A locality-sensitive hash for real vectors

Tyler Neylon

Abstract

We present a simple and practical algorithm for the $c-$approximate near neighbor problem ($c-$NN): given n points $P \subset \mathbb{R}^d$ and radius R, build a data structure which, given $q \in \mathbb{R}^d$, can with probability $1-\delta$ return a point $p \in P$ with $\mathsf{dist}(p,q) \leq cR$ if there is any $p^* \in P$ with $\mathsf{dist}(p^*,q) \leq R$. For $c = d+1$, our algorithm deterministically ($\delta = 0$) preprocesses in time $O(nd \log d)$, space $O(dn)$, and answers queries in expected time $O(d^2)$; this is the first known algorithm to deterministically guarantee an $O(d)-$NN solution in constant time with respect to n for all ℓ_p metrics. A probabilistic version empirically achieves useful c values ($c < 2$) where c appears to grow minimally as $d \to \infty$. A query time of $O(d \log d)$ is available, providing slightly less accuracy. These techniques can also be used to approximately find (pointers between) all pairs $x, y \in P$ with $\mathsf{dist}(x,y) \leq R$ in time $O(nd \log d)$.

The key to the algorithm is a locality-sensitive hash: a mapping $h : \mathbb{R}^d \to U$ with the property that $h(x) = h(y)$ is much more likely for nearby x, y. We introduce a somewhat regular simplex which tessellates $\mathbb{R}^d$, and efficiently hash each point in any simplex of this tessellation to all $d+1$ corners; any points in neighboring cells will be hashed to a shared corner and noticed as nearby points. This method is completely independent of dimension reduction, so that additional space and time savings are available by first reducing all input vectors.

1 Introduction.

In this paper, we focus on a practical variation of the traditional *nearest neighbor search problem* (NNS): given points $P \subset \mathbb{R}^d$ and query point $q \in \mathbb{R}^d$, find $p \in P$ which minimizes $\mathsf{dist}(p,q)$. The usual approach is to preprocess P so that a later query can quickly retrieve the closest point.

Although much work has been done to efficiently solve NNS, the known non-probabilistic approaches thus far are not much more efficient than a brute force linear search over P, except in small dimensions d (see [15] for approaches in small d). Therefore, practical algorithms have been developed which can efficiently solve probabilistic approximate versions of NNS. These less exact variations still support many applications, including data mining, information retrieval, media databases (such as images or video), pattern recognition, and duplicate detection. For example, a video database may want to avoid the insertion of videos which are slightly altered versions of existing data, and may do so by solving an approximate version, such as $c-$NN (defined below), on real vector representations of the videos. In this case, we expect a near-duplicate video q to be many times closer to a particular point in P than any other, so a fast approximate algorithm is very useful.

The authors of [11] show how to reduce a probabilistic approximate version of NNS to a version in which the allowed distances between p and q are restricted by a fixed parameter R. In particular, let us define the *$c-$near neighbor problem* ($c-$NN) as follows:

DEFINITION 1.1. *Given points $P \subset \mathbb{R}^d$, radius $R > 0$, and probability tolerance $\delta > 0$, preprocess P so that for any $q \in \mathbb{R}^d$, we can, with probability at least $1-\delta$, find $p \in P$ with $\mathsf{dist}(p,q) < cR$ whenever there is a $p' \in P$ with $\mathsf{dist}(p',q) < R$.*

The value $c \geq 1$ is the *approximation factor*; ideally we can solve this for $c = 1+\epsilon$, where both error terms ϵ and δ are small. The exact value of R is immaterial since input points can easily be scaled to match the fixed radius of any particular $c-$NN implementation.

The work in [11] also introduced the idea of a *locality-sensitive hash* — a hash function h with the property that $h(x) = h(y)$ is more likely for nearby points x, y. A good locality-sensitive hash can serve as a basis for solving $c-$NN, and thus for approximate approaches to NNS as well. Since [11], a number of such hashes have been proposed for use on various point sets (including strings [13] and families of subsets [3]), and in various metrics (cf. [7], [2], [8]).

In this paper, we focus on solving $c-$NN in $\mathbb{R}^d$ under the ℓ_p norm for $p \in [1, \infty]$ (equation (2.6)), with emphasis on ℓ_2 (equation (2.12)). Previous work has centered around reducing the time complexity dependence on $n = |P|$. However, locality-sensitive hashes more directly solve $c-$NN than variants of NNS, and lend themselves to approaches more sensitive to dimension d, accuracy c, and certainty $1-\delta$, than to speed in terms of n. We propose that any advantages previously conveyed by smaller theoretical exponents on n can be expressed in the form of better speed, probability of success, and approximation factors within this framework.

1.1 Anatomy of a locality-sensitive hash

A typical locality-sensitive hashing scheme on $\mathbb{R}^d$ consists of three components: *dimension reduction*, the *local-*

ity hash, and *amplification*. It is widely utilized that a random projection of n points in $\mathbb{R}^d$ to a smaller space $\mathbb{R}^t$, $t = O(\log n)$, is likely to preserve interpoint distances ([7], [12]). Thus we can apply a locality hash to a reduced vector with good results.

In addition, most locality hashes improve with some form of amplification – a repetition of some portion of the hashing scheme which improves the accuracy. Examples include using multiple hash tables with slightly different locality hash functions, or performing multiple queries via random points close to the given $q \in \mathbb{R}^d$ ([14]).

The locality hash we present in §2 is independent of either dimension reduction or amplification, so that both of these techniques can be used to augment the base performance. This paper focuses on the locality hash itself; we do not directly address the question of how to optimally combine the hash with these techniques.

1.2 Our contributions We present two versions of a simple and general algorithm based on a tessellation of $\mathbb{R}^d$ by simplices (a simplex is the $d-$dimensional generalization of a triangle — the convex hull of $d+1$ points in general position). In each case, any given point is hashed to all $d+1$ corners of the cell (one simplex) in the tessellation which contains the point. In this way, we can be sure to match any two points in neighboring cells of the tessellation.

By using simplices instead of hypercubes, we can work with $O(d)$ corners instead of $O(2^d)$; thus we confront the curse of dimensionality. We can achieve storage in time $O(d \log d)$ per point, and lookup in expected time $O(d^2)$, or $O(d \log d)$ at some accuracy cost as explained in §2.4.

Our main theoretical result (in §2) is to show that one version of our hash deterministically ($\delta = 0$) solves $c-\mathsf{NN}$ with $c = 2d$ in the ℓ_p norm for any $p \in [1, \infty]$, and that the other version gives an approximate factor $c \leq d+1$ for ℓ_2. Previous algorithms ([4]) can solve $c-\mathsf{NN}$ in query time independent of n, with $c = O(d)$ in ℓ_1, and in select other ℓ_p metrics via embedding ([5], [9], [10]). To our knowledge, this is the first proven to achieve $c = O(d)$ for all ℓ_p, $p \in [1, \infty]$.

We also give empirical evidence (in §3) that the algorithm is practical as a probabilistic approach to $c-\mathsf{NN}$ for smaller c on a wide range of dimensions. For example, using artificial data, we approximate the quantity $\beta_{.1}$, an estimated upper bound on c for $\delta = 0.1$ (90% confidence). In table 1, we see that this algorithm achieves $\beta_{.1} = 1.6$ in $\mathbb{R}^{300}$ with at least 90% confidence, without dimension reduction.

	$d =$	10	100	300	10	20
hash	$L_t =$	5	5	5	$d+1$	$d+1$
here		1.6	1.7	1.6	1.5	1.5
pstable [7]		5.6	4.7	5.1	3.9	2.8
sphere grid [2]		6.0	5.1	5.1	4.1	3.1
unary [8]		8.3	6.0	5.6	4.4	3.3

Table 1: Empirical estimates of $\beta_{.1} = D_{.05}/D_{.95}$, further explained in §3.

2 The algorithm

In this section, we describe a general class of hash algorithms which provide guarantees of finding close enough neighbors while excluding those sufficiently far apart. Next we describe a particular version of this algorithm, which we refer to as *the orthogonal version*, which is easy to implement and analyze. Finally we present an improved version, *the vertex-transitive version*, along with some thoughts towards improved time and space complexity.

2.1 The general method In this section, we suppose that we have a *partition* $\mathcal{P}$ which is a family of subsets of $\mathbb{R}^n$ with the following properties: (*i*) Each element $c \in \mathcal{P}$, called a *cell*, is the convex hull of finitely-many points; (*ii*) $\mathbb{R}^n = \cup\mathcal{P}$, while the intersection of any two cells has zero measure; and (*iii*) any bounded region contains finitely many cells. The general framework of our algorithm, described next, works for any such partition.

The partition algorithm Based on such a partition, we are now ready to define a general locality-sensitive hashing algorithm. The idea is to build a map from each corner of the partition to a list of all data points in adjacent cells. Conceptually, start with a fast-lookup mapping (such as a hash map) from every corner of every cell to an empty linked list (sparse, created lazily). For point $x \in \mathbb{R}^d$, choose a cell $c \in \mathcal{P}$ containing x, and append (a pointer to) x to the list mapped from each corner of c. This process can be repeated for an arbitrary number of points, at each step augmenting the lists mapped to by each corner. We say that two points x, y collide in this hash if any corner point maps to both x and y; write this as $x \sim y$. This general algorithm is also outlined in the pseudocode in figure 1.

In practice, the result of a query will be a set of lists of points; it may suffice to simply work with the first point found, avoiding the time required to parse all the points.

An advantage of this algorithm is that it deterministically guarantees that all close enough points are considered nearby, while all far enough are not. The chal-

```
def locality_hashes(x):
  cell c = cell_containing(x)
  return corners(c)

def preprocess_points(P):
  map = {} # An empty mapping.
  for p in P:
    for h in locality_hashes(p):
      map[h].append(p)
  return map

def lookup_query(map, q):
  neighbors = [] # An empty list.
  for h in locality_hashes(q):
    neighbors.append(map[h])
  return neighbors
```

Figure 1: The general partition-based LSH algorithm

lenge from here is to preserve speed by choosing a good partition. A naive choice of $\mathcal{P}$, such as all unit hypercubes cornered at integer points, can result in many corners (2^d) per cell, which in turn requires running time exponential in d. Accordingly, we will now consider how our choice of $\mathcal{P}$ affects the performance of this algorithm.

Time and space complexity. The resources used by this algorithm are dominated by the calls to `locality_hash`. Suppose one call to `locality_hash` takes time τ and returns an object of size $\sigma = O(\tau)$. Our preprocessing time of n points then takes $O(n\tau)$ time, and $O(n\sigma)$ space. If we do not bother to traverse the neighbors list and simply return pointers to these lists, then a single lookup also takes time $O(\tau)$.

When we fill in the details below, (§2.4) we'll see that we can achieve $\tau = d \log d$ and $\sigma = d$, where d is the dimensionality of our points. Unfortunately, verifying the identity of each hash key requires $O(d)$ time, so that queries (but not storage) are actually $O(d^2)$. If we're willing to probabilistically verify hash keys at the cost of some false collisions, we can still achieve $O(d \log d)$ lookups.

Accuracy. Next we'd like to give some useful characterizations concerning which pairs of points are, or are not, considered nearby neighbors of this algorithm. Our main goal will be to discover, for a given partition $\mathcal{P}$, the optimal values D_0 and D_1 such that

$$\begin{aligned} \mathsf{dist}(x,y) > D_0 \;&\Rightarrow\; x \not\sim y \quad \text{and} \\ \mathsf{dist}(x,y) < D_1 \;&\Rightarrow\; x \sim y. \end{aligned}$$

This algorithm solves $c{-}\mathsf{NN}$ where $c = D_0/D_1$ and $\delta = 0$. The notation here is inspired by the idea, expanded below, that D_p, for $p \in (0,1)$, can represent the distance at which

$$\mathsf{dist}(x,y) = D_p \;\Rightarrow\; P(x \sim y) = p$$

in a certain probabilistic context.

We'll work with a particular type of partition which is more conducive to analysis. A *sliced partition* is a division of $\mathbb{R}^d$ into cells created by cutting the space along a series of parallel hyperplanes so that every cell is bounded by exactly two hyperplanes in each direction. We further require that the hyperplanes be oriented in finitely-many directions, and that every bounded subset of $\mathbb{R}^d$ intersects finitely-many of the hyperplanes. It is not hard to verify that a sliced partition is a special case of our general partition described above.

As a quick example, the hypercube partition — in which we cut $\mathbb{R}^d = \{x = (x_1, x_2, \ldots, x_d)\}$ along every hyperplane $x_i = j;\ i \in [d], j \in \mathbb{Z}$ — is a canonical example of a sliced partition (notation: $[d] := \{1, 2, \ldots, d\}$).

We can effectively define $D_1 = D_1(\mathcal{P})$ as the quantity

$$D_1 := \inf\{\mathsf{dist}(x,y) : x \not\sim y; x, y \in \mathbb{R}^d\}. \tag{2.1}$$

Let $c_x \sim c_y$ indicate that cells c_x and c_y share a boundary point. Toward characterizing D_1 in terms of the cells of $\mathcal{P}$, we'll say that points x, y are *barely neighbors* when $x \sim y$ yet there are cells c_x, c_y with $x \in c_x$, $y \in c_y$, and $c_x \not\sim c_y$. This may occur when x, y are in the boundary of neighboring cells, or the same cell, and this boundary is shared with two cells which are not neighbors of each other.

LEMMA 2.1. *Given any points $x \not\sim y$, there are barely neighboring points $x' \sim y'$ between x and y along the line segment $\overline{xy}$.*

We defer this proof and others in this section to the appendix. This last lemma clarifies that

$$D_1 = \inf\{\mathsf{dist}(x,y) : x, y \text{ are barely neighbors}\}, \tag{2.2}$$

so that, in finding D_1, we may focus exclusively on such point pairs. Combining this fact with the next lemma will allow us to gain a powerful tool for extracting D_1 from the shape of the cells in any sliced partition.

Let $\mathsf{corners}(c)$ denote the minimal set of points whose convex hull is the cell c. We will say that points x, y *cross* a cell c iff $x, y \in c$ and there are disjoint subsets $C_x, C_y \subset \mathsf{corners}(c)$ with $x \in \mathsf{convex}(C_x)$ and $y \in \mathsf{convex}(C_y)$. Intuitively, we can think of this notion as stipulating that any edge path from x to y must contain at least one full edge.

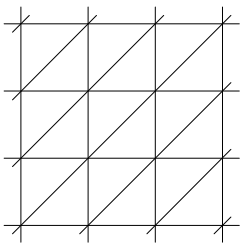

Figure 2: The orthogonal partition $\mathcal{O}$ in $\mathbb{R}^2$

LEMMA 2.2. *Suppose x, y are points in cell c in sliced partition $\mathcal{P}$. Then x, y are barely neighbors $\Leftrightarrow$ x, y cross c.*

Lemma 2.1 allowed us to restrict the definition (2.1) of D_1 to the reduced form of (2.2). With lemma 2.2, we can move one step further, in sliced partitions, to

$$D_1 = \inf\{\mathsf{dist}(x, y) : x, y \text{ cross some cell } c\}. \tag{2.3}$$

In particular, if all the cells of a sliced partition $\mathcal{P}$ are isometric, then D_1 is exactly the minimum distance of any points which cross the canonical cell. We can now formally define $D_0 = D_0(\mathcal{P}) := \sup\{\mathsf{dist}(x, y) : x \sim y; x, y \in \mathbb{R}^d\}$.

The next result summarizes the main points we will use from this section.

PROPERTY 2.3. *If $\mathcal{P}$ is a sliced partition with isometric cells, then $D_1 = \min\{\mathsf{dist}(x, y) : x, y \text{ cross } c\}$, where c is the canonical cell. Moreover, if there exist colinear points x, y, z such that $\overline{xy}$ is the diameter of one cell, and $\overline{yz}$ the diameter of another, then $D_0 = 2\max\{\mathsf{dist}(x, y) : x, y \in c\} = 2\,\mathsf{diam}(c)$.*

Applying this result to hypercubes, we see that $D_0 = 2\sqrt{d}$, and $D_1 = 1$ in ℓ_2. But each hypercube has 2^d corners, which would lead to exponential time complexity. The next two sections are devoted to finding the values of D_0, D_1 for two sliced partitions whose canonical cell is the convex hull of $d+1$ corners, allowing for much faster algorithms.

2.2 The orthogonal partition $\mathcal{O}$ We define the *orthogonal partition*, denoted by $\mathcal{O}$, as the sliced partition given by the slices $x_i = z : z \in \mathbb{Z}, i \in [d]$, and $x_i - x_j = z : z \in \mathbb{Z}, i \neq j \in [d]$, in $\mathbb{R}^d = \{x : x = (x_1, x_2, \ldots, x_d)\}$. We will see below that all the cells in this partition are isometric, so that we can determine D_0 and D_1 from a single cell, using property 2.3.

To begin, notice that this partition is a refinement of the hypercube partition; it contains strictly more slices. Also notice that every integer-cornered hypercube is sliced in the same way, so we may conveniently focus on the partition of $[0, 1]^d$. Given any permutation $\pi : [d] \to [d]$, we can define the set $S_\pi := \{x : 0 \leq x_{\pi(1)} \leq x_{\pi(2)} \leq \cdots \leq x_{\pi(d)} \leq 1\}$. Each inequality defines a halfspace; since S_π is a bounded, nondegenerate intersection of $d+1$ halfspaces in $\mathbb{R}^d$, it is a simplex. For any permutations $\pi_1 \neq \pi_2$, the set $S_{\pi_1} \cap S_{\pi_2}$ has measure zero. It is also clear that $\cup_\pi S_\pi = [0, 1]^d$. These simplices are precisely the cells in $[0, 1]^d$ of the orthogonal partition.

We also note that, for any permutations $\pi_1 \neq \pi_2$, the mapping $y = f(x)$ on $\mathbb{R}^d$ which follows $y_{\pi_2(i)} = x_{\pi_1(i)}$, is an isometry (in any ℓ_p) mapping $S_{\pi_1} \to S_{\pi_2}$. Thus all $d!$ cells in the hypercube are isometric, and since the same decomposition of the cube is repeated throughout space, all cells of the entire partition are isometric.

Hence we can focus on a particular cell to study. We'll choose S_{id}, where id is the identity permutation, and S_{id} is the cell for which $0 \leq x_1 \leq x_2 \leq \cdots \leq x_d \leq 1$. This cell is cornered by the points $f_i := (\underbrace{0, 0, \ldots, 0}_{d-i}, \underbrace{1, 1, \ldots, 1}_{i})$, for $i = 0, \ldots, d$.

Whenever $a \leq b \leq c$, we have $\langle f_c, f_a\rangle = \langle f_b, f_a\rangle$ so that $\langle f_c - f_b, f_a\rangle = 0$. Now consider any triangle of corners f_i, f_j, f_k with $i < j < k$. Then $\langle f_k - f_j, f_j - f_i\rangle = \langle f_k - f_j, f_j\rangle - \langle f_k - f_j, f_i\rangle = 0 - 0 = 0$. Every boundary triangle in every cell of this partition is a right triangle — this is why we call it the orthogonal partition.

$D_1(\mathcal{O})$ Next we'll find the value of $D_1(\mathcal{O})$ in any ℓ_p norm. To begin, we'll need the following elementary

LEMMA 2.4. *If $(x_i)_{i=1}^d$, $(y_i)_{i=1}^d$ are nondecreasing sequences with $\min(x_i, y_i) \leq \max(x_{i-1}, y_{i-1})$, then*

$$||x - y||_1 \geq \max(x_d, y_d) - \min(x_1, y_1).$$

Proof. Let $a_i = \min(x_i, y_i)$ and $b_i = \max(x_i, y_i)$. Then $||x - y||_1 = \sum_{i=1}^d (b_i - a_i) \geq \sum_{i=2}^d (b_i - b_{i-1}) + (b_1 - a_1) = b_d - a_1$. □

We're now ready for the main result of this section,

THEOREM 2.5.

$$D_1(\mathcal{O}) = d^{1/p-1} \text{ in } \ell_p$$

Proof. We'll begin by showing that

$$D_1 \geq 1 \text{ in } \ell_1. \tag{2.4}$$

Since $||x||_p \geq d^{1/p-1}||x||_1$, for any $x \in \mathbb{R}^d$, this will give us

$$D_1 \geq d^{1/p-1} \text{ in } \ell_p. \tag{2.5}$$

Suppose that x, y cross S_{id}. Then there are distinct corner sets $F_x, F_y \subset \{f_0, f_1, \ldots, f_d\}$ whose convex hulls

contain x and y, respectively. We can see that each point can also be viewed as a nondecreasing sequence, and that $x_i < x_{i+1} \Rightarrow y_i = y_{i+1}$ and $y_i < y_{i+1} \Rightarrow x_i = x_{i+1}$, since, e.g., $x_i < x_{i+1} \Rightarrow f_{d-i} \in F_x$, so that $z_i = z_{i+1}$ for any point $z \in \mathsf{convex}(F_y)$, including y. Our points x, y meet the conditions of lemma 2.4. Since x, y cross S_{id}, we must have $\min(x_1, y_1) = 0, \max(x_d, y_d) = 1$, and thus $||x - y||_1 \geq 1$. By property 2.3, this gives us (2.4).

It remains to be seen that we can find crossing points $x, y \in S_{\mathsf{id}}$ which actually achieve the lower bound given by (2.5). To do so, let $x = (1/d) \cdot (1, 1, 3, 3, 5, 5, \ldots, 2\lceil d/2 \rceil - 1)$ and $y = (1/d) \cdot (0, 2, 2, 4, 4, \ldots, 2\lfloor d/2 \rfloor)$. Then $x \in \mathsf{convex}(F_x)$, where $f_i \in F_x$ iff the parity of i matches that of d, and $y \in \mathsf{convex}(F_y)$, with F_y the complement of F_x. So x, y do indeed cross S_{id}. We also have $x - y = (1/d) \cdot (1, -1, 1, -1, \ldots, \pm 1)$, and $||x - y||_p = d^{1/p-1}$, which completes the proof. □

$D_0(\mathcal{O})$ Each cell in $[0, 1]^d$ contains the line segment from $\vec{0}$ to $\vec{1}$, which is a diameter of the hypercube — and hence also of each cell — in any ℓ_p norm. We can also clearly see that this diameter does occur in a colinear fashion as required by property 2.3, so that $D_0(\mathcal{O}) = 2d^{1/p}$ in ℓ_p.

We can summarize these results as $\frac{1}{2}D_0(\mathcal{O}) = 1$ in ℓ_∞ and $D_1(\mathcal{O}) = 1$ in ℓ_1.

Recall that D_0/D_1 gives us the value of the approximation factor c for the non-probabilistic version of $c-\mathsf{NN}$. We can summarize our findings thus far by the surprisingly simple equation

$$c = D_0/D_1 = 2d \text{ in } \ell_p, \text{ for any } p \in [1, \infty] \tag{2.6}$$

for the orthogonal partition.

So far, we've examined the theoretical deterministic ($\delta = 0$) bounds provided by the orthogonal partition. In order to turn this into actual code, we still need to check how feasible it is to compute the corners of a cell containing any given point. We do this in the next section.

Computing `locality_hashes(x)` in the orthogonal partition In this section we suppose there is a point $x \in \mathbb{R}^d$ for which we want to compute `locality_hashes(`x`)`. We do this by implicitly finding $c_x = \mathsf{cell}(x)$, the cell in $\mathcal{O}$ containing x; and then finding the set $\mathsf{corners}(c_x)$, which is the ultimate output of `locality_hashes` needed for the overall locality sensitive hash.

Any point $x = (x_1, x_2, \ldots, x_d) \in \mathbb{R}^d$ can be decomposed as $x = x_{\mathsf{int}} + x_{\mathsf{frac}}$, where $x_{\mathsf{int}} = (\lfloor x_1 \rfloor, \lfloor x_2 \rfloor, \ldots, \lfloor x_d \rfloor)$. Since every ($\mathbb{Z}^d-$cornered) hypercube looks the same in $\mathcal{O}$, the corners of c_x are exactly $x_{\mathsf{int}} + \mathsf{corners}(\mathsf{cell}(x_{\mathsf{frac}}))$. Thus it suffices to find the corners of $\mathsf{cell}(x)$ for any $x \in [0, 1]^d$.

```
def locality_hashes(x):
  a_corner = x_int = map(math.floor, x)
  corners = [] # an empty list
  add_corner(corners, a_corner)
  π = sort_as_permu(x - x_int)
  d = len(x)
  for i in 1..d:
    a_corner[π[d-i]] += 1
    add_corner(corners, a_corner)
  return corners
```

Figure 3: The orthogonal partition's algorithm

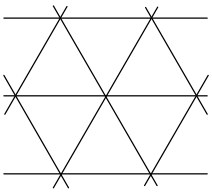

Figure 4: A rotated snapshot of $\mathcal{V}$ in $\mathbb{R}^2$ — a grid of equilateral triangles

Suppose $x \in S_\pi$ for some permutation $\pi : [d] \to [d]$. This means that

$$x_{\pi(1)} \leq x_{\pi(2)} \leq \cdots \leq x_{\pi(d)}. \tag{2.7}$$

Our job is to find this permutation based on the value of x. In other words, we simply have to sort the coordinates of x, where we think of sorting x as finding the permutation π which puts the coordinates in ascending order.

Once we know π, the corners are those of $[0, 1]^d$ which follow (2.7). Let e_i denote the canonical unit basis vector $e_i = (\underbrace{0, 0, \ldots, 0}_{i-1}, 1, \underbrace{0, 0, \ldots, 0}_{d-i})$. Then $\mathsf{corners}(S_\pi) = \{\vec{0}, e_{\pi(d)}, e_{\pi(d)} + e_{\pi(d-1)}, \ldots, \vec{1}\}$.

The pseudocode in figure 3 computes `locality_hashes(x)` for a point x given as a list of coordinates.

2.3 The vertex-transitive partition $\mathcal{V}$ In figure 2, we saw the $\mathbb{R}^2$ version of partition $\mathcal{O}$, in which each cell is an isosceles right triangle. Intuitively, one may be tempted to improve this partition by searching for even more regular cells, since this would seem to give a lower D_0/D_1 ratio. The next partition we discuss achieves this type of improvement.

We can define the *vertex-transitive partition*, denoted $\mathcal{V}$, as a particular linear transformation of $\mathcal{O}$. For any dimension d, let matrix $T = T(d)$ have diag-

onal elements $(1+(d-1)\sqrt{d+1})/d$ and off-diagonals $(1-\sqrt{d+1})/d$. Then $\mathcal{V}$ is characterized by the slices Th for each hyperplane h which is a slice of $\mathcal{O}$.

We say that a polytope p is *vertex transitive* iff, for every pair of vertices v_1, v_2, there is an isometry $\iota : p \to p$ with $\iota(v_1) = v_2$. The following fact is proved in the appendix:

PROPERTY 2.6. *In ℓ_2, the cells of $\mathcal{V}$ are isometric, and each is a vertex-transitive simplex.*

It is interesting to note that, beyond $\mathbb{R}^2$, regular simplices cannot tessellate $\mathbb{R}^d$, so that this may be considered a reasonable compromise in which some form of regularity is maintained within an isometric, simplicial tiling of space.

In the following analysis, we restrict ourselves to ℓ_2.

Since $\mathcal{V}$ is isometric, we can focus on a single particular cell to study — we choose $T(S_{\mathsf{id}})$. The corners of this cell are the points $p_i := T(f_i)$. This means that

$$p_i = \left(\frac{i}{d}\right)\vec{1} \;+\; \sqrt{d+1}\big(\underbrace{-i,\ldots,-i}_{d-i \text{ times}}, \underbrace{d-i,\ldots,d-i}_{i \text{ times}}\big), \tag{2.8}$$

from which it is a straightforward computation to see that $\langle p_i, p_j\rangle = i(d+1-j)$ for $i \le j$, and that $||p_i - p_j||_2^2 = |j-i|(d+1-|j-i|)$. This surprisingly simple formula reveals that the diameter of $T(S_{\mathsf{id}})$ is clearly given between any two points p_i, p_j which approximate $j - i = (d+1)/2$:

$$D_0(\mathcal{V}) = 2\sqrt{\left\lceil\frac{d}{2}\right\rceil\left(\left\lfloor\frac{d}{2}\right\rfloor+1\right)} = \begin{cases} d+1 & \text{odd } d, \\ \sqrt{d(d+2)} & \text{even } d. \end{cases}$$

We will invoke a general lemma giving the shortest distance from any face of $T(S_{\mathsf{id}})$ to the convex hull of the other vertices. By minimizing over all faces, this allows us to find the shortest distance between any points which cross the cell; this distance is exactly $D_1(\mathcal{V})$.

The following lemma deals with subsets of vertices whose indices are in $N_d := \{0, 1, \ldots, d\}$. It will be useful to consider the indices as residue classes modulo $d+1$. To this end, we will extend the usual interval notation $[a, b) \subset N_d$ to allow the case $a > b$, defined by: $i \in [a, b)$ iff $i \in N_d$ and $(a \le i$ or $i < b)$.

LEMMA 2.7. *Suppose $A \subset N_d$ and $A = \cup_{i=1}^{k}[a_i, b_i)$, written so that k is minimized (i.e., there are no contiguous or overlapping intervals).*

Then the points $x_A := \frac{1}{2k}\sum_i p_{a_i} + p_{b_i-1 \pmod{d+1}}$ and $x_B := \frac{1}{2k}\sum_i p_{b_i} + p_{a_i-1 \pmod{d+1}}$ minimize the distance between $\mathsf{convex}(\{p_i\}_{i\in A})$ and $\mathsf{convex}(\{p_i\}_{i\in B})$. Furthermore, $||x_A - x_B||_2^2 = \frac{d+1}{2k}$.

Proof. Let $y := \frac{2k}{d+1}(x_A - x_B)$. The first goal of the proof is to see that

$$y \perp p_i - p_j \quad \text{when either } i, j \in A \text{ or } i, j \in B. \tag{2.9}$$

This suffices to show that $\overline{x_A x_B}$ is the shortest line segment between the two affine linear subsets.

In order to decompose y, we introduce the notations $e_{-i} := e_{d+1-i}$ and $e_0 := -\vec{1}$. We will write t_i for $T(e_i)$, and $t_i^{(2)}$ for $T^2(e_i)$. Observe that $(d+1)y = \sum_i p_{a_i} - p_{a_i-1} - (p_{b_i} - p_{b_i-1}) = \sum T(e_{-a_i} - e_{-b_i}) = \sum t_{-a_i} - t_{-b_i}$.

Observe that $T = \sqrt{d+1}(I - \mu J)$, where $\mu = (1 - 1/\sqrt{d+1})/d$ and J is the all-one matrix, so that $T^2 = (d+1)I - J$. Then, for $i > 0$,

$$\begin{aligned}\langle t_{-i}, p_j\rangle = \langle e_{-i}, T^2 f_j\rangle &= \left\langle e_{-i}\,,\ \sum_{k=d+1-j}^{d} t_k^{(2)} \right\rangle \\ &= (d+1)(i \le j) - j,\end{aligned} \tag{2.10}$$

where the notation $n(\mathsf{boolean})$ denotes value n if the boolean is true, 0 otherwise.

Using (2.8), we also have $\langle t_0, p_j\rangle = \langle -\vec{1}, p_j\rangle = -j$. Using this, we can re-write (2.10) to include the $i = 0$ case as $\langle t_{-i}, p_j\rangle = (d+1)\big(j \in [i, 0)\big) - j$, where $[0, 0)$ is interpreted as the empty set.

Let $g_i := t_{-a_i} - t_{-b_i}$. Then $\frac{1}{d+1}\langle g_i, p_j\rangle = 1(j \in [a_i, b_i)) - 1(0 \in [a_i, b_i))$. Let $w := \#\{i : 0 \in [a_i, b_i)\}$. Notice that $y = \frac{1}{d+1}\sum_i g_i$, so that

$$\langle y, p_j\rangle = \frac{1}{d+1}\sum_i \langle g_i, p_j\rangle = \begin{cases} 1-w & \text{if } j \in A \\ -w & \text{if } j \in B. \end{cases} \tag{2.11}$$

From this, we have that $\langle y, p_i - p_j\rangle = 0$ whenever $i, j \in A$ or $i, j \in B$. This finishes the demonstration of (2.9), the first half of this proof.

Next we confirm the distance between x_A and x_B. Using (2.11), $(d+1)\langle y, y\rangle = \sum_i \langle y, (p_{a_i} + p_{b_i-1}) - (p_{b_i} + p_{a_i-1})\rangle = 2k$. Thus $||x_A - x_B||_2^2 = \left(\frac{d+1}{2k}\right)^2\langle y, y\rangle = \frac{d+1}{2k}$, which concludes the proof. □

The lemma tells us that we can achieve the minimum distance by maximizing k, the number of intervals needed to express A (or B). For instance, we could always choose $A = \{i \in N_d : i \text{ is even}\}$, in which case $k = \lceil d/2\rceil$. Hence $D_1(\mathcal{V}) = \sqrt{(d+1)/d}$ for even d, and 1 for odd d.

Combining this with $D_0(\mathcal{V})$, we see that, in ℓ_2,

$$c = \frac{D_0}{D_1} = \begin{cases} d+1 & \text{if } d \text{ is odd,} \\ d\sqrt{1+\frac{1}{d+1}} & \text{if } d \text{ is even.} \end{cases} \tag{2.12}$$

We always have $c \le d+1$, so that the guaranteed locality-sensitive hashing accuracy of $\mathcal{V}$ is essentially

twice as good as that of $\mathcal{O}$; although the accuracy of $\mathcal{V}$ has only been verified in ℓ_2, whereas our analysis of $\mathcal{O}$ applies for any ℓ_p.

Computing `locality_hashes(x)` in $\mathcal{V}$ We could summarize the definition of this partition via $\mathcal{V} = T(\mathcal{O})$. Accordingly, if `locality_hashes_in_O(x)` computes the corners of the cell in $\mathcal{O}$ containing x, then `locality_hashes_in_O(`T^{-1}`x)` does the same for $\mathcal{V}$.

We have $T^{-1} = \frac{1}{\sqrt{d+1}}I + \mu J$, where $\mu = \left(1 - \frac{1}{\sqrt{d+1}}\right)\frac{1}{d}$. If $y = T^{-1}x$, then $y_i = x_i/\sqrt{d+1} + \mu s_x$, where $s_x = \sum_i x_i$. This is computable in $O(d)$ time.

2.4 Time and space complexities As in §2.1, we will use σ to denote the size of the return value of `locality_hashes`; and τ to denote the time needed for computation. The algorithms presented here need at least $\tau = \Omega(d \log d)$ worst-case time in order to sort the coordinates of x_{frac}, as described in §2.2. Unfortunately, the output has size $\sigma = \theta(d^2)$ — there are $d+1$ corners, each in $\mathbb{Z}^d$ — so that we need at least $\tau = \Omega(d^2)$ time simply to store and return all coordinates of each cell corner.

In the next section, we mention a method to reduce these complexities.

A trick for faster hashes The bottleneck in the algorithms thus far is the fact that we need at least $\Omega(d^2)$ space to store and retrieve the list of all $d+1$ corners (each in $\mathbb{Z}^d$) of the cell containing x.

If the hashed value $h(v)$ of each vector $v \in \mathbb{Z}^d$ is of the form $h(v) \equiv \langle u, v\rangle \pmod N$ for some constant vector u and constant modulus N, then we can achieve complexity $O(d \log d)$ for σ and τ. To do so, it suffices to first compute $h(x_{\mathsf{int}})$ and then subsequently add $u_{\pi(i)}$ modulo N, for each $i = d, d-1, \ldots, 1$, where π is the sorting permutation as discussed in §2.2. For fast lookups, we only need to store $(x_{\mathsf{int}}, \pi^{-1})$ in a new table, and a value k associated with each hash key corresponding to $x_{\mathsf{int}} + e_{\pi(d)} + \ldots + e_{\pi(k)}$. The exact i^{th} coordinate of a key can be retrieved in constant time as $(x_{\mathsf{int}})_i + 1(k \le \pi^{-1}(i))$.

Fast lookups can be achieved by only examining $\log d$ random coordinates to verify the key of each hash table entry. This allows for $O(d \log d)$ query time with a small risk of extra hash table collisions ([1]).

3 Empirical performance

In this section we present empirical evidence on artificial data suggesting that our algorithm compares favorably against previously known locality hashes.

3.1 Related hashes We compare our algorithm to three other locality hashes for real vectors.

The first, presented in [8], is based on a projection of the points into Hamming space, followed by a projection onto $\mathbb{Z}^k$. This algorithm is very simple to implement, although it requires a fixed bound on the input vectors; because it utilizes the unary representation of coordinates, we refer to it as the unary hash.

In [7], an algorithm was presented based primarily on dimension reduction tailored to preserve distances well in certain ℓ_p metrics. The reduced vector is then abridged in a coordinate-wise fashion into $\mathbb{Z}^k$. Intuitively, the equivalence classes of this hash can be thought of as hypercubes in the reduced space. We refer to this as the pstable hash.

The authors of [2] specifically design their hash so that the equivalence classes, in a reduced space, are spheres (or occluded spheres) instead of hypercubes, which will always result when concatenating a series of 1-dimensional hashes. We refer to this as the sphere-grid hash, as it is built on a sequence of grids of spheres.

Also mentioned in [2] is the idea to use the Leech lattice for $d = 24$, or dimension reductions thereto. We have not compared against the Leech lattice primarily because, as a locality hash (i.e., without dimension reduction), it exists for only one d, while we are interested in the ability of other hashes which also scale to arbitrary dimensions.

3.2 Test data and performance metrics All of the locality hashes were tested by repeatedly generating a uniform random point $x \in [0, C]^d$, then a random unit vector $w \in \mathbb{R}^d$, and checking to see if $x \sim x + Dw$ for a given distance D, where $x \sim y$ means x, y are considered nearby according to whichever algorithm is being tested. In this way, we simulate data that has been randomly shifted and rotated before processing, and can estimate the probability $P(x \sim y \,|\, \mathsf{dist}(x,y) = D)$, over the probability space of this random data movement, for each distance D. Effectively, we are estimating worst-case bounds on the approximation factor c for any query/point pair.

Figure 5 shows $P(x \sim y|D)$ versus distance D for each algorithm. For easier comparison, the scale of each algorithm has been chosen so that the 50% point is at $D = 1$; better accuracy is indicated by a steeper descent around this threshold. We exclude sphere grid from the $d = 500$ graph because our implementation had difficulty scaling to this dimension.

It is also interesting to estimate which approximation factors are available at certain confidence levels for each algorithm. Let $f(D) := P(x \sim y|\mathsf{dist}(x,y) = D)$. We assume that f is a strictly decreasing function when $f(D) \in (0,1)$, which appears to be true for all tested algorithms. Then we can define $D_p := f^{-1}(p)$ for

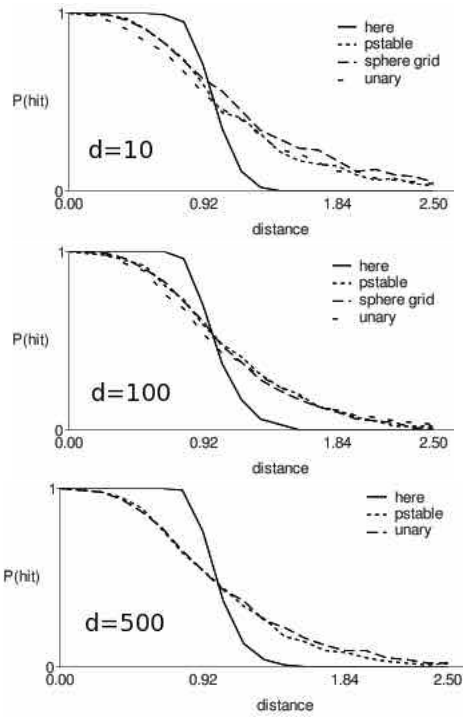

Figure 5: Hit probabilities versus point distance for dimensions 10, 100, and 500, top to bottom.

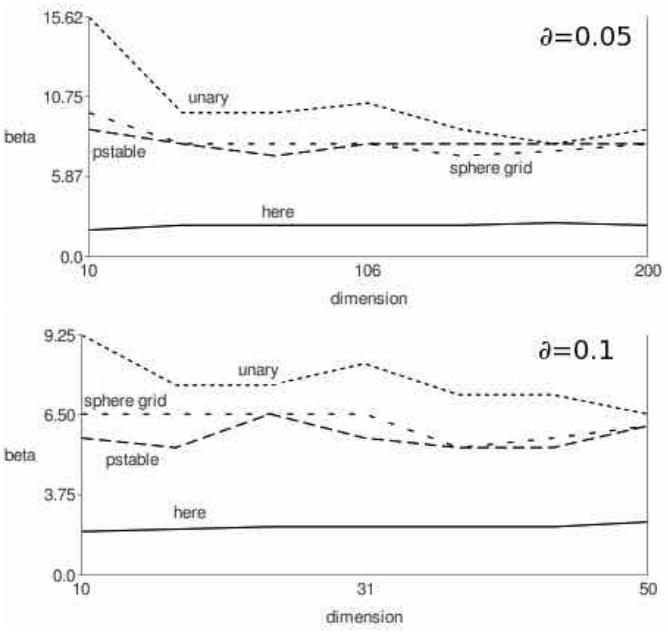

Figure 6: Left: $\beta_{0.05}$, right: $\beta_{0.1}$ versus dimension.

hash	$\beta_{.01}$	$\beta_{.1}$	$\beta_{.3}$
unary	18	6.2	3.1
sphere grid	16	5.6	2.8
pstable	16	5.1	2.8
here	2.2	1.6	1.4

Table 2: β_δ estimates for $d = 20$.

$p \in (0, 1)$. Next, let $\beta_\delta := D_{\delta/2}/D_{1-\delta/2}$. We know with confidence at least $1-\delta$ that $\mathsf{dist}(x, y) < D_{1-\delta/2} \Rightarrow x \sim y$, and $\mathsf{dist}(x, y) > D_{\delta/2} \Rightarrow x \not\sim y$; when both are true, we have $c \leq \beta_\delta$ for any single point p.

Figure 6 plots $\beta_{.05}$ and $\beta_{.1}$ for various dimensions. Note that c grows minimally as d increases. Table 2 gives β estimates in $d = 20$ for confidence levels 99%, 90%, and 70%.

Parameters Our choices for the parameters of each algorithm was based primarily on a requirement that each query lookup be fast, and next on practical choices hinted at by the various authors.

Some previous algorithms use a parameter L to indicate the number of hash tables used, where each hash table $t \in [L]$ is slightly altered, such as by translating the input point by some vector $u_t \in \mathbb{R}^d$. In those cases, the number of hash buckets per point (we'll call this L_b) was equal to the number of hash tables (call this L_t) — but here, we conceptually have $d+1$ buckets per point in a single table. Because speed is a focus of this paper, we compare algorithms with the same L_t value, which is a more uniform measure of time complexity. In particular, we use the constant value $L_t = 5$ for most empirical data. In table 1, we see that setting $L_t = d + 1$ offers modest accuracy gains at the cost of slower running time (figure 7). All empirical data is run using the $O(d \log d)$ query time version of our algorithm.

For unary, we chose to use the fixed number of projected coordinates per hash vector $k = 700$, a value suggested in [8]. For pstable, we used $r = 1$ as the denominator for each 1-dimensional hash. In both pstable and sphere grid, we used $k = 2 \log d$ as the reduced dimension (the reduced dimension is called t in [2]). In sphere grid, we used $\sigma = 2$; larger values (such as $\sigma = 4$, as suggested in [2]) required an order of magnitude more grids to fill the space, and were thus significantly slower.

Reproducing the experiments Source code to easily reproduce all our experiments, along with scripts to produce our particular graphs and tables, is available from: `http://thetangentspace.com/lsh/`

4 Future work

A natural next step is to corroborate the empirical performance measurements with provable timing-vs-accuracy bounds.

Many locality-sensitive hash families $\mathcal{H}$ (including those considered here) can be intuitively understood as

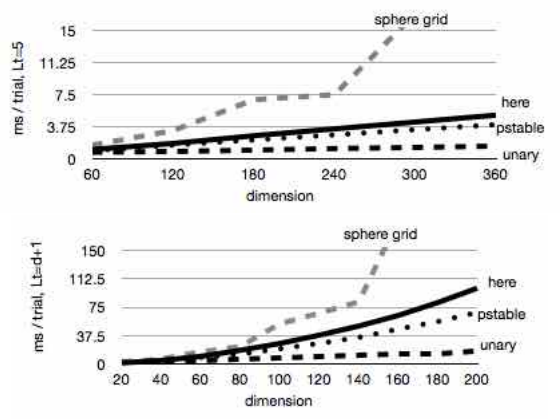

Figure 7: Timing data, in ms per trial. A trial is two point storages and one lookup. Note the different scales on both axes, which further emphasize the (not unexpected) speed difference between the $L_t = 5$ and $L_t = d + 1$ cases.

matching a point x primarily to an equivalence class $\mathsf{near}(x, i) := \{y \mid h_i(y) = h_i(x)\}$ for each particular hash $h_i \in \mathcal{H}$, and ultimately to a larger set $\mathsf{near}(x, I) := \cup_{i \in I} \mathsf{near}(x, i)$. It is inevitable that $x \sim y$ for any $y \in \mathsf{core}(x, I) := \cap_{i \in I} \mathsf{near}(x, i)$. Ideally, we would also have

$$\mathsf{near}(x, I) = \cup_{y \in \mathsf{core(x,I)}} B(y, R),$$

where R is the radius of detection for the hash. Intuitively, this perspective gives us a strong hint toward making sure that each $\mathsf{near}(x, i)$ is sphere-like — an intuition followed by [2], and, less directly, here.

It's not too difficult to check that $\mathsf{near}(x, I)$ for our hash is a Voronoi cell in a lattice solving the sphere-packing problem for $d = 2, 3$, although not for any larger d. Perhaps there is a trade-off in maintaining that the core set is a simplex while each $\mathsf{near}(\cdot, i)$ set is sphere-like. The excellent sphere-packing properties of the Leech lattice ([6]) suggest it would give a good individual hash ([2]), and perhaps perform better when complemented by the other Neimeier lattices (although they could not fit exactly into our simplicial framework since there are only 24 of them — we need $d + 1 = 25$ corners, one from each hash).

It has not escaped our notice that the tessellations presented in this paper may themselves be of geometric interest. Are they, in some sense, the "most regular" simplices which isometrically tile space? For the purposes of locality-sensitive hashes, we propose the following optimality of our tessellation:

CONJECTURE 4.1. *Among all sliced partitions into isometric simplices, the canonical simplex c of $\mathcal{V}$ minimizes the ratio between diameter and the shortest distance between any points which cross c.*

5 Thanks

Thanks to Dennis Shasha, Mehryar Mohri, Corinna Cortes, and Manfred Warmuth for their invaluable mentoring and generous support.

References

[1] D. Achlioptas. *Database-friendly random projections*, Symposium on principles of database systems, 2001.

[2] A. Andoni and P. Indyk, *Near-optimal hashing algorithm for approximate nearest neighbor in high dimensions*, Communications of the ACM, 51:117–122, 2008.

[3] A. Broder, M. Charikar, A. Frieze and M. Mitzenmacher, *Min-wise independent permutations*, Proceedings of the 30th annual ACM symposium on theory of computing, 1998.

[4] T.M. Chan, *Approximate nearest neighbor queries revisited*, Disc. Comp. Geom., 20(1998), pp. 359–373.

[5] T.M. Chan, *Closest-point problems simplified on the RAM*, Proceedings of the ACM-SIAM Symposium on discrete algorithms, 2002.

[6] J.H. Conway and N.J.A. Sloane, *Sphere packings, lattices, and groups*, Springer, 1993.

[7] M. Datar, N. Immorlica, P. Indyk and V. Mirrokni, *Locality-sensitive hashing scheme based on p-stable distributions*, Proceedings of the 20th annual symposium on computational geometry, 2004.

[8] A. Gionis, P. Indyk and R. Motwani, *Similarity search in high dimensions via hashing*, Proceedings of the 25th international conference on very large databases, 1999.

[9] V. Guruswami, J. Lee and A. Razborov, *Almost Euclidean subspaces of ℓ_1 via expander codes*, Proceedings of the ACM-SIAM Symposium on discrete algorithms, 2008.

[10] P. Indyk, *Uncertainty principles, extractors, and explicit embeddings of ℓ_2 into ℓ_1*, Proceedings of the 39th annual ACM symposium on theory of computing, 2007.

[11] P. Indyk and R. Motwani, *Approximate nearest neighbor: toward removing the curse of dimensionality*, Proceedings of the 35th IEEE Symposium on Foundations of Computer Science, 1998.

[12] W. Johnson and J. Lindenstrauss, *Extensions of Lipschitz maps into a Hilbert space*, Contemp. Math., 26(1984), pp. 189–206.

[13] G. Manku, A. Jain and A. Sarma, *Detecting near-duplicates for web crawling*, Proceedings of the 16th international conference on WWW, 2007.

[14] R. Panigrahy, *Entropy-based nearest neighbor algorithm in high dimensions*, Proceedings of the ACM-SIAM Symposium on discrete algorithms, 2006.

[15] H. Samet, *Foundations of Multidimensional and Metric Data Structures*, Elsevier, 2006.

A Appendix

Proof of lemma 2.1.

Let y' be the first point on line $\overline{xy}$ moving from y to x which is the entry point to a new cell; i.e., such that there is some cell $c_{y'}$ for which $y' \in c_{y'}$ but $y'' \notin c_{y'}$ for any previous point y'' on this line. Note that we may have $y' = y$.

Notice that y' must be on the boundary of the new cell, so that there is some cell c' containing both y and y'. Thus, if $y' \sim x$, then we are done, since x is in some cell $c_x \not\sim c'$. From here on, we'll assume that $y' \not\sim x$.

Let $x_0 = x$, and, similar to our choice of y', let x_1 be the next point on the line moving from x to y which is the entry point to a new cell. We can continue to define a sequence $x_1, x_2, \ldots$ with each successive point being the next along $\overline{xy}$ toward y which is the entry point to a new cell.

Let x_k be the first in this sequence such that $x_k \sim y'$. Since $x \not\sim y'$, we have $k > 0$. As above (for y'), there must be a cell c_k containing both x_{k-1} and x_k. By our choice of x_k, we know that $c_k \not\sim c'$. Thus $x' = x_k$ and y' are the desired pair of barely neighboring points. □

Figure 8 shows an example case to illustrate the above proof.

Proof of lemma 2.2. We begin with the $\Rightarrow$ direction. Suppose x, y don't cross c. If either point is not in the boundary of c, then clearly they cannot be barely neighbors. Instead, let's assume that the corner sets $C_x, C_y \subset \mathsf{corners}(c)$ with $x \in \mathsf{convex}(C_x), y \in \mathsf{convex}(C_y)$, both contain a common point $z \in C_x \cap C_y$.

If there were a cell c' containing x but not z, then the polytope $c \cap c'$ would be defined by corners excluding z, yet this shape would include x. This contradicts the necessity of z in C_x, so there cannot be any such cell. This same argument applies to all cells containing y. Therefore, any pair of cells c', c'' containing x, y must also share the point z, so that $c' \sim c''$, and x, y cannot be barely neighbors.

Next we show the $\Leftarrow$ direction.

We will refer to any $k-$dimensional set of the form $\mathsf{convex}(F)$, for any $F \subset \mathsf{corners}(c)$, as a *$k-$face* of c. Thus a point is a 0-face, an edge is a 1-face, etc.

We claim that, for every $k-$face f of c, there is another cell c' such that $c \sim c'$ and $c \cap c' = f$.

First we'll complete the proof using this claim, and then justify it. Suppose that x, y cross cell c. Let f_x, f_y denote the minimal $k-$faces of c which contain x and y, respectively. Now choose cells c_x, c_y so that $x \in c_x, y \in c_y$ and $c_x \cap c = f_x$, $c_y \cap c = f_y$, using the claim. If $c_x \sim c_y$, then there is some point $z \in c_x \cap c_y$. By the convexity of the cells, c_x and c_y contain the line segments $\overline{xz}$ and $\overline{yz}$ respectively. This means that there can be no slicing hyperplane which intersects the triangle formed by x, y, z, which in turn implies that $z \in c$ (see figure 9). By our minimal choices of the $k-$faces, we have that $z \in f_x$ and $z \in f_y$. But this contradicts the fact that x, y cross c! Hence there can exist no common point z between c_x, c_y; and $c_x \not\sim c_y$ so that x, y are barely neighbors.

Now let's justify the claim that there is a cell c' with $c \cap c' = f$ for every $k-$face f of cell c.

Suppose f is a $k-$face of cell c. Then f is the intersection of $d-k$ hyperplanes with c, where each hyperplane is a slice of the partition that determines some $d-1-$face of c. Let c' be any cell on the opposite side of all of these hyperplanes, and with $c \sim c'$. If $x \in c \cap c'$, then x must be a point in each of these $d-k$ hyperplanes, so that $x \in f$. It is also clear that $f \subset c \cap c'$; so we may conclude that $f = c \cap c'$, which was our goal. □

Proof of property 2.6.

$\mathcal{V}$ is isometric in ℓ_2 Let's check that $\mathcal{V}$ is an isometric partition — that every pair of cells is isometric. Similar to our previous argument, we'll begin by showing that every image, under T, of a lattice-cornered hypercube is isometric to the image of $[0,1]^d$.

Let u denote the unit hypercube $[0,1]^d$ and $\tilde{u} = T(u)$. Similarly, let c denote an offset hypercube $c = u + z, z \in \mathbb{Z}^d$, and $\tilde{c} = T(c)$. We would like to find an isometry from $\tilde{c} \to \tilde{u}$. Let A_z be the shift $A_z(x) = x - z$, so that $A_z(c) = u$. Then clearly $\tilde{A}_z := T \circ A_z \circ T^{-1}$ maps $\tilde{c}$ to $\tilde{u}$. Notice that

$$(TA_zT^{-1})(x) = T(T^{-1}(x) - z) = x - Tz,$$

so that $\tilde{A}_z$ is indeed an isometry.

Next we'd like to check that any two simplices $T(S_{\pi_1}), T(S_{\pi_2})$ are isometric as well. Suppose $\pi : [d] \to [d]$ is a permutation, and let U_π denote the mapping given by $y_i = x_{\pi(i)}$ when $y = U_\pi(x)$. At this point we note that T can be written in the form $T = \alpha I + \beta J$, where I is the identity matrix, J is the all-one matrix, $\alpha = \sqrt{d+1}$, and $\beta = (1 - \sqrt{d+1})/d$. Then

$$\begin{aligned} U_\pi^T T U_\pi &= U_\pi^T(\alpha I + \beta J)U_\pi \\ &= \alpha I + \beta(U_\pi^T J U_\pi) = \alpha I + \beta J = T. \end{aligned}$$

Using the fact that U_π is unitary (so $U_\pi^T = U_\pi^{-1}$), we can see that $TU_\pi T^{-1} = U_\pi$. So if $U_\pi : S_{\pi_1} \to S_{\pi_2}$, then we still have $U_\pi : T(S_{\pi_1}) \to T(S_{\pi_2})$, also isometrically.

As a final note, observe that any two isometries $\tilde{A} = T \circ A \circ T^{-1}$ and $\tilde{B} = T \circ B \circ T^{-1}$ may be composed, and preserve the conjugate relationship $\tilde{A} \circ \tilde{B} = T \circ (A \circ B) \circ T^{-1}$. This justifies that a series of the above tranformations — translations and permutations

— suffices to produce an isometry between any two cells of $\mathcal{V}$, and this partition can be studied by considering a single canonical cell.

The canonical cell is vertex-transitive in ℓ_2 Next we'll choose a particular canonical cell to study, and justify the name of this partition by showing that it has the type of polytope regularity known as vertex transitivity. Recall that a polytope p is *vertex transitive* iff, for every pair of vertices v_1, v_2, there is an isometry $\iota : p \to p$ with $\iota(v_1) = v_2$.

Our canonical cell in $\mathcal{V}$ is $T(S_{\mathsf{id}})$, with corners $p_i := T(f_i)$. To show that this cell is vertex-transitive, it will suffice to construct mappings $m_{i,j} : T(S_{\mathsf{id}}) \to T(S_{\mathsf{id}})$, for any $i, j \in \{0, 1, \ldots, d\}$, such that $m_{i,j}$ is an isometry and $m_{i,j}(p_i) = p_j$.

We proceed by constructing a (non-isometric) map $V : S_{\mathsf{id}} \to S_{\mathsf{id}}$ which acts as a shift operator on the corners of this simplex. If we define the matrix

$$W := \begin{pmatrix} \vec{-1} & I \\ -1 & \vec{0} \end{pmatrix},$$

and let $V(x) := Wx + f_1$, then

$$\begin{aligned} V(f_i) &= f_{i+1} \quad \text{for } i < d; \\ V(f_d) &= f_0. \end{aligned}$$

Since V is an affine transformation, $V(\mathsf{convex}(f_0, f_1, \ldots, f_d)) = \mathsf{convex}(V(f_0, f_1, \ldots, f_d))$, verifying that $V : S_{\mathsf{id}} \to S_{\mathsf{id}}$.

To operate in the new partition, we extend V via $\tilde{V} := TVT^{-1}$. Clearly, $\tilde{V}$ maps the canonical cell to itself, and shifts the corners $p_i \to p_{i+1} \pmod{d+1}$. Hence any desired mapping $p_i \to p_j$ can be acheived by the correct number of iterations of $\tilde{V}$; that is, we are close to seeing that $m_{i,j} = \tilde{V}^k$, where $k \equiv j - i \pmod{d+1}$.

The last step is to confirm that $\tilde{V}$ is an isometry. This is the case iff $\tilde{W} = TWT^{-1}$ is itself isometric. Let $\mu := (1 - 1/\sqrt{d+1})/d$, so that $T = \sqrt{d+1}(I - \mu J)$; and $T^{-1} = \frac{1}{\sqrt{d+1}}(I + \sqrt{d+1}\mu J)$. Then $\tilde{W} = (I - \mu J)W(I + \sqrt{d+1}\mu J)$, and from here one may tediously simplify the corresponding formulae for $\langle w_i, w_j \rangle$ (where w_i is the i^{th} column of $\tilde{W}$) to confirm that $\tilde{W}^T\tilde{W} = I$.

This suffices to demonstrate that $T(S_{\mathsf{id}})$ is indeed a vertex-transitive simplex. □

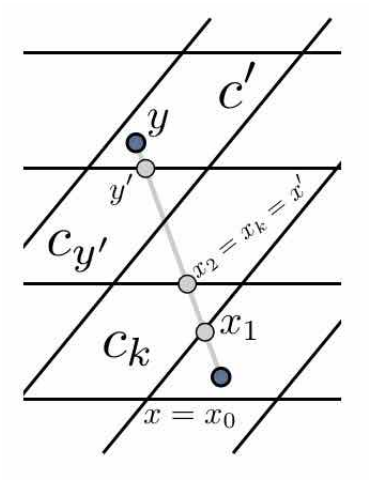

Figure 8: Example from the proof of lemma 2.1. Here, $x \not\sim y$ and $x' \sim y'$ are barely neighbors

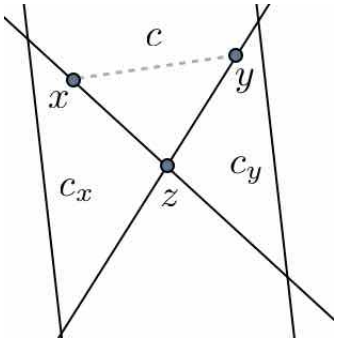

Figure 9: For the proof of lemma 2.2: If $\overline{xz}$ and $\overline{yz}$ are both intact – i.e., do not intersect a slicing hyperplane – then all of Δxyz must exist in a single cell.