# FLANN - Fast Library for Approximate Nearest Neighbors

## User Manual

Marius Muja, mariusm@cs.ubc.ca
David Lowe, lowe@cs.ubc.ca

October 13, 2026

# 1 Introduction

We can define the *nearest neighbor search (NSS)* problem in the following way: given a set of points $P = p_1, p_2, \ldots, p_n$ in a metric space $X$, these points must be preprocessed in such a way that given a new query point $q \in X$, finding the point in $P$ that is nearest to $q$ can be done quickly.

The problem of nearest neighbor search is one of major importance in a variety of applications such as image recognition, data compression, pattern recognition and classification, machine learning, document retrieval systems, statistics and data analysis. However, solving this problem in high dimensional spaces seems to be a very difficult task and there is no algorithm that performs significantly better than the standard brute-force search. This has lead to an increasing interest in a class of algorithms that perform approximate nearest neighbor searches, which have proven to be a good-enough approximation in most practical applications and in most cases, orders of magnitude faster that the algorithms performing the exact searches.

FLANN (Fast Library for Approximate Nearest Neighbors) is a library for performing fast approximate nearest neighbor searches. FLANN is written in the C++ programming language. FLANN can be easily used in many contexts through the C, MATLAB, Python, and Ruby bindings provided with the library.

## 1.1 Quick Start

This section contains small examples of how to use the FLANN library from different programming languages (C++, C, MATLAB, Python, and Ruby).

- **C++**

```
// file flann_example.cpp

#include <flann/flann.hpp>
#include <flann/io/hdf5.h>

#include <stdio.h>

int main(int argc, char** argv)
{
    int nn = 3;

    flann::Matrix<float> dataset;
    flann::Matrix<float> query;
    flann::load_from_file(dataset, "dataset.hdf5","dataset");
    flann::load_from_file(query, "dataset.hdf5","query");

    flann::Matrix<int> indices(new int[query.rows*nn], query.rows, nn);
    flann::Matrix<float> dists(new float[query.rows*nn], query.rows, nn);

    // construct an randomized kd-tree index using 4 kd-trees
    flann::Index<flann::L2<float> > index(dataset, flann::KDTreeIndexParams(4));
    index.buildIndex();

    // do a knn search, using 128 checks
    index.knnSearch(query, indices, dists, nn, flann::SearchParams(128));

    flann::save_to_file(indices,"result.hdf5","result");
```

```
    delete[] dataset.ptr();
    delete[] query.ptr();
    delete[] indices.ptr();
    delete[] dists.ptr();

    return 0;
}
```

- **C**

```
/* file flann_example.c */

#include "flann.h"
#include <stdio.h>
#include <assert.h>

/* Function that reads a dataset */
float* read_points(char* filename, int *rows, int *cols);

int main(int argc, char** argv)
{
   int rows,cols;
   int t_rows, t_cols;
   float speedup;

   /* read dataset points from file dataset.dat */
   float* dataset = read_points("dataset.dat", &rows, &cols);
   float* testset = read_points("testset.dat", &t_rows, &t_cols);

   /* points in dataset and testset should have the same dimensionality */
   assert(cols==t_cols);

   /* number of nearest neighbors to search */
   int nn = 3;
   /* allocate memory for the nearest-neighbors indices */
   int* result = (int*) malloc(t_rows*nn*sizeof(int));
   /* allocate memory for the distances */
   float* dists = (float*) malloc(t_rows*nn*sizeof(float));

   /* index parameters are stored here */
   struct FLANNParameters p = DEFAULT_FLANN_PARAMETERS;
   p.algorithm = FLANN_INDEX_AUTOTUNED;  /* or FLANN_INDEX_KDTREE, FLANN_INDEX_KMEANS, ... /*
   p.target_precision = 0.9;  /* want 90% target precision */

   /* compute the 3 nearest-neighbors of each point in the testset */
   flann_find_nearest_neighbors(dataset, rows, cols, testset, t_rows,
result, dists, nn, &p);

   ...
   free(dataset);
   free(testset);
   free(result);
   free(dists);

   return 0;
}
```

- **MATLAB**

```
% create random dataset and test set
dataset = single(rand(128,10000));
testset = single(rand(128,1000));

% define index and search parameters
```

```
params.algorithm = 'kdtree';
params.trees = 8;
params.checks = 64;

% perform the nearest-neighbor search
[result, dists] = flann_search(dataset,testset,5,params);
```

- **Python**

```
from pyflann import *
from numpy import *
from numpy.random import *

dataset = rand(10000, 128)
testset = rand(1000, 128)

flann = FLANN()
result,dists = flann.nn(dataset,testset,5,algorithm="kmeans",
               branching=32, iterations=7, checks=16);
```

- **Ruby**

```
require 'flann' # also requires NMatrix

dataset = NMatrix.random([10000,128])
testset = NMatrix.random([1000,128])

index   = Flann::Index.new(dataset) do |params|
  params[:algorithm]  = :kmeans
  params[:branching]  = 32
  params[:iterations] = 7
  params[:checks]     = 16
end
speedup = index.build! # this is optional

results, distances = index.nearest_neighbors(testset, 5)

index.save "my_index.save"

# Alternatively, without an index:
results, distances = Flann.nearest_neighbors(dataset, testset, 5,
                     algorithm: :kmeans, branching: 32,
                     iterations: 7, checks: 16)
```

# 2 Downloading and compiling FLANN

FLANN can be downloaded from the following address:

`http://www.cs.ubc.ca/~mariusm/flann`

After downloading and unpacking, the following files and directories should be present:

- `bin`: directory various for scripts and binary files
- `doc`: directory containg this documentation
- `examples`: directory containg examples of using FLANN

- `src`: directory containg the source files
- `test`: directory containg unit tests for FLANN

To compile the FLANN library the *CMake*[1] build system is required. Below is an example of how FLANN can be compiled on Linux (replace x.y.z with the corresponding version number).

```
$ cd flann-x.y.z-src
$ mkdir build
$ cd build
$ cmake ..
$ make
```

On windows the steps are very similar:

```
> "C:\Program Files\Microsoft Visual Studio 9.0\VC\vcvarsall.bat"
> cd flann-x.y.z-src
> mkdir build
> cd build
> cmake ..
> nmake
```

There are several compile options that can be configured before FLANN is compiled, for example the build type (Release, RelWithDebInfo, Debug) or whether to compile the C, Python or the MATLAB bindings. To change any of this options use the `cmake-gui` application after `cmake` has finished (see figure 1).

```
> cmake-gui .
```

## 2.1 Upgrading from a previous version

This section contains changes that you need to be aware of when upgrading from a previous version of FLANN.

**Version 1.8.0** Due to changes in the library, the on-disk format of the saved indexes has changed and it is not possible to load indexes previously saved with an older version of the library.

**Version 1.7.0** A small breaking API change in `flann::Matrix` requires client code to be updated. In order to release the memory used by a matrix, use:

```
delete[] matrix.ptr();
```

instead of:

```
delete[] matrix.data;
```

[1] http://www.cmake.org/

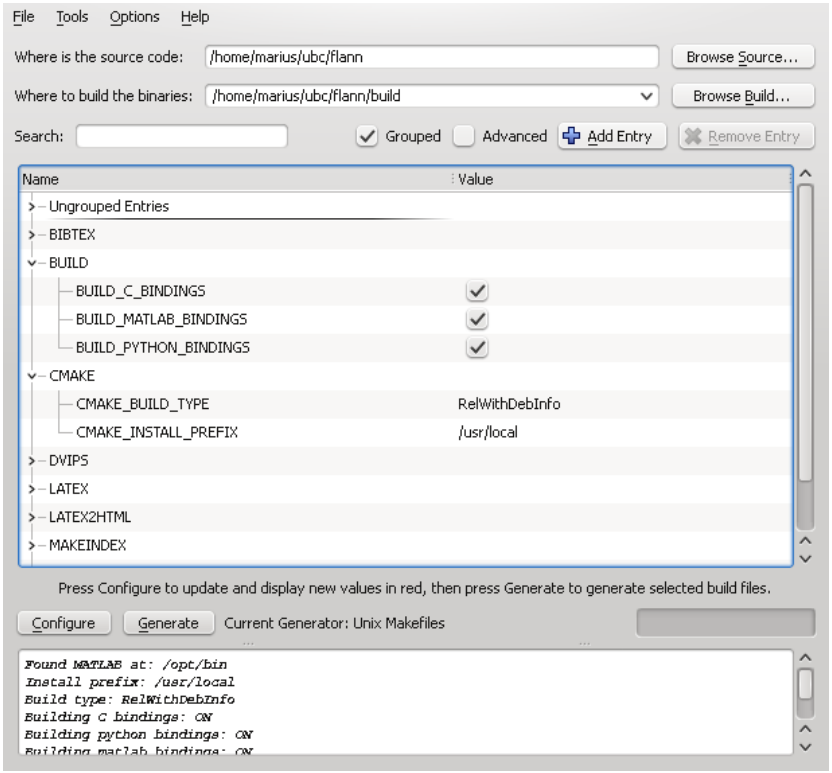



Figure 1: Configuring the FLANN compile options

or:

```
matrix.free();
```

The member `data` of `flann::Matrix` is not publicly accessible any more, use the `ptr()` method to obtain the pointer to the memory buffer.

## 2.2 Compiling FLANN with multithreading support

For taking advantage of multithreaded search, the project that uses FLANN needs to be compiled with a compiler that supports the OpenMP standard and the OpenMP support must be enabled. The number of cores to be used can be selected with the `cores` field in the `SearchParams` structure. By default a single core will be used. Setting the `cores` field to zero will automatically use as many threads as cores available on the machine.

# 3 Using FLANN

## 3.1 Using FLANN from C++

The core of the FLANN library is written in C++. To make use of the full power and flexibility of the templated code one should use the C++ bindings if possible. To use the C++ bindings you only need to include the the library header file `flann.hpp`. An example of the compile command that must be used will look something like this:

```
g++ flann_example.cpp -I $FLANN_ROOT/include -o flann_example_cpp
```

where `$FLANN_ROOT` is the library main directory.

The following sections describe the public C++ API.

### 3.1.1 flann::Index

The FLANN nearest neighbor index class. This class is used to abstract different types of nearest neighbor search indexes. The class is templated on the distance functor to be used for computing distances between pairs of features.

```
namespace flann
{
    template<typename Distance>
    class Index
    {
        typedef typename Distance::ElementType ElementType;
        typedef typename Distance::ResultType DistanceType;
    public:
        Index(const IndexParams& params, Distance distance = Distance() );

        Index(const Matrix<ElementType>& points, const IndexParams& params,
                Distance distance = Distance() );

        ~Index();

        void buildIndex();

        void buildIndex(const Matrix<ElementType>& points);

        void addPoints(const Matrix<ElementType>& points,
                       float rebuild_threshold = 2);

        void removePoint(size_t point_id);

        ElementType* getPoint(size_t point_id);

        int knnSearch(const Matrix<ElementType>& queries,
                      Matrix<int>& indices,
                      Matrix<DistanceType>& dists,
                      size_t knn,
                      const SearchParams& params);

        int knnSearch(const Matrix<ElementType>& queries,
                      std::vector< std::vector<int> >& indices,
```

```
                        std::vector<std::vector<DistanceType> >& dists,
                        size_t knn,
                        const SearchParams& params);

        int radiusSearch(const Matrix<ElementType>& queries,
                         Matrix<int>& indices,
                         Matrix<DistanceType>& dists,
                         float radius,
                         const SearchParams& params);

        int radiusSearch(const Matrix<ElementType>& queries,
                          std::vector< std::vector<int> >& indices,
                          std::vector<std::vector<DistanceType> >& dists,
                          float radius,
                          const SearchParams& params);

        void save(std::string filename);

        int veclen() const;

        int size() const;

        IndexParams getParameters() const;

        flann_algorithm_t getType() const;

    };
}
```

**The Distance functor**

The distance functor is a class whose `operator()` computes the distance between two features. If the distance is also a kd-tree compatible distance it should also provide an `accum_dist()` method that computes the distance between individual feature dimensions. A typical distance functor looks like this (see the `dist.h` file for more examples):

```
template<class T>
struct L2
{
    typedef bool is_kdtree_distance;

    typedef T ElementType;
    typedef typename Accumulator<T>::Type ResultType;

    template <typename Iterator1, typename Iterator2>
    ResultType operator()(Iterator1 a, Iterator2 b, size_t size,
                          ResultType /*worst_dist*/ = -1) const
    {
        ResultType result = ResultType();
        ResultType diff;
        for(size_t i = 0; i < size; ++i ) {
            diff = *a++ - *b++;
            result += diff*diff;
        }
        return result;
```

```
    }

    template <typename U, typename V>
    inline ResultType accum_dist(const U& a, const V& b, int) const
    {
        return (a-b)*(a-b);
    }
};
```

In addition to `operator()` and `accum_dist()`, a distance functor should also define the `ElementType` and the `ResultType` as the types of the elements it operates on and the type of the result it computes.

If a distance functor can be used as a kd-tree distance (meaning that the full distance between a pair of features can be accumulated from the partial distances between the individual dimensions) a typedef `is_kdtree_distance` should be present inside the distance functor. If the distance is not a kd-tree distance, but it's a distance in a vector space (the individual dimensions of the elements it operates on can be accessed independently) a typedef `is_vector_space_distance` should be defined inside the functor. If neither typedef is defined, the distance is assumed to be a metric distance and will only be used with indexes operating on generic metric distances.

**flann::Index::Index** Constructs a nearest neighbor search index for a given dataset.

```
Index(const IndexParams& params, Distance distance = Distance() );

Index(const Matrix<ElementType>& points, const IndexParams& params,
        Distance distance = Distance() );
```

**points** Matrix containing the features(points) that should be indexed, stored in a row-major order (one point on each row of the matrix). The size of the matrix is $num_features \times dimensionality$.

**params** Structure containing the index parameters. The type of index that will be constructed depends on the type of this parameter. The possible parameter types are:

**LinearIndexParams** When passing an object of this type, the index will perform a linear, brute-force search.

```
struct LinearIndexParams : public IndexParams
{
};
```

**KDTreeIndexParams** When passing an object of this type the index constructed will consist of a set of randomized kd-trees which will be searched in parallel.

```
struct KDTreeIndexParams : public IndexParams
{
```

```
      KDTreeIndexParams( int trees = 4 );
};
```

**trees** The number of parallel kd-trees to use. Good values are in the range [1..16]

**KMeansIndexParams** When passing an object of this type the index constructed will be a hierarchical k-means tree.

```
struct KMeansIndexParams : public IndexParams
{
        KMeansIndexParams( int branching = 32,
                        int iterations = 11,
                        flann_centers_init_t centers_init = FLANN_CENTERS_RANDOM,
                        float cb_index = 0.2 );
};
```

**branching** The branching factor to use for the hierarchical k-means tree

**iterations** The maximum number of iterations to use in the k-means clustering stage when building the k-means tree. If a value of -1 is used here, it means that the k-means clustering should be iterated until convergence

**centers_init** The algorithm to use for selecting the initial centers when performing a k-means clustering step. The possible values are CENTERS_RANDOM (picks the initial cluster centers randomly), CENTERS_GONZALES (picks the initial centers using Gonzales' algorithm) and CENTERS_KMEANSPP (picks the initial centers using the algorithm suggested in [AV07])

**cb_index** This parameter (cluster boundary index) influences the way exploration is performed in the hierarchical kmeans tree. When `cb_index` is zero the next kmeans domain to be explored is choosen to be the one with the closest center. A value greater then zero also takes into account the size of the domain.

**CompositeIndexParams** When using a parameters object of this type the index created combines the randomized kd-trees and the hierarchical k-means tree.

```
struct CompositeIndexParams : public IndexParams
{
        CompositeIndexParams( int trees = 4,
                           int branching = 32,
                           int iterations = 11,
                           flann_centers_init_t centers_init = FLANN_CENTERS_RANDOM,
                           float cb_index = 0.2 );
};
```

**KDTreeSingleIndexParams** When passing an object of this type the index will contain a single kd-tree optimized for searching lower dimensionality data (for example 3D point clouds)

```
struct KDTreeSingleIndexParams : public IndexParams
{
      KDTreeSingleIndexParams( int leaf_max_size = 10 );
};
```

**max_leaf_size** The maximum number of points to have in a leaf for not branching the tree any more.

**KDTreeCuda3dIndexParams** When passing an object of this type the index will be a single kd-tree that is built and performs searches on a CUDA compatible GPU. Search performance is best for large numbers of search and query points. For more information, see section 3.1.10

```
struct KDTreeCuda3dIndexParams : public IndexParams
{
    KDTreeCuda3dIndexParams( int leaf_max_size = 64 );
};
```

**max_leaf_size** The maximum number of points to have in a leaf for not branching the tree any more.

**HierarchicalClusteringIndexParams** When passing an object of this type the index constructed will be a hierarchical clustering index. This type of index works with any metric distance and can be used for matching binary features using Hamming distances.

```
struct HierarchicalClusteringIndexParams : public IndexParams
{
    HierarchicalClusteringIndexParams(int branching = 32,
                                flann_centers_init_t centers_init = FLANN_CENTERS_RANDOM,
                                int trees = 4, int leaf_max_size = 100)
};
```

**branching** The branching factor to use for the hierarchical clustering tree

**centers_init** The algorithm to use for selecting the initial centers when performing a k-means clustering step. The possible values are CENTERS_RANDOM (picks the initial cluster centers randomly), CENTERS_GONZALES (picks the initial centers using Gonzales' algorithm) and CENTERS_KMEANSPP (picks the initial centers using the algorithm suggested in [AV07])

**trees** The number of parallel trees to use. Good values are in the range [3..8]

**leaf_size** The maximum number of points a leaf node should contain.

**LshIndexParams** When passing an object of this type the index constructed will be a multi-probe LSH (Locality-Sensitive Hashing) index. This type of index can only be used for matching binary features using Hamming distances.

```
struct LshIndexParams : public IndexParams
{
    LshIndexParams(unsigned int table_number = 12,
                   unsigned int key_size = 20,
                   unsigned int multi_probe_level = 2);
};
```

**table_number** The number of hash tables to use

**key_size** The length of the key in the hash tables

**multi_probe_level** Number of levels to use in multi-probe (0 for standard LSH)

**AutotunedIndexParams** When passing an object of this type the index created is automatically tuned to offer the best performance, by choosing the optimal index type (randomized kd-trees, hierarchical kmeans, linear) and parameters for the dataset provided.

```
struct AutotunedIndexParams : public IndexParams
{
        AutotunedIndexParams( float target_precision = 0.9,
                              float build_weight = 0.01,
                              float memory_weight = 0,
                              float sample_fraction = 0.1 );
};
```

**target_precision** Is a number between 0 and 1 specifying the percentage of the approximate nearest-neighbor searches that return the exact nearest-neighbor. Using a higher value for this parameter gives more accurate results, but the search takes longer. The optimum value usually depends on the application.

**build_weight** Specifies the importance of the index build time raported to the nearest-neighbor search time. In some applications it's acceptable for the index build step to take a long time if the subsequent searches in the index can be performed very fast. In other applications it's required that the index be build as fast as possible even if that leads to slightly longer search times.

**memory_weight** Is used to specify the tradeoff between time (index build time and search time) and memory used by the index. A value less than 1 gives more importance to the time spent and a value greater than 1 gives more importance to the memory usage.

**sample_fraction** Is a number between 0 and 1 indicating what fraction of the dataset to use in the automatic parameter configuration algorithm. Running the algorithm on the full dataset gives the most accurate results, but for very large datasets can take longer than desired. In such case using just a fraction of the data helps speeding up this algorithm while still giving good approximations of the optimum parameters.

**SavedIndexParams** This object type is used for loading a previously saved index from the disk.

```
struct SavedIndexParams : public IndexParams
{
      SavedIndexParams( std::string filename );
};
```

**filename** The filename in which the index was saved.

### 3.1.2 flann::Index::buildIndex

Builds the nearest neighbor search index. There are two versions of the `buildIndex` method, one that uses the points provided as argument and one that uses the points provided to the constructor when the object was constructed.

```
void buildIndex();

void buildIndex(const Matrix<ElementType>& points);
```

### 3.1.3 flann::Index::addPoints

The `addPoints` method can be used to incrementally add points to the index after the index was build. To avoid the index getting unbalanced, the `addPoints` method has the option of rebuilding the index after a large number of points have been added. The `rebuild_threshold` parameter controls when the index is rebuild, by default this is done when it doubles in size (`rebuild_threshold` = 2).

```
void addPoints(const Matrix<ElementType>& points, float rebuild_threshold = 2);
```

### 3.1.4 flann::Index::removePoint

The `removePoint` method removes one point with the specified `point_id` from the index. The indices (of the remaining points) returned by the nearest neighbor operations do not change when points are removed from the index.

```
void removePoint(size_t point_id);
```

### 3.1.5 flann::Index::getPoint

The `getPoint` method returns a pointer to the data point with the specified `point_id`.

```
ElementType* getPoint(size_t point_id);
```

### 3.1.6 flann::Index::knnSearch

Performs a K-nearest neighbor search for a set of query points. There are two signatures for this method, one that takes pre-allocated `flann::Matrix` objects for returning the indices of and distances to the neighbors found, and one that takes `std::vector<std::vector>` that will re resized automatically as needed.

```
int Index::knnSearch(const Matrix<ElementType>& queries,
                Matrix<int>& indices,
                Matrix<DistanceType>& dists,
                size_t knn,
                const SearchParams& params);

int Index::knnSearch(const Matrix<ElementType>& queries,
                std::vector< std::vector<int> >& indices,
                std::vector<std::vector<DistanceType> >& dists,
                size_t knn,
                const SearchParams& params);
```

**queries** Matrix containing the query points. Size of matrix is (*num_queries* × *dimentionality*)

**indices** Matrix that will contain the indices of the K-nearest neighbors found (size should be at least *num_queries* × *knn* for the pre-allocated version).

**dists** Matrix that will contain the distances to the K-nearest neighbors found (size should be at least *num_queries* × *knn* for the pre-allocated version). *Note:* For Euclidean distances, the `flann::L2` functor computes squared distances, so the value passed here needs to be a squared distance as well.

**knn** Number of nearest neighbors to search for.

**params** Search parameters. Structure containing parameters used during search.

**SearchParameters**

```
struct SearchParams
{
        SearchParams(int checks = 32,
                float eps = 0,
                bool sorted = true);

        int checks;
        float eps;
        bool sorted;
        int max_neighbors;
        tri_type use_heap;
        int cores;
        bool matrices_in_gpu_ram;
};
```

**checks** specifies the maximum leafs to visit when searching for neighbours. A higher value for this parameter would give better search precision, but also take more time. For all leafs to be checked use

the value `FLANN_CHECKS_UNLIMITED`. If automatic configuration was used when the index was created, the number of checks required to achieve the specified precision was also computed, to use that value specify `FLANN_CHECKS_AUTOTUNED`.

**eps** Search for eps-approximate neighbors (only used by KDTreeSingleIndex and KDTreeCuda3dIndex).

**sorted** Used only by radius search, specifies if the neighbors returned should be sorted by distance.

**max_neighbours** Specifies the maximum number of neighbors radius search should return (default: -1 = unlimited). Only used for radius search.

**use_heap** Use a heap data structure to manage the list of neighbors internally (possible values: FLANN_False, FLANN_True, FLANN_Undefined). A heap is more efficient for a large number of neighbors and less efficient for a small number of neighbors. Default value is FLANN_Undefined, which lets the code choose the best option depending on the number of neighbors requested. Only used for KNN search.

**cores** How many cores to assign to the search (specify 0 for automatic core selection).

**matrices_in_gpu_ram** for GPU search indicates if matrices are already in GPU ram.

### 3.1.7 flann::Index::radiusSearch

Performs a radius nearest neighbor search for a set of query points. There are two signatures for this method, one that takes pre-allocated `flann::Matrix` objects for returning the indices of and distances to the neighbors found, and one that takes `std::vector<std::vector>` that will resized automatically as needed.

```
int Index::radiusSearch(const Matrix<ElementType>& queries,
                  Matrix<int>& indices,
                  Matrix<DistanceType>& dists,
                  float radius,
                  const SearchParams& params);

int Index::radiusSearch(const Matrix<ElementType>& queries,
                  std::vector< std::vector<int> >& indices,
                  std::vector<std::vector<DistanceType> >& dists,
                  float radius,
                  const SearchParams& params);
```

**queries** The query point. Size of matrix is ($num_queries \times dimentionality$).

**indices** Matrix that will contain the indices of the K-nearest neighbors found. For the pre-allocated version, only as many neighbors are returned as many columns in this matrix. If fewer neighbors are found than columns

in this matrix, the element after that last index returned is -1. In case of the std::vector version, the rows will be resized as needed to fit all the neighbors to be returned, except if the "max_neighbors" search parameter is set.

**dists** Matrix that will contain the distances to the K-nearest neighbors found. The same number of values are returned here as for the `indices` matrix. *Note:* For Euclidean distances, the `flann::L2` functor computes squared distances, so the value passed here needs to be a squared distance as well.

**radius** The search radius.
*Note:* For Euclidean distances, the `flann::L2` functor computes squared distances, so the value passed here needs to be a squared distance as well.

**params** Search parameters.

The method returns the number of nearest neighbors found.

#### 3.1.8 flann::Index::save

Saves the index to a file.

```
void Index::save(std::string filename);
```

**filename** The file to save the index to

#### 3.1.9 flann::hierarchicalClustering

Clusters the given points by constructing a hierarchical k-means tree and choosing a cut in the tree that minimizes the clusters' variance.

```
template <typename Distance>
int hierarchicalClustering(const Matrix<typename Distance::ElementType>& points,
        Matrix<typename Distance::ResultType>& centers,
        const KMeansIndexParams& params,
        Distance d = Distance())
```

**features** The points to be clustered

**centers** The centers of the clusters obtained. The number of rows in this matrix represents the number of clusters desired. However, because of the way the cut in the hierarchical tree is choosen, the number of clusters computed will be the highest number of the form $(branching - 1) * k + 1$ that's lower than the number of clusters desired, where *branching* is the tree's branching factor (see description of the KMeansIndexParams).

**params** Parameters used in the construction of the hierarchical k-means tree

The function returns the number of clusters computed.

### 3.1.10 flann::KdTreeCuda3dIndex

FLANN provides a CUDA implementation of the kd-tree build and search algorithms to improve the build and query speed for large 3d data sets. This section will provide all the necessary information to use the `KdTreeCuda3dIndex` index type.

**Building:** If CMake detects a CUDA install during the build (see section 2), a library `libflann_cuda.so` will be built.

**Basic Usage:** To be able to use the new index type, you have to include the FLANN header this way:

```
#define FLANN_USE_CUDA
#include <flann/flann.hpp>
```

If you define the symbol `FLANN_USE_CUDA` before including the FLANN header, you will have to link `libflann_cuda.so` or `libflann_cuda_s.a` with your project. However, you will not have to compile your source code with `nvcc`, except if you use other CUDA code, of course.

You can then create your index by using the `KDTreeCuda3dIndexParams` to create the index. The index will take care of copying all the data from and to the GPU for you, both for index creation and search.

A word of caution: A general guideline for deciding whether to use the CUDA kd tree or a (multi-threaded) CPU implementation is hard to give, since it depends on the combination of CPU and GPU in each system and the data sets. For example, on a system with a Phenom II 955 CPU and a Geforce GTX 260 GPU, the maximum search speedup on a synthetic (random) data set is a factor of about 8-9 vs the single-core CPU search, and starts to be reached at about 100k search and query points. (This includes transfer times.) Build time does not profit as much from the GPU acceleration; here the benefit is about 2x at 200k points, but this largely depends on the data set. The only way to know which implementation is best suited is to try it.

**Advanced Usage:** In some cases, you might already have your data in a buffer on the GPU. In this case, you can re-use these buffers instead of copying the buffers back to system RAM for index creation and search. The way to do this is to pass GPU pointers via the `flann::Matrix` inputs and tell the index via the index and search params to treat the pointers as GPU pointers.

```
thrust::device_vector<float4>  points_vector( n_points );
// ... fill vector here...

float* gpu_pointer = (float*)thrust::raw_pointer_cast(&points_vector[0]);
flann::Matrix<float> matrix_gpu(gpu_pointer,n_points,3, 4);

flann::KDTreeCuda3dIndexParams params;
params["input_is_gpu_float4"]=true;
flann::Index<flann::L2<float> > flannindex( matrix_gpu, params  );
flannindex.buildIndex();
```

First, a GPU buffer of float4 values is created and filled with points. [2]

Then, a GPU pointer to the buffer is stored in a flann matrix with 3 columns and a stride of 4 (the last element in the `float4` should be zero).

Last, the index is created. The `input_is_gpu_float4` flag tells the index to treat the matrix as a gpu buffer.

Similarly, you can specify GPU buffers for the search routines that return the result via flann matrices (but not for those that return them via `std::vectors`). To do this, the pointers in the index and dist matrices have to point to GPU buffers and the `cols` value has to be set to 3 (as we are dealing with 3d points). Here, any value for `stride` can be used.

```
flann::Matrix<int> indices_gpu(gpu_pointer_indices,n_points, knn, stride);
flann::Matrix<float> dists_gpu(gpu_pointer_dists,n_points, knn, stride);

flann::SearchParams params;
params.matrices_in_gpu_ram = true;

flannindex.knnSearch( queries_gpu ,indices_gpu,dists_gpu,knn, params);
```

Note that you cannot mix matrices in system and CPU ram here!

**Search Parameters:** The search routines support three parameters:

- `eps` - used for approximate knn search. The maximum possible error is $e = d_{best} * eps$, i.e. the distance of the returned neighbor is at maximum $eps$ times larget than the distance to the real best neighbor.
- `use_heap` - used in knn and radius search. If set to true, a heap structure will be used in the search to keep track of the distance to the farthest neighbor. Beneficial with about $k > 64$ elements.
- `sorted` - if set to true, the results of the radius and knn searches will be sorted in ascending order by their distance to the query point.
- `matrices_in_gpu_ram` - set to true to indicate that all (input and output) matrices are located in GPU RAM.

## 3.2 Using FLANN from C

FLANN can be used in C programs through the C bindings provided with the library. Because there is no template support in C, there are bindings provided for the following data types: `unsigned char`, `int`, `float` and `double`. For each of the functions below there is a corresponding version for each of the for data types, for example for the function:

[2] For index creation, only `float4` points are supported, `float3` or structure-of-array (SOA) representations are currently not supported since `float4` proved to be best in terms of access speed for tree creation and search.

```
flan_index_t flann_build_index(float* dataset, int rows, int cols, float* speedup,
        struct FLANNParameters* flann_params);
```

there are also the following versions:

```
flan_index_t flann_build_index_byte(unsigned char* dataset,
        int rows, int cols, float* speedup,
        struct FLANNParameters* flann_params);
flan_index_t flann_build_index_int(int* dataset,
        int rows, int cols, float* speedup,
        struct FLANNParameters* flann_params);
flan_index_t flann_build_index_float(float* dataset,
        int rows, int cols, float* speedup,
        struct FLANNParameters* flann_params);
flan_index_t flann_build_index_double(double* dataset,
        int rows, int cols, float* speedup,
        struct FLANNParameters* flann_params);
```

### 3.2.1 flann_build_index()

```
flan_index_t flann_build_index(float* dataset,
        int rows,
        int cols,
        float* speedup,
        struct FLANNParameters* flann_params);
```

This function builds an index and return a reference to it. The arguments expected by this function are as follows:

**dataset, rows and cols** - are used to specify the input dataset of points: dataset is a pointer to a rows × cols matrix stored in row-major order (one feature on each row)

**speedup** - is used to return the approximate speedup over linear search achieved when using the automatic index and parameter configuration (see section 3.3.1)

**flann_params** - is a structure containing the parameters passed to the function. This structure is defined as follows:

```
struct FLANNParameters
{
        enum flann_algorithm_t algorithm; /* the algorithm to use */

        /* search time parameters */
        int checks; /* how many leafs (features) to check in one search */
        float eps; /* eps parameter for eps-knn search */
        int sorted; /* indicates if results returned by radius search should be
                sorted or not */
        int max_neighbors; /* limits the maximum number of neighbors should be
                returned by radius search */
        int cores; /* number of paralel cores to use for searching */
```

```
        /*  kdtree index parameters */
        int trees; /* number of randomized trees to use (for kdtree) */
        int leaf_max_size;

        /* kmeans index parameters */
        int branching; /* branching factor (for kmeans tree) */
        int iterations; /* max iterations to perform in one kmeans cluetering
                (kmeans tree) */
        enum flann_centers_init_t centers_init; /* algorithm used for picking the
                initial cluster centers for kmeans tree */
        float cb_index; /* cluster boundary index. Used when searching the kmeans
                tree */

        /* autotuned index parameters */
        float target_precision; /* precision desired (used for autotuning, -1
                otherwise) */
        float build_weight; /* build tree time weighting factor */
        float memory_weight; /* index memory weigthing factor */
        float sample_fraction; /* what fraction of the dataset to use for autotuning */
        /* LSH parameters */
        unsigned int table_number_; /** The number of hash tables to use */
        unsigned int key_size_; /** The length of the key in the hash tables */
        unsigned int multi_probe_level_; /** Number of levels to use in multi-probe
                LSH, 0 for standard LSH */

        /* other parameters */
        enum flann_log_level_t log_level; /* determines the verbosity of each flann
                function */
        long random_seed; /* random seed to use */
};
```

The `algorithm` and `centers_init` fields can take the following values:

```
enum flann_algorithm_t
{
        FLANN_INDEX_LINEAR = 0,
        FLANN_INDEX_KDTREE = 1,
        FLANN_INDEX_KMEANS = 2,
        FLANN_INDEX_COMPOSITE = 3,
        FLANN_INDEX_KDTREE_SINGLE = 4,
        FLANN_INDEX_HIERARCHICAL = 5,
        FLANN_INDEX_LSH = 6,
        FLANN_INDEX_KDTREE_CUDA = 7, // available if compiled with CUDA
        FLANN_INDEX_SAVED = 254,
        FLANN_INDEX_AUTOTUNED = 255,
};

enum flann_centers_init_t {
        FLANN_CENTERS_RANDOM = 0,
        FLANN_CENTERS_GONZALES = 1,
        FLANN_CENTERS_KMEANSPP = 2
};
```

The `algorithm` field is used to manually select the type of index used. The `centers_init` field specifies how to choose the inital cluster centers when performing the hierarchical k-means clustering (in case the

algorithm used is k-means): `FLANN_CENTERS_RANDOM` chooses the initial centers randomly, `FLANN_CENTERS_GONZALES` chooses the initial centers to be spaced apart from each other by using Gonzales' algorithm and `FLANN_CENTERS_KMEANSPP` chooses the initial centers using the algorithm proposed in [AV07].

The fields: `checks`, `cb_index`, `trees`, `branching`, `iterations`, `target_precision`, `build_weight`, `memory_weight` and `sample_fraction` have the same meaning as described in 3.1.1.

The `random_seed` field contains the random seed useed to initialize the random number generator.

The field `log_level` controls the verbosity of the messages generated by the FLANN library functions. It can take the following values:

```
enum flann_log_level_t
{
        FLANN_LOG_NONE = 0,
        FLANN_LOG_FATAL = 1,
        FLANN_LOG_ERROR = 2,
        FLANN_LOG_WARN = 3,
        FLANN_LOG_INFO = 4,
        FLANN_LOG_DEBUG = 5
};
```

### 3.2.2 flann_find_nearest_neighbors_index()

```
int flann_find_nearest_neighbors_index(flann_index_t index_id,
        float* testset,
        int trows,
        int* indices,
        float* dists,
        int nn,
        struct FLANNParameters* flann_params);
```

This function searches for the nearest neighbors of the `testset` points using an index already build and referenced by `index_id`. The `testset` is a matrix stored in row-major format with `trows` rows and the same number of columns as the dimensionality of the points used to build the index. The function computes `nn` nearest neighbors for each point in the `testset` and stores them in the `indices` matrix (which is a trows $\times$ nn matrix stored in row-major format). The memory for the `result` matrix must be allocated before the `flann_find_nearest_neighbors_index()` function is called. The distances to the nearest neighbors found are stored in the `dists` matrix.

### 3.2.3 flann_find_nearest_neighbors()

```
int flann_find_nearest_neighbors(float* dataset,
        int rows,
        int cols,
        float* testset,
```

```
        int trows,
        int* indices,
        float* dists,
        int nn,
        struct FLANNParameters* flann_params);
```

This function is similar to the `flann_find_nearest_neighbors_index()` function, but instread of using a previously constructed index, it constructs the index, does the nearest neighbor search and deletes the index in one step.

### 3.2.4 flann_radius_search()

```
int flann_radius_search(flann_index_t index_ptr, /* the index */
        float* query, /* query point */
        int* indices, /* array for storing the indices found (will be modified) */
        float* dists, /* similar, but for storing distances */
        int max_nn,  /* size of arrays indices and dists */
        float radius, /* search radius (squared radius for euclidian metric) */
        struct FLANNParameters* flann_params);
```

This function performs a radius search to single query point. The indices of the neighbors found and the distances to them are stored in the `indices` and dists `arrays`. The `max_nn` parameter sets the limit of the neighbors that will be returned (the size of the `indices` and `dists` arrays must be at least `max_nn`).

### 3.2.5 flann_save_index()

```
int flann_save_index(flann_index_t index_id,
        char* filename);
```

This function saves an index to a file. The dataset for which the index was built is not saved with the index.

### 3.2.6 flann_load_index()

```
flann_index_t flann_load_index(char* filename,
        float* dataset,
        int rows,
        int cols);
```

This function loads a previously saved index from a file. Since the dataset is not saved with the index, it must be provided to this function.

### 3.2.7 flann_free_index()

```
int flann_free_index(flann_index_t index_id,
        struct FLANNParameters* flann_params);
```

This function deletes a previously constructed index and frees all the memory used by it.

### 3.2.8 flann_set_distance_type

This function chooses the distance function to use when computing distances between data points.

```
void flann_set_distance_type(enum flann_distance_t distance_type, int order);
```

**distance_type** The distance type to use. Possible values are

```
enum flann_distance_t
{
        FLANN_DIST_EUCLIDEAN = 1,
        FLANN_DIST_L2 = 1,
        FLANN_DIST_MANHATTAN = 2,
        FLANN_DIST_L1 = 2,
        FLANN_DIST_MINKOWSKI = 3,
        FLANN_DIST_MAX = 4,
        FLANN_DIST_HIST_INTERSECT = 5,
        FLANN_DIST_HELLINGER = 6,
        FLANN_DIST_CHI_SQUARE = 7,
        FLANN_DIST_KULLBACK_LEIBLER = 8,
        FLANN_DIST_HAMMING = 9,
        FLANN_DIST_HAMMING_LUT = 10,
        FLANN_DIST_HAMMING_POPCNT = 11,
        FLANN_DIST_L2_SIMPLE = 12,
};
```

**order** Used in for the `FLANN_DIST_MINKOWSKI` distance type, to choose the order of the Minkowski distance.

### 3.2.9 flann_compute_cluster_centers()

Performs hierarchical clustering of a set of points (see 3.1.9).

```
int flann_compute_cluster_centers(float* dataset,
      int rows,
      int cols,
      int clusters,
      float* result,
      struct FLANNParameters* flann_params);
```

See section 1.1 for an example of how to use the C/C++ bindings.

## 3.3 Using FLANN from MATLAB

The FLANN library can be used from MATLAB through the following wrapper functions: `flann_build_index`, `flann_search`, `flann_save_index`, `flann_load_index`, `flann_set_distance_type` and `flann_free_index`. The `flann_build_index` function creates a search index from the dataset points, `flann_search` uses this index to perform nearest-neighbor searches, `flann_save_index` and `flann_load_index` can be used to save and load an index to/from disk, `flann_set_distance_type` is

used to set the distance type to be used when building an index and `flann_free_index` deletes the index and releases the memory it uses.

The following sections describe in more detail the FLANN matlab wrapper functions and show examples of how they may be used.

### 3.3.1 flann_build_index

This function creates a search index from the initial dataset of points, index used later for fast nearest-neighbor searches in the dataset.

```
[index, parameters, speedup] = flann_build_index(dataset, build_params);
```

The arguments passed to the `flann_build_index` function have the following meaning:

`dataset` is a $d \times n$ matrix containing $n$ $d$-dimensional points, stored in a column-major order (one feature on each column)

`build_params` - is a MATLAB structure containing the parameters passed to the function.

The `build_params` is used to specify the type of index to be built and the index parameters. These have a big impact on the performance of the new search index (nearest-neighbor search time) and on the time and memory required to build the index. The optimum parameter values depend on the dataset characteristics (number of dimensions, distribution of points in the dataset) and on the application domain (desired precision for the approximate nearest neighbor searches). The `build_params` argument is a structure that contains one or more of the following fields:

`algorithm` - the algorithm to use for building the index. The possible values are: `'linear'`, `'kdtree'`, `'kmeans'`, `'composite'` or `'autotuned'`. The `'linear'` option does not create any index, it uses brute-force search in the original dataset points, `'kdtree'` creates one or more randomized kd-trees, `'kmeans'` creates a hierarchical kmeans clustering tree, `'composite'` is a mix of both kdtree and kmeans trees and the `'autotuned'` automatically determines the best index and optimum parameters using a cross-validation technique.

**Autotuned index:** in case the algorithm field is `'autotuned'`, the following fields should also be present:

`target_precision` - is a number between 0 and 1 specifying the percentage of the approximate nearest-neighbor searches that return the exact nearest-neighbor. Using a higher value for this parameter gives more accurate results, but the search takes longer. The optimum value usually depends on the application.

`build_weight` - specifies the importance of the index build time raported to the nearest-neighbor search time. In some applications it's acceptable for the index build step to take a long time if the subsequent searches in the index can be performed very fast. In other applications it's required that the index be build as fast as possible even if that leads to slightly longer search times. (Default value: 0.01)

`memory_weight` - is used to specify the tradeoff between time (index build time and search time) and memory used by the index. A value less than 1 gives more importance to the time spent and a value greater than 1 gives more importance to the memory usage.

`sample_fraction` - is a number between 0 and 1 indicating what fraction of the dataset to use in the automatic parameter configuration algorithm. Running the algorithm on the full dataset gives the most accurate results, but for very large datasets can take longer than desired. In such case, using just a fraction of the data helps speeding up this algorithm, while still giving good approximations of the optimum parameters.

**Randomized kd-trees index:** in case the algorithm field is `'kdtree'`, the following fields should also be present:

`trees` - the number of randomized kd-trees to create.

**Hierarchical k-means index:** in case the algorithm type is `'means'`, the following fields should also be present:

`branching` - the branching factor to use for the hierarchical kmeans tree creation. While kdtree is always a binary tree, each node in the kmeans tree may have several branches depending on the value of this parameter.

`iterations` - the maximum number of iterations to use in the kmeans clustering stage when building the kmeans tree. A value of -1 used here means that the kmeans clustering should be performed until convergence.

`centers_init` - the algorithm to use for selecting the initial centers when performing a kmeans clustering step. The possible values are 'random' (picks the initial cluster centers randomly), 'gonzales' (picks the initial centers using the Gonzales algorithm) and 'kmeanspp' (picks the initial centers using the algorithm suggested in [AV07]). If this parameters is omitted, the default value is 'random'.

`cb_index` - this parameter (cluster boundary index) influences the way exploration is performed in the hierarchical kmeans tree. When `cb_index` is zero the next kmeans domain to be explored is choosen to be the one with the closest center. A value greater then zero also takes into account the size of the domain.

**Composite index:** in case the algorithm type is `'composite'`, the fields from both randomized kd-tree index and hierarchical k-means index should be present.

The `flann_build_index` function returns the newly created `index`, the `parameters` used for creating the index and, if automatic configuration was used, an estimation of the `speedup` over linear search that is achieved when searching the index. Since the parameter estimation step is costly, it is possible to save the computed parameters and reuse them the next time an index is created from similar data points (coming from the same distribution).

### 3.3.2 flann_search

This function performs nearest-neighbor searches using the index already created:

```
[result, dists] = flann_search(index, testset, k, parameters);
```

The arguments required by this function are:

`index` - the index returned by the `flann_build_index` function

`testset` - a $d \times m$ matrix containing $m$ test points whose k-nearest-neighbors need to be found

`k` - the number of nearest neighbors to be returned for each point from `testset`

`parameters` - structure containing the search parameters. Currently it has only one member, `parameters.checks`, denoting the number of times the tree(s) in the index should be recursively traversed. A higher value for this parameter would give better search precision, but also take more time. If automatic configuration was used when the index was created, the number of checks required to achieve the specified precision is also computed. In such case, the parameters structure returned by the `flann_build_index` function can be passed directly to the `flann_search` function.

The function returns two matrices, each of size $k \times m$. The first one contains, in which each column, the indexes (in the dataset matrix) of the $k$ nearest neighbors of the corresponding point from testset, while the second one contains the corresponding distances. The second matrix can be omitted when making the call if the distances to the nearest neighbors are not needed.

For the case where a single search will be performed with each index, the `flann_search` function accepts the dataset instead of the index as first argument, in which case the index is created searched and then deleted in one step. In this case the parameters structure passed to the `flann_search` function must also contain the fields of the `build_params` structure that would normally be passed to the `flann_build_index` function if the index was build separately.

```
[result, dists] = flann_search(dataset, testset, k, parameters);
```

### 3.3.3 flann_save_index

This function saves an index to a file so that it can be reused at a later time without the need to recompute it. Only the index will be saved to the file, not also the data points for which the index was created, so for the index to be reused the data points must be saved separately.

```
flann_save_index(index, filename)
```

The argumenst required by this function are:

`index` - the index to be saved, created by `flann_build_index`

`filename` - the name of the file in which to save the index

### 3.3.4 flann_load_index

This function loads a previously saved index from a file. It needs to be passed as a second parameter the dataset for which the index was created, as this is not saved together with the index.

```
index = flann_load_index(filename, dataset)
```

The argumenst required by this function are:

`filename` - the file from which to load the index

`dataset` - the dataset for which the index was created

This function returns the index object.

### 3.3.5 flann_set_distance_type

This function chooses the distance function to use when computing distances between data points.

```
flann_set_distance_type(type, order)
```

The argumenst required by this function are:

`type` - the distance type to use. Possible values are: `'euclidean'`, `'manhattan'`, `'minkowski'`, `'max_dist'` ($L_infinity$ - distance type is not valid for kd-tree index type since it's not dimensionwise additive), `'hik'` (histogram intersection kernel), `'hellinger'`,`'cs'` (chi-square) and `'kl'` (Kullback-Leibler).

`order` - only used if distance type is `'minkowski'` and represents the order of the minkowski distance.

### 3.3.6 flann_free_index

This function must be called to delete an index and release all the memory used by it:

```
flann_free_index(index);
```

### 3.3.7 Examples

Let's look at a few examples showing how the functions described above are used:

### 3.3.8 Example 1:

In this example the index is constructed using automatic parameter estimation, requesting 70% as desired precision and using the default values for the build time and memory usage factors. The index is then used to search for the nearest-neighbors of the points in the testset matrix and finally the index is deleted.

```
dataset = single(rand(128,10000));
testset = single(rand(128,1000));

build_params.algorithm = 'autotuned';
build_params.target_precision = 0.7;
build_params.build_weight = 0.01;
build_params.memory_weight = 0;

[index, parameters] = flann_build_index(dataset, build_params);

result = flann_search(index,testset,5,parameters);

flann_free_index(index);
```

### 3.3.9 Example 2:

In this example the index constructed with the parameters specified manually.

```
dataset = single(rand(128,10000));
testset = single(rand(128,1000));

index = flann_build_index(dataset,struct('algorithm','kdtree','trees',8));

result = flann_search(index,testset,5,struct('checks',128));

flann_free_index(index);
```

### 3.3.10 Example 3:

In this example the index creation, searching and deletion are all performed in one step:

```
dataset = single(rand(128,10000));
testset = single(rand(128,1000));

[result,dists] = flann_search(dataset,testset,5,struct('checks',128,'algorithm',...
    'kmeans','branching',64,'iterations',5));
```

## 3.4 Using FLANN from python

The python bindings are automatically installed on the system with the FLANN library if the option `BUILD_PYTHON_BINDINGS` is enabled. You may need to set the `PYTHONPATH` to the location of the bindings if you installed FLANN in a non-standard location. The python bindings also require the numpy package to be installed.

To use the python FLANN bindings the package `pyflann` must be imported (see the python example in section 1.1). This package contains a class called FLANN that handles the nearest-neighbor search operations. This class containg the following methods:

`def build_index(self, dataset, **kwargs)` :

This method builds and internally stores an index to be used for future nearest neighbor matchings. It erases any previously stored index, so in order to work with multiple indexes, multiple instances of the FLANN class must be used. The `dataset` argument must be a 2D numpy array or a matrix, stored in a row-major order (a feature on each row of the matrix). The rest of the arguments that can be passed to the method are the same as those used in the `build_params` structure from section 3.3.1. Similar to the MATLAB version, the index can be created using manually specified parameters or the parameters can be automatically computed (by specifying the target_precision, build_weight and memory_weight arguments).

The method returns a dictionary containing the parameters used to construct the index. In case automatic parameter selection is used, the dictionary will also contain the number of checks required to achieve the desired target precision and an estimation of the speedup over linear search that the library will provide.

`def nn_index(self, testset, num_neighbors = 1, **kwargs)` :

This method searches for the `num_neighbors` nearest neighbors of each point in `testset` using the index computed by `build_index`. Additionally, a parameter called checks, denoting the number of times the index tree(s) should be recursivelly searched, must be given.

Example:

```
from pyflann import *
from numpy import *
from numpy.random import *

dataset = rand(10000, 128)
testset = rand(1000, 128)

flann = FLANN()
params = flann.build_index(dataset, algorithm="autotuned", target_precision=0.9, log_level = "info");
print params

result, dists = flann.nn_index(testset,5, checks=params["checks"]);
```

`def nn(self, dataset, testset, num_neighbors = 1, **kwargs)` :
This method builds the index, performs the nearest neighbor search and deleted the index, all in one step.

`def save_index(self, filename)` :
This method saves the index to a file. The dataset from which the index was built is not saved.

`def load_index(self, filename, pts)` :
Load the index from a file. The dataset for which the index was built must also be provided since it is not saved with the index.

`def set_distance_type(distance_type, order = 0)` :
This function (part of the pyflann module) sets the distance type to be used. See 3.3.5 for possible values of the distance_type.

See section 1.1 for an example of how to use the Python and Ruby bindings.

# References

[AV07] D. Arthur and S. Vassilvitskii. k-means++: the advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035. Society for Industrial and Applied Mathematics Philadelphia, PA, USA, 2007.