Exploring Generative Models in Hyperbolic Space

Raymond Liu

Advisor: Professor Ryan Adams

Submitted in partial fulfillment

of the requirements for the degree of

Bachelor of Science in Engineering

Department of Computer Science

Princeton University

May 2023

Abstract

Generative models are powerful machine learning models that have the ability to probabilistically generate unique, realistic samples of data. These models can generate anything from hyper-realistic images of faces to 3D models of chairs. Recent publicly-available generative models such as ChatGPT and DALL-E have gained widespread attention and usage. However, generative models often have high-dimensional latent space representations of the data, which results in the latent space vectors used to generate outputs being difficult to interpret.

In this paper, we introduce an application for exploring generative models in hyperbolic space. We adopt the hyperboloid model of hyperbolic geometry and implement a regular tiling system for this model. We implement the hyperbolic world in the Unity game engine and link the hyperbolic world with a Flask server that produces output images from LAFITE, a state-of-the-art generative model. Finally, we run simulated experiments to evaluate the effectiveness of our system as a tool for human-in-the-loop optimization of generative models of varying dimensions.

This project is published online[1] using Unity WebGL.

Code for the Unity implementation is available at:

`https://github.com/rl27/MercatorUnity`

Code for the model server and the midpoint server is available at:

`https://github.com/rl27/MercatorUnityServer`.

[1] `https://play.unity.com/mg/other/build-rj5-1`

Acknowledgements

I would like to express my deepest gratitude to my advisor, Prof. Ryan Adams, for his guidance and support throughout the long history of this project. His feedback, expertise, and encouragement have been invaluable to me. I am also grateful to Jeffrey Cheng, who was an invaluable source of help during this project's conception.

I am deeply thankful for my family and my friends, all of whom have been a constant source of encouragement and support throughout my thesis and my academic career.

To Dad - my star & my reason.

Contents

List of Algorithms

List of Figures

Chapter 1

Introduction

Generative models are machine learning models that have the powerful ability to learn the features of a given set of data, then use these features to probabilistically generate unique, realistic samples. These models can generate a wide variety of high-quality content such as hyper-realistic images of faces, 3D models of chairs, and melodies. Recent publicly-available generative models such as ChatGPT and DALL-E have gained widespread attention and usage. However, generative models can have high-dimensional latent space representations of the data, which results in the latent space vectors used to generate outputs being difficult to interpret.

In this thesis, we introduce a system that allows users to visualize and exploring correlated outputs from generative models in hyperbolic space. Unlike Euclidean space, hyperbolic space expands exponentially due to having negative curvature, which makes it far less limiting than Euclidean space to explore, and can potentially allow a user to explore an enormous distance in a short period of time. Our system makes use of this by displaying generated images in a simulated hyperbolic world, with outputs being correlated to each other based on distance within the world. Users can easily and intuitively visualise correlated outputs and attempt to find desired outputs through exploration of this world.

Chapter 2

Background and Related Work

Hyperbolic space is a generally under-explored topic, but there have been a few prominent simulations of hyperbolic space. HyperRogue, originally released in 2011, is a 2D roguelike video game played from the top-down perspective that allows players to explore an infinitely-generated hyperbolic world [8]. Hyperbolica, released in 2022, is a 3D puzzle video game that similarly takes place in a hyperbolic world [5]. While these projects focus on the novelty of hyperbolic geometry, we additionally focus on applications of generative models.

Interactive optimization of latent spaces is something that has been tackled in many different ways and with many different kinds of content. The work of Zhou et al. involved displaying four probabilistically generated melodies at a time, with users selecting a melody to then use to generate four more, with there being a slider to control exploration vs. exploitation [19, 18]. Chiu et al. used generated images of faces with a slider to search over 1D subspaces of the latent space [3], while Chong et al. allowed users to edit images of faces before generating more [4]. Brochu et al. used generated animations and allowed users to tune a variety of parameters [2]. In this work, we introduce a new system that uses regular tilings in hyperbolic space with image outputs, although theoretically our work could be extended to generate and display any kind of output.

Many other latent space visualizations are static and represented on 2D scatter plots, either directly using two intrinsic latent dimensions or using axes from dimensionality reduction methods such as t-SNE or PCA [9]. They do not allow for exploration of the latent space, and the representation can easily be incorrect or misinterpreted due to its lack of dimensionality and the latent spaces' lack of physical units, leaving the original latent space vague [1]. Our system aims to address this problem through the use of an explorable hyperbolic world.

Chapter 3

Approach

3.1 Hyperbolic Geometry

3.1.1 The Hyperboloid Model

There are many different models of hyperbolic geometry, including the Poincaré disk, the Poincaré half-plane, Klein-Beltrami, the hemisphere, and the hyperboloid. Each model has its own advantages and disadvantages. For instance, the Poincaré disk preserves angles, but lines are arcs and can appear curved. On the other hand, Klein-Beltrami preserves straight lines, but not angles.

We chose to use the hyperboloid model of hyperbolic geometry internally, as it is an intuitive, easy-to-understand model with multiple analogs to Euclidean and spherical geometry. Subsequent sections and described implementations will also use the hyperboloid model.

Coordinate System Convention

The Unity game engine uses a Y-up coordinate system, meaning the Y-axis is a vertical line in the game world, and the XZ plane is horizontal. Therefore, given a vector $v = (v_0, v_1, v_2)$, verticality is represented by v_1. The rest of this thesis will follow this convention.

Our hyperboloid model is thus represented by the equation

$$y^2 = 1 + x^2 + z^2 \tag{3.1}$$

This defines a hyperboloid of two sheets; our system uses only the positive-Y sheet.

3.1.2 Disk Projections

The hyperboloid model is infeasible for users to interact with in a Euclidean game engine. Therefore, we use a projection of the hyperboloid model. The Poincaré disk model can be obtained by stereographic projection of points on the hyperboloid through the $y = 0$ plane to the point $(0, -1, 0)$, so a point (a, b, c) is projected to $(\frac{a}{b+1}, 0, \frac{c}{b+1})$. The Klein-Beltrami model can be obtained by projection through the $y = 1$ plane to $(0, 0, 0)$, so a point (a, b, c) is projected to $(\frac{a}{b}, 0, \frac{c}{b})$. These projections are visualised in Figure 3.1. The Poincaré disk projection is the default projection used in the Unity game world, although users also have the option of using the Klein-Beltrami projection.

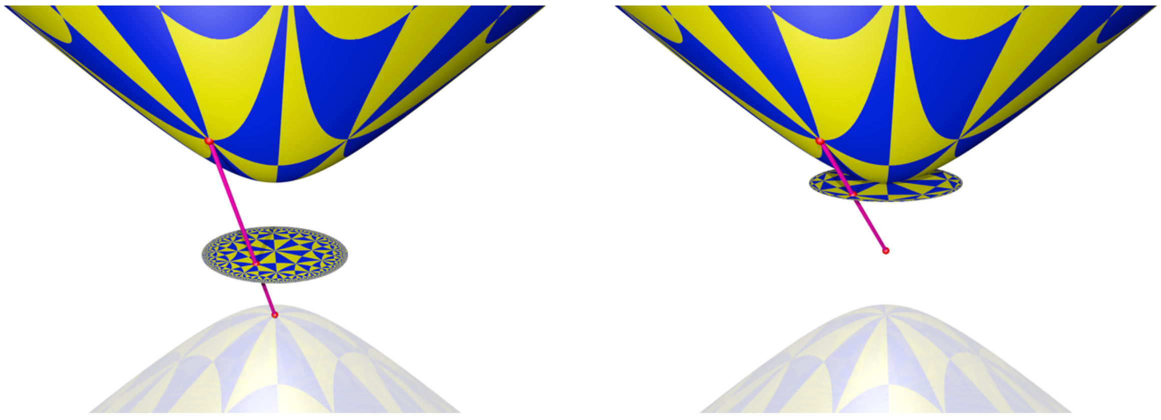

(a) Projection to the Poincaré disk (b) Projection to the Klein-Beltrami model

Figure 3.1: Projections of the hyperboloid model. From Gagern [16].

3.2 Geometry on the Hyperboloid

Given a vector $x = (x_0, x_1, x_2)$ in Minkowski 3-space, the Minkowski quadratic form is conventionally defined as $q(x) = -x_0^2 + x_1^2 + x_2^2$. However, because we are using a Y-up coordinate system, we instead define the quadratic form as

$$q(x) = -x_1^2 + x_0^2 + x_2^2 \tag{3.2}$$

Similarly, the corresponding Minkowski bilinear form for our system is defined as

$$p(x, y) = -x_1 y_1 + x_0 y_0 + x_2 y_2 \tag{3.3}$$

The quadratic form q is analogous to taking the squared magnitude of a vector in Euclidean space; the bilinear form p is analogous to taking the inner product of two vectors in Euclidean space.

Distances

The distance between two points x, y on the hyperboloid is given by

$$\text{dist}(x, y) = \cosh^{-1}(-p(x, y)) \tag{3.4}$$

Normalization

Normalization in hyperbolic space is analogous to Euclidean space. A vector v in hyperbolic space is normalized by dividing by the square root of its quadratic form.

$$\text{norm}(v) = \frac{v}{\sqrt{|q(v)|}} \tag{3.5}$$

The vector $\text{norm}(v)$ satisfies the property that $q(\text{norm}(v)) = \pm 1$. Normalization preserves angles between vectors.

Angles

Given points A, B, C on the hyperboloid, the angle $\angle BAC$ between the two geodesic lines $\overrightarrow{AB}$ and $\overrightarrow{AC}$ is measured by calculating the tangents at the point of intersection. Let v and w be the tangent vectors. The angle is given by

$$\angle vw = \cos^{-1}\left(\frac{p(v,w)}{\sqrt{q(v)\cdot q(w)}}\right) \tag{3.6}$$

Note that this is essentially the same as normalizing v and w first, then performing $\cos^{-1}(p(v,w))$.

Translations

A vector v can be translated by t units in the x direction by performing

$$L_x(t)v = \begin{bmatrix} \cosh t & \sinh t & 0 \\ \sinh t & \cosh t & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_0 \\ v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} v_0 \cosh t + v_1 \sinh t \\ v_0 \sinh t + v_1 \cosh t \\ v_2 \end{bmatrix} \tag{3.7}$$

Similarly, a translation by t units in the z direction is given by

$$L_z(t)v = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cosh t & \sinh t \\ 0 & \sinh t & \cosh t \end{bmatrix} \begin{bmatrix} v_0 \\ v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} v_0 \\ v_2 \sinh t + v_1 \cosh t \\ v_2 \cosh t + v_1 \sinh t \end{bmatrix} \tag{3.8}$$

Translations preserve angles and straight lines.

Rotations

A vector v can be rotated by θ radians counter-clockwise around the hyperboloid by performing

$$R(\theta)v = \begin{bmatrix} \cos\theta & 0 & -\sin\theta \\ 0 & 1 & 0 \\ \sin\theta & 0 & \cos\theta \end{bmatrix} \begin{bmatrix} v_0 \\ v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} v_0\cos\theta - v_2\sin\theta \\ v_1 \\ v_0\sin\theta + v_2\cos\theta \end{bmatrix} \tag{3.9}$$

Geodesics

The intersection between the hyperboloid and any plane that contains the origin defines a straight line on the hyperboloid.

Given two basis vectors u and w of the plane that satisfy the properties $q(u) = 1$, $q(w) = -1$, and $p(u, w) = 0$, the following parametric equation defines the geodesic g passing through u and w.

$$g(u, w, t) = u\cosh t + w\sinh t \tag{3.10}$$

The value t represents the hyperbolic distance from u to $g(t)$ in the direction of w.

Given two points u and v on the hyperboloid, we can derive the vector w that satisfies the properties listed above, i.e. $p(u, w) = 0$ and $q(w) = -1$.

$$\begin{aligned} \operatorname{dir}(u, v) = w &= \operatorname{norm}(v - u \cdot \frac{p(u,v)}{q(u)}) \\ &= \frac{v - u \cdot \frac{p(u,v)}{q(u)}}{\sqrt{q(v - u \cdot \frac{p(u,v)}{q(u)})}} \end{aligned} \tag{3.11}$$

Midpoints

The midpoint on the hyperboloid between two points u and v is the point lying on the geodesic containing u and v and equidistant to u and v. It has a simple formula.

$$\operatorname{mid}(u, v) = \operatorname{norm}(u + v) \tag{3.12}$$

Tangent vectors

Given two points u and v on the hyperboloid, the line tangent to the geodesic containing u and v at point u is defined by

$$l(t) = u + t \cdot \left(\frac{v - u \cdot \frac{p(u,v)}{q(u)}}{\sqrt{q(v - u \cdot \frac{p(u,v)}{q(u)})}} \right) \tag{3.13}$$

3.3 Regular Tilings

Hyperbolic space expands exponentially with distance travelled. While this property is useful for the purpose of exploration, it also becomes impossible to properly represent every coordinate in a reasonably-sized hyperbolic world using floating-point numbers. Inspired by the tiling systems used in HyperRogue and Hyperbolica [8, 5], we use a tiling system so that the system only needs to deal with tiles and coordinates local to the user.

A tiling consists of regular n-gons, with k of these meeting at each vertex. In order for an integer pair (n, k) to define a hyperbolic tiling, it must satisfy the inequality $(n - 2)(k - 2) > 4$. The derivation for this inequality is given in (A.1) in Appendix A. Users may select any n and k that are valid according to this inequality, with the system default being $(n, k) = (4, 5)$. Figure 3.2 illustrates a $(4, 5)$ tiling.

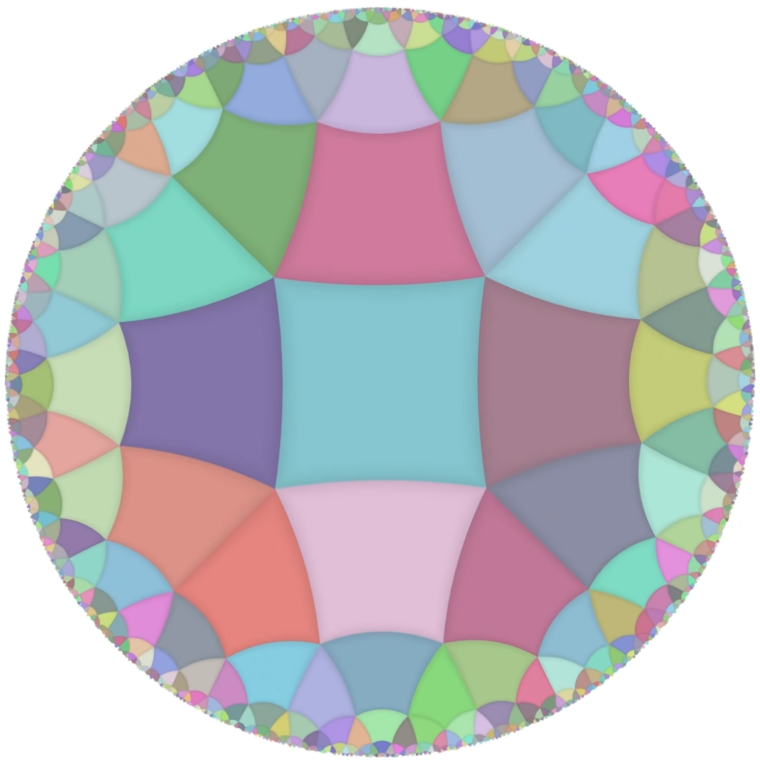


Figure 3.2: A (4,5) regular tiling. From Hyperbolica [5].

3.3.1 Tile Size

The size of each n-gon depends on both n and k - larger values of n and/or k mean each n-gon must be larger in order for the tiling to work. Given an integer pair (n, k), suppose that an n-gon centered at $(0, 1, 0)$ on the hyperboloid has a vertex $v_0 = (x, \sqrt{x^2 + 1}, 0)$ on the hyperboloid. The value of x may be calculated in terms of n and k:

$$x = \sqrt{\cot^2 \frac{\pi}{n} \cot^2 \frac{\pi}{k} - 1} \tag{3.14}$$

We present our derivation for this equation in (A.2) in Appendix A.

Following this, for an integer pair (n, k), in order to construct the first regular n-gon on the hyperboloid, we calculate $x = \sqrt{\cot^2 \frac{\pi}{n} \cot^2 \frac{\pi}{k} - 1}$, then place the first vertex at the point $v_0 = (x, \sqrt{x^2 + 1}, 0)$. The remaining vertex locations are found by rotating v_0 around the hyperboloid in increments of $\frac{2\pi}{n}$.

3.4 Latent Vector Generation

The tiling system presented in Section 3.3 provides a convenient approach to generating new samples - each tile corresponds to one latent space vector, each of which is used by the generative model to generate a sample. The latent space vectors are sampled using Gaussian process (GP) regression. The posterior covariance between nearby previously-generated vectors and to-be-generated vectors is calculated based on the geodesic distance between the corresponding tiles in the hyperbolic world. Lastly, latent space vectors are drawn from a multivariate Gaussian distribution using the posterior covariance. As users explore the hyperbolic world, new samples are generated on nearby tiles that do not already have a sample.

We first define the kernel function k, which takes two sets of coordinates x, x'. This is a squared exponential kernel with two hyperparameters - the signal variance σ^2 and the lengthscale ℓ.

$$k(x, x') = \sigma^2 \exp\left(-\frac{\text{dist}(x, x')^2}{\ell}\right) = \sigma^2 \exp\left(-\frac{\cosh^{-1}(-p(x, x'))^2}{\ell}\right) \tag{3.15}$$

Then, given two sets of tile coordinates a and b, we can compute the covariance matrix K. Let $K_{i,j}$ be the value in row i, column j of K. Let a_i be the i-th coordinate in a and let b_j be the j-th coordinate in b. Then for each $K_{i,j}$, we calculate $K_{i,j} = k(a_i, b_j)$.

We then compute three covariance matrices. Let x_0 be the training points - the set of m coordinates of nearby tiles that already have a corresponding latent space vector. Let x_1 be the testing points - set of n coordinates of new tiles for which we will generate latent space vectors.

- The training kernel matrix K_0 uses x_0, x_0 as inputs and has dimension $m \times m$.
- The training-testing kernel matrix K_1 uses x_0, x_1 and has dimension $m \times n$.
- The testing kernel matrix K_2 uses x_1, x_1 and has dimension $n \times n$.

From these matrices we compute the posterior covariance:

$$\Sigma = K_2 - K_1^\top K_0^{-1} K_1 \tag{3.16}$$

Let the dimensionality of the latent space be d. Let x be the set of latent space vectors corresponding to the tile coordinates in a. For each dimension i in the latent space, slice x by taking $s_i = [x_0[i], x_1[i], ..., x_m[i]]^\top$, then compute the posterior mean:

$$\mu_i = K_1^\top K_0^{-1} s_i \tag{3.17}$$

Then sample a vector v_i of size n from the multivariate normal distribution $\mathcal{N}_n(\mu_i, \Sigma)$.

Finally, after repeating this process for every dimension i, we obtain d vectors of size n. We collect these vectors into a matrix with dimension $n \times d$, which gives n new latent vectors, each of size d. These latent vectors are used to generate the images corresponding to the n tiles whose coordinates are contained in b.

The hyperparameters to the Gaussian process σ^2 and ℓ affect how similar the generated images are. These are set by the user and can be altered at any time while interacting with our system.

3.5 Generative Models

There is a wide variety of generative models to choose from. Recently, diffusion models such as DALL-E 2 and Imagen [12, 14] have shown impressive performance on text-to-image generation [11]. However, recent GAN models such as StyleGAN-T achieve better, more continuous latent space interpolations than diffusion models [15].

The code for StyleGAN-T is not publicly available at the time of writing, so we instead use LAFITE, an older GAN model with fast generating speed and good performance on text-to-image generation [20].

Chapter 4

Implementation

The hyperbolic world is implemented using the Unity game engine and the C# programming language, and is deployed online using Unity WebGL. Latent vector and image generation is handled with Python and Flask servers and deployed using Google Cloud services.

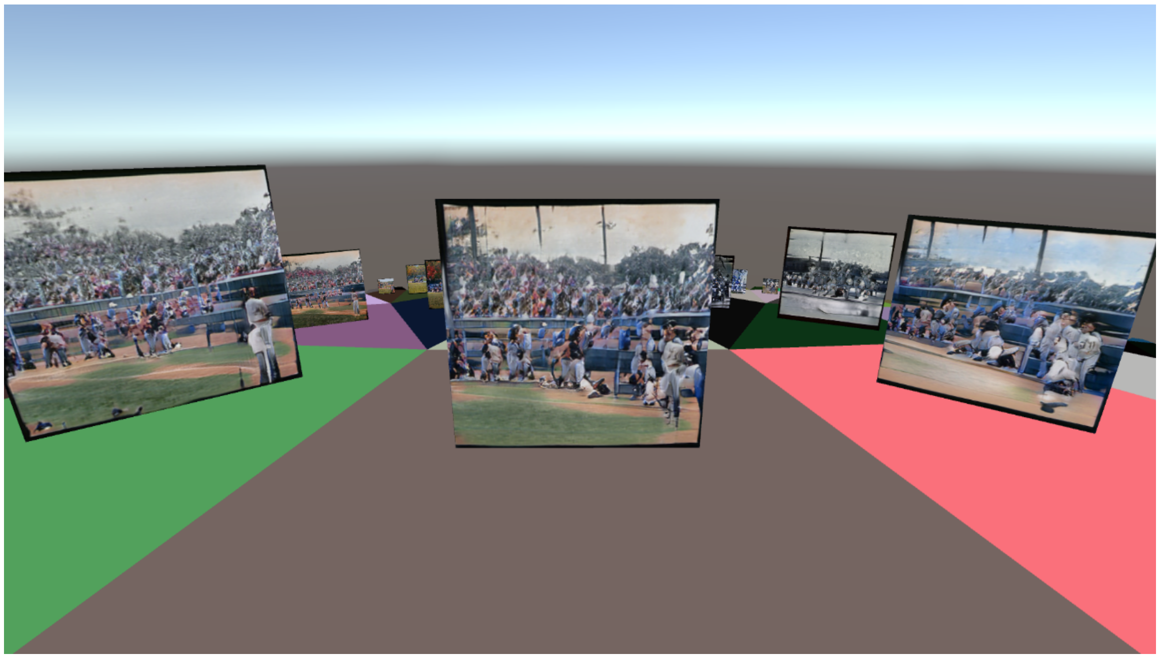


Figure 4.1: A screenshot of images displayed in the hyperbolic world.

4.1 Transformations

We first introduce two 3×3 transformations to be used in the implementation. These transformations use the translations L_x, L_z defined in equations 3.7 and 3.8.

$$T_{xz}(x_0, z_0) = L_x(\sinh^{-1}(x_0))L_z\Bigg(\sinh^{-1}\left(\frac{z_0}{\cosh(\sinh^{-1}(x_0))}\right)\Bigg) \quad (4.1)$$

$$R_{xz}(x_0, z_0) = L_z\Bigg(-\sinh^{-1}\left(\frac{z_0}{\cosh(\sinh^{-1}(x_0))}\right)\Bigg)L_x(-\sinh^{-1}(x_0)) \quad (4.2)$$

The transformation T_{xz} satisfies the property that, given a point $v = (x_0, y_0, z_0)$ on the hyperboloid:

$$T_{xz}(x_0, z_0)\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} x_0 \\ y_0 \\ z_0 \end{bmatrix} = v$$

The transformation R_{xz} performs the reverse transformation of T_{xz} and satisfies the following property:

$$R_{xz}(x_0, z_0)v = R_{xz}(x_0, z_0)\begin{bmatrix} x_0 \\ y_0 \\ z_0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

T_{xz} and R_{xz} provide convenient ways to transform points on the hyperboloid to and from $(0, 1, 0)$. As both of these transformations are essentially just composites of x and z translations, we can apply either of them to every point in the hyperbolic world simultaneously, while preserving all key properties such as distance between points, straight lines, angles, and so on. These transformations are used in Section 4.2.4 and Section 4.3.

4.2 Tiling System

The tiling system consists of three main types of objects: tiles, vertices, and edges. Each tile contains a list of its n vertices and n edges, both in counter-clockwise order. Each vertex contains a list of up to k incident edges in counter-clockwise order. Each edge contains references to the two vertices incident to it and references for up to two tiles that contain the edge.

4.2.1 Vertex and Edge Creation

We first define the vertex and edge creation process. An edge is created using two vertices; the edge holds references to the two vertices, and the edge is added to each of the two vertices' lists of edges.

Upon creation, a vertex may either be initialized or uninitialized. If uninitialized, its position is not defined, and its list of edges may contain fewer than k edges. In order to initialize a vertex, it is given a position, and edges are created to fill its list to size k; each of these new edges contains the given vertex and a new uninitialized vertex. Although these new edges and uninitialized vertices have no defined location, their order in the list still defines their order (i.e. counter-clockwise from each other) in the hyperbolic world.

4.2.2 Tile Creation

On initialization of the system, the center of the first tile is placed at $(0, 1, 0)$. The locations of the vertices of this tile are calculated using the method described in Section 3.3.1 (Tile Sizes) - the location of the first vertex is calculated based on n and k, then the remaining vertex locations are found by rotating the first location around the hyperboloid. The full creation process for the first tile and its corresponding vertices and edges is described in Algorithm 1.

Algorithm 1 Vertex creation for the first tile

```
1: Input $n$                                   ▷ $n$ is the number of number of vertices per tile
2: Input $k$                                   ▷ $k$ is the number of tiles per vertex
3: procedure CreateTile(n, k)
4:     Define t: the new tile
5:     t.center := $(0, 1, 0)$
6:     $x := \sqrt{\cot^2 \frac{\pi}{n} \cot^2 \frac{\pi}{k} - 1}$              ▷ Eq. 3.14
7:     v := new vertex($(x, \sqrt{x^2 + 1}, 0)$)
8:     t.vertices.add(v)
9:     e := t.vertices[0].edges[0]
10:    e.tiles.add(t); t.edges.add(e)
11:    for $1 \leq i < n$ do
12:        v := e.vertex1 if v $\neq$ e.vertex1 else e.vertex2    ▷ Get uninitialized vertex
13:        v.initialize()             ▷ Initialize; create up to $k$ edges and create vertices
14:        v.position = new vertex($R(\frac{2\pi}{n})$ t.vertices[$i - 1$].position)    ▷ Eq. 3.9
15:        t.vertices.add(v)
16:        e := v.prevEdge(e)                     ▷ v's previous edge in CCW order
17:        e.tiles.add(t); t.edges.add(e)
18:    end for
19:    t.vertices[0].edges(1) := e                              ▷ Merge edges
20:    e.uninitializedVertex := t.vertices[0]          ▷ Set e's uninitialized vertex
21:    return t
22: end procedure
```

After creating the first tile, each subsequent new tile is created using a currently-existing reference tile and reference edge; the new tile contains the reference edge and is a neighbor to the reference tile. In order to create this tile, we must first discover all existing edges and vertices, then create new edges and vertices in addition to the pre-existing ones. The full procedure is described in Algorithm 2.

On line 10 of Algorithm 2 we use a function `getVerts` that returns an edge's vertices in counter-clockwise order relative to an input tile. This function simply cycles through the tile's vertices to find the two edge vertices, then returns those two vertices in the correct order. Although inefficient, this provides a location-agnostic method for getting the vertices in the correct order, which can handle situations where the location of the edge is too far away from the origin to be reliably accurate. An alternative method that returns the vertices in CCW order relative to a given location

Algorithm 2 Vertex creation from an existing tile

```
Input ref_tile, ref_edge                                  ▷ Updated reference tile and edge
Input n, k
procedure CreateFromExisting(ref_tile, ref_edge, n, k)
    Define t: the new tile
    ref_edge.tiles.add(t); t.edges.add(ref_edge)
    v := ref_tile.center
    u := mid(ref_edge.vertex1, ref_edge.vertex2)                                  ▷ Eq. 3.12
    w := dir($u, v$) = norm($v - u \cdot \frac{p(u,v)}{q(u)}$)                    ▷ Eq. 3.11
    t.center := $g(v, w, 2 \cdot \text{dist}(v, u))$                              ▷ Eq. 3.10
    vs = ref_edge.getVerts(ref_tile)    ▷ gets vertices in CCW order relative to tile
    v0 := vs[0]
    v1 := vs[1]
    t.vertices.add(v0)
    e1 := v1.nextEdge(ref_edge)                              ▷ $v1$'s next edge in CCW order
    while v0 $\neq$ v0 and e has uninitialized vertex do        ▷ Find existing vertices
        t.vertices.addFront(v1)
        v1 := e1.vertex1 if v1 $\neq$ e1.vertex1 else e1.vertex2
        e1.tiles.add(t); t.edges.add(e1)
        e1 := v1.nextEdge(ref_edge)
    end while
    if v1 != v0 then            ▷ Didn't fully loop around; need to complete vertices
        t.vertices.addFront(v1)
        e2 := v0.prevEdge(e2)
        e2.tiles.add(t); t.edges.add(e2)
        d1 := dir(t.center, vs[1].position)
        d2 := dir(t.center, vs[0].position)
        d3 := dir(d1, d2)       ▷ Direction used to construct new vertex locations
        $x := \sqrt{\cot^2 \frac{\pi}{n} \cot^2 \frac{\pi}{k} - 1}$
        for $2 \leq i < n-$t.vertices.size+2 do
            d4 := d1 $\cdot \cos(i \cdot \frac{2\pi}{n})$ + d3 $\cdot \sin(i \cdot \frac{2\pi}{n})$
            v0 := e2.vertex1 if v0 $\neq$ e2.vertex1 else e2.vertex2
            v0.initialize() if !v0.initialized
            v0.position := $g$(t.center, d4, $\cosh^{-1} \sqrt{x^2 + 1}$)
            t.vertices.add(v0)
            e2 = v0.prevEdge(e2)
            e2.tiles.add(t); t.edges.add(e2)
        end for
        merge(e1, e2)              ▷ Similar to merging process in CreateTile(())
    end if
    return t
end procedure
```

on the hyperboloid is shown in Algorithm 3.

Algorithm 3 Location-based method for returning two vertices in CCW order

```
1: Input vertex1, vertex2                                   ▷ The two vertices
2: Input $l$                      ▷ $l$ is the relative location for the vertices
3: procedure getVerts(vertex1, vertex2, $l$)
4:     v1 := proj(vertex1.position) - proj($l$.position)    ▷ Projection from Sec. 3.1.2
5:     v2 := proj(vertex2.position) - proj($l$.position)
6:     a1 := atan2(v1.z, v1.x)
7:     a2 := atan2(v2.z, v2.x)
8:     a3 := (a2 - a1 + $2\pi$) % ($2\pi$)
9:     if a3 $> \pi$ then
10:        return {vertex2, vertex1}
11:    else
12:        return {vertex1, vertex2}
13:    end if
14: end procedure
```

4.2.3 Setting Vertex Locations

Given a tile with its location and its corresponding vertex positions that are all known to be accurate, Algorithm 4 defines the process for determining and setting the location and the vertex positions of an existing adjacent tile, i.e. a tile that shares an edge with the given one. This is similar to part of Algorithm 2 - we first update the center, then wrap around the to-be-updated vertices and set their locations using a calculated direction vector.

There are two alternate methods for setting vertex locations. Both of these rely on the locations of all of the reference tile's vertices. The first method wraps CCW around the to-be-updated tile and simultaneously wraps CCW around the reference tile; updated vertex locations are calculated by drawing geodesic lines from vertices on the reference tile through the midpoint of the reference edge. The second method wraps CCW around the to-be-updated tile and clockwise around the reference tile; updated vertex locations are essentially calculated by reflecting across the edge. Specifically, given a reference vertex v and u = norm(ref_tile.center - updated_center), the

Algorithm 4 Setting adjacent tile positions

```
1: Input ref_tile, ref_edge                          ▷ Updated reference tile and edge
2: procedure SetVertexLocs(ref_tile, ref_edge)
3:     Define t: the current tile
4:     v := ref_tile.center
5:     u := mid(ref_edge.vertex1, ref_edge.vertex2)
6:     w := $\mathrm{dir}(u, v) = \mathrm{norm}(v - u \cdot \frac{p(u,v)}{q(u)})$
7:     t.center := $g(v, w, 2 \cdot \mathrm{dist}(v, u))$
8:     v0, v1 := ref_edge.getVerts(ref_tile)
9:     e := v0.prevEdge(ref_edge)
10:    d1 := dir(t.center, v1.position)
11:    d2 := dir(t.center, v0.position)
12:    d3 := dir(d1, d2)
13:    $x := \sqrt{\cot^2 \frac{\pi}{n} \cot^2 \frac{\pi}{k} - 1}$
14:    for $2 \leq i < n$ do
15:        d4 := d1 $\cdot \cos(i \cdot \frac{2\pi}{n})$ + d3 $\cdot \sin(i \cdot \frac{2\pi}{n})$
16:        v0 := e.vertex1 if v0 $\neq$ e.vertex1 else e.vertex2
17:        v0.position := $g(\text{t.center}, \text{d4}, \cosh^{-1} \sqrt{x^2 + 1})$
18:        e = v0.prevEdge(e)
19:    end for
20: end procedure
```

corresponding location is calculated as $v - 2p(v, u) \cdot u$.

We do not use either of these two methods in our system. This is because they are empirically less accurate than the one in Algorithm 4, as they rely on all of the reference tile's vertex locations and perform calculations across larger distances.

4.2.4 Updating and Expansion

At any given moment, the user will be standing on a tile, which we will refer to as the *current tile*. As the user moves around (Section 4.3), at every frame the current tile's position must be updated along with its vertex locations. Additionally, during movement, the rotation of the world constantly changes as a byproduct of the space being hyperbolic. As a result, we must keep track of an angle θ, which is used to rotate the vertex locations.

The algorithm for updating the current tile is described in Algorithm 5. We set the tile's position by transforming $(0, 1, 0)$ using an up-to-date position vector. We initialize the first vertex's position using the method described in Section 3.3.1 (Tile Sizes), then rotate it around the hyperboloid to get the positions of all other vertices. Finally, we transform all vertices using the same transform that was applied to the tile center.

Algorithm 5 Updating current tile and vertex locations

```
Input $t$                                   ▷ $t$ is the current tile
Input $p$                                   ▷ $p$ is the updated position of the tile's center
Input $\theta$                              ▷ $\theta$ is the updated angle of the tile
procedure UpdateCurrentTile($t, p, \theta$)
    $t$.center := $T_{xz}(p.x, p.z)\ (0, 1, 0)$
    $v_1, v_2, ..., v_n \leftarrow$ locations of the vertices of $t$
    $x := \sqrt{\cot^2 \frac{\pi}{n} \cot^2 \frac{\pi}{k} - 1}$
    $v_1 := R(\theta)\ (x, \sqrt{x^2+1}, 0)^\top$
    for $2 \leq i \leq n$ do
        $v_i := R(\frac{2\pi}{n})\ v_{i-1}$
    end for
    for $1 \leq i \leq n$ do
        $v_i := T_{xz}(p.x, p.z)\ v_i$
    end for
end procedure
```

After updating the current tile, we expand to neighboring tiles, updating or creating tiles along the way. We perform a breadth-first search (BFS) starting at the current tile and expanding outwards. The BFS stops upon reaching tiles that are further away on the hyperboloid from $(0, 1, 0)$ than a given distance - this distance may be changed by the user to fit performance or visualization needs.

Every tile that the BFS reaches has its location and its vertex locations updated using Algorithm 4. These tiles are then rendered into the hyperbolic world; any tile not reached by the BFS is not updated, and is thus not rendered.

The expansion process is detailed in Algorithm 6.

Algorithm 6 Expanding outwards

```
Input $t$                                         ▷ $t$ is the updated current tile
Input $r$                                         ▷ $r$ is the maximum distance to expand to
procedure EXPAND(t, r)
    queue := {t}
    while !queue.isEmpty() do
        $t_1$ := queue.pop()
        if dist$((0, 1, 0), t_1) > r$ then continue
        end if
        for edge $e \in t_1$.edges do
            if e.numTiles == 2 then               ▷ Case: $t_1$ has a neighbor $t_2$ via $e$
                $t_2$ := e.tile1 if $t_1 \neq$ e.tile1 else e.tile2
            else                                  ▷ Case: need to create a new tile $t_2$
                $t_2$ := CREATEFROMEXISTING$(t_1, e)$
            end if
            if !$t_2$.isUpdated then              ▷ Update and add $t_2$ to queue if needed
                $t_2$.isUpdated = True
                SETVERTEXLOCS$(t_2, e)$
                queue.add$(t_2)$
            end if
        end for
    end while
end procedure
```

4.2.5 Mitigating Inaccuracy

Updating all tile and vertex locations as previously described leads to inaccuracies that quickly compound when expanding positions far away from $(0, 1, 0)$ on the hyperboloid. This causes problems at moderate render distances, even when using double-precision numbers. To solve this issue, every time we calculate a new position for a tile or a vertex, we normalize it to the hyperboloid (Eq 3.5). Doing so allows calculations to remain accurate even at extreme distances on the hyperboloid.

4.3 Movement

When in the hyperbolic world, users may move forward, backward, left, and/or right via keyboard input, and can look around via mouse input. At all times, the system tracks the user's position u and the direction $\phi \in [-\pi, \pi]$ that the user is facing. The system also tracks the tile that the user is standing on. On initialization, the user is positioned at $(0, 1, 0)$ on the hyperboloid.

We define a constant m to be the movement speed. During any frame, if the user is pressing the input to go forward, their position is moved from u to $g(u, d, m)$ (Eq 3.10), where $d = (\cos\phi, 0, \sin\phi)$ is the direction vector that is defined by ϕ and is tangent to the hyperboloid at $(0, 1, 0)$. Likewise, if the user is moving backward, their position is moved from u to $g(u, d, -m)$. If the user is moving right, their new position is $g(u, r, m))$, where $r = (\cos(\phi - \frac{\pi}{2}), 0, \sin(\phi - \frac{\pi}{2}))$ is the direction to the right of where the user is facing. If the user is moving left, their new position is $g(u, r, -m)$.

At every frame, the system checks to see if the user has moved away from $(0, 1, 0)$. If so, we apply the transform $R_{xz}(u_x, u_z)$ to the entire world, which moves the user back to $(0, 1, 0)$ and moves all tiles accordingly. We calculate the new position of the current tile by applying $R_{xz}(u_x, u_z)$ to its current position. We additionally update the tile angle θ as follows:

1. Select the first vertex in the current tile's list of vertices.
2. Apply $R_{xz}(u_x, u_z)$ to the location of this vertex. Let the result be v.
3. Calculate $v = R_{xz}(p_x, p_z)v$, where p is the newly-calculated position of the current tile.
4. Set $\theta = \text{atan2}(v_z, v_x)$.

This allows the updating algorithm to correctly set the current tile's vertex locations. If we do not set the angle during movement then the user's direction will constant changing during movement, even without mouse input.

Finally, if there exists a tile that is closer to the user than the current tile, we set that tile to be the new current tile. We update the new tile's angle as follows:

1. Select the first vertex in the new current tile's list of vertices.
2. Calculate $v = R_{xz}(p_x, p_z)$, where p is the position of the current tile.
3. Set $\theta = \text{atan2}(v_z, v_x)$.

4.3.1 Accounting for Frame Rate

Different systems may have different performance when running the system. Since user position updates are done every frame, users with lower frame rates will move slower through the world than users with higher frame rates under the currently defined constant m movement speed. To account for this, we divide m by the frame rate, which we calculate by keeping track of the time elapsed between frames[1].

[1]There is no built-in way to track frame rate in the Unity game engine.

4.4 Tile Grouping

Let two tiles t_1, t_2 be *adjacent* if they share either an edge or a vertex. We determine if t_1, t_2 are adjacent by measuring the distances between their centers. Using the equation for determining tile size (3.14), let $x = \sqrt{\cot^2 \frac{\pi}{n} \cot^2 \frac{\pi}{k} - 1}$. Then t_1 and t_2 are adjacent if the distance between the two tile centers is less than $2 \cdot \cosh^{-1} \sqrt{x^2+1} + \epsilon$ for some small value[2] of ϵ such as $\epsilon = 10^{-6}$. Intuitively, the value $\cosh^{-1} \sqrt{x^2+1}$ measures the distance from a tile's center to any of its vertices.

Whenever the user enters a tile that doesn't already have an associated latent vector, that tile and all adjacent tiles that also have no associated image are grouped together. Their locations on the hyperboloid are batched into a set of testing points for Gaussian process regression. The positions of all currently-updated tiles that do have an associated latent vector are used as training points, with the latent vectors used as labels. The testing points, training points, and labels are all sent in an HTTP request to a Flask server; newly-generated latent vectors and images are then sent back.

Our system additionally allows users to group together all nearby vector-less tiles with a button press, even if the current tile already has a vector. The current tile is not included in the grouping in this case.

4.5 Model Deployment

As mentioned in Section 3.5, we use the LAFITE model [20] to generate images. This model is housed by a Flask server; the model and server are built into a Docker container, which is then uploaded to Google Vertex AI [7], which exposes an endpoint that can receive HTTP requests.

A Unity client can then send data to this endpoint; the data consists of tile

[2]The purpose of ϵ is to deal with potential floating-point inaccuracies.

coordinates and latent vectors, which are formulated in a JSON object as training points, labels, and testing points as described in Section 4.4. The server receives the data and performs GP regression to generate latent vectors for the testing points, then generates the corresponding images using the LAFITE model. The images are binary-encoded to the Base64 format and are sent along with the latent vectors to the Unity client via HTTP response - the client decodes and renders the images and stores the new latent vectors to be used as future training labels. Figure 4.2 provides a high-level overview of this workflow.

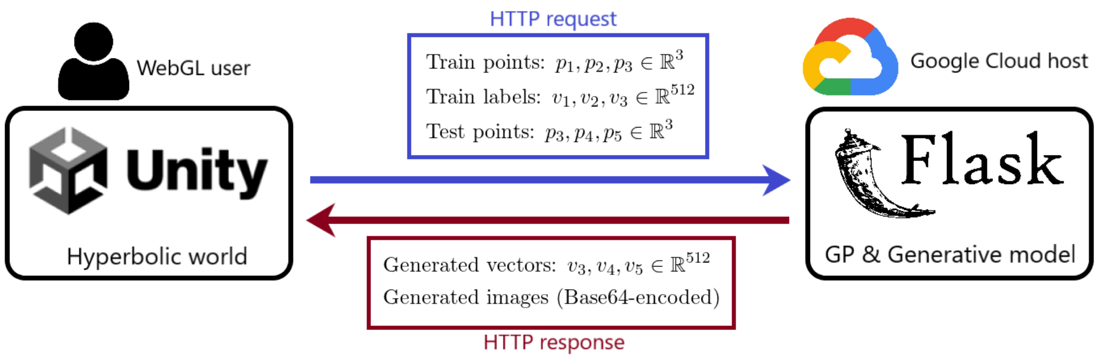

WebGL user
Unity
Hyperbolic world
HTTP request
Train points: $p_1, p_2, p_3 \in \mathbb{R}^3$
Train labels: $v_1, v_2, v_3 \in \mathbb{R}^{512}$
Test points: $p_3, p_4, p_5 \in \mathbb{R}^3$
Google Cloud host
Flask
GP & Generative model
Generated vectors: $v_3, v_4, v_5 \in \mathbb{R}^{512}$
Generated images (Base64-encoded)
HTTP response

Figure 4.2: High-level overview of a typical HTTP workflow.

Note that sending a request to this endpoint requires a Google Cloud service account with an associated private key. In order to prevent Unity users from having direct access to this key, we use an additional Flask server that serves as a midpoint between the Unity client and the Vertex AI model server. This midpoint server forwards data to and from the Unity client and the model server; it uses the private key and the Vertex AI Python API to query the endpoint. This server is deployed to Google App Engine [6].

Chapter 5

Evaluation

A practical use for our project is for searching through a latent space in order to generate various types of high-quality content such as images or 3D models. We evaluate our system by performing multiple simulated experiments where we attempt to find a target latent vector and/or image in the latent space by moving through the hyperbolic world; we compare the similarities between target outputs and ultimately-generated outputs. These experiments simulate human-in-the-loop exploration and optimization of models in the hyperbolic world.

The first set of experiments involves randomly generating a target vector and initializing the world with random latent vectors - we experiment with various different latent space dimensionalities to assess the effectiveness of our system at both low and high dimensions. The second set of experiments utilizes the LAFITE model's text-to-image capabilities to initialize the world with vectors/images that are already close to a the target. Finally, we experiment with increasing n and/or k in the tiling system to determine the effectiveness of the use of hyperbolic space in our system.

5.1 Random Targets and Initialization

In the first experiment, we generate a target image using a randomly-generated 512-dimensional unit vector, then use the hyperbolic world with $(n, k) = (4, 5)$ to attempt to find the closest image to the target. Specifically, for multiple iterations we (1) generate images for all adjacent tiles that do not already have an associated image, then (2) move to the tile with the closest associated latent vector to the target vector in terms of cosine distance (Equation 5.1). When comparing cosine distances, we only use vectors generated in the current iteration. We use cosine distance as a similarity metric rather than, say, Euclidean distance because Euclidean distance and other p-norms tends to lose meaning at higher dimensionalities [10].

$$D_C(u, v) = 1 - \frac{u \cdot v}{\|u\|\|v\|} = 1 - \frac{\sum_{i=1}^{512} u_i v_i}{\sqrt{\sum_{i=1}^{512} u_i^2}\sqrt{\sum_{i=1}^{512} v_i^2}} \tag{5.1}$$

Latent vectors are initialized randomly in the first iteration. The coordinates of the origin tile and all adjacent tiles are used to create a kernel matrix K using the method described in Section 3.4 (Latent Vector Generation). The Cholesky decomposition of this matrix is multiplied by a $t \times d$ matrix with values drawn from $\mathcal{N}(0, 1)$, where t is the number of coordinates and d is the latent space dimension. Each row of the result is normalized to the d-sphere before being used as a latent vector because the LAFITE model requires normalized input vectors.

Across several initial experiments with a static signal variance of $\sigma^2 = 0.002^2$, we find that the cosine distance quickly decreases within the first few iterations, but then plateaus at around $D_C = 0.7$. With a static signal variance of $\sigma^2 = 0.001^2$, the cosine distance decreases at a slower rate and plateaus much later at around $D_C = 0.45$ after many iterations (Figure 5.1).

Therefore, during the experiment, whenever the cosine distance between the target vector and the closest recently-generated latent vector increases for three consecutive

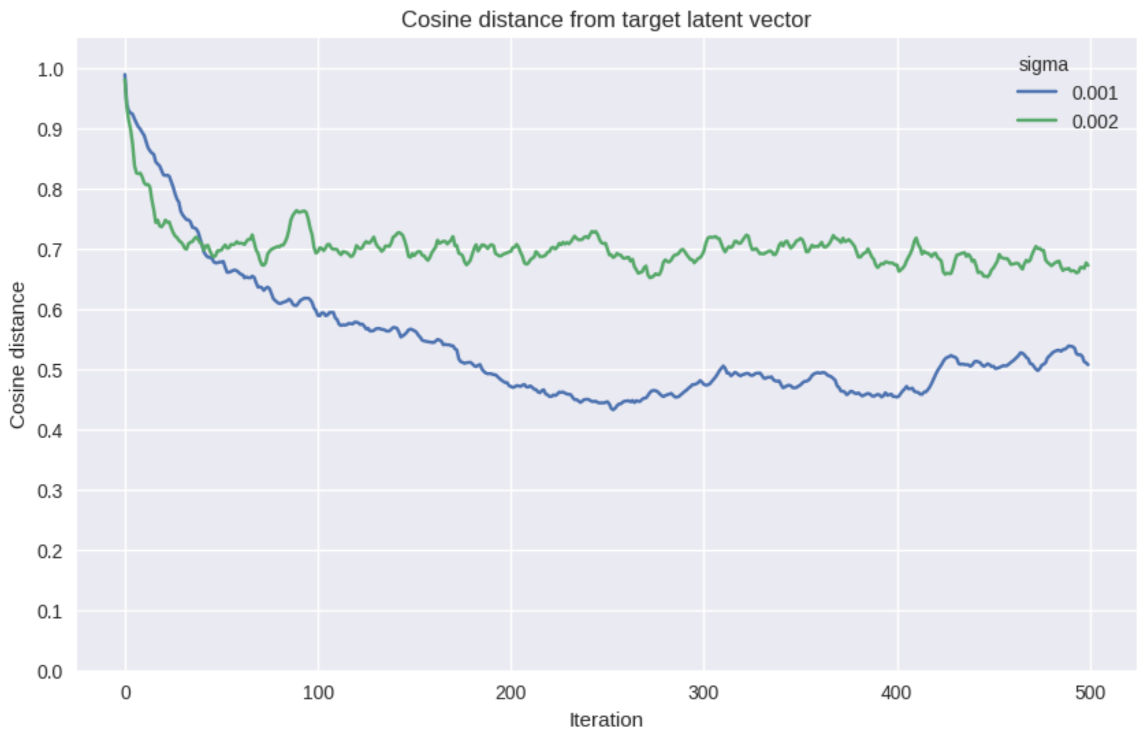
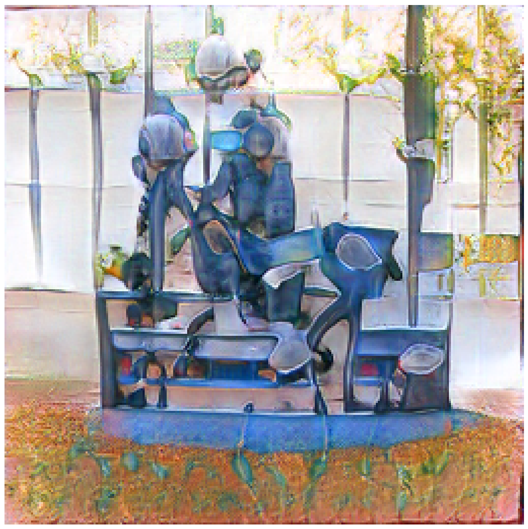


Figure 5.1: Cosine distances between target vector and closest generated vector for different σ^2 values.

iterations, we reduce the signal variance by setting $\sigma = \sigma/\sqrt{2}$. This method simulates the user balancing the exploration-exploitation tradeoff during the optimization process in a similar fashion to Zhou et al [19]. The process begins with a higher σ in order to explore more, then the user may decrease σ as they get closer to their target output. The signal variance for the GP in this experiment is initialized to $\sigma^2 = 0.002^2$, and the lengthscale is set to $\ell = 2.0$; these values work well for the 512-dimensional latent space based on multiple trial runs.

Figure 5.2 shows a sample target image, as well as the final output after 500 generative iterations in the hyperbolic world. These two images are visually similar and share multiple visual features. Figure 5.3 shows every tenth image generated over the first 200 iterations, allowing us to visualise how the intermediate images gradually become more similar to the target image.


(a) Target image

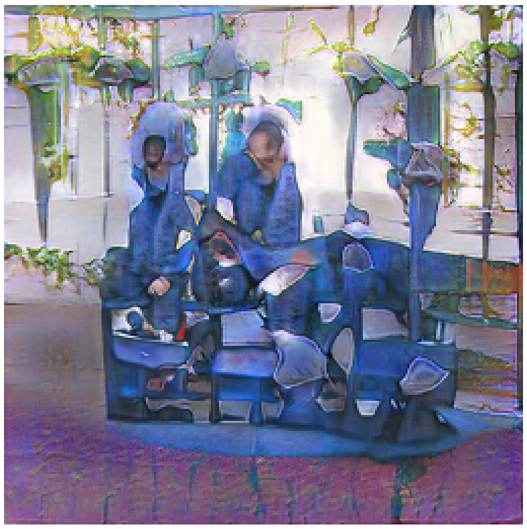

(b) Final generated image

Figure 5.2: Comparison of target image and the closest generated output.

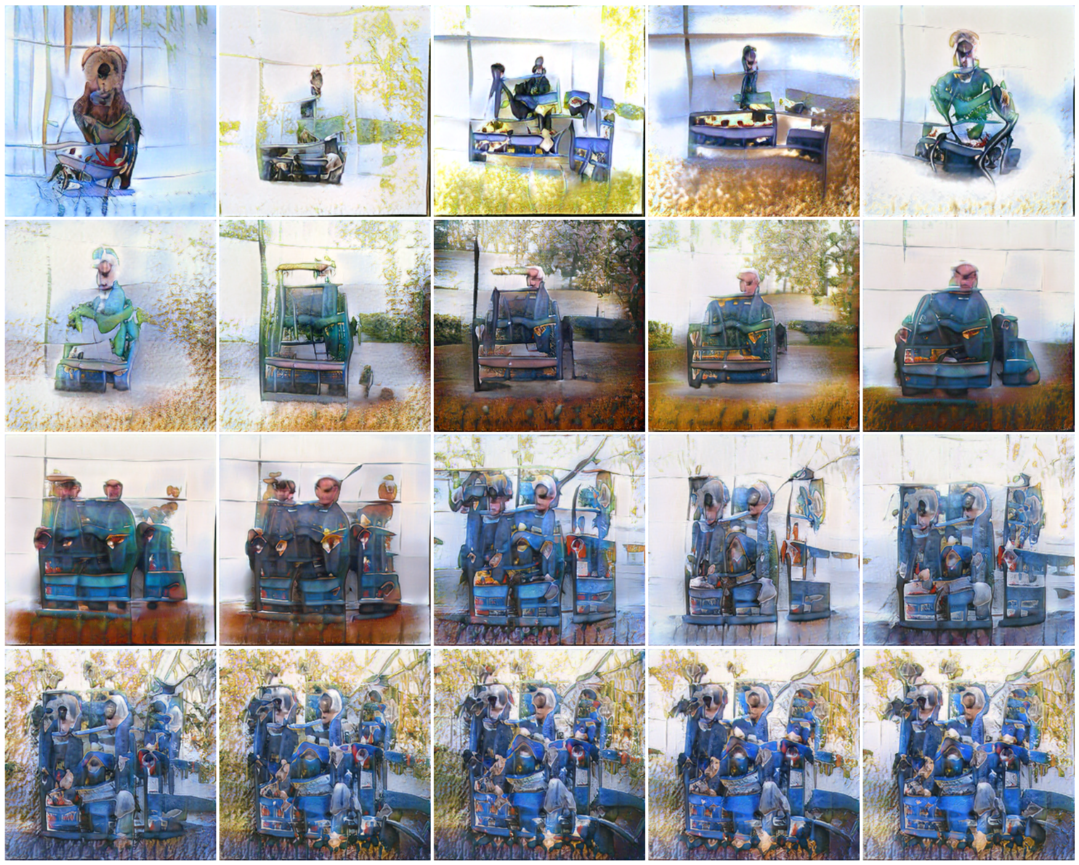


Figure 5.3: Images generated over the first 200 iterations.

In addition to recording the cosine distances for the 512-dimensional latent vectors used by LAFITE, we perform the same experiment and record the distances using latent vectors that have 256, 128, 64, and 32 dimensions. These distances are visualised in Figure 5.4 for each dimension listed. Note that when using 32-dimensional latent vectors, we instead initialize the signal variance to be $\sigma^2 = 0.005^2$, which allows the cosine distance to converge more quickly than when using $\sigma^2 = 0.002^2$. Increasing the initial signal variance did not make a significant difference on larger dimensionalities.

Figure 5.4 illustrates a weakness in our system. Bayesian optimization is known to not scale well to high-dimensional spaces, i.e. spaces with greater than 10 or 20 dimensions, due to the exponentially-increasing amount of space that needs to be covered [17, 3, 18]. While our system quickly converges to the target vector in a 32-dimensional latent space, it requires hundreds of iterations to attain good results at much higher dimensions such as 512.

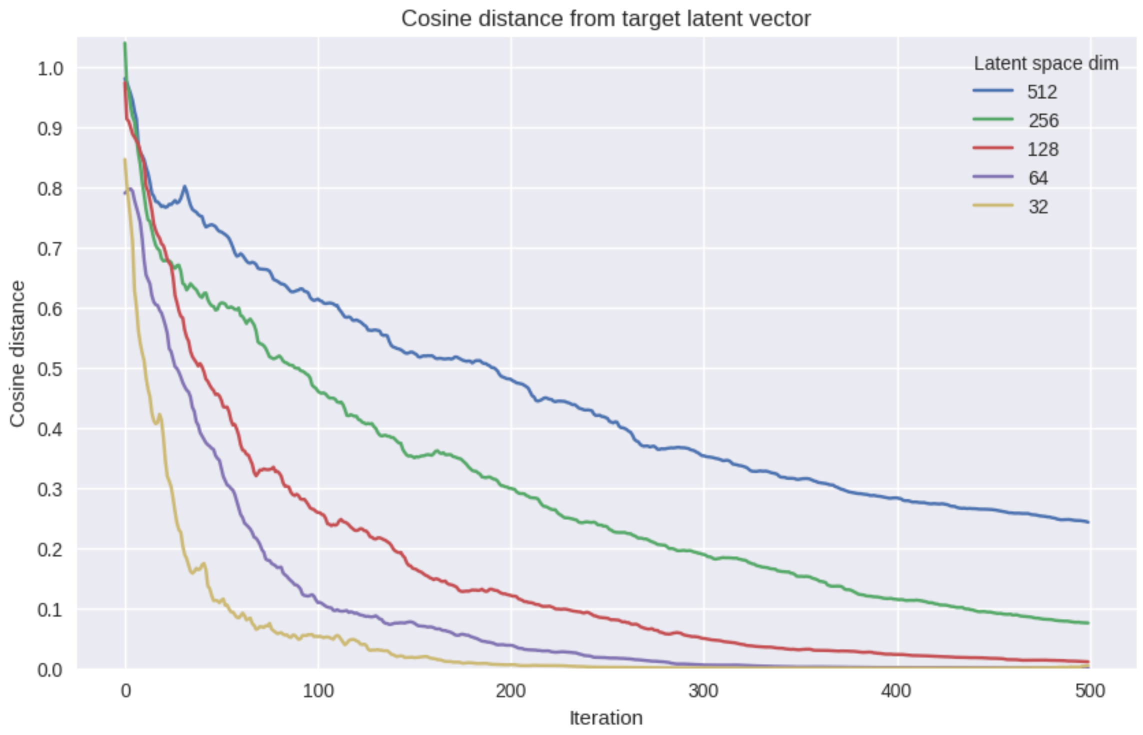


Figure 5.4: Cosine distances between target vector and closest generated vector for different dimensionalities.

5.2 Focused Initialization

The LAFITE model is capable of generating latent vector embeddings for input sentences. This allows users of our system to initialize the first image to be much closer to their target vector than if it were initialized randomly. Therefore, we perform another experiment to analyze the effectiveness of our system in this scenario. We first generate a vector embedding for the sentence "A crowd watching baseball players at a game." To create a target, we add a noise vector n drawn from the distribution $n \sim \mathcal{N}_{512}(0, 0.05)$ to the embedding, then normalize the output to the 512-sphere. This simulates a scenario where a user imagines an image, then starts the exploration process with a fairly similar image by inputting a descriptive sentence. The imagined image uses the vector embedding with added noise, while the starting image uses the vector embedding alone.

We ran this experiment with multiple values of σ; the cosine distances for each σ value are plotted in Figure 5.5. We see that the initial value for σ can make a significant difference at early iterations, although after many iterations the difference is mostly mitigated. Additionally, we are ultimately able to get significantly closer to the target output in terms of cosine distance ($D_C \approx 0.15$) than when using random initialization ($D_C \approx 0.25$).

The initial image generated from the sentence, the target image, and the final generated image from the run with $\sigma = 0.001$ are shown in Figure 5.6. The base image for the sentence "A crowd watching baseball players at a game" is visually somewhat similar to the noisily-generated target image; the target image and the final output are visually very similar and share many visual features, such as the layout of the grass and dirt, as well as the positions and shapes of the people on the dirt and in the background.

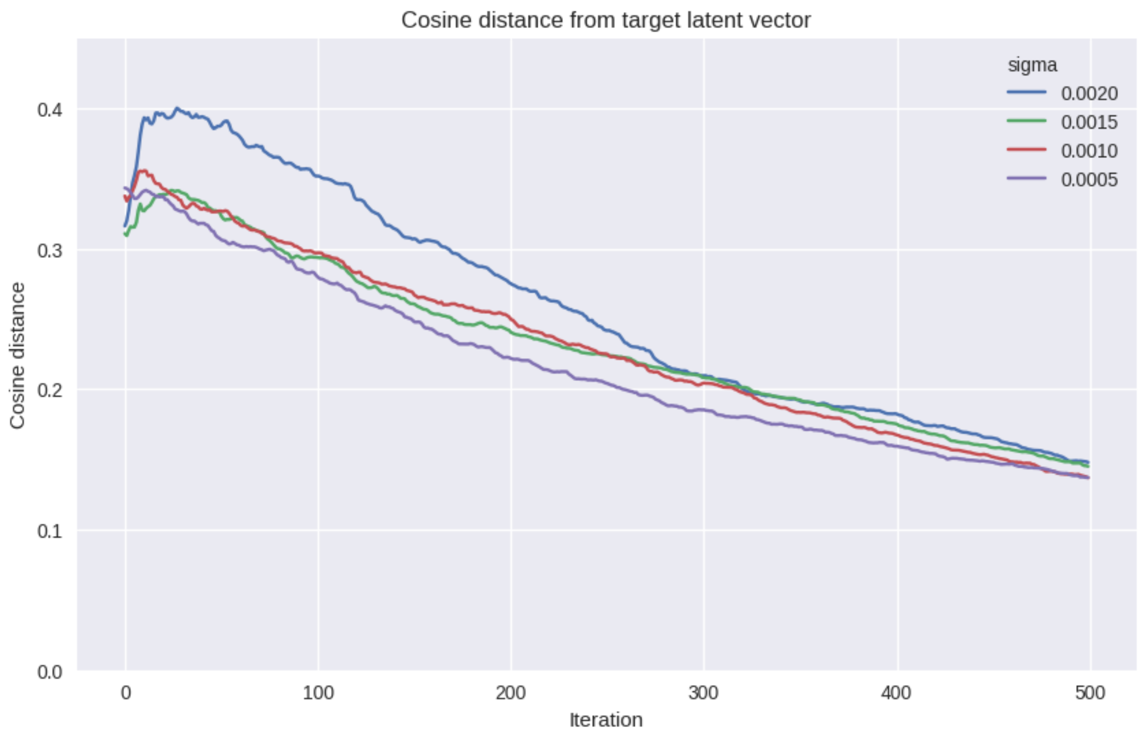
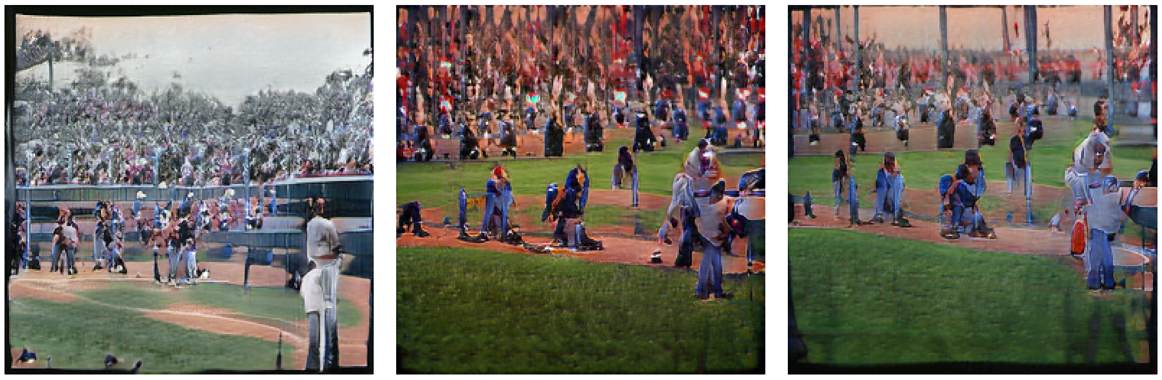


Figure 5.5: Cosine distances when using targeted initialization.



(a) Initial image (b) Target image (c) Final generated image

Figure 5.6: Initial image, target image, and the closest generated output.

5.3 Increasing n and k

To make full use of the hyperbolic world and the tiling system, a user may use different values for n and k. We perform the same simulated experiment as Section 5.1 (Random Targets and Initialization), but rather than using $(n, k) = (4, 5)$, we instead use $(4, 6), (4, 7), (5, 5), (6, 5), (6, 7)$. Each of these pairs represents an increase in n and/or an increase in k from the $(4, 5)$ pair that was used previously. We use an initial signal variance of $\sigma = 0.002$ and increase the lengthscale as n and k increase, based on the formula for tile size in Equation 3.14.

The recorded cosine distances are shown in 5.7. We see that as we increase n and/or k, the cosine distance decreases more quickly at earlier iterations and eventually converges to lower values after many iterations. This makes intuitive sense, as with an increased (n, k) we can generate more samples at a time and explore more of the hyperbolic world at each iteration.

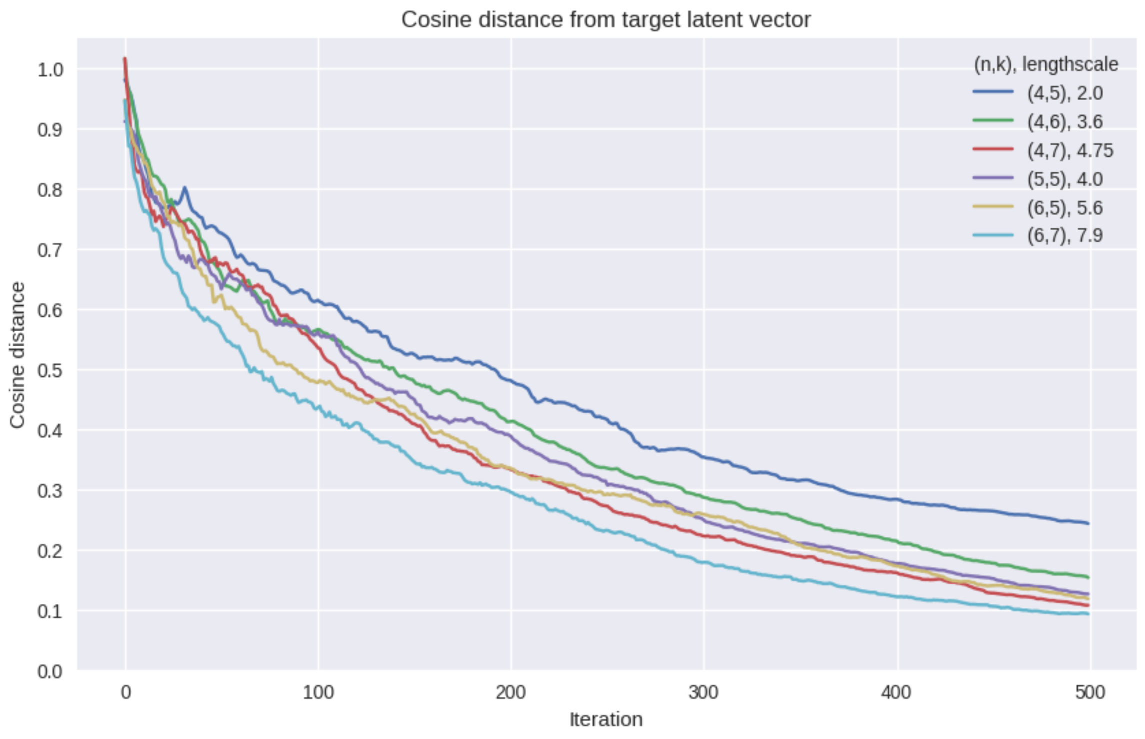


Figure 5.7: Cosine distances for various (n, k), and ℓ values.

Chapter 6

Conclusions and Future Work

6.1 Summary

This work introduces an application for exploring generative models in hyperbolic space. We define a set of geometric functions and a tiling system for the hyperboloid model of hyperbolic geometry. We implement the hyperbolic world in the Unity game engine and link the hyperbolic world with a Flask server that uses GP regression to produce output images from the LAFITE generative model. Finally, through multiple simulated experiments, we find that our system can reliably, albeit slowly, find target outputs through exploration of the hyperbolic world.

There are currently few resources available that detail the steps necessary to simulating hyperbolic space. This report and the associated code serves as a simple, working example of a simulation of hyperbolic space, and may be used as a starting point for those interested in modeling hyperbolic space and exploring ways to optimize generative models within hyperbolic space.

6.2 Limitations and Future Work

The biggest limitation of our system is its poor scalability to high dimensions, although increasing the number of vertices per tile and/or increasing the number of tiles per vertex can help mitigate this issue. A potential path of future work is to utilize a method to reduce the dimensionality of a high-dimensional latent space in order to allow users to converge more quickly to target outputs; for instance, the work done by Zhou et al. involves training a VAE to reduce the latent space of a music VAE from 512 to 4 dimensions [19].

Additionally, we only performed simulated experiments, which do not necessarily translate to practice. In a simulated experiment our system always chooses the optimal action towards a stationary target; real users' judgment of output quality may be unclear or even change during exploration.

In the Unity implementation of the hyperbolic world, vertices are projected to the correct locations on the Poincaré disk; however, edges remain straight in this projection when they should be curved. This is a result of only specifying projected vertex locations; future work could add curvature to the edges.

As part of this project, we have implemented several image generation models, with LAFITE being the currently-deployed model. Future work could add more models that generate various high-quality content beyond images, such as 3D models or melodies, and additionally deploy more than one model at a time for users to choose from.

Appendix A

Proofs

A.1 Derivation for Valid (n, k) Hyperbolic Tilings

Given k regular n-gons meeting at one point, the sum of the interior angles of each n-gon vertex meeting at that point is $k \cdot \frac{n-2}{n}\pi$. If this quantity is equal to 2π then (n, k) defines a tiling in Euclidean space; if it is less than 2π then it is a spherical tiling; if it is greater than 2π then it is a hyperbolic tiling.

$$\begin{aligned}
k \cdot \frac{n-2}{n}\pi &> 2\pi \\
\frac{n-2}{n} &> \frac{2}{k} \\
1 - \frac{2}{n} &> \frac{2}{k} \\
1 &> \frac{2}{n} + \frac{2}{k} \\
nk &> 2(n+k) \\
nk - 2n - 2k &> 0 \\
nk - 2n - 2k + 4 &> 4 \\
(n-2)(k-2) &> 4 \qquad \text{(A.1)}
\end{aligned}$$

A.2 Derivation for Tile Size Given (n, k)

Let $\theta = \frac{2\pi}{n}$. By rotating this vertex counterclockwise around the hyperboloid by θ and $-\theta$, we obtain $v_1 = (x\cos\theta, \sqrt{x^2+1}, x\sin\theta)$ and $v_2 = (x\cos\theta, \sqrt{x^2+1}, -x\sin\theta)$, the two vertices adjacent to v_0. As described in Section 3.2, the angle between the line containing v_0, v_1 and the line containing v_0, v_2 at the intersection point v_0 is given by $\cos^{-1}(w_1, w_2)$, where w_1, w_2 are the tangent vectors to the two lines at v_0. Additionally, since there are k n-gons meeting at v_0, this angle must be equal to $\phi = \frac{2\pi}{k}$.

We first calculate w_1 and w_2.

$$\begin{aligned}
w_1 &= \operatorname{norm}\left(v_1 - v_0 \cdot \frac{p(v_0, v_1)}{q(v_1)}\right) \\
&= \operatorname{norm}((x + x^3)(\cos\theta - 1), \sqrt{x^2+1}(x^2(\cos\theta - 1)), x\sin\theta) \\
&= \frac{((x + x^3)(\cos\theta - 1), \sqrt{x^2+1}(x^2(\cos\theta - 1)), x\sin\theta)}{\sqrt{q\left(((x + x^3)(\cos\theta - 1), \sqrt{x^2+1}(x^2(\cos\theta - 1)), x\sin\theta)\right)}} \\
&= \frac{((x + x^3)(\cos\theta - 1), \sqrt{x^2+1}(x^2(\cos\theta - 1)), x\sin\theta)}{\sqrt{-(x^2+1)x^4(\cos\theta - 1)^2 + (x + x^3)^2(\cos\theta - 1)^2 + x^2\sin^2\theta}} \\
&= \frac{((x + x^3)(\cos\theta - 1), \sqrt{x^2+1}(x^2(\cos\theta - 1)), x\sin\theta)}{\sqrt{(x^4 + x^2)(\cos\theta - 1)^2 + x^2\sin^2\theta}} \\
&= \frac{((1 + x^2)(\cos\theta - 1), \sqrt{x^2+1}(x(\cos\theta - 1)), \sin\theta)}{\sqrt{(x^2+1)(\cos\theta - 1)^2 + \sin^2\theta}}
\end{aligned}$$

Applying the same process for w_2, we get

$$w_2 = \frac{((1 + x^2)(\cos\theta - 1), \sqrt{x^2+1}(x(\cos\theta - 1)), -\sin\theta)}{\sqrt{(x^2+1)(\cos\theta - 1)^2 + \sin^2\theta}}$$

Calculating the angle between w_1 and w_2:

$$\begin{aligned}
\cos\phi &= p(w_1, w_2) \\
&= \frac{-(x^2+1)x^2(\cos\theta-1)^2 + (x^2+1)^2(\cos\theta-1)^2 - \sin^2\theta}{(x^2+1)(\cos\theta-1)^2+\sin^2\theta} \\
&= \frac{(x^2+1)(\cos\theta-1)^2 - \sin^2\theta}{(x^2+1)(\cos\theta-1)^2+\sin^2\theta}
\end{aligned}$$

Finally, solving for x:

$$\begin{aligned}
\cos\phi((x^2+1)(\cos\theta-1)^2+\sin^2\theta) &= (x^2+1)(\cos\theta-1)^2-\sin^2\theta \\
x^2\left((\cos\phi-1)(\cos\theta-1)^2\right) &= -(\cos\phi-1)(\cos\theta-1)^2 - \cos\phi\sin^2\theta - \sin^2\theta \\
x^2 &= \frac{-(\cos\phi-1)(\cos\theta-1)^2 - \cos\phi\sin^2\theta - \sin^2\theta}{(\cos\phi-1)(\cos\theta-1)^2} \\
&= -\frac{-\cos\phi\sin^2\theta-\sin^2\theta}{(\cos\phi-1)(\cos\theta-1)^2} - 1 \\
&= \frac{\sin^2\theta(\cos\phi+1)}{(1-\cos\phi)(\cos\theta-1)^2} - 1 \\
&= \frac{\sin^2\theta}{(\cos\theta-1)^2}\frac{1+\cos\phi}{1-\cos\phi} - 1 \\
&= \cot^2\frac{\theta}{2}\cot^2\frac{\phi}{2} - 1 \\
x &= \sqrt{\cot^2\frac{\pi}{n}\cot^2\frac{\pi}{k} - 1} \qquad \text{(A.2)}
\end{aligned}$$

Bibliography

[1] G. Arvanitidis, L. K. Hansen, and S. Hauberg. Latent space oddity: on the curvature of deep generative models, 2017.

[2] E. Brochu, T. Brochu, and N. d. Freitas. A Bayesian Interactive Optimization Approach to Procedural Animation Design. In M. Popovic and M. Otaduy, editors, *Eurographics/ ACM SIGGRAPH Symposium on Computer Animation*. The Eurographics Association, 2010.

[3] C.-H. Chiu, Y. Koyama, Y.-C. Lai, T. Igarashi, and Y. Yue. Human-in-the-loop differential subspace search in high-dimensional latent space. *ACM Trans. Graph.*, 39(4), aug 2020.

[4] T. Chong, I.-C. Shen, I. Sato, and T. Igarashi. Interactive optimization of generative image modelling using sequential subspace search and content-based guidance. *Computer Graphics Forum*, 40(1):279–292, 2021.

[5] Non-Euclidean Geometry Explained - Hyperbolica Devlog #1. `https://youtu.be/zQo_S3yNa2w`. Accessed: 2023-02-22.

[6] App Engine Application Platform — Google Cloud. `https://cloud.google.com/appengine`. Accessed: 2023-04-11.

[7] Vertex AI — Google Cloud. `https://cloud.google.com/vertex-ai`. Accessed: 2023-04-11.

[8] E. Kopczyński, D. Celińska, and M. Čtrnáct. Hyperrogue: Playing with hyperbolic geometry. In D. Swart, C. H. Séquin, and K. Fenyvesi, editors, *Proceedings of Bridges 2017: Mathematics, Art, Music, Architecture, Education, Culture*, pages 9–16, Phoenix, Arizona, 2017. Tessellations Publishing.

[9] Y. Liu, E. Jun, Q. Li, and J. Heer. Latent space cartography: Visual analysis of vector space embeddings. *Computer Graphics Forum*, 38(3):67–78, 2019.

[10] E. M. Mirkes, J. Allohibi, and A. Gorban. Fractional norms and quasinorms do not help to overcome the curse of dimensionality. *Entropy*, 22(10):1105, sep 2020.

[11] Text-to-Image Generation on COCO. `https://paperswithcode.com/sota/text-to-image-generation-on-coco`. Accessed: 2023-04-02.

[12] A. Ramesh, P. Dhariwal, A. Nichol, C. Chu, and M. Chen. Hierarchical text-conditional image generation with clip latents, 2022.

[13] W. F. Reynolds. Hyperbolic geometry on a hyperboloid. *The American Mathematical Monthly*, 100(5):442–455, 1993.

[14] C. Saharia, W. Chan, S. Saxena, L. Li, J. Whang, E. Denton, S. K. S. Ghasemipour, B. K. Ayan, S. S. Mahdavi, R. G. Lopes, T. Salimans, J. Ho, D. J. Fleet, and M. Norouzi. Photorealistic text-to-image diffusion models with deep language understanding, 2022.

[15] A. Sauer, T. Karras, S. Laine, A. Geiger, and T. Aila. Stylegan-t: Unlocking the power of gans for fast large-scale text-to-image synthesis, 2023.

[16] M. Von Gagern. *Creation of Hyperbolic Ornaments – Algorithmic and Interactive Methods.* PhD thesis, Fakultät für Mathematik, 07 2014.

[17] Z. Wang, F. Hutter, M. Zoghi, D. Matheson, and N. de Freitas. Bayesian optimization in a billion dimensions via random embeddings, 2016.

[18] Y. Zhou, Y. Koyama, M. Goto, and T. Igarashi. Generative melody composition with human-in-the-loop bayesian optimization, 2020.

[19] Y. Zhou, Y. Koyama, M. Goto, and T. Igarashi. Interactive exploration-exploitation balancing for generative melody composition. In *26th International Conference on Intelligent User Interfaces*, IUI '21, page 43–47, New York, NY, USA, 2021. Association for Computing Machinery.

[20] Y. Zhou, R. Zhang, C. Chen, C. Li, C. Tensmeyer, T. Yu, J. Gu, J. Xu, and T. Sun. LAFITE: towards language-free training for text-to-image generation. *CoRR*, abs/2111.13792, 2021.