Hierarchical spectral clustering

Yves Brise

September 23, 2005

Professor: Emo Welzl
Advisors: Joachim Giesen, Dieter Mitsche

Contents

1 Introduction

The goal of this semester project was to implement and test two algorithms that deal with the clustering problem. That is, the problem of grouping n possibly related items into k disjoint clusters. It goes without saying that related items should be grouped together, whereas unrelated items should be distributed into different clusters. This kind of problem can, as an example, be motivated by the task of thematically grouping documents in a newsgroup. Given a collection of sports-related newsletters, what are all the documents concerning basketball? This is one of the questions that one is hoping to answer reliably and efficiently.
Such a partitioning can be modeled as a graph. The elements become the vertices of the graph and two vertices are connected by an edge if and only if the associated elements belong to the same cluster. From this graph we can derive a similarity matrix, which is square and symmetric. In our case we use the $n \times n$ adjacency matrix, having zeros between elements that do not belong to the same cluster and ones between elements that do. In the model of planted partitions, that we will use, the algorithms do not have access to the adjacency matrix, or else the reconstruction will be trivial. Instead the model assumes that the matrix representing the perfect clusters is disturbed by changes that occur with a certain probability, very much like one would imagine the clustering information to be when derived from a loosely related set of elements such as a collection of articles.

Spectral clustering tries to exploit the spectral properties of the similarity matrix in order to reconstruct the clusters only from that information. Important parameters are, of course, the probabilities by which the matrix is changed, but also the size and the number of clusters.
The idea behind hierarchical clustering is very simple. The reconstruction of the clusters is divided into different steps that form a hierarchy of processing. The two algorithms presented, `ClusterHierarchical` and `SpectralReconstruct`, are intended to be applied to the data in sequential order.
The first step is done by `ClusterHierarchical`, which tries to sort out elements that are not classifiable, that is which can not be associated with any of the clusters. The algorithm divides the elements into two different groups and only clears one of these groups for the further processing by the following steps. The elements from the other group are discarded. In our model we will call this the planted partition detection problem, or

just detection for short (see 2.2).
The second step is done by `SpectralReconstruct`, which attempts to do the actual grouping into clusters. We call this the planted partition reconstruction problem, or reconstruction for short.

Here is an outline of the different sections of the thesis.
Section (2) describes the mathematical models and the spectral properties of the adjacency matrix. It also leads to the estimation of the number of clusters, which is a crucial ingredient of the two algorithms. The detection and the reconstruction problem basically rely on the same model, but nevertheless a few differences have to be explained.
The next section (3) presents the algorithms in pseudo code and gives a short prose description of what they are doing.
Section (4) is an experience report about the problems that occurred during the implementation, as well as a sort of user manual for the program FASD, which is one of the results of this semester project.
Finally, the section (5) shows the results of experiments that were made. In these experiments we generate random clustering data and then measure the error of the detection algorithm `ClusterHierarchical` for different parameters. This is followed by a conclusion and an outlook on what future work could comprise.
In the appendix we add and comment on some of the source code, especially those parts that are preoccupied with the calling of external libraries that we use.

2 Model & Problem

2.1 Planted Partition Model

In this subsection we introduce the planted partition reconstruction problem and define two quality measures that can be used to compare different partitioning algorithms. We first introduce the $A(\varphi, p, q)$ distribution, see also McSherry [3].

$A(\varphi, p, q)$ **distribution.**
Given a surjective function $\varphi : \{1, \ldots, n\} \to \{1, \ldots, k\}$ and probabilities $p, q \in (0, 1)$ with $p > q$. The $A(\varphi, p, q)$ distribution is a distribution on the set of $n \times n$ symmetric, 0-1 matrices with zero trace. Let $\hat{A} = (\hat{a}_{ij})$ be a matrix drawn from this distribution. It is $\hat{a}_{ij} = 0$ if $i = j$ and for $i \neq j$,

$$\begin{array}{rcll} P(\hat{a}_{ij} = 1) & = & p & \text{if } \varphi(i) = \varphi(j) \\ P(\hat{a}_{ij} = 0) & = & 1 - p & \text{if } \varphi(i) = \varphi(j) \\ P(\hat{a}_{ij} = 1) & = & q & \text{if } \varphi(i) \neq \varphi(j) \\ P(\hat{a}_{ij} = 0) & = & 1 - q & \text{if } \varphi(i) \neq \varphi(j), \end{array}$$

independently. The *matrix of expectations* $A = (a_{ij})$ corresponding to the $A(\varphi, p, q)$ distribution is given as

$$\begin{array}{rcll} a_{ij} & = & 0 & \text{if } i = j \\ a_{ij} & = & p & \text{if } \varphi(i) = \varphi(j) \text{ and } i \neq j \\ a_{ij} & = & q & \text{if } \varphi(i) \neq \varphi(j) \end{array}$$

Lemma 1 (Füredi and Komlós [1], Krivelevich and Vu [2]) *Let $\hat{A}$ be a matrix drawn from the $A(\varphi, p, q)$ distribution and A be the matrix of expectations corresponding to this distribution. Let $c = \min\{p(1 - p), q(1 - q)\}$ and assume that $c^2 \gg (\log n)^6/n$. Then*

$$|A - \hat{A}| \leq \sqrt{n}$$

with probability at least $1 - 2e^{-c^2 n/8}$. Here $|\cdot|$ denotes the L_2 matrix norm, i.e., $|B| = \max_{|x|=1} |Bx|$.

Planted partition reconstruction problem. Given a matrix $\hat{A}$ drawn from the $A(\varphi, p, q)$ distribution. Assume that all clusters $\varphi^{-1}(l), l \in \{1, \ldots, k\}$ have the same size n/k. Then the function φ is called a *partition function.* The planted partition reconstruction problem asks to reconstruct φ up to a permutation of $\{1, \ldots, k\}$ from $\hat{A}$ only.

Quality of a reconstruction algorithm. A planted partition reconstruction algorithm takes a matrix $\hat{A}$ drawn from the distribution $A(\varphi, p, q)$ as input and outputs a function $\psi : \{1, \dots, n\} \to \{1, \dots, k'\}$. There are two natural measures to assess the quality of the reconstruction algorithm.

(1) The probability of correct reconstruction, i.e.,

$$P[\varphi = \psi \text{ up to a permutation of } \{1, \dots, k\}].$$

(2) The distribution of the number of elements in $\{1, \dots, n\}$ misclassified by the algorithm. The definition for the number of misclassifications used here (see also Meila et al. [4]) is as the size of a maximum matching on the weighted, complete bipartite graph whose vertices are the original clusters $\varphi^{-1}(i), i \in \{1, \dots, k\}$ and the clusters $\psi^{-1}(j), j \in \{1, \dots, k'\}$ produced by the algorithm. The weight of the edge $\{\varphi^{-1}(i), \psi^{-1}(j)\}$ is $|\varphi^{-1}(i) \cap \psi^{-1}(j)|$, i.e. the size of the intersection of the clusters. The matching gives a pairing of the clusters defined by φ and ψ. Assume without loss of generality that always $\varphi^{-1}(i)$ and $\psi^{-1}(i)$ are paired. Then the number of misclassifications is given as

$$n - \sum_{i=1}^{\min\{k,k'\}} |\varphi^{-1}(i) \cap \psi^{-1}(i)|.$$

2.2 Disturbed Planted Partition Model

The assessment of `ClusterHierarchical` requires a slightly different model. We introduce a special symbol to represent the set of elements that do not belong to any cluster, and we have to adapt the measurement of error.

$D(\varphi, p, q)$ **distribution.**
Given a surjective function $\varphi : \{1, \dots, n\} \to \{1, \dots, k\} \cup \{\chi\}$ and probabilities $p, q \in (0, 1)$ with $p > q$. The $D(\varphi, p, q)$ distribution is a distribution on the set of $n \times n$ symmetric, 0-1 matrices with zero trace. Let $\hat{D} = (\hat{d}_{ij})$ be a matrix drawn from this distribution. It is $\hat{d}_{ij} = 0$ if $i = j$

and for $i \neq j$,

$$\begin{array}{rcll} P(\hat{d}_{ij} = 1) &=& p & \text{if } \varphi(i) = \varphi(j) \neq \chi \\ P(\hat{d}_{ij} = 0) &=& 1 - p & \text{if } \varphi(i) = \varphi(j) \neq \chi \\ P(\hat{d}_{ij} = 1) &=& q & \text{if } \varphi(i) = \chi \vee \varphi(j) = \chi \vee \varphi(i) \neq \varphi(j) \\ P(\hat{d}_{ij} = 0) &=& 1 - q & \text{if } \varphi(i) = \chi \vee \varphi(j) = \chi \vee \varphi(i) \neq \varphi(j) \\ , \end{array}$$

independently. The *matrix of expectations* $D = (d_{ij})$ corresponding to the $D(\varphi, p, q)$ distribution is given as

$$\begin{array}{rcll} d_{ij} &=& 0 & \text{if } i = j \\ d_{ij} &=& p & \text{if } i \neq j \text{ and } \varphi(i) = \varphi(j) \neq \chi \\ d_{ij} &=& q & \text{if } i \neq j \text{ and } \varphi(i) = \chi \vee \varphi(j) = \chi \vee \varphi(i) \neq \varphi(j) \end{array}$$

That means that elements that do not belong to any cluster get connected to any other element with probability q. Elements that do belong to clusters get connected to elements from the same cluster with probability p and with probability q to any other element.

Planted partition detection problem. Given a matrix $\hat{D}$ drawn from the $D(\varphi, p, q)$ distribution. The planted partition detection problem asks to find all elements $i \in \{1, \ldots, n\}$ with $\varphi(i) = \chi$. Let us define the size of this special cluster $n_{bad} := \varphi^{-1}(\chi)$. Then we can also define $n_{ok} := n - n_{bad}$. Assume that all other cluster $\varphi^{-1}(l), l \in \{1, \ldots, k\}$, have the same size $s := n_{ok}/k$. Let us further define $r_{bad} := n_{bad}/n$ to be the ratio of elements that do not belong to a cluster.

Quality of the planted partition detection. A planted partition detection algorithm takes a matrix $\hat{D}$ drawn from the distribution $D(\varphi, p, q)$ as input and outputs a function $\zeta : \{1, \ldots, n\} \to \{+, -\}$. The function divides the elements into two groups. There is $P := \zeta^{-1}(+)$ (P for positive), the set of elements that the algorithm assumes to belong to a cluster, and $N := \zeta^{-1}(-)$ (N for negative), the set of elements that do not belong to any cluster according to the algorithm. The name of the algorithm presented later, `ClusterHierarchical`, is motivated by its purpose to apply this preprocessing step to the data. The following measure of error is used to evaluate the quality of a detection algorithm.

(1) An algorithm that tries to assign an element $e \in \{1, .., n\}$ to either P or N may mistakenly decide that the element does not belong to any cluster, in spite of $\varphi(e) \neq \chi$. Therefore, it will put the element

in the class of negative decisions. We define *false negative* to be the set of elements that were mistakenly put in the negative class N. It is also convenient to define the ratio $false^-$ to be the ratio of mistakes within the class N.

$$false^- = \frac{|\{e \in N | \varphi(e) \neq \chi\}|}{n_{ok}} = \frac{|false\ negative|}{n_{ok}}$$

(2) Similarly, we will refer to *false positive* as the subset of the elements $i \in P$ that do not actually belong to a cluster, i.e. $\varphi(i) = \chi$. Again, we also define $false^+$ to be the ratio of wrong decisions within the class P.

$$false^+ = \frac{|\{e \in P | \varphi(e) = \chi\}|}{n_{bad}} = \frac{|false\ positive|}{n_{bad}}$$

2.3 Spectral Properties For Partitioning

Any real symmetric $n \times n$ matrix has n real eigenvalues and $\mathbb{R}^n$ has a corresponding eigenbasis. Here we are concerned with two types of real symmetric matrices. First, any matrix $\hat{A}$ drawn from an $A(\varphi, p, q)$ distribution. Second, the matrix A of expectations corresponding to the distribution $A(\varphi, p, q)$.
We want to denote the eigenvalues of $\hat{A}$ by $\hat{\lambda}_1 \geq \hat{\lambda}_2 \geq \ldots \geq \hat{\lambda}_n$ and the vectors of a corresponding orthonormal eigenbasis of $\mathbb{R}^n$ by $v_1, \ldots, v_n$, i.e., it is $\hat{A}v_i = \hat{\lambda}_i v_i$, $v_i^T v_j = 0$ if $i \neq j$ and $v_i^T v_i = 1$, and the $v_1, \ldots, v_n$ span the whole $\mathbb{R}^n$.
For the sake of analysis we want to assume here without loss of generality that the matrix A of expectations has a block diagonal structure, i.e., the elements in the i-th cluster have indices from $\frac{n}{k}(i-1)+1$ to $\frac{n}{k}i$ in $\{1, \ldots, n\}$. It is easy to verify that the eigenvalues $\lambda_1 \geq \ldots \geq \lambda_n$ of A are $(\frac{n}{k}-1)p + (n - \frac{n}{k})q$, $\frac{n}{k}(p-q) - p$ and $-p$ with corresponding multiplicities 1, $k-1$ and $n-k$, respectively. A possible orthonormal basis of the eigenspace corresponding to the k largest eigenvalues of A is u_i, $i = 1, \ldots, k$, whose j-th coordinates are given as follows,

$$u_{ij} = \begin{cases} \sqrt{\frac{k}{n}}, & j \in \{\frac{n}{k}(i-1)+1, \ldots, \frac{n}{k}i\} \\ 0, & \text{else.} \end{cases}$$

Theorem 1 (Weyl)

$$\max\{|\lambda_i - \hat{\lambda}_i| \mid i \in \{1, \ldots, n\}\} \leq |A - \hat{A}|.$$

Spectral separation. The *spectral separation* $\delta_k(A)$ of the eigenspace of the matrix A of expectations corresponding to its k largest eigenvalues from its complement is defined as the difference of the k-th and the $(k+1)$-th eigenvalue, i.e., it is $\delta_k(A) = \frac{n}{k}(p-q)$.

Projection matrix. The matrix $\hat{P}$ that projects any vector in $\mathbb{R}^n$ to the eigenspace corresponding to the k largest eigenvalues of a matrix $\hat{A}$ drawn from the distribution $A(\varphi, p, q)$, i.e., the projection onto the space spanned by the vectors $v_1, \ldots, v_k$, is given as

$$\hat{P} = \sum_{i=1}^{k} v_i v_i^T.$$

The matrix P that projects any vector in $\mathbb{R}^n$ to the eigenspace corresponding to the k largest eigenvalues of the matrix A of expectations can be characterized even more explicitly. Its entries are given as

$$p_{ij} = \begin{cases} \frac{k}{n}, & \varphi(i) = \varphi(j) \\ 0, & \varphi(i) \neq \varphi(j) \end{cases}$$

Lemma 2 *All the k largest eigenvalues of $\hat{A}$ are larger than $\sqrt{n}$ and all the $n-k$ smallest eigenvalues of $\hat{A}$ are smaller than $\sqrt{n}$ with probability at least $1 - 2e^{-c^2n/8}$ provided that n is sufficiently large and $k < \frac{p-q}{4}\sqrt{n}$.*

Proof. Plugging in our assumption that $k < \frac{p-q}{4}\sqrt{n}$ gives that the k largest eigenvalues of A are larger than $4\sqrt{n} - p > 2\sqrt{n}$. By the lemma of Füredi and Komlós it is $|A - \hat{A}| \leq \sqrt{n}$ with probability at least $1 - 2e^{-c^2n/8}$. Now it follows from Weyl's theorem that the k largest eigenvalues of $\hat{A}$ are larger than $\sqrt{n}$ with probability at least $1 - 2e^{-c^2n/8}$. Since the $n-k$ smallest eigenvalues of A are $-p$ it also follows that the $n-k$ smallest eigenvalues of $\hat{A}$ are smaller than $\sqrt{n}$ with probability at least $1 - 2e^{-c^2n/8}$. $\square$

Lemma 3 *With probability at least $1 - 2e^{-c^2n/8}$ it holds*

$$\frac{n}{k}(p-q) - p - \sqrt{n} \leq \hat{\lambda}_2 \quad \textit{and} \quad \hat{\lambda}_2 \frac{k}{n} - \frac{k}{\sqrt{n}} \leq p - q,$$

provided n is sufficiently large.

PROOF. It holds $\lambda_2 = \frac{n}{k}(p-q) - p$. By combining Weyl's theorem and the lemma of Füredi and Komlós we get that with probability at least $1 - 2e^{-c^2 n/8}$ it holds

$$\hat{\lambda}_2 \in \left[\frac{n}{k}(p-q) - p - \sqrt{n},\ \frac{n}{k}(p-q) - p + \sqrt{n}\right].$$

Hence with the same probability

$$\hat{\lambda}_2 \frac{k}{n} - \frac{k}{\sqrt{n}} \leq p - q \leq \hat{\lambda}_2 \frac{k}{n} + \frac{k}{\sqrt{n}} + \frac{k}{n}.,$$

where we used $p \leq 1$ for the upper bound and $p \geq 0$ for the lower bound. □

Remark.
All the considerations in this subsection apply to matrices drawn from the $A(\varphi, p, q)$ distribution only, so strictly speaking, we can not derive anything about the $D(\varphi, p, q)$ distribution from that. Lemma 2 suggests that we can estimate the number clusters, k, with a high probability by counting the eigenvalues that are greater than $\sqrt{n}$. This is a very important fact that we use in our algorithm `SpectralReconstruct`. However, in a similar derivation it can be shown that this procedure is also applicable to the case of the $D(\varphi, p, q)$ distribution, as done in the algorithm `ClusterHierarchical`.

3 Algorithms

3.1 SpectralReconstruct

Now we have all prerequisites at hand that we need to describe our spectral algorithm to solve the planted partition reconstruction problem.

```
SpectralReconstruct($\hat{A}$)
 1  $k' :=$ number of eigenvalues of $\hat{A}$ that are larger than $\sqrt{n}$.
 2  $\hat{P} :=$ projection matrix computed from the $k'$ largest eigenvectors
          $v_1, \ldots, v_{k'}$ of $\hat{A}$.
 3  for $i = 1$ to $n$ do
 4     $R_i :=$ set of row indices which are among the $\frac{n}{k'}$ largest entries of
             the $i$-th column of $\hat{P}$.
 5     for $j = 1$ to $n$ do
 6        $c_{ij} := \begin{cases} 1, & j \in R_i \\ 0, & \text{else} \end{cases}$
 7     end for
 8     $c_i := (c_{i1}, \ldots, c_{in})^T$
 9  end for
10  $I := \{1, \ldots, n\}$; $l := 1$
11  while exists an unmarked index $i \in I$ do
12     $C_l := \emptyset$
13     for each $j \in I$ do
14        if $c_i^T c_j > \frac{4n}{5k'}$ do
15           $C_l := C_l \cup \{j\}$
16        end if
17     end for
18     if $|C_l| \geq \left(1 - \sqrt{\frac{160\sqrt{n}}{\lambda_2 - 3\sqrt{n}}}\right) \frac{n}{k'}$ do
19        $I := I \setminus C_l$; $l := l + 1$
20     else
21        mark index $i$.
22     end if
23  end while
24  $C_l := I$
25  return $C_1, \ldots, C_l$
```

In line 1 the number of planted clusters, k', is estimated. The estimate is motivated by Lemma 2. In line 2 the projection matrix $\hat{P}$ that belongs to $\hat{A}$ is computed. From line 3 to line 9 for each column i of $\hat{A}$ a vector $c_i \in \{0,1\}^n$ with exactly $\frac{n}{k'}$ entries that are one is computed. In lines 10 to 24 the actual reconstruction of the clusters takes place. Roughly speaking, two indices i, j are put into the same cluster if the Hamming distance of the corresponding vectors c_i and c_j is small (test in line 14). A cluster as created in lines 12 to 17 is not allowed to be too small (test in line 18), otherwise its elements get distributed into other clusters that are going to be constructed in future executions of the body of the while-loop. Note that the algorithm runs in time polynomial in n and only makes use of quantities that can be deduced from $\hat{A}$, i.e., it does not need to know the values of p, q and k.

3.2 ClusterHierarchical

This algorithm deals with the removal of elements that do not belong to a cluster. The input is the symmetric similarity matrix $\hat{D}$, and the output is the matrix $\hat{D}$, only restricted to a certain subset of its columns and rows. Elements that do not seem to belong to any cluster are detected and discarded for the further reconstruction. Let v_{ij} be the j^{th} entry of the eigenvector corresponding to the i^{th}-largest eigenvalue.

```
ClusterHierarchical($\hat{D}$)
1  $k'$ := number of eigenvalues of $\hat{D}$ that are larger than $\sqrt{n}$.
2  $w_j := \sum_{i=1}^{k'} |v_{ij}|, 1 \le j \le n$
3  for $j = 1$ to $n$ do
4     if $w_j > T$ then $c_j := 1$
5     else $c_j := 0$
6  end for
7  return $\hat{D}$ restricted to rows and columns $j$ with $c_j = 1$.
```

In line 1 the estimated number of clusters, k', is calculated. The estimate is motivated by Lemma 2. In line 2 the component-wise sum of the k' largest eigenvectors is calculated. In lines 3 to 6 a characteristic vector c is computed indicating the status of each element. If $c_j = 0$ then the element j is considered to be a member of the set N, or alternatively $\zeta(j) = -$. If $c_j = 1$ then the element j is considered to be a member of the set P. In the last line the algorithm returns the similarity matrix $\hat{D}'$ restricted to the elements from P.

4 Implementation

To test the algorithms on random data we have implemented a C program called FASD, which stands for 'Finding All Similar Documents' the motivation and original title of my semester project. For calculating the eigenvalues and eigenvectors of the adjacency matrix the implementation makes use of the TRLAN library.

4.1 Random Number Generator

All experiments were done with random matrices. **Drand48** from the C standard library was used to generate the random matrices.

```
#include <stdlib.h>
#include <time.h>

double number;

srand48(time(NULL));
number = drand48();
```

Drand48 generates double precision pseudo-random numbers, uniformly distributed in the interval $(0, 1)$. The routine works by generating a sequence of 48-bit integer values, X_i, according to the linear congruential formula:

$$X_{n+1} = (aX_n + c) \bmod m, n \geq 0$$

where $m = 2^{48}$, $a = 5DEECE66D_{16}$, and $c = B_{16}$.
The random number generator is initialized every time the program is started. This is done by the call to **srand48(**long int *seed* **)**. As seed the system time in seconds since the 1.1.1970 is used, obtained by the call to **time(**long int *sec**)**.

4.2 Building Under Cygwin

A considerable amount of time was used to make TRLAN accessible on a Windows XP machine. That is why we describe here the steps that led to the final solution, in order to facilitate further work. There were basically two different approaches. The Microsoft Visual Studio C++, and the Cygwin solution. In the end we have only managed to get the

Cygwin solution running. TRLAN and the other libraries are written in Fortran. Our choice of favored implementation language was C/C++. However, the interfacing between the two languages sometimes proved to be treacherous.

Microsoft Visual Studio C++ It is possible to compile Fortran code with the Visual Studio, but only with an integrated Fortran compiler such as Intel Visual Fortran compiler. It is possible to set up a project and compile TRLAN to a library. It is rather tedious work since one has to manually add those files to the project that are needed by TRLAN from LAPACK and BLAS directories. In the file collection that comes with this semester project there is a list of all the files that are needed. If you do it manually, it is good practice to include all the TRLAN files to the project and compile it once. Then the compiler will complain about undefined references. The missing files have the same name as the functions that were not found. After adding all missing files, one has to compile again to find all the undefined references that were caused by the newly added files. One has to repeat that process until there are no more undefined references.
We used a trial version of the Intel Visual Fortran Compiler and tried to make it work. Unfortunately, we did not manage to call any of the Fortran functions from C/C++. We suppose with some additional work it should be possible to compile everything under Visual Studio. Anyway, we had to look for a quicker way to the solution, that is why in the end we developed the Cygwin solution. One other problem with the Visual Studio solution is that it is not free software, of course. The Intel Fortran Compiler and the Microsoft Visual Studio are commercial products, so the redistributability and usability of the FASD code would be depending on the availability of these programs.

Cygwin Cygwin is a open source emulation of Linux/UNIX like systems on Windows PCs. It consists of a DLL (cygwin1.dll), which acts as an emulation layer providing substantial POSIX (Portable Operating System Interface) system call functionality, and a collection of tools, which provide a Linux look and feel. The Cygwin DLL works with all x86 versions of Windows since Windows 95. The API follows the Single Unix Specification as much as possible, and then Linux practice. Using Cygwin it is possible to compile all the libraries with the Makefile that they come with, and more importantly there are free compilers available for C/C++ as well as for Fortran. As C/C++ compiler we used **g++**

and as Fortran compiler we used **g95**.
First, one has to compile TRLAN with **g95** to make a Cygwin library. Then, the FASD program is compiled with **g++**. The linking has to be done with **g95** again. Unfortunately, we did not manage to give the **g95** linker access to the C++ library, which led to the inconvenience of not being able to use any C++ syntax. That is the reason why FASD is written in pure C. It would have been favorable if we could have used C++ standard libraries, but again, after a lot of tinkering around we had to decide for the running version. In the appendix there are some excerpts from the source code that show how the interfacing between TRLAN and FASD is done.

4.3 TRLAN, LAPACK and BLAS

TRLAN is a program designed to find the eigenvalues and the corresponding eigenvectors of sparse symmetric matrices. It is especially optimized to find the extreme eigenvalues, using a restarted Lanczos algorithm. This is convenient for us since we only need the largest eigenvalues for our algorithms. TRLAN is written in Fortran95 and had to be built from source. For a complete description of TRLAN please refer to http://crd.lbl.gov/∼kewu/trlan.html, the web page of TRLAN.
TRLAN makes use of LAPACK (Linear algebra PACKage) and BLAS (Basic Linear Algebra), two freely available Fortran77 libraries. The libraries provide efficient implementations of matrix and vector operations, that are needed by TRLAN. For a complete description and the source code of LAPACK and BLAS, refer to www.netlib.org.

4.4 FASD

General FASD is a simple command line based C program that lets you generate random matrices and run the described algorithms on those matrices. In particular, you can:

- save/load matrices.
- generate different random matrices.
- run `ClusterHierarchical` on the loaded matrix.
- run `SpectralReconstruct` on the loaded matrix.

- run series of experiments with `ClusterHierarchical` using different parameters.

The commands will be explained in the following. Typing 'fasd.exe' will run the program from the Cygwin command line. FASD will enter the first command loop and prompt you for the command to execute. From here you can load matrices, generate random matrices, and run series of experiments. Most importantly you can run 'help' from here to get a short description of all commands.

First Command Loop
The program will prompt

```
command:
```

and wait for user input. This is a list of all commands that you can run from the first prompt:

- **load:** Type 'load' and press enter. The program will ask you for a file name. Enter the name of a file that has columns separated by spaces and rows separated by line breaks. The first line of the file should be the string 'size: n' with n the actual dimension of the matrix. The format can always be checked by generating a random matrix, saving it, and opening it with a text editor.

- **params:** The command **params** lets you change the parameters p and q as defined in section (2). The command will prompt you for two real numbers between 0 and 1. The first one will be the new value for p and the second will be the new value for q.

  One more note: Instead of typing 'params' + enter, and then two times a number + enter, one can directly write 'params p q' + enter. The appended parameters will be recognized correctly. This applies to all commands. If you know the format and the order of the following prompts you can append their parameters to the first command prompt. Just separate the command line parameters by spaces.

- **rand:** Type 'rand' and press enter to run the command. FASD will prompt you for two integer numbers, n the size of the problem and k the number of clusters. The command **rand** will generate a random matrix with at least k clusters. If n is not a multiple of k then the size of the clusters is $\lfloor \frac{n}{k} \rfloor$ and the remaining elements will be packed into their own cluster of smaller size. The probability that elements within a cluster are connected is p, and the probability that elements between different clusters are connected is q. Remember that p and q can be set with **params**.

- **randdist:** Type 'randdist' and press enter. FASD will prompt you for three numbers. As in **rand** it will prompt for n and k. This time it will also prompt for a number, $r_{bad} \in (0, 1)$, to specify the ratio of unclassifiable elements that are added to the matrix. First **randdist** determines n_{ok} and n_{bad} as defined in section (2.2). The first n_{ok} entries of the matrix are treated just as in the case of **rand**. Ideally, n_{ok} should be multiple of k. The probabilities are p and q again. The last n_{bad} entries of the matrix are connected to any other element with the probability q, just as two elements that belong to the undisturbed part of the matrix but not to the same cluster.

- **seriesCH:** This command runs an automated series of experiments with random matrices for `ClusterHierarchical`. Random matrices of the same type as described in **randdist** are generated and the algorithm `ClusterHierarchical` tries to identify the bad entries. The results are written to a file named 'fasdCH.output'. Be careful, results from previous experiments will be overwritten without prompt. The measure of error is the same as defined in the section (2.2). As to the exact range of parameters we have to refer to the source code. To change the range for lets say k, one has to change that in the source code and recompile the program. Also, the exact format of the output file can be specified easily in the source code.

- **seriesHR:** This command is very similar to the command **seriesCH** in the sense that it also conducts parameter studies, but this time for the planted reconstruction problem with no bad elements and the algorithm `SpectralReconstruct`. The results are written to a file named 'fasdHR.output'. The measure of error is defined on section (2.1). Again, the range of parameters and the output format have to be changed in the source code, and the program has to be recompiled to change the setting of the experiment.

- **quit:** Typing this command, will result in FASD freeing all of its memory and exiting.

Second Command Loop

In case **load**, **rand**, or **randdist** have been run, there will be a specific matrix loaded in the program. TRLAN will automatically be called to get the eigenvalue decomposition of the matrix. FASD enters another mode and prompts you for an action, this time asking for an action that should be applied to the currently loaded matrix.

```
action:
```

The different options that you have, are:

- **save:** Typing 'save' and pressing enter will result in FASD asking you for a filename. If you type a filename, FASD will store the currently loaded matrix to that file, overwriting any present file. The matrix will be stored in a simple space separated format. The first line of the file will be 'size: n' where n is the size of the stored matrix. On the second line follows the first row of the matrix, its values separated by spaces. On the third line is the second row of the matrix, and so on.

- **spectral:** Typing this keyword and pressing enter lets the algorithm `SpectralReconstruct` run on the loaded matrix. The reconstructed set of clusters is printed out as a vector of integers. The size of the vector is n and the value of the integer indicates the cluster that this element belongs to according to the reconstruction.

- **cluster:** Runs the algorithm `ClusterHierarchical` on the loaded matrix. The separation between elements that belong to a cluster and elements that do not is printed out as a vector of size n. The vector consists of zeros and ones only. A zero indicates that the corresponding index belongs to the disturbed part of the matrix, i.e. is considered an element that does not belong to a cluster. A one stands for an element that belongs to a cluster. Since the matrices generated by **randdist** assign high indices to the disturbed entries, the vector should have ones in the beginning and as many zeros at the end as there are elements that do not belong to a cluster.

- **stop:** Unloads the current matrix and leaves the second command loop. FASD will enter the first command loop again and prompt you for a command. Type 'quit' from there to completely leave the program.

5 Experiments

The following subsections describe different experiments that were done with FASD. The variables that we use are all defined in section (2.2).

5.1 Threshold For `ClusterHierarchical`

An important parameter for the algorithm `ClusterHierarchical` is the threshold T, that is used to separate the elements. The goal of the experiments is to find this threshold. We use the following function for T,

$$T = \frac{a}{\sqrt{n}},$$

which reduces the problem to finding a.

This first series of experiments investigates the dependence of a on k. Therefore we fix the following parameters. The number of elements in each cluster is 35. The ratio of elements that do not belong to a cluster, r_{bad}, is 0.3. Then we fix p to 0.9 and q to 0.1. Remember that $w_j = \sum_{i=1}^{k'} |v_{ij}|$ is the sum of the j^{th} components of the k' eigenvectors that belong to the k' largest eigenvalues. This w_j is compared with T to decide if the element j should be considered as belonging to a cluster or not.
The experiment conducts a parameter study for a, letting it run from 1.0 to 4.0 in steps of 0.1. Different values for k were tried. Figure (1) shows the plots of the results for $2 \leq k \leq 5$. The size of the clusters is fixed, that is why the size of the problem, n, grows with increasing k. For every different setup of k and a, 30 random experiments have been run. The error measures $false^-$ and $false^+$ have been averaged over these trials.

The plots that we can see in figure (1) suggest a linear function for the parameter a. For every k there is a very distinct interval where both errors, $false^-$ and $false^+$, tend to zero. Increasing k by one seems to shift this interval to the right by about 0.6 units of a. That led to the threshold

$$T = \frac{1.6 + 0.6(k' - 2)}{\sqrt{n}}$$

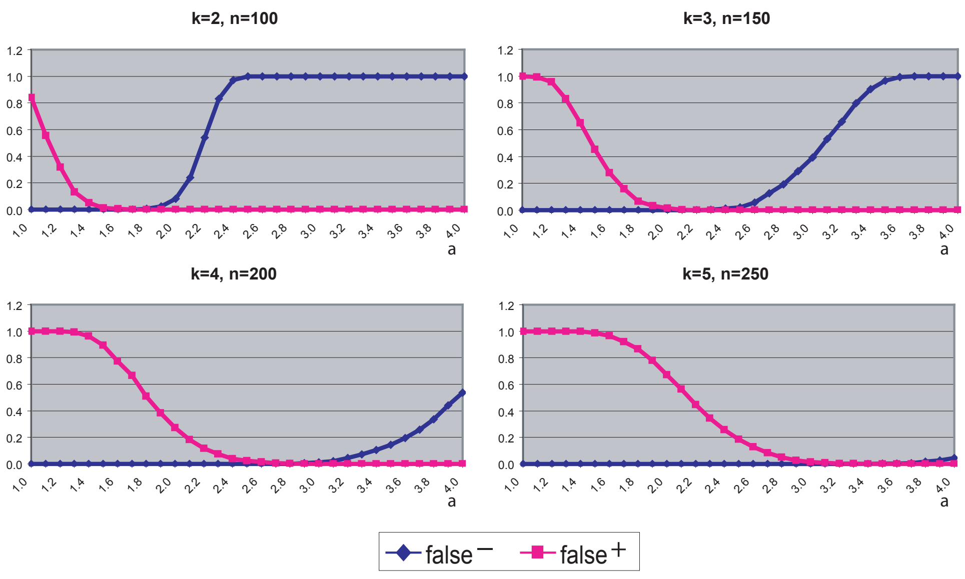

Figure 1: The x-axis shows the parameter a from the threshold, and the y-axis shows the ratio of wrongly classified elements. The size n of the matrix varies since the size of the clusters is fixed.

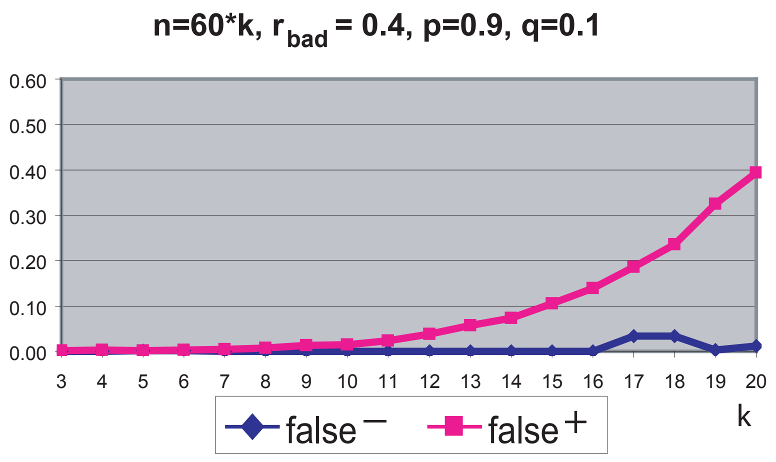

Figure 2: The x-axis shows the number of clusters, k, and the y-axis shows the ratio of wrongly classified elements. The size n of the matrix varies since the size of the clusters is fixed.

For a relatively small value of k this threshold detects the bad elements very well. In figure (2) we can see the results for a broader range of k. We use the linear threshold that was suggested by the last experiment.

The size of the clusters is fixed to 36 and the ratio of elements that do not belong to clusters, r_{bad}, is 0.4. That is why the size, n, amounts to $60k$. The probabilities are set as follows, $p = 0.9$ and $q = 0.1$. We let k run from 3 to 20. For every setup of k, 30 trials have been calculated to average the error. It may be mentioned that this experiment includes the computation of the largest matrices that have been calculated. For the case of $k = 20$ the size of the matrices is 1200×1200.

5.2 Increasing Number Of Bad Elements

In this experiment we try to find out how `ClusterHierarchical` behaves when confronted with an increasing number of elements that do not belong to any cluster. As the threshold T we use the linear threshold from the previous subsection. This time the size, n, is fixed to 300. As before, $p = 0.9$, and $q = 0.1$. The ratio of elements that do not belong to clusters, r_{bad}, was varied between 0.0 and 0.95, increasing by steps of 0.05. The number of clusters is varied between 2 and 6. For every setup of k and r_{bad}, 30 random experiments have been calculated to average the error.
In this experiment the cluster size, s, varies because n is fixed. It is important to note that also the smaller s will make the planted partition detection harder, not only the increasing r_{bad}.

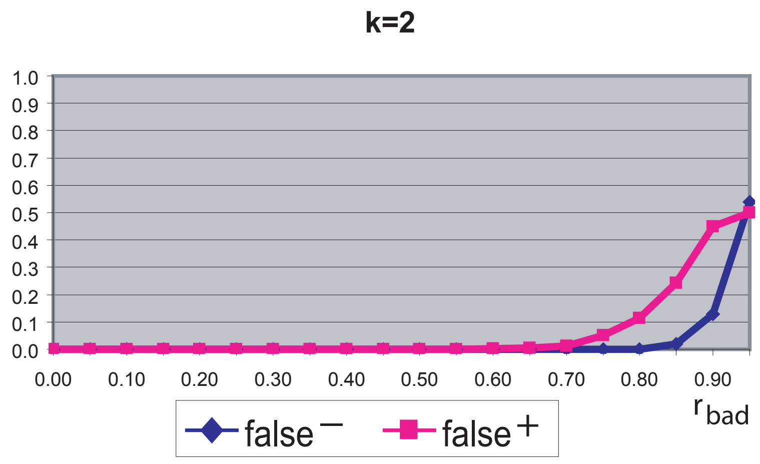

Figure 3: The x-axis shows r_{bad}, the ratio of elements that do not belong to any cluster. The y-axis shows $false^{-}$ and $false^{+}$. $n = 300$, $k = 2$.

In figure (3) we can see the plots of $false^-$ and $false^+$ for $k = 2$. The algorithm `ClusterHierarchical` manages to make an almost perfect detection for $r_{bad} \leq 0.7$. In this setup, $r_{bad} = 0.7$ means that there are 210 elements that do not belong to clusters, and 90 that do. The size of a cluster is therefore 45. If r_{bad} increases further then the detection collapses. First $false^+$ rises, shortly before $false^-$ rises too.
In the end, $false^-$ and $false^+$ come close to 0.5. Note that these values are attained by a detection algorithm that will simply distribute the elements to P or N with probability 0.5. As expected, the result for $r_{bad} = 0.95$ can be called a complete miss.

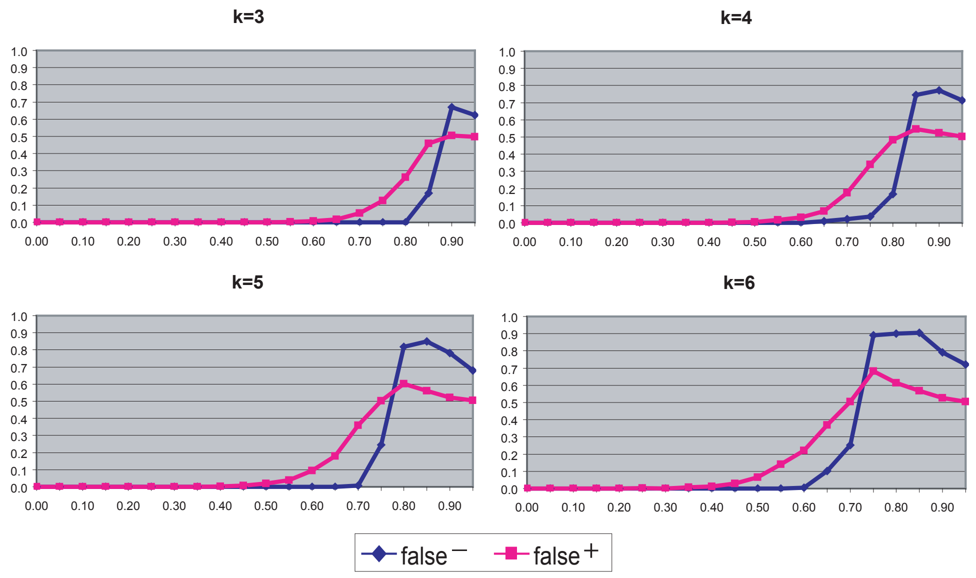

Figure 4: The x-axis shows r_{bad}, the ration of elements that do not belong to any cluster. The y-axis shows $false^-$ and $false^+$. $n = 300$.

In figure (4) there are the plots of $false^-$ and $false^+$ for other k. With increasing k we notice that the point at which the detection becomes error prone shifts to the left. As mentioned, this is at least partly due to the smaller cluster size that results from the fixed n.
An interesting point about this series of plots is that, for r_{bad} greater than a certain value, the two measures of error rise above 0.5. So more than half of the elements of each class is actually in the wrong class. This degeneration of the detection coincides with the collapse of the estimation of k in `ClusterHierarchical`. If the eigenvalues of the clusters become

to small then not all of them will be accounted for in the estimation of k. `ClusterHierarchical` will attempt detection in the belief that there are less clusters than there are in reality ($k' < k$).
For example, for $k = 5$ and $r_{bad} \geq 0.7$ the estimation of k collapses. Shortly after that point the values for $false^-$ and $false^+$ rise above 0.5. It turns out that the estimation of k always collapses before the errors cross the 0.5 marker.

5.3 Varying p & q

In this experiment we are interested in the effect that varying p and q have on the detection algorithm `ClusterHierarchical`. As the threshold T we use the linear threshold from the previous sections. In all the experiments from this subsection, r_{bad} is set to 0.4. That means that 40% of the elements do not belong to clusters. The size of the problem, n, the number and size of the clusters, and the value for p and q themselves are varied in the different runs. The actual values are indicated in the label or the caption of the plot. For every setup of p, q, n, and k, 30 random experiments have been calculated to average the error.

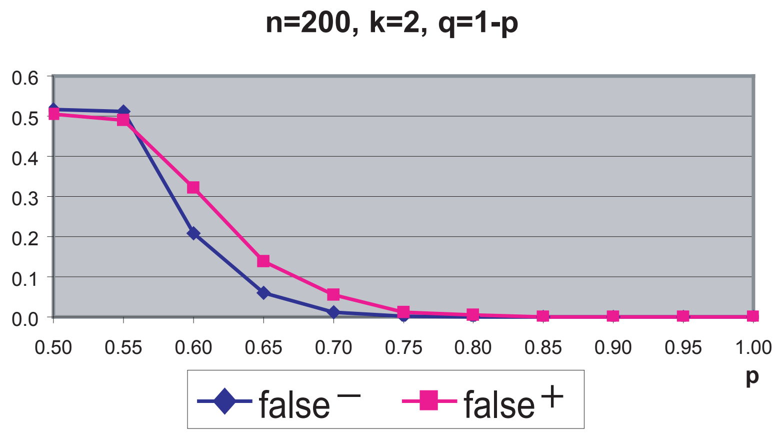

Figure 5: The x-axis shows p, the probability that two elements from the same cluster are grouped together by the similarity matrix. The y-axis shows the normalized error. The probability q is set to $1 - p$.

In figure (5) we can see the plot of a run, where $q = p - 1$, starting from the completely random distribution for $\hat{D}$ with $p = q = 0.5$. Then p is increased in steps of 0.05. Consequently, q is decreased by 0.05, thus increasing $p-q$ by 0.1. If we look at $x = p - q$, then we find ourselves in the symmetric situation around 0.5, that is $p = 0.5 + \frac{x}{2}$ and $q = 0.5 - \frac{x}{2}$. The plot shows that for $p > 0.75$ the detection is perfect.

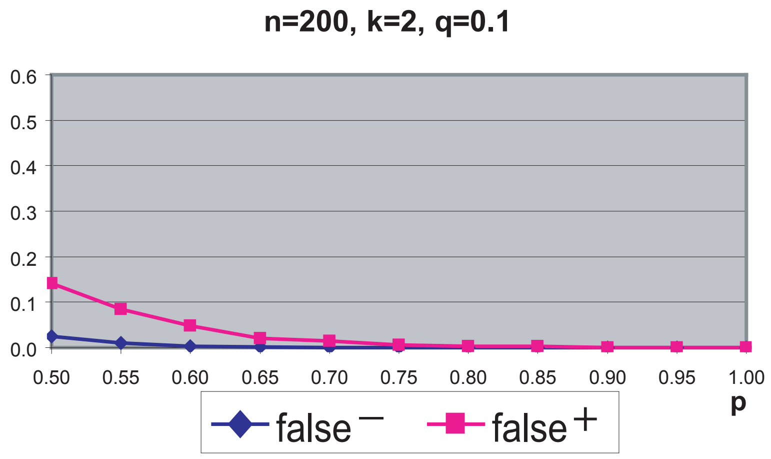

Figure 6: The x-axis shows p, the probability that two elements from the same cluster are grouped together by the similarity matrix. The y-axis shows the error percentage. The probability q is $1 - p$.

The situation in figure (6) is only slightly different. In fact, the only difference in the choice of the parameters is that here we fix p to 0.1. The parameter p is again increased by steps of 0.05, starting from 0.5. It is clear that the error is much smaller for the same value of p, but in fact if we look at $p - q$ then the setup $p = 0.7$, $q = 0.3$ from the previous run shows slightly better detection than we see in this run with parameters $p = 0.5$, $q = 0.1$. Both parameter choices have $p - q = 0.4$ though. When the interval between p and q is shifted around from its centered position around 0.5, then the detection seems to become more difficult for the same $p - q$.

Finally, in figure (7) you can see $false_k^-$ and $false_k^+$ for different values of k, namely $k \in \{2, 4, 7\}$. The ratio of elements that do not belong to clusters, r_{bad}, is 0.4 for all following choices of parameters. Again, as in the first series, $q = 1 - p$, and p runs from 0.05 to 1.0 in steps of 0.05. The size of the problem, $n = 60k$, depends on k, thus making the problem size of the $k = 7$ runs equal to 420.

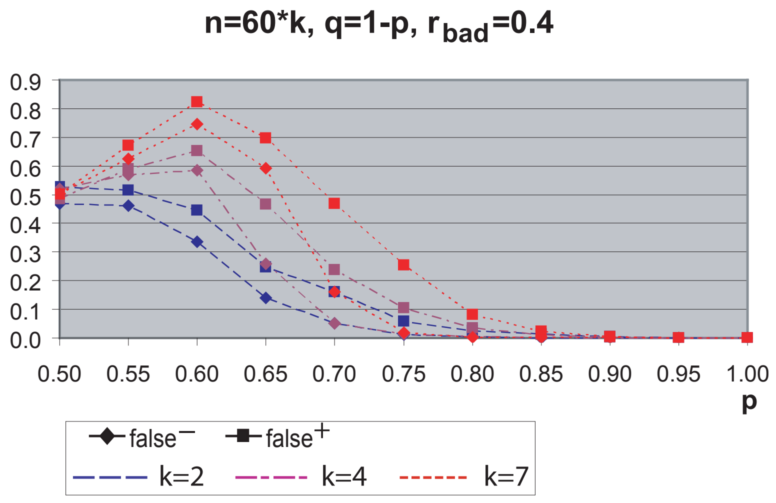

Figure 7: The x-axis shows p, the probability that two elements from the same cluster are grouped together by the similarity matrix. The y-axis shows the normalized error. The probability q is fixed to 0.1.

In the plot one can see that, in general, an increasing k makes the detection harder. That seems to have something to do with the degeneration of the linear threshold as in figure (2).

6 Conclusion

One of the the main products of this work is the implementation of the two algorithms `ClusterHierarchical` and `SpectralReconstruct`. With their help one can do the described two-stage clustering of a set of elements that are characterized by the similarity matrix $\hat{A}$, only exploiting the spectral properties of the matrix, i.e. without knowing k, p, and q. The algorithm `SpectralReconstruct` is replaceable by any other partition reconstruction algorithm. The focus of the experimental work lay clearly on the evaluation of the algorithm `ClusterHierarchical`, with the help of which one can sort out elements that do not belong to any cluster.

Concerning the algorithm `ClusterHierarchical`, it is the threshold value T that is of special interest. The goal was to find out if and how the threshold depends on the parameters k, p and q. Also, it is of very practical interest for what ranges of the parameters the method works. This work proposes a threshold that is a linear function of k. As for the dependence of p and q we could not find evidence for any. However, the experiments show the ranges for which the algorithm correctly detects the elements that should be subjected to the second stage of the clustering process. Obviously if p gets too small and q to large, or alternatively if the difference of p and q gets too small, then the detection collapses.
It has to be mentioned that it is not clear yet if the results from the section about spectral properties apply to the values for n that we used in the experiments. All the results are asymptotical results for $n \to \infty$, so it is questionable if we chose a large enough n. However, the results are satisfactory in the sense that we managed to detect the elements that belong to clusters for a reasonable range of k, p, and q. Matrices of moderate size ($200 \geq n \geq 500$) could typically be calculated in the order of minutes, on a 3.0 GHz Pentium 4, 2GB RAM, personal computer. The largest matrices ($1000 \geq n \geq 1200$) that we computed took about half an hour per single matrix, including the random generation of the matrix.

As for the algorithm `SpectralReconstruct`, no series of experiments were conducted, since the computation of the maximum matching that is needed for the measurement of error is not yet implemented. The algorithm depends on two thresholds that are also derived from asymptotical considerations, therefore it is not clear to what value they should be set exactly to minimize the error of the planted partition reconstruction. Manual adjustment of the thresholds showed that it is indeed possible to achieve a flawless reconstruction of the clusters.

7 Outlook

As already mentioned the next step could be to implement the maximum matching used to measure the error of the reconstruction. Then the two thresholds of `SpectralReconstruct` could be scrutinized in order to come up with practical values that can be substituted for the asymptotical thresholds. Only then will it be possible to assess the whole two-stage clustering process as described in the introduction.

A Source Code

In this appendix we have included some of the source code of the FASD program.

Global Variables. First of all, we include the definitions of some global variables, that will be needed in the following code fragments.

```
int size = 0; // size of the problem
int nec = 0; // number of eigenvalues needed
double **mat; // variable to store the matrix
double *eval, *evec, *wrk; // workspace for trlan
short *marker; // result of trlan


int NROW = 0;
int LOHI = 1; //computes the 'large' eigenpairs
int MAXLAN = 0;
int MEV = 0;

int k = 0; //number of clusters
float p = 0.9f;
float q = 0.1f;
```

Matrix-Vector Multiplication. The following routine had to be implemented to pass it to TRLAN as an argument. TRLAN requires the caller to provide his or her own multiplication routine that implements the multiplication of the stored matrix A with one or more vectors. The parameters are *nRow*, the number of rows of A, *nCol* the number of columns. *xIn* and *yOut* are two one dimensional arrays that are used to store the vectors to multiply or the resulting vectors respectively. *ldx* and *ldy* are the number if elements in the vectors.

```
///////////////////////////////////////////////////////////
void mult_vectors(int *nRow, int *nCol, double *xIn,
                        int *ldx, double *yOut, int *ldy) {
///////////////////////////////////////////////////////////
    int i, j, k;
    double  *xTmp, *yTmp;

  for (i=0; i<*nCol; ++i) {
```

```
  yTmp = yOut + *ldy * i;
    for (j=0; j<*nRow; ++j, ++yTmp){
      xTmp = xIn + *ldx * i;
      *yTmp = 0.0f;
      for (k=0; k<*nRow; ++k, ++xTmp){
        *yTmp += (*xTmp) * ((double)mat[j][k]);
      }
    }
  }
}
```

Initialization. The following routine initializes the workspace and sets the maximal relative error that is allowed for the computation of the eigenpairs. This routine is always called once at the beginning of the eigenvalue computation.

```
///////////////////////////////////////////////////////////
void init_TRLan(){
///////////////////////////////////////////////////////////
  int i;

    for (i=0; i<MEV; ++i) eval[i]=0.0;
    for (i=0; i<NROW; ++i) evec[i] = 1.0;
    /* relative tolerance on residual norms */
    wrk[0] = 1.4901E-8;
}
```

Stepwise Calculation. The following routine is needed to make use of the converging computation of TRLAN. Instead of computing all eigenpairs from start, one can define how many eigenpairs are wanted, and also one can gradually extend the result to more eigenvalues by leaving the workspace unchanged and call TRLAN again with a request for more eigenvalues. This is done by this routine. *ned* specifies the number of eigenpairs that should be computed, initially set to one. Then, *ned* is increased step after step until the absolute value of the next computed eigenvalue becomes too small.
call_TRLan(ned, print) is the call that actually calls the Fortran function. *print* is a integer flag that indicates if output to the standard output should be done or not.

```
///////////////////////////////////////////////////////////
```

```
void call_TRLan_gradually(int print){
///////////////////////////////////////////////////////////
  int ned = 1;
  int tmpK = 1;
  float cmp = sqrt(size);

  nec = 0;

  nec = call_TRLan(ned, print);
  if (print) {
  printf("%f > nec %d\n", eval[nec - ned - 1], nec);
  }
  while ((ned <= size) && (nec > 0)
    && (eval[nec - ned - 1] > cmp)) {
    tmpK++;
    ned++;
    nec = call_TRLan(ned, print);
  if (print) {
    printf("k:::::::::::::::%d\n", tmpK);
    printf("%f > nec %d\n", eval[nec - ned - 1], nec);
  }
  }
  k = tmpK;
  if (print) {
  printf("k:::::::end::::::%d\n", k);
  }
}
```

Interfacing TRLAN. The following function does the call to the Fortran library. *ipar* is an array of integers to pass all the arguments to the function call. For an explanation of all the parameters please refer to the user manual of TRLAN.

```
///////////////////////////////////////////////////////////
int call_TRLan(int ned, int print){
///////////////////////////////////////////////////////////

  if (print) {
    printf("\n****************************************\n");
    printf("** TRLan\n");
  printf("***************************************\n");
  printf("Matrix loaded: %d x %d\n", size, size);
  printf("Starting eigenvalue decomposition...\n\n");
}

int i, j, lwrk, nrow, mev, ipar[32];
  double tt, to, tp, tr;


  nrow = NROW; mev = MEV;
  lwrk = MAXLAN*(MAXLAN+10);
  ipar[0] = 0;
  ipar[1] = LOHI;
  ipar[2] = ned;
  ipar[3] = nec; // nec = 0
  ipar[4] = MAXLAN;
  ipar[5] = 1; // restarting scheme 1
  ipar[6] = 2000; // maximum number of MATVECs
  ipar[7] = 0; // MPI_COMM
  ipar[8] = 10; // verboseness
  //Fortran IO unit number for log messages
  ipar[9] = 99;
  // iguess -- 1 = supply the starting vector
  ipar[10] = 1;
  // checkpointing flag -- write about -5 times
  ipar[11] = -5;
  // Fortran IO unit number for checkpoint files
  ipar[12] = 98;
  // floating-point operations per MATVEC per PE
  ipar[13] = 3*NROW;


  /* call Fortran function TRLAN77 */
  F_TRLAN(mult_vectors, ipar, &nrow, &mev,
        eval, evec, &nrow, wrk, &lwrk);


if (print){
  printf("\nThe eigenvalues:\n");
  for (i=0; i<ipar[3]; i++){
      printf("E(%i) = %25.17g %16.4e\n", i+1,
              eval[i], wrk[i]);
    }
    // output the k biggest eigenvectors
    printf("\nThe %d biggest eigenvectors:\n", ned);
    for (j = 0; j < size; ++j){
      for (i=ipar[3]-1; i>ipar[3]-ned-1; --i){
        printf("%f ", evec[i*size+j]);
      }
      printf("\n");
    }
    printf("**************************************\n");
    printf("** End of TRLan\n");
    printf("**************************************\n\n");
  }

  return ipar[3]; //return nec

}
```

References

[1] Z. Füredi and J. Komlós. The eigenvalues of random symmetric matrices. *Combinatorica I*, 3:233–241, 1981.

[2] M. Krivelevich and V. H. Vu. On the concentration of eigenvalues of random symmetric matrices. *Microsoft Technical Report*, 60, 2000.

[3] F. McSherry. Spectral partitioning of random graphs. *Proceedings of 42nd IEEE Symosium on Foundations of Computer Science*, pages 529–537, 2001.

[4] M. Meila and D. Verma. A comparison of spectral clustering algorithms. *UW CSE Technical report 03-05-01.*