COMPUTATIONAL VERIFICATION OF THE CONE CONJECTURE

A thesis presented to the faculty of
San Francisco State University
In partial fulfilment of
The Requirements for
The Degree

Master of Arts
In
Mathematics

by

Tsz Leong Chan

San Francisco, California

December 2018

CERTIFICATION OF APPROVAL

I certify that I have read *COMPUTATIONAL VERIFICATION OF THE CONE CONJECTURE* by Tsz Leong Chan and that in my opinion this work meets the criteria for approving a thesis submitted in partial fulfillment of the requirements for the degree: Master of Arts in Mathematics at San Francisco State University.

Dr. Joseph Gubeladze
Professor of Mathematics

Dr. Matthias Beck
Professor of Mathematics

Dr. Serkan Hosten
Professor of Mathematics

COMPUTATIONAL VERIFICATION OF THE CONE CONJECTURE

Tsz Leong Chan
San Francisco State University
2018

The set of polyhedral pointed rational cones form a partially ordered set with respect to elementary additive extensions of certain type. This poset captures a global picture of the interaction of all possible rational cones with the integer lattice and, also, provides an alternative approach to another important poset, the poset of normal polytopes. One of the central conjectures in the field is the so called Cone Conjecture: the order on cones is the inclusion order. The conjecture has been proved only in dimensions up to 3. In this work we develop an algorithmic approach to the conjecture in higher dimensions. Namely, we study how often two specific types of cone extensions generate the chains of cones in dimensions 4 and 5, whose existence follows from the Cone Conjecture. A naïve expectation, explicitly expressed in a recently published paper by Dr. Gubeldaze, is that these special extensions suffice to generate the desired chains. This would prove the conjecture in general and was the basis of the proof of the 3-dimesional case. Our extensive computational experiments show that in many cases the desired chains are in fact generated, but there are cases when the chain generation process does not terminate in reasonable time. Moreover, the fast generation of the desired chains fails in an interesting way—the complexity of

the involved cones, measured by the size of their Hilbert bases, grows roughly linearly in time, making it less and less likely that we have a terminating process. This phenomenon is not observed in dimension 3. Our computations can be done in arbitrary high dimensions. We make a heavy use of SAGE, an open-source mathematics software system, and Normaliz, a C++ package designed to compute the Hilbert bases of cones.

I certify that the Abstract is a correct representation of the content of this thesis.

Chair, Thesis Committee Date

ACKNOWLEDGMENTS

Dr. Matthias Beck, Dr. Joseph Gubeladze, and Dr. Serkan Hosten for their continued support at San Francisco State University; Dr. Jean-Phillip Labbé (Berlin), Dr. Winfred Bruns (Osnabrück), Dr. Sebastian Gutsche (Siegen), Dr. Matthias Koeppe (UC Davis), Dr. Moritiz Firsching (Berlin), Dr. Yuen Zhou (UKY) for their guidance during a SAGE coding sprint hosted at The Institute of Mathematics and its Applications at University of Minnesota in April 2018; Ms. Juniper Overbeck, for advising with coding standards; Dr. Sana Patel, for supporting me throughout the writing of this thesis; and my family and friends, for helping me stay sane through it all.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Chapter 1

Introduction

Normal polytopes and rational cones are important objects in integer programming, optimization, combinatorics, and algebraic geometry [3]. Recently, Bruns, Gubeladze, and Michałek [4] initiated a novel approach to these discrete-convex objects by introducing partial orders on them. The resulting poset of normal polytopes is a discrete model of the continuum of convex compacta in Euclidean spaces. Understanding properties of this poset is a challenge and only partial results in this direction are known to date. Later, Gubeladze and Michałek [11] defined a partial order on the set rational cones. The latter poset seems more amenable to analysis than the poset of normal polytopes. What is important, if the partial order on cones is trivial, i.e., coincides with the inclusion order (the Cone Conjecture), then the cones provide a handle on the poset of normal polytopes via the homogenization map. Gubeladze and Michałek were able to prove the Cone Conjecture in dimension 3 and Paffenholz provided some com-

putational evidence in dimension 4.

In this thesis we develop computational methods to analyze the Cone Conjecture in higher dimensions and provide big data in dimensions 4 and 5, far exceeding the computational results by Paffenholz. Namely, we use two specific types of moves inside the poset of cones (the so-called "Bottom Up" and "Top Down" moves) to link two randomly generated cones, one containing the other. The Cone Conjecture states that any such a pair of cones is linked by a chain. The mentioned specific moves often produce such chains. But there are non-terminating examples as well. In order to get a deeper insight into the complexity of the poset of cones, we keep track of the size of Hilbert bases of the intermediate cones and the lengths of pivotal vectors for the bottom-up and top-down moves. The conclusion is that the method used by Gubeladze and Michałek to prove the conjecture in dimension 3 is not appropriate in higher dimensions. New ideas, needed to tackle the conjecture in general, may come from the observed strange monotonicity trends in the Hilbert basis size along the chains of bottom-up and top-down moves.

In Sections 2 and 3, we give a brief introductory exposition on polytopes, cones, normal polytopes and rational cones. Then, in Sections 4 and 5, we give an exposition on the detailed techniques and implemented algorithms, along with many computational results.

Chapter 2

Background

In this project, we are interested in pointed, polyhedral rational cones, which are a special case of polyhedra. Here, we will give rigorous definitions, which are the foundations necessary to discuss the computational experimentation. While the following definitions are general and can be applied to any vector space with an inner product, we are in particular interested in $\mathbb{R}^d$.

2.1 Polyhedra

First, we begin with the idea of linear subspaces of a vector space. Linear subspace is the set of vectors that vanishes on a linear form $a_1x_1 + \cdots + a_nx_n = 0$ for some scalar coefficients. In this paper, we are interested when the vector space is $\mathbb{R}^d$. Consider the linear subspace shifted away from the origin; we will arrive at the first major concept we need to build up to the definitions of polyhedra

and polytopes:

Definition 2.1 (Affine subspace)**.** Affine subspaces are the translates of linear subspaces. The dimension of an affine subspace is the dimension of the associated linear subspace.

Affine subspaces of dimension 0, 1 and 2 are called points, lines and planes, respectively.

Let V be a vector space and $\alpha : V \to \mathbb{R}$ be an affine form, which is a function given by $\alpha(x) = \lambda(x) + \alpha_0$, with a unique linear operator λ on V. We define a hyperplane by the following construction [3].

Definition 2.2 (Affine Hyperplane)**.**

$$H_\alpha = \{x \in \mathbb{R}^d : \alpha(x) = 0\}$$

A *hyperplane* is the set of vectors in $\mathbb{R}^d$ that vanish on the affine form $\alpha(x)$. A hyperplane is an affine subspace of dimension $d - 1$.

Thus, when evaluating any point $x \in \mathbb{R}^d$ that is not on the hyperplane, $\alpha(x) > 0$ or $\alpha(x) < 0$. This gives us a sense of partitioning, as the hyperplane will partition the space into two halves. Imagine cutting $\mathbb{R}^d$ into two parts using a $(d-1)$-dimensional plane; each of the two parts is an affine halfspace.

Definition 2.3 (Open and Closed Affine Halfspaces)**.** An open affine halfspace

is defined as

$$H_\alpha^> = \{x \in V : \alpha(x) > 0\}$$

for some affine form $\alpha(x)$. A closed halfspace H_α^+ is defined as the union of H_α and $H_\alpha^>$.

Formally, according to standard graduate texts in mathematics on geometry [3, 16], we can define a *polyhedron* as the intersection of halfspaces:

Definition 2.4 (H-Polyhedron)**.** A subset $P \subset V$ is called an H-polyhedron if it is the intersection of finitely many closed affine halfspaces. The dimension of the polyhedron is determined by the dimension of the smallest affine subspace containing P. If $\dim(P) = d$, we call P a d-polyhedron.

As one can imagine, since the H-Polyhedron definition depends on an intersection of halfspaces, there exist special hyperplanes for each H-polyhedron where each hyperplane is formed from the associated halfspaces. Furthermore, there are hyperplanes that "touch" the H-polyhedron, and how they "touch" the H-polyhedron is how we define faces of lower dimensions. An example in $\mathbb{R}^3$ is a cube, which will have 6 facets of dimension 2, 12 faces (edges) of dimension 1, and 8 faces (vertices) of dimension 0. This condition is formalized for higher dimensions as follows:

Definition 2.5 (Support Hyperplane, Face)**.** A hyperplane H is called a support

hyperplane of the polyhedron P if P is contained in one of the two closed halfspaces bounded by H, and $H \cap P \neq \emptyset$.

In particular, the intersection $F = H \cap P$ is a face of P, and H is called a support hyperplane associated with F.

An H-polyhedron is the intersection of finitely many closed halfspaces, and each halfspace is associated with an affine hyperplane; the intersection of these affine hyperplanes and the polytope form the facets, or faces of dimension $d-1$. Other affine hyperplanes may also satisfy the support hyperplane definition, but the intersection may be lower dimensional.

Visually, in $\mathbb{R}^3$, we can imagine support hyperplanes as hyperplanes (in $\mathbb{R}^3$ hyperplanes are simply 2-dimensional planes) that touch the polyhedron's boundary without cutting through the object, and the part of the polytope that the hyperplane "touches" is a face. A 0,1 and $d-1$ dimensional face of a polytope is called a vertex, an edge and a facet, respectively.

2.2 Convex Polytopes

The polytopes in this paper are always convex, so first we define convexity and the convex hull of a set.

A set $X \subset \mathbb{R}^d$ is *convex* when between any two points in X, the line segment joining the two points is also contained in X. More formally:

Definition 2.6 (Convex)**.** A set $X \subset \mathbb{R}^d$ is convex if for every $x, y \in X$:
$\{\lambda x + (1-\lambda)y : 0 \leq \lambda \leq 1\} \subset X$.

As a remark, the intersection of any convex set is again convex. More importantly for any arbitrary set $K \subset \mathbb{R}^d$, there exists a smallest convex set containing K constructed as the intersection of all the convex sets that contain K. Alternatively, one can say that the convex hull of K contains every line segment joining any two elements in the set K.

Definition 2.7 (Convex hull)**.** Given $K \subset \mathbb{R}^d$, the convex hull of K, denoted $\text{conv}(K)$, is the smallest convex set that contains K:

$$\begin{aligned}\text{conv}(K) &= \bigcap \{X \subset \mathbb{R}^d : K \subseteq X, X \text{ convex}\} \\ &= \{\lambda_1 x_1 + \cdots + \lambda_n : \{x_1, \ldots, x_n\} \subseteq K, \lambda_i \geq 0, \sum_{i=1}^{n} \lambda_i = 1\}.\end{aligned}$$

We will explore two different definitions of the polytope; they are unique representations for each polytope and mathematically equivalent through a non-trivial proof.

Definition 2.8 (V-polytope)**.** A V-polytope is the convex hull of a finite set of points in $\mathbb{R}^d$.

The other definition that depends on the previous section is the halfspace representation of a polytope:

Definition 2.9 (H-polytope)**.** An H-polytope is an H-polyhedron that is bounded. Equivalently, an H-polytope P contains no rays of the form $\{x + ty : t \geq 0\}$ for any $x \in P, y \neq 0$.

Finally, a *polytope* is a set $P \subseteq \mathbb{R}^d$ which can be represented as a V-polytope or a H-polytope. Like the definition for the dimension of a polyhedron, the dimension of a polytope is the dimension of the smallest affine subspace that contains this polytope.

2.3 Cones

Cones are intimately related to polytopes and polyhedra. In this narrative to discuss the main theorem of polytope theory, we will show how to embed both H-polyhedra and V-polyhedra of dimension d into $\mathbb{R}^{d+1}$ as cones. This intimate relation between cones and polytopes is also the motivation later for the relationship between normal polytopes and rational cones.

A cone in a vector space (or $\mathbb{R}^d$) is defined as the following:

Definition 2.10 (Cone)**.** A set $C \subset \mathbb{R}^d$ is a cone if for every $x \in C, \lambda x \in C$ for every $\lambda \geq 0$.

Clearly, the intersection of cones is again a cone, and some trivial cases are the empty set $\{\emptyset\}$ and the whole space $\mathbb{R}^d$.

Given any set $K \subset \mathbb{R}^d$, we are also interested in the smallest cone that contains this set. Similar to the construction of the convex hull, the *conical hull* can be characterized as the intersection of all the cones that contain K or as non-negative linear combinations of all the elements of K:

Definition 2.11 (Conical hull)**.** Given a set $K \subseteq \mathbb{R}^d$, the conical hull of K, denoted $\text{cone}(K)$, is the smallest cone that contains K, and

$$\begin{aligned}(K) &= \bigcap\{C \subseteq \mathbb{R}^d : K \subseteq C, C \text{ is a cone}\},\\ &= \{\lambda_1 x_1 + \cdots + \lambda_n x_n : \{x_1, \ldots, x_k\} \subseteq K, \lambda_i \geq 0\}.\end{aligned}$$

In particular, we'll be interested in the case when K is a finite set.

The *Minkowski sum* of two sets $P, Q \subseteq \mathbb{R}^d$ is $P + Q = \{p + q : p \in P, q \in Q\}$. With this operation, we can define a V-polyhedron as the Minkowski sum of the conical hull of a finite set of vectors and the convex hull of another finite set of vectors:

Definition 2.12 (V-polyhedron)**.** A *V-polyhedron* denotes any finitely generated convex-conical combination, in other words, a set $P \subset \mathbb{R}^d$ of the form

$$P = \text{conv}(V) + \text{cone}(Y)$$

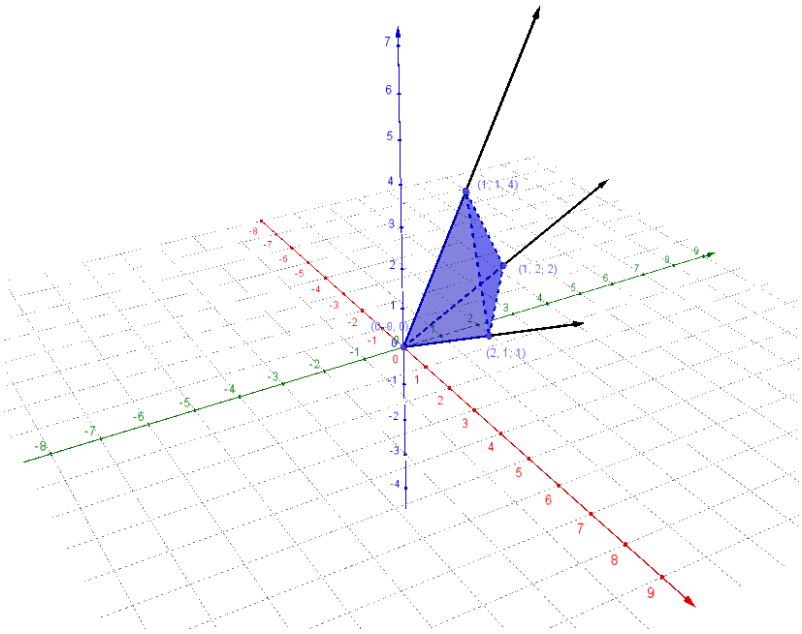


Figure 2.1: Example of cone(K) for some finite $K \subset \mathbb{R}^3$.

for some finite sets $V, Y \subset \mathbb{R}^d$.

From this definition, a *V-polytope* is a bounded *V-polyhedron*.

To draw the definitive relationship between the unique characterization of the V-polytope as the convex hull of finitely many points and H-polytope as the intersection of finite halfspaces, one strategy is to show that there exists a unique homogenous cone in dimension $d + 1$, according to [16].

2.3.1 Homogenization of Polyhedron

The narrative in [16] introduced the homogenization map, by simply adjoining an extra coordinate to the vectors with the intention to reduce the narrative to show that a V-polyhedra in $\mathbb{R}^d$ is also an H-polyhedra in $\mathbb{R}^d$ by associating a well-defined cone with each type in $\mathbb{R}^{d+1}$ in the following ways, with the goal of embedding the polyhedra in the plane $x_0 = 1$ and forming a cone to represent such a polyhedra without loss of information.

If P_H is some H-polyhedron, i.e., $P_H = \{x \in \mathbb{R}^d : Ax \leq z, A \in \mathbb{R}^{m\times d}, z \in \mathbb{R}^m\}$, we can create a cone in $\mathbb{R}^{d+1}$ by encoding the inequalities $a_i\mathbf{x} \leq z_i$ into $a_i\mathbf{x} \leq z_i x_0$ with $x_0 \geq 0$. More specifically, let

$$C(P_H) = \begin{bmatrix} -1 & 0 & \cdots & 0 \\ -z_0 & a_{11} & \cdots & a_{1d} \\ \vdots & \vdots & \ddots & \vdots \\ -z_m & a_{m1} & \cdots & a_{md} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_d \end{bmatrix} \leq \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}. \tag{2.1}$$

Note that when $x_0 = 1$, we have the polytope P in the plane intersecting $C(P)$, and that $C(P)$ and P are both H-polyhedra.

Similarly, if P_V is some V-polyhedra where $P_V = \text{conv}(U) + \text{cone}(W)$ for some finite sets $U, W \subset \mathbb{R}^d$, we can form a $C(P_V)$ in $\mathbb{R}^{d+1}$ as follows:

$$C(P_V) = \text{cone}\left(\begin{bmatrix}1\\u_{11}\\\vdots\\u_{d1}\end{bmatrix}, \cdots, \begin{bmatrix}1\\u_{1j}\\\vdots\\u_{dj}\end{bmatrix}, \begin{bmatrix}0\\w_{11}\\\vdots\\w_{d1}\end{bmatrix}, \cdots, \begin{bmatrix}0\\w_{1k}\\\vdots\\w_{dk}\end{bmatrix}\right), \tag{2.2}$$

The construction creates a cone in $\mathbb{R}^{d+1}$, as the conical hull of these vectors, with the property that

$$P = \left\{\mathbf{x} \in \mathbb{R}^d : \begin{bmatrix}1\\\mathbf{x}\end{bmatrix} \in C(P)\right\}.$$

And the points at infinity (the part of P that is defined using the conical hull of Y) is recorded on the plane $x_0 = 0$.

Theorem 2.1 (Main theorem of polytope theory: Cone Version)**.** *A cone $C \subseteq \mathbb{R}^d$ is finitely generated combination of vectors*

$$C = \text{cone}(V) \textit{ for some finite } V \subset \mathbb{R}^d$$

if and only if it is a finite intersection of closed linear halfspaces

$$C = \{x \in \mathbb{R}^d : Ax \leq 0\} \textit{ for some } A \in \mathbb{R}^{m \times d}\}.$$

The proof of this theorem is nontrivial and can be found in Section 1.3 of [16].

With the information from Theorem 2.1, and the aforementioned construction of cones using V-polyhedra and H-polyhedra, together they imply (through the construction of the homogenization map) the following theorem.

Theorem 2.2 (Main Theorem of Polytope Theory: Polyhedral version)**.** *A subset $P \subseteq \mathbb{R}^d$ is the Minskowski sum of the convex hull of a finite point set and the conical hull of another finite point set (a* V-polyhedron*)*

$$P = \text{conv}(V) + \text{cone}(Y), \textit{ for some finite } V, Y \subset \mathbb{R}^d$$

if and only if it is a finite intersection of halfspaces (an H-polytope*)*

$$P = \{x \in \mathbb{R}^d : Ax = z, \textit{ for some } A \in \mathbb{R}^{m \times d}, z \in \mathbb{R}^m\}.$$

And the above, once we set the "bounded" condition, is the Polytope version.

2.3.2 Recession Cones for Convex Sets

Given a convex set $A \subset V$ where V is an inner product space (in our case, $V = \mathbb{R}^d$). A vector y is a *direction of recession* if starting at any x in A and going indefinitely along y, we never cross the relative boundary of A to points outside A. Formally speaking:

Definition 2.13 (Recession Cone of a Convex Set)**.** Given $A \subset V$ is convex, then

the *recession cone* of A is defined as:

$$\text{rec}(A) = \{y \in V : x + \lambda y \in A \text{ for every } x \in A, \lambda \geq 0\}.$$

Note that $\vec{0}$ is always in $\text{rec}(A)$, and that for a bounded set A, the recession cone $\text{rec}(A)$ is just the zero vector.

Similarly, a vector y is in the *linearity space* of the convex set $A \subset V$ if starting at any $x \in A$ and going indefinitely along y in BOTH directions, we can never cross the relative boundary of A to points outside of A. Formally:

Definition 2.14 (Linearity Space of a Convex Set)**.** Given $A \subset V$ is convex, then the *linearity space* of A is defined as

$$\text{lineal}(A) = \{y \in V : x + \lambda y \in A \text{ for every } x \in A, \lambda \in \mathbb{R}\}.$$

Note that the linearity space is a vector subspace of $\mathbb{R}^d$ in its own right, and the set A is a strict subset of $\mathbb{R}^d$ if and only if $\dim(\text{lineal}(A)) < d$. If we construct a vector subspace W such that $\{\vec{0}\} = W \cap \text{lineal}(A)$ and $W + \text{lineal}(A) = \mathbb{R}^d$ by choosing basis elements that are linearly independent from all basis elements of $\text{lineal}(A)$ (from linear algebra, $\dim W = d - \dim(\text{lineal}(A))$) and then using this complement vector subspace W, we can decompose a convex set with non-

empty linearity space into two parts by a Minkowski sum:

$$P = \text{lineal}(P) + (P \cap W)$$

where the linearity space of $(P \cap W)$ is zero. In other words, $(P \cap W)$ is a convex set that does not contain any lines.

Definition 2.15 (Pointed polyhedra)**.** A polyhedra P is pointed if $\text{lineal}(P) = \{0\}$; in other words, a polyhedra is pointed if it does not contain any lines.

Another perspective: a polyhedra P is pointed whenever there exists some affine halfspace that strictly contains P.

Chapter 3

Lattice Polytopes and Rational Cones

3.1 Lattice Polytopes

Definition 3.1 (Lattice in $\mathbb{R}^d$)**.** For any basis B of $\mathbb{R}^d$, the additive subgroup of all integer linear combinations of B forms a lattice.

We're interested in $\mathbb{Z}^d$, which is formed using the standard unit vectors of $\mathbb{R}^d$, and is the set of vectors with all integer entries.

Definition 3.2 (Lattice Polytope)**.** A lattice polytope is a polytope whose vertices are in $\mathbb{Z}^d$.

Often, we're interested in the lattice elements contained by a polytope P, which we denote as $\mathrm{L}(P) = \mathbb{Z}^d \cap P$.

3.1.1 Normal Polytopes

We shall now build the definition for normal polytope, which are important objects in combinatorial commutative algebra and toric algebraic geometry. The study of these objects in a fixed dimension d, denoted $\mathrm{NPol}(d)$ as a partially ordered set is the direct motivation leading to the cone conjecture.

Definition 3.3 (Normal polytope)**.** A lattice polytope P is *normal* if it satisfies the following condition:

$$n \in \mathbb{Z}_+, z \in \mathrm{L}(nP) \implies \exists x_1, \dots x_n \in \mathrm{L}(P) \text{ such that } x_1 + \cdots + x_n = z.$$

$\mathbb{Z}_+$ here denote the positive integers.

In other words, a lattice polytope P is normal if the lattice elements of the n-th dilate of P is the Minkowski sum of n copies of the lattice points of P.

Remark. In some texts, where a lattice polytope is defined using an affine lattice instead of $\mathbb{Z}^d$, there is a technical distinction between *integrally closed* and *normal*. Specifically, a polytope P is integrally closed if and only if P is normal and the affine lattice of P is a direct summand of $\mathbb{Z}^d$ [10].

Normal polytopes have been an object of study since the 1990's. A precursor to this is Pick's theorem from the 19th century which, when reworded in the modern and generalized language of normal polytopes, implies that all lattice polygons are normal. However, these objects and their poset structure are highly

sensitive to a change in dimension. In higher dimensions, starting with 3, the normal polytopes form a small portion of all lattice polytopes.

3.1.2 Partial Order on Normal Polytopes

Motivated by physics, one measures the smallest possible changes as analogs of "potential" of the jump [10]. Furthermore, the set of normal polytopes of the same dimension d form a partial order, and it is explicitly defined in [4]. We give a concise exposition of this partially ordered set (poset) here. First, a pair (P, Q) of normal polytopes of equal dimension is called a *quantum jump* if $P \subset Q$ and Q has exactly one more lattice point than P.

Definition 3.4 (Quantum Jump)**.** Let $P, Q \subset \mathbb{R}^d$ be normal polytopes. Then (P, Q) is a quantum jump when $P \subset Q$ and $\#\mathsf{L}(P) + 1 = \#\mathsf{L}(Q)$.

If we fix the ambient dimension, and we consider the set of full dimensional normal polytopes, we can form a partially ordered set. A partially ordered set (or poset) is a set taken together with a partial order on it.

Definition 3.5 (Partial Order, Partially Ordered Set (Poset))**.** A *relation* "$\leq$" is a partial order [15] on a set P if it satisfies:

1. Reflexivity: $a \leq a \; \forall a \in P$,

2. Antisymmetry: $a \leq b \wedge b \leq a \implies a = b$

3. Transivity: $a \leq b \wedge b \leq c \implies a \leq c$.

Formally, a partially ordered set is defined as an ordered pair $P = (X, \leq)$, where X is called the ground set of P and $\leq$ is the partial order of P [13].

Armed with these definitions of quantum jumps and partial order, we can define a partial order on the set of full dimensional normal polytopes in $\mathbb{R}^d$, denoted $\mathsf{NPol}(d)$.

Definition 3.6 (Partially Ordered Set of $\mathsf{NPol}(d)$)**.** Let P, Q be normal polytopes, then $P < Q$ if and only if there exists a *finite* sequence of normal polytopes of the form

$$P = P_0 \subset \cdots P_{n-1} \subset P_n = Q \tag{3.1}$$

$$\text{such that } \#\mathsf{L}(P_i) = \#\mathsf{L}(P_{i-1}) + 1, \text{ for } i = 1, \ldots, n. \tag{3.2}$$

We may consider the relation $<$ as the discrete analogue of the set theoretic inclusion between convex compact subsets of $\mathbb{R}^d$.

According to [4], in the late 1980's, there were two conjectures that aimed to give a clear and succinct characterization of the "normal point configurations". Given that P is some lattice polytope in $\mathbb{R}^d$, *Unimodular Cover (UC)* was a conjecture which stated that P is normal if and only if P is the union of unimodular simplices (polytopes whose vertices are an affine basis of $\mathbb{Z}^d$), and *Integral Carathéodory Property (ICP)* was a conjecture which stated that P is

normal if and only if for an arbitrary natural number $c \in \mathbb{N}$ and an arbitrary integer point $z \in \mathsf{L}(cP)$ there exist $x_1, \ldots, x_{d+1} \in \mathsf{L}(P)$ and positive integers $a_1, \ldots, a_{d+1} \in \mathbb{Z}_+$ such that $z = \sum_{i=1}^{d+1} a_i x_i$ and $\sum_{i=1}^{d+1} a_i = c$. By examining $\mathsf{NPol}(d)$, in particular the *minimal elements*, called *tight polytopes*, counter examples to (UC) and (ICP) were found in [2, 5].

In particular, we're interested in the *maximal* elements of the partially ordered set $\mathsf{NPol}(d)$. Given some poset $(P, \leq)$, an element $x \in P$ is maximal when there are no elements in the poset greater than x. Similarly, an element $y \in P$ is *minimal* when $\nexists\, x \in P$ such that $x < y$.

While the definition of the poset order is simple to write down, the study of the structure of this partially ordered set is difficult, with the existence of maximal elements in this poset in dimension ≥ 4 [11]. By direct searching using random walks, one can find explicit examples of maximal elements in $\mathsf{NPol}(4)$. Furthermore, the existence of *tight* polytopes, or non-trivial minimal elements has been known for all dimensions ≥ 4. The existence of nontrivial minimal elements in $\mathsf{NPol}(3)$ is open, and similarly, the existence of maximal elements in $\mathsf{NPol}(4)$ is also unknown [4].

Recent research has shown that (ICP) is strictly weaker than (UC), and the strategy of shrinking normal polytope was used, and chances are that the weaker property (ICP) is retained longer than the stronger condition (UC), which is lost earlier in this procedure. This is an indication that the poset $\mathsf{NPol}(d)$ is important

in understanding the normality property [1, 2, 5].

In dimensions $d \geq 4$, finding non-trivial minimal elements is relatively simple, since given any $P \in \mathsf{NPol}(d)$ and any $Q \in \mathsf{NPol}(e)$, their *product polytope* is again a minimal element in $\mathsf{NPol}(d+e)$; but finding maximal elements is computationally challenging, and as of the year 2016, there are only a handful of maximal normal polytopes found in dimension 4 and 5 [4].

3.2 Rational Cones

The difficulty in studying $\mathsf{NPol}(d)$ motivated a study on the set of rational cones, which form a partially ordered set in their own right. In fact, the partial order structure of the set of pointed rational cones preserves the partial order of normal polytopes. In particular, the poset of normal polytopes of dimension $d-1$ embed into the poset of cones of dimension d via the so-called homogenization map, and the characterization of this larger poset approximates the partial order in $\mathsf{NPol}(d)$.

3.2.1 Pointed Rational Polyhedral Cones

We are interested in a subset of polyhedra: the set of full-dimensional pointed, rational, polyhedral cones in $\mathbb{R}^d$, denoted as $\mathrm{Cone}(d)$. The reason we are interested in pointed polyhedral rational cones arises from their connection to

the normal polytopes. In particular, $\mathsf{NPol}(d-1)$ embeds into $\mathrm{Cone}(d)$ via homogenization map, as stated in section 2.3.1. Moreover, the poset structure in $\mathsf{NPol}(d-1)$ is actually preserved in $\mathrm{Cone}(d)$, once we define a partial order on $\mathrm{Cone}(d)$.

We shall parse the definition of this object by each term.

Definition 3.7 (Pointed cone)**.** We call a cone *pointed* whenever there is no nonzero element $c \in C$ with $-c \in C$.

This may be characterized by the language in the section on polytopes; cones in this text are all polyhedral, so a pointed cone is a cone whose linearity space is $\{0\}$. Another characterization of this type of cone is that there exists an affine hyperplane H such that $C \cap H$ is a polytope of dimension $\dim(C) - 1$ [3].

Definition 3.8 (Rational cone)**.** A cone C is called rational when

$$C = \{a_1 x_1 + \cdots + a_d x_d : a_1, \ldots, a_d \in \mathbb{R}_+\} \text{ for some } x_1, \ldots, x_d \in L$$

where $L \subseteq \mathbb{Q}^d$ is a lattice in $\mathbb{R}^d$. In our particular case, we are interested when $L = \mathbb{Z}^d$.

Henceforth in this thesis, the word *cones* shall contain all of the above assumptions. Since polyhedral cones can be described as the conical hull of finite set of vectors, we give a name for this set of vectors: the first nonzero lattice points of the edges of C are called the *extremal generators* of C, written as

ExtGen(C). This condition leads us to give the definition of a primitive vector. A vector v is primitive if it is the generator of the monoid $\mathsf{L}(\mathbb{R}_+ v)$, which by definition of primitive vector is equivalent to a vector in $\mathbb{Z}^d$ whose entries are all coprime.

3.2.2 Hilbert Basis

For any rational cone C, the lattice elements $\mathsf{L}(C)$ are closed under addition and must contain the additive identity $\vec{0}$. Given some pointed rational cone $C \subset \mathbb{R}^d$, the lattice elements $\mathsf{L}(C)$ form an algebraic structure called a monoid under vector addition. Recall that a monoid is a set that is closed under a binary operation with a unit, and it is a set with a structure that generalizes groups by relaxing the definition so that the binary operation does not need to be closed under inverses.

By Gordan's lemma, this monoid is finitely generated. When a cone is also pointed, the set of generators are finite and unique. With these properties, one can obtain a set of finitely many irreducible elements that is a complete system of generators, which must be contained by every other system of generators. The proof for the existence and uniqueness of this set of minimal and unique generators is found in [3, 9].

Definition 3.9 (Hilbert Basis of Cone C)**.** Given a pointed rational cone C, The *Hilbert basis* of $\mathrm{L}(C)$ is a minimal set of vectors in $\mathbb{Z}^d$ such that every integer

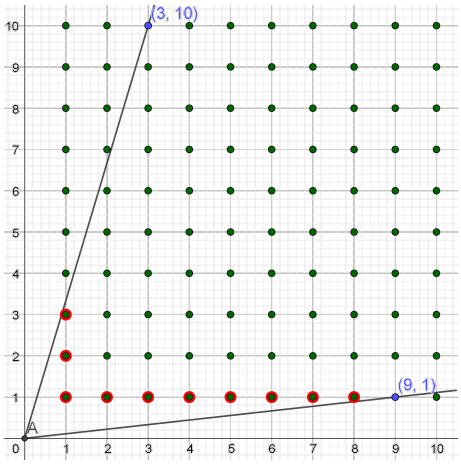


Figure 3.1: Indecomposable elements of the lattice of a cone in $\mathbb{R}^2$.

vector in C is an integer conical combination of the vectors in the Hilbert basis with integer coefficients [8].

In particular, note that $\text{ExtGen}(C) \subseteq \text{Hilb}(C)$, as shown in **Figure 3.1**.

When studying the lattice structure of the cone, loosely speaking the Hilbert basis can help us determine the "complexity" of the structure.

Now that we are equipped with the definition of a Hilbert basis, we can tie the connection between the normality of a lattice polytope $P \subset \mathbb{R}^{d-1}$ and $C(P) \subset \mathbb{R}^d$:

Remark. $P \subset \mathbb{R}^{d-1}$ is normal if and only if the Hilbert basis of the cone associated with P is exactly the lattice points embedded in the layer $x_0 = 1$, i.e., $\text{Hilb}(C(P)) = \{(1, x) | x \in L(P)\}$

3.3 The Partially Ordered Set of Rational Cones

Given a fixed dimension d, then the set of all cones in $\mathbb{R}^d$ form a partially ordered set [11]. An important and interesting aspect of rational cones is to study the lattice elements of the cone.

Definition 3.10 (The Partially Ordered set of Pointed Rational Cones in $\mathbb{R}^d$, $\mathrm{Cone}(d)$)**.** The set of rational cones in $\mathbb{R}^d$, denoted $\mathrm{Cone}(d)$, forms a poset, where given $C, D \in \mathrm{Cone}(d)$, then we order $C < D$ if and only if there exists a sequence of cones $C_0, \ldots C_n$ such that

$$C = C_0 \subset \cdots \subset C_{n-1} \subset C_n = D, \tag{3.3}$$

$$L(C_i) = L(C_{i-1}) + \mathbb{Z}_+ x \text{ with } x \in C_i \backslash C_{i-1}, \quad i = 1, \ldots, n. \tag{3.4}$$

If $n = 1$, we call $C \subset D$ an *elementary extension*, or *elementary descend* if read backwards.

In particular, the order can be imposed on cones C and D whenever they form a finite chain of cones that satisfy containment; moreover, the lattice of each pair of consecutive cones C_i and C_{i+1} have the property that the outer cone's lattice submonoid, $L(C_{i+1})$, is the Minkowski sum of the inner cone's lattice with one and only one vector external to the inner cone.

While the partial order of $\mathsf{NPol}(x)$ is notoriously difficult to analyze, the partial order in Cones(d) is conjectured to be the inclusion order, which also has sig-

nificant implication of the topology of the geometric realization of this partially ordered set $\mathsf{Cones}(d)$ [11]. In recent mathematics research, the set of rational cones are important objects in toric algebraic geometry, combinatorial commutative algebra, geometric combinatorics and integer programming [11, 3].

The Hilbert bases of cones are difficult to characterize; as such, general results are currently available only in lower dimensions, with counter-examples to conjectures to inform our journey to understanding these objects in higher dimensions.

Gubeladze and Michałek proved that the poset order is the inclusion order in dimension 3 (Theorem 3.2 in [11]).

Currently, the state-of-the-art research on normal polytopes involves examining unimodular triangulations [12]. Analogous to this is the process of triangulating cones into *simplicial* cones. In order to discuss some preliminary standard results on cones, we recall the definition of a few terms:

- A cone C is called *simplicial* if $\mathsf{ExtGen}(C)$ are linearly independent.
- A cone $C \subset \mathbb{R}^d$ is called unimodular if $\mathsf{Hilb}(C)$ is a part of a basis of $\mathbb{Z}^d$.
- A triangulation of a cone C into simplicial cones is called *unimodular* if the cones in the triangulation are unimodular.
- A triangulation of a cone C is called *Hilbert* if the set of extremal generators of the involved cones equals $\mathsf{Hilb}(C)$.

The following is proved in [3], the last item was rediscovered from the perspective of toric geometry in 1994-1995 (details can be found in the bibliography of [11]).

Lemma 3.1 (Standard results on cones)**.**

(a) Let $C \subset \mathbb{R}^d$ be a cone and $v \in \mathsf{L}(C)$ be a nonzero element on an edge of C. Then $\mathsf{L}(C) + \mathbb{Z}v = \mathsf{L}(C_0) + \mathbb{Z}v \cong \mathsf{L}(C_0) \times \mathbb{Z}v$ for some $C_0 \subset \mathbb{R}^d$ with $v \notin C_0$.

(b) Let $C \subset \mathbb{R}^d$ be a nonzero cone and $w \in \mathsf{L}(C)$ be an element in the relative interior of C. Then

$$\mathsf{L}(C) + \mathbb{Z}w = \mathsf{L}(RC).$$

(c) Every nonzero cone has a unimodular triangulation.

(d) Every 2-cone has a unique Hilbert triangulation that is unimodular.

(e) Every 3-dimensional cone has a unimodular Hilbert triangulation.

With these standard results, one can give an alternate formulation of the poset condition in Cones(d), and the proof using the remark above is again in [11] .

Lemma 3.2 (Alternative Characterization of Elementary Extension)**.** *Let $C \subset \mathbb{R}^d$ be a nonzero cone and $v \in \mathbb{Z}^d$ be a primitive vector with $\pm v \notin C$. (This guarantees that $C + \mathbb{R}_+ v$ is pointed). Assume $H \subset \mathbb{R}^d \backslash \{0\}$ is an affine hyperplane, meeting the cone*

$D = C + \mathbb{R}_+ v$ transversally. Let $v' = R_+ v \cap H$. (Obtain v' by scaling v so that v' is on H.) Then $C \subset D$ is an elementary extension in $\mathrm{Cones}(d)$ *if and only if there exists unimodular cones $U_1, \ldots, U_n \subset D$ that satisfy the following conditions:*

(i) $v \in U_i, i = 1, \ldots, n$

(ii) $D = C \bigcup \left(\bigcup_{i=1}^{n} U_i \right)$

(iii) $\left\{ \mathbb{R}_+((U_i \cap H) - v') \right\}_{i=1}^{n}$ *is a triangulation of the cone* $\mathbb{R}_+((D \cap H) - v')$.

In $\mathrm{Cone}(d)$, there is an important subposet that connects cones to normal polytopes: The set of cones in $(\mathbb{R}^{d-1} \times \mathbb{R}_{>0}) \cup \{0\}$ is denoted $\mathrm{Cone}^+(d)$. Note that the homogenization map sends $\mathsf{NPol}(d-1)$ into Cone^+.

3.3.1 Relationship between Normal Polytopes and Rational Cones

The partial order on the set of normal polytopes is preserved; i.e., the homogenization map (equations 2.1 and 2.2, which are non-trivially equivalent) is monotonic with respect to the partial order. This means that $P \subset Q \in \mathrm{NPol}(d-1)$, then $C(P) \subset C(Q) \in \mathrm{Cones}^+(d)$ and $(P < Q) \implies C(P) < C(Q)$. However, the converse is not necessarily true. In the following example, found in [4]:

Example 3.1. Suppose

$$P = \mathrm{conv}(\{(0,0,2), (0,0,1), (0,1,3), (1,0,0), (2,1,2), (1,2,1)\}) \in \mathsf{NPol}(3).$$

Then removing the first or the second vertex yields a non-normal polytope, yet

$$Q = \text{conv}(\{0,1,3),(1,0,0),(2,1,2),(1,2,1)\}) \subset P \in \mathsf{NPol}(3).$$

Thus, this pair of normal polytopes $P \subset Q$ in $\mathsf{NPol}(3)$ does not satisfy the poset condition, i.e. $Q \not< P$, but when considering their images in $\text{Cone}^{+}(d)$, namely, $C(Q) \subset C(P)$, one can find a finite chain that that connects the two in the poset of cones:

$$\begin{aligned} C(P) &= \text{cone}\{(0,0,1,1),(0,0,2,1),(0,1,3,1),(1,0,0,1),(1,2,1,1),(2,1,2,1)\}) \\ C(Q) &= \text{cone}(\{(0,1,3,1),(1,0,0,1),(1,2,1,1),(2,1,2,1)\}). \end{aligned}$$

Then

$$\begin{aligned} C(P) = D_0 &= \text{cone}(\{(0,0,1,1),(0,0,2,1),(0,1,3,1),(1,0,0,1),(2,1,2,1),(1,2,1,1)\}) \\ D_1 &= \text{cone}(\{(0,0,2,1),(0,1,3,1),(1,0,0,1),(1,2,1,1),(2,1,2,1)\}) \\ D_2 &= \text{cone}(\{(0,1,3,1),(1,0,0,1),(2,1,3,2),(2,1,2,1),(1,2,1,1)\}) \\ D_3 &= \text{cone}(\{(0,1,3,1),(1,0,0,1),(2,1,2,1),(1,2,1,1)\}) = C(Q). \end{aligned}$$

The above was found using the Hilbert descend, or the "top down" method. Note that this required Hilbert basis elements at height two; so the polytopes

formed by $\left\{ x \in \mathbb{R}^d : \begin{bmatrix} 1 \\ x \end{bmatrix} \in C_i \right\}$ are not normal.

3.3.2 Cone Conjecture

The cone conjecture in [11] states that for every d, the order in Cones(d) is the inclusion order.

This conjecture is already proven in dimension ≤ 3. For dimension 4, Andreas Paffenholz had implemented height-1 extensions and Hilbert basis descends based on many randomly generated cones C and vector v, with $\pm v \notin C$, which support the hypothesis that there are no non-terminating processes of either type.

The research presented in this thesis aims to refine this computational experimentation, and to generalize the implementations to any $C \subset D \in \mathbb{R}^d$ with any $d \geq 2$, in particular with a focus on dimension 5.

3.3.3 Height 1 Extensions

Cones(d) contains many elementary extensions of two different types, which makes this poset essentially different from $\mathsf{NPol}(d-1)$.

Let $C \subset \mathbb{R}^d$ be a full dimensional cone and have $v \in \mathbb{Z}^d$ on the exterior of C with $\pm v \notin C$. We then denote the set $\mathbb{F}^+(v)$ as the set of facets visible from v. Each support hyperplane of the facets of C has a linear form $ht_F(v)$ where

$ht_F(C) \geq 0$. From the perspective of any point that is on the "other side" of this support hyperplane ($ht_F(v) < 0$) would consider this facet visible.

We collect the visible parts of the boundary ∂C, and we'll label this set

$$C^+(v) = \bigcup_{F \in \mathbb{F}^+(v)} F,$$

i.e., take the union of all the facets visible to v. We then take the conical hull of C and v to form a new cone $D = C + \mathbb{R}_+ v$. By this construction, there exists an increasing sequence of rational numbers

$$0 < \lambda_1 < \lambda_2 < \cdots$$

such that

$$\lambda_1 = \frac{1}{\max(-\mathrm{ht}_F(v))} \quad \text{for } F \in \mathbb{F}^+(v)$$

and $\lim_{k\to\infty} \lambda_k = \infty$, satisfying the following conditions:

$$\mathsf{L}(D \backslash C) = \bigcup_{k=1}^{\infty} \mathsf{L}(\lambda_k v + C^+(v))$$

$$\mathsf{L}(\lambda_k v + C^+(v)) \neq \emptyset \quad \text{for } k = 1, 2, \ldots$$

When $\lambda_1 = 1$, or equivalently, $\mathrm{ht}_F(v) = -1$ for all $F \in \mathbb{F}^+(v)$. In this special case, we say D is a *height 1 extension* of C. While all height 1 extensions are

elementary extensions, the converse is not true [4].

Lemma 3.3. *The* Bottom Up *algorithm is an implementation of height-1 extensions, which generate cones* $C = C_0 \subset C_1 \subset C_2 \subset \cdots$ *such that each pair* $C_n \subset C_{n+1}$ *satisfy the poset condition* ***3.4****.*

3.3.4 Hilbert Bases Descends

Given some cone D_i we can generate a cone D_{i+1} such that $D_{i+1} \subset D_i$ by taking the Hilbert basis of D (the original outer cone), and removing one extremal vector from this list. We then take the conical hull of the remaining vectors, giving us a $D_{i+1} \subset D$, where the lattice elements of D_i by construction is exactly $\mathsf{Hilb}(D_{i+1}) + v$ for some v not in D_{i+1}. This can be applied in such a way where given any arbitrary full dimensional lattice cone $C \subset D$.

Lemma 3.4. *The Top Down algorithm generates cones* $D = D_0 \supset D_1 \supset D_2 \supset \cdots$ *such that each pair* $D_{n+1} \subset D_n$ *satisfy the poset condition* ***3.4****.*

As an application, these two types of extensions give us:

Lemma 3.5. $\mathrm{Cones}(d)$ *has neither maximal nor minimal elements, other than the minimal element 0.*

Proof. Given some cone $C \in \mathrm{Cones}(d)$, apply a Hilbert descend to find a cone that is lower on a poset, and apply a height-1 extension to obtain a cone higher on a poset chain. ❖

What does it mean that we do not know whether 0 is the smallest element? This means that while we know the 0 cone is contained in all other cones, we do not know if given any $C \supset 0$ we can find a finite sequence of cones so that $0 < C$ based on **3.4**.

A question was asked by Gubeladze and Michałek in [11] on the elementary extensions: "Do either height 1 extensions or Hilbert basis descends generate the same poset Cones(d)?"

This above question is considered in five different ways in this paper, and in particular we focus on questions (3) and (4):

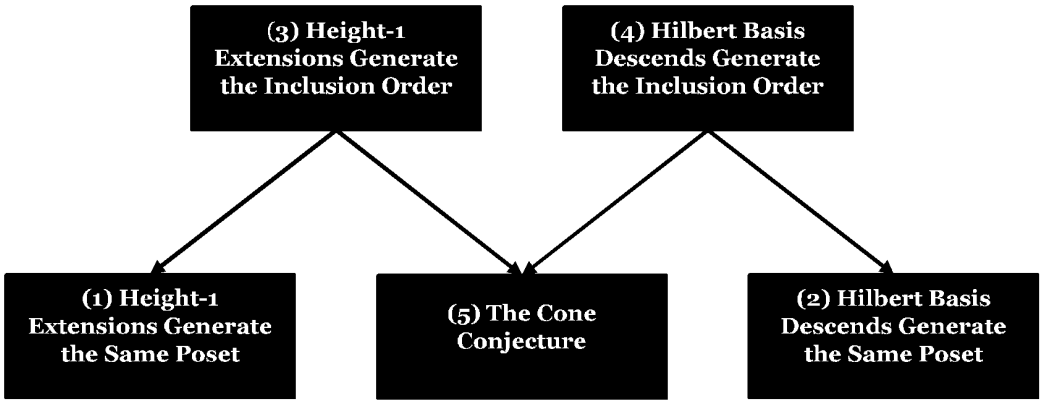


Figure 3.2: Logical implications of the questions.

(1) Do Height-1 extensions generate Cones(d)? This is a separate statement from the cone conjecture, since not all elementary extensions are of the form

of Height-1 extensions.

(2) Do Hilbert descends generate Cones(d)?

(3) Do Height-1 extensions generate inclusion order? This is a stronger statement than the cone conjecture and would imply (1) if true. If this is true, then for any $C \subset D$ one can generate a finite chain $C = C_0 < C_1 < \ldots < C_n = D$ using height 1 extensions.

(4) Do Hilbert descends generate the inclusion order?

(5) The cone conjecture.

The positive answer towards questions (3) and (4) above implies the cone conjecture, because they are both logically stronger statements. Furthermore, they imply questions (1) and (2) as well, respectively.

The proof given in dimension 3 shows that the answer to question 5 is true in [11]; but it does not show 1 - 4. Our work provides statistical evidence in dimension 4 and 5 suggesting that the answer to questions 3 and 4 is no.

Chapter 4

Experimental Procedure

Since we currently do not have a theoretical approach to the cone conjecture aside from a direct proof in dimension 3, we used a computational approach for dimension 4 and 5 in the hopes for further insight. Below we present the details on the implementation of the two elementary extensions in higher dimensions.

4.1 Experimental Method

The exhaustive study of cones by computational means is challenging, even with severe restrictions on the number of extremal generators in $\mathbb{Z}^d$ to be some number close to d. Since we are studying full dimensional cones, all cones C must have $\#\mathrm{Ext}(C) \geq d$, with further restrictions on the coordinates, the problem quickly becomes intractable.

While we were not able to attempt an exhaustive search, the combinatorics

question itself is interesting:

Question. How many full dimensional cones can we form in $\mathbb{Z}^d$ with $n \geq d$ vectors, where each extremal generator is in $\{\mathbf{x} \in \mathbb{Z}^d : 0 \leq x_i \leq k\}$?

Some work towards this: We have an upper bound, since this region has $(k+1)^d - 1$ points, we have at most $\binom{(k+1)^d-1}{n}$ choices, with an *unknown* probability for the choices that are linearly independent.

For our purposes, we would not only need the number of full dimensional cones, but also list these vectors explicitly; thus, we arranged the vectors as columns of a matrix, and attempt to find its determinate to check for linear independence. However, physically running this naïve algorithm will lead to astronomical numbers of steps, even for $d = 5$, $n = 5$ and $k = 2$, we would have to check $\binom{3^5}{5} = \binom{242}{5} \approx 6.63 \times 10^9$ matrices. After collecting the linearly independent combinations, one must then find combinations that satisfy containment, and after that we decided that this path is physically impossible to do within the time constraints for a master's thesis.

Thus, we utilize randomly generated pairs of full dimensional cones in order to examine the poset of cones. While most of our experiments use random cones, we have also included a simple text UI that allows for proper input of a list of extremal generators for a pair of cones, and the associated logical verification to ensure we satisfy the conditions associated with the cone conjecture.

Furthermore, we give a full description of the two algorithms described be-

low, and test whether the algorithms terminate. If the implementation show evidence that there are no non-terminating processes of either type, then this also gives strong evidence that the conjecture is true. Whenever the algorithms terminate, they give a finite sequence of cones from C to D that satisfy stricter conditions than the order statement.

4.1.1 Technical Requirements

The implementation of the experiment uses the development branch of `SAGE` [14], and the PyNormaliz package to link `Normaliz` [6]. The experiment was written in the standard python format. Finally, the experiment was run on the operating system `Ubuntu 16.04 LTS`.

A current version of the code for this experiment is hosted on `GitHub` [7].

4.2 Algorithms and Procedures

4.2.1 Terminal User Interface vs. Python Scripts

As the way this experiment is designed, one may run experiments in batch by writing a script in Python, or use the built in terminal user interface. While simple, this terminal user interface was built from scratch using dictionaries and UI functions built over time.

Using `JSON` file formats, the code is written so that a user may create a new

experiment or load an incomplete experiment, and an option to copy an experiment either its current state or just the initial conditions.

In the creation of a new experiment, a user must first choose the dimension, and then set boundaries for the generation of cones or manually input vectors. The manual input part also does a simple sanity check, where it verifies:

1. Vectors are of the form $(x_1, ..., x_d) : x_i \in \mathbb{Z}$ for $i = 1, ..., d$;
2. The outer cone is full dimensional and pointed;
3. Each vector of the inner cone must be contained by the outer cone;
4. The inner cone must be full dimensional (if the outer cone is pointed then the inner cone is automatically pointed).

4.2.2 Randomly Generating Cones

First, we generate random vectors, by using the built-in `SAGE` random integer generators. In particular, we generate vectors in $\mathbb{Z}^d$ of the form $\{x_1, \ldots, x_d\}$ where $|x_i| \leq k$ for some sensible k and $x_d > 0$, which forces our cones to be within one halfspace. In doing so, we do not lose generality, as by a simple affine transformation we can then study cones that are limited by other halfspaces.

For any experiment, the first step (if not by user input) is to initiate our data structures by randomly generating $C, D \in \text{Cones}(d)$ where $C \subset D$ and satisfy the above conditions.

1. Retrieve a particular dimension d, and the user provides number of n generators, either by a terminal user interface or by a constructor in a separate python script.

2. Unless the user specifies the cones, we generate vectors $\mathbf{c}_i \in \mathbb{Z}^d, i = 1, \ldots, n$, with a user chosen n such that $n \geq d$ (so that we have a full-dimensional cone). We force each of the vectors $\mathbf{c} \in \{x \in \mathbb{Z}^d : x_d > 0\}$.

3. We then take the conical hull of these randomly generated vectors, and then define this as our outer cone $D = \text{Cone}(\{c_1, \ldots, c_n\})$, and repeat until the cone is full dimensional and we have n rays. (Since some vectors may be linearly dependent, as n grows large above d this process of "organically" generating a cone may slow down.)

4. Next, we generate vectors randomly and check that each is contained in the outer cone D. Once we have n vectors of this form, we take the conical hull. Using `SAGE`, we verify the cone must be full dimensional first.

These steps ensure that C and D are both pointed cones, with $C \subset D$.

4.2.3 Top Down

This algorithm uses Hilbert Descents to "shave" D down to C.

1. Let $D_0 = D$, i.e., we set the initial cone to be the outer cone D.

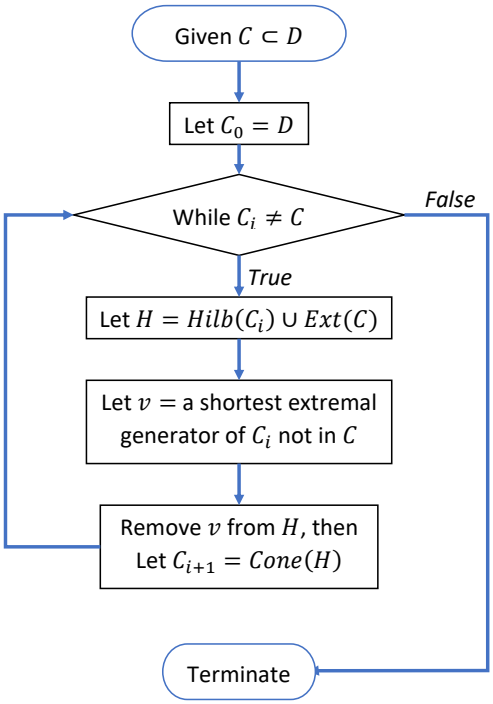


Figure 4.1: Flow chart for the *Top Down* algorithm.

2. Let $H = \text{Hilb}(D_0)$, i.e., we collect the indecomposable elements of $L(D_0)$.

3. Let $D_1 = \text{Cone}(H \backslash \{v\})$, i.e., we take the conical hull of all the indecomposable elements of $L(D_0)$ after removing v from the list.

4. While $D_i \neq C$,

   (a) Let H be a list containing the Hilbert basis of D_i.

   (b) Collect the set extremal generators of D that are not in C and collect

these vectors into a list $E_i = \text{ExtGen}(D_i) \cap C^c$.

(c) Loop through E_i and record the lengths of each vector. Then choose one of the vectors with the smallest Euclidean norm from this list, call this v_i.

(d) Return $D_{i+1} = \text{Cone}(\text{Hilb}(D_i) \backslash \{v_i\} \cup C)$, i.e., remove v_i from the list H, and return the conical hull of $H \backslash \{v_i\}$.

4.2.4 Bottom Up

We grow the inner cone to the outer cone by height-1 extensions. Previous results have shown that this method produces a cone more quickly than the *Top Down* algorithm.

In order to describe the algorithm, we define a special type of polyhedron called a *zonotope*:

Definition 4.1 (Zonotope)**.** A zonotope is a set of points in d-dimensional space constructed from vectors v_i by taking the sum of $a_i v_i$, where each a_i is a scalar between 0 and 1. Given $V = \{v_1, \ldots, v_k\}$:

$$\text{Zonotope(V)} = \{x \in \mathbb{R}^d : x = a_1 v_1 + \cdots + a_k v_k \text{ and } 0 \leq a_i \leq 1 \text{ for each } a_i\}.$$

1. Let $C_0 = C$.

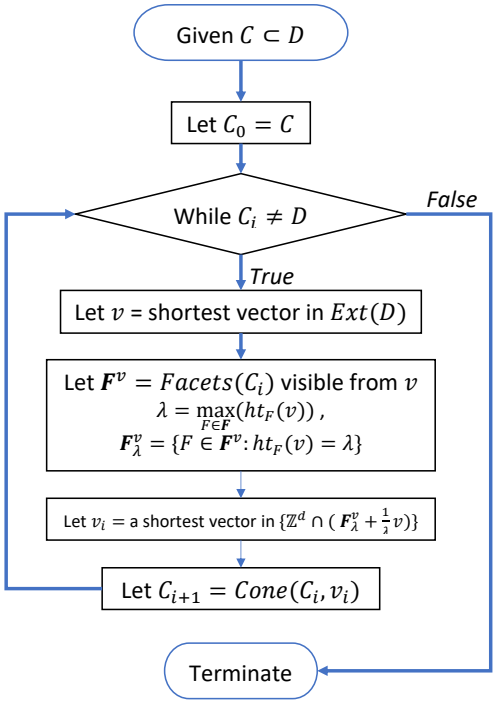


Figure 4.2: Flow chart for the *Bottom Up* algorithm.

2. While $C_i \neq D$:

   (a) Collect the set of facets of C_i visible from v:

      i. Let $\mathbb{F}^v$ be the set of facets of C visible from v, i.e., H_α is a support hyperplane of C associated with a facet F.

      ii. F is visible from v if $C \subset H^+_\alpha \implies v \subset (H^+_\alpha)^c$. Computationally, we can simply check that C and v have opposite signs when evaluated using α; i.e., if $\alpha(c) < 0$ for every $c \in C$, then F is visible

from v if $\alpha(v) > 0$.

(b) Let $ht_F(v) = \alpha(v)$ given that α is the linear form associated with the support hyperplane of the facet F.

(c) Let $\lambda = \max_{F \in \mathbb{F}}(ht_F(v))$. Collect the set $\mathbb{F}^v_\lambda = \{F \in \mathbb{F}^v : ht_F(v) = \lambda\}$.

(d) Shift this set $\mathbb{F}^v_\lambda$ along $\frac{1}{\lambda}v$, and find the shortest integer lattice point. Looping through each shifted facet F in $\mathbb{F}^v_\lambda$:

- Computationally, we take the extremal generators of each facet F and from a zonotope Z_F. Shift Z_F by $\frac{1}{\lambda}v$.
- Once we have $Z_F + \frac{1}{\lambda}v$, collect the vectors $L(Z_F + \frac{1}{\lambda}v)$ and append this to a list E_i.

  Remark. The zonotope $Z_F + \frac{1}{\lambda}v$ always contains at least one lattice element.

(e) Loop through E_i and find the shortest vector, called this v_i.

(f) Return $C_{i+1} = \text{Cone}(C \cup v_i)$.

The two algorithms can be used alternatively, and we examine the difference of the two algorithms by comparing the number of elements of the Hilbert basis of each successive cone.

Lemma 4.1. *The* Top Down *and* Bottom Up *algorithms yield the same chain of cones in* $\mathbb{R}^2$.

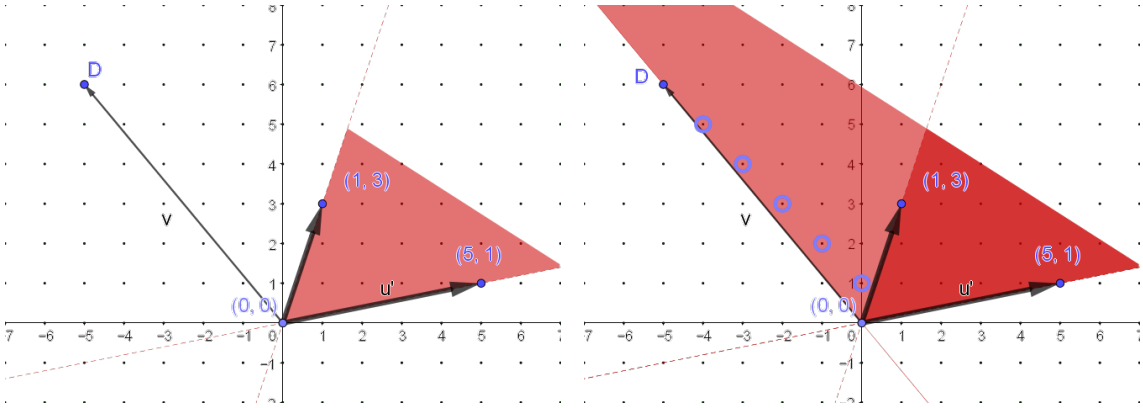


Figure 4.3: Demonstration of the *Top Down* and *Bottom Up* algorithms in $\mathbb{R}^2$.

Yet, for $d > 2$, in $\mathbb{R}^d$ the algorithms do not always generate the same cones, demonstrated in the data section below.

4.3 Objects and Data Structure Overview

The object oriented version currently existing on `Github` allows for advantages over the procedural version in several ways. In particular, the Hilbert basis calculation for each cone used in the Top Down algorithm is computationally expensive, and the most elementary data structure in this design was created specifically to avoid doing multiple calculations on the same cone.

Three central objects were created for this experiment, and here we present the essential attributes and methods of the code. The reader may find the source code in **Chapter 5**.

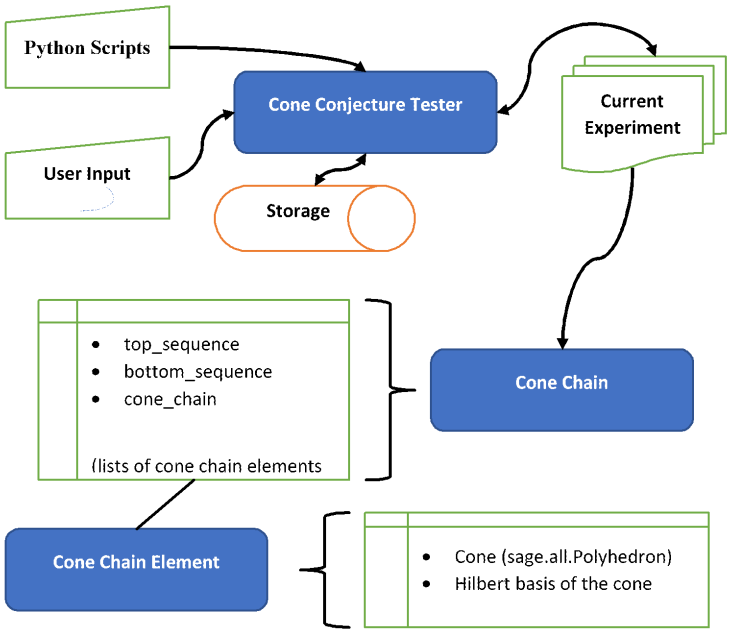


Figure 4.4: Visualization of Object-Oriented Design.

1. `cone_chain_element.py` Object file that contains a `SAGE` polyhedron object (which is our cone) and the associated data and methods:

   - Essential Attributes:

     – `cone, cone_rays_list`: The `SAGE` polyhedron object and the extremal generators of the cone.

     – `generation_step, algorithm_used`: to denote what step this cone was generated and what algorithm was used. 0 if initial cones.

For algorithm used, the letter "i" denotes initial cone, "t" denotes top down, "b" denotes bottom up.

- `hilbert_basis` Storing Hilbert basis of the cone in this object. This ensures that the calculation is only done once.

- Essential Methods:
  - `get_hilbert_basis(self)` Method to interface with the data in the object, with the logic to guarantee that `Normaliz` is used at most once for each `ConeChainElement` object.
  - `output_details(self)` Prints to terminal details about this particular cone. Used in other I/O methods elsewhere.

2. `cone_chain.py` Object file that contains three lists of `ConeChainElements` designed to represent three valid cone poset chains.

   - Attributes:
     - `inner_cone, outer_cone` the cones initializing the experiment.
     - `top_sequence`: first contains `outer_cone`, then every cone generated by the `top_down` algorithm.
     - `bottom_sequence`: first contains `inner_cone`, then every cone generated by the `bottom_up` algorithm.
     - `cone_chain`: generated only when the either `top_down / bottom_up` terminates, and arranges it so that the `inner_cone` is the first ele-

ment and the `outer_cone` is the last element.

- Methods:
  - `top_down` algorithm, implements Hilbert descends.
  - `bottom_up` algorithm, implements height-1 extensions.

3. cone_conjecture_tester.py Wrapper object that deals with all UI, creation/initialization of experiments, loading/saving operations. This also contains functions to generate graphs and save summaries.

Chapter 5

Source Code

Included here are code that is written in `Python` for this research, and they are also concurrently hosted on

`http://www.github.com/TimmyChan/ConeThesis`

`cone_tools.py` This file contains mathematical functions written for the project:

- `cone_containment(C,D)`: Verifies C is contained in D
- `shortest_vector(vectorlist)`, `longest_vector(vectorlist)`: Returns the shortest and longest vectors from a list, respectively.
- `make_primitive(vectlist)`: Returns a primitive vector along the ray generated by some vector in $\mathbb{Z}^d$.
- `generate_random_vector(dim, rmax=10)`: Generates a random vector in $v \in \mathbb{Z}^d$ such that $|v_i| \leq d$ and $v_d \geq 1$. The last condition is used to

guarantee the following function always generates a cone contained in the halfspace $x_d > 0$, so it contains no lines. Using the previous function we always generate a random *primitive* vector.

- `generate_cone(dim, rmax=10,numgen=10), generate_inner_cone(outer)`: Generates a cone using random vectors constrained by the dimension (dim) and the random generation bounds. Numgen is the number of vectors used to generate the cone. The latter generates a cone that is guaranteed to be contained by the outer cone.

- `extremal_generators_outside_inner_cone(inner, outer)`: This is used to find a list of extremal generators at the beginning of the Top Down algorithm.

- `visible_facets(cone,vect), facets_with_max_lambda(visiblefacets,v)`: These are used in the Bottom Up algorithm.

- `vector_sum(listofvectors), zonotope_generators(vectlist)`: Generating a zonotope in $\mathbb{Z}^d$ using the power set of a set of vectors, then taking sum of each element of the power set, and adjoining the empty set as the origin.

5.1 cone_tools.py

```
#!/usr/bin/env sage
""" Contains methods required to generate cones randomly."""

import sys
import sage.all
import experiment_io_tools



def rays_list_to_json_array(rays_list):
        ''' Tool to convert from SAGE numbers to Python native integers
            '''
        return None if rays_list is None else [[int(i) for i in v] for v
            in rays_list]

def gcd_of_list(args):
        """Greatest Common Divisor of a list of integers
        Intended for use to scale a lattice vector
        Args:
                args (list of int)
        Returns:
                int: Greatest Common Divisor of args.
```

```
        """
        return reduce(sage.all.gcd, args)


def cone_containment(C,D):
        """Verifies if cone C is contained in (or equals to) of D
        Args:
                C (SAGE.geometry.Polyhedron): Cone with Normaliz Backend
                D (SAGE.geometry.Polyhedron): Cone with Normaliz Backend
        Returns:
                sofar (boolean): True for C contained in D, False otherwise
                    .
        """
        sofar = True
        for ray in C.rays():
                # loop through every extremal generators of C.
                # If they're all in D then C is in D.
                sofar = (sofar and D.contains(ray))
        return sofar


def shortest_vector(vectorlist):
        """Given a list of SAGE vectors, return shortest WRT Euclidean
```

```
        norm.
    Args:
        vectorlist (list of SAGE vectors): vectors,
    Returns:
        min(vectorlist,key=lambda:x.norm()) (SAGE vector): one with
            shortest norm.
        if the list is empty or there's some other error, return
            None
    """
    try:
        return min(vectorlist, key = lambda x: x.norm())
    except:
        return None


def longest_vector(vectorlist):
    """Given a list of SAGE vectors, return longest WRT Euclidean norm
        .
    Args:
        vectorlist (list of SAGE vectors): vectors,
    Returns:
        min(vectorlist,key=lambda:x.norm()) (SAGE vector): one with
            longest norm.
```

```
            if the list is empty or there's some other error, return
                None
        """
        try:
            return max(vectorlist, key = lambda x: x.norm())
        except:
            return None


####################################
# TOOLS FOR GENERATING RANDOM CONES #
####################################

def make_primitive(vectlist):
        """Given some vector v in Z^d, return primitive of v
        Args:
            vectlist (list of integers): list of length d representing
                some vector in Z^d
        Returns:
            vector(primvectlist) (SAGE vector): v * 1/GCD(entries of v)
                .
        """
        gcd = gcd_of_list(vectlist)
```

```
        primvectlist = [(i / gcd) for i in vectlist]
        return sage.all.vector(primvectlist)

def generate_random_vector(dim, rmax=10):
        """ Generate a random vector in Z^d
        Args:
                dim (int): ambient dimension
                rmax (int): upperbound for random number generator
        Returns:
                vect (SAGE vector): random vector of the form (x_1,...,x_(n
                    -1),x_n)
                where -rmax < x_1,...,x_(n-1) < rmax
                and 1 < x_n < rmax.
                This guarentees that the cones generated with this vector
                    lie within
                the halfspace x_n > 0, so forces all cones generated this
                    way to be pointed.
        """
        vectlist = [sage.all.randint(-rmax,rmax) for i in range(dim-1)]
        # make the first n-1 entries
        vectlist.append(sage.all.randint(1,rmax))
        # append the last entry
        vect = make_primitive(vectlist)
```

```
        # make a primative vector and return it.
        return vect

def generate_cone(dim, rmax=10, numgen=5):
        """ Generates a random SAGE polyhedral cone C with Normaliz
            backend
        where C is pointed, proper,full dimensional and lies strictly in
        the halfspace x_d > 0.
        Args:
                dim (int): dimension of the ambient space, number of
                    entries in vector
                rmax (int): max number for random number generator
                numgen (int): number of generators. (Default = 10)
        Returns:
                Temp (SAGE.geometry.Polyhedron): SAGE cone object with
                    Normaliz backend.
        """
        if numgen < dim:    # catch: if numgen < dim, guarenteed not full
            dimensional.
                numgen = int(dim)   # force numgen to have at least be full
                     dimensional.

        vects = [generate_random_vector(dim,rmax) for i in range(numgen)]
```

```
            # list of random vectors
        # conical hull of vectors in list vects.
        temp = sage.all.Polyhedron(rays=[sage.all.vector(v) for v in vects
            ],backend='normaliz')

        if dim > 2:
                # the number of extremal generators is only greater than
                # the dimension if the dimension is greater than 2.
                while len(temp.rays_list())<numgen:
                        #keep appending random vectors until we have a full
                            dimensional cone.
                        vects.append(generate_random_vector(dim,rmax))
                        temp = sage.all.Polyhedron(rays=[sage.all.vector(v)
                            for v in vects],backend='normaliz')
                        # keep tacking on random vectors, eventually
                        # the convex hull will be full dimensional (this
                            doesn't take long for low dimensions like 4 or 5)
        return temp

def generate_inner_cone(outer, rmax=10, numgen=5):
        """Generates a full dimensional cone that is contained by outer.
        Args:
                outer (SAGE.geometry.Polyhedron): Outer Cone
```

```
        rmax: upperbound for random number generator.
        numgen (int): number of generators
Returns:
        inner (SAGE.geometry.Polyhedron): Inner Cone contained by
            outer cone
"""
dim = outer.dimension()
# store the dimension of the outer cone
if numgen < dim:
        numgen = int(dim)
vectlist = [] # empty list to collect the generator of the inner
    cone

while len(vectlist) < numgen:
        temp_vect = generate_random_vector(dim, rmax)
        if outer.contains(temp_vect):
                vectlist.append(temp_vect)

inner = sage.all.Polyhedron(rays=[sage.all.vector(v) for v in
    vectlist],backend='normaliz')
if dim > 2:
        # the number of extremal generators is only greater than
        # the dimension if the dimension is greater than 2.
```

```
            while len(inner.rays_list())<numgen:
                    #keep looping until we have a full dimensional cone.
                    temp_vect = generate_random_vector(dim, rmax)
                    if outer.contains(temp_vect):
                            vectlist.append(temp_vect)
                    del inner
                    inner = sage.all.Polyhedron(rays=[sage.all.vector(v)
                        for v in vectlist],backend='normaliz')
                    # keep tacking on random vectors, eventually
                    # the convex hull will be full dimensional
        return inner

##################################
# TOOLS FOR BOTTOM UP ALGORITHM #
##################################

def extremal_generators_outside_inner_cone(inner, outer):
        """ Given inner, outer cone pair return extremal generators of
            outer cone
                not contained by inner cone
        Args:
                inner (SAGE.geometry.Polyhedron): cone of dimension d
                outer (SAGE.geometry.Polyhedron): contains inner cone of
```

```
            same ambient dimension
        Returns:
            ext_gens_final (List of SAGE vectors): list of extremal
                generators not in inner cone.
        """
        # grab the outer cone's extremal generators and loop through each
            one:
        ext_gens = outer.rays_list()
        for r in outer.rays_list():
            # if r is inside of inner cone, discard it.
            if (inner.contains(r)):
                ext_gens.remove(r)
        # convert the lists into vectors
        ext_gens_final = [sage.all.vector(i for i in v) for v in ext_gens]

        return ext_gens_final


def visible_facets(cone,vect):
        ''' C is a full dimensional cone, v is external to C and returns
                a list of facets [f1,f2,...,fn], where each f_i is
                    visible WRT v
            Args:
```

```
                cone (sage.all.Polyhedron): Some cone
                vect (sage vector): a vector outside of cone.
        Returns:
                visiblefacets (List of face): Returns a list of
                    faces that are
                        visible to vect
'''
if cone.contains(vect):
        return None
else:
        #print("v = {}\n C = \n{}".format(v,C.rays_list()))
        numfacets = len(cone.faces(cone.dim()-1)) # counts the
            number of facets of codimension 1

        facets = cone.faces(cone.dim()-1)
        visiblefacets = []
        for facet in facets:
                #print("Facet {}:".format(facets.index(facet)))
                facet_ineq = facet.ambient_Hrepresentation(0)
                #print("\tambient halfspace {}".format(facet_ineq))
                facet_visibile = (facet_ineq.eval(sage.all.vector(
                    vect)) < 0)
                #print facet_ineq.eval(vector(v))
```

```
                    #print("\trays = {}".format(facet.as_polyhedron().
                        rays_list()))
                    if facet_visibile:
                            visiblefacets.append(facet)

            return visiblefacets


def facets_with_max_lambda(visiblefacets,v):
        ''' given visible facets, find max lambda '''
        return max(visiblefacets, key = lambda x: abs(x.
            ambient_Hrepresentation(0).eval(sage.all.vector(v))))


def vector_sum(listofvectors):
        ''' Returns vector sum of a given list of vectors takes a list of
            vectors of the same dimension
        returns a vector that is the termwise sum of each vector in the
            list.
        '''
        length = len(listofvectors)
        dim = len(listofvectors[0])
```

```
        summand = [0 for i in range(dim)]
        for i in range(length):
                for d in range(dim):
                        summand[d] = summand[d] + listofvectors[i][d]
        return summand


def zonotope_generators(vectlist):
        ''' Form a zonotope given v_1,...,v_n. Gives the vertices of said
            zonotope.
        Args:
                vectlist (list of vectors): assumes they're all same
                    dimension.
        Returns:
                zonogens (list of vectors): vertices of a
                    zonotope_generators generated
                        by vectlist.
        '''

        # First create the powerset of the list and remove the empty set.
        dimension = len(vectlist[0])
        combo = list(sage.all.powerset(vectlist))
        # remove the empty set and replace it with the 0 vector
```

```
        combo.remove([])
        combo.append([[0 for i in range(dimension)]])
        # take the vector sum of each set in the modified power set.
        zonogens = [vector_sum(c) for c in combo]
        return zonogens


def sanity_check(inner_cone,outer_cone):
        '''
        Args:
                inner_cone (sage.all.Polyhedron): Cone with Normaliz
                    backend
                outer_cone (sage.all.Polyhedorn): Cone with Normaliz
                    backend
        Returns:
                boolean: True if inner_cone and outer_cone satisifes
                    experiment constraints, false otherwise.
        '''
    if not inner_cone.is_full_dimensional():
        print("inner_cone is not full dimensional!")
        experiment_io_tools.printseparator()
        experiment_io_tools.printseparator()
        print("RESTARTING INPUT!")
```

```
if not outer_cone.is_full_dimensional():
    print("outer_cone is not full dimensional!")
    experiment_io_tools.printseparator()
    experiment_io_tools.printseparator()
    print("RESTARTING INPUT!")
if not inner_cone.lines_list() == []:
    print("inner_cone is not proper!")
    experiment_io_tools.printseparator()
    experiment_io_tools.printseparator()
    print("RESTARTING INPUT!")
if not outer_cone.lines_list() == []:
    print("outer_cone is not proper!")
    experiment_io_tools.printseparator()
    experiment_io_tools.printseparator()
    print("RESTARTING INPUT!")
if not cone_containment(inner_cone,outer_cone):
    print("inner_cone is not in outer_cone!")
    experiment_io_tools.printseparator()
    experiment_io_tools.printseparator()
    print("RESTARTING INPUT!")
return (inner_cone.is_full_dimensional() and outer_cone.
    is_full_dimensional() and inner_cone.lines_list() == [] and
    outer_cone.lines_list() == [] and cone_containment(inner_cone,
```

```
outer_cone))
```

5.2 cone_chain_element.py

```
#!/usr/bin/env sage

"""ConeChain

This module defines ConeChainElement, an object that will contain
    experimental data for each cone.
"""

import sage.all
import cone_tools
import experiment_io_tools
import json


class ConeChainElement(object):
        """ A data structure to hold a cone and its hilbert basis
        and associated experimental data.
        Class Attribute:
                (currently not used) num_hilbert_calc (int): Keeps track of
                    the number of times we call Hilbert basis calculations.
        Attributes:
```

```
        cone (sage.all.Polyhedron): the cone object to be stored
        cone_rays_list (list of lists): extremal generators of cone
             object (mostly for JSON)
        generation_step (int):  step in the generation process
        algorithm_used (str):  "i" = initial step
               "t" = top down
               "b" = bottom up
        hilbert_basis (list of lists): Hilbert basis of cone
"""
# MAY NOT BE NECESSARY num_hilbert_calc = 0

def __init__(self,cone,generation_step=0, algorithm_used="i",
                     hilbert_basis=None, hilbert_basis_size=None,
                     longest_hilbert_basis_element=None,
                     longest_hilbert_basis_element_length=None):
        self.cone = cone
        self.cone_rays_list = cone.rays_list()
        self.generation_step = generation_step
        self.algorithm_used = algorithm_used
        self.hilbert_basis = hilbert_basis
        self.hilbert_basis_size = hilbert_basis_size
        self.longest_hilbert_basis_element =
            longest_hilbert_basis_element
```

```
        self.longest_hilbert_basis_element_length =
            longest_hilbert_basis_element_length

    def get_hilbert_basis(self,forced=False):
        """ Retreives the hilbert basis, only calculates once. """
        # if the hilbert_basis data is empty, generate it using
            Normaliz and store it
        if self.hilbert_basis == None or forced:
            self.hilbert_basis = list(self.cone.
                integral_points_generators()[1])
            self.hilbert_basis_size = len(self.hilbert_basis)
            self.longest_hilbert_basis_element = cone_tools.
                longest_vector(self.hilbert_basis)
            self.longest_hilbert_basis_element_length = float(
                sage.all.vector(self.
                get_longest_hilbert_basis_element()).norm())
        #return the stored value.

        return list(self.hilbert_basis)

    def get_longest_hilbert_basis_element(self):
        if self.longest_hilbert_basis_element == None:
            self.longest_hilbert_basis_element = cone_tools.
```

```
                longest_vector(self.get_hilbert_basis())
        return self.longest_hilbert_basis_element

def hilbert_graph_data_size(self):
        if self.hilbert_basis_size == None:
                self.hilbert_basis_size = float(len(self.
                    get_hilbert_basis()))
        return self.hilbert_basis_size

def hilbert_graph_data_length(self):
        if self.longest_hilbert_basis_element_length == None:
                self.longest_hilbert_basis_element_length = float(
                    sage.all.vector(self.
                    get_longest_hilbert_basis_element()).norm())
        return self.longest_hilbert_basis_element_length

def rays_list(self):
        """Returns the extremal generators of the cone
        self.cone.rays_list() (list of lists): normaliz returns
            this.
        """
        return self.cone_rays_list
```

```
        def output_details(self):
                """ Prints basic details about this particular cone. """
                print("rays = {}".format(self.rays_list()))
                print("generated on step {} with algorithm {}".format(self.
                    generation_step,self.algorithm_used))
                if self.hilbert_basis == None:
                        L = 0
                else:
                        L = len(self.hilbert_basis)
                print("number of elements in hilbert_basis = {}".format(L))


        def calculate(self):
                if self.hilbert_basis == None:
                        self.hilbert_basis = list(self.cone.
                            integral_points_generators()[1])
                self.longest_hilbert_basis_element = cone_tools.
                    longest_vector(self.hilbert_basis)



class ConeChainElementEncoder(json.JSONEncoder):
        def default(self, obj):
                if isinstance(obj, ConeChainElement):
                        cone_rays_list_json = cone_tools.
```

```
                    rays_list_to_json_array(obj.cone_rays_list)
                hilb_basis = cone_tools.rays_list_to_json_array(obj.
                    hilbert_basis)
                longest = None if obj.hilbert_basis is None else [
                    int(i) for i in obj.longest_hilbert_basis_element
                    ]
                return {'cone_rays_list' : cone_rays_list_json,
                        'generation_step' : obj.generation_step,
                        'algorithm_used' : obj.algorithm_used,
                        'hilbert_basis' : hilb_basis,
                        'hilbert_basis_size': obj.
                            hilbert_graph_data_size(),
                        'longest_hilbert_basis_element': longest,
                        'longest_hilbert_basis_element_length': obj.
                            hilbert_graph_data_length()}
            return json.JSONEncoder.default(self, obj)

class ConeChainElementDecoder(json.JSONDecoder):
        def __init__(self, *args, **kwargs):
                json.JSONDecoder.__init__(self, object_hook=self.
                    object_hook, *args, **kwargs)

        def object_hook(self, dictionary):
```

```
if 'cone_rays_list' in dictionary:
        cone = sage.all.Polyhedron(rays=dictionary['
            cone_rays_list'],backend='normaliz')

if dictionary['hilbert_basis'] == 'null':
        return ConeChainElement(cone,
                dictionary['generation_step'],
                dictionary['algorithm_used'])

else:
        return ConeChainElement(cone,
                dictionary['generation_step'],
                dictionary['algorithm_used'],
                [sage.all.vector(v) for v in dictionary['
                    hilbert_basis']],
                dictionary['hilbert_basis_size'],
                sage.all.vector(dictionary['
                    longest_hilbert_basis_element']),
                dictionary['
                    longest_hilbert_basis_element_length'])
```

5.3 cone_chain.py

```
#!/usr/bin/env sage

"""ConeChain

This module contains the an object that will contain a sequence of cones
and defines ConeChainElement, an object that will contain experimental
    data for each cone.
"""
import sage.all
import cone_tools
import experiment_io_tools
import json
import sys
from cone_chain_element import ConeChainElement, ConeChainElementEncoder,
    ConeChainElementDecoder
import pylab as plt
import datetime, os


class ConeChain(object):
        """ Initializes with two cones (assuming containment)
```

```
We wish the represent the data so that a sequence of cones (poset)

        C = C_0 < ... < C_n = D

can be "grown" from two ends, the tail or the head so that

bottom_sequence: C = C_0 < C_1 < ...
top_sequence: D = D_0 > D_1 > ...

After some steps, we expect C_n and D_k (for some finite n and k)
        to satisify the poset condition. When this happens, we will
            glue the two together;
cone_poset_chain: [C_0, C_1, ..., C_n, D_k, D_(k-1), ..., D_0]
        Note that top_sequence needs to be "glued" backwards for
            the containment to make sense!

Attributes:
        outer_cone (sage.all.Polyhedron): outer cone
        inner_cone (sage.all.Polyhedron): inner cone, assume
            outer_cone contains inner_cone
        outer_cone_rays_list (list of lists of integers): Extremal
            generators of outer cone
```

```
        inner_cone_rays_list (list of lists of integers): Extremal
            generators of inner cone
        top_sequence (list of ConeChainElements): Begins with
            outer_cone
        bottom_sequence (list of ConeChainElements): Begins with
            inner_cone
        cone_poset_chain (list of ConeChainElements): Begins empty
            until glue()
                intended to contain the order of the
                    algorithmetically generated poset chain.
        sequence_complete (boolean): flag to see if the sequence is
             ready for gluing
        valid_poset (boolean): flag to see if the entire chain fits
             poset condition.
    """
    def __init__(self, inner, outer,
                 top_seq=None, bottom_seq=None,
                 poset_chain=None, seq_comp=False, valid = True):
        """Initiate using cones, then initialize data """
        self.outer_cone = outer
        self.inner_cone = inner
        self.outer_cone_rays_list = outer.rays_list()
        self.inner_cone_rays_list = inner.rays_list()
```

```
        self.dimension = outer.dimension()
        # Lists to store poset elements
        self.top_sequence = [ConeChainElement(outer)] if top_seq is
            None else top_seq
        self.bottom_sequence = [ConeChainElement(inner)] if
           bottom_seq is None else bottom_seq
        self.cone_poset_chain = [] if poset_chain is None else
           poset_chain

        # Internal logic flags
        # sequence complete whenever the poset condition is
           satisified,
        # so default is False.
        self.sequence_complete = seq_comp
        # sequence is considered a poset chain
        # when verfiication is checked for every consecutive pair
           of cones.
        self.valid_poset = valid

def current_inner(self):
        """ get the current inner cone
        Args: none
```

```
        Returns: bottom_sequence[-1].cone (sage.all.Polyhedron)
        """
        return self.bottom_sequence[-1].cone


def current_outer(self):
        """ get the current outer cone
        Args: none
        Returns: top_sequence[-1].cone (sage.all.Polyhedron)
        """
        return self.top_sequence[-1].cone


def append_top(self, somecone):
        """ Appends a cone to the top sequence
        Args:
                somecone (sage.all.Polyhedron): A polyhedral cone
        Returns: none.
        """
        self.top_sequence.append(ConeChainElement(somecone,self.
           number_of_steps(),"t"))
```

```
def append_bottom(self, somecone):
        """ Appends a cone to the bottom sequence
        Args:
                somecone (sage.all.Polyhedron): A polyhedral cone
        Returns: none.
        """
        self.bottom_sequence.append(ConeChainElement(somecone,self.
            number_of_steps(),"b"))


def top_down(self, steps=1):
        """ Top down algorithm
        Args: none
        Returns: True if top_down completes the sequence
                        False if top_down isn't complete.
        """

        if self.sequence_complete:
                print("Sequence already complete.")
                return True

        # Collect the set of extremal generators of the
            intermediate cone that is not in C
        extremal_gens_outside_inner_cone =  cone_tools.
```

```
    extremal_generators_outside_inner_cone(self.
    bottom_sequence[-1].cone, self.top_sequence[-1].cone)

if len(extremal_gens_outside_inner_cone) == 0:
        print("Ended up with the same cone, run self.
            check_complete()")
        return self.check_complete()



vector_to_remove = cone_tools.shortest_vector(
    extremal_gens_outside_inner_cone)

intermediate_hilb = self.top_sequence[-1].get_hilbert_basis
    ()

try:
        intermediate_hilb.remove(vector_to_remove)
except:
        print("Not sure what happened here but {} is not in
            intermediate_hilb".format(vector_to_remove))

new_generators = intermediate_hilb + self.bottom_sequence
    [-1].cone.rays_list()
```

```
        self.append_top(sage.all.Polyhedron(rays=new_generators,
            backend='normaliz'))

        # minor recursion; if the number of steps is not the
            default, then repeat by
        # returning its results.
        if steps > 1:
                return self.top_down(steps-1)
        return self.check_complete()

def bottom_up(self,steps=1):
        """ Bottom Up algorithm
        Args: none
        Returns: True if top_down completes the sequence
                        False if top_down isn't complete.
        """

        if self.sequence_complete:
                print("Sequence already complete.")
                return True

        current_inner = self.bottom_sequence[-1].cone
```

```
current_outer = self.top_sequence[-1].cone

# find all the extremal generators of outer cone outside of
    current inner cone
vlist = cone_tools.extremal_generators_outside_inner_cone(
   current_inner, current_outer)

# if the list is empty, we should be done.
if len(vlist) == 0:
       print("Ended up with the same cone, run self.
           check_complete()")
       return self.check_complete()

# find the longest of the vectors strictly outside
   current_inner
longestv = cone_tools.longest_vector(vlist)

# collect visible facets WRT longestv
visible_facets = cone_tools.visible_facets(current_inner,
   longestv)

# out of those, choose the one with the max lambda
visible_max_lambda_facet = cone_tools.
```

```
    facets_with_max_lambda(visible_facets, longestv)

# collect all the extremal generators of the facet
facet_generators = visible_max_lambda_facet.as_polyhedron()
    .rays_list()

# form a zonotope (collect the list of vertices of the
    zonotope) using facet_generators
preshift_zonotope_gens = cone_tools.zonotope_generators(
    facet_generators)

# 1/lambda as discussed in paper. This is the shift factor
    needed to guarentee the zonotope will
# contain at least one lattice point.
shift_factor = 1/ abs(visible_max_lambda_facet.
    ambient_Hrepresentation(0).eval(sage.all.vector(longestv
    )))

# the origin, a point on the zonotope, will be shifted also
     (so the image is just the shift vector)
shift_vector = [shift_factor*i for i in longestv]

# take all the generators, shift them, and then include the
```

```
    shift vector
shifted_generators = [[gen[i] + shift_vector[i] for i in
    range(self.dimension)] for gen in preshift_zonotope_gens
    ]
shifted_generators.append(shift_vector)

# formally generate the zonotope now
zono = sage.all.Polyhedron(vertices=shifted_generators,
    backend='normaliz')

# find the lattice points in the zonotope
lattice_points_in_zono = list(zono.integral_points())

# then find the shortest one
shortest_lattice_point_in_zono = cone_tools.shortest_vector
    (lattice_points_in_zono)

self.append_bottom(current_inner.convex_hull(sage.all.
    Polyhedron(rays=[shortest_lattice_point_in_zono],backend
    ='normaliz')))

if steps > 1:
        return self.bottom_up(steps-1)
```

```
        else:
                return self.check_complete()


def number_of_steps(self):
        """ Returns the number of steps """
        if self.sequence_complete:
                return len(self.cone_poset_chain) - 2
        else:
                return len(self.top_sequence) + len(self.
                    bottom_sequence) - 2


def verify_validity(self):
        """ Loops through the sequences as appropriate and checks
            poset conditions
        Note: This is a computationally heavy function, and should
            not be used frequently
                For optimization, this should be ran once per
                    experiment, instead of for every trial.
        Args: none
        Returns: True - sequence is valid (not necessarily complete
            )
```

```
"""
# if the sequence is complete, loop through each
    consecutive pair
# and verify the entire sequence is good.
# break out and return false if even just one of them fail
self.valid_poset = True
if self.sequence_complete:
        # the length of the sequence - 1 because we're
            looking at consecutive pairs.
        for i in range(self.number_of_steps()-1):
                if not self.poset_check(self.cone_poset_chain
                    [i], self.cone_poset_chain[i+1]):
                        self.valid_poset = False
# otherwise, go through top down and bottom up
else:
        if len(self.bottom_sequence) > 1:
                for i in range(len(self.bottom_sequence)-1):
                        if not self.poset_check(self.
                            bottom_sequence[i], self.
                            bottom_sequence[i+1]):
                                self.valid_poset = False
        if len(self.top_sequence) > 1:
                for i in range(len(self.top_sequence)-1):
```

```
                                if not self.poset_check(self.
                                    top_sequence[-(i+1)], self.
                                    top_sequence[-(i+2)]):
                                        self.valid_poset = False
        return self.valid_poset


def check_complete(self):
        """ Checks if the sequence is complete
        1) verify the poset conditions on the last entries of
                top_sequence and bottom_sequence.
        2) if the poset condition is met, glue the bottom_sequence
            and
                top_sequence together into cone_poset_chain.
        Args: Nothing
        Returns: Nothing.
        """
        # sequence is complete if the poset condition is met for
            the two intermediate cones
        self.sequence_complete = self.poset_check(self.
            bottom_sequence[-1],
```

```
        # if the sequence is complete, we should glue them together
            .
        if self.sequence_complete:
                if len(self.cone_poset_chain) == 0:
                        self.glue()
                return True
        else:
                return False

def poset_check(self, inner, outer):
        """ Verifies if the Poset condition is met by inner and
            outer
        Default behavior for same cone given is to return True.
        We do this by checking the hilbert basis of inner, then
            outer,
        and verify that:
                1) One or less extremal generator of outer is
                    outside inner, call this v
                2) Hilbert basis of outer take away v should be a
                    subset of
                        the Hilbert basis of inner.
```

```
    Args:
            inner (ConeChainElement): "inner" cone
            outer (ConeChainElement): "outer" cone (assume inner
                 is contained)
    Returns:
            poset_condition (Boolean):  True if C, D satisify
                the poset condition
                           or if they're the same cone;
                   False otherwise.
    """

    # retreive the hilbert basis
    hilbert_inner = inner.get_hilbert_basis()
    hilbert_outer = outer.get_hilbert_basis()



    if hilbert_inner == hilbert_outer:
            return True
    # Finding extremal generator of D not in C
    v = cone_tools.extremal_generators_outside_inner_cone(inner
        .cone,outer.cone)


    if len(v) > 1:
```

```
        # if there's more than one extremal generator
            outside of C,
        # this cannot satisify the poset condition.
        return False
# Removing the extremal generator (should be just one) from
    hilbert_outer
#print("DEBUG: v[0] = {}".format(v[0]))
if len(v) == 1:
        try:
                hilbert_outer.remove(v[0])
        except:
                print("Failed trying to remove \n\t{} from \n
                    {}".format(v[0],hilbert_outer))

# Assume that the poset condition is satisified at this
   point, then
# loop through each vector in the Hilbert basis of D,
        poset_condition = True
        for vect in hilbert_outer:
                # the poset condition will remain true as
                    long as
                # each vect in Hilbert basis of D is also
                # contained in the Hilbert basis of D
```

```
                    poset_condition = poset_condition and (vect
                        in hilbert_inner)

            return poset_condition
        if len(v) == 0:
            return True

def glue(self):
        """ Glue bottom_sequence and top_sequence and store into
        cone_poset_chain

        1) Check if sequence_complete flag is true;
        2) if yes, join the bottom_sequence and top_sequence into
            cone_poset_chain
        so that cone_poset_chain begins with bottom_sequence, then
            top_sequence in reverse order
        """
        if self.sequence_complete:
            # get the index of the last element of each sequence
                .
            # If we run bottom up or top down purely, one of the
                sequence
            # ends with inner_cone or outer_cone, creating an
```

```
                overlap.
            # if the sequence's ends are the same cone, just pop
                one WLOG
            temp_seq = [i for i in self.top_sequence]

            if self.bottom_sequence[-1].cone == self.
                top_sequence[-1].cone:
                    temp_seq.pop() # Remove one of the repeated
                        cones

            # if we end up with some different cones:
            self.cone_poset_chain = [] + self.bottom_sequence
            self.cone_poset_chain = self.cone_poset_chain + [
                temp_seq[-(i+1)] for i in range(len(temp_seq))]

def output_to_terminal(self):
        """ Prints essencial information about the sequence
            Args: None
            Returns: None
        """
        experiment_io_tools.new_screen("Printing summary
            information about the cone chain:")
        print("inner_cone has generators: \n{}\n".format(self.
```

```
            inner_cone.rays_list()))
        print("outer_cone has generators: \n{}\n".format(self.
            outer_cone.rays_list()))
        print("\tsequence_complete = {}\n".format(self.
            sequence_complete))
        print("\ttop_sequence has length {}\n".format(len(self.
            top_sequence)))
        print("\tbottom_sequence has length {}\n".format(len(self.
            bottom_sequence)))
        print("\tcone_poset_chain has length {}\n".format(len(self.
            cone_poset_chain)))
        #print("we have used Normaliz for hilbert basis calculation
             {} times.".format(ConeChainElement.num_hilbert_calc))
        experiment_io_tools.pause()
        experiment_io_tools.new_screen()




def chain_details(self):
        """ Prints the details for each cone in the chain
                Args: None
                Returns: None
```

```
"""
if self.sequence_complete:
        experiment_io_tools.new_screen("Printing details of
           the (complete) chain.")
        # only need to use case 2,
        # go through each cone in the cone_poset_chain:
        i = 0
        for tcone in self.cone_poset_chain:
                print("cone_poset_chain"+"[{}]:".format(i))
                tcone.output_details()
                i = i + 1
                # pause every fifth cone
                if sage.all.mod(i,5) == 0:
                        experiment_io_tools.pause()

else:
        # do the same as above but with bottom_sequence
        # and top sequence instead.
        switcher = {
                0: ("bottom_sequence",self.bottom_sequence),
                1: ("top_sequence", self.top_sequence),
        }
        for key in range(2):
```

```
                    experiment_io_tools.new_screen("Printing
                        details of the (incomplete) chain.")
                    print(switcher[key][0])
                    i = 0
                    for tcone in switcher[key][1]:
                            print(switcher[key][0]+"[{}]:".format(
                                i))
                            tcone.output_details()
                            i = i + 1
                            if sage.all.mod(i,5) == 0:
                                    experiment_io_tools.pause()
                    experiment_io_tools.pause()
        # now print a summary
        self.output_to_terminal()
        experiment_io_tools.new_screen()

def generate_hilbert_graphs(self, folder=None, experiment_name=
    None):
        ''' Generates the graphs for the length of longest hilbert
            basis element,
                and also the number of vectors in the hilbert basis.
        '''
        directory = "DATA/{}d/Hilbert Graphs of Unnamed Experiments
```

```
    /".format(self.dimension) if folder is None else folder
filename = str(datetime.datetime.now()) if experiment_name
    is None else experiment_name
try:
        os.makedirs(directory, 0755)
except:
        NotImplemented



switcher = {"top_sequence" : self.top_sequence,
                      "bottom_sequence" : self.
                          bottom_sequence,
                      "cone_poset_chain" : self.
                          cone_poset_chain}


i = 0
for name in switcher:
        if len(switcher[name]) > 1:
                length_filename = name + " LENGTH {} steps.
                    png".format(self.number_of_steps())
                length_filename_no_step = name + " LENGTH.png
                    "
```

```
hilbert_graph_data_length = [cone.
    hilbert_graph_data_length() for cone in
    switcher[name]]
plt.figure(i)

plt.plot(hilbert_graph_data_length)
plt.xlabel("Number of Steps: {}".format(name)
    )
plt.ylabel("Vector Norm")
plt.title("Length of the longest element in
    the Hilbert Basis: {}".format(name))
plt.savefig(directory + length_filename)
plt.savefig(directory +
    length_filename_no_step)
plt.close(i)

size_filename = name + " SIZE {} steps.png".
    format(self.number_of_steps())
size_filename_nostep = name + " SIZE.png"
hilbert_graph_data_size = [cone.
    hilbert_graph_data_size() for cone in
    switcher[name]]
plt.figure(i+1)
```

```
                    plt.xlabel("Number of Steps: {}".format(name)
                        )
                    plt.ylabel("Number of Vectors")
                    plt.title("Size of the Hilbert Basis: {}".
                        format(name))
                    plt.plot(hilbert_graph_data_size)
                    plt.savefig(directory + size_filename)
                    plt.savefig(directory + size_filename_nostep)
                    plt.close(i+1)
                i += 2


def recalc(self):
        ''' force self to recalc every hilbert basis. '''
        print("\t\tRecalculating Hilbert basis for top_sequence..."
            )
        for cone in self.top_sequence:
                cone.get_hilbert_basis(forced=True)

        print("\t\tRecalculating Hilbert basis for bottom_sequence
            ...")
        for cone in self.bottom_sequence:
                cone.get_hilbert_basis(forced=True)
```

```
                print("\t\tRecalculating Hilbert basis for cone_poset_chain
                    ...")
                for cone in self.cone_poset_chain:
                        cone.get_hilbert_basis(forced=True)



class ConeChainEncoder(json.JSONEncoder):
        def default(self,obj):
                if isinstance(obj, ConeChain):
                        outer_cone_rays_list_json = cone_tools.
                            rays_list_to_json_array(obj.outer_cone_rays_list)
                        inner_cone_rays_list_json = cone_tools.
                            rays_list_to_json_array(obj.inner_cone_rays_list)
                        top_seq = [json.dumps(cone_element,cls=
                            ConeChainElementEncoder) for cone_element in obj.
                            top_sequence]
                        bottom_seq = [json.dumps(cone_element,cls=
                            ConeChainElementEncoder) for cone_element in obj.
                            bottom_sequence]
                        poset_chain = [json.dumps(cone_element,cls=
```

```
                        ConeChainElementEncoder) for cone_element in obj.
                        cone_poset_chain]

                    return {'outer_cone_rays_list' :
                        outer_cone_rays_list_json,
                            'inner_cone_rays_list' :
                                inner_cone_rays_list_json,
                            'dimension' : obj.dimension,
                            'top_sequence': top_seq,
                            'bottom_sequence': bottom_seq,
                            'cone_poset_chain': poset_chain,
                            'sequence_complete': obj.sequence_complete,
                            'valid_poset': obj.valid_poset}
            return json.JSONEncoder.default(self, obj)

class ConeChainDecoder(json.JSONDecoder):
        def __init__(self, *args, **kwargs):
                json.JSONDecoder.__init__(self, object_hook=self.
                    object_hook, *args, **kwargs)
        def object_hook(self, dictionary):
                if 'outer_cone_rays_list' in dictionary:
                        outer = sage.all.Polyhedron(rays=dictionary['
                            outer_cone_rays_list'],backend='normaliz')
```

```
            if 'inner_cone_rays_list' in dictionary:
                    inner = sage.all.Polyhedron(rays=dictionary['
                        inner_cone_rays_list'],backend='normaliz')
            top_seq = [json.loads(cone_element,cls=
                ConeChainElementDecoder) for cone_element in dictionary[
                'top_sequence']]
            bottom_seq = [json.loads(cone_element,cls=
                ConeChainElementDecoder) for cone_element in dictionary[
                'bottom_sequence']]
            poset_chain = [json.loads(cone_element,cls=
                ConeChainElementDecoder) for cone_element in dictionary[
                'cone_poset_chain']]
            return ConeChain(inner, outer, top_seq, bottom_seq,
                poset_chain, dictionary['sequence_complete'], dictionary
                ['valid_poset'])



class ConeChainInitialConditionExtractor(json.JSONDecoder):
        def __init__(self, *args, **kwargs):
                json.JSONDecoder.__init__(self, object_hook=self.
                    object_hook, *args, **kwargs)
        def object_hook(self, dictionary):
```

```
if 'outer_cone_rays_list' in dictionary:
        outer = sage.all.Polyhedron(rays=dictionary['
            outer_cone_rays_list'],backend='normaliz')
if 'inner_cone_rays_list' in dictionary:
        inner = sage.all.Polyhedron(rays=dictionary['
            inner_cone_rays_list'],backend='normaliz')
return ConeChain(inner, outer)
```

5.4 cone_conjecture_tester.py

```
#!/usr/bin/env sage
""" Wrapper of the experiment, contains all interface stuff
"""
#!/usr/bin/env sage

"""ConeConjectureTester

This module contains the an object that will contain a sequence of cones
and defines ConeChainElement, an object that will contain experimental
    data for each cone.

"""

import sage.all
import json
import sys
from cone_chain_element import ConeChainElement, ConeChainElementEncoder,
    ConeChainElementDecoder
import pylab as plt
import datetime, os
```

```
import experiment_io_tools
import cone_tools
import cone_chain


class ConeConjectureTester(object):
        """All user interface with the Experiment object goes here.
        Class Attributes:
                text_* (string): menu strings used for every experiment's
                    Terminal UI
                main_menu_dict_* (dict): dictionaries where the key is the
                    choice and the value is the menu texts.

        UI Attributes:
                batch_mode (boolean): flag to denote if batch mode is
                    happening
                loaded (boolean): flag to denote if the object contains an
                    active experiment.

        """
        text_main_title = "Cone Conjecture Tester v1.0"

        text_create = "Create new experiment"
```

```
text_load_continue = "Load and continue existing experiment"
text_load_copy = "Load and copy existing experiment to new file"
text_load_initial = "Load only initial values of previous
    experiment"
text_manual_input = "Create new experiment (manual_input)"
text_save_exit = "Save and exit"

text_summary = "Display summary of current experiment"
text_run_experiment = "Run current experiment with current
    settings"
text_print_graphs = "Generate and save graphical data."
text_display_all_details = "Display all details"


main_menu_dict_initial = {1: text_create,
       2: text_load_continue,
       3: text_load_copy,
       4: text_load_initial,
       8: text_manual_input,
       1337: text_save_exit}

main_menu_dict_loaded = {  0: text_run_experiment,
       1: text_create,
```

```
        2: text_load_continue,
        3: text_load_copy,
        4: text_load_initial,
        5: text_summary,
        6: text_print_graphs,
        7: text_display_all_details,
        8: text_manual_input,
        1337: text_save_exit}


# names of the algorithms
text_top_down = "Top Down"
text_bottom_up = "Bottom Up"
text_alternating = "Alternating"

# not zero
run_mode_dict = { 1: text_top_down,
                                    2: text_bottom_up,
                                    3: text_alternating}

def __init__(self, dim=None, expr_name=None, some_cone_chain=None,
     runmode=None, batchmode=False, directory=None, numgen=None,
    steps=None, rmax=2):
```

```
        '''
        Attributes:
                dimension (int): dimension of the experiment, expecting dim
                    >= 2
                experiment_name (string): the name of the experiment
                directory (string): the directory the experiment will be
                    contained
                raw_data_path (string): the path to the .JSON file that
                    contains all raw data
                bound (int): the bound on the absolute value of the
                    coordinates of the randomly generated vectors
                run_mode (int): 1 = Top Down, 2 = Bottom Up, 3 =
                    Alternating
                steps (int): Steps to run before saving and prompting to
                    continue again.
                alternation_constant (int): number of steps to run each
                    algorithm before switching.
                current_cone_chain (ConeChain): object to contain the
                    mathematical data
                num_gen (int): number of extremal generators used for the
                    randomly generated cones.
        '''
                self.dimension = dim #dimension
```

```
self.experiment_name = expr_name #experiment name
self.directory = directory # default will be DATA/{}d/
    expr_name/
self.raw_data_path = None #default will be DATA/{}D/
    expr_name/expr_name.json


self.bound = rmax # maximum value a coordinate will have


self.run_mode = 0 if runmode is None else runmode # see
    run_mode_dict for options

self.steps = 200 if steps is None else steps
self.alternation_constant = 1 # number of steps of top down
     and bottom up to run as you alternate


self.current_cone_chain = some_cone_chain
self.batch_mode = batchmode

self.num_gen = numgen
```

```
        self.loaded = False if self.current_cone_chain is None else
            True
        if not batchmode:
                self.steps = 100 if steps is None else steps #
                    default number of steps to run between printouts
                self.begin()


def begin(self):
        running = True
        while running:
                main_menu_choice = self.main_menu()
                """{  0: text_run_experiment,
                              1: text_create,
                              2: text_load_continue,
                              3: text_load_copy,
                              4: text_load_initial,
                              5: text_summary,
                              9: text_save_exit}"""
                #0
                if ConeConjectureTester.main_menu_dict_loaded[
                    main_menu_choice] == ConeConjectureTester.
                    text_run_experiment:
                        # run a loaded experiment
```

```
            self.run_mode = self.run_mode_menu()
            self.run_experiment()

        #1
        elif ConeConjectureTester.main_menu_dict_loaded[
            main_menu_choice] == ConeConjectureTester.
            text_create:
                # make a new experiment
                self.create_experiment()

        #2
        elif ConeConjectureTester.main_menu_dict_loaded[
            main_menu_choice] == ConeConjectureTester.
            text_load_continue:
                # load an old experiment
                self.load_experiment()

        #3
        elif ConeConjectureTester.main_menu_dict_loaded[
            main_menu_choice] == ConeConjectureTester.
            text_load_copy:
                # load an old experiment
                self.load_experiment()
```

```
            #ask for new naem
            self.ask_experiment_name()
            #update all paths
            self.update_paths()
            # save data to new spot
            self.save_file()

        #4
        elif ConeConjectureTester.main_menu_dict_loaded[
            main_menu_choice] == ConeConjectureTester.
            text_load_initial:
            # load an old experiment
            self.load_experiment(initial_condition=True)

        #5
        elif ConeConjectureTester.main_menu_dict_loaded[
            main_menu_choice] == ConeConjectureTester.
            text_summary:
            # load an old experiment
            self.current_cone_chain.output_to_terminal()
            self.save_summary()

        #6
```

```
elif ConeConjectureTester.main_menu_dict_loaded[
    main_menu_choice] == ConeConjectureTester.
    text_print_graphs:
        # print the graphs
        self.print_graphs()


#7:
elif ConeConjectureTester.main_menu_dict_loaded[
    main_menu_choice] == ConeConjectureTester.
    text_display_all_details:
        self.current_cone_chain.output_to_terminal()
        self.current_cone_chain.chain_details()

#8:
elif ConeConjectureTester.main_menu_dict_loaded[
    main_menu_choice] == ConeConjectureTester.
    text_manual_input:
        self.manual_create_experiment()



#1337
```

```
                elif ConeConjectureTester.main_menu_dict_loaded[
                    main_menu_choice] == ConeConjectureTester.
                    text_save_exit:
                        # exit
                        self.check_loaded()
                        if self.loaded:
                                self.current_cone_chain.
                                    output_to_terminal()
                                print('Printing graphs...')
                                self.current_cone_chain.
                                    generate_hilbert_graphs(self.
                                    directory, self.experiment_name)
                                print("Saving summary...")
                                self.save_summary()
                                print('Saving to file...')
                                self.save_file()

                        running = False


def print_graphs(self):
        ''' Generates and saves graphs to the appropriate directory
            .'''
```

```
        print('Printing graphs to {}'.format(self.directory))
        self.current_cone_chain.generate_hilbert_graphs(self.
            directory, self.experiment_name)
        #experiment_io_tools.pause()


def main_menu(self):
        """ (Terminal UI) Main menu"""
        if self.loaded:
                return experiment_io_tools.menu(ConeConjectureTester
                    .main_menu_dict_loaded, ConeConjectureTester.
                    text_main_title, self.file_setting_string()+
                    experiment_io_tools.separator())
        else:
                return experiment_io_tools.menu(ConeConjectureTester
                    .main_menu_dict_initial, ConeConjectureTester.
                    text_main_title)



def run_mode_menu(self):
        """ (Terminal UI) Asks user for the mode they want to run.
            """
        self.check_loaded()
        if self.loaded:
```

```
                return experiment_io_tools.menu(ConeConjectureTester
                    .run_mode_dict, "Choose Run Mode for '{}'".format
                    (self.experiment_name))
        else:
                return 0


def deduce_run_mode(self):
        """ (Internal logic function) Determines what mode a loaded
            experiment has even
                if runmode was not saved. """
        self.check_loaded()
        if self.loaded:
                if self.batch_mode:
                        top_down_count = len(self.current_cone_chain.
                            top_sequence)
                        bottom_up_count = len(self.current_cone_chain
                            .bottom_sequence)
                        if top_down_count > bottom_up_count and
                            bottom_up_count == 1:
                                self.run_mode=1 # topdown
                                return
                        elif top_down_count < bottom_up_count and
                            top_down_count ==1:
```

```
                                self.run_mode = 2 # bottomup
                                return
                            elif top_down_count >= bottom_up_count -1 and
                                top_down_count <= bottom_up_count + 1:
                                self.run_mode = 3
                                return
                        self.run_mode = self.run_mode_menu()
                else:
                        return 0

    def file_setting_string(self):
            """ returns a string that shows settings """
            settings = "Current Settings: \n\tDimension = {}\n".format(
                self.dimension)
            settings += "\tExperiment Name = {}\n".format(self.
                experiment_name)
            settings += "\tRaw Data path = {}\n".format(self.
                raw_data_path)
            return settings


    def generate_cones(self):
            """ Takes input from user either by script or by the UI and
```

```
            generates cones """
        if self.bound is None or not self.batch_mode:
            self.ask_bound()
        if self.num_gen is None or not self.batch_mode:
            self.ask_num_gen()
        print("Generating New Cones...")
        cones_unacceptable = True
        while cones_unacceptable:
            outer_cone = cone_tools.generate_cone(self.dimension
                , rmax=self.bound, numgen=self.num_gen)
            inner_cone = cone_tools.generate_inner_cone(
                outer_cone, rmax=self.bound, numgen=self.num_gen)
            cones_unacceptable = (inner_cone.rays_list() ==
                outer_cone.rays_list())
        self.current_cone_chain = cone_chain.ConeChain(inner_cone,
            outer_cone)
        print("Geneation successful!")

    def create_experiment(self):
        """Asks the relevant information then checks to see if
            randomly generated
        or user inputted"""
        experiment_io_tools.new_screen(ConeConjectureTester.
```

```
            text_create)
        self.ask_dimension()
        self.ask_experiment_name()
        self.update_paths(self.experiment_name)

        self.generate_cones()

        self.save_file("Initial Conditions")
        self.save_file()
        self.check_loaded()

        experiment_io_tools.pause()



def manual_create_experiment(self):
        """ PUre UI version of creating experiment, user manually
            inputs everything """
        experiment_io_tools.new_screen(ConeConjectureTester.
            text_manual_input)
        self.ask_dimension()
        self.ask_experiment_name()
        self.update_paths(self.experiment_name)
```

```
        self.manual_input()

        self.save_file("Initial Conditions")
        self.save_file()
        self.check_loaded()

        experiment_io_tools.pause()


def batch_create_experiment(self,experiment_name=None):
        """Creates experiment without running it. Does not involve
            TUI unless there are errors."""
        if experiment_name==None:
                self.update_paths(self.experiment_name)
        else:
                self.update_paths(experiment_name)

        if self.bound is None:
                self.ask_bound()
        if self.num_gen is None:
                self.ask_num_gen()

        self.check_loaded()
```

```
        if not self.loaded:
                self.generate_cones()

        self.save_file("Initial Conditions")
        self.save_file()



def load_experiment(self,initial_condition=False):
        """ Terminal UI for loading a experiment. """
        if self.batch_mode is False:
                experiment_io_tools.new_screen(ConeConjectureTester.
                    text_load_continue)

        user_choice = -1337
        accept_expr_choice = False
        while (not accept_expr_choice) and (self.batch_mode is
            False):
                self.ask_dimension()
                try:
                        # get the list of experiments and sort it
                        possible_experiment_names = os.listdir("DATA
                            /{}d/".format(self.dimension))
```

```
            possible_experiment_names.sort()
            # Ask user by making a menu...
            choose_experiment_menu = {i+1:
               possible_experiment_names[i] for i in
               range(len(possible_experiment_names)) }
            choose_experiment_menu.update({-1:"... back
               to main menu..."})
            user_choice = experiment_io_tools.menu(
               choose_experiment_menu)
            if user_choice <> -1:
                   self.experiment_name =
                       choose_experiment_menu[user_choice
                       ]
                   accept_expr_choice =
                       experiment_io_tools.query_yes_no("
                       Your choice is '{}'. \n\tAccept?".
                       format(self.experiment_name))
            else:
                   return
    except:
            experiment_io_tools.pause("Problem finding or
                opening folder, likely non-existent path
               .")
```

```
        if user_choice <> -1:
                self.update_paths(self.experiment_name)
                self.load_file(initial_condition)
                self.check_loaded()
                experiment_io_tools.pause()
                self.current_cone_chain.output_to_terminal()


def load_file(self,initial_condition=False,custom_name=None):
        """ Loads a json file into the current_cone_chain
                Args:
                        initial_condition (boolean):  True if seeking
                             only the initial cones,
                                False if seeking the whole file (
                                    Default)
                        custom_name (string): Sets the name of the
                            experiment to be loaded
        """

        # if no custom name, then simply load with current settings
            (TUI version)
        # if there is a custom name, update paths to the
```

```
    appropriate experiment name then attempt to load.

self.update_paths()
if custom_name is None:
       filepath = self.raw_data_path
else:
       filepath = "DATA/{}d/{}/{}.json".format(self.
           dimension, custom_name,custom_name)


print("Loading file: {}".format(filepath))
try:
       with open(filepath, 'r') as fp:
              if initial_condition:
                     self.current_cone_chain = json.load(fp
                         , cls=cone_chain.
                         ConeChainInitialConditionExtractor
                         )
                     if not self.batch_mode or self.
                         experiment_name is None:
                            self.ask_experiment_name()
                     self.update_paths()


              else:
```

```
                                self.current_cone_chain = json.load(fp
                                    , cls=cone_chain.ConeChainDecoder)
                print('\tloading successful...')
        except:
                print('\tA file loading error has occured.')
        self.save_file()

def save_file(self, filename=None):
        '''Saves the data from the current cone chain to a file in
            self.directory
                Args: filename (string): The name of the file with
                    no extensions.
        '''

        self.update_paths()
        if filename is None:
                file_path = self.raw_data_path
        else:
                file_path = self.directory + filename +".json"

        with open(file_path, 'w') as fp:
                json.dump(self.current_cone_chain, fp, cls=
                    cone_chain.ConeChainEncoder,sort_keys=True,
```

```
                    indent=4, separators=(',', ': '))

def check_loaded(self):
        ''' Verifies that current_cone_chain is loaded by verifying
            it's not "none '''
        if self.current_cone_chain is None:
                self.loaded = False
        else:
                self.loaded = True



def update_paths(self, new_expr_name=None):
        ''' A function to update self.directory and self.
           raw_datapath
                Args: new_expr_name (string): The name of the
                    experiment we wish to save/load from
        '''

        # logic to determine the file name in a clear way:
        # if the experiment_name is not set and there's no custom
           name provided:
        if new_expr_name is None and self.experiment_name is None:
                self.ask_experiment_name()
```

```
elif new_expr_name is not None:
        self.experiment_name = new_expr_name

# usually set, but just in case...
if self.dimension is None:
        self.ask_dimension()

#set the directory and raw data path to be new
self.directory = "DATA/{}d/".format(self.dimension) + self.
    experiment_name + "/"
self.raw_data_path = self.directory + self.experiment_name
    + ".json"

#make the directory if it doesn't exist
try:
        os.makedirs(self.directory, 0755)
except:
        NotImplemented
        #print("Error making directory {}".format(self.
            directory))
#print("DEBUG: raw_data_path = {}".format(self.
    raw_data_path))
```

```
def save_summary(self,folder=None,experiment_name=None):
        """ Writes current summary of data to a text file."""
        summary_name = "Data Summary.txt"
        try:
                os.makedirs(self.directory, 0755)
        except:
                NotImplemented
        fileobj = open(self.directory + summary_name , "w")
        fileobj.write("inner_cone has generators: \n{}\n".format(
            self.current_cone_chain.inner_cone.rays_list()))
        fileobj.write("outer_cone has generators: \n{}\n".format(
            self.current_cone_chain.outer_cone.rays_list()))
        fileobj.write("\tsequence_complete = {}\n".format(self.
            current_cone_chain.sequence_complete))
        fileobj.write("\ttop_sequence has length {}\n".format(len(
            self.current_cone_chain.top_sequence)))
        fileobj.write("\tbottom_sequence has length {}\n".format(
            len(self.current_cone_chain.bottom_sequence)))
        fileobj.write("\tcone_poset_chain has length {}\n".format(
            len(self.current_cone_chain.cone_poset_chain)))
        fileobj.close()

def ask_experiment_name(self):
```

```
        """ (Terminal UI) Asks user for the experiment name and
            sets self.experiment_name"""
        if self.experiment_name is not None:
                accept_name = not experiment_io_tools.query_yes_no("
                    Current experiment name set to be {}. Change
                    current setting? ".format(self.experiment_name))
        else:
                accept_name = False
        while not accept_name:
                self.experiment_name = str(raw_input("Experiment
                    Name: "))
                accept_name = experiment_io_tools.query_yes_no("\
                    tYou entered '{}'. Accept?".format(self.
                    experiment_name))


def ask_dimension(self):
        """ (Terminal UI) Asks user for the dimension and sets self
            .dimension"""
        if self.dimension <> None:
                valid_dimension = experiment_io_tools.query_yes_no("
                    Current dimension set to be {}. Keep current
                    setting? ".format(self.dimension))
```

```
        else:
                valid_dimension = False

        while not valid_dimension:
                self.dimension = experiment_io_tools.ask_int("Enter
                    dimension: ")
                if self.dimension > 1:
                        valid_dimension = True

def ask_bound(self):
        """ (Terminal UI) Asks user for the bound of the random
            number generator and sets self.bound"""
        if self.bound > 0:
                accept_bound = experiment_io_tools.query_yes_no("
                    Current random numbers bound at +/-{}. Keep
                    current setting? ".format(self.bound))
        else:
                accept_bound = False

        while not accept_bound:
                self.bound = experiment_io_tools.ask_int("Enter the
                    absolute bound for coordinates in these vectors:
                    ")
```

```
                if self.bound > 0:
                        accept_bound = True


def ask_num_gen(self):
        """ (Terminal UI) Asks user for the number of generators
            needed"""
        if self.num_gen is not None:
                accept_num = experiment_io_tools.query_yes_no("
                    Current number of generators = {}. Keep settings?
                    ".format(self.num_gen))
        else:
                accept_num = False

        while not accept_num:
                self.num_gen = experiment_io_tools.ask_int("Enter
                    the number of generators for each cone: ")
                if self.num_gen >= self.dimension:
                        accept_num = True
                else:
                        print("Please input a number greater or equal
                             to the dimension chosen for a full
                            dimensional cone.")
```

```
def ask_steps(self):
        """ (Terminal UI) Asks user for the number of times to run
            the algorithms between saving and sets self.steps"""
        if self.steps > 0:
                accept_steps = experiment_io_tools.query_yes_no("
                    Currently '{}' is set to run for [{}] steps. Keep
                     current setting? ".format(ConeConjectureTester.
                    run_mode_dict[self.run_mode],self.steps))
        else:
                accept_steps = False

        while not accept_steps:
                self.steps = experiment_io_tools.ask_int("Enter the
                    number of steps to run {}: ".format(
                    ConeConjectureTester.run_mode_dict[self.run_mode
                    ]))
                if self.steps > 0:
                        accept_steps = True

def ask_alternation_constant(self):
        """ (Terminal UI) Asks user for the number of times to run
```

```
            top_down/bottom_up before alternating
                CURRENTLY NOT USED... DEFAULT SET TO 1.
            """

            if self.steps > 0:
                accept_alternation_constant = experiment_io_tools.
                    query_yes_no("'{}' is currently set to switch
                    every {} step(s). Keep current setting? ".format(
                    ConeConjectureTester.run_mode_dict[self.run_mode
                    ],self.alternation_constant,self.steps))
            else:
                accept_alternation_constant = False

            while not accept_alternation_constant:
                self.alternation_constant = experiment_io_tools.
                    ask_int("Enter the alternating constant: ".format
                    (ConeConjectureTester.run_mode_dict[self.run_mode
                    ]))
                if self.stalternation_constanteps > 0:
                    accept_alternation_constant = True


def run_experiment(self):
```

```
""" Runs the experiment using current settings
        Assumes:
        1) current_cone_chain is not empty
        2) If you're in batch mode, self.run_mode() is set.
"""
# ask run_mode (1 - Top Down, 2 - Bottom up, 3 -
    Alternating?)
if self.run_mode is None or self.run_mode == 0 or self.
    run_mode not in ConeConjectureTester.run_mode_dict.keys
    ():
        self.deduce_run_mode()


# verify number of steps to run between printing/saving
    data
if self.batch_mode is False:
        self.ask_steps()
        experiment_io_tools.new_screen(self.experiment_name)
        user_continue = experiment_io_tools.query_yes_no("
            Begin running '{}'?".format(ConeConjectureTester.
            run_mode_dict[self.run_mode]))
else:
        user_continue = True
```

```
# Beginning running experiment
original_count = self.current_cone_chain.number_of_steps()
while user_continue:
        print("\trunning '{}' for {} steps...".format(
            ConeConjectureTester.run_mode_dict[self.run_mode
            ],self.steps))
        steps_ran_this_sitting = self.current_cone_chain.
            number_of_steps()-original_count
        if ConeConjectureTester.run_mode_dict[self.run_mode]
             == ConeConjectureTester.text_top_down:
                self.current_cone_chain.top_down(self.steps)

        elif ConeConjectureTester.run_mode_dict[self.
            run_mode] == ConeConjectureTester.text_bottom_up:
                self.current_cone_chain.bottom_up(self.steps)

        elif ConeConjectureTester.run_mode_dict[self.
            run_mode] == ConeConjectureTester.
            text_alternating:
                for step_counter in range(self.steps):
                        if sage.all.mod(step_counter, 2*self.
                            alternation_constant) < self.
```

```
                    alternation_constant:
                        self.current_cone_chain.
                            top_down(self.
                            alternation_constant)
                else:
                        self.current_cone_chain.
                            bottom_up(self.
                            alternation_constant)

# At the end of each loop we save some data...
self.print_graphs()
print("Saving summary...")
self.save_summary()
print('Saving to file...')
self.save_file()

if self.batch_mode is False:
        user_continue = experiment_io_tools.
            query_yes_no("Completed {} steps this run
             so far.\n\tSaved data and printed graph.
             Continue?".format(self.
            current_cone_chain.number_of_steps()-
            original_count))
```

```
                else:
                        user_continue = False



def manual_input(self):
        """ Terminal UI function to allow for manual input of
            experiments. """
        continueinput = True

        while continueinput:
                outer_generators = []

                while True:
                        try:
                                experiment_io_tools.new_screen("
                                    Entering Extremal Generators of
                                    Outer Cone")

                                print("Current list of extremal
                                    generators for the OUTER cone: {}"
                                    .format(outer_generators))
                                experiment_io_tools.printseparator()
```

```
handle = raw_input("Please enter an
    extremal generator \"x_1,...,x_d\"
     of the outer cone without quotes,
     \nor type \"finish\" when done: "
    )
if str(handle).lower() == "finish":
        if len(outer_generators) +1 <
            self.dimension:
                print("Not enough
                    generators for a
                    full dimensional
                    cone!")
        else:
                break
else:
        handlelist = [ int(i) for i in
             handle.split(",")]
        if len(handlelist) <> self.
            dimension:
                print("Incorrect
                    dimension.")
        else:
                outer_generators.append
```

```
                                    (handlelist)
        except Exception as inputerror:
                print("Input error detected: {}".
                    format(inputerror))

temp_outer = sage.all.Polyhedron(rays=
    outer_generators,backend='normaliz')

inner_generators = []
while True:
        try:
                experiment_io_tools.new_screen("
                    Entering Extremal Generators of
                    Inner Cone")

                print("Current list of extremal
                    generators for the INNER cone: {}"
                    .format(inner_generators))
                experiment_io_tools.printseparator()
                handle = raw_input("Please enter an
                    extremal generator \"x_1,...,x_d\"
                     of the inner cone without quotes,
                     \nor type \"finish\" when done: "
```

```
    )
if handle.lower() == "finish":
        if len(inner_generators) <
            self.dimension:
                print("Not enough
                    generators for a
                    full dimensional
                    cone!")
                break
        else:
                break
else:
        handlelist = [int(i) for i in
            handle.split(",")]
        if len(handlelist) <> self.
            dimension:
                print("Incorrect
                    dimension.")
        elif temp_outer.contains(
            handlelist):
                inner_generators.append
                    (handlelist)
        else:
```

```
                                        print("{} is not
                                            contained in outer
                                            cone!".format(
                                            handlelist))
                        except Exception as inputerror:
                                print("Input error detected: {}".
                                    format(inputerror))

                temp_inner = sage.all.Polyhedron(rays=
                    inner_generators,backend='normaliz')
                continueinput = not cone_tools.sanity_check(
                    temp_inner,temp_outer)
                if continueinput:
                        print("Some error occured, please do this
                            again.")
        self.current_cone_chain = cone_chain.ConeChain(temp_inner,
            temp_outer)

##################################################
# DEBUG FUNCTION: Recalculate all Hilbert Basis #
##################################################

def recalc_experiment(self):
```

```
print("Recalculating '{}'".format(self.experiment_name))

totalsteps = self.current_cone_chain.number_of_steps()
print("There are {} cones.".format(totalsteps))
self.current_cone_chain.recalc()
```

Chapter 6

Computational Results

6.1 Data Collection Technique

We examine cones in dimension 4 and 5:

dimension	number of extremal generators	$k \in \mathbb{Z} : \lvert x_i \rvert < k$
4	(4,5)	2
5	(5,6)	(1,2)

Table 6.1: Conditions tested.

The code was written in an object oriented fashion which allows for use in other python scripts. Thus, the experiments are generated in batch; each experiment is named "{} generators {} bound {letter}" where the first "{}" is the number of extremal generators and the latter is the bound on the absolute value on each coordinate. Files named with the same {letter} will have the same initial conditions. We run a fixed number of steps at the beginning, with a separate script that continues any existing experiment a fixed number of steps

or loop for a fixed amount of time. The details of how the script looks can be found in the appendix.

We collect and record the number of steps taken for top down as well as bottom up for a particular pair of cones. Then we examine the Hilbert basis in the algorithmically generated poset chains. To study the complexity of the cones in the poset chain, we record the number of elements in the Hilbert basis, or $\#\mathsf{Hilb}(C)$ and also examine the Euclidean norm of the longest vector in the Hilbert basis of C. This information is presented in graphical form, using `figure` and `plot` functions in the standard Python package `pylab`.

Note that the way the graphs are arranged, the top down sequence begins with the ”top” cone, or the ”outer” cone. In contrast, the bottom up sequence begins with ”bottom” or ”inner” cone.

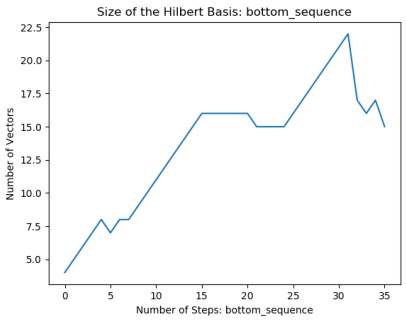
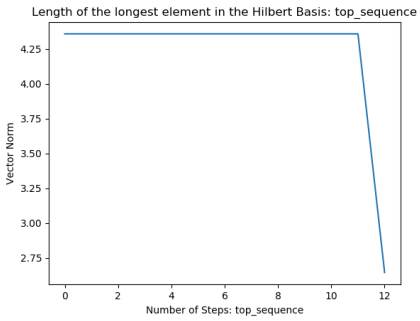
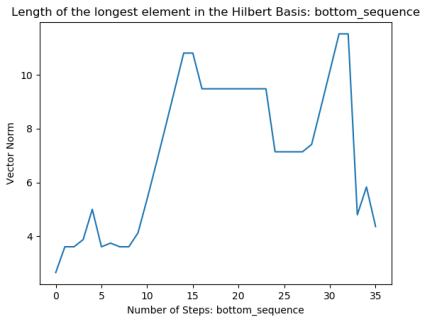
Interestingly, some cones seem to terminate only on one of the algorithms!

6.2 Data in dimension 4

6.2.1 4 generators 2 bound A

Initial Conditions:

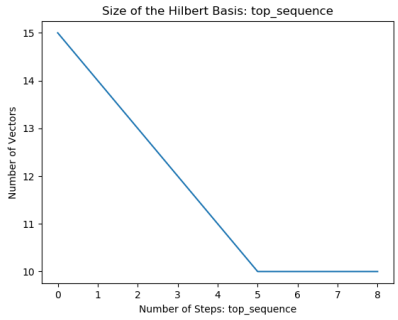
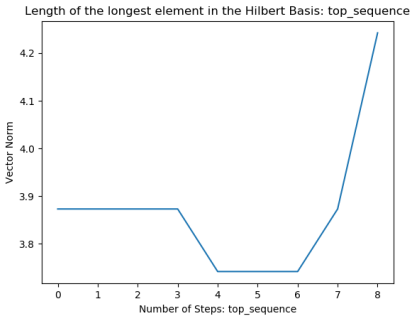
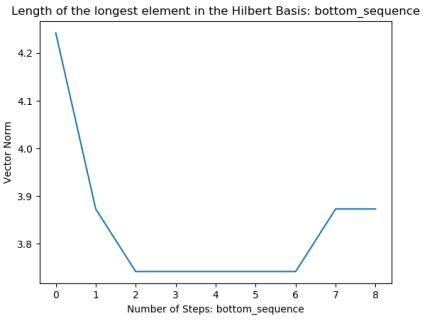
```
inner_cone has generators:
[[0, 0, -2, 1], [1, -1, -1, 2], [1, 0, -1, 1], [1, 1, -2, 1]]
outer_cone has generators:
[[-1, -2, 0, 2], [0, 0, -2, 1], [1, -2, 0, 1], [2, 2, -2, 1]]
```

Top Down	Bottom Up
sequence_complete = True top_sequence has length 13 bottom_sequence has length 1 cone_poset_chain has length 13	sequence_complete = True top_sequence has length 1 bottom_sequence has length 36 cone_poset_chain has length 36
	
	

6.2.2 4 generators 2 bound B

Initial Conditions:

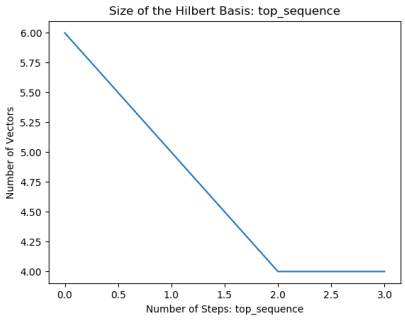
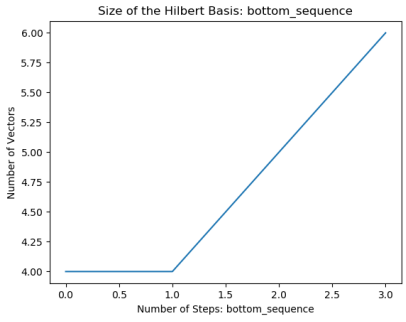
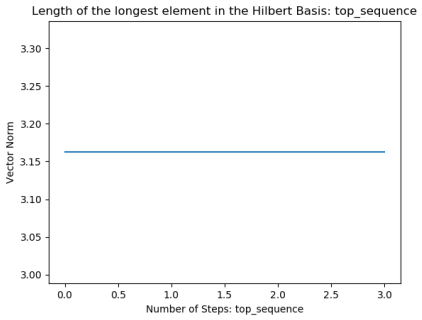
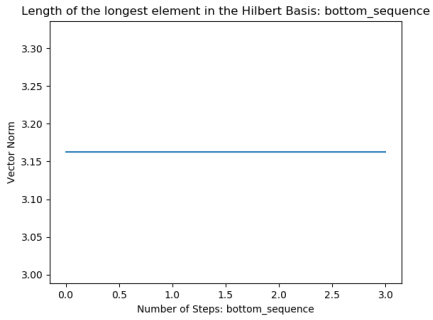
```
inner_cone has generators:
[[-2, 1, -1, 1], [0, -1, -1, 2], [1, -1, -1, 1], [2, 0, -1, 2]]
outer_cone has generators:
[[-2, -1, 0, 2], [-2, 1, -1, 1], [1, -1, -1, 1], [2, 0, -1, 2]]
```

Top Down	Bottom Up
sequence_complete = True top_sequence has length 9 bottom_sequence has length 1 cone_poset_chain has length 9	sequence_complete = True top_sequence has length 1 bottom_sequence has length 9 cone_poset_chain has length 9
	
	

6.2.3 4 generators 2 bound C

Initial Conditions:

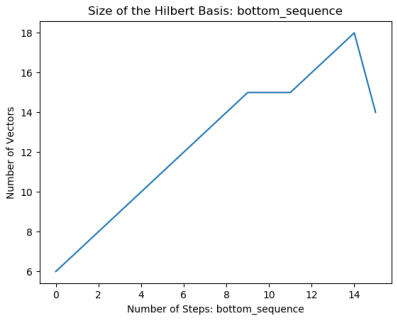
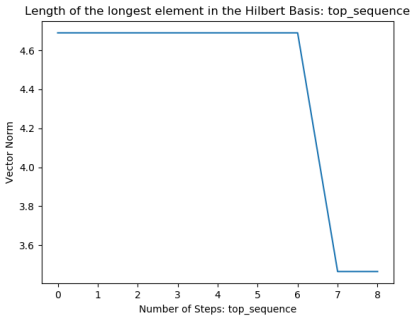
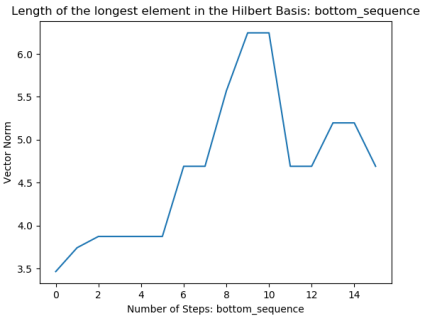
```
inner_cone has generators:
[[-1, -1, 0, 1], [0, 0, -1, 2], [0, 0, 0, 1], [2, 1, -1, 2]]
outer_cone has generators:
[[-1, -1, 0, 1], [0, 0, -2, 1], [1, 1, 1, 1], [2, 1, -1, 2]]
```

Top Down	Bottom Up
sequence_complete = True top_sequence has length 4 bottom_sequence has length 1 cone_poset_chain has length 4	sequence_complete = True top_sequence has length 1 bottom_sequence has length 4 cone_poset_chain has length 4
	
	

6.2.4 4 generators 2 bound D

Initial Conditions:

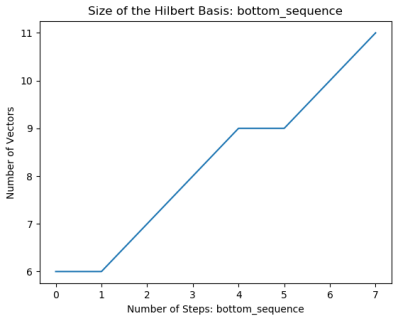
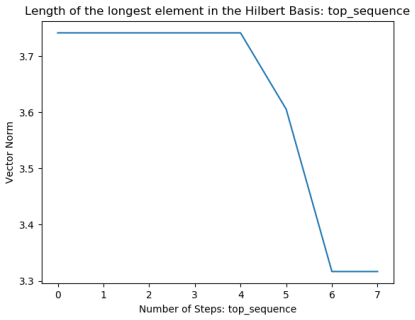
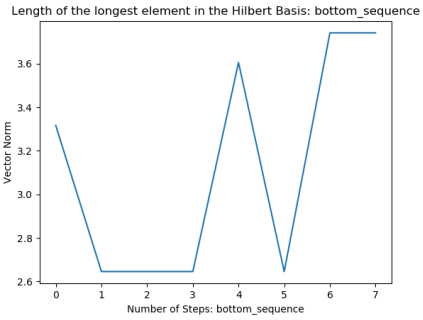
```
inner_cone has generators:
[[-2, -1, 1, 2], [-1, 0, 1, 2], [0, -1, 0, 2], [0, -1, 1, 2]]
outer_cone has generators:
[[-2, -1, 1, 2], [-1, 0, 1, 2], [0, 0, -1, 1], [2, -2, 1, 2]]
```

Top Down	Bottom Up
sequence_complete = True top_sequence has length 9 bottom_sequence has length 1 cone_poset_chain has length 9	sequence_complete = True top_sequence has length 1 bottom_sequence has length 16 cone_poset_chain has length 16
	
	

6.2.5 4 generators 2 bound E

Initial Conditions:

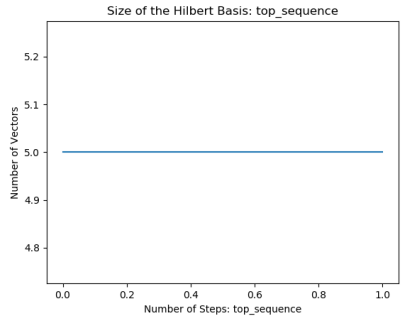
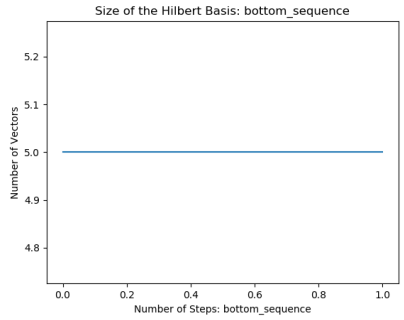
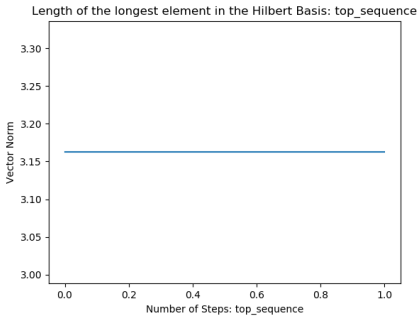
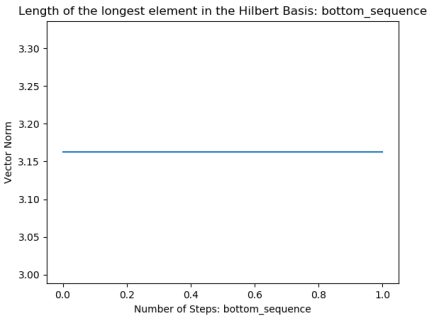
```
inner_cone has generators:
[[-1, 0, -1, 1], [-1, 1, 1, 2], [0, -1, 1, 1], [0, 0, 0, 1]]
outer_cone has generators:
[[-1, 0, -1, 1], [-1, 2, 1, 1], [0, -1, 1, 1], [1, 0, 1, 2]]
```

Top Down	Bottom Up
sequence_complete = True top_sequence has length 8 bottom_sequence has length 1 cone_poset_chain has length 8	sequence_complete = True top_sequence has length 1 bottom_sequence has length 8 cone_poset_chain has length 8
	
	

6.2.6 4 generators 2 bound F

Initial Conditions:

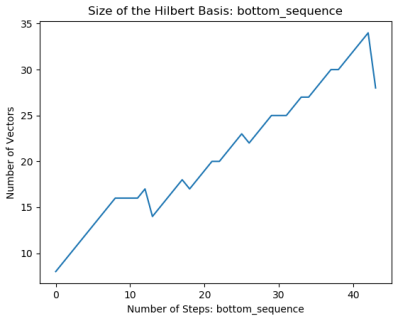
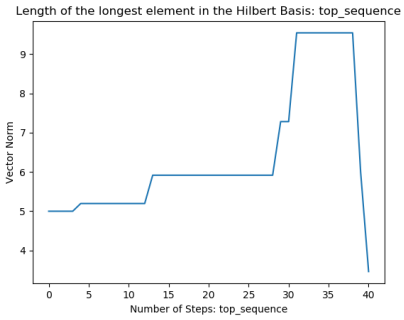
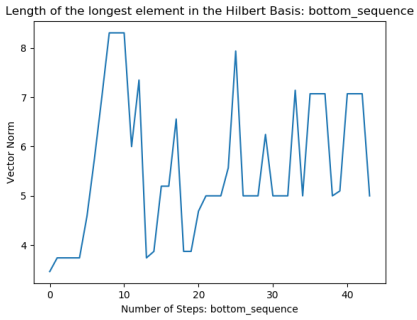
```
inner_cone has generators:
[[-1, 0, -1, 1], [1, -1, -2, 1], [1, -1, 0, 2], [1, -1, 2, 2]]
outer_cone has generators:
[[-1, 0, -1, 1], [0, 0, 2, 1], [1, -1, -2, 1], [1, -1, 2, 2]]
```

Top Down	Bottom Up
sequence_complete = True top_sequence has length 2 bottom_sequence has length 1 cone_poset_chain has length 2	sequence_complete = True top_sequence has length 1 bottom_sequence has length 2 cone_poset_chain has length 2
	
	

6.2.7 4 generators 2 bound G

Initial Conditions:

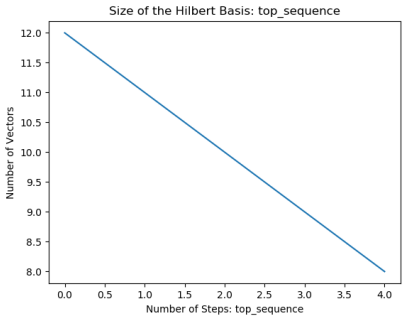
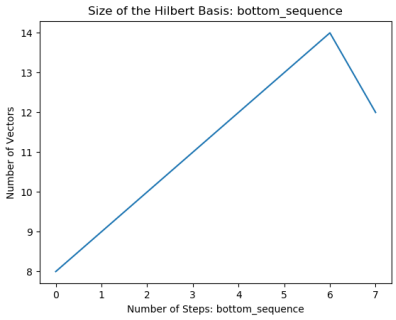
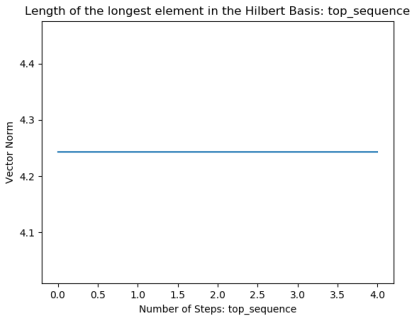
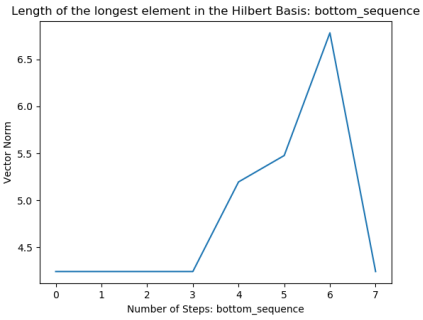
```
inner_cone has generators:
[[-1, -1, 0, 2], [-1, 1, -1, 2], [-1, 1, 0, 2], [1, 0, -2, 2]]
outer_cone has generators:
[[-2, -2, 1, 2], [-1, 1, -2, 2], [-1, 2, 2, 1], [1, 0, -2, 2]]
```

Top Down	Bottom Up
sequence_complete = True top_sequence has length 41 bottom_sequence has length 1 cone_poset_chain has length 41	sequence_complete = True top_sequence has length 1 bottom_sequence has length 44 cone_poset_chain has length 44
	
	

6.2.8 4 generators 2 bound H

Initial Conditions:

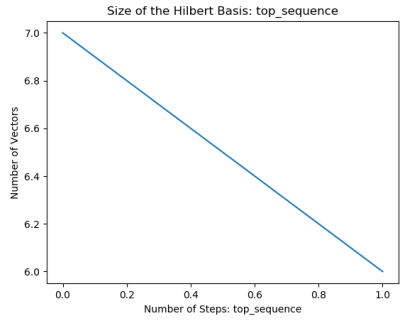
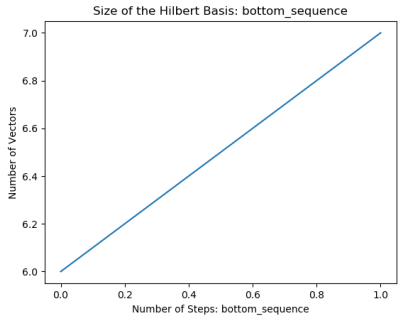
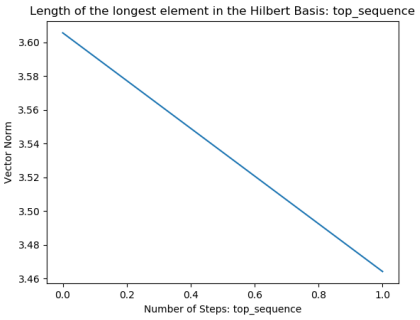
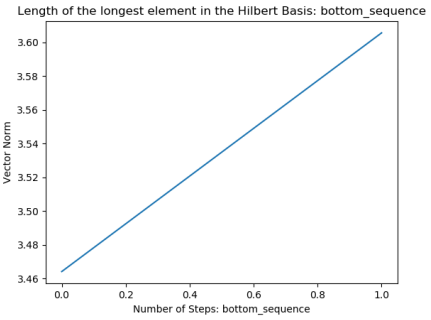
```
inner_cone has generators:
[[-2, -1, -1, 2], [1, -1, -1, 2], [2, -1, -2, 2], [2, 0, -1, 2]]
outer_cone has generators:
[[-2, -1, -1, 2], [-1, -2, 0, 2], [2, -1, -2, 2], [2, 0, -1, 2]]
```

Top Down	Bottom Up
sequence_complete = True top_sequence has length 5 bottom_sequence has length 1 cone_poset_chain has length 5	sequence_complete = True top_sequence has length 1 bottom_sequence has length 8 cone_poset_chain has length 8
	
	

6.2.9 4 generators 2 bound I

Initial Conditions:

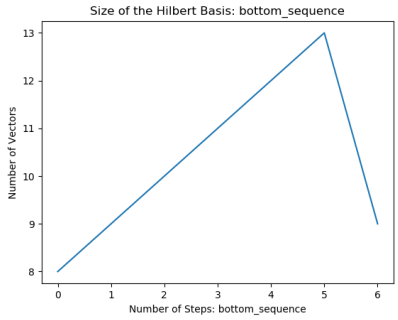
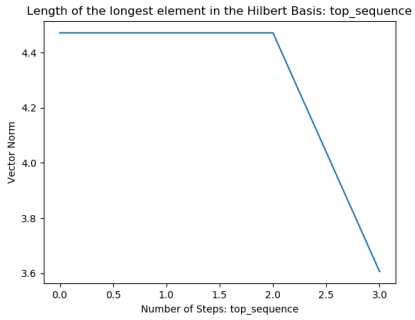
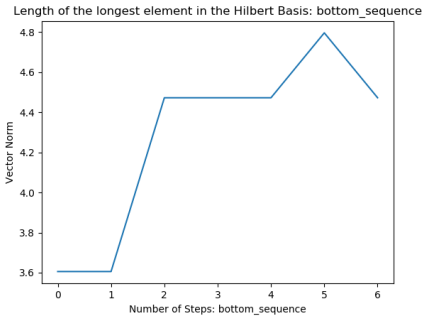
```
inner_cone has generators:
[[-1, -2, 0, 2], [0, -2, 1, 2], [1, -1, 1, 1], [2, 1, 1, 2]]
outer_cone has generators:
[[-2, -2, -1, 2], [0, -2, 1, 2], [1, -1, 1, 1], [2, 1, 1, 2]]
```

Top Down	Bottom Up
sequence_complete = True top_sequence has length 2 bottom_sequence has length 1 cone_poset_chain has length 2	sequence_complete = True top_sequence has length 1 bottom_sequence has length 2 cone_poset_chain has length 2
	
	

6.2.10 4 generators 2 bound J

Initial Conditions:

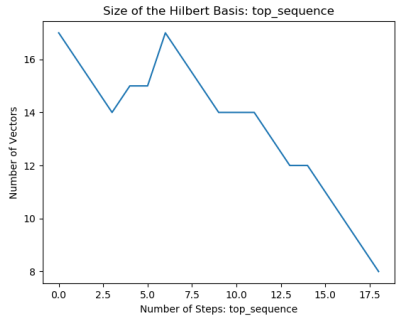
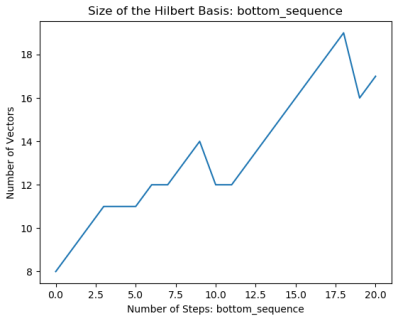
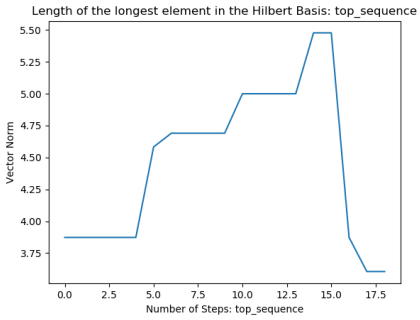
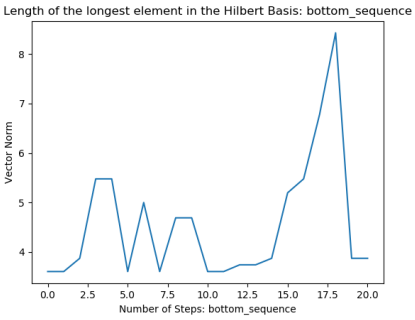
```
inner_cone has generators:
[[1, -2, 1, 1], [2, -2, -1, 1], [2, -1, -1, 2], [2, -1, 2, 2]]
outer_cone has generators:
[[1, -2, 1, 1], [1, 0, 1, 1], [1, 1, -1, 2], [2, -2, -1, 1]]
```

Top Down	Bottom Up
sequence_complete = True top_sequence has length 4 bottom_sequence has length 1 cone_poset_chain has length 4	sequence_complete = True top_sequence has length 1 bottom_sequence has length 7 cone_poset_chain has length 7
	
	

6.2.11 5 generators 2 bound A

Initial Conditions:

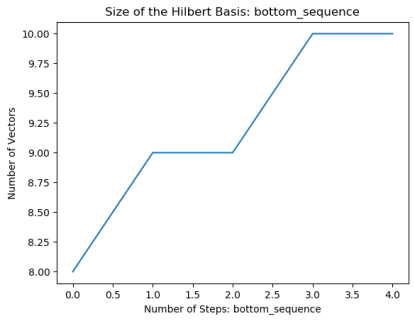
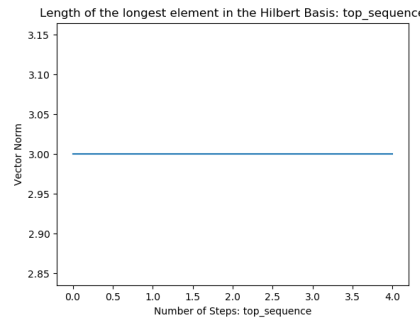
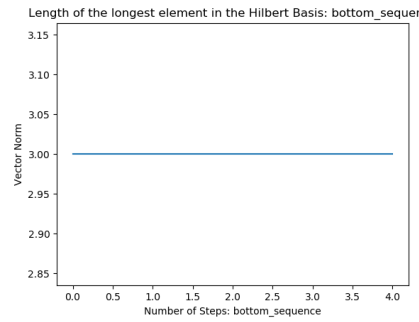
```
inner_cone has generators:
[[-1, 1, 0, 1], [0, -1, 1, 2], [0, 0, 0, 1], [1, 0, 0, 2], [1, 0, 2, 1]]
outer_cone has generators:
[[-1, 1, 0, 1], [0, -1, 1, 2], [1, 0, -1, 2], [1, 0, 2, 1], [1, 2, -2, 2]]
```

Top Down	Bottom Up
sequence_complete = True top_sequence has length 19 bottom_sequence has length 1 cone_poset_chain has length 19	sequence_complete = True top_sequence has length 1 bottom_sequence has length 21 cone_poset_chain has length 21
	
	

6.2.12 5 generators 2 bound B

Initial Conditions:

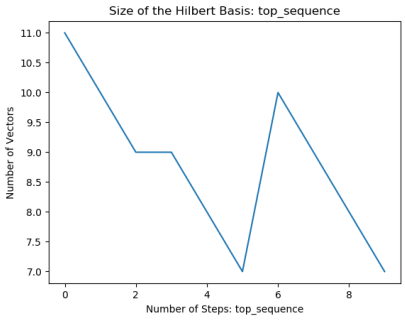
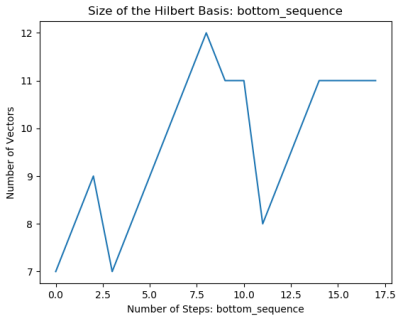
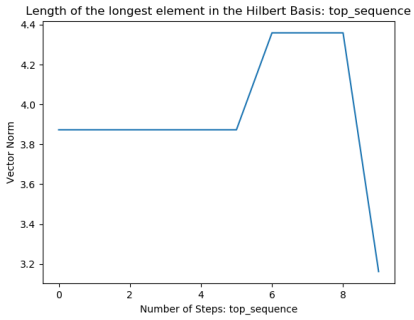
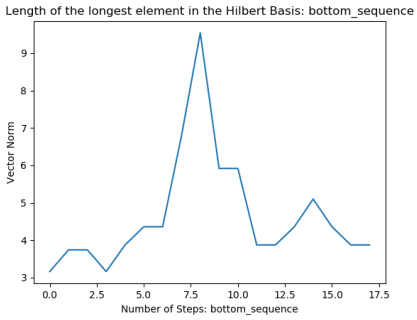
```
inner_cone has generators:
[[-2, 1, 0, 2], [-1, 0, -1, 1], [-1, 0, 1, 1], [-1, 0, 2, 2], [1, 0, -1,
    2]]
outer_cone has generators:
[[-2, 0, 1, 1], [-2, 1, 0, 2], [-1, 0, -2, 1], [-1, 0, 2, 2], [1, 0, -1,
    1]]
```

Top Down	Bottom Up
sequence_complete = True top_sequence has length 5 bottom_sequence has length 1 cone_poset_chain has length 5	sequence_complete = True top_sequence has length 1 bottom_sequence has length 5 cone_poset_chain has length 5
	
	

6.2.13 5 generators 2 bound C

Initial Conditions:

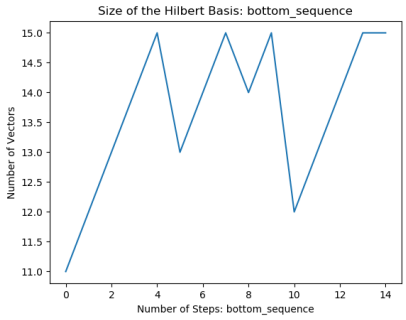
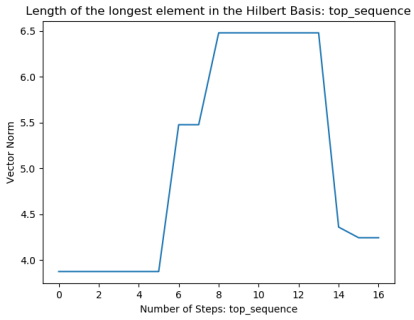
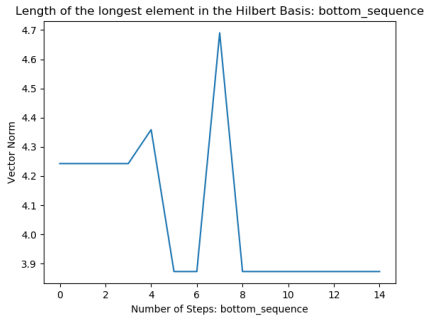
```
inner_cone has generators:
[[-1, -1, 0, 1], [-1, 0, 1, 2], [0, -1, 2, 2], [0, 1, 1, 2], [1, -2, 2,
    1]]
outer_cone has generators:
[[-2, 0, 1, 2], [-1, -1, 0, 1], [1, -2, 2, 1], [1, 0, 2, 1], [1, 2, 1, 1]]
```

Top Down	Bottom Up
sequence_complete = True top_sequence has length 10 bottom_sequence has length 1 cone_poset_chain has length 10	sequence_complete = True top_sequence has length 1 bottom_sequence has length 18 cone_poset_chain has length 18
	
	

6.2.14 5 generators 2 bound D

Initial Conditions:

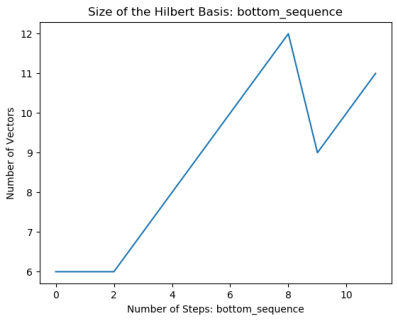
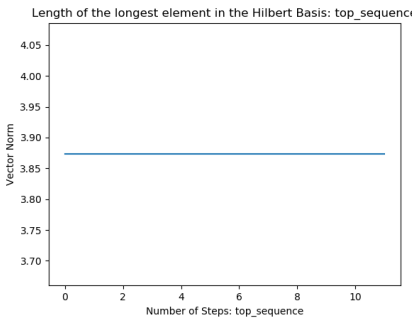
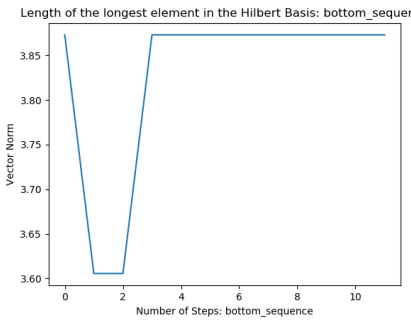
```
inner_cone has generators:
[[-1, 1, 1, 2], [0, 0, 0, 1], [1, -2, 1, 2], [1, 1, 0, 2], [1, 2, -1, 2]]
outer_cone has generators:
[[-2, 2, 1, 2], [0, 0, 0, 1], [1, -2, 1, 2], [1, -1, 2, 1], [1, 2, -1, 2]]
```

Top Down	Bottom Up
sequence_complete = True top_sequence has length 17 bottom_sequence has length 1 cone_poset_chain has length 17	sequence_complete = True top_sequence has length 1 bottom_sequence has length 15 cone_poset_chain has length 15
	
	

6.2.15 5 generators 2 bound E

Initial Conditions:

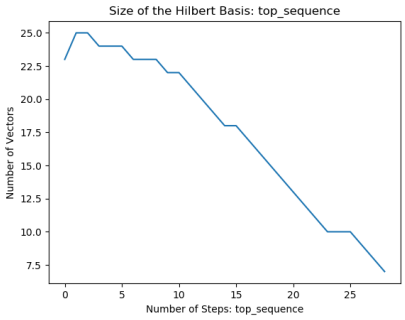
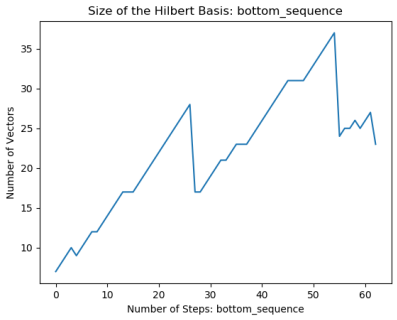
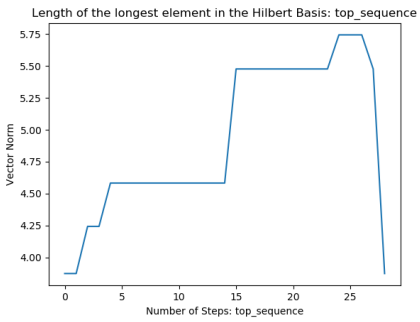
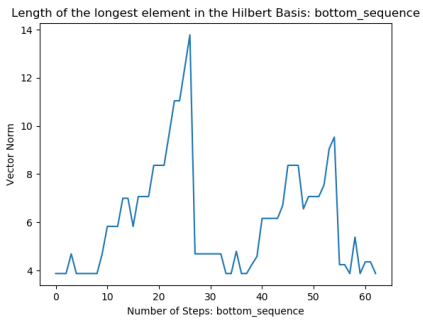
```
inner_cone has generators:
[[0, 0, -1, 2], [0, 1, 0, 1], [1, -1, -1, 2], [1, -1, 0, 1], [2, -1, -1,
    2]]
outer_cone has generators:
[[0, -1, -2, 2], [0, -1, 0, 2], [0, 1, 0, 1], [1, -1, 1, 2], [2, -2, -1,
    1]]
```

Top Down	Bottom Up
sequence_complete = True top_sequence has length 12 bottom_sequence has length 1 cone_poset_chain has length 12	sequence_complete = True top_sequence has length 1 bottom_sequence has length 12 cone_poset_chain has length 12
	
	

6.2.16 5 generators 2 bound F

Initial Conditions:

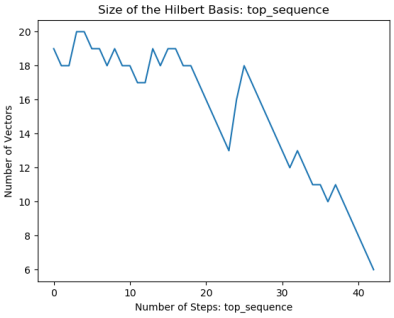
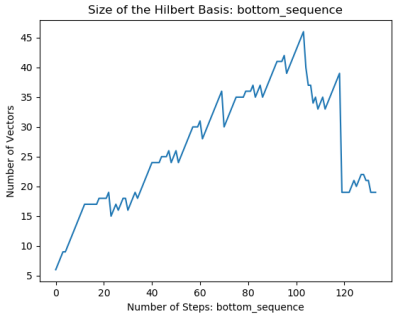
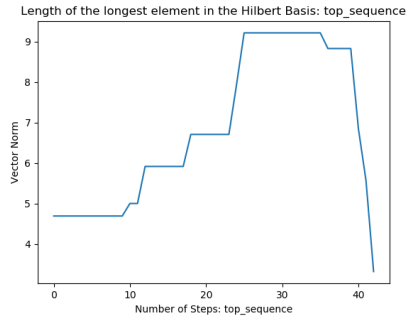
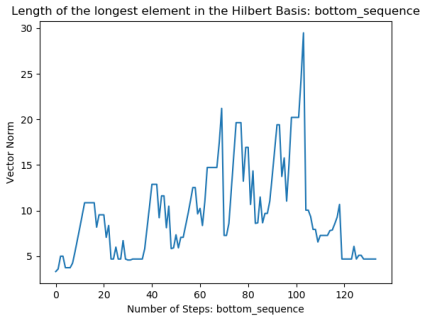
```
inner_cone has generators:
[[0, 0, -1, 1], [0, 0, 1, 1], [1, 0, -1, 1], [1, 1, -1, 2], [2, 1, 1, 2]]
outer_cone has generators:
[[-1, 1, -2, 1], [0, -2, 1, 2], [0, 0, 2, 1], [1, -1, -2, 2], [2, 1, -1,
    1]]
```

Top Down	Bottom Up
sequence_complete = True top_sequence has length 29 bottom_sequence has length 1 cone_poset_chain has length 29	sequence_complete = True top_sequence has length 1 bottom_sequence has length 63 cone_poset_chain has length 63
	
	

6.2.17 5 generators 2 bound G

Initial Conditions:

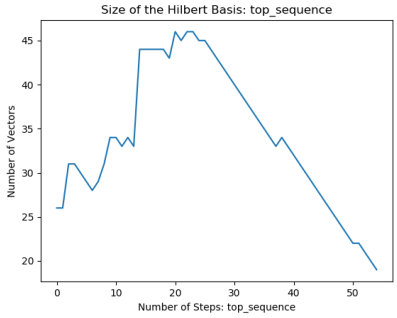
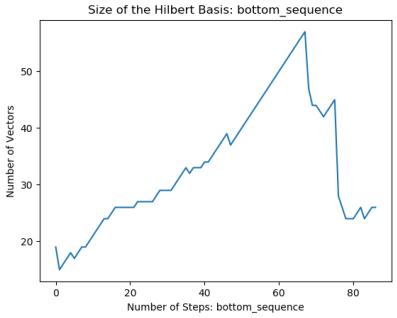
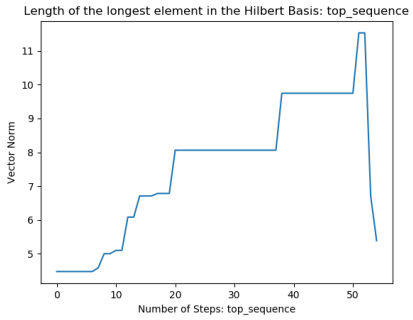
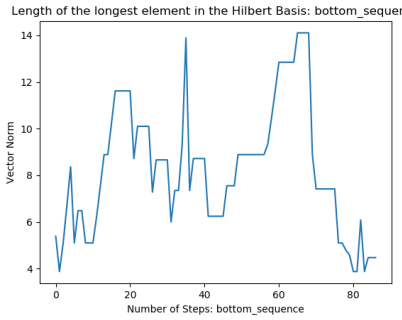
```
inner_cone has generators:
[[-2, -1, 0, 2], [-1, 0, 0, 2], [0, 0, 0, 1], [1, -1, -1, 2], [1, 0, -1,
    2]]
outer_cone has generators:
[[-2, -1, 0, 2], [-1, 2, 0, 2], [0, -1, 1, 1], [1, 0, -2, 1], [1, 0, 0,
    1]]
```

Top Down	Bottom Up
sequence_complete = True top_sequence has length 43 bottom_sequence has length 1 cone_poset_chain has length 43	sequence_complete = True top_sequence has length 1 bottom_sequence has length 134 cone_poset_chain has length 134
	
	

6.2.18 5 generators 2 bound H

Initial Conditions:

```
inner_cone has generators:
[[-2, 1, -2, 2], [-2, 2, -1, 2], [-1, 0, 2, 2], [-1, 1, -2, 2], [0, 2, 1,
    2]]
outer_cone has generators:
[[-2, 2, -2, 1], [-1, -1, -1, 1], [-1, 0, 2, 2], [0, 0, -2, 1], [0, 2, 1,
    2]]
```

Top Down	Bottom Up
sequence_complete = True top_sequence has length 55 bottom_sequence has length 1 cone_poset_chain has length 55	sequence_complete = True top_sequence has length 1 bottom_sequence has length 87 cone_poset_chain has length 87
	
	

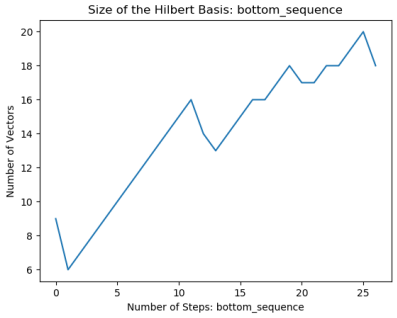
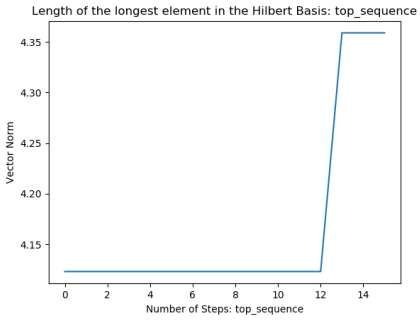
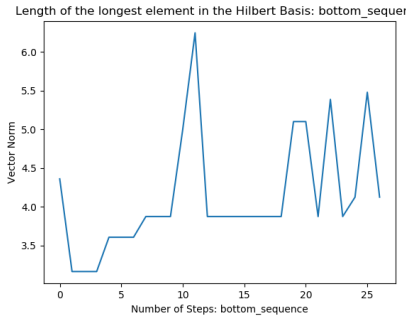
6.2.19 5 generators 2 bound I

Initial Conditions:

```
inner_cone has generators:
[[-1, -2, -2, 1], [0, -1, 1, 2], [0, 0, 0, 1], [1, -2, 0, 2], [1, -1, 0,
    1]]
outer_cone has generators:
[[-1, -2, -2, 1], [0, -1, 2, 2], [0, 0, -2, 1], [0, 1, 1, 2], [2, -1, 0,
    1]]
```

Top Down	Bottom Up

`sequence_complete = True`
`top_sequence has length 16`
`bottom_sequence has length 1`
`cone_poset_chain has length 16`
	`sequence_complete = True`
	`top_sequence has length 1`
	`bottom_sequence has length 27`
	`cone_poset_chain has length 27`

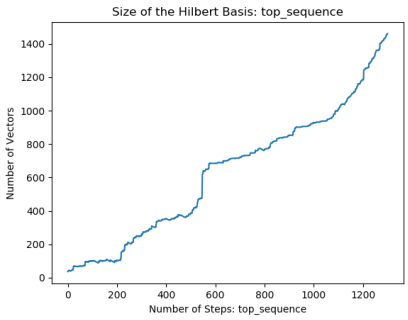
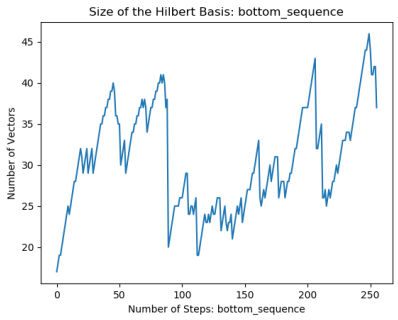
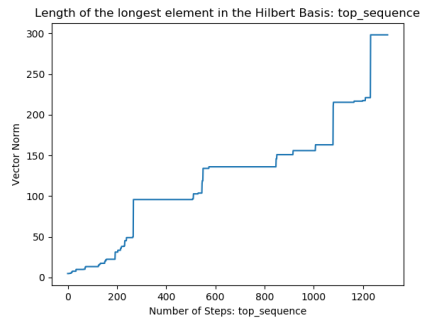
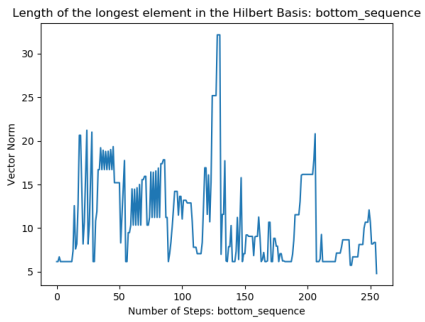
6.2.20 5 generators 2 bound J

Initial Conditions:

```
inner_cone has generators:
[[-1, 2, -2, 2], [1, 1, 0, 2], [2, -1, 0, 2], [2, -1, 1, 2], [2, 1, -2,
    1]]
outer_cone has generators:
[[-1, -2, 1, 2], [-1, 2, -2, 2], [1, -2, 2, 1], [2, 0, 2, 1], [2, 1, -2,
    1]]
```

Top Down	Bottom Up

`sequence_complete = False`
`top_sequence has length 1301`
`bottom_sequence has length 1`
`cone_poset_chain has length 0`
	`sequence_complete = True`
	`top_sequence has length 1`
	`bottom_sequence has length 256`
	`cone_poset_chain has length 256`

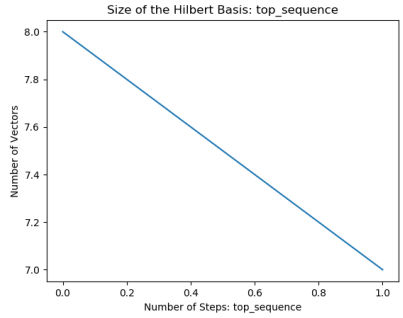
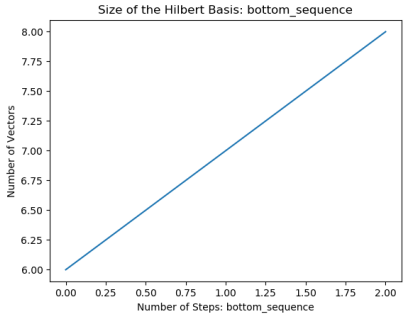
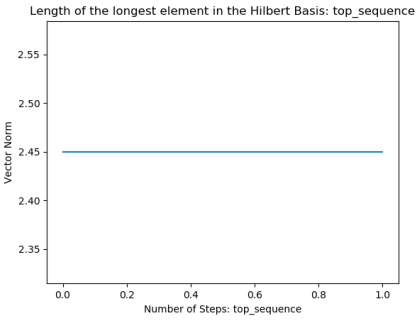
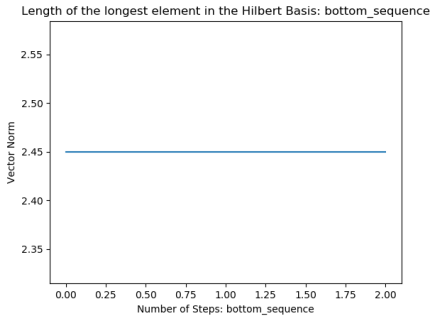
	
	

6.3 Data in dimension 5

6.3.1 5 generators 1 bound A

Initial Conditions:

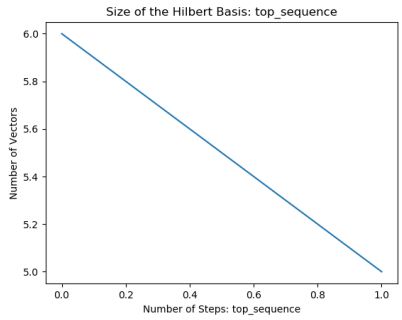
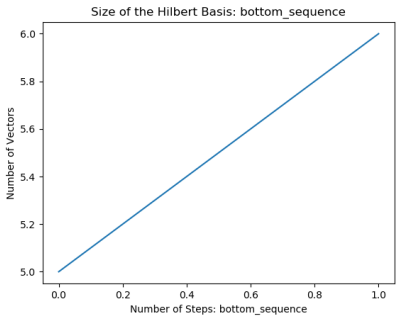
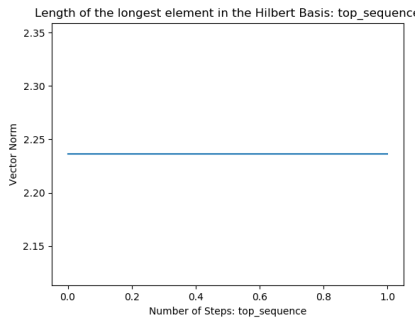
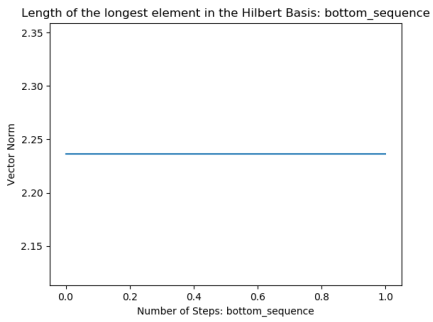
```
inner_cone has generators:
[[-1, -1, 1, -1, 1], [-1, 0, 0, 0, 1], [0, 0, -1, 0, 1], [0, 0, 1, -1, 1],
    [0, 1, 0, -1, 1]]
outer_cone has generators:
[[-1, -1, 1, -1, 1], [-1, 0, 0, 0, 1], [0, 0, -1, 0, 1], [0, 1, 0, -1, 1],
    [1, 1, 1, -1, 1]]
```

Top Down	Bottom Up
sequence_complete = True top_sequence has length 2 bottom_sequence has length 1 cone_poset_chain has length 3	sequence_complete = True top_sequence has length 1 bottom_sequence has length 3 cone_poset_chain has length 3
	
	

6.3.2 5 generators 1 bound B

Initial Conditions:

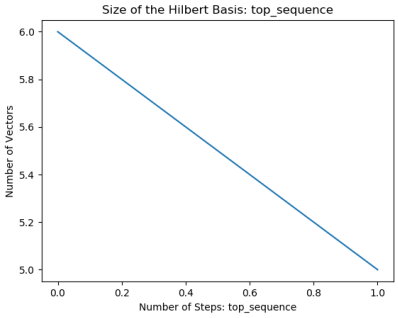
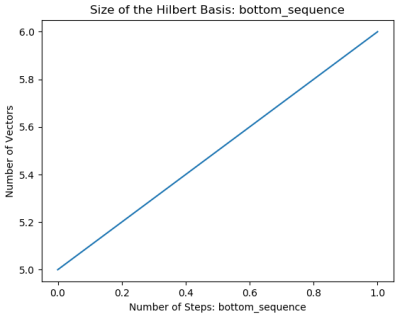
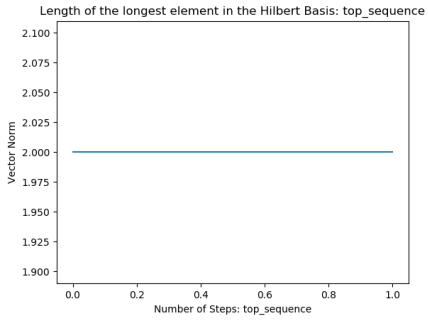
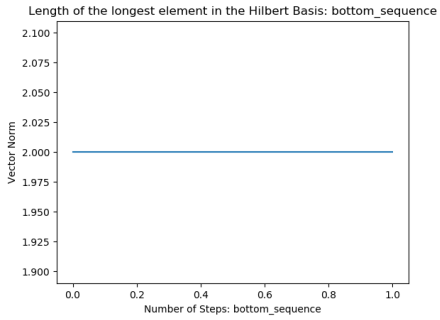
```
inner_cone has generators:
[[0, 1, -1, 1, 1], [1, -1, 1, -1, 1], [1, 0, 0, -1, 1], [1, 0, 0, 0, 1],
    [1, 1, 0, 1, 1]]
outer_cone has generators:
[[0, 1, -1, 1, 1], [1, -1, 1, -1, 1], [1, 0, 0, 0, 1], [1, 1, -1, -1, 1],
    [1, 1, 0, 1, 1]]
```

Top Down	Bottom Up
sequence_complete = True top_sequence has length 2 bottom_sequence has length 1 cone_poset_chain has length 2	sequence_complete = True top_sequence has length 1 bottom_sequence has length 2 cone_poset_chain has length 2
	
	

6.3.3 5 generators 1 bound C

Initial Conditions:

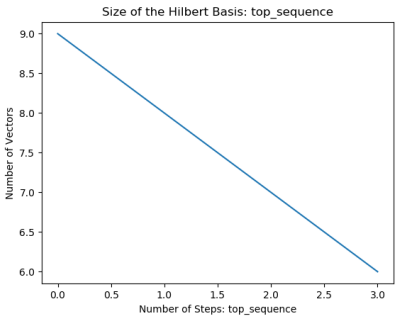
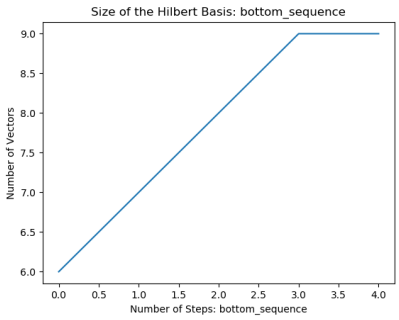
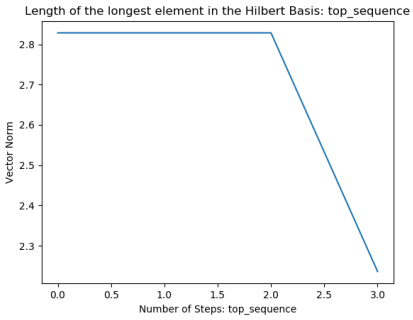
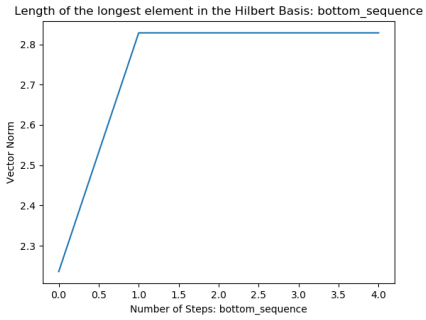
```
inner_cone has generators:
[[-1, 0, 0, 0, 1], [0, 0, -1, -1, 1], [0, 0, 0, 0, 1], [1, -1, 1, 0, 1],
    [1, 0, -1, 0, 1]]
outer_cone has generators:
[[-1, 0, 0, 0, 1], [-1, 1, 0, 0, 1], [0, 0, -1, -1, 1], [1, -1, 1, 0, 1],
    [1, 0, -1, 0, 1]]
```

Top Down	Bottom Up
sequence_complete = True top_sequence has length 2 bottom_sequence has length 1 cone_poset_chain has length 2	sequence_complete = True top_sequence has length 1 bottom_sequence has length 2 cone_poset_chain has length 2
	
	

6.3.4 5 generators 1 bound D

Initial Conditions:

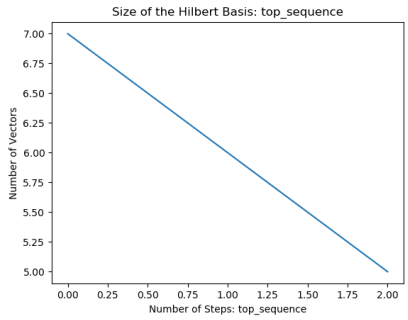
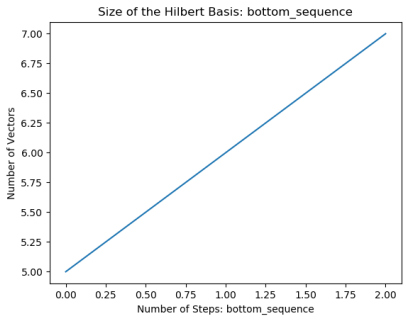
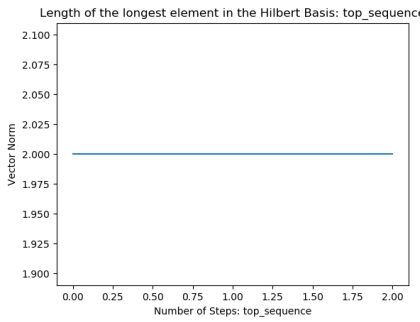
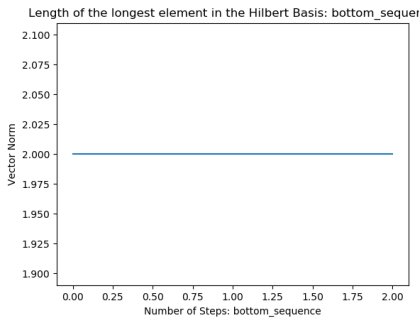
```
inner_cone has generators:
[[-1, 0, -1, 1, 1], [-1, 1, 0, 0, 1], [0, 1, 0, -1, 1], [1, -1, 1, -1, 1],
    [1, 0, 0, 0, 1]]
outer_cone has generators:
[[-1, 0, -1, 1, 1], [-1, 1, 0, 0, 1], [0, 1, 0, -1, 1], [1, -1, 1, -1, 1],
    [1, 1, -1, 1, 1]]
```

Top Down	Bottom Up
sequence_complete = True top_sequence has length 4 bottom_sequence has length 1 cone_poset_chain has length 4	sequence_complete = True top_sequence has length 1 bottom_sequence has length 5 cone_poset_chain has length 5
	
	

6.3.5 5 generators 1 bound E

Initial Conditions:

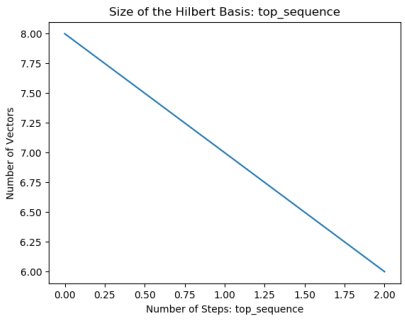
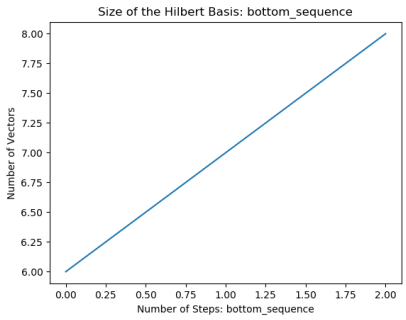
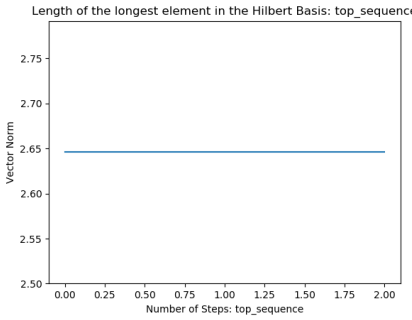
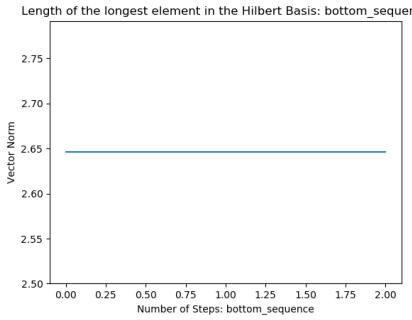
```
inner_cone has generators:
[[-1, 1, 0, -1, 1], [0, -1, -1, 0, 1], [0, -1, 1, -1, 1], [0, 0, 0, 0, 1],
    [0, 1, 0, 0, 1]]
outer_cone has generators:
[[-1, 1, 0, -1, 1], [0, -1, -1, 0, 1], [0, -1, 0, 1, 1], [0, -1, 1, -1,
   1], [0, 1, 0, 0, 1]]
```

Top Down	Bottom Up
sequence_complete = True top_sequence has length 3 bottom_sequence has length 1 cone_poset_chain has length 3	sequence_complete = True top_sequence has length 1 bottom_sequence has length 3 cone_poset_chain has length 3
	
	

6.3.6 5 generators 1 bound F

Initial Conditions:

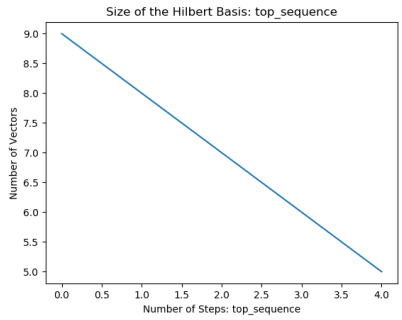
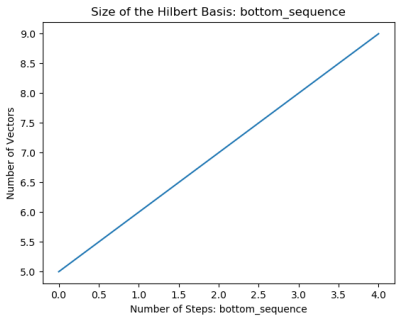
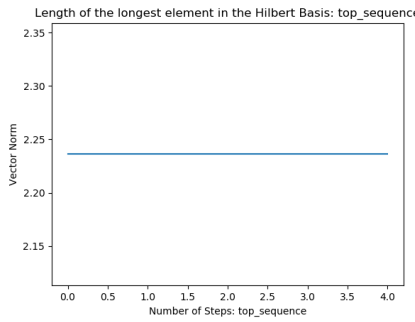
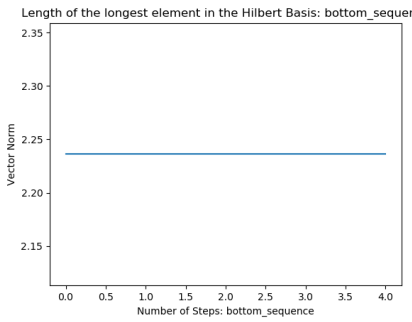
```
inner_cone has generators:
[[-1, 0, 1, -1, 1], [0, 0, 0, 0, 1], [0, 1, 0, 0, 1], [1, 0, 0, 1, 1], [1,
    1, 1, -1, 1]]
outer_cone has generators:
[[-1, 0, 1, -1, 1], [0, 1, 0, 0, 1], [1, 0, -1, 1, 1], [1, 0, 0, 1, 1],
   [1, 1, 1, -1, 1]]
```

Top Down	Bottom Up
sequence_complete = True top_sequence has length 3 bottom_sequence has length 1 cone_poset_chain has length 3	sequence_complete = True top_sequence has length 1 bottom_sequence has length 3 cone_poset_chain has length 3
	
	

6.3.7 5 generators 1 bound G

Initial Conditions:

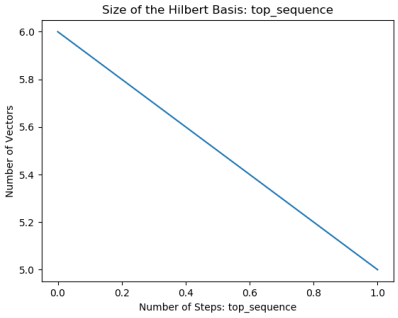
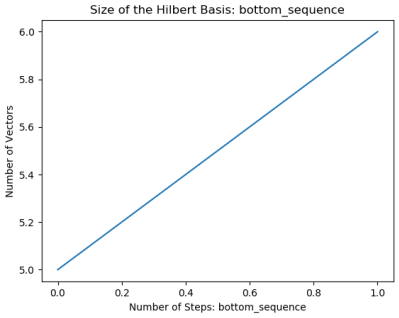
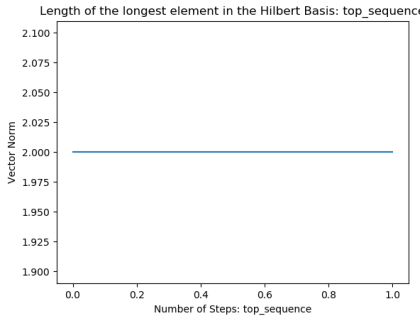
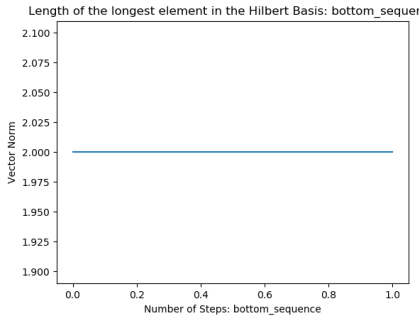
```
inner_cone has generators:
[[-1, 1, -1, -1, 1], [-1, 1, -1, 0, 1], [0, -1, 0, 0, 1], [0, 0, 0, 0, 1],
    [0, 1, -1, 1, 1]]
outer_cone has generators:
[[-1, 1, -1, -1, 1], [-1, 1, -1, 1, 1], [0, -1, 0, 0, 1], [0, -1, 1, -1,
   1], [1, 1, -1, 1, 1]]
```

Top Down	Bottom Up
sequence_complete = True top_sequence has length 5 bottom_sequence has length 1 cone_poset_chain has length 5	sequence_complete = True top_sequence has length 1 bottom_sequence has length 5 cone_poset_chain has length 5
	
	

6.3.8 5 generators 1 bound H

Initial Conditions:

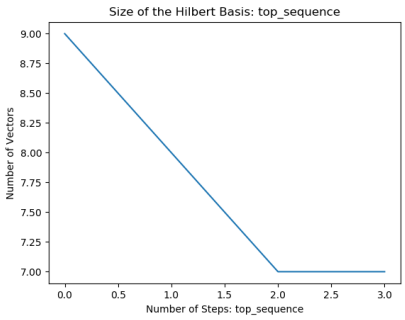
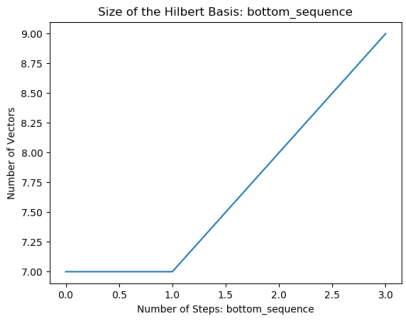
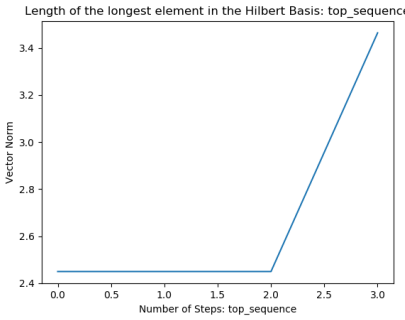
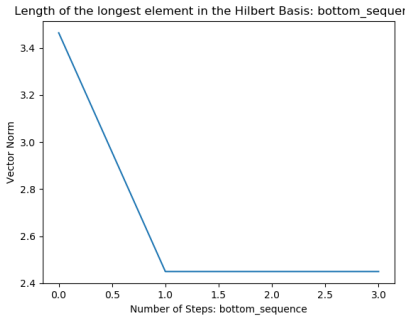
```
inner_cone has generators:
[[-1, 0, 0, -1, 1], [-1, 1, 1, 0, 1], [0, 0, 0, 0, 1], [0, 1, 0, -1, 1],
    [0, 1, 0, 0, 1]]
outer_cone has generators:
[[-1, 0, 0, -1, 1], [-1, 1, 1, 0, 1], [0, 1, 0, -1, 1], [0, 1, 0, 0, 1],
    [1, 0, 0, 1, 1]]
```

Top Down	Bottom Up
sequence_complete = True top_sequence has length 2 bottom_sequence has length 1 cone_poset_chain has length 2	sequence_complete = True top_sequence has length 1 bottom_sequence has length 2 cone_poset_chain has length 2
	
	

6.3.9 5 generators 1 bound I

Initial Conditions:

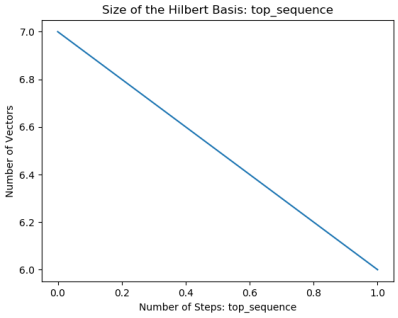
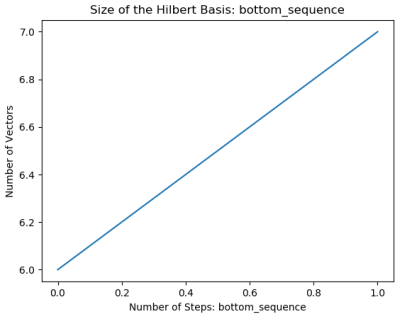
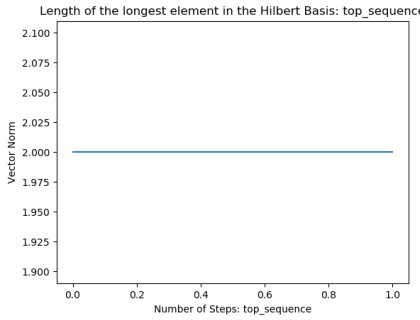
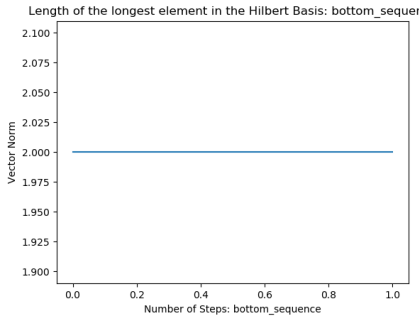
```
inner_cone has generators:
[[-1, 0, 0, -1, 1], [0, -1, -1, 1, 1], [0, 0, 0, 0, 1], [0, 1, 0, 1, 1],
    [1, 1, -1, 1, 1]]
outer_cone has generators:
[[-1, 0, 0, -1, 1], [-1, 1, 1, 1, 1], [0, -1, -1, 1, 1], [0, 0, 0, 0, 1],
    [1, 1, -1, 1, 1]]
```

Top Down	Bottom Up
sequence_complete = True top_sequence has length 4 bottom_sequence has length 1 cone_poset_chain has length 4	sequence_complete = True top_sequence has length 1 bottom_sequence has length 4 cone_poset_chain has length 4
	
	

6.3.10 5 generators 1 bound J

Initial Conditions:

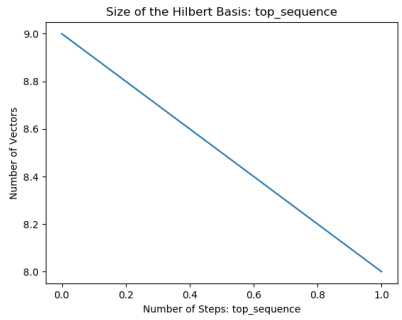
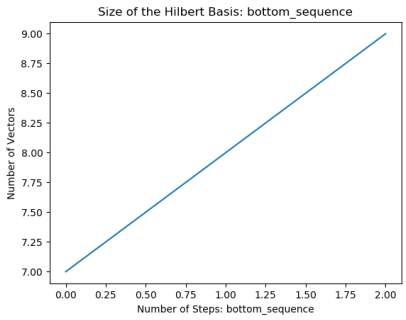
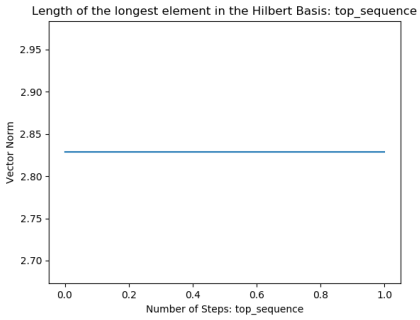
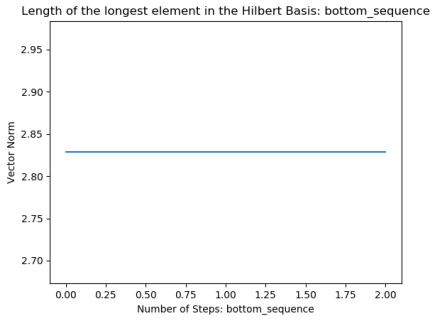
```
inner_cone has generators:
[[-1, -1, 0, 0, 1], [-1, 0, -1, 1, 1], [0, 0, 0, 0, 1], [0, 1, 0, 1, 1],
    [1, 0, -1, 1, 1]]
outer_cone has generators:
[[-1, -1, 0, 0, 1], [-1, 0, -1, 1, 1], [0, 1, 0, 1, 1], [1, 0, -1, 1, 1],
    [1, 0, 0, -1, 1]]
```

Top Down	Bottom Up
sequence_complete = True top_sequence has length 2 bottom_sequence has length 1 cone_poset_chain has length 2	sequence_complete = True top_sequence has length 1 bottom_sequence has length 2 cone_poset_chain has length 2
	
	

6.3.11 6 generators 1 bound A

Initial Conditions:

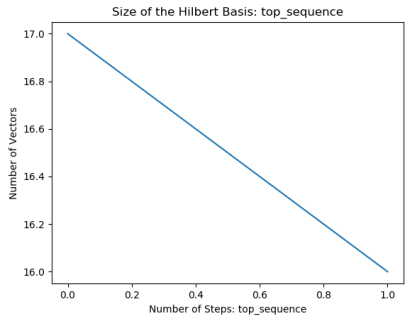
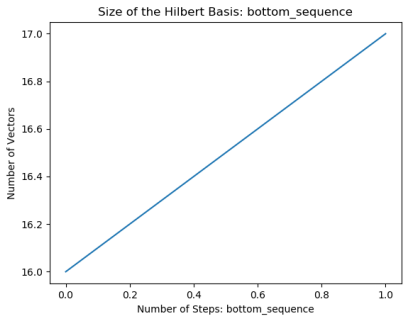
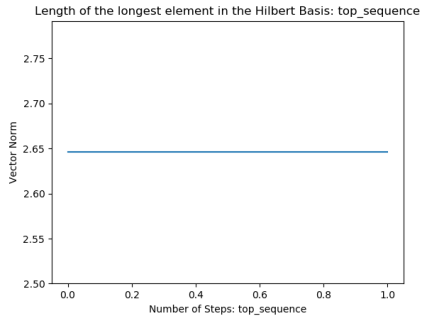
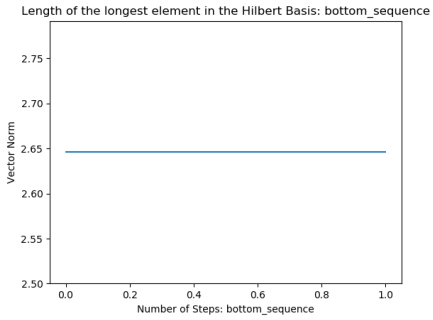
```
inner_cone has generators:
[[-1, 0, -1, -1, 1], [-1, 1, -1, -1, 1], [0, 0, 0, -1, 1], [1, 0, -1, 0,
    1], [1, 1, 0, -1, 1], [1, 1, 0, 0, 1]]
outer_cone has generators:
[[-1, 0, -1, -1, 1], [-1, 1, -1, -1, 1], [0, 0, 0, -1, 1], [1, 0, -1, 0,
    1], [1, 1, 0, -1, 1], [1, 1, 0, 1, 1]]
```

Top Down	Bottom Up
sequence_complete = True top_sequence has length 2 bottom_sequence has length 1 cone_poset_chain has length 3	sequence_complete = True top_sequence has length 1 bottom_sequence has length 3 cone_poset_chain has length 3
	
	

6.3.12 6 generators 1 bound B

Initial Conditions:

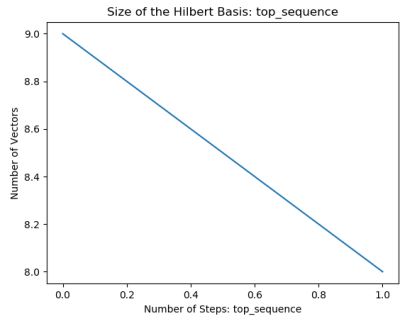
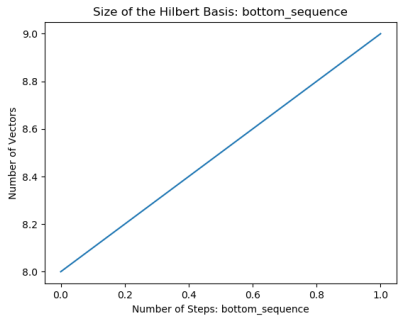
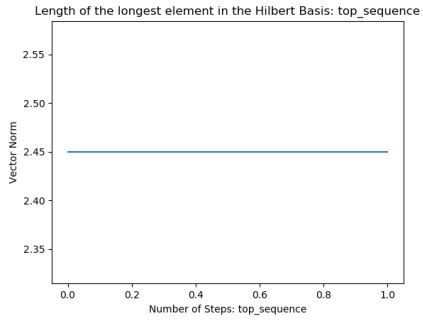
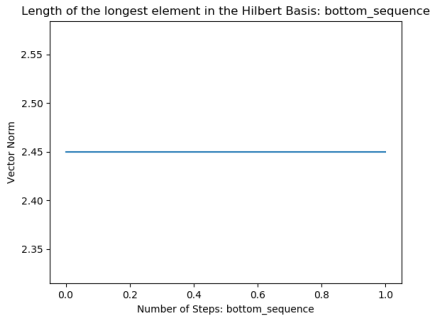
```
inner_cone has generators:
[[-1, 1, 1, 0, 1], [0, -1, -1, -1, 1], [0, 0, 1, 0, 1], [0, 1, 0, 1, 1],
    [1, -1, 0, 0, 1], [1, 1, 1, -1, 1]]
outer_cone has generators:
[[-1, 1, 1, 0, 1], [0, -1, -1, -1, 1], [0, 1, 0, 1, 1], [1, -1, 0, 0, 1],
    [1, -1, 1, 0, 1], [1, 1, 1, -1, 1]]
```

Top Down	Bottom Up
sequence_complete = True top_sequence has length 2 bottom_sequence has length 1 cone_poset_chain has length 2	sequence_complete = True top_sequence has length 1 bottom_sequence has length 2 cone_poset_chain has length 2
	
	

6.3.13 6 generators 1 bound C

Initial Conditions:

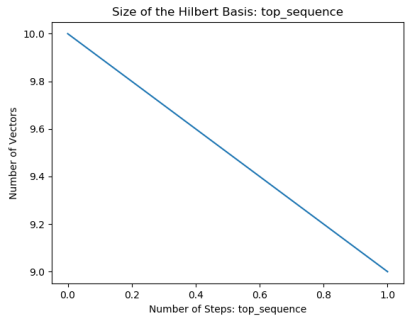
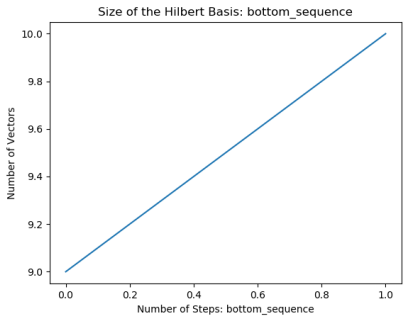
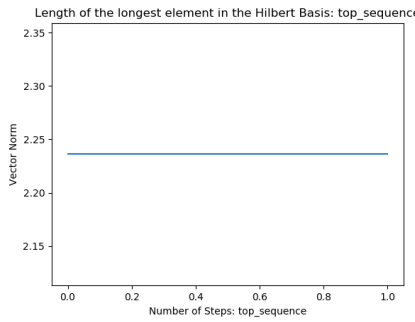
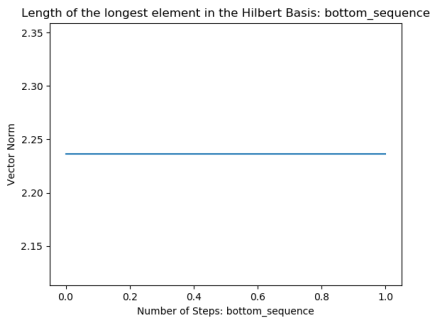
```
inner_cone has generators:
[[-1, -1, 1, 1, 1], [-1, 0, 0, -1, 1], [0, -1, 0, 0, 1], [0, 0, 0, -1, 1],
    [0, 1, 0, 0, 1], [0, 1, 1, 1, 1]]
outer_cone has generators:
[[-1, -1, 1, 1, 1], [-1, 0, 0, -1, 1], [0, 0, 0, -1, 1], [0, 1, 0, 0, 1],
   [0, 1, 1, 1, 1], [1, -1, -1, -1, 1]]
```

Top Down	Bottom Up
sequence_complete = True top_sequence has length 2 bottom_sequence has length 1 cone_poset_chain has length 2	sequence_complete = True top_sequence has length 1 bottom_sequence has length 2 cone_poset_chain has length 2
	
	

6.3.14 6 generators 1 bound D

Initial Conditions:

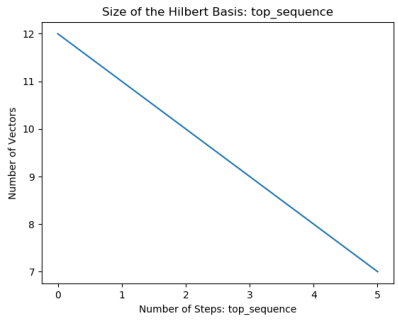
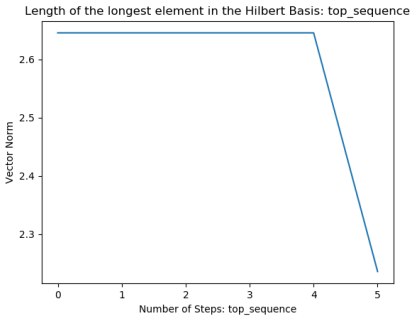
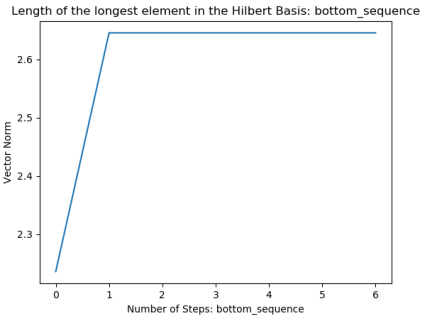
```
inner_cone has generators:
[[-1, -1, 0, 0, 1], [-1, 1, -1, -1, 1], [-1, 1, 0, -1, 1], [0, 0, 0, 1,
    1], [0, 1, -1, -1, 1], [1, 1, 0, -1, 1]]
outer_cone has generators:
[[-1, -1, 0, 0, 1], [-1, 1, -1, -1, 1], [-1, 1, 1, -1, 1], [0, 0, 0, 1,
    1], [0, 1, -1, -1, 1], [1, 1, 0, -1, 1]]
```

Top Down	Bottom Up
sequence_complete = True top_sequence has length 2 bottom_sequence has length 1 cone_poset_chain has length 2	sequence_complete = True top_sequence has length 1 bottom_sequence has length 2 cone_poset_chain has length 2
	
	

6.3.15 6 generators 1 bound E

Initial Conditions:

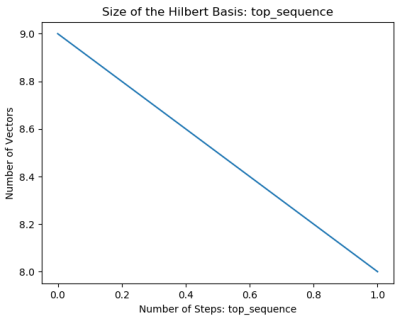
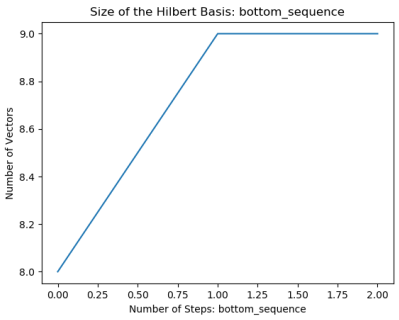
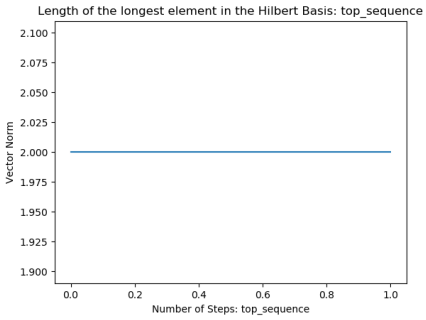
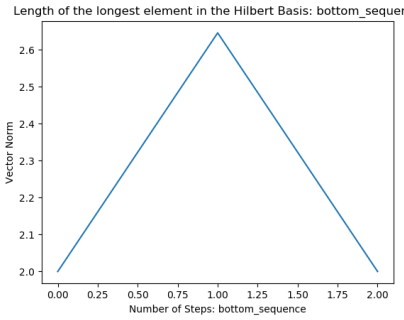
```
inner_cone has generators:
[[-1, -1, -1, 1, 1], [0, -1, 0, 0, 1], [1, -1, 0, 0, 1], [1, 0, 0, 0, 1],
    [1, 0, 0, 1, 1], [1, 1, 1, 1, 1]]
outer_cone has generators:
[[-1, -1, -1, 1, 1], [-1, -1, 0, 0, 1], [1, -1, 0, 0, 1], [1, 0, -1, -1,
    1], [1, 0, 0, 1, 1], [1, 1, 1, 1, 1]]
```

Top Down	Bottom Up
sequence_complete = True top_sequence has length 6 bottom_sequence has length 1 cone_poset_chain has length 6	sequence_complete = True top_sequence has length 1 bottom_sequence has length 7 cone_poset_chain has length 7
	
	

6.3.16 6 generators 1 bound F

Initial Conditions:

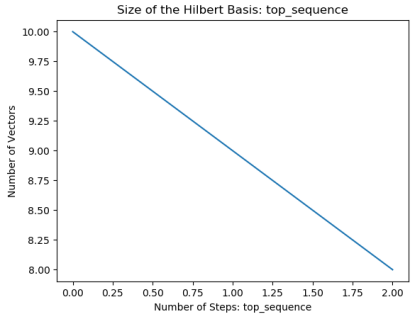
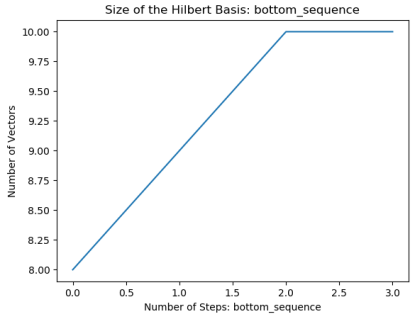
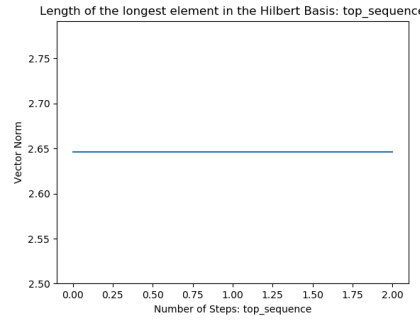
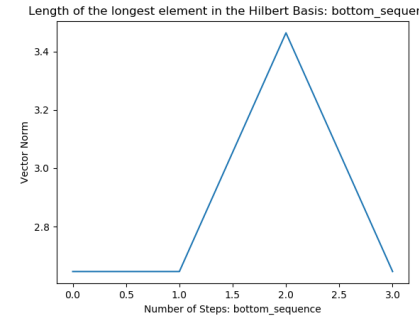
```
inner_cone has generators:
[[-1, 0, 0, 1, 1], [-1, 1, 0, 0, 1], [-1, 1, 0, 1, 1], [0, -1, 1, -1, 1],
    [0, 0, -1, 1, 1], [0, 1, 1, -1, 1]]
outer_cone has generators:
[[-1, -1, 0, 1, 1], [-1, 1, 0, 0, 1], [-1, 1, 0, 1, 1], [0, -1, 1, -1, 1],
     [0, 0, -1, 1, 1], [0, 1, 1, -1, 1]]
```

Top Down	Bottom Up
sequence_complete = True top_sequence has length 2 bottom_sequence has length 1 cone_poset_chain has length 2	sequence_complete = True top_sequence has length 1 bottom_sequence has length 3 cone_poset_chain has length 3
	
	

6.3.17 6 generators 1 bound G

Initial Conditions:

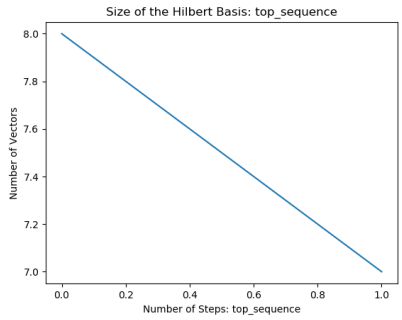
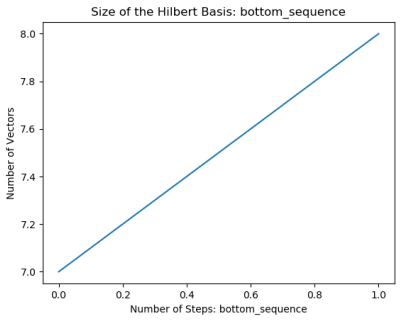
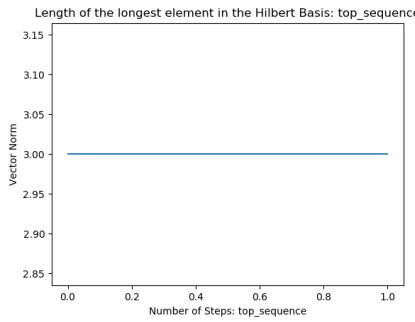
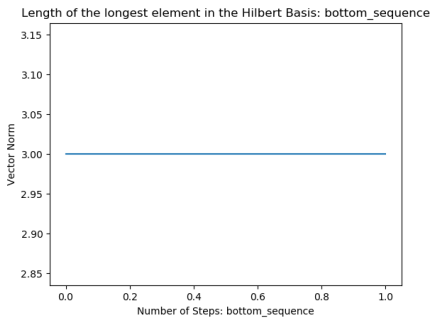
```
inner_cone has generators:
[[-1, 1, -1, 0, 1], [0, -1, -1, 1, 1], [0, 0, 0, 0, 1], [0, 0, 1, 0, 1],
    [0, 1, 1, 0, 1], [1, 1, 0, 1, 1]]
outer_cone has generators:
[[-1, 1, -1, 0, 1], [0, -1, -1, 1, 1], [0, 0, 1, 0, 1], [0, 1, 1, 0, 1],
    [1, -1, 0, 0, 1], [1, 1, 0, 1, 1]]
```

Top Down	Bottom Up
sequence_complete = True top_sequence has length 3 bottom_sequence has length 1 cone_poset_chain has length 3	sequence_complete = True top_sequence has length 1 bottom_sequence has length 4 cone_poset_chain has length 4
	
	

6.3.18 6 generators 1 bound H

Initial Conditions:

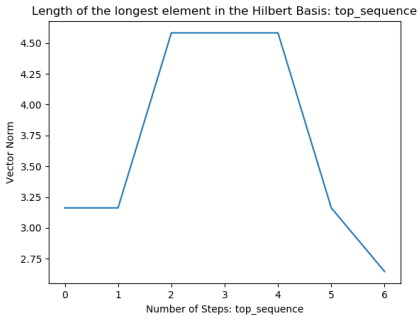
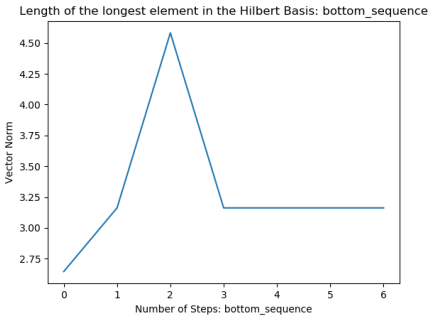
```
inner_cone has generators:
[[-1, -1, 0, -1, 1], [-1, 0, -1, 1, 1], [-1, 0, 0, 0, 1], [-1, 1, -1, 0,
    1], [-1, 1, 0, 0, 1], [0, -1, 0, 0, 1]]
outer_cone has generators:
[[-1, -1, 0, -1, 1], [-1, 0, -1, 1, 1], [-1, 0, 1, 0, 1], [-1, 1, -1, 0,
    1], [-1, 1, 0, 0, 1], [0, -1, 0, 0, 1]]
```

Top Down	Bottom Up
sequence_complete = True top_sequence has length 2 bottom_sequence has length 1 cone_poset_chain has length 2	sequence_complete = True top_sequence has length 1 bottom_sequence has length 2 cone_poset_chain has length 2
	
	

6.3.19 6 generators 1 bound I

Initial Conditions:

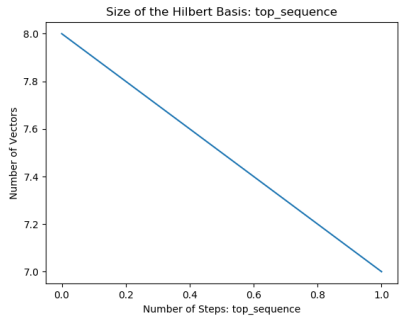
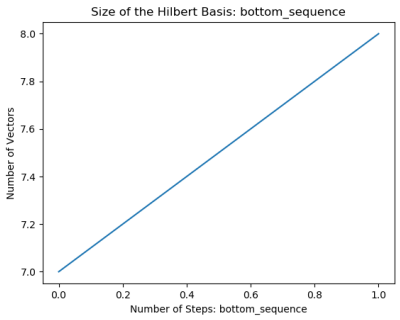
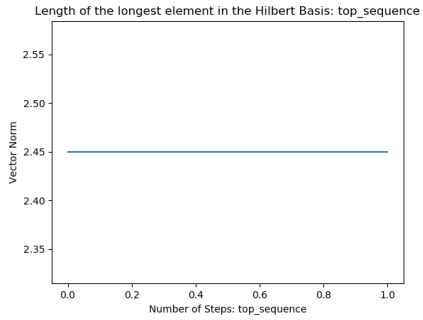
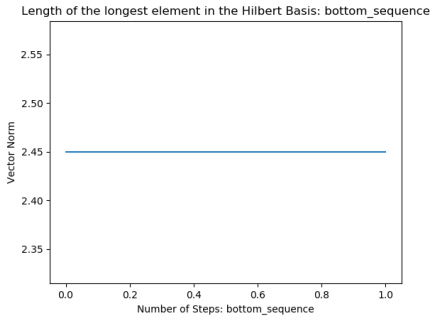
```
inner_cone has generators:
[[-1, 0, -1, -1, 1], [-1, 1, 1, -1, 1], [0, -1, 1, -1, 1], [1, -1, 1, 0,
    1], [1, -1, 1, 1, 1], [1, 0, 1, 0, 1]]
outer_cone has generators:
[[-1, 0, -1, -1, 1], [-1, 1, 1, -1, 1], [0, -1, 1, -1, 1], [1, -1, 1, -1,
    1], [1, -1, 1, 1, 1], [1, 0, 1, 0, 1]]
```

Top Down	Bottom Up
sequence_complete = True top_sequence has length 7 bottom_sequence has length 1 cone_poset_chain has length 7	sequence_complete = True top_sequence has length 1 bottom_sequence has length 7 cone_poset_chain has length 7
	
	

6.3.20 6 generators 1 bound J

Initial Conditions:

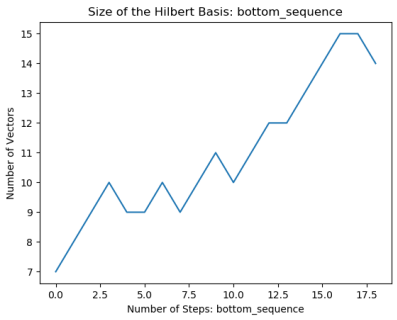
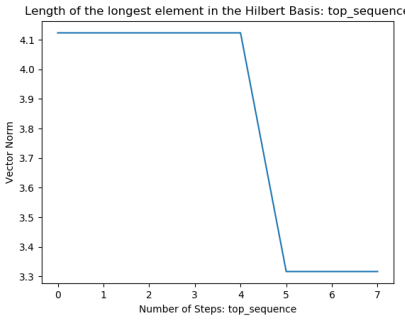
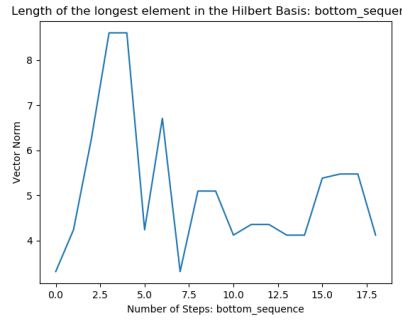
```
inner_cone has generators:
[[-1, -1, 0, 0, 1], [-1, 0, 0, -1, 1], [0, 0, 1, 0, 1], [0, 0, 1, 1, 1],
    [1, 0, 1, -1, 1], [1, 1, 1, 0, 1]]
outer_cone has generators:
[[-1, -1, 0, 0, 1], [-1, 0, 0, -1, 1], [-1, 0, 1, 0, 1], [0, 0, 1, 1, 1],
    [1, 0, 1, -1, 1], [1, 1, 1, 0, 1]]
```

Top Down	Bottom Up
sequence_complete = True top_sequence has length 2 bottom_sequence has length 1 cone_poset_chain has length 2	sequence_complete = True top_sequence has length 1 bottom_sequence has length 2 cone_poset_chain has length 2
	
	

6.3.21 5 generators 2 bound A

Initial Conditions:

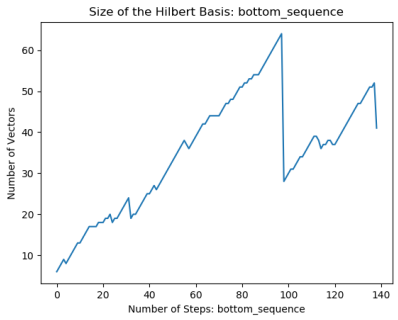
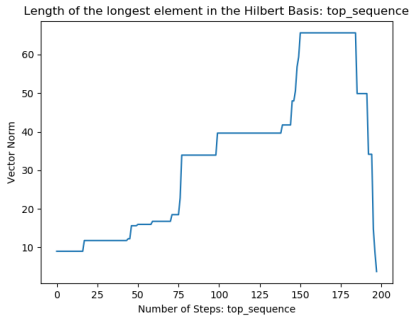
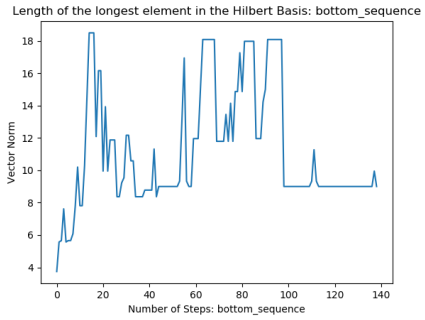
```
inner_cone has generators:
[[-1, -1, 1, 1, 1], [-1, -1, 1, 2, 2], [-1, 0, -1, -2, 2], [-1, 0, 1, 2,
    1], [0, 1, 0, 1, 2]]
outer_cone has generators:
[[-1, -2, 1, 1, 1], [-1, 0, -1, -2, 2], [-1, 0, 1, 1, 1], [-1, 0, 1, 2,
    1], [2, 2, -2, -1, 2]]
```

Top Down	Bottom Up
sequence_complete = True top_sequence has length 8 bottom_sequence has length 1 cone_poset_chain has length 8	sequence_complete = True top_sequence has length 1 bottom_sequence has length 19 cone_poset_chain has length 19
	
	

6.3.22 5 generators 2 bound B

Initial Conditions:

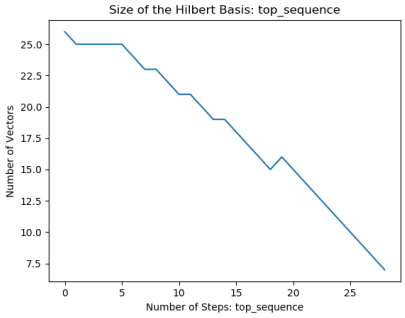
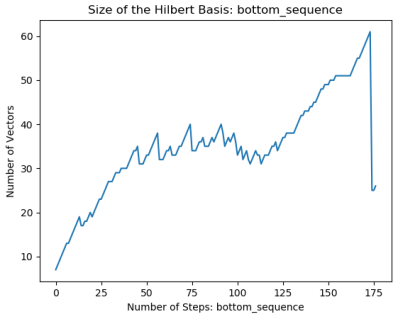
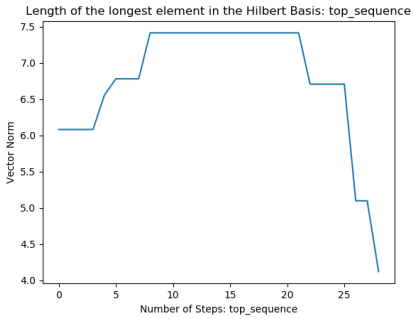
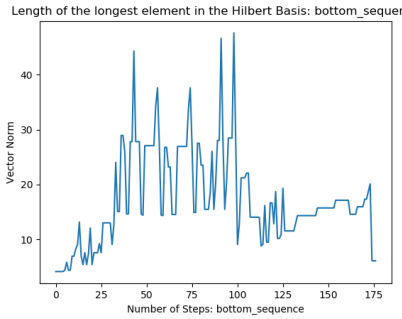
```
inner_cone has generators:
[[1, 0, 0, -1, 1], [1, 0, 0, -1, 2], [2, 1, -1, -2, 2], [2, 1, 1, -2, 2],
    [2, 1, 2, -1, 2]]
outer_cone has generators:
[[-1, -1, 0, 1, 2], [1, -1, -1, -1, 1], [1, 0, 2, -2, 1], [2, 1, 2, -1,
    2], [2, 2, -1, -2, 2]]
```

Top Down	Bottom Up
sequence_complete = True top_sequence has length 198 bottom_sequence has length 1 cone_poset_chain has length 198	sequence_complete = True top_sequence has length 1 bottom_sequence has length 139 cone_poset_chain has length 139
	
	

6.3.23 5 generators 2 bound C

Initial Conditions:

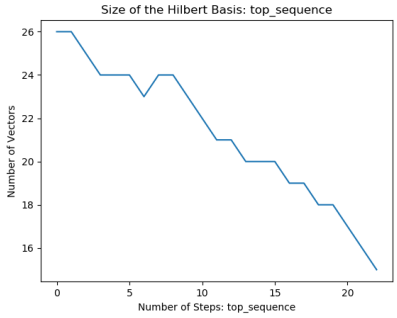
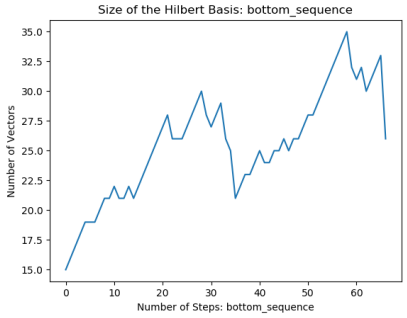
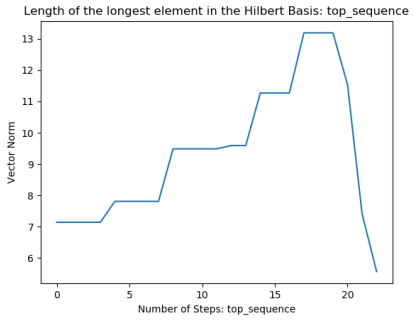
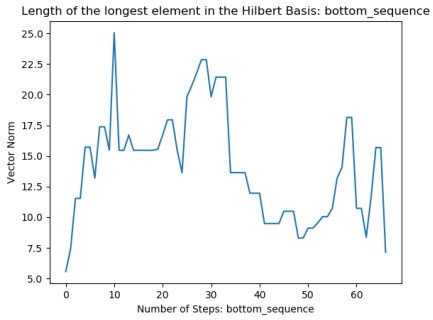
```
inner_cone has generators:
[[-2, -2, -2, -1, 2], [-1, 1, -1, 1, 1], [0, 0, 0, 0, 1], [1, 0, 1, -1,
    2], [1, 2, 0, -2, 2]]
outer_cone has generators:
[[-2, -2, -2, -1, 2], [-1, 1, -1, 1, 1], [0, 2, 1, 2, 2], [1, -1, 1, -1,
    1], [1, 2, 0, -2, 1]]
```

Top Down	Bottom Up
sequence_complete = True top_sequence has length 29 bottom_sequence has length 1 cone_poset_chain has length 29	sequence_complete = True top_sequence has length 1 bottom_sequence has length 177 cone_poset_chain has length 177
	
	

6.3.24 5 generators 2 bound D

Initial Conditions:

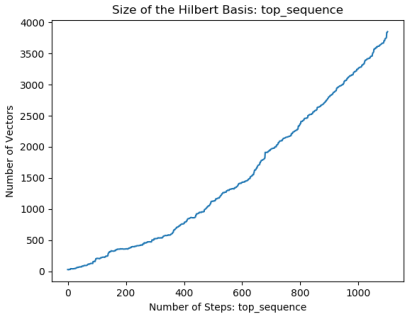
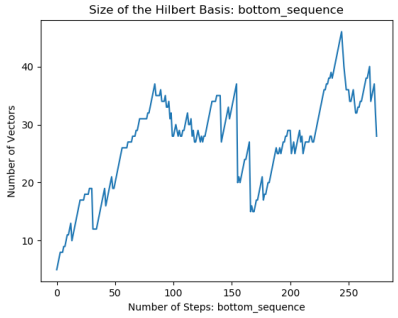
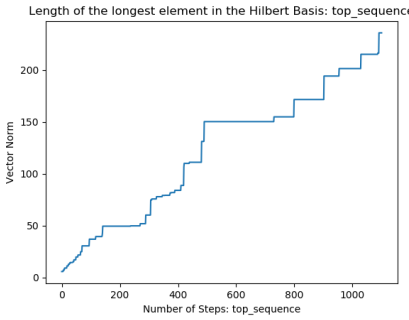
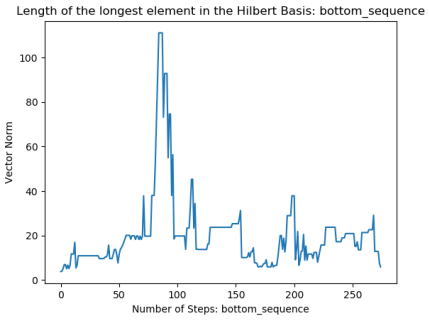
```
inner_cone has generators:
[[-2, -1, -2, -1, 2], [-2, 0, -1, -2, 2], [-1, -1, -1, 0, 2], [-1, 2, 0,
    2, 2], [1, -2, 1, 2, 1]]
outer_cone has generators:
[[-2, -1, -2, -1, 2], [-2, 0, -1, -2, 2], [-1, -1, -2, 0, 2], [-1, 2, 0,
    2, 2], [1, -2, 1, 2, 1]]
```

Top Down	Bottom Up
sequence_complete = True top_sequence has length 23 bottom_sequence has length 1 cone_poset_chain has length 23	sequence_complete = True top_sequence has length 1 bottom_sequence has length 67 cone_poset_chain has length 67
	
	

6.3.25 5 generators 2 bound E

Initial Conditions:

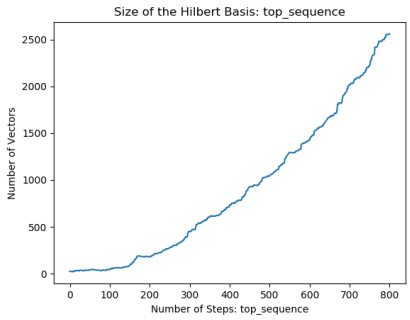
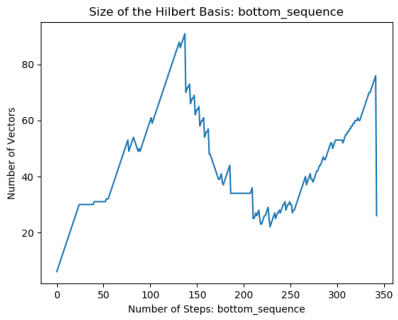
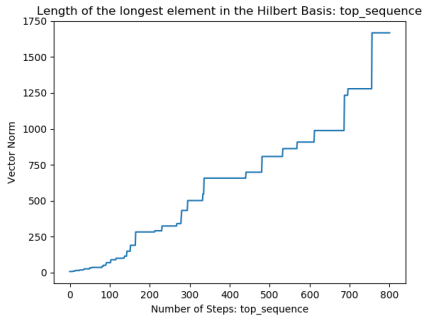
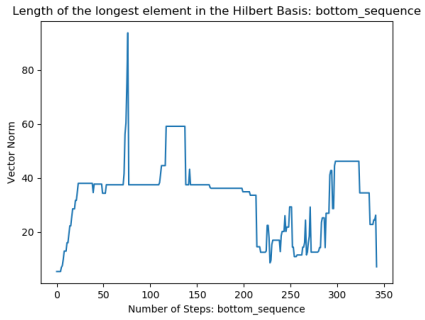
```
inner_cone has generators:
[[-2, -1, -1, 2, 2], [0, -2, -1, -1, 1], [0, -1, -2, -1, 2], [0, 0, -1, 0,
    1], [1, 2, -1, -1, 1]]
outer_cone has generators:
[[-2, -1, -1, 2, 2], [0, -2, -1, -1, 1], [0, 2, -2, 0, 1], [1, -1, 0, 2,
   2], [1, 2, -1, -1, 1]]
```

Top Down	Bottom Up
sequence_complete = False top_sequence has length 1101 bottom_sequence has length 1 cone_poset_chain has length 0	sequence_complete = True top_sequence has length 1 bottom_sequence has length 275 cone_poset_chain has length 275
	
	

6.3.26 5 generators 2 bound F

Initial Conditions:

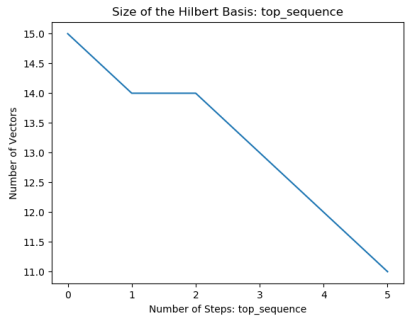
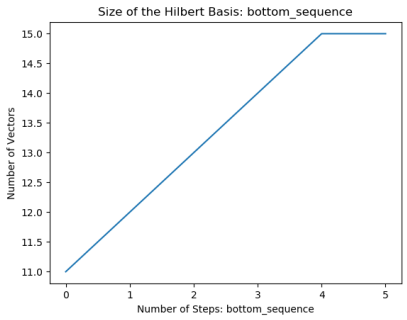
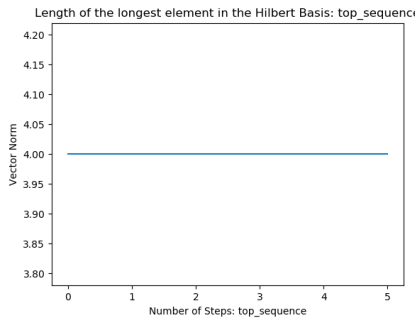
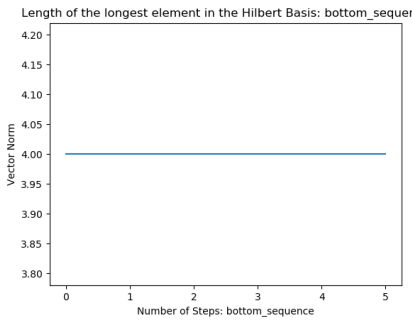
```
inner_cone has generators:
[[-2, -1, 2, -2, 2], [-1, -1, 1, -1, 2], [-1, 0, 1, 0, 2], [0, -2, -2, -1,
    2], [0, -1, 0, 0, 1]]
outer_cone has generators:
[[-2, -1, 2, -2, 2], [-1, 1, 1, 1, 2], [0, -2, -2, -1, 2], [0, -1, 0, 0,
   1], [1, -1, 0, -1, 1]]
```

Top Down	Bottom Up
sequence_complete = False top_sequence has length 801 bottom_sequence has length 1 cone_poset_chain has length 0	sequence_complete = True top_sequence has length 1 bottom_sequence has length 343 cone_poset_chain has length 343
	
	

6.3.27 5 generators 2 bound G

Initial Conditions:

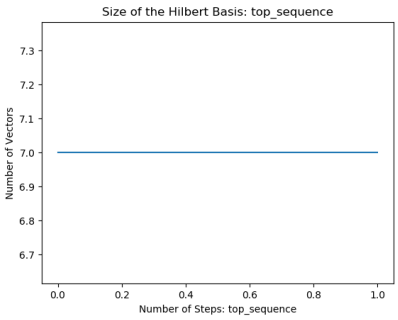
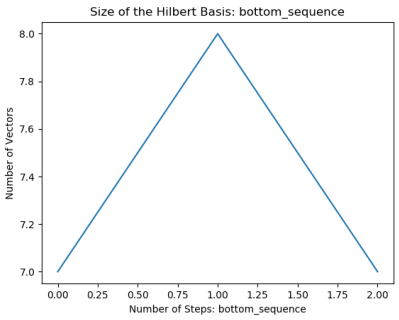
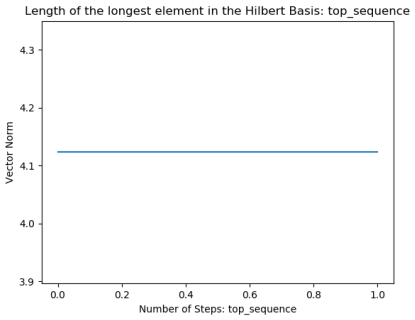
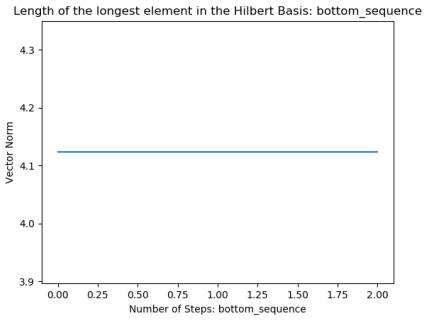
```
inner_cone has generators:
[[-1, -1, -1, 2, 2], [-1, -1, 1, 2, 2], [-1, 0, -1, 0, 1], [-1, 0, 0, 1,
   2], [1, 1, 2, -2, 2]]
outer_cone has generators:
[[-1, -1, 1, 2, 2], [-1, 0, -1, 0, 1], [-1, 1, 0, 0, 2], [0, -1, 0, 2, 1],
    [1, 1, 2, -2, 2]]
```

Top Down	Bottom Up
sequence_complete = True top_sequence has length 6 bottom_sequence has length 1 cone_poset_chain has length 6	sequence_complete = True top_sequence has length 1 bottom_sequence has length 6 cone_poset_chain has length 6
	
	

6.3.28 5 generators 2 bound H

Initial Conditions:

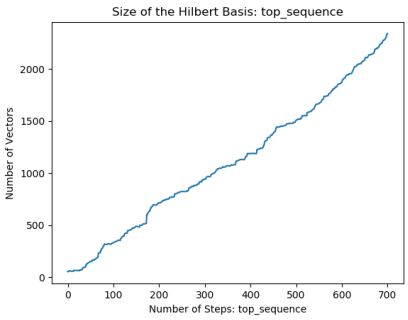
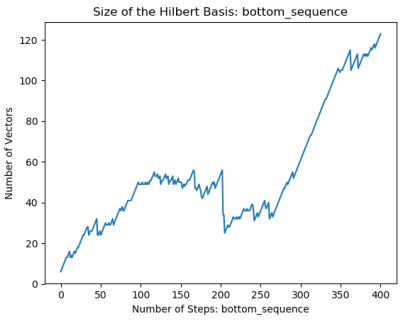
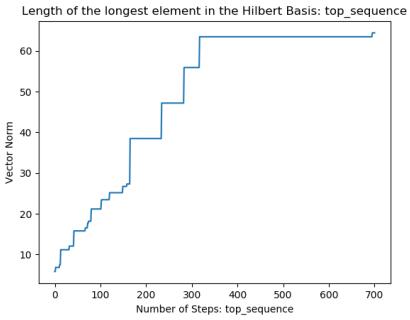
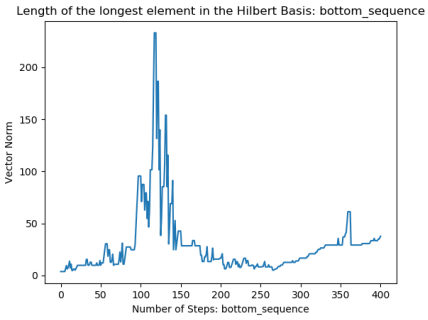
```
inner_cone has generators:
[[-1, -2, -2, 2, 1], [-1, 0, -1, 2, 2], [-1, 1, -2, -1, 1], [1, 2, 0, 1,
   2], [2, 2, 1, 2, 2]]
outer_cone has generators:
[[-1, -2, -2, 2, 1], [-1, 1, -2, -1, 1], [0, 1, 1, 1, 1], [1, 2, 0, 1, 2],
    [2, 2, 1, 2, 2]]
```

Top Down	Bottom Up
sequence_complete = True top_sequence has length 2 bottom_sequence has length 1 cone_poset_chain has length 2	sequence_complete = True top_sequence has length 1 bottom_sequence has length 3 cone_poset_chain has length 3
	
	

6.3.29 5 generators 2 bound I

Initial Conditions:

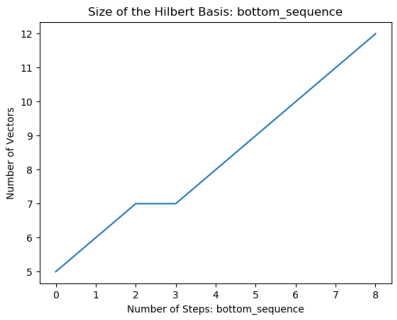
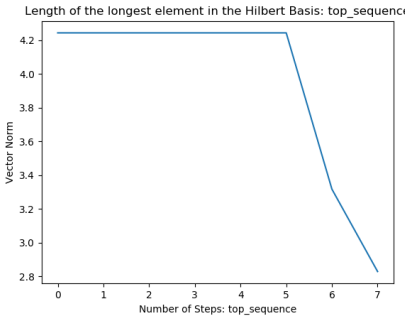
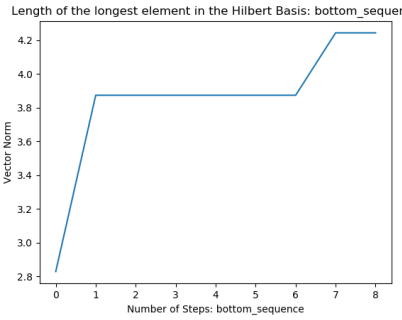
```
inner_cone has generators:
[[0, 1, 0, -1, 2], [1, 1, 2, 0, 2], [1, 2, -2, 0, 1], [1, 2, -2, 2, 1],
    [1, 2, -1, 1, 2]]
outer_cone has generators:
[[-1, 0, 1, -2, 2], [1, -1, -1, -1, 1], [1, 1, 2, 0, 2], [1, 2, -2, -1,
    1], [1, 2, -2, 2, 1]]
```

Top Down	Bottom Up
sequence_complete = False top_sequence has length 701 bottom_sequence has length 1 cone_poset_chain has length 0	sequence_complete = False top_sequence has length 1 bottom_sequence has length 401 cone_poset_chain has length 0
	
	

6.3.30 5 generators 2 bound J

Initial Conditions:

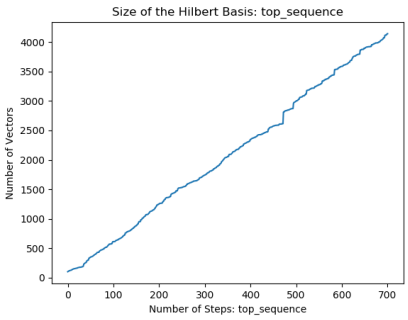
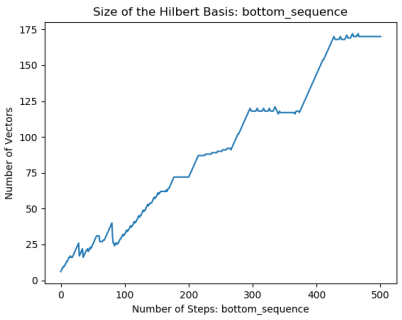
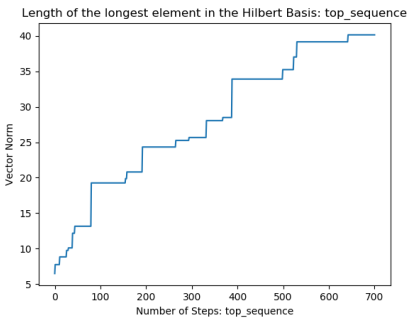
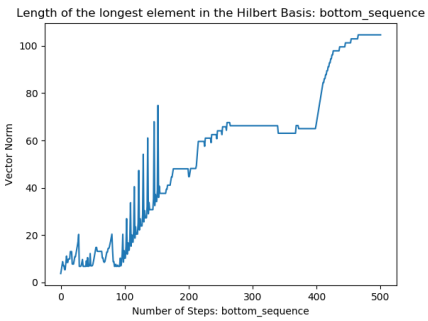
```
inner_cone has generators:
[[-1, -2, -1, 1, 1], [-1, -1, 0, 0, 2], [-1, 0, 0, -1, 2], [0, -1, -1, 0,
    1], [0, 0, -1, -1, 2]]
outer_cone has generators:
[[-1, -2, -1, 1, 1], [-1, 0, 0, -1, 2], [-1, 0, 2, 0, 2], [0, -1, -1, 0,
    1], [1, 2, -1, -2, 2]]
```

Top Down	Bottom Up
sequence_complete = True top_sequence has length 8 bottom_sequence has length 1 cone_poset_chain has length 8	sequence_complete = True top_sequence has length 1 bottom_sequence has length 9 cone_poset_chain has length 9
	
	

6.3.31 6 generators 2 bound A

Initial Conditions:

```
inner_cone has generators:
[[1, -1, 1, 0, 1], [2, -2, -2, -1, 1], [2, -2, -1, -1, 2], [2, -2, 0, -1,
    2], [2, -1, 1, 0, 2], [2, -1, 2, 0, 2]]
outer_cone has generators:
[[-2, -2, -2, 1, 2], [-1, 2, 1, 1, 2], [2, -2, -2, -1, 1], [2, -2, 2, -2,
    1], [2, -1, 1, -1, 1], [2, -1, 2, 1, 1]]
```

Top Down	Bottom Up
sequence_complete = False top_sequence has length 701 bottom_sequence has length 1 cone_poset_chain has length 0	sequence_complete = False top_sequence has length 1 bottom_sequence has length 501 cone_poset_chain has length 0
	
	

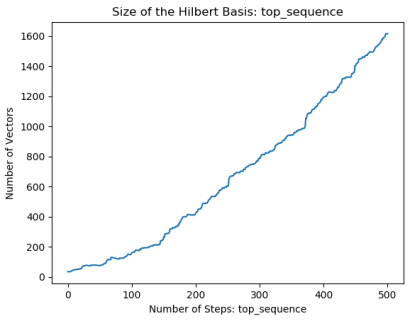
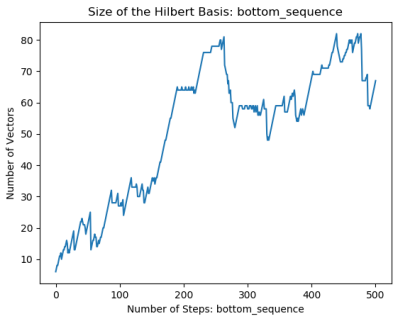
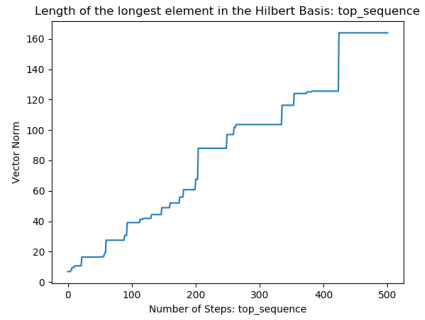
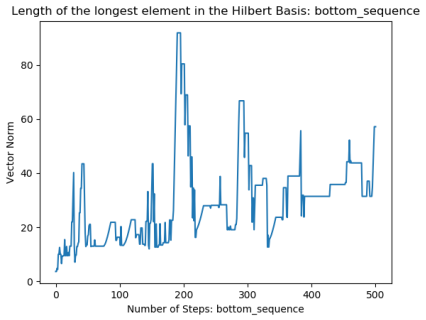
6.3.32 6 generators 2 bound B

Initial Conditions:

```
inner_cone has generators:
[[-2, -1, 1, 1, 2], [-1, -1, 0, 1, 2], [0, -1, -2, 2, 2], [1, -1, -2, 0,
    2], [1, 0, -1, 2, 1], [1, 0, 1, -1, 1]]
outer_cone has generators:
[[-2, -1, -1, 2, 1], [-2, -1, 1, 1, 2], [-2, 0, 1, 2, 2], [1, -1, -2, 0,
    2], [1, 0, -1, 2, 1], [1, 0, 1, -1, 1]]
```

Top Down	Bottom Up

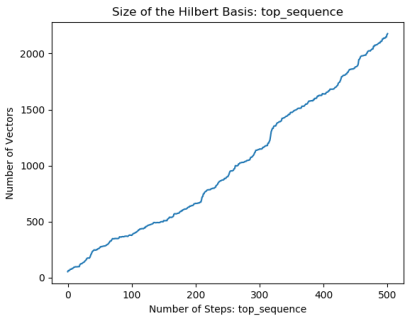
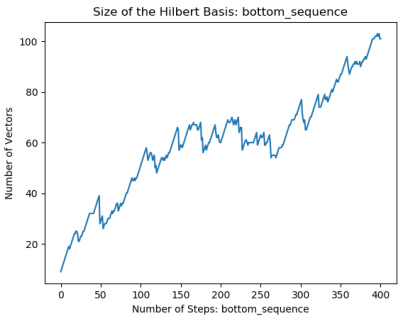
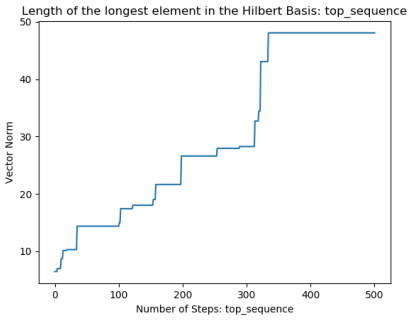
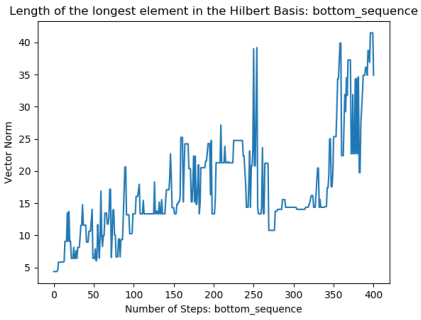
`sequence_complete = False`
`top_sequence has length 501`
`bottom_sequence has length 1`
`cone_poset_chain has length 0`
	`sequence_complete = False`
	`top_sequence has length 1`
	`bottom_sequence has length 501`
	`cone_poset_chain has length 0`

6.3.33 6 generators 2 bound C

Initial Conditions:

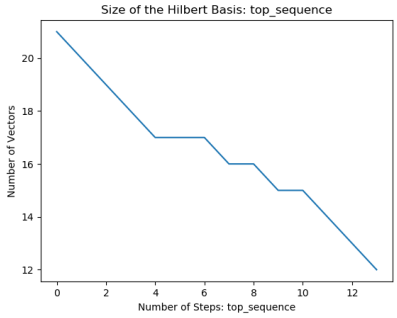
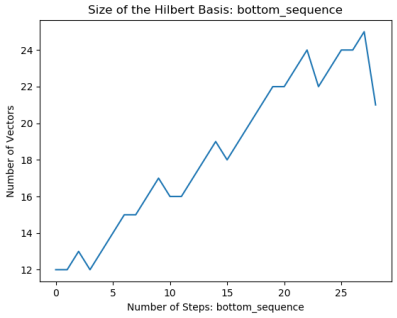
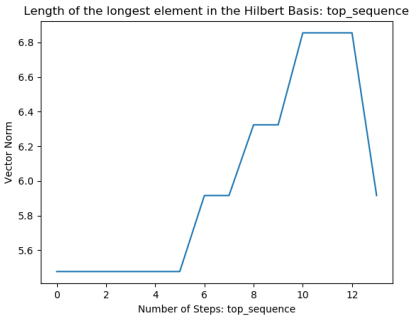
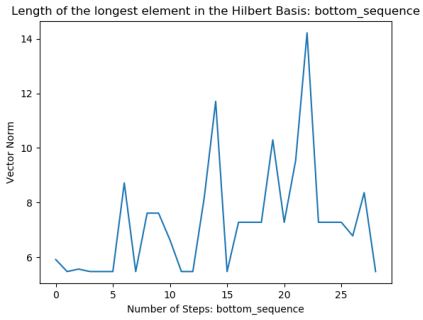
```
inner_cone has generators:
[[-2, -1, -1, 0, 2], [-1, -1, -1, 1, 2], [-1, 0, 0, 1, 2], [-1, 0, 1, 2,
    2], [-1, 2, -1, 2, 2], [0, 0, 0, 1, 1]]
outer_cone has generators:
[[-2, -2, -2, -1, 2], [-1, -1, -2, -1, 1], [-1, 0, 1, 1, 1], [-1, 2, -1,
    2, 2], [0, 0, 2, 0, 1], [2, -2, -2, 2, 1]]
```

Top Down	Bottom Up
sequence_complete = False top_sequence has length 501 bottom_sequence has length 1 cone_poset_chain has length 0	sequence_complete = False top_sequence has length 1 bottom_sequence has length 401 cone_poset_chain has length 0
	
	

6.3.34 6 generators 2 bound D

Initial Conditions:

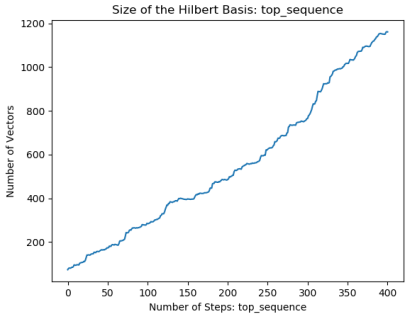
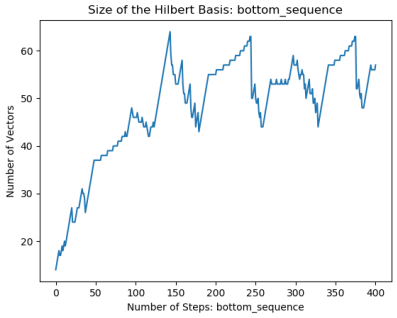
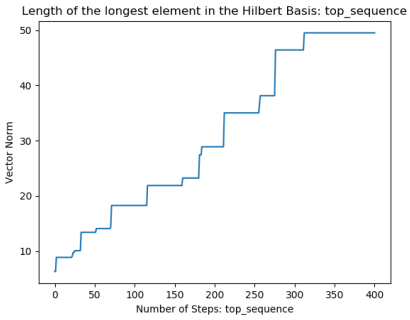
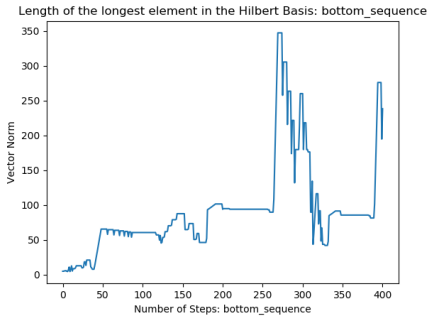
```
inner_cone has generators:
[[-1, -2, -1, 1, 1], [0, -2, -1, 1, 2], [0, -2, -1, 2, 2], [1, -2, 0, 1,
   2], [1, -1, 0, 1, 1], [1, 1, 1, 2, 2]]
outer_cone has generators:
[[-1, -2, -1, 1, 1], [0, -2, -1, -1, 2], [0, -2, -1, 2, 2], [1, -1, 0, 1,
   1], [1, 1, 1, 2, 2], [2, -1, 1, 0, 2]]
```

Top Down	Bottom Up
sequence_complete = True top_sequence has length 14 bottom_sequence has length 1 cone_poset_chain has length 14	sequence_complete = True top_sequence has length 1 bottom_sequence has length 29 cone_poset_chain has length 29
	
	

6.3.35 6 generators 2 bound E

Initial Conditions:

```
inner_cone has generators:
[[-1, -2, 0, -2, 2], [-1, 1, -1, -1, 2], [-1, 1, -1, 0, 2], [-1, 2, 0, 2,
    2], [0, -1, -1, -1, 2], [0, 1, -1, 0, 2]]
outer_cone has generators:
[[-2, 2, 0, 1, 1], [-1, -2, 0, -2, 2], [-1, 2, -2, -1, 2], [0, 0, -1, -2,
    2], [1, -1, -2, 1, 2], [1, 0, 0, 1, 1]]
```

Top Down	Bottom Up
sequence_complete = False top_sequence has length 401 bottom_sequence has length 1 cone_poset_chain has length 0	sequence_complete = False top_sequence has length 1 bottom_sequence has length 401 cone_poset_chain has length 0
	
	

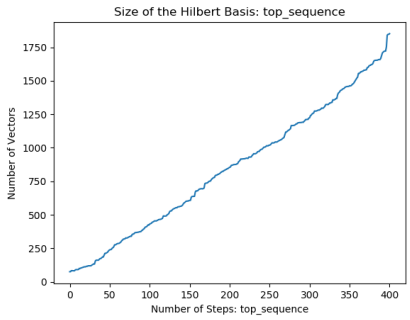
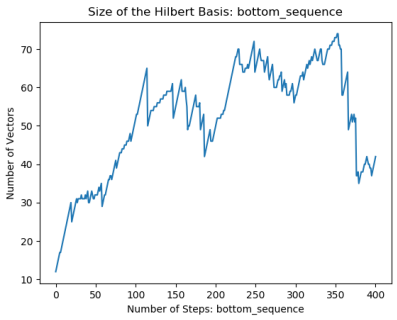
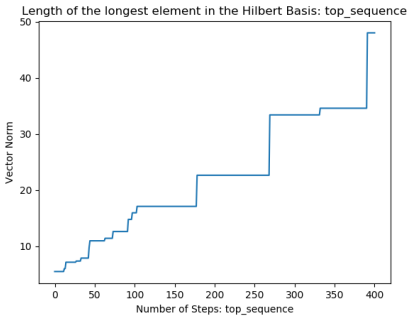
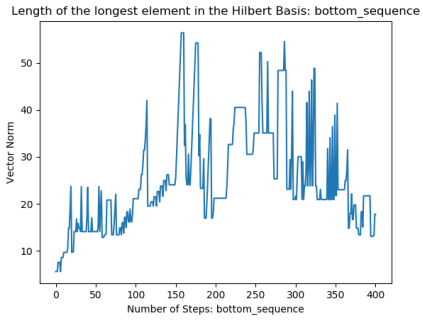
6.3.36 6 generators 2 bound F

Initial Conditions:

```
inner_cone has generators:
[[0, 0, 0, 0, 1], [0, 1, -1, -1, 2], [0, 1, 0, 1, 2], [1, 0, -2, 1, 2],
    [1, 0, -1, 0, 2], [1, 1, 1, 0, 2]]
outer_cone has generators:
[[-2, -1, 2, 0, 2], [-1, 1, 1, 0, 1], [1, -1, 0, 0, 2], [1, 0, -2, 2, 1],
    [1, 1, -2, -2, 1], [2, 1, 2, 0, 2]]
```

Top Down	Bottom Up

`sequence_complete = False`
`top_sequence has length 401`
`bottom_sequence has length 1`
`cone_poset_chain has length 0`
	`sequence_complete = False`
	`top_sequence has length 1`
	`bottom_sequence has length 401`
	`cone_poset_chain has length 0`

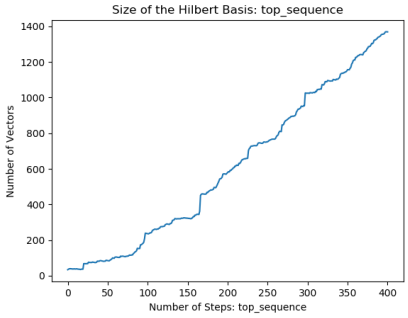
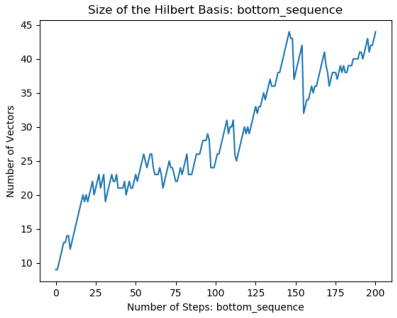
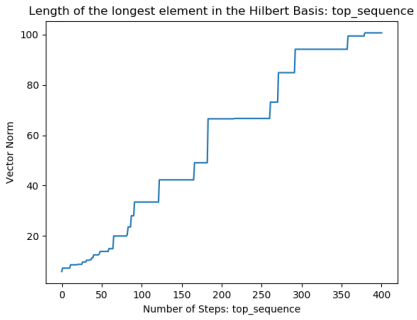
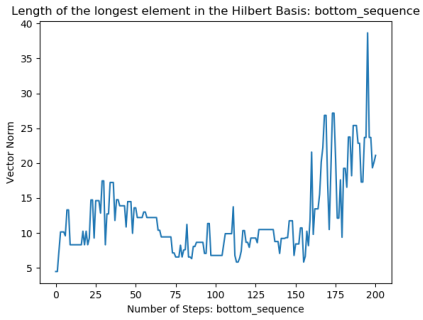
6.3.37 6 generators 2 bound G

Initial Conditions:

```
inner_cone has generators:
[[-2, 2, -2, 1, 2], [-1, 0, 0, 1, 2], [-1, 2, -2, -1, 1], [-1, 2, -2, -1,
    2], [0, -1, 2, -2, 2], [0, 0, -1, 1, 2]]
outer_cone has generators:
[[-1, 0, 0, 2, 1], [-1, 1, -2, 1, 2], [-1, 2, -2, -1, 1], [0, -1, 2, -2,
    2], [1, -1, -1, 2, 2], [2, -1, 0, 2, 2]]
```

Top Down	Bottom Up

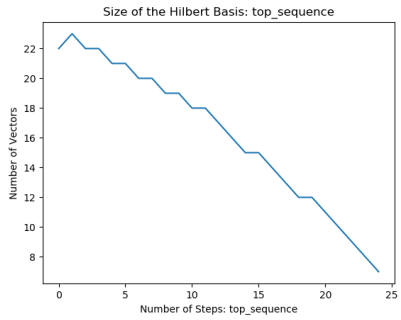
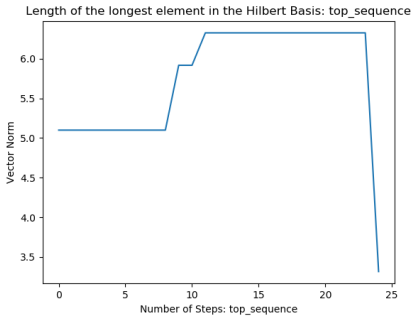
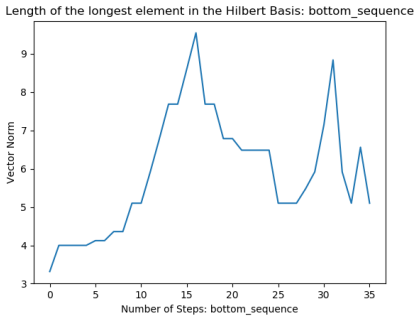
`sequence_complete = False`
`top_sequence has length 401`
`bottom_sequence has length 1`
`cone_poset_chain has length 0`
	`sequence_complete = False`
	`top_sequence has length 1`
	`bottom_sequence has length 201`
	`cone_poset_chain has length 0`

6.3.38 6 generators 2 bound H

Initial Conditions:

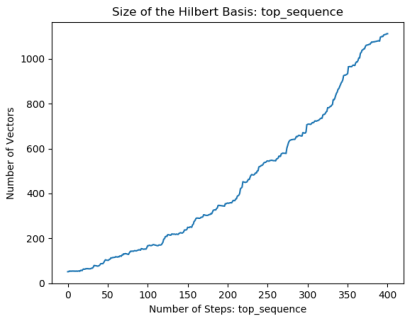
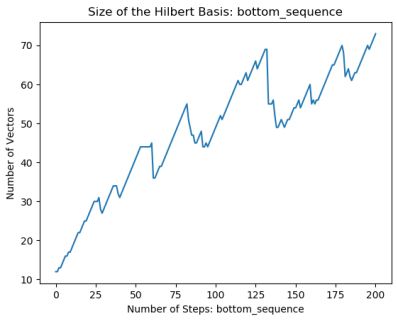
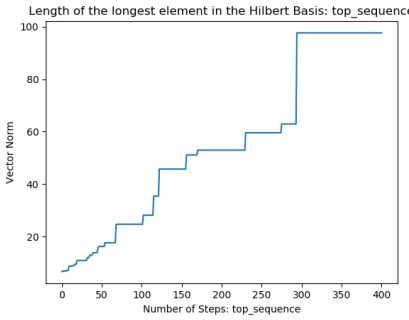
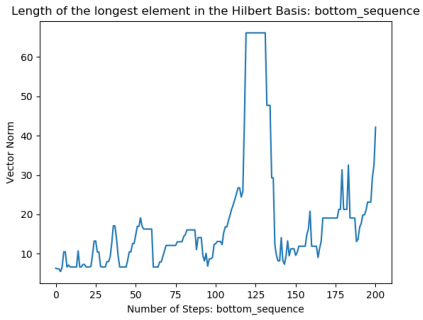
```
inner_cone has generators:
[[-1, 0, -1, 0, 2], [-1, 0, 0, 1, 2], [-1, 1, 0, 1, 2], [0, -1, 0, 0, 2],
    [0, 1, -1, 0, 1], [1, -1, 1, 0, 1]]
outer_cone has generators:
[[-2, 1, -1, 1, 1], [-2, 2, -1, 1, 2], [-1, -2, -2, -1, 2], [0, 1, -1, 0,
    1], [0, 1, 2, 1, 2], [1, -1, 1, 0, 1]]
```

Top Down	Bottom Up
sequence_complete = True top_sequence has length 25 bottom_sequence has length 1 cone_poset_chain has length 25	sequence_complete = True top_sequence has length 1 bottom_sequence has length 36 cone_poset_chain has length 36
	
	

6.3.39 6 generators 2 bound I

Initial Conditions:

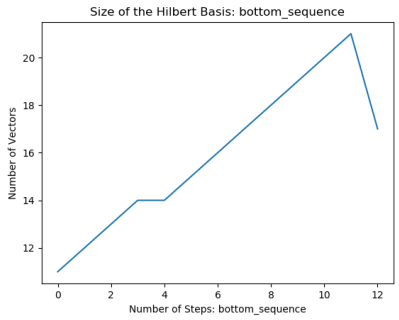
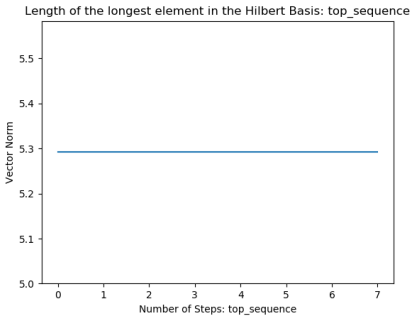
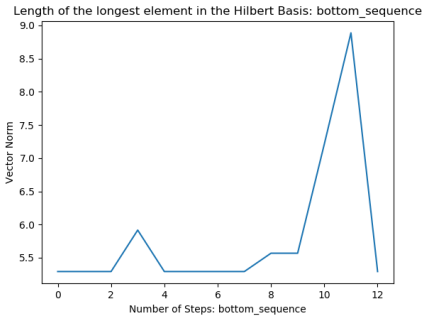
```
inner_cone has generators:
[[-2, 2, 1, -2, 2], [0, 0, 0, 0, 1], [0, 1, 0, 0, 2], [1, 1, -1, 0, 2],
    [1, 1, 0, -1, 2], [2, 2, -1, 1, 2]]
outer_cone has generators:
[[-2, 2, 1, -2, 2], [-1, -1, 0, 1, 2], [0, 1, 1, 1, 2], [1, 1, -2, 0, 2],
    [1, 1, 0, -1, 2], [2, 2, -1, 1, 2]]
```

Top Down	Bottom Up
sequence_complete = False top_sequence has length 401 bottom_sequence has length 1 cone_poset_chain has length 0	sequence_complete = False top_sequence has length 1 bottom_sequence has length 201 cone_poset_chain has length 0
	
	

6.3.40 6 generators 2 bound J

Initial Conditions:

```
inner_cone has generators:
[[-2, 2, 2, -1, 1], [-1, 1, -2, -1, 1], [0, 2, 1, 0, 2], [1, 0, -2, 0, 2],
    [1, 1, 0, 1, 2], [1, 2, 2, 1, 2]]
outer_cone has generators:
[[-2, 2, 2, -1, 1], [-1, 1, -2, -1, 1], [1, 0, -2, 0, 2], [1, 1, 0, 1, 2],
    [1, 1, 2, 1, 1], [1, 2, 0, 0, 2]]
```

Top Down	Bottom Up
sequence_complete = True top_sequence has length 8 bottom_sequence has length 1 cone_poset_chain has length 8	sequence_complete = True top_sequence has length 1 bottom_sequence has length 13 cone_poset_chain has length 13
	
	

6.4 Further Explorations using Alternating algorithm

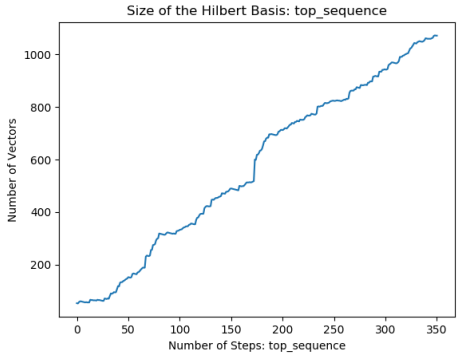
As some of the above experiments did not terminate as we expected, we device an alternating algorithm using a combination of the `bottom_up` and `top_down` algorithms. The results are recorded below. None of the initial conditions with non-terminating results terminated with this alternating method.

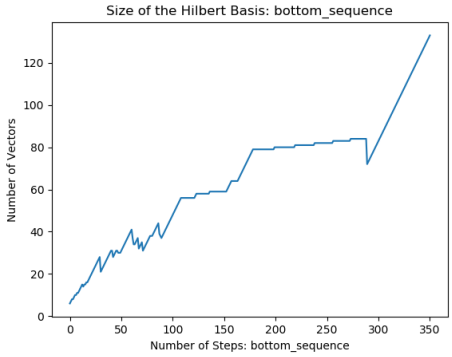
6.4.1 5d 5 generators 2 bound I alternating

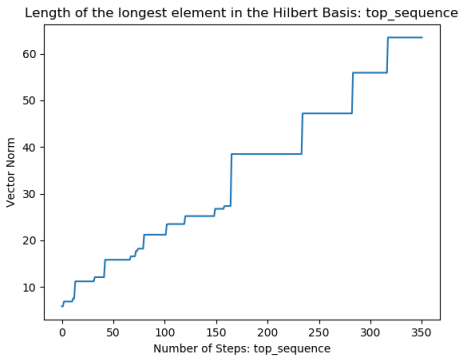
Initial Conditions:

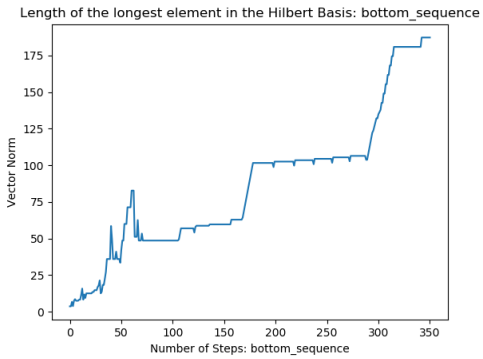
```
inner_cone has generators:
[[0, 1, 0, -1, 2], [1, 1, 2, 0, 2], [1, 2, -2, 0, 1], [1, 2, -2, 2, 1],
    [1, 2, -1, 1, 2]]
outer_cone has generators:
[[-1, 0, 1, -2, 2], [1, -1, -1, -1, 1], [1, 1, 2, 0, 2], [1, 2, -2, -1,
    1], [1, 2, -2, 2, 1]]
        sequence_complete = False
        top_sequence has length 201
        bottom_sequence has length 201
        cone_poset_chain has length 0
```

Top Sequence	Bottom Sequence





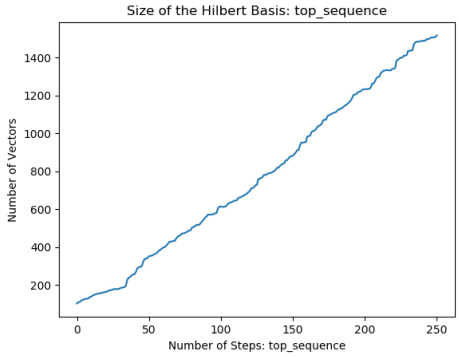
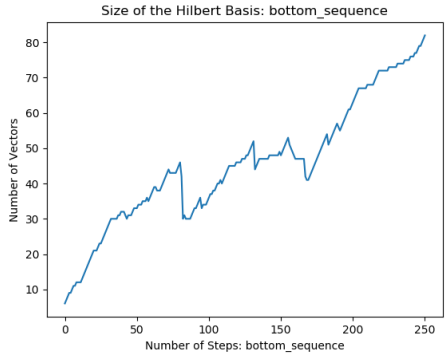
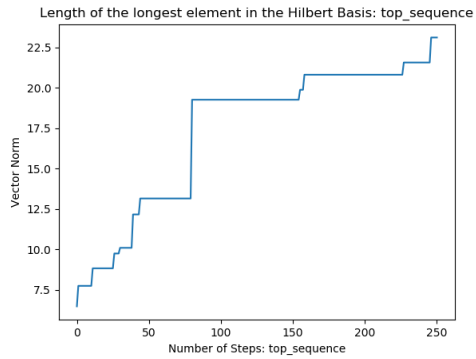
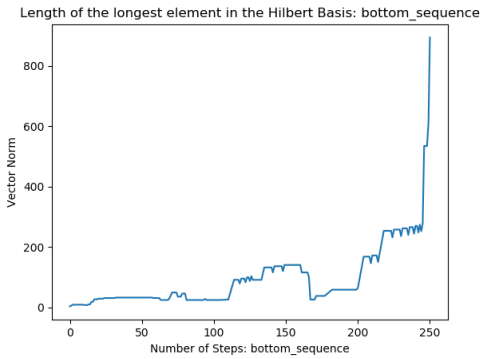




6.4.2 5d 6 generators 2 bound A alternating

Initial Conditions:

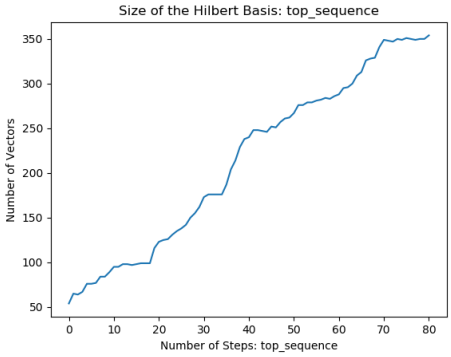
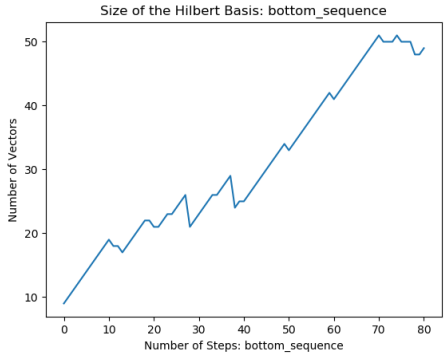
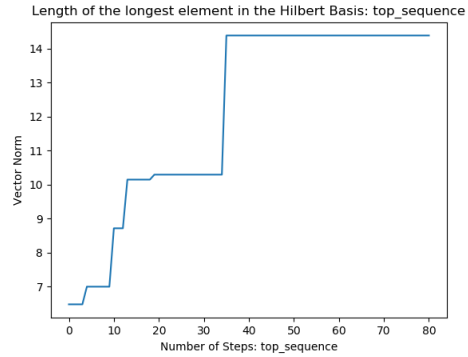
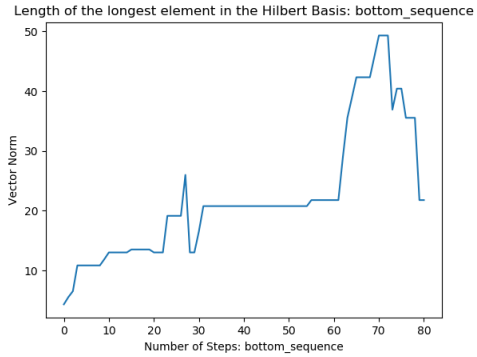
```
inner_cone has generators:
[[1, -1, 1, 0, 1], [2, -2, -2, -1, 1], [2, -2, -1, -1, 2], [2, -2, 0, -1,
    2], [2, -1, 1, 0, 2], [2, -1, 2, 0, 2]]
outer_cone has generators:
[[-2, -2, -2, 1, 2], [-1, 2, 1, 1, 2], [2, -2, -2, -1, 1], [2, -2, 2, -2,
    1], [2, -1, 1, -1, 1], [2, -1, 2, 1, 1]]
        sequence_complete = False
        top_sequence has length 101
        bottom_sequence has length 101
        cone_poset_chain has length 0
```

Top Sequence	Bottom Sequence
	
	

6.4.3 5d 6 generators 2 bound C alternating

Initial Conditions:

```
inner_cone has generators:
[[-2, -1, -1, 0, 2], [-1, -1, -1, 1, 2], [-1, 0, 0, 1, 2], [-1, 0, 1, 2,
   2], [-1, 2, -1, 2, 2], [0, 0, 0, 1, 1]]
outer_cone has generators:
[[-2, -2, -2, -1, 2], [-1, -1, -2, -1, 1], [-1, 0, 1, 1, 1], [-1, 2, -1,
   2, 2], [0, 0, 2, 0, 1], [2, -2, -2, 2, 1]]
       sequence_complete = False
       top_sequence has length 51
       bottom_sequence has length 51
       cone_poset_chain has length 0
```

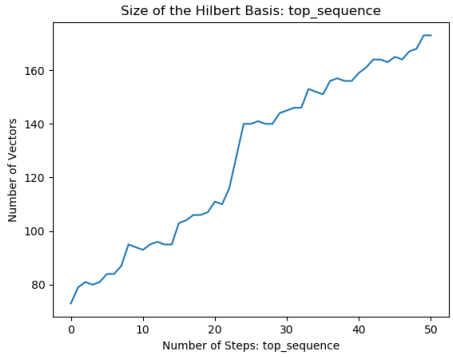
Top Sequence	Bottom Sequence
	
	

6.4.4 5d 6 generators 2 bound E alternating

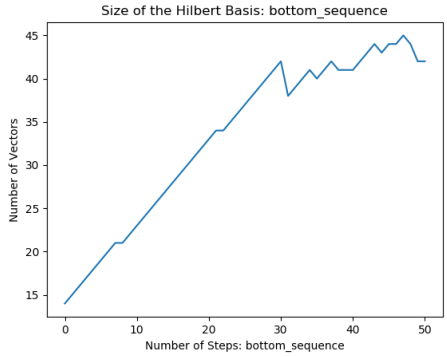
Initial Conditions:

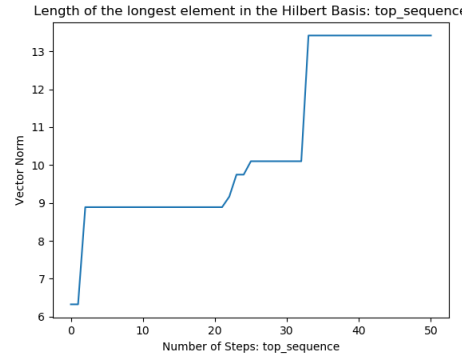
```
inner_cone has generators:
[[-1, -2, 0, -2, 2], [-1, 1, -1, -1, 2], [-1, 1, -1, 0, 2], [-1, 2, 0, 2,
    2], [0, -1, -1, -1, 2], [0, 1, -1, 0, 2]]
outer_cone has generators:
[[-2, 2, 0, 1, 1], [-1, -2, 0, -2, 2], [-1, 2, -2, -1, 2], [0, 0, -1, -2,
    2], [1, -1, -2, 1, 2], [1, 0, 0, 1, 1]]
        sequence_complete = False
        top_sequence has length 51
        bottom_sequence has length 51
        cone_poset_chain has length 0
```

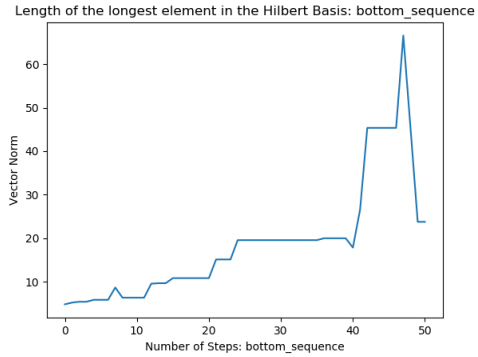
Top Sequence



Bottom Sequence





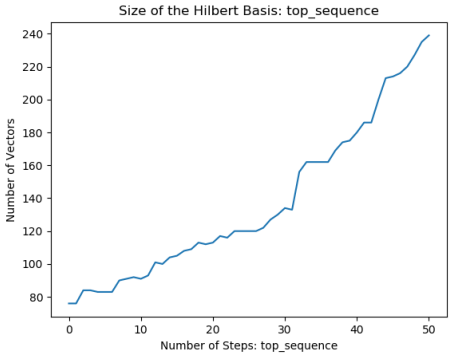


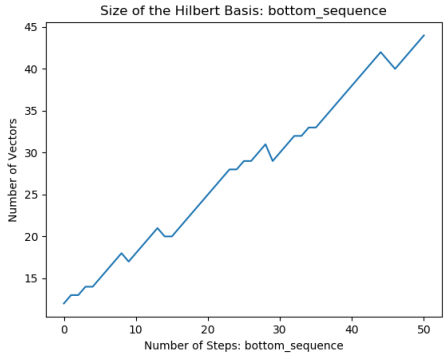
6.4.5 5d 6 generators 2 bound F alternating

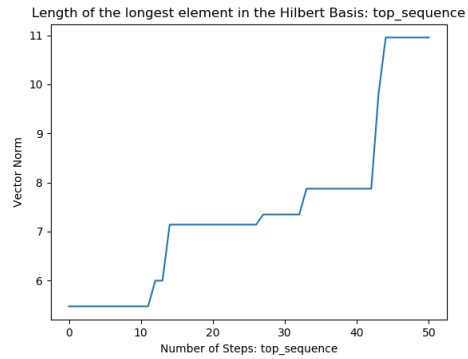
Initial Conditions:

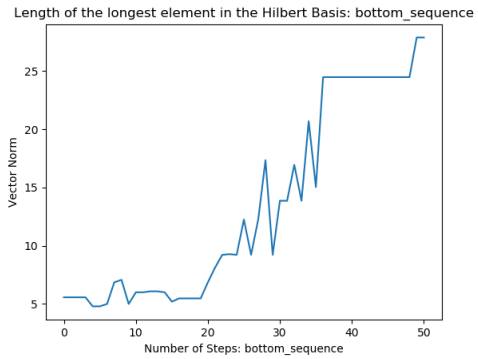
```
inner_cone has generators:
[[0, 0, 0, 0, 1], [0, 1, -1, -1, 2], [0, 1, 0, 1, 2], [1, 0, -2, 1, 2],
    [1, 0, -1, 0, 2], [1, 1, 1, 0, 2]]
outer_cone has generators:
[[-2, -1, 2, 0, 2], [-1, 1, 1, 0, 1], [1, -1, 0, 0, 2], [1, 0, -2, 2, 1],
    [1, 1, -2, -2, 1], [2, 1, 2, 0, 2]]
        sequence_complete = False
        top_sequence has length 51
        bottom_sequence has length 51
        cone_poset_chain has length 0
```

Top Sequence	Bottom Sequence







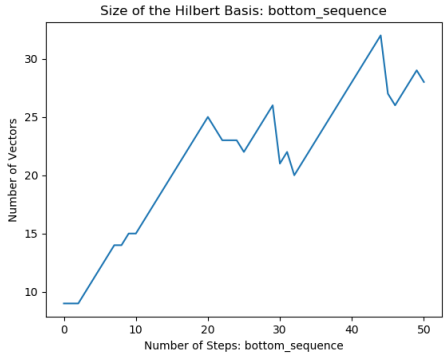


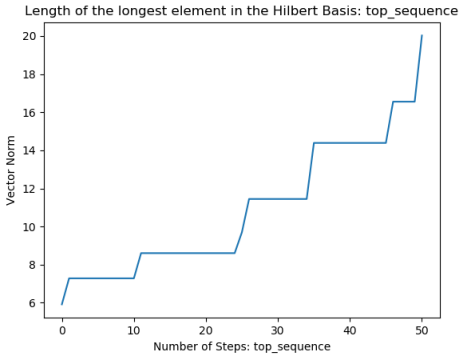
6.4.6 5d 6 generators 2 bound G alternating

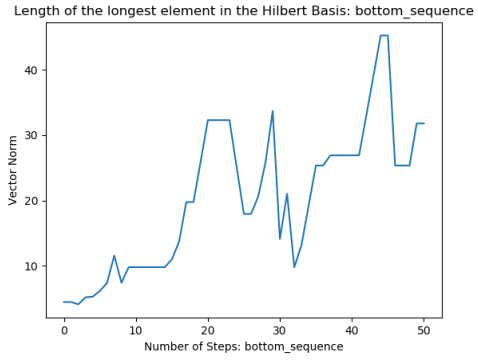
Initial Conditions:

```
inner_cone has generators:
[[-2, 2, -2, 1, 2], [-1, 0, 0, 1, 2], [-1, 2, -2, -1, 1], [-1, 2, -2, -1,
    2], [0, -1, 2, -2, 2], [0, 0, -1, 1, 2]]
outer_cone has generators:
[[-1, 0, 0, 2, 1], [-1, 1, -2, 1, 2], [-1, 2, -2, -1, 1], [0, -1, 2, -2,
    2], [1, -1, -1, 2, 2], [2, -1, 0, 2, 2]]
        sequence_complete = False
        top_sequence has length 51
        bottom_sequence has length 51
        cone_poset_chain has length 0
```

Top Sequence | **Bottom Sequence**





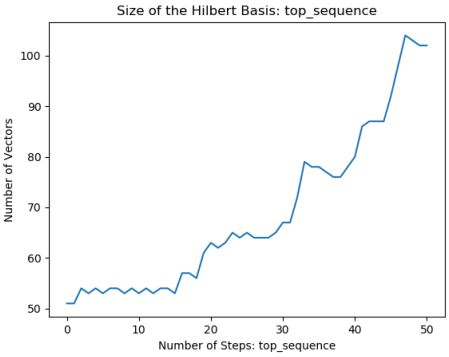


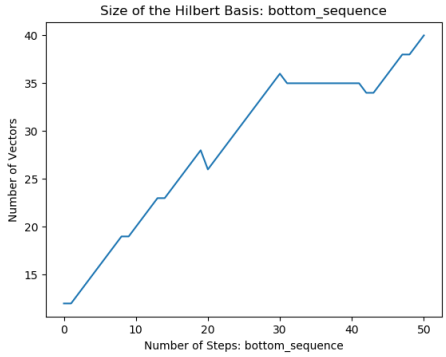
5d 6 generators 2 bound I alternating

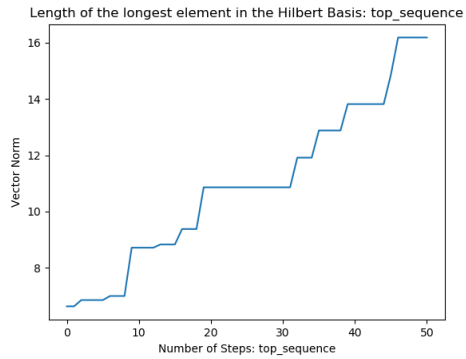
Initial Conditions:

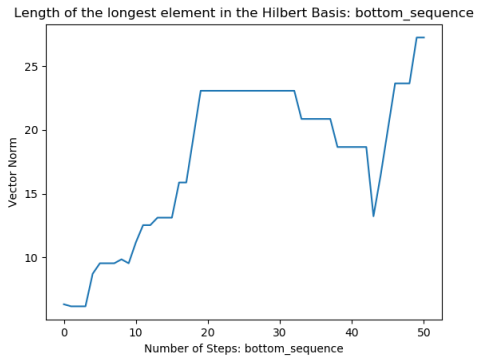
```
inner_cone has generators:
[[-2, 2, 1, -2, 2], [0, 0, 0, 0, 1], [0, 1, 0, 0, 2], [1, 1, -1, 0, 2],
    [1, 1, 0, -1, 2], [2, 2, -1, 1, 2]]
outer_cone has generators:
[[-2, 2, 1, -2, 2], [-1, -1, 0, 1, 2], [0, 1, 1, 1, 2], [1, 1, -2, 0, 2],
    [1, 1, 0, -1, 2], [2, 2, -1, 1, 2]]
        sequence_complete = False
        top_sequence has length 51
        bottom_sequence has length 51
        cone_poset_chain has length 0
```

Top Sequence | **Bottom Sequence**









Chapter 7

Conclusion

The data in dimension 4 and 5 both have examples of non-terminating sequences. The results in **subsections 6.2.20, 6.3.25, 6.3.26, 6.3.29, 6.3.31, 6.3.32, 6.3.33, 6.3.35, 6.3.36, 6.3.37, 6.3.39** show that the top down algorithm will demonstrate a roughly linear increase of the size of Hilbert basis as the number of steps in the algorithm increases. This makes it less and less likely as the number of steps grows that we have a terminating process. This evidence suggests the negative answer to questions (3) and (4) at the end of **section 3.3.4**

Some of the experiments do not terminate on the top down algorithm, but terminate on the bottom up algorithm. This suggests that the bottom up algorithm moves in wider steps than the top down algorithm. The expectation is further supported by the fact that when both procedures terminate, the bottom up algorithm does so in a fewer steps than the top down algorithm.

7.1 Further Study

The algorithms can see some further changes. For example, the extremal generator removed by the top down algorithm perhaps can be chosen based on some "greedy" algorithm instead, where we examine the volume of the cones of the possible choices before choosing. However, this may lead to worse computational efficiency, as at each step one would have to calculate the volume of a cone, usually represented as the volume of the parallelepiped created by the extremal rays.

An exhaustive search of particular classes of cones, or cones in regions might become feasible if we can find a quick way to explicitly generate full dimensional cones in a particular region.

On the computer science side, modifying the project so that it is compatible with `Docker` would allow for computation on a stronger machine, which could provide more certainty towards the questions (3) and (4).

Bibliography

[1] Winfried Bruns. "On the Integral Carathéodory Property". In: *Experimental Mathematics* 16.3 (2007), pp. 359–365. DOI: 10.1080/10586458.2007.10129007.

[2] Winfried Bruns and Joseph Gubeladze. "Normality and covering properties of affine semigroups". In: *Journal für die reine und angewandte Mathematik (Crelles Journal)* 1999.510 (1999). DOI: 10.1515/crll.1999.044.

[3] Winfried Bruns and Joseph Gubeladze. *Polytopes, Rings, and K-theory*. Springer, 2009.

[4] Winfried Bruns, Joseph Gubeladze, and Mateusz Michałek. "Quantum Jumps of Normal Polytopes". In: *Discrete & Computational Geometry* 56.1 (2016), pp. 181–215. DOI: 10.1007/s00454-016-9773-7.

[5] Winfried Bruns et al. "A counterexample to an integer analogue of Carathéodorys theorem". In: *Journal für die reine und angewandte Mathematik (Crelles Journal)* 1999.510 (1999). DOI: 10.1515/crll.1999.045.

[6] Winfried Bruns et al. *Normaliz. Algorithms for rational cones and affine monoids*. Available at https://www.normaliz.uni-osnabrueck.de.

[7] Tsz L Chan. *TimmyChan/ConeThesis*. May 2018. URL: https://github.com/TimmyChan/ConeThesis/.

[8] Robert T. Firla and Günter. M. Ziegler. "Hilbert Bases, Unimodular Triangulations, and Binary Covers of Rational Polyhedral Cones". In: *Discrete & Computational Geometry* 21.2 (1999), pp. 205–216. DOI: 10.1007/pl00009416.

[9] *Gordan's lemma*. Jan. 2018. URL: https://en.wikipedia.org/wiki/Gordan's_lemma.

[10] Joseph Gubeladze. *Normal polytopes*. URL: http://math.sfsu.edu/gubeladze/publications/fpsac2010.pdf.

[11] Joseph Gubeladze and Mateusz Michałek. "The poset of rational cones". In: *Pacific Journal of Mathematics* 292.1 (Jan. 2018), pp. 103–115. DOI: 10.2140/pjm.2018.292.103.

[12] Haase et al. *Existence of unimodular triangulations - positive results*. Dec. 2017. URL: https://arxiv.org/abs/1405.1687v3.

[13] Matt Insall and Eric W Weisstein. *Partially Ordered Set*. Apr. 2018. URL: http://mathworld.wolfram.com/PartiallyOrderedSet.html.

[14] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 8.1)*. http://www.sagemath.org. 2018.

[15] Eric W Weisstein. *Partial Order*. Apr. 2018. URL: http://mathworld.wolfram.com/PartialOrder.html.

[16] Günter M. Ziegler. *Lectures on Polytopes*. Springer, 2007.

Appendices

The appendices include the source code for the following files:

- `experiment_io_tools.py` This is a small package of i/o scripts written for this experiment.
- `batch_run_experiments.py` This is a demonstration of how to create experiments using the `ConeConjectureTester` object
- `batch_continue.py` This is a demonstration of how to load and continue experiments already created.
- `generate_latex_files.py` This is the script used to generate the data latex files.

Appendix A

experiment_io_tools.py

```
""" Module to contain all I/O functions used by
cone conjecture experimet.
"""



import sys


def query_yes_no(question, default="yes"):
        """Ask a yes/no question via raw_input() and return their answer.
        "question" is a string that is presented to the user.
        "default" is the presumed answer if the user just hits <Enter>.
                It must be "yes" (the default), "no" or None (meaning
                an answer is required of the user).
```

```
    The "answer" return value is True for "yes" or False for "no".
    Source: http://code.activestate.com/recipes/577058/)
    """
    valid = {"yes": True, "y": True, "ye": True,
                   "no": False, "n": False}
    if default is None:
           prompt = " [y/n] "
    elif default == "yes":
           prompt = " [Y/n] "
    elif default == "no":
           prompt = " [y/N] "
    else:
           raise ValueError("invalid default answer: '%s'" % default)

    while True:
           sys.stdout.write(question + prompt)
           choice = raw_input().lower()
           if default is not None and choice == '':
                  return valid[default]
           elif choice in valid:
                  return valid[choice]
           else:
                  sys.stdout.write("Please respond with 'yes' or 'no'
```

```
                        "
                                                "(or 'y' or 'n').\n")


def new_screen(header=None):
        """ Prints new screen in terminal regardless of OS """
        print("\033[H\033[J")
        if header is not None:
                boxprint(header)

def boxprint(string,symbol='#'):
        """ prints a box around a string using symbol """
        length = len(string)+4
        mainline = []
        mainline.append(symbol)
        mainline.append(' ')
        mainline.append(string)
        mainline.append(' ')
        mainline.append(symbol)
        mainlinestring = "".join(str(e) for e in mainline)

        horizontalboarder = [symbol for i in range(length)]
        horizontalboarderstring = "".join(str(e) for e in
```

```
            horizontalboarder)
        print(horizontalboarderstring)
        print(mainlinestring)
        print(horizontalboarderstring)

def pause(pausestring="Press Enter to continue..."):
        try:
                input("\n"+pausestring)
        except:
                None
```

Appendix B

batch_run_experiments.py

```
#!/usr/bin/env sage

import sage.all
import os
import cone_conjecture_tester as cct
import string

import argparse

def main(dimension,bound,conditions):
        possiblenames = ["{} generators {} bound {}".format(g,b,string.
            ascii_uppercase[char]) for g in range(dimension,dimension+2)
                for b in range(1,bound+1)
                for char in range(10)]
```

```
possiblenames += ["{} generators {} bound {} bottomup".format(g,b,
    string.ascii_uppercase[char]) for g in range(dimension,
    dimension+2) for b in range(1,bound+1) for char in range(10)]
possiblenames.sort()
print("Possible names: \n{}".format(possiblenames))

alreadyopen = os.listdir("DATA/{}d".format(dimension))
alreadyopen.sort()
print("Already started: \n{}".format(alreadyopen))

for n in range(2):
        numgen = dimension + n
        for condition in range(10):
                #topdown
                mode = 1
                expr_name = "{} generators {} bound {}".format(
                    numgen,bound,string.ascii_uppercase[condition])
                if expr_name not in alreadyopen:
                        print("Beginning {}...".format(expr_name))
                        tester = cct.ConeConjectureTester(dim=
                            dimension,expr_name=expr_name,runmode=
                            mode,batchmode=True,numgen=numgen,rmax=
                            bound)
```

```
        tester.batch_create_experiment()
        tester.run_experiment()
else:
        print("Skipping {}...".format(expr_name))

#bottomup
mode = 2
expr_name_bottomup = expr_name + " bottomup"
if expr_name_bottomup not in alreadyopen:
        tester = cct.ConeConjectureTester(dim=
            dimension,expr_name=expr_name_bottomup,
            runmode=mode,batchmode=True,numgen=numgen
            ,rmax=bound)
        tester.load_file(initial_condition=True,
            custom_name=expr_name) #load up the
            associated top down init conditions
        tester.save_file("Initial Conditions")
        tester.save_file()
        tester.save_summary()

        print("Beginning {}...".format(
            expr_name_bottomup))
        tester.run_mode = 2
```

```
                        tester.batch_mode = True
                        tester.run_experiment()
                else:
                        print("Skipping {}...".format(
                            expr_name_bottomup))

if __name__ == "__main__":
        parser = argparse.ArgumentParser(description="Creates new
            experiments if they are not yet open.")
        parser.add_argument("dimension", type=int, default=5, help="This
            is the dimension we are testing")
        parser.add_argument("bound", type=int, default=2, help="This is
            the bound on the absolute value of the coordinates")
        parser.add_argument("conditions", type=int, default=10, help="The
            number of experiments (unique initial conditions)")
        args = parser.parse_args()

        main(args.dimension, args.bound, args.conditions)
```

Appendix C

batch_continue.py

```
#!/usr/bin/env sage

import sage.all
import os
import cone_conjecture_tester as cct
import string
import experiment_io_tools
import time

if __name__ == '__main__':
        '''accept_dimension = False
        while not accept_dimension:
                dimension = experiment_io_tools.ask_int("Dimension? ")
                if dimension >1 :
```

```
                accept_dimension = True
        else:
                print("\tEnter valid dimension please.")
'''
steps = 200
accept_steps = experiment_io_tools.query_yes_no("Current number of
    steps to continue is [{}]. Keep settings?".format(steps))
while not accept_steps:
        steps = experiment_io_tools.ask_int("Steps? ")
        if steps >= 1:
                accept_steps = True
        else:
                print("\tEnter a positive integer please.")

run_time = 30 # MINUTES

accept_time = experiment_io_tools.query_yes_no("Run for {} minutes
    . Keep settings?".format(run_time))
while not accept_time:
        run_time = experiment_io_tools.ask_int("Time limit? ")
        if run_time > 0:
                accept_time = True
        if run_time <= 0:
```

```
                print("\tEnter a positive integer please.")
start_time = time.time()
finish_time = start_time + 60* run_time # run this for 30 minutes

while time.time() <= finish_time:
        for dimension in range(4,6):
                open_experiments = os.listdir("DATA/{}d".format(
                    dimension))
                open_experiments.sort()

                for experiment in open_experiments:
                        try:
                                tester = cct.ConeConjectureTester(
                                    dimension,expr_name=experiment,
                                    batchmode=True,steps=steps)
                                tester.load_file()
                                if not tester.current_cone_chain.
                                    sequence_complete:
                                        tester.run_experiment()
                                        tester.print_graphs()
                                        tester.save_file()
                                        tester.save_summary()
                                else:
```

```
                        print("{} already complete.
                            Skipping...".format(
                            experiment))

                except:
                        with open("batch_errors.log",'a') as
                            fp:
                            fp.write("{} Error loading/
                                saving ".format(time.time()
                                ) + experiment + "\n")
                            fp.close()
                        print("Error loading/saving " +
                            experiment +". Logged and moving
                            on...")

                print("\t\t\tRUN TIME: {} seconds".format(
                    round(time.time()-start_time,2)))
                if time.time()> finish_time:
                        break

        if time.time()> finish_time:
                break
```

Appendix D

generate_latex_files.py

```
#!/usr/bin/env sage


import sage.all
import os
import string


if __name__ == '__main__':
        alphabet = [string.ascii_uppercase[char] for char in range(10)]
        for dim in [4,5]:
                for letter in alphabet:
                        for b in range(dim -3):
                                if dim == 4:
                                        bound = b +2
                                else:
```

```
        bound = b+1
for i in range(2):
        numgen = i + dim

        experiment_topdown = "{} generators {}
             bound {}".format(numgen, bound,
            letter)
        directory_topdown = "DATA/{}d/".format
            (dim)+experiment_topdown +"/"

        experiment_bottomup = "{} generators
            {} bound {} bottomup".format(
            numgen, bound, letter)
        directory_bottomup = "DATA/{}d/".
            format(dim)+experiment_bottomup +"
            /"

        latex_directory = "Python Generated
            Latex Files/"

        topdown_summary_path =
            directory_topdown + "Data Summary.
            txt"
```

```
bottomup_summary_path =
    directory_bottomup + "Data Summary
    .txt"

topdown_summary_string = ""
topdown_string = ""
bottomup_summary_string = ""
bottomup_string = ""

with open(topdown_summary_path,'r') as
     fp:
        topdown_array = fp.readlines()
        fp.close()

i = 0
for line in topdown_array:
        if i < 4:
                topdown_summary_string
                    += line
        else:
                topdown_string += line
                    [1:]
        i +=1
```

```
with open(bottomup_summary_path,'r')
    as fp:
        bottomup_array = fp.readlines
            ()
        fp.close()


i = 0
for line in bottomup_array:
        if i < 4:
                bottomup_summary_string
                     += line
        else:
                bottomup_string += line
                    [1:]
        i +=1


initial_conditions_string = ""
if topdown_summary_string <>
    bottomup_summary_string:
        print("Check this experiment:
```

```
            {}".format(
            experiment_topdown))
        initial_conditions_string += "
            Topdown:\n"+
            topdown_summary_string + "\
            nBottomup:\n"+
            bottomup_summary_string
else:
        initial_conditions_string =
            topdown_summary_string

latex_preamble =  "\\documentclass[10
    pt]{article}\n\\begin{document}\n"
latex_init_condit = "\\textbf{Initial
    Conditions:}\n\\begin{SAGE}\n" +
    initial_conditions_string + "\n\\
    end{SAGE}\n"
latex_tabular =  "\\begin{tabular}{c|c
    }\n\\textbf{Top Down} & \\textbf{
    Bottom Up} \\\\ \\hline \n\\begin{
    SAGE}\n"
latex_tabular +=  topdown_string + "\\
    end{SAGE} \n&\n\\begin{SAGE}\n"
```

```
latex_tabular += bottomup_string + "\\
    end{SAGE} \n\\\\ \\hline\n\n"
latex_tabular +=  '\\begin{minipage
    }{.45\\textwidth}\n\\
    includegraphics[width=\\textwidth
    ]{"'+"DATA/{}d/".format(dim)+
    experiment_topdown+'/top_sequence
    SIZE"}\n'
latex_tabular +=  "\\end{minipage} &\n
    "
latex_tabular += '\\begin{minipage
    }{.45\\textwidth}\n\\
    includegraphics[width=\\textwidth
    ]{"'+"DATA/{}d/".format(dim)+
    experiment_bottomup+'/
    bottom_sequence SIZE"}\n'
latex_tabular +=  "\\end{minipage}
    \\\\ \\\\\n\\hline \\\\"
latex_tabular +=  '\\begin{minipage
    }{.45\\textwidth}\n\\
    includegraphics[width=\\textwidth
    ]{"'+"DATA/{}d/".format(dim)+
    experiment_topdown+'/top_sequence
```

```
    LENGTH"}\n'
latex_tabular +=  "\\end{minipage} &\n
    "
latex_tabular += '\\begin{minipage
    }{.45\\textwidth}\n\\
    includegraphics[width=\\textwidth
    ]{"'+"DATA/{}d/".format(dim)+
    experiment_bottomup+'/
    bottom_sequence LENGTH"}\n'
latex_tabular +=  "\\end{minipage}\n\\
    end{tabular}\n\\end{document}"

latex_code = latex_preamble +
    latex_init_condit + latex_tabular

with open(latex_directory + "{}d ".
    format(dim) + experiment_topdown +
     ".tex", 'w') as fp:
        fp.write(latex_code)
```