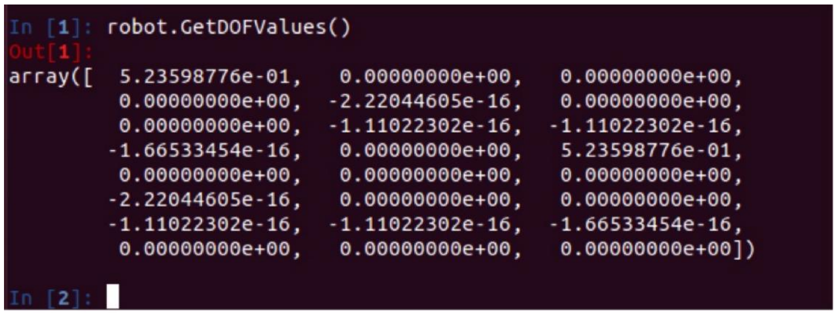
Nearest Pose Retrieval Based on K-d tree

Contents

1.1 Introduction of KDTree

Since HERB has 24 degrees of freedom[1] which create a high-dimensional pose space, if each joint orientation is represented as one dimension and thus a pose is represented as a vector, as shown in following picture.



```
In [1]: robot.GetDOFValues()
Out[1]:
array([  5.23598776e-01,   0.00000000e+00,   0.00000000e+00,
         0.00000000e+00,  -2.22044605e-16,   0.00000000e+00,
         0.00000000e+00,  -1.11022302e-16,  -1.11022302e-16,
        -1.66533454e-16,   0.00000000e+00,   5.23598776e-01,
         0.00000000e+00,   0.00000000e+00,   0.00000000e+00,
        -2.22044605e-16,   0.00000000e+00,   0.00000000e+00,
        -1.11022302e-16,  -1.11022302e-16,  -1.66533454e-16,
         0.00000000e+00,   0.00000000e+00,   0.00000000e+00])

In [2]: 
```

There are times when we want to retrieve poses that meet certain requirements, for instance, we want to reach a nearest rest pose, which is a starting pose to many animations, to prepare for next step animations. Such search may take very long as the pose library gets huge, if we perform a linear search by brutal force. The sheer dimensions could cause the curse of dimensionality. (Chap 2)

When a measure such as a Euclidean distance is defined using many coordinates, there is little difference in the distances between different pairs of samples.

[1] may vary on adding or dropping components, and methods of calculation

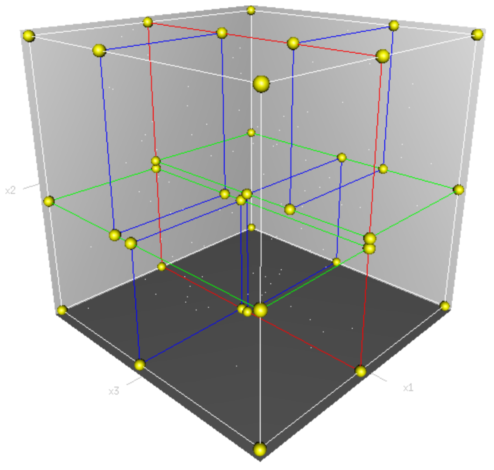
One way to illustrate the "vastness" of high-dimensional Euclidean space is to compare the proportion of a hypersphere with radius r and dimension d, to that of a hypercube with sides of length 2r, and equivalent dimension.

The volume of such a sphere is: $\frac{2r^d\pi^{d/2}}{d\Gamma(d/2)}$. The volume of the cube would be: $(2r)^d$. As the dimension d of the space increases, the hypersphere becomes an insignificant volume relative to that of the hypercube. This can clearly be seen by comparing the proportions as the dimension d goes to infinity:

$\frac{\pi^{d/2}}{d2^{d-1}\Gamma(d/2)} \to 0$ as $d \to \infty$.

In computer science, a k-d tree (short for k-dimensional tree) is a space-partitioning data structure for organizing points in a k-dimensional space. k-d trees are a useful data structure for several applications, such as searches involving a multidimensional search key (e.g. range searches and nearest neighbor searches). k-d trees are a special case of binary space partitioning trees.

The k-d tree is a binary tree in which every node is a k-dimensional point. Every non-leaf node can be thought of as implicitly generating a splitting hyperplane that divides the space into two parts, known as half-spaces. Points to the left of this hyperplane are represented by the left subtree of that node and points right of the hyperplane are represented by the right subtree. The hyperplane direction is chosen in the following way: every node in the tree is associated with one of the k-dimensions, with the hyperplane perpendicular to that dimension's axis. So, for example, if for a particular split the "x" axis is chosen, all points in the subtree with a smaller "x" value than the node will appear in the left subtree and all points with larger "x" value will be in the right subtree. In such a case, the hyperplane would be set by the x-value of the point, and its normal would be the unit x-axis.



Using KDTree, we could potentially minimize the time cost of retrieving the nearest pose, even if we have a huge pose library.

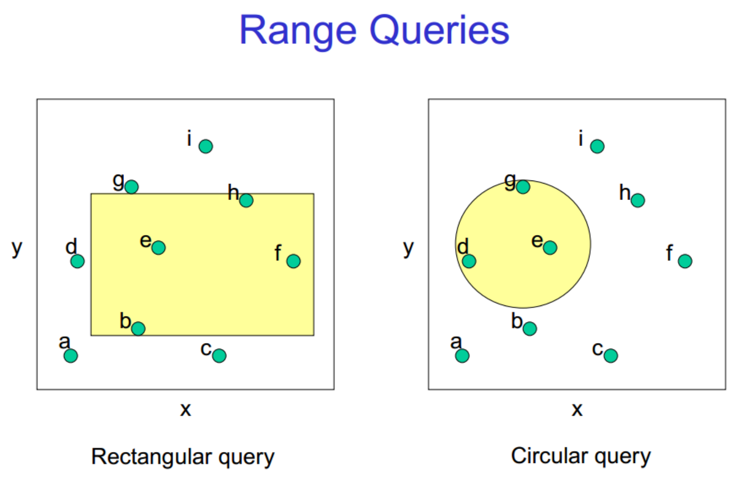
1.2 The curse of dimensionality

The curse of dimensionality refers to various phenomena that arise when analyzing and organizing data in high-dimensional spaces (often with hundreds or thousands of dimensions) that do not occur in low-dimensional settings such as the three-dimensional physical space of everyday experience.

There are multiple phenomena referred to by this name in domains such as numerical analysis, sampling, combinatorics, machine learning, data mining and databases. The common theme of these problems is that when the dimensionality increases, the volume of the space increases so fast that the available data become sparse. This sparsity is problematic for any method that requires statistical significance. In order to obtain a statistically sound and reliable result, the amount of data needed to support the result often grows exponentially with the dimensionality. Also organizing and searching data often relies on detecting areas where objects form groups with similar properties; in high dimensional data however all objects appear to be sparse and dissimilar in many ways which prevents common data organization strategies from being efficient.

1.3 Nearest neighbor search (NNS)

Nearest neighbor search (NNS), also known as proximity search, similarity search or closest point search, is an optimization problem for finding closest (or most similar) points. Closeness is typically expressed in terms of a dissimilarity function: The less similar are the objects the larger are the function values. Formally, the nearest-neighbor (NN) search problem is defined as follows: given a set S of points in a space M and a query point $q \in M$, find the closest point in S to q. Donald Knuth in vol. 3 of The Art of Computer Programming (1973) called it the post-office problem, referring to an application of assigning to a residence the nearest post office. A direct generalization of this problem is a k-NN search, where we need to find the k closest points.



Most commonly M is a metric space and dissimilarity is expressed as a distance metric, which is symmetric and satisfies the triangle inequality. Even more common, M is taken to be the d-dimensional vector space where dissimilarity is measured using the Euclidean distance, Manhattan distance or other distance metric. However, the dissimilarity function can be arbitrary. One example are asymmetric Bregman divergences, for which the triangle inequality does not hold.

1.4 Python Implementation of KDtree

```
from collections import namedtuple
from operator import itemgetter
from pprint import pformat

class Node(namedtuple('Node', 'location left_child right_child')):
    def __repr__(self):
        return pformat(tuple(self))

def kdtree(point_list, depth=0):
    try:
        k = len(point_list[0]) # assumes all points have the same dimension
    except IndexError as e: # if not point_list:
        return None
    # Select axis based on depth so that axis cycles through all valid values
    axis = depth % k
    # Sort point list and choose median as pivot element
    point_list.sort(key=itemgetter(axis))
    median = len(point_list) // 2 # choose median

    # Create node and construct subtrees
    return Node(
        location=point_list[median],
        left_child=kdtree(point_list[:median], depth + 1),
        right_child=kdtree(point_list[median + 1:], depth + 1)
    )

def main():
    """Example usage"""
    point_list = [(2,3), (5,4), (9,6), (4,7), (8,1), (7,2)]
    tree = kdtree(point_list)
    print(tree)

if __name__ == '__main__':
    main()
```

1.5 C++ Implementation of KDTree

```
void KDTree::createTree(std::vector<Photon> &v){

          if(v.size() > 0)

                    createTree(&root, v, 0, v.size(), 0);

}

// search the nearest n points of given p



my_pq KDTree::searchN(const Vector3 &p, const Vector3 &n, const size_t N){

          my_pq res((comparison(p)));
          if(size > 0)

                    searchTree(p, n, N, &root, res, 0);

          return res;

}



inline bool compareX(const Photon a, const Photon b){

          return (a.position.x < b.position.x);

}



inline bool compareY(const Photon a, const Photon b){

          return (a.position.y < b.position.y);

}



inline bool compareZ(const Photon a, const Photon b){

          return (a.position.z < b.position.z);

}




// recursively create the kdtree



void KDTree::createTree(Node *node,

                    std::vector<Photon> &v, size_t st, size_t ed, int depth){



          size++;   // size of the total tree increase
```

```
            // sort the vector depends on the depth

            switch(depth%3){

            case 0:

                      std::sort(v.begin() + st, v.begin() + ed, compareX);//sort comparing X

                      break;

            case 1:

                      std::sort(v.begin() + st, v.begin() + ed, compareY);//sort comparing Y

                      break;

            case 2:

                      std::sort(v.begin() + st, v.begin() + ed, compareZ);//sort comparing Z

                      break;

            }



            // get the median of sorted v

            // median for KD tree separation

            size_t mid = (ed + st)/2;

            // assign it to the map

            node->photon = &v[mid];

            // create child recursively

            if(st < mid){

                      node->left = new Node();

                      createTree( node->left, v, st, mid, depth+1);

            }
```

```
          if(mid+1 < ed){

                    node->right = new Node();

                    createTree(node->right, v, mid+1, ed, depth+1);

          }

}



// in plane threshould 0.05

inline bool checkInPlane(const Vector3 &ph,const Vector3&p,const Vector3&n){

          return (fabs(dot(ph - p, n)) < 0.05);

}



// find the farthest point, true if replaced

inline bool replace(const Vector3 &p,const Vector3& n, my_pq &res, Photon* ph){

          if(distance(res.top()->position, p) > distance(p,ph->position)){

                    if (checkInPlane(ph->position, p,n)){

                              res.pop();

                              res.push(ph);

                              return true;

                    }

          }

          return false;

}
```

```
// search the tree, for n nearest photons



void KDTree::searchTree(const Vector3 &p, const Vector3 &n, const size_t N,

                                        const Node *node, my_pq &res, int depth){



          // if reach the leaf

          if (node->left == NULL && node->right == NULL)

          {

                    if(res.size() < N){

                              res.push(node->photon);

                    }

                    else{

                              replace(p,n,res,node->photon);

                    }

                    return;

          }

          else if(node->left == NULL){

                    searchTree(p,n,N,node->right,res, depth+1);

                    return;

          }

          else if(node->right == NULL){

                    searchTree(p,n,N,node->left,res, depth+1);

                    return;

          }
```

```
        // split depends on the x/y/z value

        int id = depth%3;

        real_t disToPlane = p[id] - node->photon->position[id];

        // mark the searched tree, true to find in left, false to find in right

        bool lt = false;

        if(disToPlane < 0.0){

                searchTree(p,n,N, node->left,res, depth+1);

        }

        else{

                searchTree(p,n,N,node->right,res, depth+1);

                lt = true;

        }



        // unwind the tree

        if(fabs(disToPlane) < distance(p,res.top()->position) || res.size() < N){

                // replace and need to check the other branch
```

```
                if(res.size() < N){

                        if (checkInPlane(node->photon->position, p,n))

                                res.push(node->photon);

                }

                else

                        replace(p, n,res,node->photon);
                        if(lt)

                                    searchTree(p,n,N, node->left,res, depth+1);

                        else

                                    searchTree(p,n,N,node->right,res, depth+1);

            }

            // not found done, just return

}
```

1.6 Reference

Wikipedia:

http://en.wikipedia.org/wiki/K-d_tree