Geometric Learning – Homework 2

Student1 name: Noam Shilony  
Student1 number:

Student2 name: Udi Gal  
Student2 number: 316564913  
  
Due date: 20/12/2020

**Question 1:**

1. We want to show that for any , that corresponding eigenvectors are orthogonal, meaning . To prove that, we want to show that . We can now write:  
     
   And for we get:
2. In order to prove the statement, let us start by proving that the permutation matrix is orthogonal, meaning . We can show it directly:   
     
   matrix multiplication associativity and the definition to conclude:
3. We can see that:  
      
   Meaning that x can be expressed as:  
    for some   
   And so for an **orthogonal** set of eigenvectors we get:  
    for some   
     
   We can then solve:  
    , we can write:  
    , and therefore:
4. Consider a matrix with as a principal submatrix.

Let   be a permutation of Rows and Columns of M (), therefore   has the same eigenvalues.

Let be the eigenvectors of and be the eigenvectors of H.

and

Let's define

As a result of the dimension theorem, the sum thus, we must have s.t . We may then write:

For the second inequality, let us define:

and

And in a similar way:

Which concludes our theorem.

1. We consider with and another with

Let us start with the (b):

**Throughout the following proof I'll use the notation , observing this operator associative respective to the matrix M.**

From Courant-Fischer min-max theorem we conclude:

Let s.t and

Now let , according to the dimension theorem:

Therefore, the maximum on holds for,

Using similar steps for (a):

From Courant-Fischer min-max theorem we conclude:

Let s.t and

Now let , according to the dimension theorem:

Therefore, the maximum on holds for,

**Question 2:**

1. The adjacency matrix .

We assume the corresponding graph is connected and with eigenvectors

1. , The biggest eigenvalue, can be expresses as . Let us define be a non-negative vector. Keep note that .

According to Rayleigh theorem, maximizes the quotient and it is therefore the eigenvector corresponding to , and must satisfy . Therefore we infer that

Let us prove that the inequality is strict. Assume that has one or more 0 valued element, it must then have at least one element of positive value as well, since is not a valid eigenvector.

There must then be at least one edge for which , for the graph to be connected (if both 0 and positive valued vertices exist, and such an edge does not exist, then it must mean that the vertices are separated into two groups of only 0 valued and only positive valued, contradicting the connectivity assumption).  
 Such vertex ‘i’ must then satisfy , however:   
Which contradicts the assumption that has a 0 valued element.  
Therefore

1. Let's take notice that is negative, otherwise the inequality is trivial.  
   Let us define be a non-negative vector. Keep note that .

Where the first inequality is the generalized triangle inequality; and the second inequality is due to the fact that maximization over a larger domain must satisfy an equal or larger maximum.

1. Let be the corresponding eigenvector to . The eigenvectors are orthogonal, meaning . Since we proved on section 2.1(a) that ( has strictly positive elements), it means that must then have at least one positive valued, and at least one negative valued element, in order for that two vectors to be orthogonal.

There must then be at least one edge for which .  
  
Let us define , be a non-negative vector. Keep note that .

Where the strict inequality is due to for an edge whose elements are of negative values, and otherwise; and the second inequality is due to the fact that maximization over a larger domain must satisfy an equal or larger maximum.

1. In order to prove that the Graph Laplacian (L) is PSD, we need to show that it has to have 0 as its **minimal** eigenvalue (). We can show that using the Courant-Fischer theorem:

It is easy to see that the last inequality is achieved (and thus ) when .  
The vector achieves the last inequality, and thus and the **Graph Laplacian is PSD**

1. Considering the two graphs and where , and note ; with , the eigenvalues of the Laplacians, , respectively.

We know that as for the adjacency matrices, where the '1' are in the and locations.   
  
Let's take notice that where again, the '1' are in the p and k vertices index.

Therefore expressing the Laplacians , when .

Observe that is of rank 1 and has eigenvalue 2 of multiplicity 1 and eigenvalue 0 of multiplicity.

Choosing in Courant-Weyl inequality, in 5(b):

Choosing in 5(a), we get:

The smallest eigenvalue of a Laplacian is always 0 therefore .

1. The eigenvalues of the Laplacian of a path graph are with multiplicity 1 (. The eigenvalues of the ring graph are  
    with multiplicity 2 (.  
     
    Of course, is an edge subtracting version of . **We have to consider the sorted eigenvalues of both Laplacians**.

Let ,

hence, it's easy to see that the eigenvalues are interlacing alternately.

**Question 3:**

**(Code in files Q3.1.py + Q3.2.py + Q3.3.py)**

1. **Code in file Q3.1.py**
2. For Pn:  
   Adjacency matrix is of the form:  
      
     
   An nxn Toeplitz matrix with 1st superdiagonal and 1st subdiagonal of value 1. Rest of the element are 0 valued.

Degree matrix is of the form:

A diagonal matrix with first and last diagonal element equal 1, and the rest are 2.

Graph Laplacian matrix is of the form:

Simply equals .

For Rn:  
Adjacency matrix is of the form:  
   
An nxn Toeplitz matrix with its 1st and (n-1) superdiagonals , and 1st and (n-1) subdiagonals of value 1. Rest of the element are 0 valued.

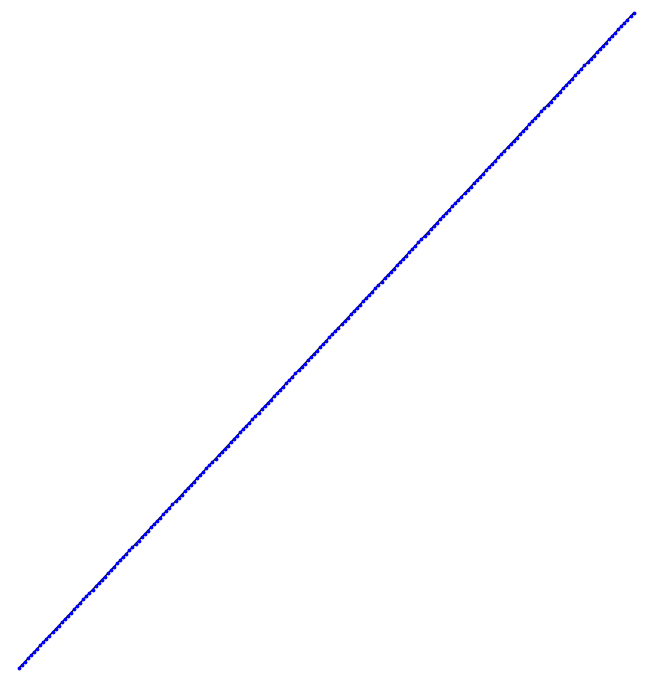
Degree matrix is of the form:

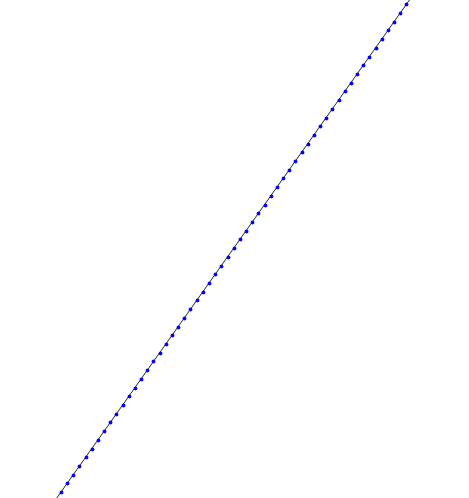
A diagonal matrix with elements of value 1.

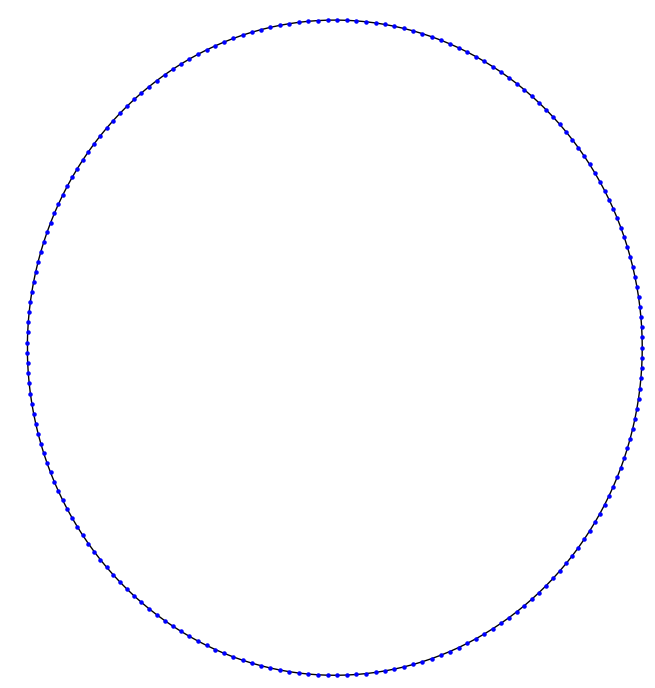
Graph Laplacian matrix is of the form:

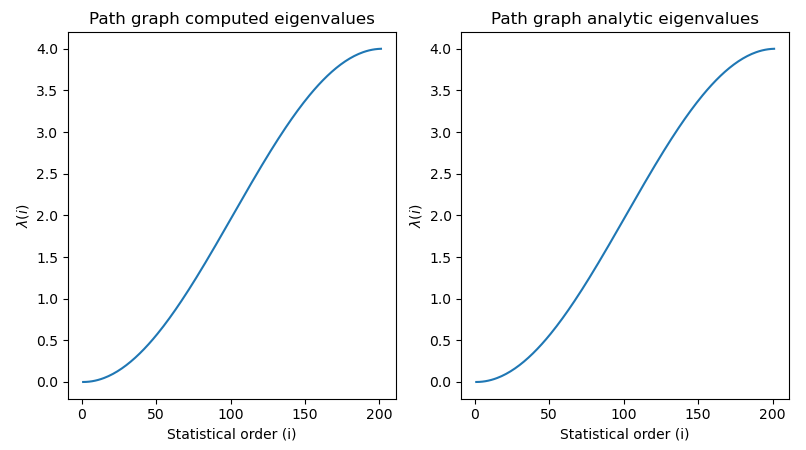
Simply equals .

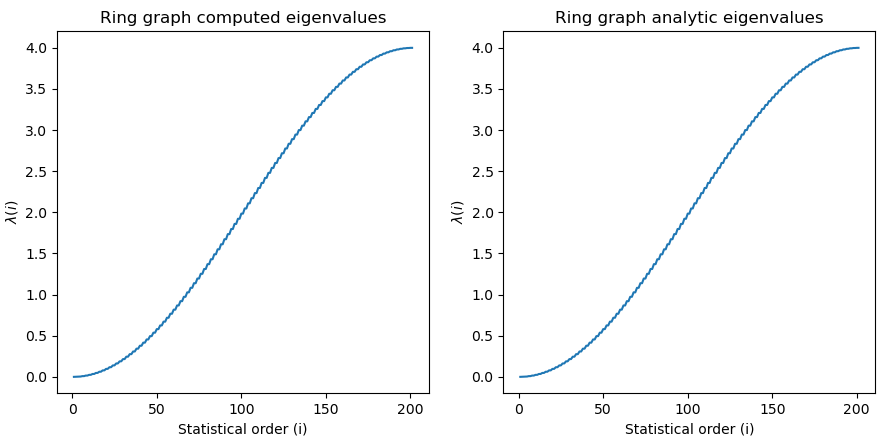
1. In code.
2. Plots:
3. Graph topology of P­201, presented as a line of nodes:

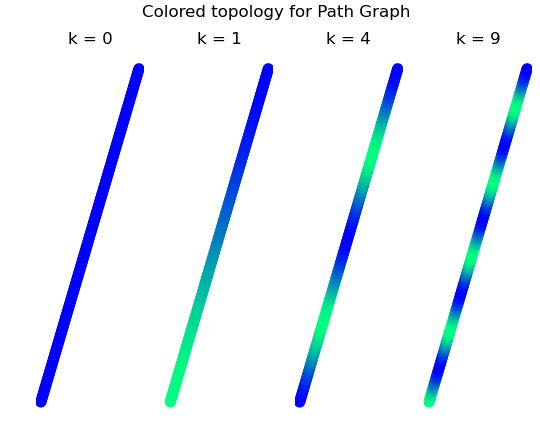


Close up, for the sake of visibility:  


Graph topology of R­201, presented as a circle of nodes:  


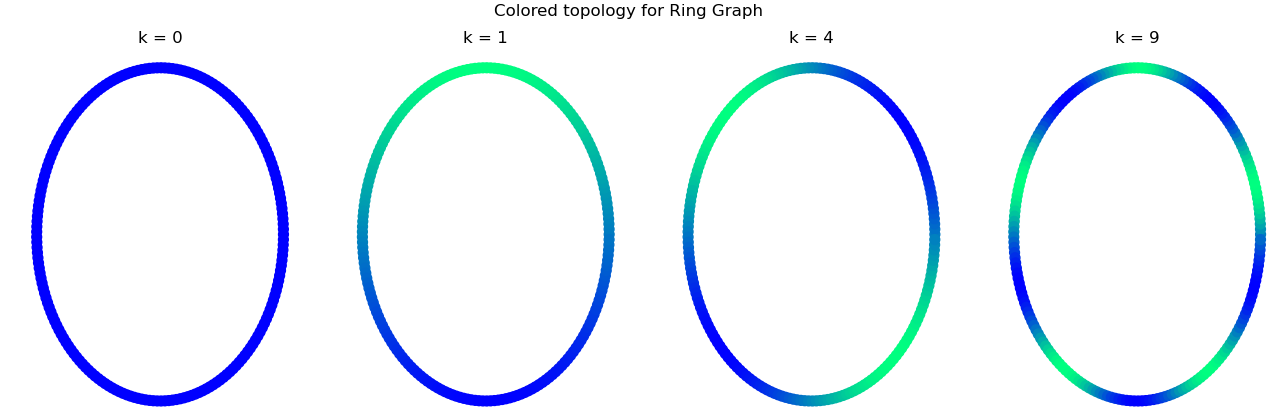
1. Computed and analytic eigenvalues of P201 as a function of their statistical order:  
   

Computed and analytic eigenvalues of R201 as a function of their statistical order:  


1. P201 colored by its first, second, fifth and tenth (K=0,1,4,9) eigenvectors:  
     
   

When compared with the analytic eigenvectors:

We can tell that as ‘k’ increases, the eigenvector takes the form of a cosine function of a higher frequency (Specifically, for , we get a vector of 1s – frequency = 0) .   
This cosine form of ‘k’ dependent frequency can be observed clearly in our plot.   
We can also see that the numbers of cycles match k/2.  
  
 R201 colored by its first, second, fifth and tenth (K=0,1,4,9) eigenvectors:

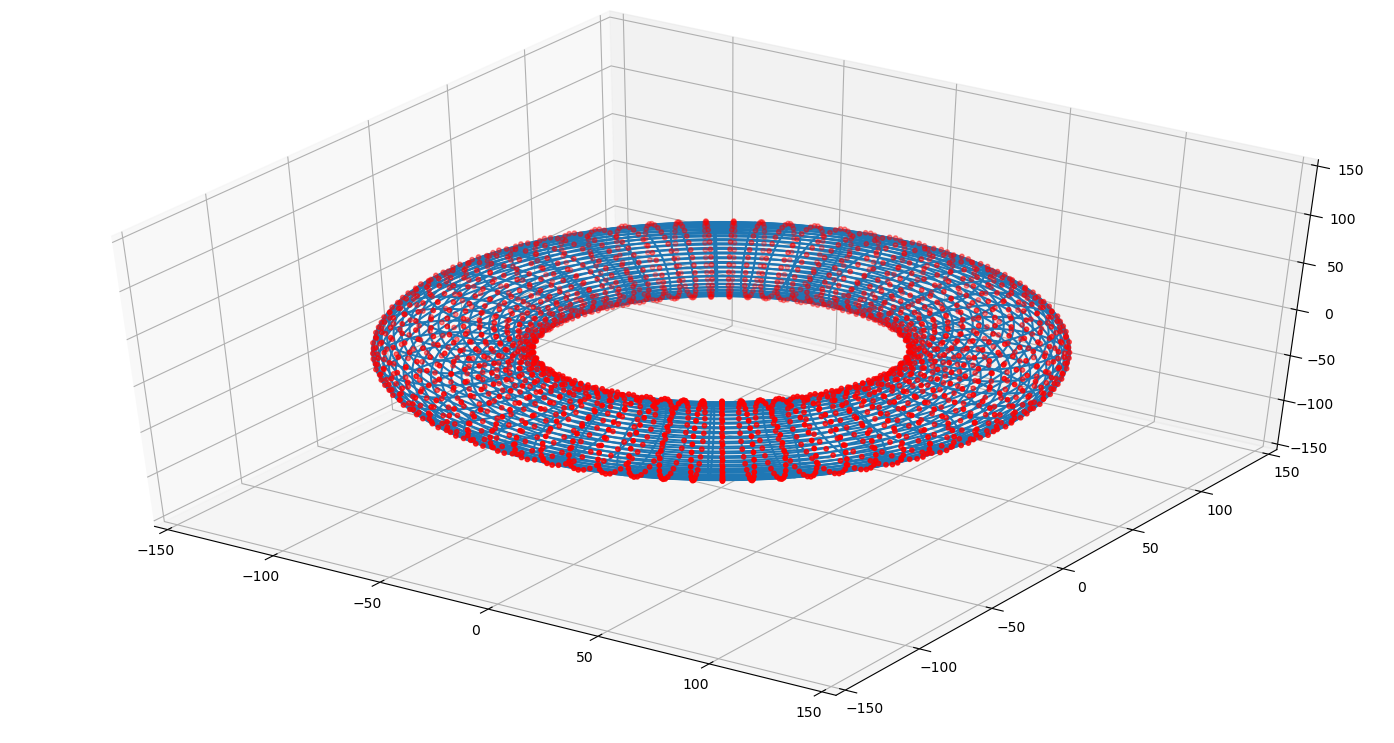


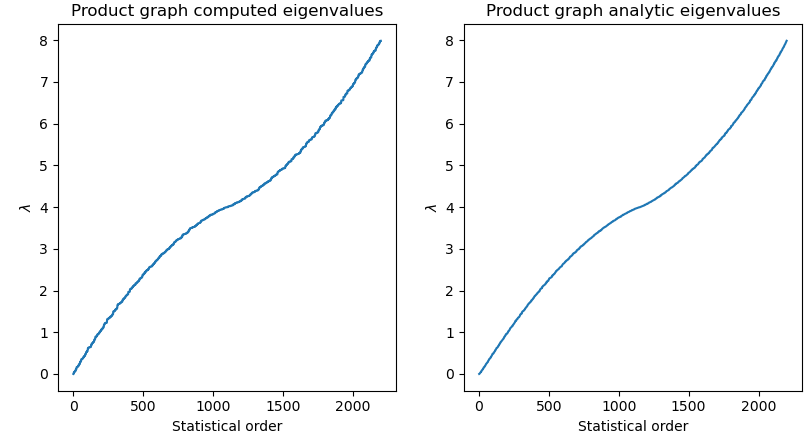
When compared with the analytic eigenvectors:

We can tell that as ‘k’ increases, the eigenvector takes the form of a cosine\sine function (alternately) of a higher frequency. Specifically:  
for , we get a vector of 1s corresponding to ,  
for 1, we get a sine vector corresponding to ,  
for , we get a cosine vector corresponding to ,  
for , we get a sine vector corresponding to ,).  
We can see in the plots that the sine eigenvectors indeed have a phase-shift compared to the cosine ones, and that the numbers of cycles match the x/y indexes.

1. When comparing the computed and analytical eigenvalues (shown in previous figures), we can see that the received values behave the same way, and take a cosine shape bounded by 0 from below and 4 from above.
2. **Code in file Q3.2.py**
3. A possible topology for such product graph could be a torus. Specifically, we chose a torus made up by 71 azimuthal circles (orthogonal to xy plane), each made up by 31 evenly spaced nodes.
4. The adjacency matrix is a block Toeplitz matrix ( with its block diagonal being blocks, and its 1st and (nx-1) block superdiagonals, and 1st and (nx-1) block subdiagonals being blocks. The rest of the blocks are matrices.  
     
   The degree matrix is a diagonal matrix with elements of value 4.

The graph Laplacian matrix simply equals .

1. In code.
2. Plots:
3. The graph topology of G­71,31:   
   The nodes are colored in red, and the edges are colored in blue. The topology formed is a torus with main radius R=100, and secondary radius r=30.
4. Computed and analytic eigenvalues of G­71,31 as a function of their statistical order:



1. TODO

**Question 4:**

**(Code in files in powermethod.py + Q4.2.py)**

2. Since p, π are probability distribution vectors, .   
   Therefore, where d is the vertex degree vector:

And we are left to show that the inequality is strict.  
  
For a connected and undirected graph, notice that:

Now since p is a probability distribution vector, we can say that:  
And for such a vertex:

1. We know that:

If some exists such that , we can infer that:

For a connected and undirected graph, such stable distribution is unique, and can be calculated as . However, for a non-connected graph, the probability to be in each of the connected components (assuming a general initial distribution, not necessarily ), remains the same as the initial probability. More formally, for connected components , for some component ‘i’:

And we can infer that for a non-connected graph, the stable distribution is unique within each component; and the stable distribution of the whole graph in dependent on the initial distribution of probability between the components.

1. Firstly, we can easily show that the walk matrix WG is column stochastic (each column’s sum is 1):

G as and the corresponding eigenvectors as . Now, using the column-stochastic property of WG, We can show, using the L1-norm ( ) we can see that for any eigenvector :  
Where (A) is the explicit multiplication under L1-norm, (B) is the generalized triangle inequality, (C) is due to the elements of WG being non-negative, and (D) is due to WG being column stochastic.  
  
We can now see that for any eigenvector :

We can infer that if an eigenvector whose corresponding eigenvalue is exists, then , and are the dominant eigenpair. since we know the stable distribution π achieves , and is such eigenvector, we can conclude that it is indeed the dominant eigenvector.

1. **Code implementation in powermethod.py + Q4.2.py**
2. Let us show the claim:
3. We can observe the following:

Now, we can infer:

1. Let us start by expressing a single update:

And likewise, for the next update:

Following this update rule for t iterations, we can infer that:

1. We can expand the matrix B and show the following:
2. Let us assume that the dominant eigenvalue is of multiplicity 1 ( ) and that the dominant eigenvector is normalized ( ).   
   These entail that .  
   Also, not that by the update rule is normalized , therefore   
   We can then infer that:

Therefore we get .  
  
Now as for the other term:

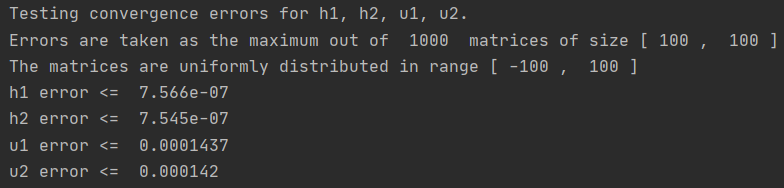
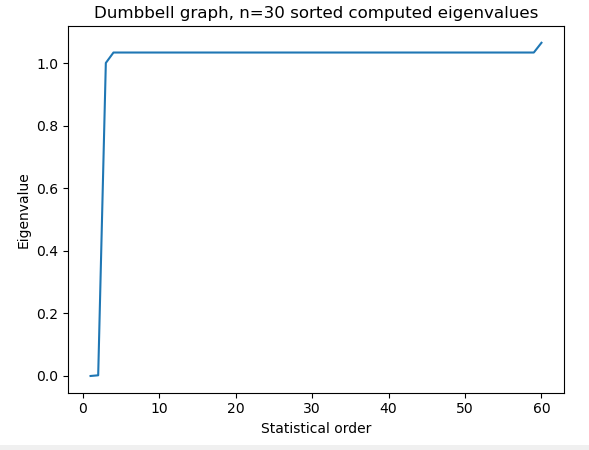
Therefore we get .

1. Let us justify the conditions for applying the power method on the random walk matrix .

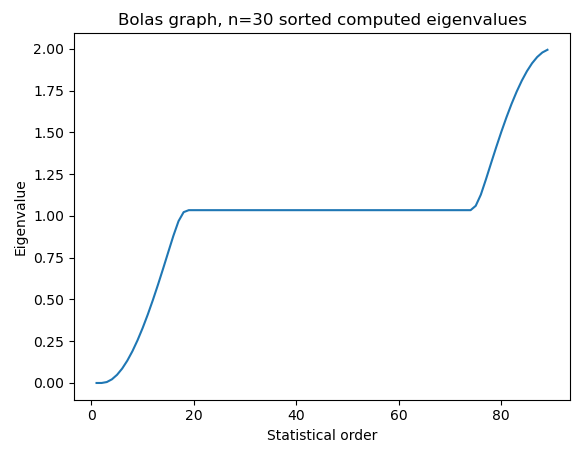
is similar to a symmetric matrix, and can be written as . Since a real symmetric matrix is diagonalizable, matrices H, V’ which satisfy   
 exist. We can then see that:  
   
Therefore, WG is diagonalizable as well, and we can then apply the method, where WG is analogical to B.  
We choose an arbitrary initial probability distribution p­0 (analogical to u0) such that (since the matrix V is of full rank).

In the general power method algorithm, the normalization of u on each step is required since:

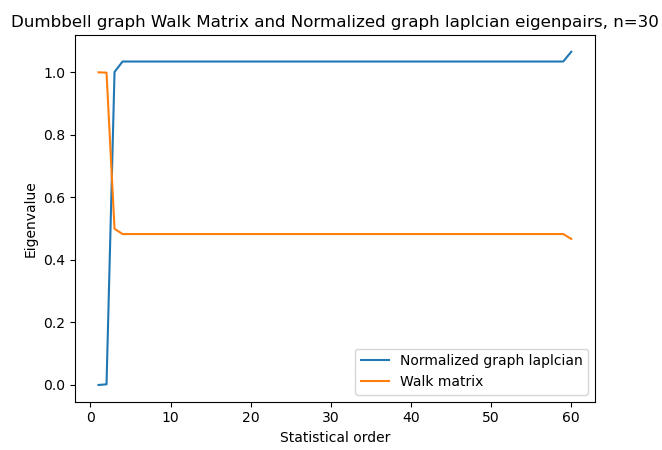
And since:   
If we didn’t normalize:  
And when , the algorithm won’t converge correctly if   
Since we proved in the lectures that the eigenvalues of WG satisfy , the element , and the algorithm converges regardless, therefore the normalization is not necessary.

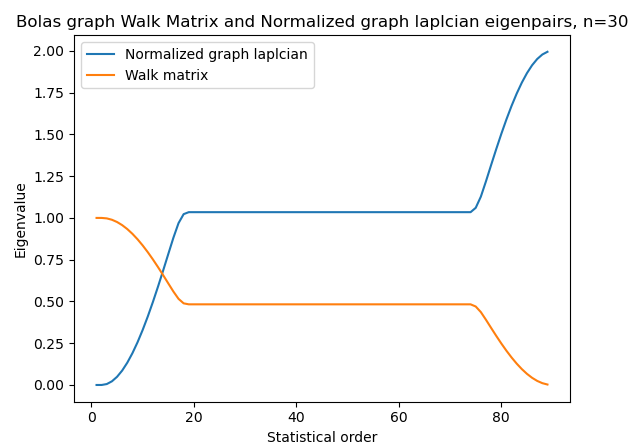
1. Implemented in powermethod.PowerMethod  
   The reason why the stopping criterion is not appropriate is because for the vector might alternate between for , and we won’t achieve the criterion. We chose a more appropriate criterion based on the convergence to the eigenvalue: .
2. Implemented in powermethod.PowerMethod2
3. Implemented in Q4.2.py  
   If we denote the computed values as ; And the eigen decomposition results as ; We can express the errors as:  
   The testing scale and results are as follows:  
     
   We can tell the results are fairly successful, as the errors are consistently small.
4. **Code implementation in Q4.3.py**
5. Computed in code.  
   Dn  is a graph with 2n vertices (=60 for n=30).  
   Bn is a graph with 3n-1 vertices (=89 for n=30).
6. Sorted eigenvalues for Normalized graph Laplacian of D30:   
   

We can see that the largest eigenvalue is not bounded by 2 for the Dumbell.

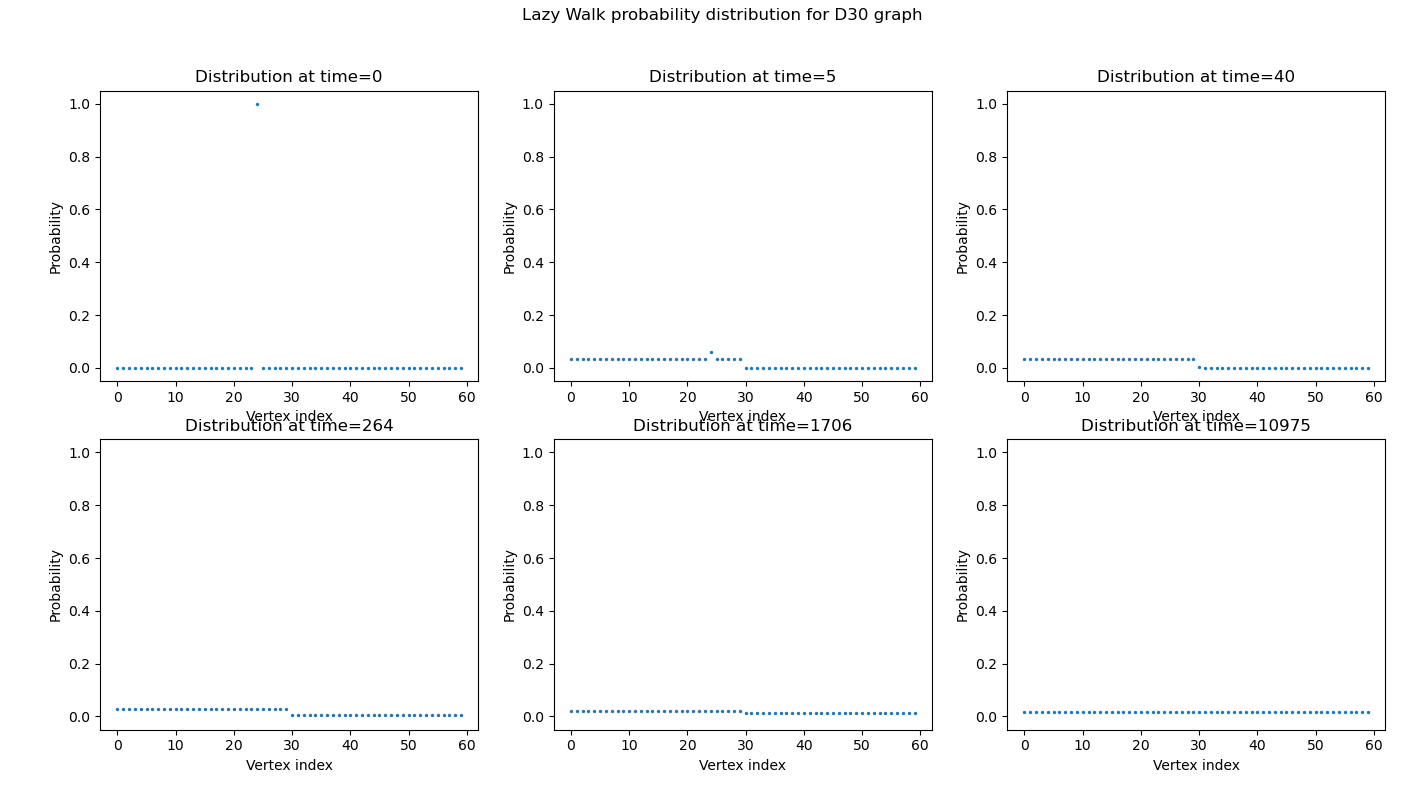
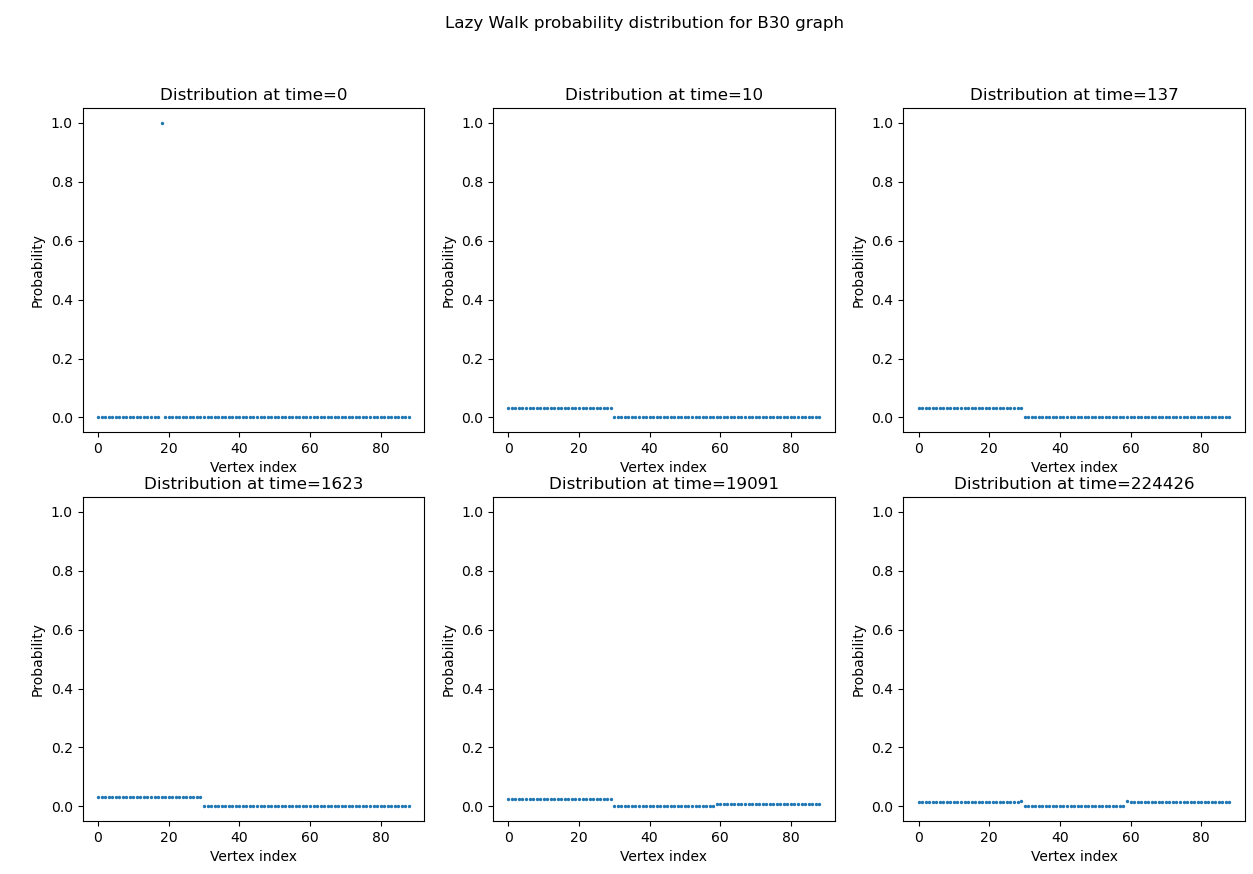
Sorted eigenvalues for Normalized graph Laplacian of B30:   


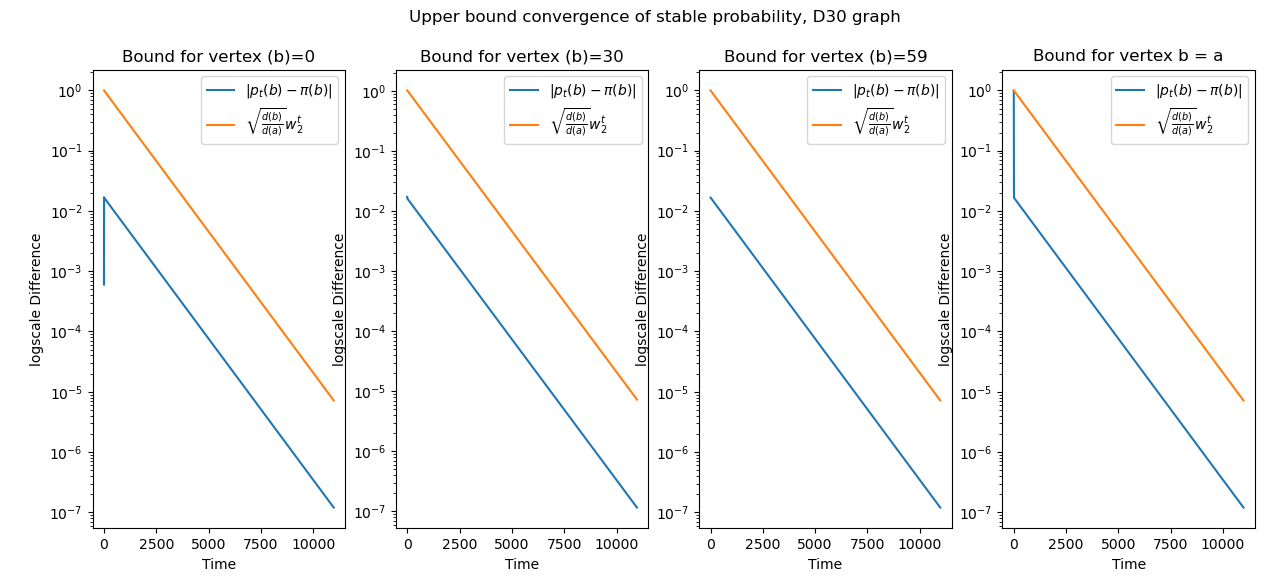
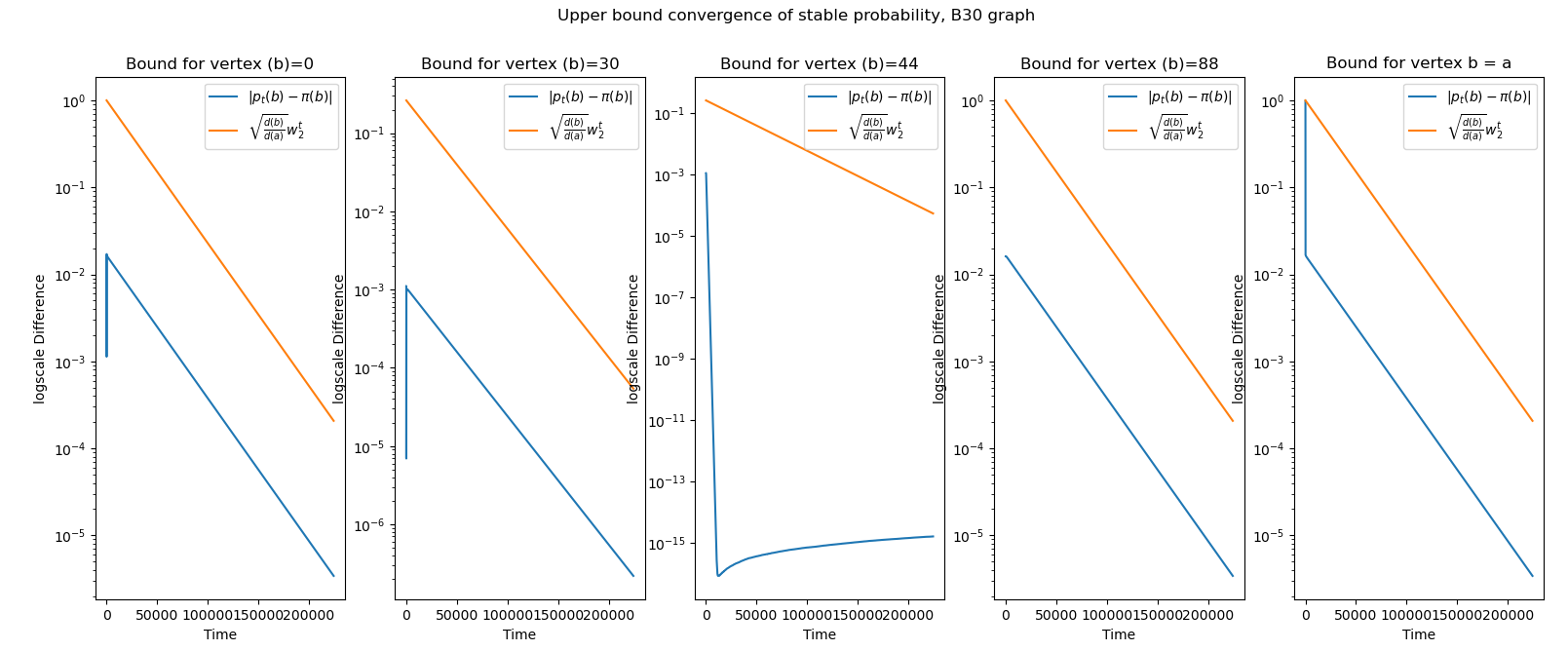
We can see that the largest eigenvalue is tightly bounded by 2 for the Bola graph, however a value of 2 is **not** achieved since the graph is not bipartite.

1. Sorted eigenvalues for Normalized graph Laplacian and Walk matrix of D30:  
     
     
     
   Sorted eigenvalues for Normalized graph Laplacian and Walk matrix of B30:



We can see that for both graphs, the eigenvalues of , and the eigenvalues of follow the relation shown in the recitation: .

1. Intermediate stated of pt for D30:   
   We can observe that when the initial probability is in the first cluster, the distribution of the cluster vertices evens out quickly, and after many iterations evens out with the second cluster as well.  
     
    Intermediate stated of pt for B30:   
     
   We can notice a similar pattern as in the dumbbell case, however it takes far more time (about an order of magnitude) for the two clusters to even out, since the connectivity of the graph is low, due to the “neck” between the two clusters, making it far less likely to make it from one cluster to the other.

* Demonstrating the convergence bound for various vertices as a function of the time for D­­30:   
  
* b = 0 represents the first cluster.
* b = 30 represents the connecting vertices.
* b = 59 represents the second cluster.
* a is the initial vertex.  
    
  Demonstrating the convergence bound for various vertices as a function of the time for B­30:  
  
* b = 0 represents the first cluster.
* b = 30 represents the vertices connecting the cluster with the “neck”.
* b = 44 represents the “neck” part of the graph. We can notice that the bound decreases slower, since these vertices have d(b)=2, unlike the usual n-1 or n.
* b = 88 represents the second cluster.
* a is the initial vertex.

**Question 5:**

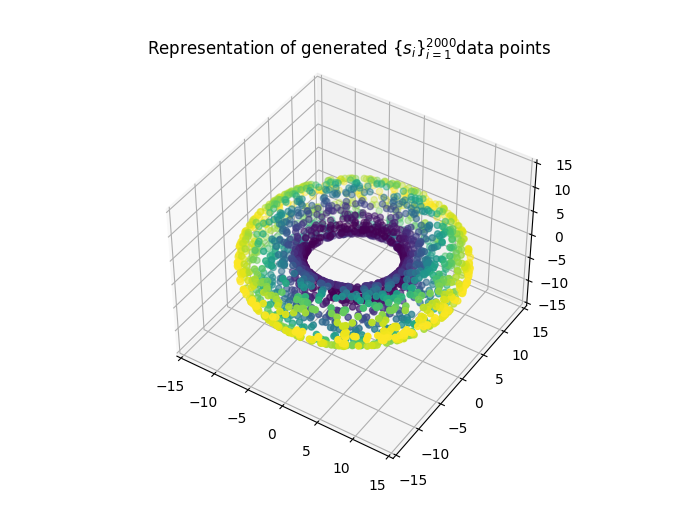
**(Code in**

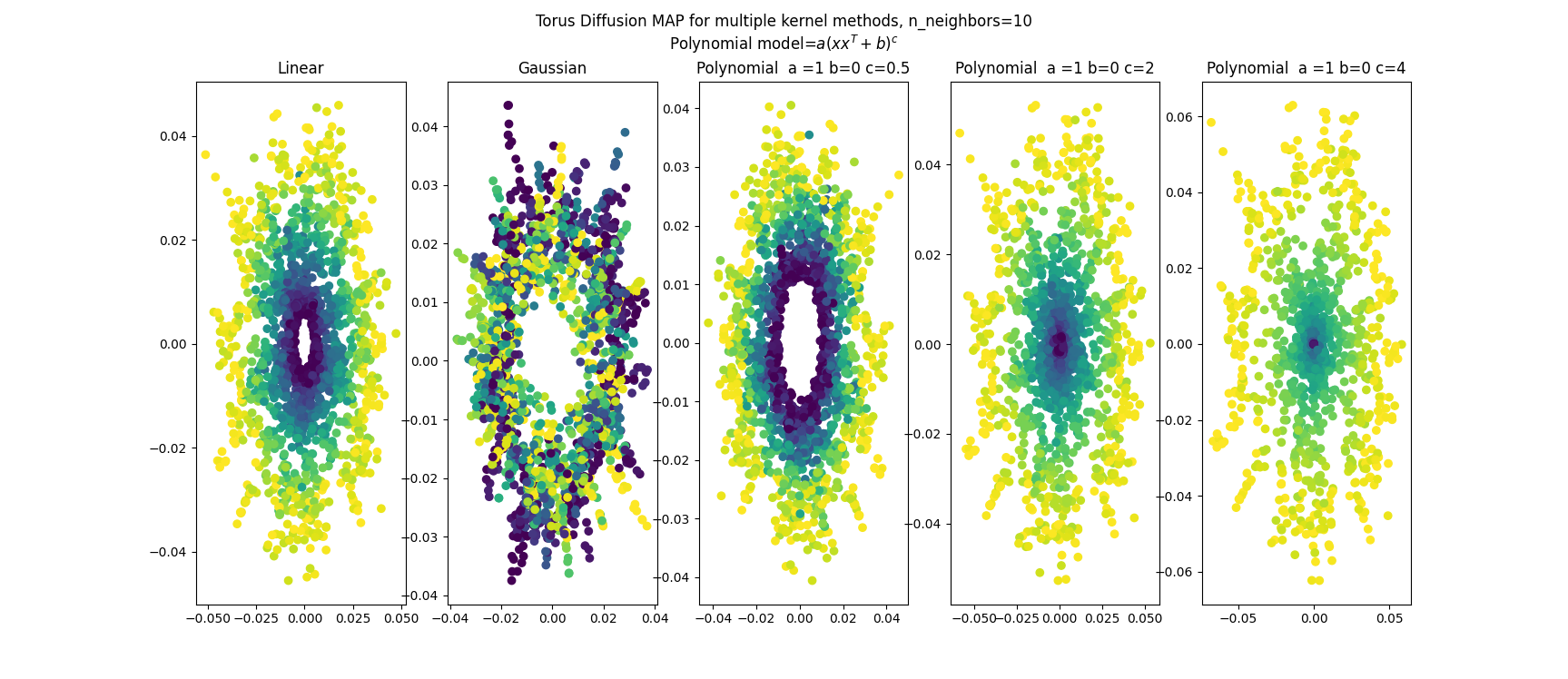
* + - 1. Implemented in   
         In addition to the standard arguments, our function receives:
* normalized – binary flag that tells the function to center and normalize the data before creating the affinity matrix.
* kwargs – in order to get parameters for the different kernel methods.

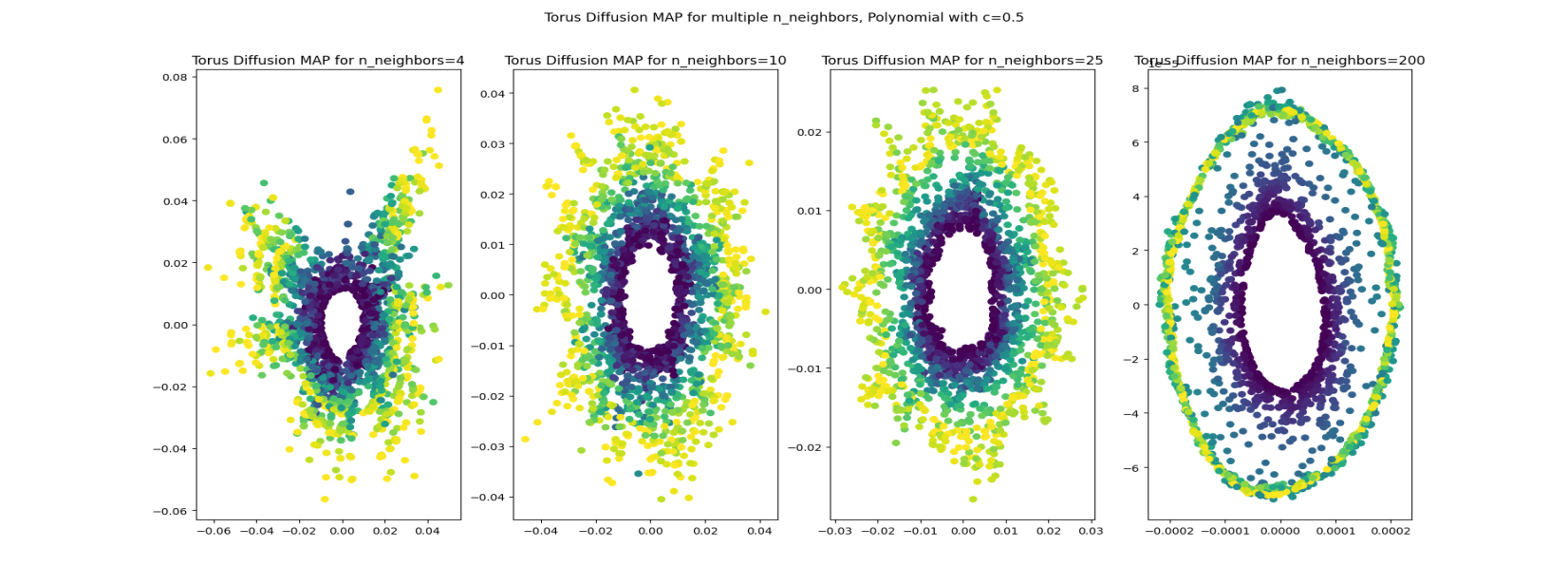
Our function supports 4 kernel methods:

* ‘Gaussian’ – as shown in the lecture
* ‘Linear’ –
* ‘Polynomial’ – , the additional parameters are achieved via kwargs.
* ‘Unit’ – Does not apply a kernel, return a KNN adjacency matrix.  
  + - 1. We chose to implement Diffusion\_Maps and Locally\_Linear\_Embedding.  
         For both reasons, the neighbors are chosen via KNN (with Euclidean distance). There is also an option of n\_neighbors = None, which implies that the graph is fully connected (weights based on kernel).  
           
         Diffusion\_Maps – supports all implemented kernels. A parameter t, which predetermines the number of iterations was added.  
          Locally\_Linear\_Embedding – Supports the ‘Unit’ kernel only.
      2. Apply:

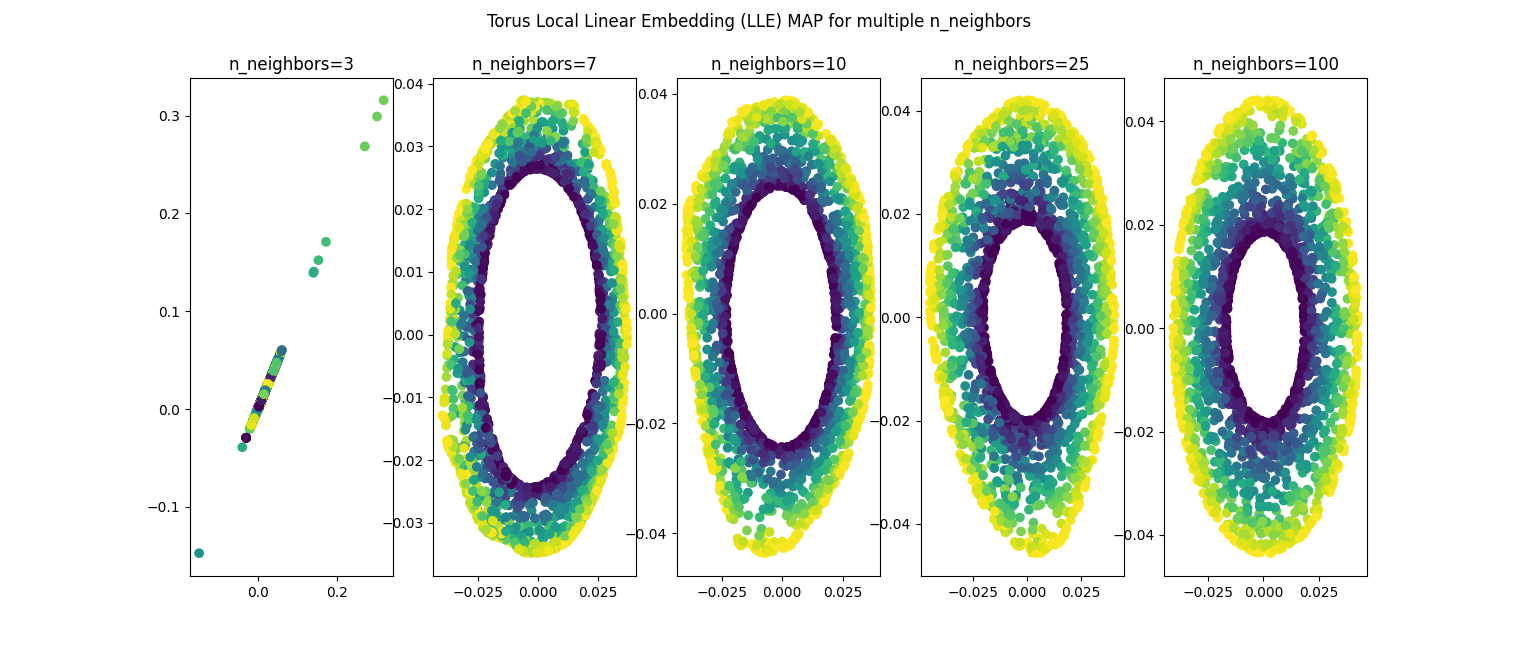
1. **Torus:**

Generating the following points of a torus:  


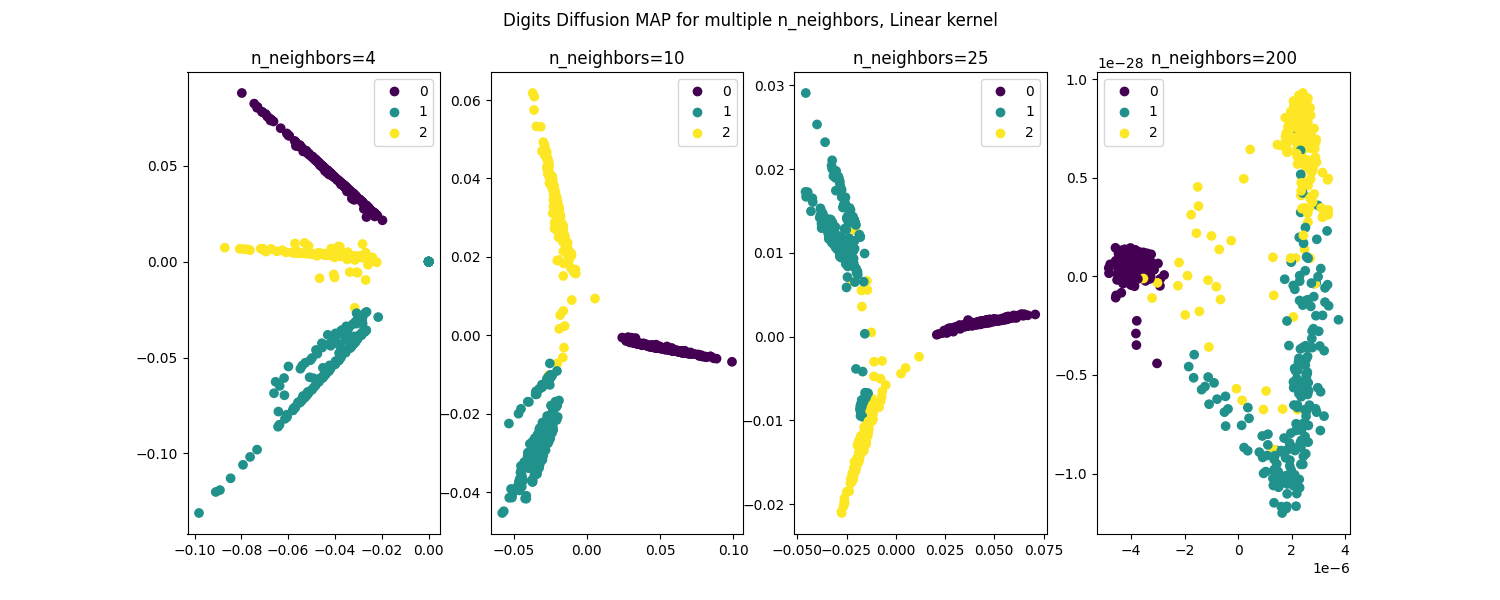
We will try to project the 3D torus onto a 2D plane. The vertices were colored by their radius.  
  
**Let us apply Diffusion\_Maps:**  
  
  
  
  
  
  
  
  
  
  
Firstly, we test the different kernels with n\_neighbors = 10:  
The best results were achieved for Linear and polynomial with (a,b,c) = (1,0,0.5) ( ).  
We decided to stick to the latter variation for the following test.  
  
Next, we can test the method for different values of n\_neighbors:

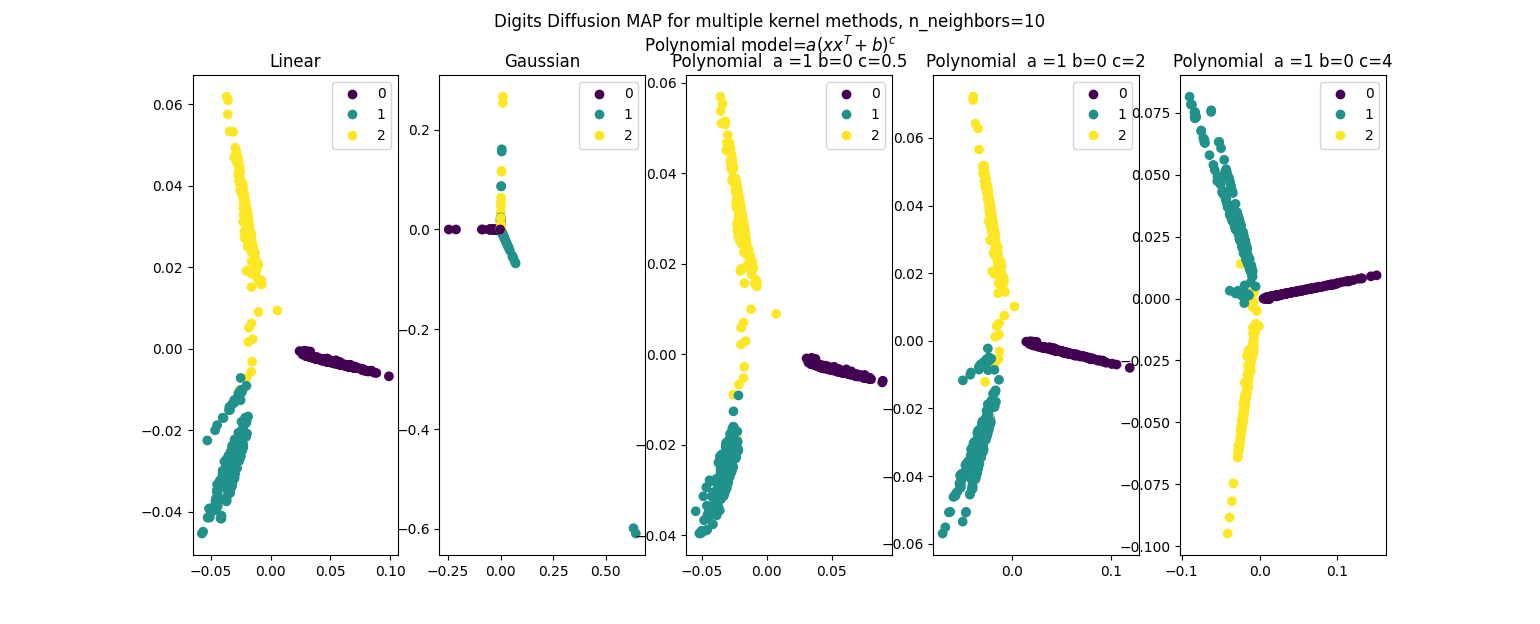
We can notice that for low number on neighbors, all the nodes are drawn towards the middle, and for high number of neighbors, the higher radius nodes are drawn towards the outer radius.

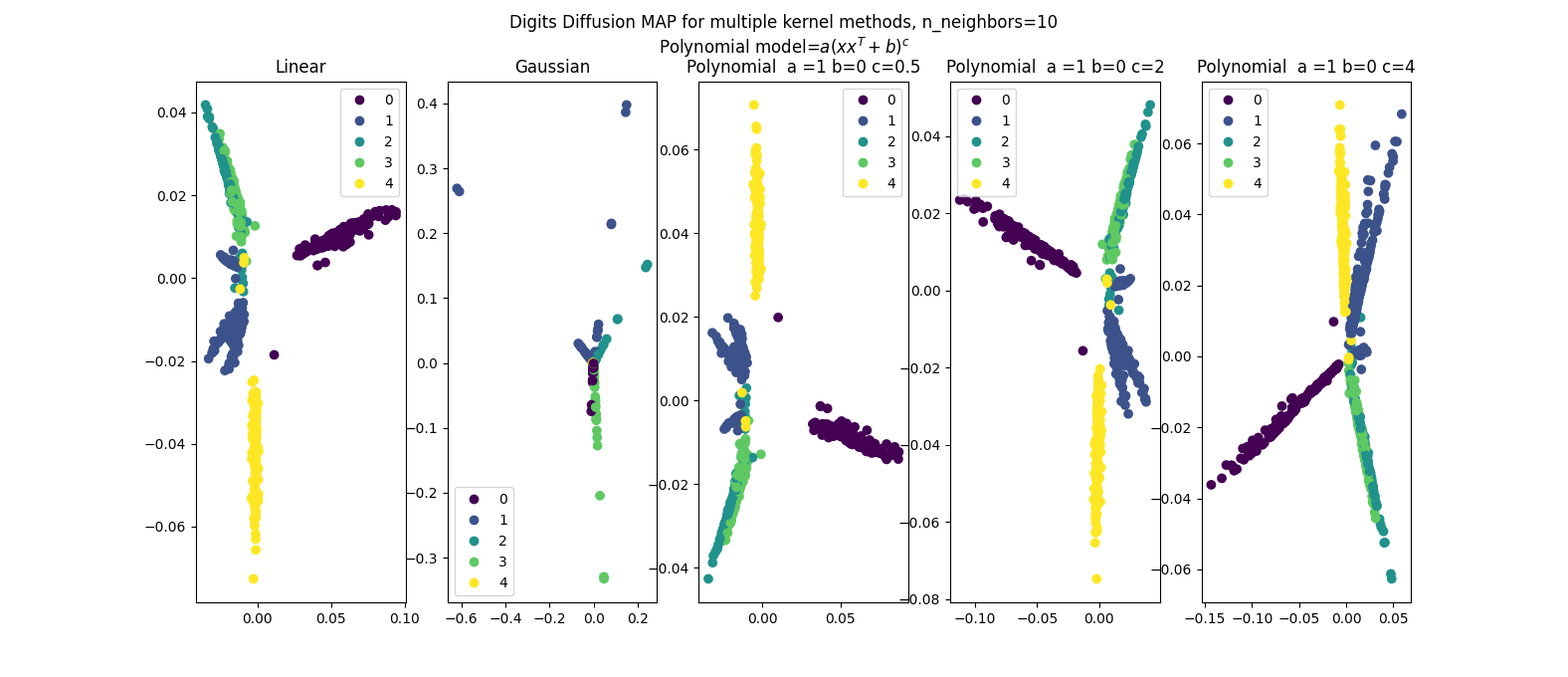
**Let us apply Locally\_Linear\_Embedding:**  
Let us test the method for different values of n\_neighbors:

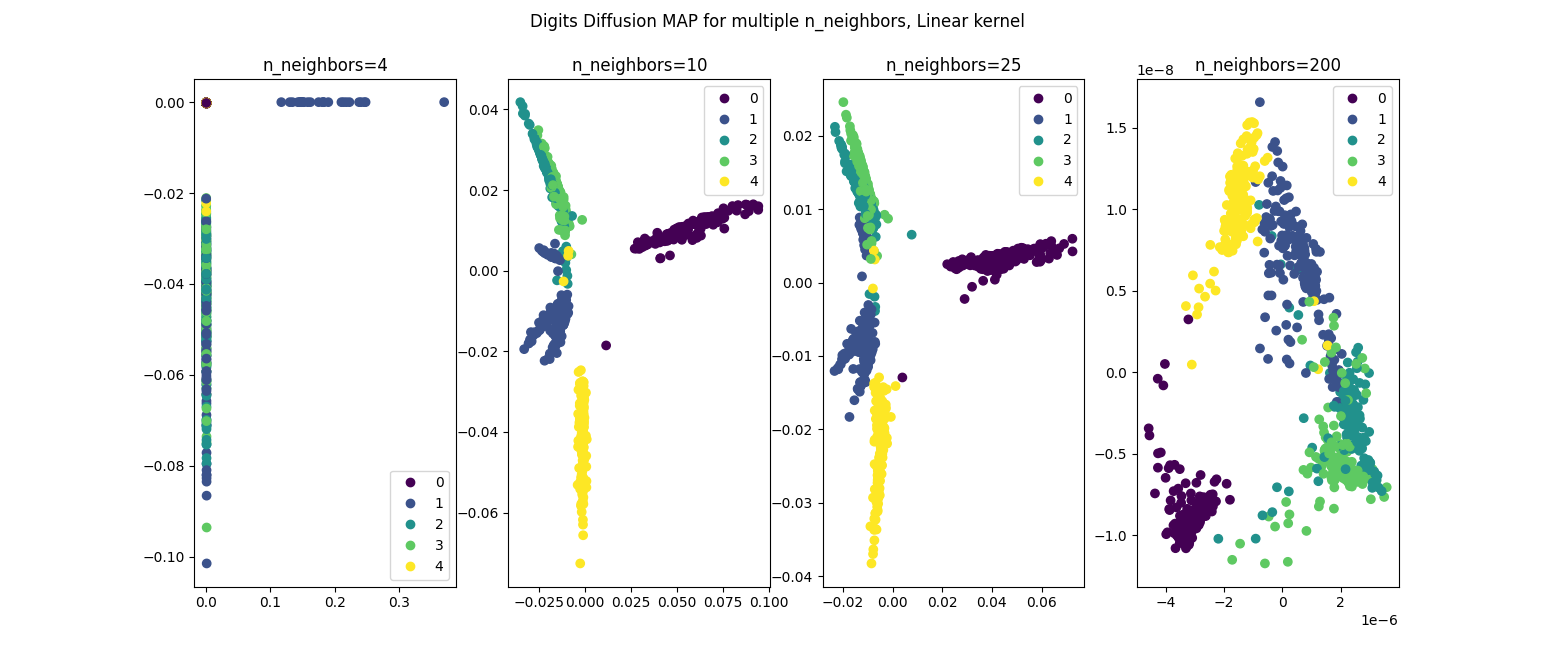
We can notice that the method performs poorly for a low number of neighbors, and it seems like the most accurate representation is achieved the higher the number of neighbors is (at least for this specific dataset).

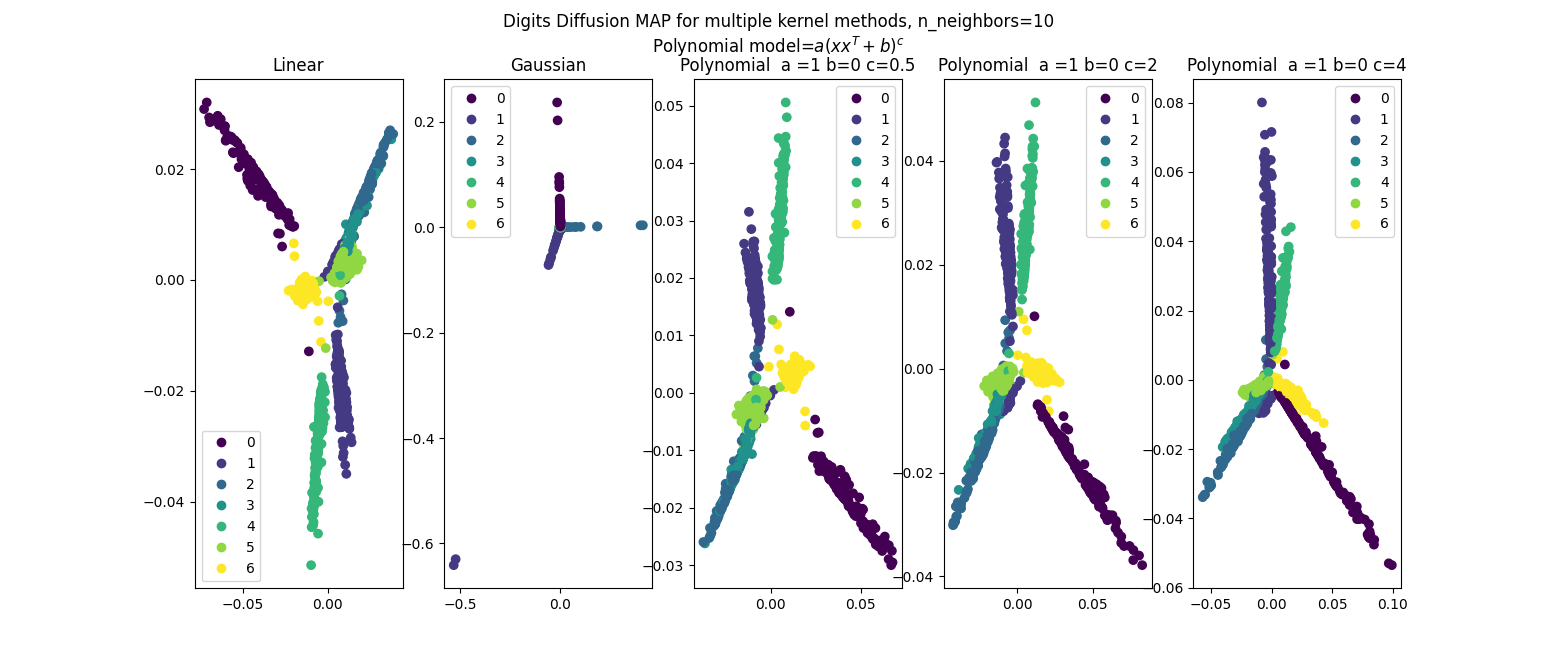
1. **Digits:**

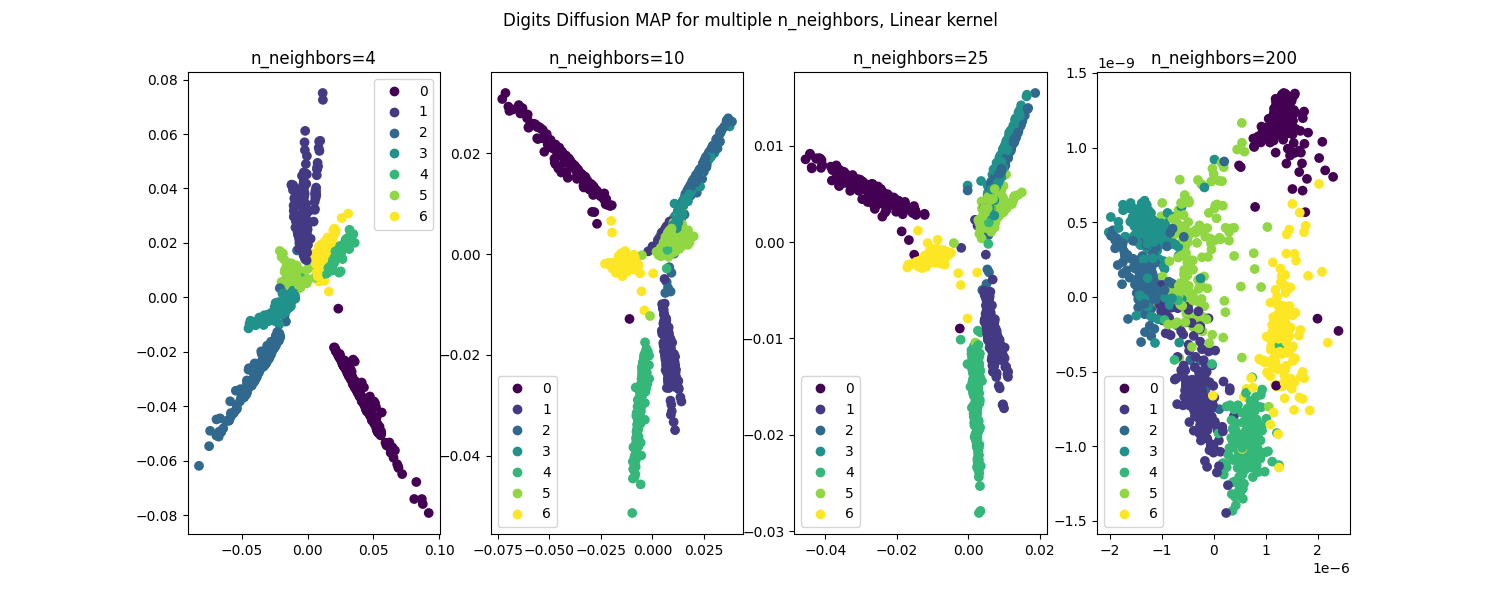
 **Let us apply Diffusion\_Maps:**  
  
  
**For n\_class = 3:**  
Firstly, we test the different kernels with n\_neighbors = 10:

The best results were achieved for Linear and polynomial with (a,b,c) = (1,0,0.5) ( ), since these were the least prone to crowding.  
We decided to stick to the former variation for the following test.  
  
Next, we can test the method for different values of n\_neighbors:  
  
We can see that the results with lower number of neighbors are easier to classify in this case.   
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
 **For n\_class = 5:**Firstly, we test the different kernels with n\_neighbors = 10:

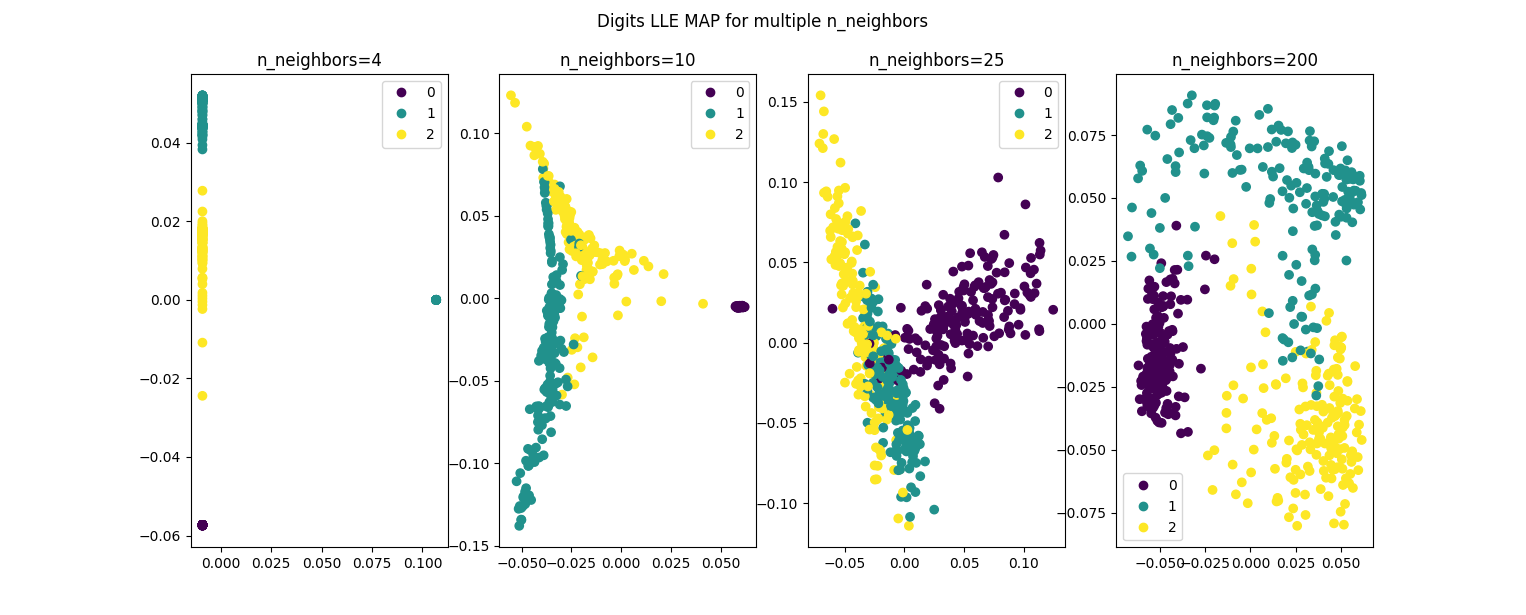
  
We can see that all the kernels tested face a problem classifying the results, especially classes 2 and 3.   
  
Next, we will test the method with Linear kernel for different values of n\_neighbors:

  
  
We can see that the result with 4 neighbors is much clustered and cannot be classified.  
We can see that as the number of neighbors increases, classes 2 and 3 are more separable, and a good value would be between 25 (classes harder to separate) and 100 (2D data too spread out).  
  
**For n\_class = 7:**  
Firstly, we test the different kernels with n\_neighbors = 10:

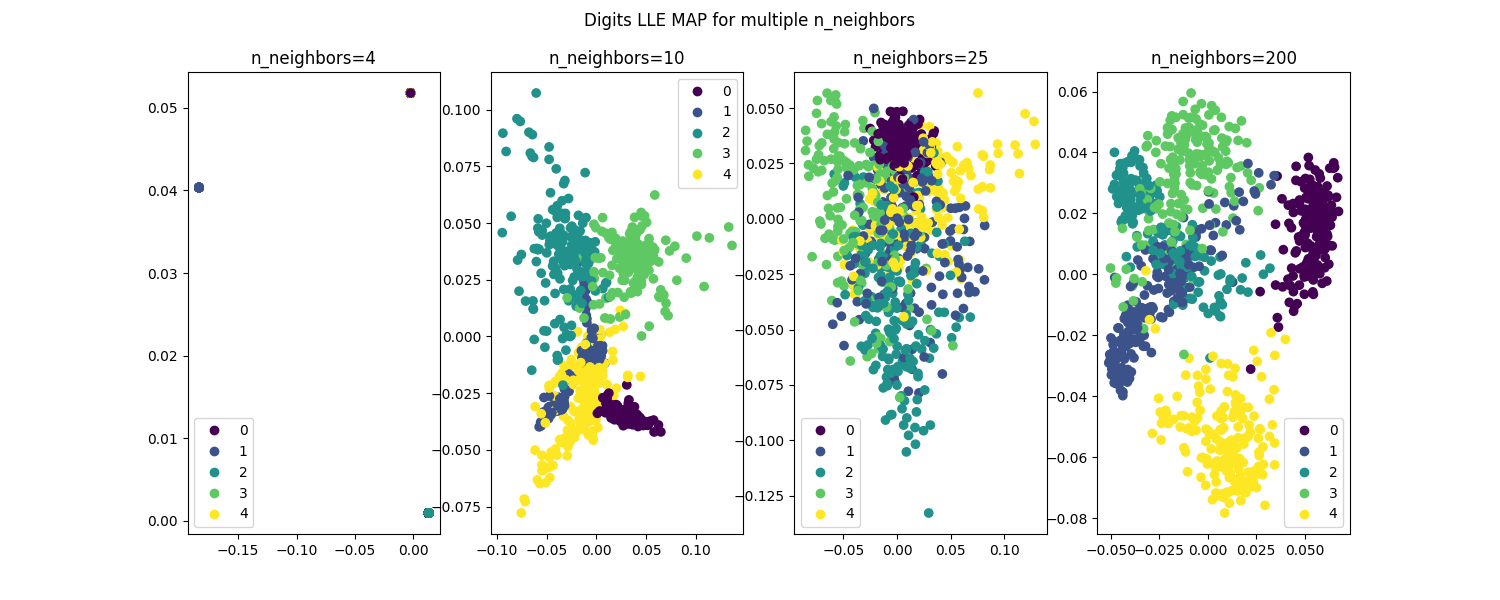
  
  
We can see that like for n\_class = 5 all the kernels tested face a problem separating classes 2 and 3. However, while the results fall victim to the crowding problem, they are distinguishable, especially for the Linear kernel, and the polynomials with c=0.5,2.  
We will stick to the Linear kernel for the following test.  
  
Next, we can test the method for different values of n\_neighbors:

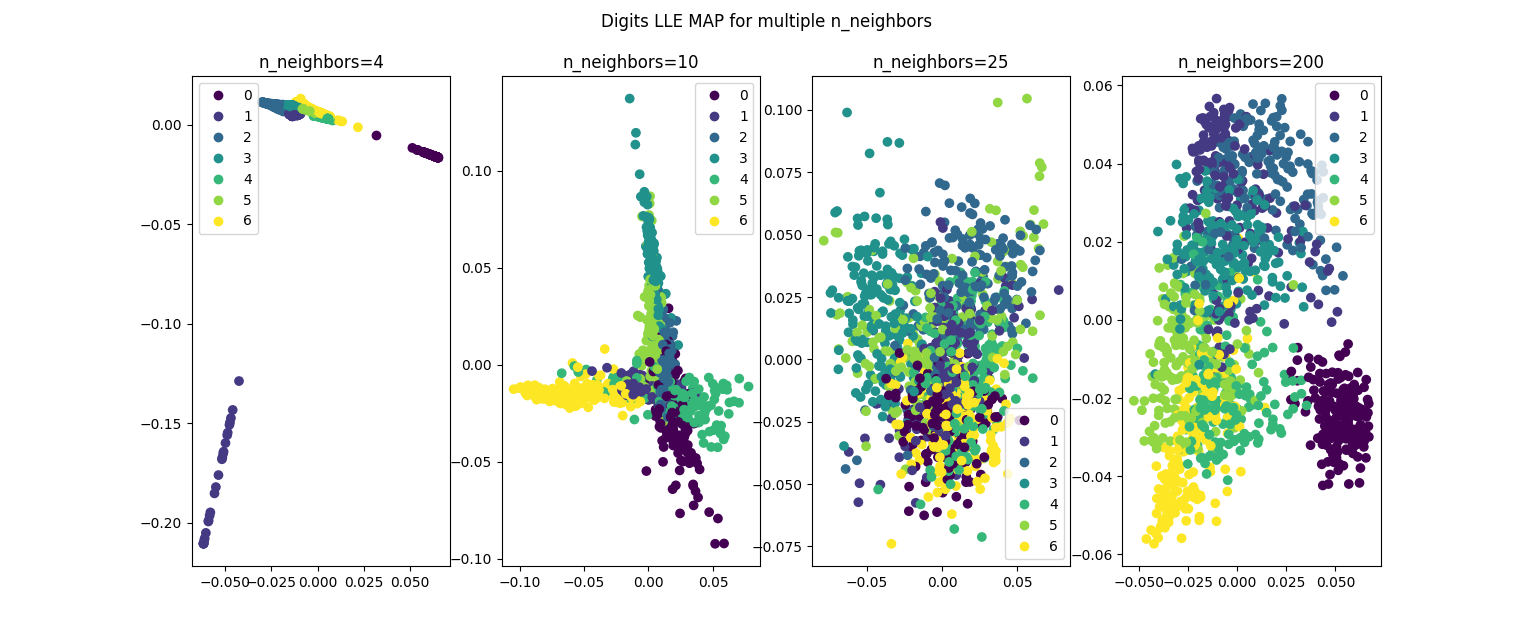
  
  
  
And the inference is similar to the case with n\_class = 5.

**Let us apply Locally\_Linear\_Embedding:  
  
For n\_class = 3, different values of n\_neighbors:**



**For n\_class = 5, different values of n\_neighbors:**

 **For n\_class = 7, different values of n\_neighbors:**



We can conclude that the LLE method is not a great fit for classifying n\_class = 5, 7, for any number of neighbors tested.   
For n\_class = 3, the results are better, however a better result can be achieved for this dataset with diffusion maps with an appropriate kernel.