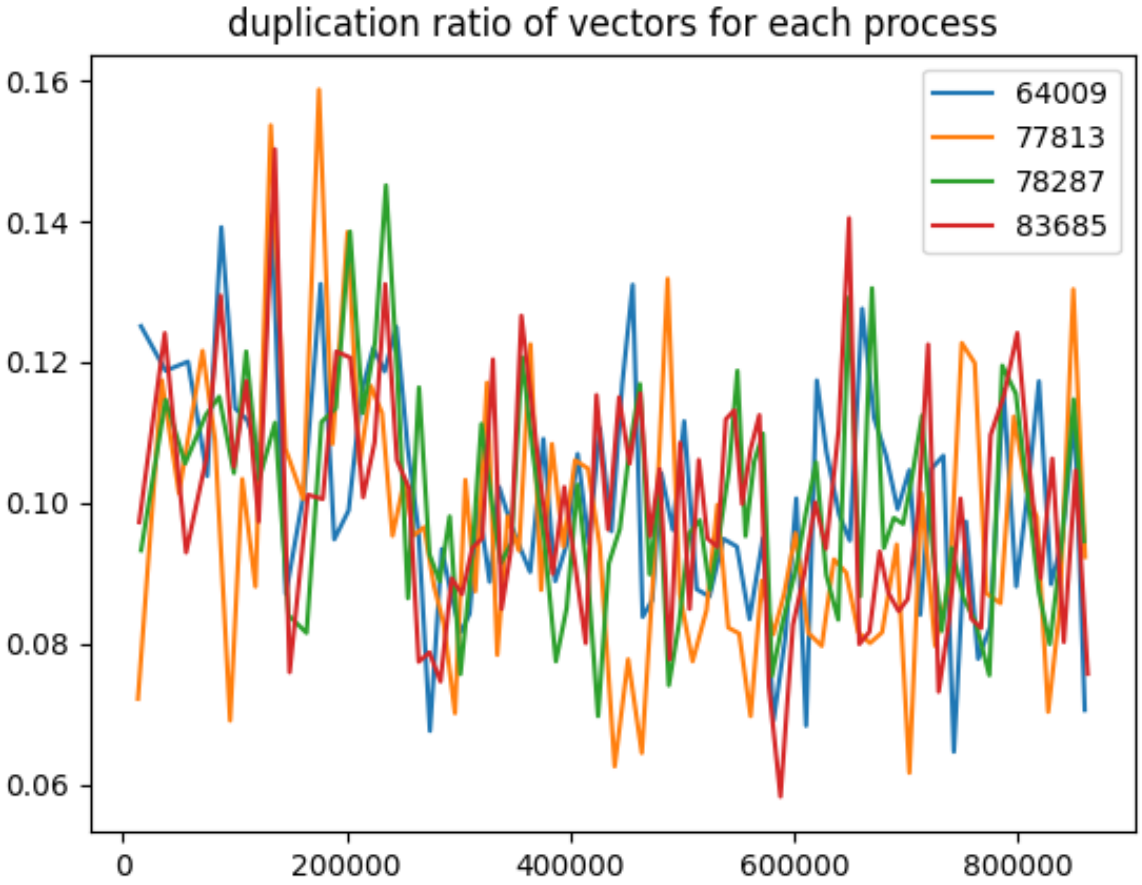

図 16　プロセスごとの経過時間に対する重複率の変化

最後に図 17 に、基底にベクトルを適用する際に、ベクトルの評価値と、基底の評価値の増減の関係のグラフを示す。increasing は、そのベクトルにより基底簡約が行われた結果、基底の評価値が増えた場合、decresing は、基底の評価値が下がった場合である。横軸が基底簡約が行われた index で、縦軸がそのベクトルの評価値である。その index の更新幅と、その index+1 の更新幅により、評価値は計算される。グラフより、後ろの index のほうが基底の評価値を下げることが多いことがわかる。評価値が高いほど、基底の評価値を下げることもわかる。

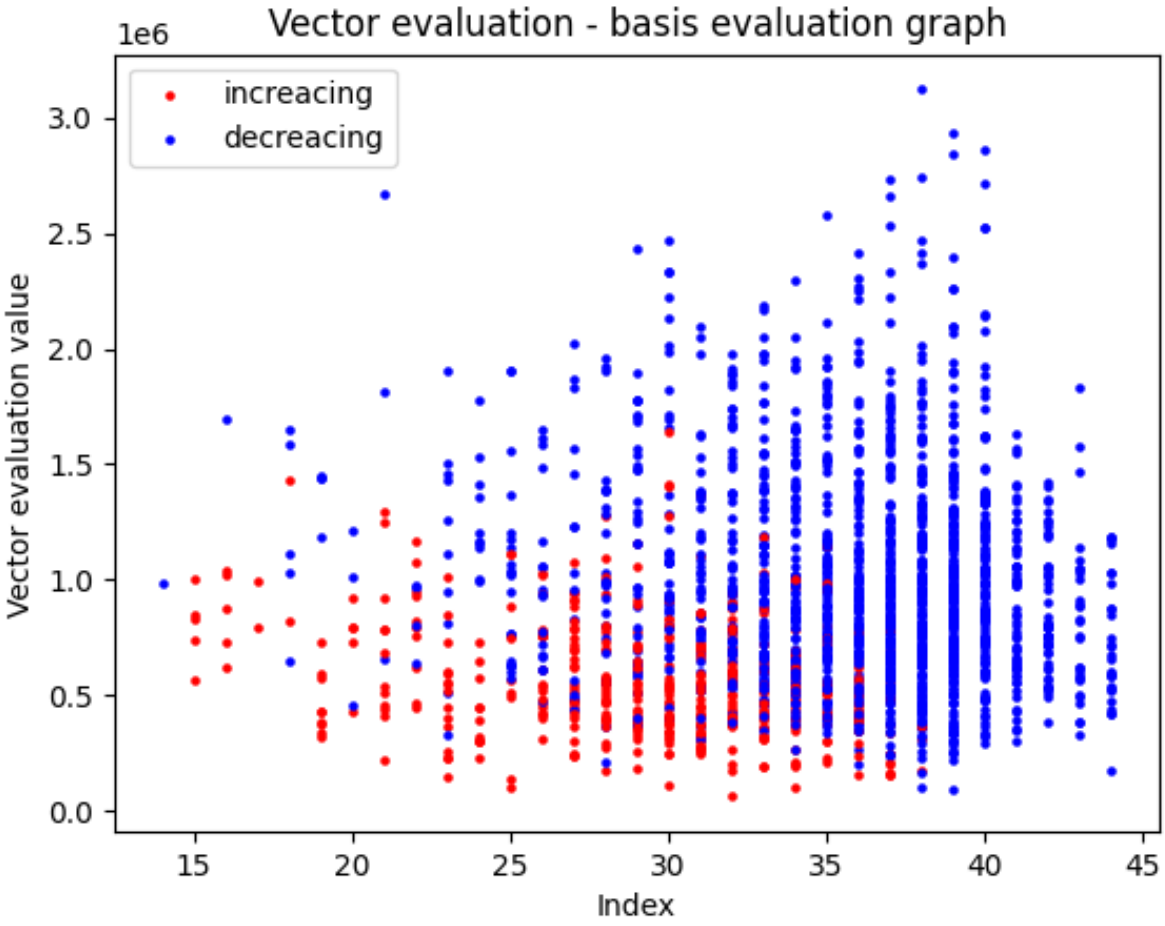

図 17　ベクトルの評価値と、基底の評価値の関係

最後に、重複が起こったベクトルと重複しなかったベクトルの sieving index 上の直交補空間での長さを時間経過でプロセスごとにグラフにしたものである。点線が重複が起こらなかったベクトルで、実線が重複が起ったベクトルのグラフである。重複が起こったベクトルのほうが sieving 部分の長さが短いことがわかる。140 次元の格子で Babai index L が 60 次元で sieving index が 80 次元での計算である。

7　まとめ

実験により、基底簡約の終盤で直交基底ベクトルの自乗長さの和を下げることにより、やや短いベクトルが得られる頻度が上がっていくことがわかった。われわれの基底簡約アルゴリズムにおいては、前後で基底が近くなる簡約の終盤においても、全体空間におけるベクトルの重複により大きく効率が落ちているということはなかった。一定程度の回数、基底簡約を行えば、ほぼ重複しないことがわかった。

本研究においてプロセス並列を用いた分散並列計算の基底簡約アルゴリズムを提案した。並列計算において、プロセス間での基底のゆるい共有が有効であることがわかった。基底を共有することでプロセス間で協力して、評価関数を改善することができる。特に基底簡約の序盤では、プロセス間の協力関係が有効に働いて、基底簡約の効率化に貢献した。基底簡約の終盤では、各プロセスが独立に振る舞うようになり、プロセスにまたがるベクトルの重複を防ぐことに成功した。メモリの制限のために高次元の sieving 計算が行えないような計算機環境においてもある程度、効率的な計算を行うことができた。ただし、メモリが大きいマシンを利用して計算した記録には追いつけていない。

基底簡約アルゴリズムの改善の今後の展望を書く。基底簡約の終盤においては、次元の高い sieving を行うことが効率的であるが、高次元の sieving には多くのベクトルが必要なためメモリの制約を受ける。メモリが少なくても高次元の sieving が行えるようにアルゴリズムの改善に取り組みたい。基底簡約の前半では次元の低い小さめな sieving を利用し、簡約が進んだ後は次元の高い基底簡約が有効であることがわかったが、現在はプロセスの再起動による手動による調整を併用しているので、今後は、全部自動で行えるようにプログラムを工夫したい。

また、今後の研究課題としては、本研究において基底簡約の方法を工夫したが、もともとのオリジナルの G6K との性能の比較をまだ、あまり行なっていないので、それを行いたい。

利益相反

本論文に関して，開示すべき利益相反関連事項はない。

謝辞

本研究は JSPS 科学研究費補助金 20K11669 の助成を受けたものです。

参考文献

[1] Martin R. Albrecht, Léo Ducas, Gottfried Herold, Elena Kirshanova, Eamonn W. Postlethwaite, and Marc Stevens. The general sieve kernel and new records in lattice reduction. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part II*, volume 11477 of *Lecture Notes in Computer Science*, pages 717–746. Springer, 2019.

[2] Michal Andrzejczak and Kris Gaj. A multiplatform parallel approach for lattice sieving algorithms. *IACR Cryptol. ePrint Arch.*, 2021:973, 2020.

[3] Yoshinori Aono and Phong Q. Nguyen. Random sampling revisited: Lattice enumeration with discrete pruning. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part II*, volume 10211 of *Lecture Notes in Computer Science*, pages 65–102, 2017.

[4] László Babai. On lovász' lattice reduction and the nearest lattice point problem. *Combinatorica*, 6(1):1–13, 1986.

[5] Joppe W. Bos, Michael Naehrig, and Joop van de Pol. Sieving for shortest vectors in ideal lattices: a practical perspective. *Int. J. Appl. Cryptogr.*, 3(4):313–329, 2017.

[6] Özgür Dagdelen and Michael Schneider. Parallel enumeration of shortest lattice vectors. In Pasqua D'Ambra, Mario Rosario Guarracino, and Domenico Talia, editors, *Euro-Par 2010 - Parallel Processing, 16th International Euro-Par Conference, Ischia, Italy, August 31 - September 3, 2010, Proceedings, Part II*, volume 6272 of *Lecture Notes in Computer Science*, pages 211–222. Springer, 2010.

[7] TU Darmstadt. SVP Challenge. `http://www.latticechallenge.org/svp-challenge/`.

[8] Léo Ducas, Marc Stevens, and Wessel van Woerden. Advanced lattice sieving on gpus, with tensor cores. Cryptology ePrint Archive, Report 2021/141, 2021. `https://ia.cr/2021/141`.

[9] Léo Ducas. Shortest vector from lattice sieving: A few dimensions for free. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018*, pages 125–145, Cham, 2018. Springer International Publishing.

[10] Masaharu Fukase and Kenji Kashiwabara. An accelerated algorithm for solving SVP based on statistical analysis. *JIP*, 23(1):67–80, 2015.

[11] Nicolas Gama, Phong Q. Nguyen, and Oded Regev. Lattice enumeration using extreme pruning. In Henri Gilbert, editor, *Advances in Cryptology - EUROCRYPT 2010, 29th*

Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30 - June 3, 2010. Proceedings, volume 6110 of *Lecture Notes in Computer Science*, pages 257–278. Springer, 2010.

[12] C. Heckler and L. Thiele. Complexity analysis of a parallel lattice basis reduction algorithm. *SIAM Journal on Computing*, 27(5):1295–1302, 1998.

[13] Jens Hermans, Michael Schneider, Johannes A. Buchmann, Frederik Vercauteren, and Bart Preneel. Parallel shortest lattice vector enumeration on graphics cards. In Daniel J. Bernstein and Tanja Lange, editors, *Progress in Cryptology - AFRICACRYPT 2010, Third International Conference on Cryptology in Africa, Stellenbosch, South Africa, May 3-6, 2010. Proceedings*, volume 6055 of *Lecture Notes in Computer Science*, pages 52–68. Springer, 2010.

[14] J. Hoffstein, J. Pipher, and J.H. Silverman. *An Introduction to Mathematical Cryptography*. Undergraduate Texts in Mathematics. Springer New York, 2008.

[15] Tsukasa Ishiguro, Shinsaku Kiyomoto, Yutaka Miyake, and Tsuyoshi Takagi. Parallel gauss sieve algorithm: Solving the svp challenge over a 128-dimensional ideal lattice. In Hugo Krawczyk, editor, *Public-Key Cryptography – PKC 2014*, pages 411–428, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[16] Ravi Kannan. Improved algorithms for integer programming and related lattice problems. In David S. Johnson, Ronald Fagin, Michael L. Fredman, David Harel, Richard M. Karp, Nancy A. Lynch, Christos H. Papadimitriou, Ronald L. Rivest, Walter L. Ruzzo, and Joel I. Seiferas, editors, *Proceedings of the 15th Annual ACM Symposium on Theory of Computing, 25-27 April, 1983, Boston, Massachusetts, USA*, pages 193–206. ACM, 1983.

[17] Thijs Laarhoven and Artur Mariano. Progressive lattice sieving. In T. Lange and R. Steinwandt, editors, *Post-Quantum Cryptography*, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), pages 292–311, Germany, 2018. Springer. 9th International Conference on Post-Quantum Cryptography (PQCrypto 2018), PQCrypto 2018 ; Conference date: 09-04-2018 Through 11-04-2018.

[18] Christoph Ludwig. *Practical Lattice Basis Sampling Reduction*. PhD thesis, Technische Universität Darmstadt Universitäts, 2005.

[19] Yoshitatsu Matsuda, Tadanori Teruya, and Kenji Kashiwabara. Efficient estimation of number of short lattice vectors in search space under randomness assumption. In Keita Emura and Takaaki Mizuki, editors, *Proceedings of the 6th on ASIA Public-Key Cryptography Workshop, APKC@AsiaCCS 2019, Auckland, New Zealand, July 8, 2019*, pages 13–22. ACM, 2019.

[20] Yoshitatsu Matsuda, Tadanori Teruya, and Kenji Kashiwabara. Efficient estimation of number of short lattice vectors in search space under randomness assumption. In *Proceed-*

ings of the 6th on ASIA Public-Key Cryptography Workshop, APKC '19, pages 13–22, New York, NY, USA, 2019. ACM.

[21] Phong Q. Nguyen and Thomas Vidick. Sieve algorithms for the shortest vector problem are practical. *Journal of Mathematical Cryptology*, 2(2):181–207, 2008.

[22] Claus-Peter Schnorr. Lattice reduction by random sampling and birthday methods. In Helmut Alt and Michel Habib, editors, *STACS 2003, 20th Annual Symposium on Theoretical Aspects of Computer Science, Berlin, Germany, February 27 - March 1, 2003, Proceedings*, volume 2607 of *Lecture Notes in Computer Science*, pages 145–156. Springer, 2003.

[23] Claus-Peter Schnorr and M. Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Math. Program.*, 66:181–199, 1994.

[24] Tadanori Teruya, Kenji Kashiwabara, and Goichiro Hanaoka. Fast lattice basis reduction suitable for massive parallelization and its application to the shortest vector problem. In Michel Abdalla and Ricardo Dahab, editors, *Public-Key Cryptography - PKC 2018 - 21st IACR International Conference on Practice and Theory of Public-Key Cryptography, Rio de Janeiro, Brazil, March 25-29, 2018, Proceedings, Part I*, volume 10769 of *Lecture Notes in Computer Science*, pages 437–460. Springer, 2018.

[25] The FPLLL development team. fplll, a lattice reduction library. Available at https://github.com/fplll/fplll, 2016.

[26] The G6K development team. G6K, the general sieve kernel. Available at https://github.com/fplll/g6k, 2018.

[27] Gilles Villard. Parallel lattice basis reduction. In *ISSAC '92*, 1992.