Spectral Toolkit of Algorithms for Graphs (STAG)

# Technical Report (2)

Peter Macgregor and He Sun

School of Informatics
University of Edinburgh
United Kingdom
{peter.macgregor, h.sun}@ed.ac.uk

| | |
|---|---|
| **Current Version** | STAG 2.0 |
| **Main Update** | Kernel Density Estimation |
| **Date** | 11th June 2024 |
| **Website** | `https://staglibrary.io` |
| **Source Code** | `https://github.com/staglibrary` |
| **Funding Information** | The development of STAG is supported by UK Research and Innovation under an EPSRC Early Career Fellowship scheme (EP/T00729X/1). |

# 1 Summary

Spectral Toolkit of Algorithms for Graphs (STAG) is an open-source C++ and Python library providing several methods for working with graphs and performing graph-based data analysis. In this technical report, we provide an update on the development of the STAG library. The report serves as a user's guide for the newly implemented algorithms, and gives implementation details and engineering choices made in the development of the library. The report is structured as follows:

- Section 2 describes the locality sensitive hashing, and the main components used in its construction.
- Section 3 describes the kernel density estimation, and the state-of-the-art algorithm for the kernel density estimation. The discussion is at a high level, and domain knowledge beyond basic algorithms is not needed.
- Section 4 describes a fast spectral clustering algorithm, whose implementation is based on efficient kernel density estimation.
- Section 5 provides a user guide to the essential features of STAG which allow a user to apply kernel density estimation and fast spectral clustering.
- Section 6 includes experiments and demonstrations of the functionality of STAG 2.0.
- Section 7 discusses several technical details; these include our choice of implemented algorithms, the default setup of parameters, and other technical choices. We leave these details to the final section, as it's not necessary for the reader to understand this when using STAG.

## 1.1 Implemented Algorithms

STAG 2.0 provides an implementation of the following algorithms:

**Euclidean Locality Sensitive Hashing.** Given a dataset $X$ consisting of $n$ data points $x_1, \ldots, x_n \in \mathbb{R}^d$, and a query point $y \in \mathbb{R}^d$, the nearest neighbour search problem is to recover the points in $X$ close to $y$. This forms the basis of many data analysis algorithms, and many algorithms have been developed to find approximate solutions. Locality Sensitive Hashing (LSH) is a tool that has been used as a building block in many such approximate nearest neighbour algorithms [2, 4], as well as other applications [8, 10, 21]. STAG 2.0 provides an efficient implementation of the Euclidean LSH scheme described by Datar et al. [13].

**Kernel Density Estimation.** Kernel density estimation (KDE) is an important problem with many applications across machine learning and statistics. Given $n$ data points $x_1, \ldots, x_n \in \mathbb{R}^d$ and a query point $q \in \mathbb{R}^d$, the goal of kernel density estimation is to approximate the value of

$$\frac{1}{n}\sum_{i=1}^{n} k(q, x_i),$$

where $k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ is a *kernel function*. The kernel density estimate can be seen as an estimate of the underlying probability density function from which the data is drawn and has been used for many applications in data science [5, 16, 23, 29]. STAG 2.0 provides an implementation of the algorithm for kernel density estimation described by Charikar et al. [10], which is based on the Euclidean locality sensitive hashing primitive.

**Fast Spectral Clustering.** Kernel density estimation algorithms have been applied for fast constructions of similarity graphs used for spectral clustering [23]. STAG 2.0 provides a fast implementation of this algorithm for constructing similarity graphs and the spectral clustering algorithm. This implementation allows spectral clustering to be applied to very large datasets in which the problem was previously intractable.

## 2 Locality Sensitive Hashing

Given data points $x_1, \ldots, x_n$ in a metric space $M$ with distance function $d : M \times M \to \mathbb{R}_{\geq 0}$, the goal of locality sensitive hashing is to preprocess the data in the way such that, given a new query point $y \in M$, an algorithm is able to quickly recover the data points close to $y$. Formally, a family $\mathcal{F}$ of hash functions $h : M \to S$ is *locality sensitive* if there are values $r \in \mathbb{R}$, $c > 1$, and $p_1 > p_2$, such that it holds for $h$ drawn uniformly at random from $\mathcal{F}$ that

$$\mathbb{P}[h(u) = h(v)] \geq p_1$$

when $d(u, v) \leq r$, and

$$\mathbb{P}[h(u) = h(v)] \leq p_2$$

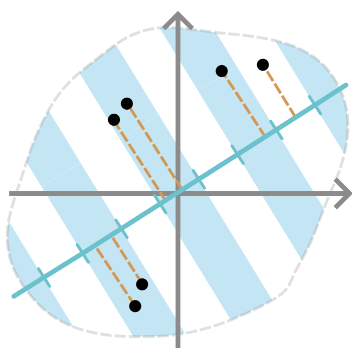
when $d(u, v) \geq c \cdot r$. That is, the collision probability of close points is higher than that of far points. Several techniques for locality sensitive hashing have been proposed. Indyk and Motwani [19] introduce locality sensitive hashing and propose a hashing scheme based on bit-sampling of the data vectors. Broder [9] defines the MinHash algorithm for similarity search on documents, which can be seen as a locality sensitive hashing scheme where the data are subsets of some discrete universe, such as in the bag-of-words model of documents. Charikar [12] presents a technique based on splitting Euclidean space with random hyperplanes and gives a locality sensitive hashing scheme for the Euclidean space with the angular distance. In the STAG library and this report, we focus on the locality sensitive hashing scheme described by Datar et al. [13] for the Euclidean metric space.

### 2.1 Random Projection

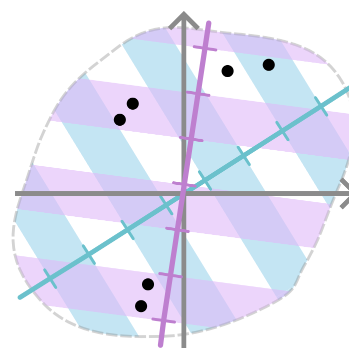
In this section we describe the basic hash functions used to build the Euclidean locality sensitive hashing scheme. For some vector $a \in \mathbb{R}^d$ and a scalar $b \in [0, 4]$, let $h_{a,b} : \mathbb{R}^d \to \mathbb{Z}$ be the function

$$h_{a,b}(x) = \left\lfloor \frac{\langle x, a \rangle + b}{4} \right\rfloor.$$

This corresponds to a projection of the vector $x \in \mathbb{R}^d$ onto $a$, followed by a discretisation into distinct buckets with the floor function. The value of $b$ determines the offset of the bucket boundaries. Figure 1a illustrates this function: the data is projected onto a discretised vector in order to determine the hash bucket for each point. By selecting $a \in \mathbb{R}^d$ from a standard $n$ dimensional normal distribution $\mathcal{N}(0, I)$, and $b$ from the uniform distribution $U(0, 4)$, we obtain a family $\mathcal{F}$ of hash functions.



(a) Projection onto a random vector.



(b) Projection onto two random vectors.

Figure 1: Demonstration of the basic unit of Euclidean LSH: projection onto a random vector. (a) Projecting onto a random vector, with discretisation into hash buckets. (b) Projecting onto multiple random vectors further divides the space. Closer points are more likely to fall into the same hash bucket.

Given two points $x_1, x_2 \in \mathbb{R}^d$ with $c \triangleq \|x_1 - x_2\|$, the probability of $h(x_1) = h(x_2)$ for $h$ drawn uniformly at random from the family $\mathcal{F}$ is shown to be

$$p(c) \triangleq \mathbb{P}\left[h(x_1) = h(x_2) \mid \|x_1 - x_2\| = c\right] = \int_0^4 \frac{1}{c} f\left(\frac{t}{c}\right)\left(1 - \frac{t}{4}\right) \mathrm{d}t,$$

where $f(\cdot)$ is the probability density function of the absolute value of the normal distribution. Solving the integral gives that

$$p(c) = -\frac{1}{2\sqrt{2\pi}}\left(c \cdot \mathrm{e}^{-\frac{8}{c^2}}\right)\left(\mathrm{e}^{\frac{8}{c^2}} - 1\right) + \mathrm{erf}\left(\frac{2\sqrt{2}}{c}\right),$$

where $\mathrm{erf}(\cdot)$ is the error function. This function is shown in Figure 2a. The STAG library provides an implementation of this LSH function.

```
#include <stag/lsh.h>
...
  // Create an LSH function drawn at random from the hash family F.
  StagInt dimension = 10;
  stag::LSHFunction func(dimension);

  // Apply the function to some data point.
  stag::DataPoint p = stag::DataPoint(dimension, &data);
  StagInt bucket = func.apply(p);

  // Compute the collision probability of two points at distance c.
  StagReal c = 1.5;
  StagReal p_c = stag::LSHFunction::collision_probability(c);
...
```

## 2.2 Boosting the Probability

By applying multiple hash functions drawn from the family $\mathcal{F}$, we can achieve different collision probabilities. Firstly, by applying $K$ hash functions independently drawn from $\mathcal{F}$ and taking the intersections of the hash buckets, we have that points collide only if they collide under all $K$ projections. Thus, the collision probability becomes

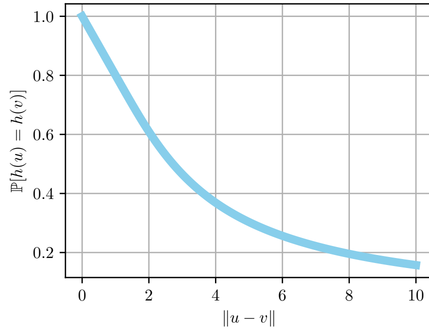
$$p(c)^K.$$

Then, by repeating the process $L$ independent times, the probability of collision in at least one of the $L$ iterations is

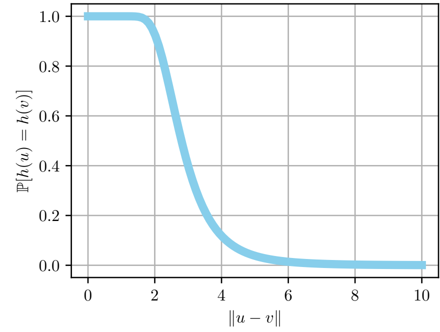
$$1 - \left(1 - p(c)^K\right)^L.$$

By carefully selecting $K$ and $L$, we can control the shape of the collision probability function. Figure 2b shows the collision probability when $K = 6$ and $L = 50$. The E2LSH class in STAG implements a Euclidean locality sensitive hash table for specified values of $K$ and $L$.

```
#include <stag/lsh.h>
...
  // Load a dataset from file.
  DenseMat data_mat = stag::load_matrix(filename);
  std::vector<stag::DataPoint> data = stag::matrix_to_datapoints(data_mat);

  // Create a Euclidean locality-sensitive hash table.
  StagInt K = 6;
  StagInt L = 50;
  stag::E2LSH hash_table(K, L, data);

  // Find the data points in the same bucket as some query point.
  stag::DataPoint q(data_mat, 0);
  std::vector<stag::DataPoint> close_points = hash_table.get_near_neighbors(q);

```



(a) Collision probability $p(c)$



(b) Collision probability for $K = 6$ and $L = 50$

Figure 2: The collision probability of two points under a hash function drawn uniformly at random from the hash family $\mathcal{F}$. Figure (a) shows the collision probability with respect to the distance between the two input vectors $u$ and $v$; Figure (b) shows the value of $1 - \left(1 - p(c)^K\right)^L$, which is the collision probability when applying $K \cdot L$ independent hash functions.

```
16     // Compute the collision probability of two points at distance c.
17     StagReal c = 1.5;
18     StagReal p_c = hash_table.collision_probability(c);
19   ...
```

## 3 Kernel Density Estimation

Given data points $x_1, \ldots, x_n \in \mathbb{R}^d$, the kernel density of a query point $q \in \mathbb{R}^d$ is defined as

$$K(q) \triangleq \frac{1}{n} \sum_{i=1}^{n} k(\|q - x_i\|),$$

where $k : \mathbb{R} \to \mathbb{R}$ is a *kernel function* satisfying that

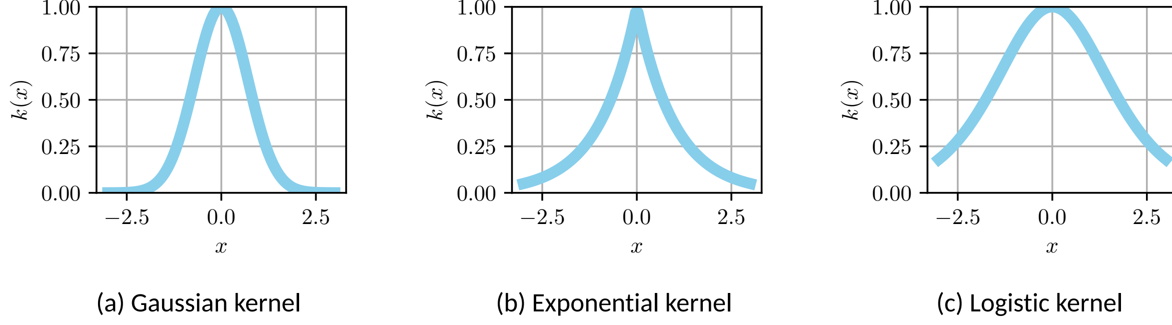
- $k(c) \in [0, 1]$,
- $k(c) = k(-c)$, and
- $k(x) \leq k(y)$ if $|x| > |y|$.

Figure 3 shows the plots of three typical kernel functions defined as follows:

- the Gaussian kernel defined by $k(x) = \mathrm{e}^{-ax^2}$ for some bandwidth parameter $a$,
- the exponential kernel defined by $k(x) = \mathrm{e}^{-a|x|}$ for a bandwidth $a$, and
- the logistic kernel defined by $k(x) = 4/(2 + \mathrm{e}^{x} + \mathrm{e}^{-x})$.

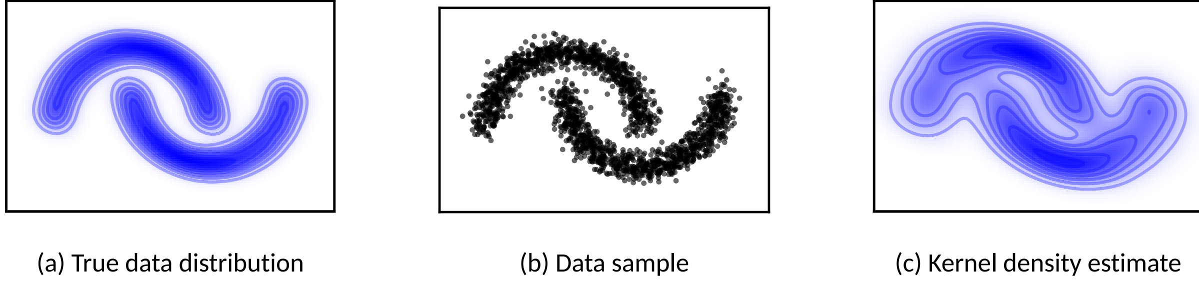
We focus on the Gaussian kernel in the remainder part of the report. Given any dataset and an appropriate choice of kernel, the kernel density is a measure of the density of the data around a particular query point. Figures 4 illustrates that one can see the density of the original data by computing the kernel density at every point of the plane. This technique gives a non-parametric estimate of the probability density from which the data is drawn [15].

Designing efficient algorithms for computing kernel density estimates is an active area of research in both theoretical and applied computer science [3, 10, 11, 20, 31]. For $n$ data points and $m$ query points in $d$ dimensions, a naive computation of the kernel density requires $O(dmn)$ time. Another natural baseline is to take a random sample of the dataset and compute an estimate of the kernel density from the sample.




(a) Gaussian kernel (b) Exponential kernel (c) Logistic kernel

Figure 3: Three kernel functions which can be used for kernel density estimation.



(a) True data distribution (b) Data sample (c) Kernel density estimate

Figure 4: Kernel density estimation provides an estimate of the probability distribution from which the data is drawn. Figure (a) shows the underlying probability distribution; Figure (b) shows the generated data points based on the probability distribution from (a); Figure (c) shows the empirical kernel density estimate of this underlying distribution.

This approach gives a $(1 \pm \varepsilon)$-approximation of the kernel density in time $O\left(\varepsilon^{-2}\mu^{-1}dm\log(m)\right)$, where $\mu$ is a lower bound on the kernel density of any query point [10]. Thus, the most interesting regime for improving the running time of kernel density estimation algorithms is when $\mu = O(1/n)$, as this is the regime in which random sampling does not improve over the naive baseline. In low dimensions, tree-based methods [16] and the celebrated fast Gauss transform [18, 34] perform very well, but suffer from the curse of dimensionality: their running time has exponential dependency on $d$. This has led to the development of methods based on locality sensitive hashing [10, 11] and nearest neighbour search [20]. Among these techniques, the CKNS algorithm described by Charikar et al. [10] is probably the most promising one: after the preprocessing in $\widetilde{O}\left(\varepsilon^{-2}\mu^{-0.25}dn\right)$ time, the algorithm gives a $(1 \pm \varepsilon)$-approximation of the kernel density of $m$ query points in $\widetilde{O}\left(\varepsilon^{-2}\mu^{-0.25}dm\right)$ time, where $\widetilde{O}(\cdot)$ hides poly-logarithmic factors of $n$. Thus, when $\mu = \Theta(1/n)$, this algorithm avoids the exponential dependency on $d$, and offers significant asymptotic speed-up over the naive method and random sampling.

### 3.1 The CKNS Algorithm

In this section, we give a high-level description of the CKNS kernel density estimation algorithm, and refer the reader to [10] for the detailed discussion and analysis of the algorithm. The CKNS algorithm is based on the common algorithmic pattern of importance sampling. Intuitively, when computing the kernel density of some point $q$, the data points closer to $q$ have more influence on the kernel density. As such, we would like to select a random sample of the data points with the sampling probability depending on the distance to the query point. To this end, for some query point $q$, we define the sets $\mathcal{L}_i \subseteq \{x_1, \ldots, x_n\}$ such that it holds for all $x_j \in \mathcal{L}_i$ that

$$2^{-i} \leq k(\|q - x_j\|) \leq 2^{-i+1}.$$

Notice that, for $i < j$, the points in $\mathcal{L}_i$ are closer to $q$ than the points in $\mathcal{L}_j$. Hence, by sampling the points in $\mathcal{L}_i$ with probability proportional to $2^{-i}$, we can obtain an estimate of the kernel density of $q$. To achieve

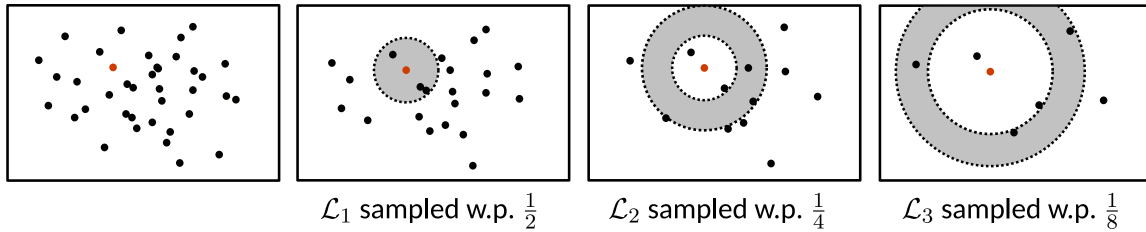



Figure 5: The CKNS algorithm first generates several samples of the data with probabilities $1/2, 1/4, \ldots, 1/n$. Then, for any query point $q$ (indicated as the red point shown above), the data points in $\mathcal{L}_i$ are recovered from the data sampled with probability $2^{-i}$.

this, the CKNS algorithm proceeds with the following steps, which are illustrated in Figure 5.

1. Create $\log n$ subsampled datasets, using sampling probabilities $\{2^{-i}\}_{i=1}^{\log n}$.
2. For each subsampled dataset, create a locality sensitive hash table to recover points in $\mathcal{L}_i$ for any query point.

Then, to estimate the kernel density of a given query point $q$, we use the locality sensitive hash table to recover the points in each level $\mathcal{L}_i$ from the dataset sampled with probability $2^{-i}$, and the kernel density estimate for $q$ is

$$\sum_{i=1}^{\log n} \sum_{\substack{x \in \mathcal{L}_i \\ x \text{ sampled w.p. } 2^{-i}}} 2^i \cdot k(\|q - x\|).$$

A direct calculation shows that this gives an unbiased estimate for the kernel density of $q$. The algorithm is repeated $O(\log n)$ times in order to return a more accurate estimate.

### 3.2 Kernel Density Estimation in STAG

STAG 2.0 provides an implementation of the CKNS KDE algorithm which is intended to be both efficient and easy to use.

```
#include <stag/kde.h>
...
  // Load a dataset from file.
  DenseMat data_mat = stag::load_matrix(filename);
  std::vector<stag::DataPoint> data = stag::matrix_to_datapoints(data_mat);

  // Construct a KDE data structure.
  StagReal a = 0.01;
  stag::CKNSGaussianKDE kde(data, a);

  // Estimate the kernel density of some query point.
  stag::DataPoint q(data_mat, 0);
  StagReal density = kde.query(q);
...
```

## 4 Fast Spectral Clustering

For any set of points $X \triangleq \{x_1, \ldots, x_n\} \subset \mathbb{R}^d$ and parameter $k \in \mathbb{N}$, the goal of spectral clustering is to partition the $n$ points into $k$ clusters such that similar points are grouped into the same cluster. Spectral clustering consists of the following three steps:

1. compute a similarity graph $G$ of $n$ vertices;

2. compute the bottom $k$ eigenvectors of the Laplacian matrix of $G$, and apply these eigenvectors to embed the vertices into $\mathbb{R}^k$;

3. apply $k$-means on the embedded points, and use $k$-means' output as the resulting clustering.

There are several methods for constructing a similarity graph in the first step [33]. One popular method to construct a fully connected similarity graph $G$ is to use a kernel function $K : \mathbb{R} \to \mathbb{R}$ in the following way: every $x_i \in X$ corresponds to a vertex $i$ in $G$, and any pair of vertices $i$ and $j$ is connected by an edge with weight $K(\|x_i - x_j\|)$. Other constructions of similarity graphs include $k$-nearest neighbour graphs and $\varepsilon$-neighbourhood graphs, and more information can be found in [27, 30, 33].

### 4.1 The MS Algorithm

One significant downside to the fully connected similarity graph is that the adjacency matrix is dense, and constructing this graph requires $\Theta(n^2)$ time and space complexity. To overcome this, Macgregor and Sun present an algorithm that constructs a similarity graph in $\widetilde{O}(T_{\mathrm{KDE}}(n))$ time, where $T_{\mathrm{KDE}}(n)$ is the time taken to compute approximate kernel density estimates for every point in the data set. The constructed graph is proven to have the same cluster structure as the fully connected one [23]; we call their algorithm the MS algorithm. For low-dimensional data [1], the *fast Gauss transform* can be applied to compute the kernel density estimates in $O(n)$ time, and Macgregor and Sun provide an implementation of their algorithm in this regime.

To give a high level description of the MS algorithm, assume that $F = (V_F, E_F, w_F)$ is the fully connected graph constructed from $X$ with the kernel function $K$, and $F$ has $k$ well-defined clusters. It is known that, if one samples every edges $e = (i, j) \in E_F$ with probability proportional to

$$\frac{w_F(i,j)}{\deg_F(i)} + \frac{w_F(i,j)}{\deg_F(j)}, \tag{1}$$

where

$$\deg_F(i) = \sum_{j=1}^{n} w_F(i,j) = \sum_{j=1}^{n} K(\|x_i - x_j\|)$$

is the degree of $i$ in $F$, then the resulting graph $G$ has $\widetilde{O}(n)$ edges and the same cluster structure as $F$ [32]. Since $\deg_F(i)$ for every $i$ is proportional to the kernel density at $x_i$, the MS algorithm demonstrates that, by applying a KDE algorithm as a black-box, one can sample every edge $(i, j)$ with approximately the same probability as (1). Based on this technique, the MS algorithm avoids the computation of all the edge weights in $F$, and achieves the total running time of $\widetilde{O}(T_{\mathrm{KDE}})$.

### 4.2 Fast Spectral Clustering in STAG

STAG 2.0 includes an implementation of the MS algorithm [23] based on the fast KDE algorithm in Section 3. The implementation is very easy to use, and requires only the data matrix and the parameter $a$ of the Gaussian kernel.

```
1  #include <stag/cluster.h>
2  #include <stag/graph.h>
3  ...
4    // Load a dataset from file.
5    DenseMat data_mat = stag::load_matrix(filename);
6    std::vector<stag::DataPoint> data = stag::matrix_to_datapoints(data_mat);
7
8    // Construct an approximate similarity graph.
9    StagReal a = 0.01;
10   stag::Graph g = stag::approximate_similarity_graph(&data, a);
```

[1] In theory, "low-dimensional" means that $d = O(1)$. In practice, the fast Gauss transform can be applied when $d \leq 5$.

```
11
12    // Perform spectral clustering on the constructed graph.
13    StagInt k = 2;
14    std::vector<StagInt> labels = stag::spectral_cluster(&g, k);
15 ...
```

# 5 User's Guide

In this section, we include a detailed user's guide for the new features added to STAG in this release. For the installation instructions and other guidance on using STAG, see our first technical report [24] and the online documentation[2]. Although the examples in this section use C++, the same functionality is available in STAG Python. Appendix A includes example code showing how to perform locality sensitive hashing, kernel density estimation, and spectral clustering with STAG Python.

## 5.1 Handling Data

All of the added functionality in this release is based on data sets in $\mathbb{R}^d$. STAG 2.0 introduces the `DenseMat` type alias which is used to represent a dense matrix in $\mathbb{R}^{n \times d}$ containing the $n$ data points.

```
1    typedef Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic, Eigen::RowMajor>
     DenseMat
```

STAG provides methods for reading and writing such data matrices to disk. If the data is stored in a space or comma separated file with one data point on each line, then the `stag::load_matrix` method can be used to read the file.

```
1    #include <stag/data.h>
2    ...
3        std::string my_filename = "data.csv";
4        DenseMat data = stag::load_matrix(my_filename);
5    ...
```

Furthermore, the `stag::save_matrix` method can be used to save a `DenseMat` to disk.

```
1    ...
2        stag::save_matrix(data, my_filename);
3    ...
```

Many methods in the STAG library take a single data point as an argument rather than the entire dataset. For this purpose, STAG 2.0 introduces the `stag::DataPoint` class, which is a wrapper around a pointer to the underlying data. The most common use case of the `stag::DataPoint` class is to point to a certain row of a `DenseMat`.

```
1    #include <stag/data.h>
2    ...
3        DenseMat data = stag::load_matrix(my_filename)
4
5        // Create a data point referencing the 10th row of the data matrix.
6        stag::DataPoint point(data, 10);
7    ...
```

A `stag::DataPoint` object can also be initialised using a C++ vector or by passing a pointer to the data directly.

## 5.2 Locality Sensitive Hashing

The `lsh.h` module provides the Euclidean locality sensitive hashing algorithm.

[2] https://staglibrary.io/

**The LSHFunction Class.** The `stag::LSHFunction` class represents a single locality sensitive hash function based on projection onto a random vector. The function is initialised with a single argument specifying the dimension of the data.

```
#include <stag/lsh.h>
...
  // Create an LSH function for 10-dimensional data.
  stag::LSHFunction f(10);
...
```

The `apply` method is used to compute the hashed value of a data point.

```
...
    // Create a data point referencing the 10th row of the data matrix.
    stag::DataPoint p(data, 10);

    // Compute the hash value of p.
    StagInt h = f.apply(p);
...
```

**The E2LSH Class.** The `stag::E2LSH` class represents a hash table constructed by combining several hash functions. As described in Section 2.2, by combining hash functions with the parameters $K$ and $L$, we can control the collision probability of points hashed into the hash table. The `stag::E2LSH` class takes $K$ and $L$ as arguments, along with a vector of `stag:DataPoint` objects to be stored in the table.

```
#include <stag/lsh.h>
#include <stag/data.h>
...
    // Load the data points.
    DenseMat data = stag::load_matrix(my_filename)
    std::vector<stag::DataPoint> points = stag::matrix_to_datapoints(&data);

    // Create the E2LSH hash table.
    StagInt K = 6;
    StagInt L = 50;
    stag::E2LSH table(K, L, points);
...
```

Once the hash table is constructed, the approximate near neighbours of a query point can be returned with the `get_near_neighbours` method.

```
...
    // Use the 10th row of the data matrix as the query point.
    stag::DataPoint q(data, 10);

    // Return the data points with the same hash value as the query.
    std::vector<stag::DataPoint> close_points = table.get_near_neighbours(q);
...
```

## 5.3 Kernel Density Estimation

The `kde.h` module provides methods for KDE with Gaussian kernel, and `stag::gaussian_kernel` computes the Gaussian kernel value between two data points.

```
#include <stag/kde.h>
#include <stag/data.h>
...
    // Load some data.
    DenseMat data = stag::load_matrix(my_filename);
    stag::DataPoint p1(data, 1);
    stag::DataPoint p2(data, 2);

```

```
        // Compute the Gaussian kernel similarity between the data points.
        StagReal a = 1;
        StagReal val = stag::gaussian_kernel(a, p1, p2);
    ...
```

**The ExactGaussianKDE Class.** The `stag::ExactGaussianKDE` class gives a method to compute the Gaussian kernel density exactly. The data structure is initialised by passing the parameter $a$ of the Gaussian kernel and a `DenseMat` with the dataset.

```
    #include <stag/kde.h>
    #include <stag/data.h>
    ...
        // Load the dataset.
        DenseMat data = stag::load_matrix(my_filename);

        // Create the kernel density data structure.
        StagReal a = 1;
        stag::ExactGaussianKDE kde(&data, a);
    ...
```

The kernel density for a query point can then be computed with the `query` method.

```
    ...
        stag::DataPoint q(data, 10);
        StagReal kernel_density = kde.query(q);
    ...
```

To query multiple points simultaneously, we can instead pass a `DenseMat` to the query method to obtain the kernel density for every row of the matrix.

```
    ...
        std::vector<StagReal> kernel_densities = kde.query(&data);
    ...
```

The time complexity of this method is $O(dmn)$ where $m$ is the number of query points, $n$ is the number of data points, and $d$ is the dimensionality of the data.

**The CKNSGaussianKDE Class.** The `stag::CKNSGaussianKDE` class provides kernel density estimates with the CKNS algorithm described in Section 3.1. There're 3 methods to initialise a `stag::CKNSGaussianKDE` object. The simplest method is to initialise with only a data matrix and the parameter $a$ of the Gaussian kernel.

```
    #include <stag/kde.h>
    #include <stag/data.h>
    ...
        // Load the dataset.
        DenseMat data = stag::load_matrix(my_filename);

        // Create the KDE data structure.
        StagReal a = 1;
        stag::CKNSGaussianKDE kde(&data, a);
    ...
```

This method will choose sensible default parameters for the CKNS algorithm and give good kernel density estimates. For more control over the trade-off between time complexity and accuracy, the `eps` and `min_mu` arguments can be used.

- The `eps` argument corresponds to the $\varepsilon$ error parameter of the CKNS KDE data structure, which is designed to return estimates within a $(1 \pm \varepsilon)$-factor of the exact kernel density. Recall that the initialisation time complexity of the data structure is $\widetilde{O}(\varepsilon^{-2}n^{1.25})$ and the query time complexity is $\widetilde{O}(\varepsilon^{-2}n^{0.25})$. The default value is $0.5$.

- The `min_mu` argument is an estimated minimum kernel density value of any query point. A smaller number will give longer pre-processing and query time complexity. If a query point has a true kernel density smaller than this value, then the data structure may return an inaccurate estimate. The default value is $1/n$.

```
    ...
        // Create the KDE data structure.
        StagReal eps = 0.5;
        StagReal a = 1;
        StagReal min_mu = 0.0001;
        stag::CKNSGaussianKDE kde(&data, a, eps, min_mu);
    ...
```

For those familiar with the details of the CKNS algorithm, the final constructor allows fine-grained control over the constants used by the data structure to control the accuracy and variance of the estimator.

- The `K1` parameter specifies the number of independent copies of the data structure to create in parallel. This controls the variance of the estimates returned by the `query` method. It is usually set to $O(\varepsilon^{-2}\log(n))$.
- The `K2_constant` parameter controls the number of hash functions used in the E2LSH hash tables within the data structure. A higher value will reduce the variance of the estimates at the cost of higher memory and time complexity. It is usually set to $O(\log(n))$.
- The CKNS algorithm samples the data points with different sampling probabilities. Setting `p_offset` to $k \geq 0$ will further subsample the data by a factor of $1/2^k$. This will speed up the algorithm at the cost of some accuracy. It is usually set to $0$.

```
    ...
        // Create the KDE data structure.
        StagReal a = 1;
        StagReal min_mu = 0.0001;
        StagInt k1 = 10;
        StagReal k2_constant = 5;
        StagInt p_offset = 1;
        stag::CKNSGaussianKDE kde(&data, a, min_mu, k1, k2_constant, p_offset);
    ...
```

Once the CKNS data structure has been initialised, the kernel density estimate for any query point can be computed with the `query` method.

```
    ...
        stag::DataPoint q(data, 10);
        StagReal kd_estimate = kde.query(q);
    ...
```

To query multiple data points, it is more efficient to pass a `DenseMat` containing the query points as rows.

```
    ...
        std::vector<StagReal> kd_estimates = kde.query(&data);
    ...
```

## 5.4 Spectral Clustering

STAG 2.0 introduces the new `stag::approximate_similarity_graph` method for constructing a similarity graph from data. The method uses the MS algorithm with the Gaussian kernel and the CKNS kernel density estimation structure. It requires only two arguments: the `DenseMat` containing the data, and the parameter $a$ of the Gaussian kernel.

```cpp
#include <stag/cluster.h>
#include <stag/data.h>
#include <stag/graph.h>
...
    // Load the dataset.
    DenseMat data = stag::load_matrix(my_filename);

    // Create the approximate similarity graph.
    StagReal a = 1;
    stag::Graph g = stag::approximate_similarity_graph(&data, a);
...
```

When the dataset has a clear cluster structure, this method is guaranteed to preserve the structure of the fully connected similarity graph, which can be constructed with the `stag::similarity_graph` method. Once the similarity graph is constructed, we can find the clusters using the `stag::spectral_cluster` method.

```cpp
...
    // Find 10 clusters in the graph.
    StagInt k = 10;
    std::vector<StagInt> labels = stag::spectral_cluster(&g, k);
...
```

# 6 Showcase Studies

In this section, we demonstrate the performance of the kernel density estimation and spectral clustering algorithms available with STAG through experiments on real-world and synthetic datasets. The kernel density estimation experiments are performed on a compute server with 64 AMD EPYC 7302 16-Core Processors and 500 Gb of RAM and the spectral clustering experiments are performed on a laptop with an 11th Gen Intel(R) Core(TM) i7-11800H @ 2.30GHz processor and 32 GB RAM. The code used to produce all experimental results is available at

`https://github.com/staglibrary/kde-experiments.`

## 6.1 Kernel Density Estimation

In this section, we compare the STAG implementation of the CKNS KDE algorithm [10] with the following alternative methods for kernel density estimation.

- **Naive**: compute the exact kernel density by measuring the kernel similarity with every data point.
- **Random sampling (rs)**: estimate the kernel density by sampling a uniformly random subset of the data.
- **DEANN** [20]: importance sampling using an approximate nearest neighbour algorithm as a black box.
- **Sklearn** [28]: kernel density estimation with the ball-tree algorithm implemented in the sklearn machine learning library.

We use the implementation of the naive, random sampling, and DEANN algorithms provided by Karppa et al. [20]. They provide two implementations of the random sampling, and DEANN algorithms: a basic implementation, and a "permuted" implementation with improved memory efficiency which we refer to as "random sampling permuted" (rsp), and DEANNP. We evaluate the algorithms on the following datasets, whose sizes are reported in Table 1:

- **aloi** [17]: colour images of objects under a variety of lighting conditions.
- **covtype** [7]: numerical and categorical data about land use.
- **glove** [29]: word embedding vectors.
- **mnist** [22]: greyscale images of handwritten digits.
- **msd** [6]: numerical and categorical metadata of songs.
- **shuttle** [26]: numerical data from sensors on the space shuttle.

Table 1: Values of $n$ and $d$ for the tested datasets

| dataset | $n$ | $d$ |
|---:|---:|---:|
| aloi | 108,000 | 128 |
| covtype | 581,012 | 54 |
| glove | 1,193,514 | 100 |
| mnist | 70,000 | 728 |
| msd | 515,345 | 90 |
| shuttle | 58,000 | 9 |

We follow the experimental setup established by Karppa et al. [20]. For each dataset, we select 10,000 points uniformly at random to form the query set and perform a grid search over the parameter space of each algorithm to compute the kernel density estimates of the query points. For each set of parameters, we measure the average relative error defined by

$$\frac{1}{m}\sum_{i=1}^{m}\left|\frac{\mu_i - \mu_i^*}{\mu_i^*}\right|,$$

where $m$ is the size of the query set, $\mu_i$ is the kernel density estimate of the $i$th query point, and $\mu_i^*$ is the true kernel density. In Table 2 we report the lowest query time in milliseconds of all parameter configurations which achieve a relative error below $0.1$. Figure 6 further shows how the optimal running time varies for different relative errors. From these results, we observe that the STAG implementation is faster than our tested algorithms for all datasets and relative errors.

Table 2: Minimum per-query time in milliseconds achieving a relative error less than $0.1$

| | Algorithm | | | | | | |
|---|---|---|---|---|---|---|---|
| dataset | stag | DEANN | DEANNP | naive | rs | rsp | sklearn |
| aloi | 0.017 | 0.306 | 0.088 | 0.448 | 0.341 | 0.077 | 26.206 |
| covtype | 0.019 | 0.934 | 0.325 | 3.124 | 0.958 | 0.141 | 158.877 |
| glove | 0.001 | 0.030 | 0.006 | 7.019 | 0.027 | 0.013 | 304.694 |
| mnist | 0.014 | 0.259 | 0.082 | 0.189 | 0.117 | 0.081 | 35.009 |
| msd | 0.011 | 0.610 | 0.267 | 1.758 | 3.355 | 0.743 | 124.321 |
| shuttle | 0.022 | 0.126 | 0.081 | 0.144 | 0.445 | 0.034 | 1.919 |

### 6.2 Spectral Clustering

In this section, we compare several spectral clustering implementations based on a variety of similarity graph constructions.

- **Sklearn FC**: the fully connected similarity graph constructed with the sklearn Python library [28].
- **Sklearn kNN**: the $k$-nearest neighbour similarity graph constructed with the sklearn Python library [28].
- **FAISS HNSW**: an approximate $k$-nearest neighbour similarity graph constructed by the hierarchical navigable small worlds algorithm [25] from the FAISS library [14].
- **FAISS IVF**: an approximate $k$-nearest neighbours similarity graph constructing using the inverted file algorithm from the FAISS library [14].

(a) `aloi` (b) `covtype`

(c) `glove` (d) `mnist`

(e) `msd` (f) `shuttle`

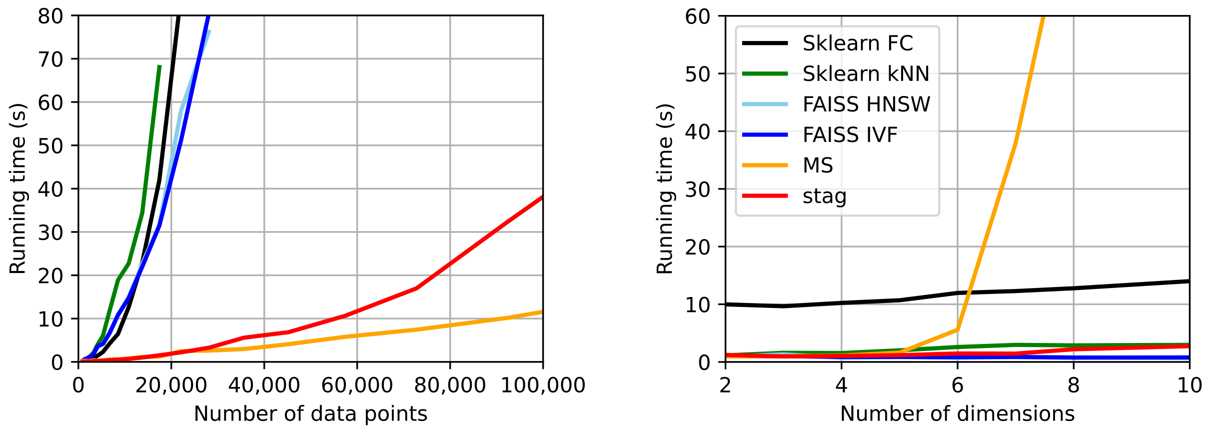
Figure 6: Comparison of running time against relative error.

- **MS**: the similarity graph construction algorithm described in Section 4.1 based on kernel density estimation with the fast Gauss transform [34].
- **STAG**: the similarity graph algorithm provided by STAG, which is based on the MS algorithm and uses the CKNS algorithm [10] for kernel density estimation.

We evaluate the algorithms on two synthetic datasets:

- **moons**: the two-moons dataset from the sklearn library. The data has two dimensions and consists of two clusters which are not linearly separable.
- **blobs**: data is generated from a mixture of Gaussian distributions using the sklearn library. We fix the number of clusters to be 10, and vary the number of dimensions and the number of data points.

The running time of the fast Gauss transform has exponential dependency on the dimension, and as such we expect the MS algorithm [23] to perform well only in low dimensions. To compare the algorithms' performance in the low dimensional setting, we first follow the experimental setup of Macgregor and Sun [23] on the two-dimensional two moons dataset. We vary the number of data points and report the running time of each algorithm in Figure 7a. Secondly, we use the blobs dataset to study how the performance of each algorithm changes as the dimension of data points increases. We fix the number of data points to be 10,000 and vary the dimensionality of the data. The running time of each algorithm is shown in Figure 7b. From these results, we can see that in the two-dimensional case, the MS algorithm outperforms STAG. On the other side, as the dimension of data points increases, the MS algorithm suffers from exponential increase of its running time, while the running time of the other algorithms is not significantly affected.




(a) Comparison on the two-dimensional moons dataset.

(b) Comparison on the blobs dataset with 10,000 points.

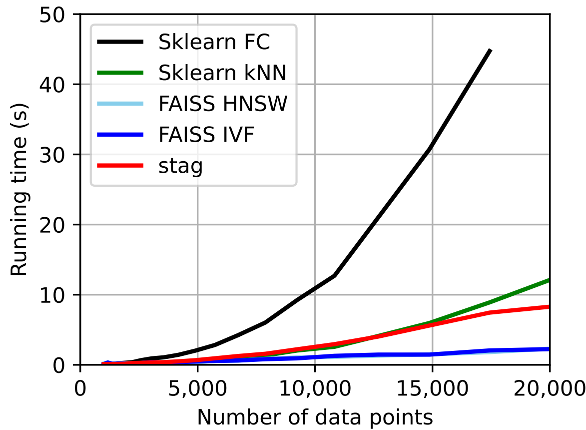
Figure 7: Comparison of different clustering algorithms' running time for low-dimensional data. For every input instance, all the algorithms perfectly return the ground truth clustering.

Finally, we evaluate the algorithms, excluding MS, on 100-dimensional data from the blobs dataset and report the running time in Figure 8. We can see that the STAG algorithm has good performance for high-dimensional data although the FAISS algorithms run faster when the number of data points is large. However, it is worth noting that the HNSW and IVF algorithms are based on heuristics for approximate nearest-neighbour search without formal theoretical guarantees. Moreover, the STAG algorithm is designed to approximate the *fully-connected* similarity graph rather than the $k$-nearest neighbour graph. Hence, among the compared algorithms, the STAG algorithm is the only one with good performance on both low and high dimensional datasets, as well as offering theoretical guarantee on it performance.

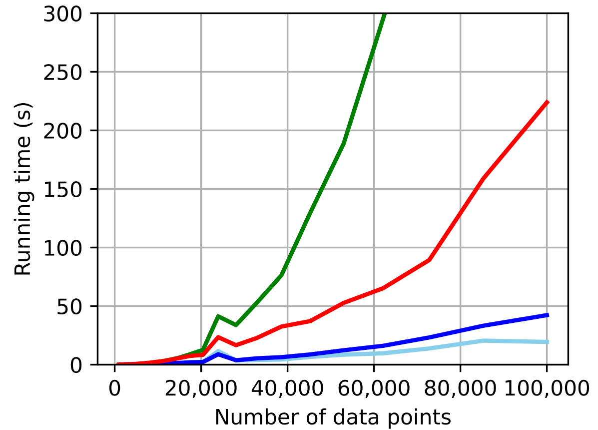
## 7 Technical Considerations

In this section we discuss some of the technical choices made in the design and implementation of STAG.

**Implementation of Euclidean LSH.** The Euclidean LSH algorithm is implemented from scratch in STAG, although the existing C implementation by Andoni [1] is consulted for reference, and we are grateful to




(a) Experimental results on smaller values of $n$




(b) Experimental results on larger values of $n$

Figure 8: Running time comparison of different clustering algorithms on 100-dimensional data from the blobs dataset. For every input instance, all the algorithms perfectly return the ground truth clustering.

Andoni and Indyk for making this implementation available. We choose to re-implement the algorithm in order to take advantage of some modern features of C++ including the `std::vector` data structure. Re-implementing Euclidean LSH also makes us easier to expose it in the STAG Python library.

**Choice of $b$ in Euclidean LSH implementation.** In the STAG implementation of the Euclidean LSH function, we select the parameter $b$ uniformly at random from the interval $[0, 4]$. The choice of $4$ is arbitrary, and choosing another constant will also work with a different corresponding collision probability function. On the use of the interval $[0, 4]$, we follow the recommendation of Andoni [1].

**Choice of implemented KDE algorithm.** In recent years, several kernel density algorithms have been proposed [11, 10, 20, 31]. Of these, we choose to implement the CKNS algorithm [10] because it is a theoretically grounded algorithm which is also relatively simple. Furthermore, prior to the release of STAG 2.0 it lacked a publicly available implementation, and we hope that the release of this implementation could encourage further research into fast algorithms for kernel density estimation in high dimensions.

## References

[1] Alexandr Andoni. E2LSH 0.1 User manual. *http://www. mit. edu/andoni/LSH/*, 2005.

[2] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*, pages 459–468. IEEE Computer Society, 2006.

[3] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt. Practical and optimal LSH for angular distance. *Advances in neural information processing systems*, 28, 2015.

[4] Alexandr Andoni and Ilya P. Razenshteyn. Optimal data-dependent hashing for approximate near neighbors. In *47th Annual ACM Symposium on Theory of Computing (STOC'15)*, pages 793–801, 2015.

[5] Arturs Backurs, Piotr Indyk, Cameron Musco, and Tal Wagner. Faster kernel matrix algebra via density estimation. In *International Conference on Machine Learning*, pages 500–510, 2021.

[6] Thierry Bertin-Mahieux, Daniel P. W. Ellis, Brian Whitman, and Paul Lamere. The million song dataset. In *Proceedings of the 12th International Society for Music Information Retrieval Conference, ISMIR*, pages 591–596, 2011.

[7] Jock A Blackard and Denis J Dean. Comparative accuracies of artificial neural networks and discriminant analysis in predicting forest cover types from cartographic variables. *Computers and Electronics in Agriculture*, 24(3):131–151, 1999.

[8] Dumitru Brinza, Matthew Schultz, Glenn Tesler, and Vineet Bafna. Rapid detection of gene–gene interactions in genome-wide association studies. *Bioinformatics*, 26(22):2856–2862, 2010.

[9] Andrei Z Broder. On the resemblance and containment of documents. In *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171)*, pages 21–29. IEEE, 1997.

[10] Moses Charikar, Michael Kapralov, Navid Nouri, and Paris Siminelakis. Kernel density estimation through density constrained near neighbor search. In *61st Annual IEEE Symposium on Foundations of Computer Science (FOCS'20)*, pages 172–183, 2020.

[11] Moses Charikar and Paris Siminelakis. Hashing-based-estimators for kernel density in high dimensions. In *58th Annual IEEE Symposium on Foundations of Computer Science (FOCS'17)*, pages 1032–1043, 2017.

[12] Moses S Charikar. Similarity estimation techniques from rounding algorithms. In *34th Annual ACM Symposium on Theory of Computing (STOC'02)*, pages 380–388, 2002.

[13] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. Locality-sensitive hashing scheme based on $p$-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262, 2004.

[14] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. The faiss library. 2024.

[15] V. A. Epanechnikov. Non-parametric estimation of a multivariate probability density. *Theory of Probability & Its Applications*, 14(1):153–158, 1969.

[16] Christopher R. Genovese, Marco Perone-Pacifico, Isabella Verdinelli, and Larry Wasserman. Nonparametric ridge estimation. *The Annals of Statistics*, 42(4):1511 – 1545, 2014.

[17] Jan-Mark Geusebroek, Gertjan J Burghouts, and Arnold WM Smeulders. The amsterdam library of object images. *International Journal of Computer Vision*, 61:103–112, 2005.

[18] Leslie Greengard and John Strain. The fast gauss transform. *SIAM Journal on Scientific & Statistical Computing*, 12(1):79–94, 1991.

[19] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *30th Annual ACM Symposium on Theory of Computing (STOC'98)*, pages 604–613, 1998.

[20] Matti Karppa, Martin Aumüller, and Rasmus Pagh. DEANN: speeding up kernel-density estimation using approximate nearest neighbor search. In *25th International Conference on Artificial Intelligence and Statistics (AISTATS'22)*, pages 3108–3137, 2022.

[21] Hisashi Koga, Tetsuo Ishibashi, and Toshinori Watanabe. Fast agglomerative hierarchical clustering algorithm using locality-sensitive hashing. *Knowledge and Information Systems*, 12:25–53, 2007.

[22] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[23] Peter Macgregor and He Sun. Fast approximation of similarity graphs with kernel density estimation. *Advances in Neural Information Processing Systems*, 36, 2023.

[24] Peter Macgregor and He Sun. Spectral toolkit of algorithms for graphs: Technical report (1). *CoRR*, abs/2304.03170, 2023.

[25] Yu A Malkov and Dmitry A Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, 42(4):824–836, 2018.

[26] NASA. Statlog (Shuttle). UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5WS31.

[27] Andrew Y Ng, Michael I Jordan, and Yair Weiss. On spectral clustering: Analysis and an algorithm. In *15th Advances in Neural Information Processing Systems (NeurIPS'01)*, pages 849–856, 2001.

[28] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[29] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[30] Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. *IEEE Transactions on pattern analysis and machine intelligence*, 22(8):888–905, 2000.

[31] Paris Siminelakis, Kexin Rong, Peter Bailis, Moses Charikar, and Philip Levis. Rehashing kernel evaluation in high dimensions. In *International Conference on Machine Learning*, pages 5789–5798, 2019.

[32] He Sun and Luca Zanetti. Distributed graph clustering and sparsification. *ACM Transactions on Parallel Computing*, 6(3):17:1–17:23, 2019.

[33] Ulrike Von Luxburg. A tutorial on spectral clustering. *Statistics and computing*, 17:395–416, 2007.

[34] Changjiang Yang, Ramani Duraiswami, Nail A. Gumerov, and Larry Davis. Improved fast Gauss transform and efficient kernel density estimation. In *9th International Conference on Computer Vision (ICCV'03)*, pages 664–671, 2003.

## A User Guide Examples using STAG Python

This section includes example code omitted from Section 5 demonstrating how to use STAG Python for locality sensitive hashing, kernel density estimation, and spectral clustering. The online documentation[3] contains the full API description of the library.

### A.1 Handling Data

The following example shows how to read and write data to a CSV file.

```
import stag.data

# Read data from disk
data = stag.data.load_matrix("data.csv")

# Write data to disk
stag.data.save_matrix(data, "my_data.csv")
```

STAG Python uses the `stag.utility.DenseMat` class to represent a data matrix. The `to_numpy()` method converts the object to a numpy matrix.

[3] `https://staglibrary.io/docs/python/`

```python
import numpy
import stag.data

# Read data from disk and convert to a numpy matrix
data = stag.data.load_matrix("data.csv")
data_np = data.to_numpy()
```

The following example shows how to create a `DataPoint` object representing one row of the data matrix.

```python
import stag.data

# Create a data point referencing the 10th row of the data matrix
data = stag.data.load_matrix("data.csv")
point = stag.data.DataPoint(data, 10)
```

### A.2 Locality Sensitive Hashing

STAG Python includes the `stag.lsh.LSHFunction` and `stag.lsh.E2LSH` classes with the same functionality as the C++ library. The following code creates and applies an LSH function.

```python
import stag.data
import stag.lsh

# Create an LSH function for 10-dimensional data.
f = stag.lsh.LSHFunction(10)

# Apply the hash function to a data point
point = stag.data.DataPoint(data, 10)
h = f.apply(point)

# Compute the collision probability for points at distance c
c = 1
prob = stag.lsh.LSHFunction.collision_probability(c)
```

The following example creates a Euclidean LSH hash table for a dataset.

```python
import stag.data
import stag.lsh

# Load the data from disk
data = stag.data.load_matrix("data.csv")

# Create the Euclidean LSH table
K = 6
L = 50
dps = [stag.data.DataPoint(data, i) for i in range(data.rows())]
table = stag.lsh.E2LSH(K, L, dps)

# Find the data points close to a query point
q = stag.data.DataPoint(data, 10)
close_points = table.get_near_neighbours(q)
```

### A.3 Kernel Density Estimation

STAG Python provides methods for kernel density estimation in the `stag.kde` module. The Gaussian kernel similarity between two points is computed in the next example.

```python
import stag.data
import stag.kde

# Load the data matrix
data = stag.data.load_matrix("data.csv")
p1 = stag.data.DataPoint(data, 1)
```

```python
p2 = stag.data.DataPoint(data, 2)

# Compute the Gaussian kernel similarity between the points
a = 1
val = stag.kde.gaussian_kernel(a, p1, p2)
```

The following example demonstrates the `stag.kde.ExactGaussianKDE` class.

```python
import stag.data
import stag.kde

# Load the data matrix
data = stag.data.load_matrix("data.csv")

# Create the kernel density data structure
a = 1
kde = stag.kde.ExactGaussianKDE(data, a)

# Compute the kernel density for a query point
q = stag.data.DataPoint(data, 10)
kernel_density = kde.query(q)

# Compute the kernel densities for many query points
kernel_densities = kde.query(data)
```

STAG Python also includes the `stag.kde.CKNSGaussianKDE` class with the same constructors as described in Section 5.3.

```python
import stag.data
import stag.kde

# Load the dataset
data = stag.data.load_matrix("data.csv")

# Create the CKNS data structure using the simple constructor
a = 1
kde = stag.kde.CKNSGaussianKDE(data, a)

# Specify the values of eps and min_mu
eps = 0.5
min_mu = 0.0001
kde = stag.kde.CKNSGaussianKDE(data, a, eps=eps, min_mu=min_mu)

# Fully specify the constants used within the data structure
k1 = 10
k2_constant = 5
p_offset = 1
kde = stag.kde.CKNSGaussianKDE(data, a, min_mu=min_mu, k1=k1, k2_constant=
  k2_constant, sampling_offset=p_offset)
```

The kernel density estimate for a query point can be computed with the `query` method.

```python
# Query a single data point
q = stag.data.DataPoint(data, 10)
kd_estimate = kde.query(q)

# Compute estimates for many data points
kd_estimates = kde.query(data)
```

## A.4 Fast Spectral Clustering

The following example demonstrates how to construct an approximate similarity graph and perform spectral clustering with STAG Python.

```python
import stag.data
import stag.graph
import stag.cluster

# Load the dataset
data = stag.data.load_matrix("data.csv")

# Create the approximate similarity graph
a = 1
g = stag.cluster.approximate_similarity_graph(data, a)

# Find 10 clusters in the graph
k = 10
labels = stag.spectral_cluster(g, k)
```