Estimating a Set in R^2 with Finite Number of Points using Minimum Spanning Tree and Multi Layer Perceptron

A Dissertation submitted in partial fulfillment of the requirement for the M.Tech (Computer Science) degree of the Indian Statistical Institute.

By
J VENKATESH BABU

Under the guidance of
Dr. C A MURTHY
Machine Intelligence Unit.
Indian Statistical Institute,
203, B.T.Road,
Calcutta 700 035.
India.

ACKNOLEDGEMENT

I express my gratitude to Dr. C A MURTHY for his guidance and constant encouragement all through the year this dissertation was in progress. He took time out of his busy schedule, to help me with the preliminaries concerning the topic of this dissertation, and also for many interesting discussions. Many of the ideas in this dissertation owe their origin to him. I also thank Dr. M K Kundu for permitting me to take printouts of the images in his unit. I thank Dr. Uma Shankar who helped me while taking the printouts of the images.

Indian Statistical Institute
203,B.T.Road,
Calcutta - 700 035, INDIA.

Certificate of Approval

This is to certify that the thesis titled **Estimating a Set in R^2 with Finite Number of Points Using Minimum Spanning Tree and Multi Layer Perceptron** submitted by **J Venkatesh Babu**, towards partial fulfillment of the requirements for the degree of M.Tech. in Computer Science at the Indian Statistical Institute, Calcutta, embodies the work done under my supervision.

Machine Intelligence Unit
Indian Statistical Institute,
Calcutta - 700 035.

Contents

Chapter 1

Introduction

1.1 Problem Specification

The problem discussed here is "Given a collection of representative points from a set of points, how to get back the original set".

Once the set is computed, some salient features of the class can then be extracted which may be useful in making decisions about a course of action (for eg. , in identification , classification etc.) to be taken later. This will also reduce storage requirement of the set.

It may be noted that in most of the real life pattern recognition problems, the complete description of a set is not known. Instead a few sampled points are usually available. Hence determining the set and its shape from sampled points is an important problem in pattern recognition.

1.2 Work Already Done

The first method by which people tried to solve this problem was constructing convex hulls. The efficient construction of convex hulls for finite sets of points in the plane is one of the most exhaustively examined problems in computational geometry. Part of the motivation is theoretical in nature. It seems to stem from the fact that the convex hull problem, like sorting, is easy to formulate and visualize. Furthermore, the convex hull problem, again like sorting, plays an important role as a component of a large number of more complex problems. Nevertheless, much of the work is motivated by the direct applications in some more practical branches of computer science.

Akl and Toussiant[2], for instance, discuss the relevance of the convex hull problem to pattern recognition. By identifying and ordering the extreme points of a point set, the convex hull serves to characterize, atleast in a rough way, the "shape" of such a set.

Jarvis[3] presents several algorithms based on so called nearest- neighbour logic that compute what he calls the "shape" of a finite set of points. The "shape", in Jarvis terminology, is a notion made concrete by the algorithms that he proposes for its construction. Besides this lack of any analytic definition, the inefficiency of Jarvis' terminology to construct the "shape" is striking drawback. Fairfeild[4] introduced a notion of the shape of a finite point set based on the closest point voronoi diagram of the set. He informally links his notion of shape with human perception but presents no concrete properties of his shapes, in particular, algorithmic results.

The disadvantage of the above methods is that they do not work for non-convex sets. Grenander[11] tried to give a method which finds the pattern class for non-convex sets. His method was , given a set of points , for each point include all the points within a circle of radius $<\epsilon$ for suitable ϵ. But the drawback with his method is that he did not give any method by which to calculate the value of ϵ.

The method suggested by Edelsbrunner et al[5] is the following. Remove from the whole space open discs of radius(r) > 0, r is to be choosen suitably, such that they do not have any intersection with the set given. But there is no automatic way of choosing r.

The method suggested by Murthy[1] is an extension of the method suggested by Grenander. He gave the method to calculate the value of ϵ in the Grenander's method. He also gave another method by which to estimate the set given a set of points. Both these methods are based on Minimum Spanning Tree and these methods will be explained chapter II.

The method suggested by Mandal et al [10] uses fuzzy logic to extend the boundary to some extent to handle the possible uncovered portions by the sample points. They argue that the boundary need not be limited by the sample points because the resulting boundary leaves certain regions not confined in the sample points, although it should be. Their method extends the boundary such that

(i) As the number of sample points increases the extended portion decreases.

(ii) The extended portions have less probability to be in the pattern class than the portions explicitly highlighted by the sample points.

1.3 Organisation

The work done in this thesis can be divided into two parts.

(i) First part deals with checking whether the method suggested by Murthy[1] works for different data sets. Different sets of data are taken and for each data the number of points is varied and the results studied.

(ii) The second part deals with giving a Multi Layer Perceptron based implementation to the above method.

Chapter II deals with the first part mentioned above. First section gives a general introduction to Minimal Spanning tree. In the second section the main algorithm used is stated. Next section gives the implementation details. Last section gives the tests and results.

Chapter III deals with item(ii) mentioned above. First section gives an introduction to Multilayer perceptron and the second section gives the back propagation algorithm on which the multilayer perceptron works. The third section gives the approach taken to solve the problem. The fourth section explains the solution we obtained for solving the problem. Fifth section gives the tests and results.

Chapter IV gives the conclusions and discussions.

Chapter 2

Graph Theoretic Approach

2.1 Introduction to Minimal Spanning Tree (MST)

We begin by reviewing some terms of graph theory. A graph consists of a set of nodes and a set of node pairs called edges. We say that an edge links the two nodes defining it and it is incident on both of them. The degree of a node is number of edges incident on it. A path between two prescribed nodes is an alternating sequence of nodes and edges with the prescribed nodes as first and last elements, all other nodes distinct, and each edge linking the two nodes adjacent to it in the sequence. A connected graph has a path between any two distinct nodes. A cycle is a path beginning and ending with the same node. A tree is a connected graph with no cycles. A subgraph of a given graph is a graph, with all of its nodes and edges from the given graph. A spanning subgraph of a given graph is a subgraph with node set identical to the node set of the given graph. A spanning tree of a graph is a spanning graph that is a tree. Note that there is a unique path between every two nodes in a tree and thus a spanning tree of a connected graph provides a path between every two nodes of the graph.

An edge weighted graph is a graph with a real number assigned to each edge. A minimal spanning tree (MST) of an edge weighted graph is a spanning tree for which the sum of edge weights is minimum. Note that a graph (collection of nodes and edges) is assumed to be given for finding MST. But in this thesis, the graph is complete graph and MST will be constructed when a set of points is given with the edge weight being the euclidean distance between the corresponding points.

2.2 Main Algorithm for Set Estimation

The algorithm being used for the set estimation is the one given by Murthy[1] which is given below.

1. Get the input set of points. Call this set S.
2. Find the distances between each pair of points.
3. Use the above distances as the edge cost to find Minimum Spanning Tree. Call this set G.
4. Let l_n be the total cost of the MST. Calculate $h_n = \sqrt{\frac{l_n}{n}}$
5. Let

$$A_{1n} = \{X : \|X - X_i\| \leq h_n\}$$
$$A_{2n} = \{X : d(X, G) \leq h_n\}$$
$$where\ d(X,G) = min_{y \in G}\ \ d(x,y)$$
$$A_{1n} \rightarrow A\ \ and$$
$$A_{2n} \rightarrow A\ \ and$$
$$as\ \ n \rightarrow \infty$$

2.3 Implementation

2.3.1 Implementation of Kruskal's Algorithm

The problem of finding minimum spanning tree can be stated as follows: Given a simple undirected graph $G = (V, E)$ and a cost function $c : E \rightarrow R$, the problem is to find a tree $T = (V, E\prime)$, where $E\prime \subseteq E$ with minimum cost. This needs clarification as to what we understand by the cost of a tree. The cost of a tree T , denoted by

$c(T)$, is $\sum_{e \in E'} c(e)$, the sum of the cost of all the edges. Since the vertex set of T and G coincides, we call T as a spanning tree. Also, the minimum spanning tree need not be unique. The Kruskal's algorithm discussed in this section is based on a "greedy" strategy. Such a strategy is not generally guaranteed to find global optimal solutions to problems. However, for minimum spanning tree problem, it can be proved that Kruskal's algorithm do yield a spanning tree with minimum cost[6].

The algorithm is as follows: We can think of an equivalence relation on the set of vertices such that the equivalence classes represent the growing trees. Also, we associate a set of edges with each of the equivalence classes such that the equivalence class together with the associated set of edges form a tree on the vertices of the class. Initially each vertex is the sole element of its equivalence class, and the associated set of edges is empty. When the algorithm terminates only one equivalence class remains and the associated set of edges is a minimum spanning tree. At each step of the algorithm, we select an edge with an end point in two different equivalence classes and merge them. Edges, both end points of which lie in the same equivalence class are permanently excluded from consideration for selection , since their selection would lead to a cycle. In order to minimize the increase in the value of total cost, the greedy approach dictates that the allowable edge of least cost to be chosen next.

Algorithm

1. Initialise the equivalence classes to a single vertex and the associated edge set to empty.
2. Pick up an minimum cost edge between two equivalence class, say the edge is e. Merge the two equivalence class. The associated edge set is the union of the two edge sets of the equivalence classes participating in the merging together with e.
3. Repeat step 2 until a minimum spanning tree is found.

Actual Implementation

1. Find all the edge weights i.e., the eucliden distance between each individual pair of points.
2. `Tree` $\leftarrow \phi$

```
3. vs $\leftarrow \phi$
4. For all vertex $v \in V$, vs $\leftarrow vs \cup \{\{v\}\}$
5. If $|$ vs $| = 1$ halt.
6. Find the edge $uv$ with minimum edge weight and remove this edge from the
   edge set.
7. Let $u, v$ belong to the sets $w_i, w_j$ respectively, in vs.
8. If $w_i \neq w_j$ then add $(u, v)$ to Tree and replace $w_i, w_j$ by $w_i \cup w_j$ in vs.
9. goto step 5.
```

From the above description of the algorithm, we see that the efficiency of the algorithm depends highly on the way the distances are stored and also on the step 6. The storage structure used for storing the distances is shown below.

$$\begin{pmatrix} d_{12} & d_{13} & d_{14} & \cdots & d_{1n} \\ d_{23} & d_{24} & \cdots & d_{2n} & \\ d_{34} & \cdots & d_{3n} & & \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ d_{n-2n-1} & d_{n-2n} & & & \\ d_{n-1n} & & & & \end{pmatrix}$$

This is not exactly a two-dimensional array. We store in row i the distances of point i with points $i+1$ to n. Here d_{ij} is the euclidean distance between point i and point j.

The step which has to be handled efficiently is step 6 of Kruskal's algorithm. Here we find the edge with min. cost. We have maintained another array of dimension n. The i^{th} element of the array gives the minimum element of i^{th} row of distance matrix. So for finding overall minimum cost edge we just have to find the minimum of this array. After removing this edge we have to update this entry which will take atmost n steps.

2.3.2 Distance Measure

In the step 5 we find the sets A_{1n} and A_{2n}. To find these sets we have used graphics support. To calculate A_{1n}, the procedure we followed is as follows : For each point in the point set we find the square of width $h_n \times h_n$ with this point as the center. Then we blacken this portion on the screen. After this procedure is repeated for all the points in the point set we will get an image which represents the obtained set such that if the pixel value is black then the corresponding point belongs to the set. Then the screen dump is taken and stored in a file.

The procedure to calculate A_{2n} is not so straight forward. We have to include all the points which are at a distance less than h_n from any point on the edges of MST. So what we have to do is, find the rectangle around each edge of the MST leaving h_n on all sides of the edge. This can be done as follows : Rotate the coordinate axis so that the y-axis becomes parellel to the given edge. Then with the transformed coordinates of the edge we can easily find the coordinates of the rectangle enclosing this edge by some simple additions and subtractions of h_n from x and y coordinates of the edge. Then the obtained vertices of the rectangle are transformed back to the original axis by rotating the coordinate axis in reverse direction. Now the portion of this rectangle is blackened on the screen. The procedure is repeated for all the edges in the MST. Then the screen dump is taken and stored in a file.

Now what remains is to check how close these sets are with the original set. For this purpose the distance measure we used is given below. We defined two functions as follows

$$f(x,y) = \begin{cases} 1 & \text{if } (x,y) \in \text{original set} \\ 0 & \text{Otherwise} \end{cases}$$

$$f'(x,y) = \begin{cases} 1 & \text{if } (x,y) \in \text{obtained set} \\ 0 & \text{Otherwise} \end{cases}$$

Distance measure is then defined as :

$$D = \sum_x \sum_y |f(x,y) - f'(x,y)|$$

2.4 Tests and Results

The algorithm is tested for different data sets. But main thrust was given for set **P**. The original set is shown in fig(i). (in the subsequent discussion set is assumed to be displayed as an image). fig(ii) shows the points in the plane when number of points is taken as 100. fig(iii) shows the MST and fig(iv) shows the set A_{1n} for the points shown in fig(ii). fig(v) shows the set A_{1n} for 300 points. fig(vi) shows the set A_{1n} for 500 points. fig(vii) shows the set obtained for 700 points and fig(viii) shows the set obtained for 1000 points. It can be noticed from the figures how as the number of points increases the obtained set tends towards the original set. Also from table I (fig(xvi)) it can be seen that the distance between the original and obtained set is decreasing with increasing number of points. fig(ix) shows A_{2n} set for 1000 points in P. fig(x) shows the original set for **B**. fig(xi) and fig(xii) shows sets A_{1n} and A_{2n} for 1000 points. fig(xiii) shows the original set for **c**. fig(xiv) and fig(xv) shows sets A_{1n} and A_{2n} respectively for 1000 points.

fig(xvi) gives the table for various values of h_n and distances from the original set for different values of number of points for set P. fig(xvii) is table for set B while fig(xviii) is table for set C.

Chapter 3

MLP Based Method

3.1 Introduction to Multi Layer Perceptron (MLP)

Multi-layer perceptrons are feed forward nets with one or more layers of nodes between the input and output nodes. MLPs overcome many of the limitations of single layer perceptrons, but were generally not used in the past because effective training algorithms were not available. This has changed with the development of new training algorithms. Although it cannot be proven that these algorithms converge as with single layer perceptron, they have been shown to be successful for many problems of interest.

The capabilities of multi-layer perceptrons stem from the nonlinearities used within nodes. If nodes were linear elements, then a single layer net with appropriately chosen weights could exactly duplicate those calculations performed by any multi-layer net.

A single layer perceptron forms half plane decision regions. A two layer perceptron can form any, possibly unbounded, convex region in space spanned by the inputs. Such regions include convex polygons, called convex hulls, and the unbounded convex regions. Here the term convex means that any line joining points on the border of a region goes only through points within that region. Convex regions are formed from intersections of the half plane regions formed by each node in the first layer of the multi layer perceptron. Each node in the first layer behaves like a single layer and has a high output only for points on one side of the hyper plane formed by its weights and offset. If weights to an output node from $N1$ first layer nodes are all 1.0 and the threshold in the output node is $N1 - e$ where $0 < e < 1$, then the output node will be high only if outputs of all the first layer nodes are high. This corresponds to performing a logical AND operation in the output node. Intersections of such half plane regions form convex regions as described above. These convex regions have at the most as many sides as there are nodes in the first layer.

This analysis provides some insight into the problem of selecting the number of nodes to use in two layer perceptron. The number of nodes must be large enough to form a decision region that is as complex as is required by a given region. It must not, however, be so large that the many weights required can not be reliably estimated from the available training data.

A three-layer perceptron can form arbitrarily complex decision regions and can separate the meshed classes. This can be proven by construction[7]. The proof depends on partitioning the desired decision region into small hypercubes (squares when there are two inputs). Each hypercube requires 2N nodes in the first layer (four nodes when there are two inputs), one for each side of the hypercube, and one node in the second layer that takes the logical AND of the outputs from the first layer nodes. The outputs of second layer nodes will be "high" only for inputs within each hypercube. Hypercubes are assigned to the proper decision regions by connecting the output of each second-layer nodes only to the output node corresponding to the decision region that node's hypercube is in and performing a logical OR operation in each output node. A logical OR operation will be performed if these connection weights from the second hidden layer to the output layer are one and thresholds in the output nodes are 0.5. This construction procedure can be generalized to use arbitrarily shaped convex regions instead of small hypercubes and is capable of generating the disconnected and non-convex regions.

The above analysis provides some insight into the problem of selecting the number of nodes to use in three-layer perceptrons. The number of nodes in the second layer must be greater than one when decision regions are disconnected or meshed and cannot be formed from one convex area. The number of second layer nodes required in the worst case is equal to the number of disconnected regions in input distributions. The number of nodes in the first layer must typically be sufficient to provide three or more edges for each convex area generated by every second layer node. There should thus typically be more than three times as many nodes in the second as in the first layer.

The above discussion centered primarily on MLPs with one output when hard limiting nonlinearities are used. Similar behavior is exhibited by MLP with multiple output nodes when sigmoidal nonlinearities are used and the decision rule is to select the class corresponding to the output node with the largest output.

3.2 The MLP algorithm

The back-propagation training algorithm[7] is an iterative gradient descent algorithm designed to minimize the mean square error between the actual output of a multilayer feed-forward perceptron and the desired output. It requires continuous differentiable non-linearities. The following algorithm assumes that the transfer function of each node is sigmoid function where the sigmoid function $f(\alpha)$ is

$$f(\alpha) = \frac{1}{(1 + e^{-(\alpha-\beta)})}$$

The following notation is used in the following algorithm.

L-level MLP	:	MLP with L-1 hidden layers.
w_{ij}^l	:	weight of the link joining node i at level l to node j at level l-1.
y_j	:	output of node j in the output layer.
x_i^l	:	output of node i at level l.
α	:	momentum factor.
η	:	gain factor.
N_l	:	Number of nodes at level l.

Step 1. `Initialize Weights`
set all weights to small random values.

Step 2. `Present Input and Desired Outputs`
Present a continuous valued input vector $x_0, x_1, \cdots, x_{N_0-1}$ and specify the desired outputs $d_0, d_1, \cdots, d_{N_L-1}$. If the net is used as a classifier then all desired outputs are set to zero except for that corresponding to the class the input is from. That desired output is 1. The input could be new on each trial or samples from a training set could be presented cyclically until weights stabilize.

Step 3. `Calculate Actual Outputs`

Use the sigmoid non-linearity from above and formulas given below to calculate the outputs

$$y_k = f(\textstyle\sum_{j=0}^{N_{L-1}} w_{kj}^L \, x_j^{L-1} - \theta) \quad 0 < k < N_L$$

$$x_k^l = f(\textstyle\sum_{j=0}^{N_{l-1}} w_{kj}^l \, x_j^{l-1} - \theta) \quad 0 < k < N_l$$

Here θ is threshold for the node.

Step 4. `Adapt Weights`

Use a recursive algorithm starting at the output nodes and working back to the first hidden layer. Adjust weights by

$$w_{ij}^l(t+1) = w_{ij}^l(t) + \eta\delta_j x_j^{l-1}$$

In this equation $w_{ij}^l(t)$ is weight at time t. If $l = L$ i.e., node j is an output node then,

$$\delta_j = y_j(1 - y_j)(d_j - y_j)$$

where d_j is desired output of node node j and y_j is the actual output. If $l < L$ i.e., node j is an internal node, then

$$\delta_j = x_j^{l-1}(1 - x_j^{l-1}) \sum_k \delta_k w_{jk}^{l+1}$$

where k is over all nodes in the layers above node j. Internal node thresholds are adapted in a similar manner by assuming they are connection weights on links from auxiliary constant-valued(-1) inputs. Convergence is faster if a momentum factor is added and weight changes are smoothed by

$$w_{ij}^l(t+1) = w_{ij}^l(t) + \eta\delta_j x_j^{l-1} + \alpha(w_{ij}^l(t) - w_{ij}^l(t-1))$$

where $0 < \alpha < 1$.

Step 5. `Repeat by going to step 2.`

For the implementation of the above algorithm the weights are stored in a three dimensional vector. That is w_{ij}^l can be represented as $wt[l][i][j]$. The first dimension gives the level number. The second and third dimension gives the node numbers at level l and $l-1$ between which the link with this weight is present. The outputs of the nodes are stored in a two dimensional array where the first dimension gives the layer number and the second dimension gives the node number. Similarly the errors are stored in a two dimensional array. Since the number of layers and number of nodes etc. are not known initially all the memory allocation is done dynamically at run time to save space. The gain factor was choosen as 0.33 and the momentum factor is choosen as 0.17.

3.3 Approach to Solve the Problem

We decided to use MLP because as explained above it can form complex decision regions. The first method tried was : From the point set take each point. Let the point be (x, y). Then form 20 four tuples (x, y, x_1, y_1) such that for ten tuples the distance between (x, y) and (x_1, y_1) is less than h_n and for ten tuples the distance is more than h_n. Then we tried to teach a MLP to learn this patterns. The idea was to teach the MLP such that if the point represented by third and fourth members of the tuple is at a distance less than h_n from the point represented by the first and second members of the tuple then the MLP should give output 1 otherwise 0.

This idea didn't work. The dicision region obtained was not what we wanted. But from this we got the idea that we can just teach an MLP to recognise points from a circle of radius h_n and use it to form A_{1n} set. This idea is explained in next section.

The main difficulty in implememtation is how to choose the MLP architecture. Also how to set the learning and gain factors. we obtained the final MLP by following the heuristics given above to choose the number of nodes and trying various combinations with different sets of initial weights.

3.4 MLP for set estimation

In this section we will try to explain the idea given at the end of the last section. Strictly speaking we will give a neural network to estimate the set A_{1n} given in chapter two which is given below for convenience

$$A_{1n} = \{X : \|X - X_i\| \leq h_n\}$$

The property of MLP to act as pattern classifier is used here to estimate A_{1n}.

We first tried to solve the following problem using which we can estimate the set. Given a set of points $P = p_1, p_2, \cdots, p_n$ where $p_i = (x_i, y_i)$ and a test point $p = (x, y)$ we have to find whether p belongs to the set obtained by P or not.

The basic approach is discussed first before giving the actual algorithm. The set A_{1n} is nothing but union of all the points included within a radius h_n around each point for all the points in the set P. The method we adopted initially is as follows : calculate h_n for the set of points and then teach an MLP to classify points within a circle of radius h_n and points which lie outside this circle. Then we can use this MLP to test whether the test point lies within the set A_{1n} or not just by checking whether this point falls within a circle of radius h_n from any point in the point set. This can be done simply by changing the origin to each point in P and giving the translated coordinates of the test point as input to the MLP. If the output of MLP is 1 for any point then the point lies within the set A_{1n}.

But the above procedure is cumbersome. For each set of points an MLP with suitable number of nodes has to be choosen and then trained. Instead of this we can choose a suitable r and an MLP which can separate points within radius r from origin as belonging to one class, and points outside this radius r as belonging to different class. Then given a test point and set of points we just have to scale the coordinates by the factor r/h_n. This is necessary because the MLP just tries to see whether the point lies within a circle of radius r. But if h_n is different from r then that is not what we want. So we scale the coordinates so that the point can be checked w.r.to circle of radius r. Now we check whether this point falls within distance r from any point in the set or not. If there exists a point in the point set from which the distance of the test point multiplied by r/h_n is less than r then we say that the test point lies within the set determined by the set of points. The beauty of this method lies in the fact that the same MLP will work for any set of points with any value of h_n. Formally the algo. is given below

1. Find $h_n = \sqrt{\frac{l_n}{n}}$ as in the algo. 2.2.
2. Choose a suitable r and an MLP and teach the MLP to separate points within radius r from origin from points outside radius r from origin.
3. Take as input the point to be tested p and the set of points P.
4. Scale the coordinates by the factor r/h_n. For each point in P translate the

origin to the point and give the translated coordinates of the test point as input to the MLP. If the output of MLP is high then the point belongs to the A_{1n} set of P.

Now what remains is to get the set. For this we varied the x and y coordinates from some minimum value to a maximum value and check whether each point (x, y) belongs to the set. If the point (x, y) belongs to the set then we store the value 1. Otherwise 0. Now this byte file can be displayed as binary image.

3.5 Tests and Results

First step is to choose the value of r. It has been choosen as 0.47 arbitrarily. A four layer MLP is choosen with 2 nodes in the input layer, 10 nodes in the first hidden layer , 3 nodes in the second hidden layer , 3 nodes in the the third hidden layer and two output nodes. The MLP was taught with 480 input points with 240 points with distance less than 0.47 and 240 points with distance greater than .47 and less than 0.94 and the back-propagation learning algorithm was applied for 200 iterations. The final error was 0.00016. Using this MLP and a set of 100 points taken from figure(i) randomly the figure(xviii) is generated by giving as inputs the points varying from (-0.5,-0.5) to (3.5,6.5) in steps of 0.1 in each direction. fig(xix) and fig(xx) shows the sets obtained for sets 8 and C.

Chapter 4

Conclusions and Discussions

4.1 Results

The method given in [1] is tested for different data sets and is found to be satisfactorily working. The distance of the obtained set from the original set decreased as the number of points increased as it should. The MLP based method also worked well. The set obtained was very close to the sets we obtained using the above method.

4.2 Further Scope

The method given by [1] was tested only for points in R^2. But this method works for points in any dimension. This can be tested in three or four dimension plane. Also the MLP based solution tries to get the set A_{1n}. A method can be given which will calculate the set A_{2n}.

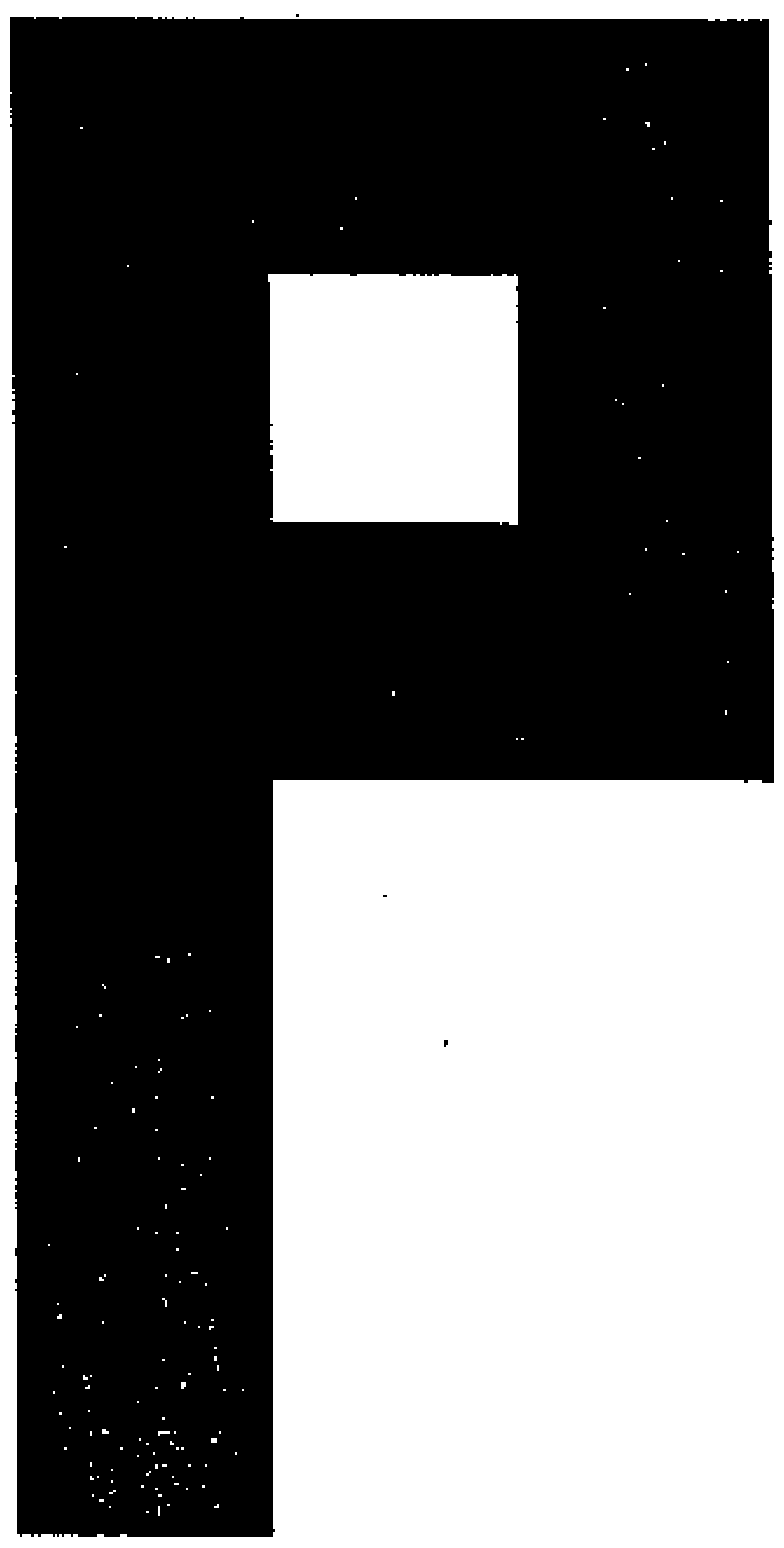
Figure 1: Original set for P

Figure 2: 100 points randomly taken from figure 1.

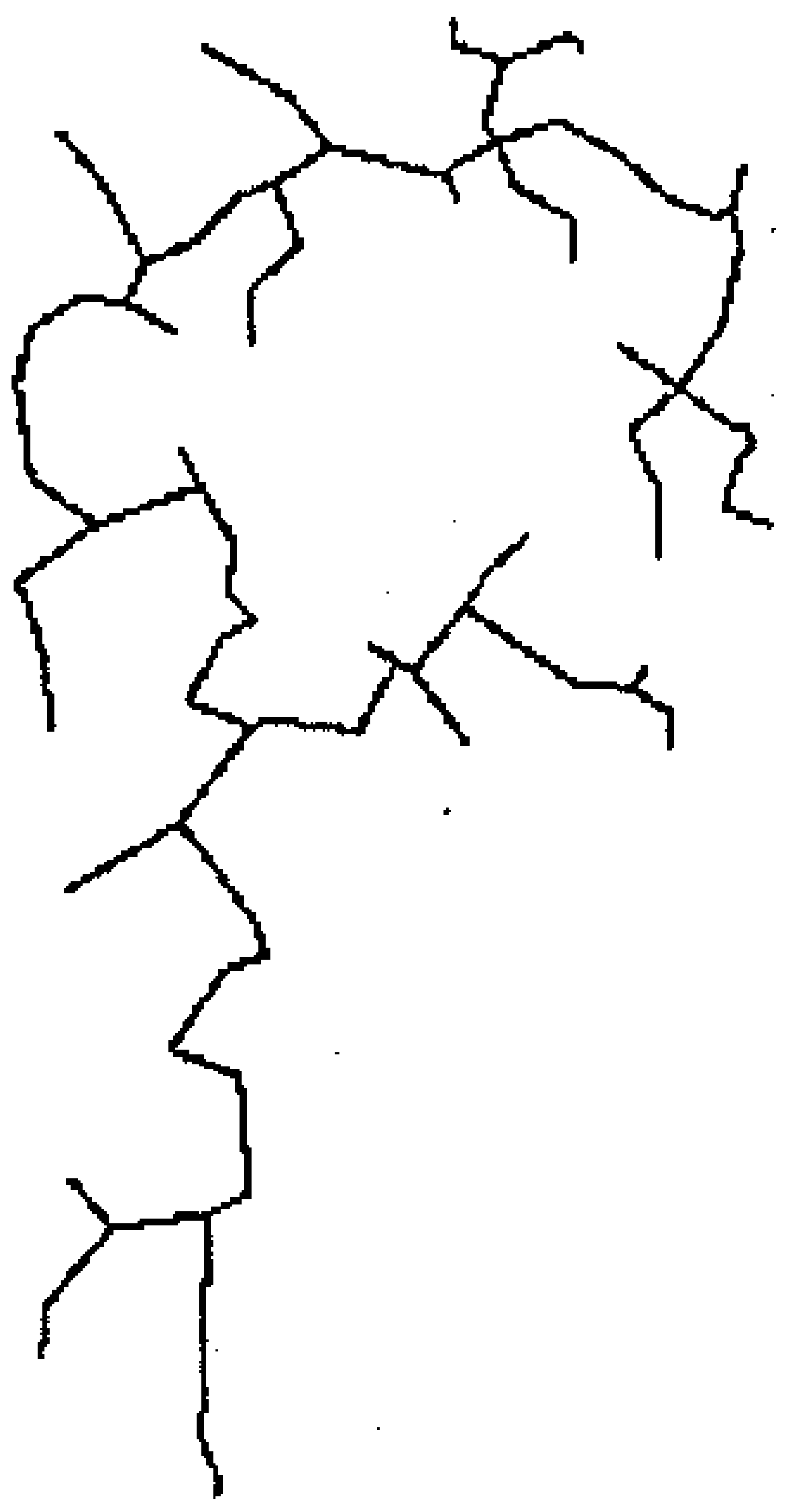

Figure 3: MST for points shown in Figure 2.

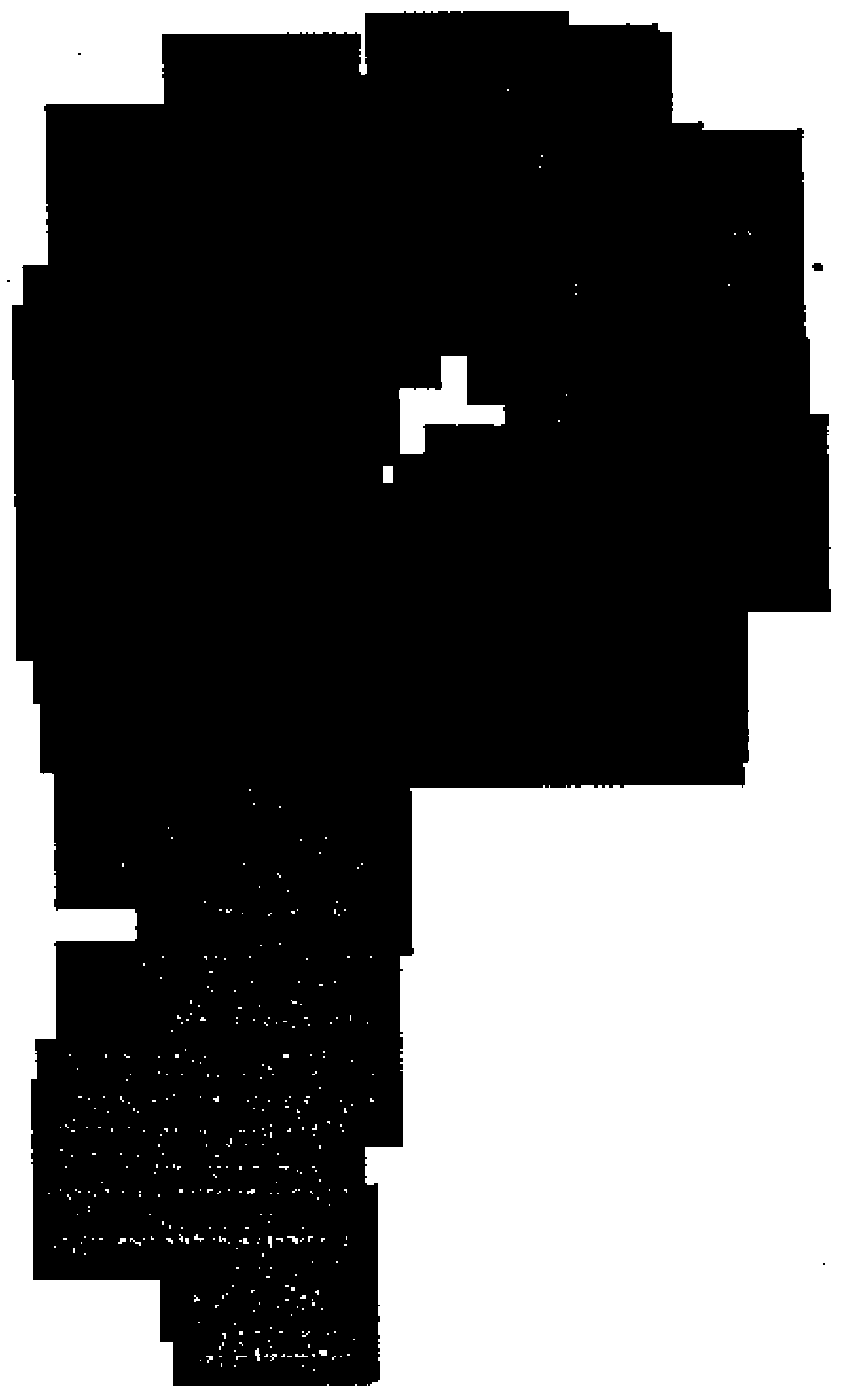

Figure 4: Set **A**(ln) with 100 points.

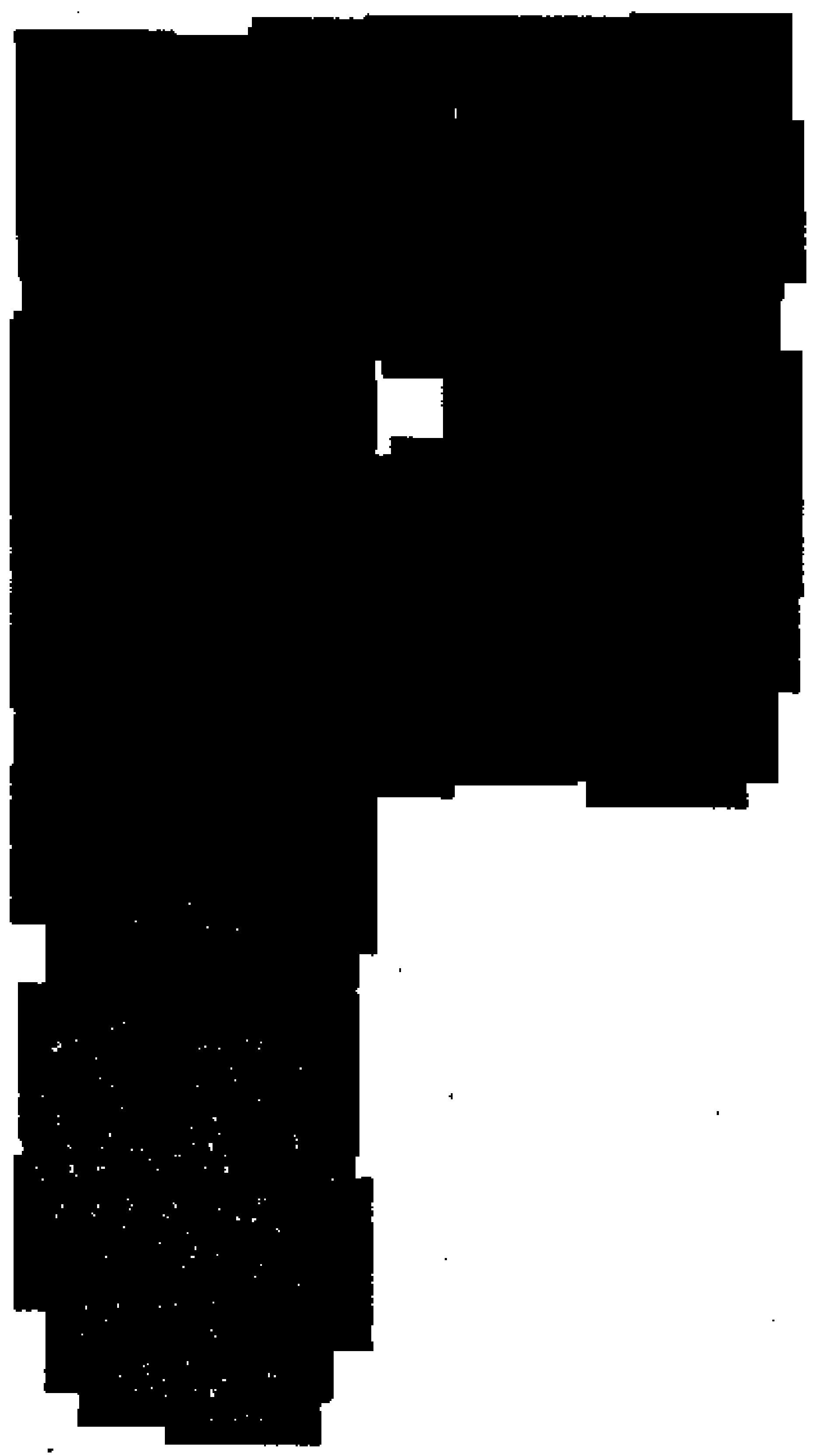
Figure 5: Set **A**(ln) with 100 points.

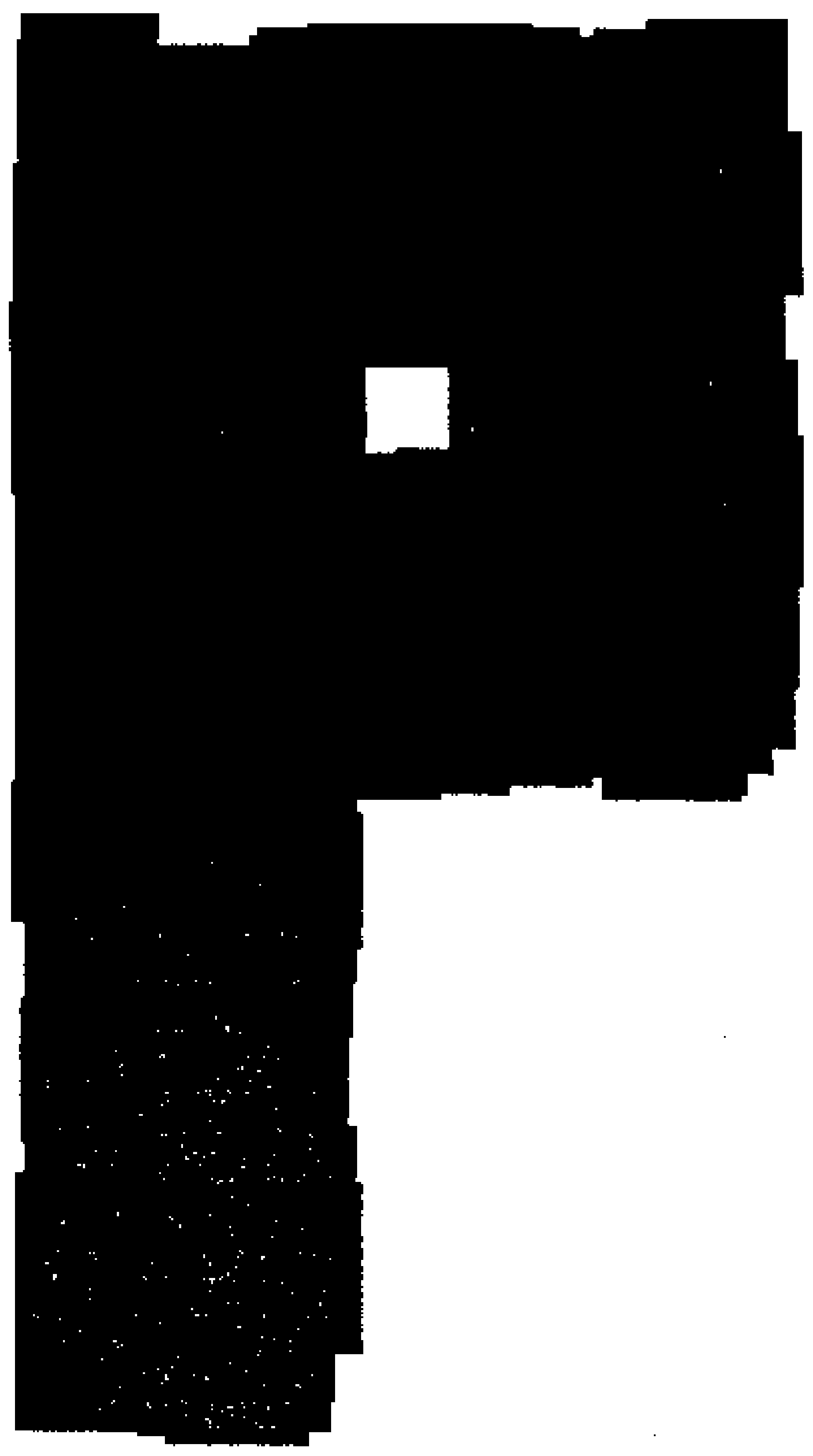
Figure 6: Set **A**(ln) with 500 points.

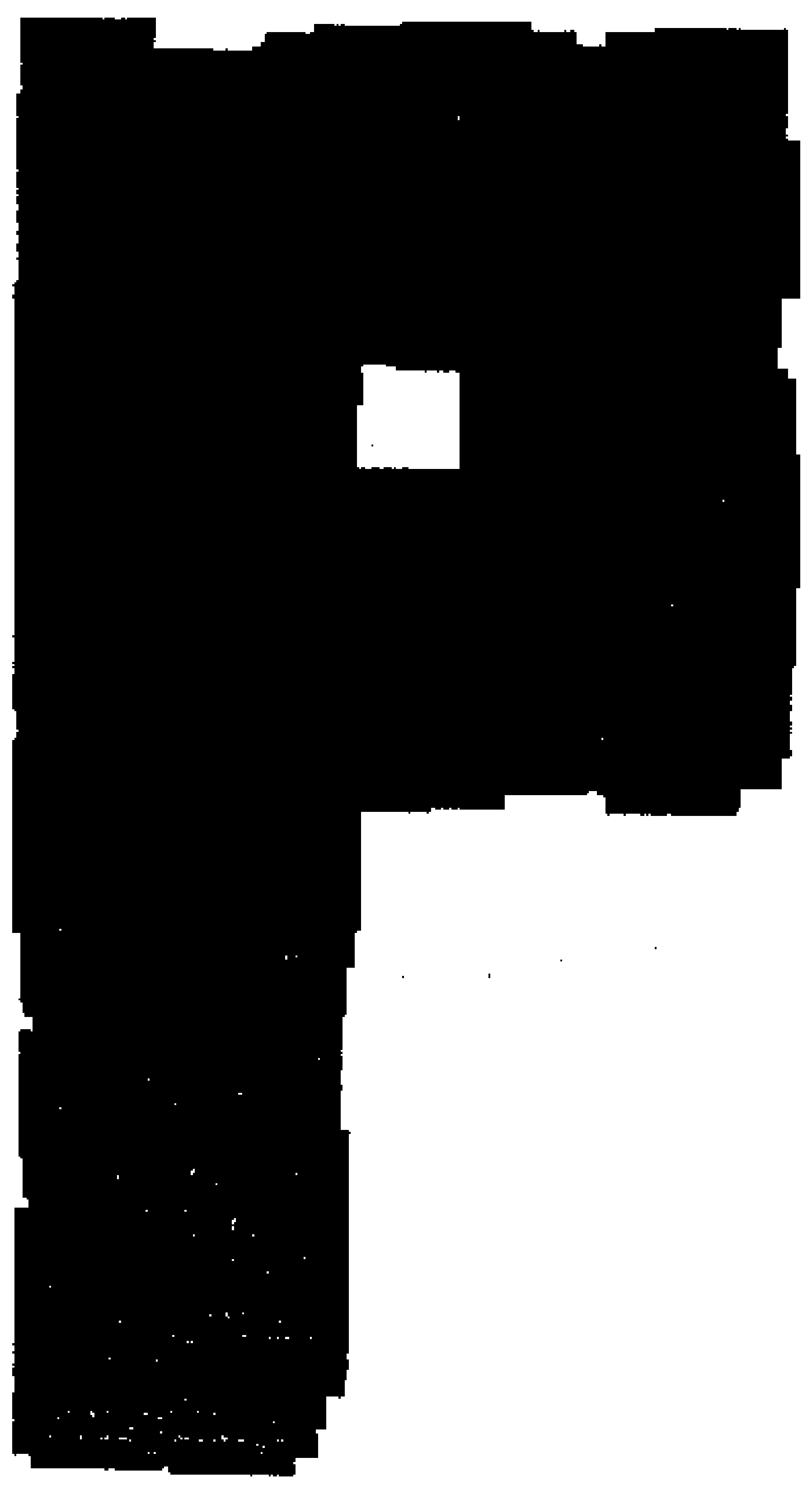

Figure 7: Set **A**(ln) with 700 points.

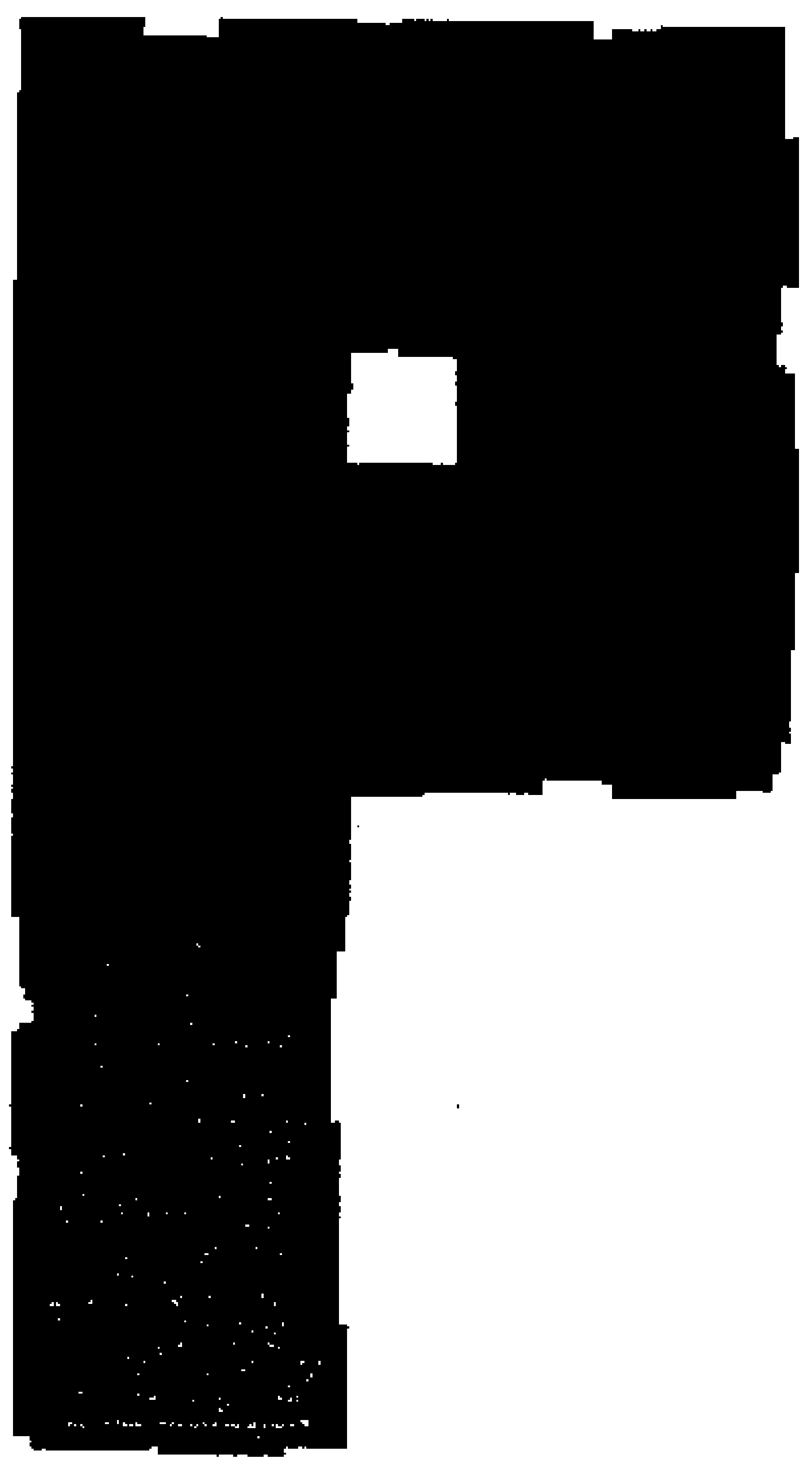

Figure 8: Set **A**(ln) with 1000 points.

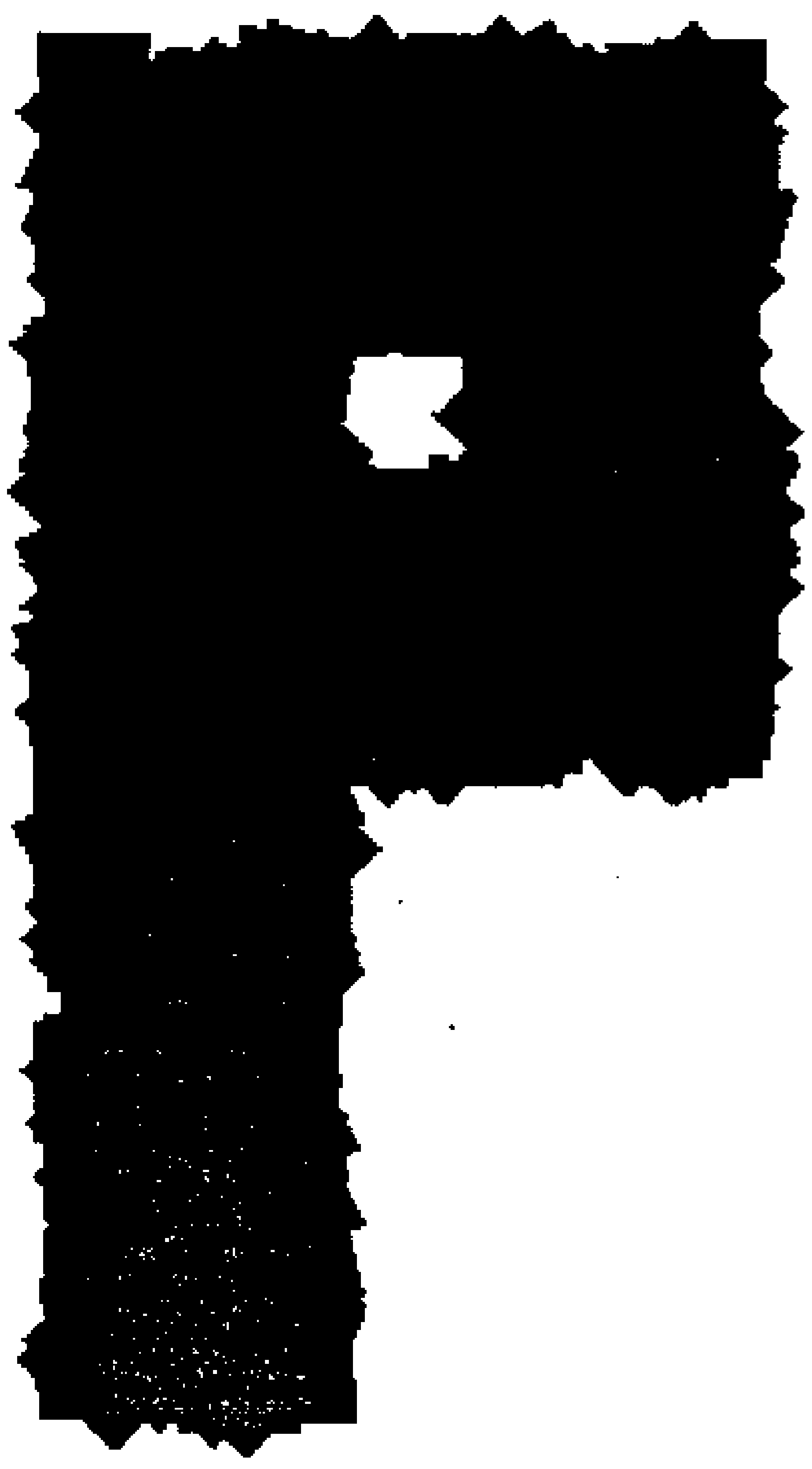

Figure 9: Set **A**(2n) with 1000 points.

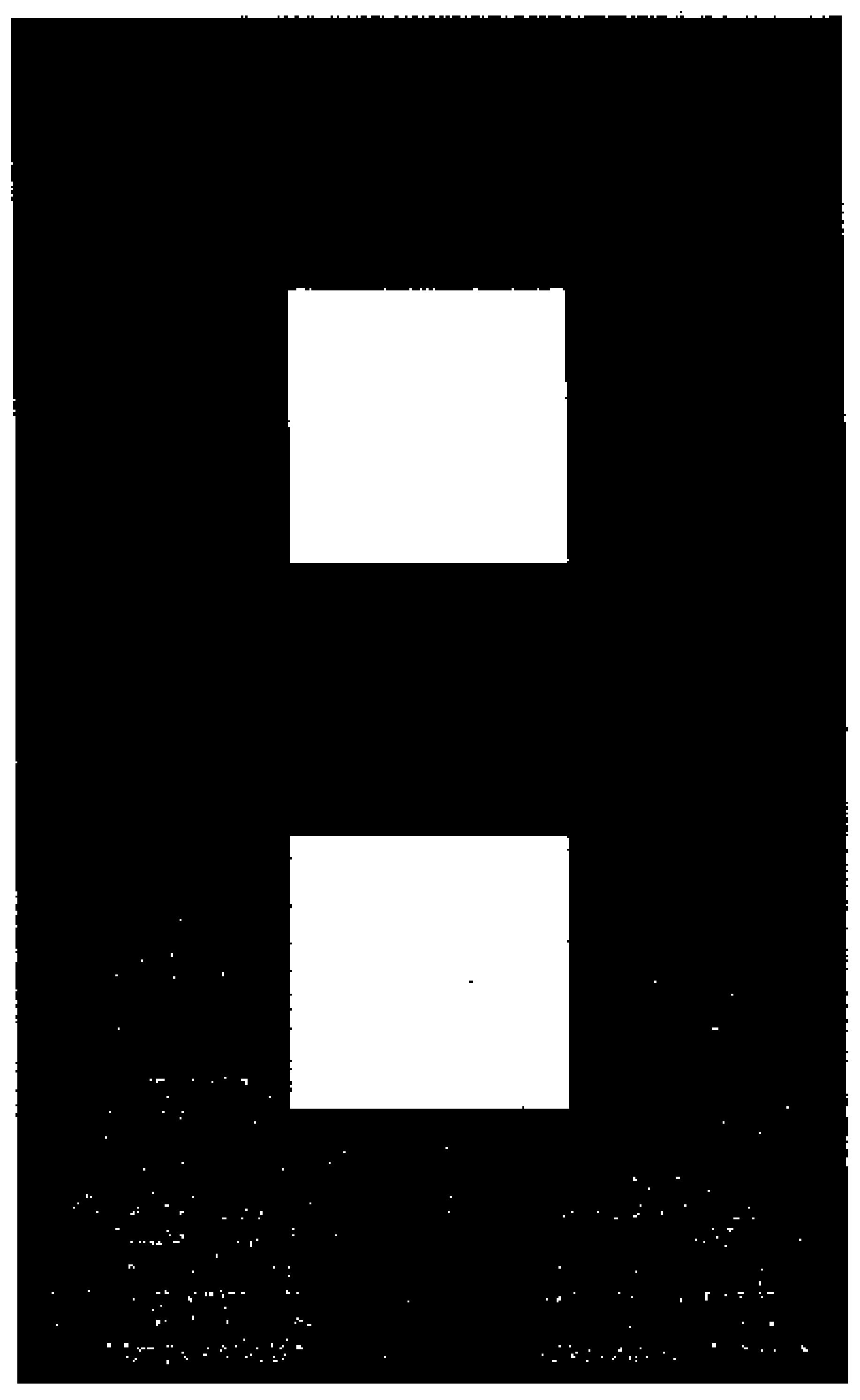

Figure 10: Original set for **8**.

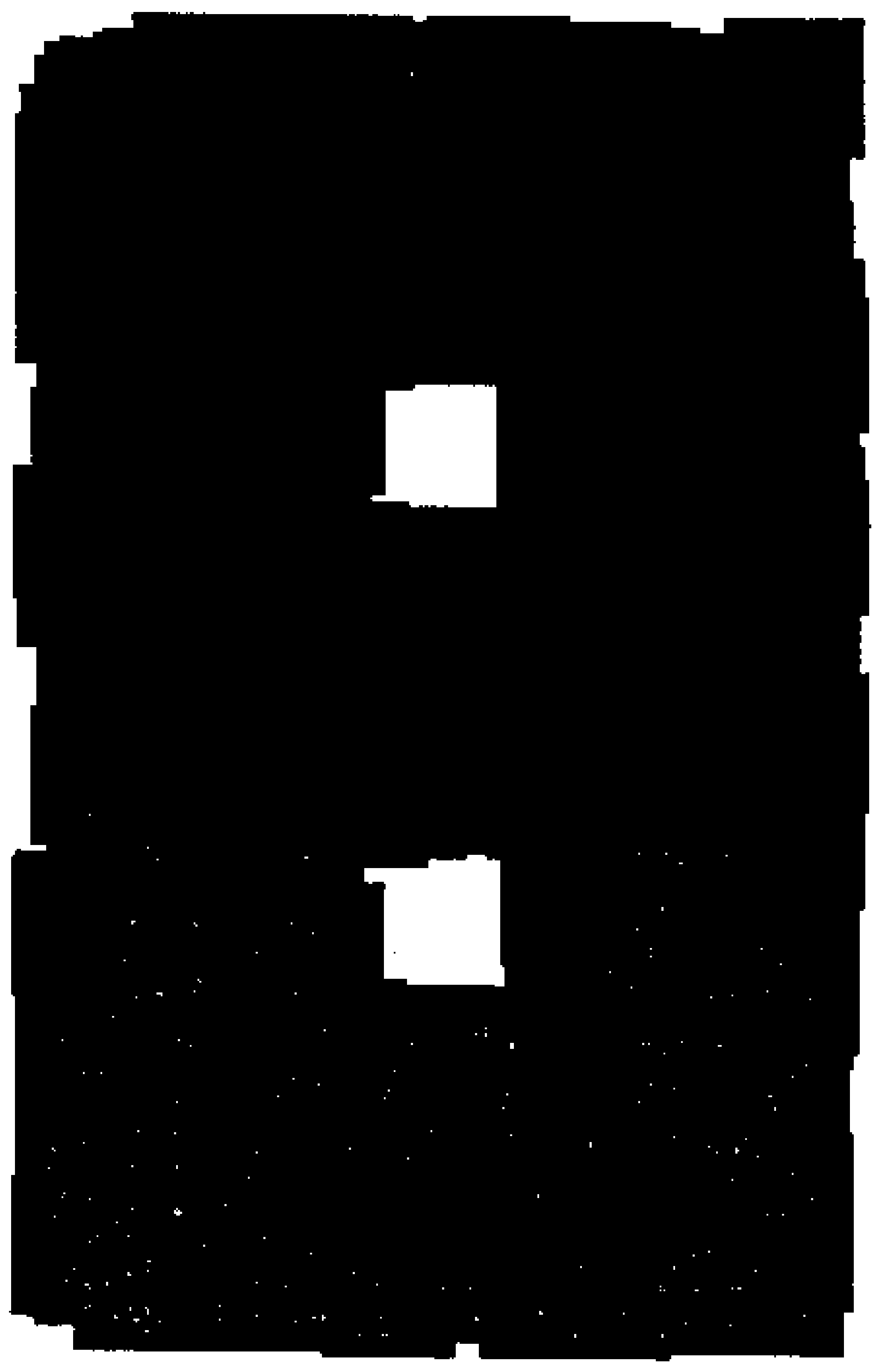
Figure 11: Set **A**(1n) for 1000 points.

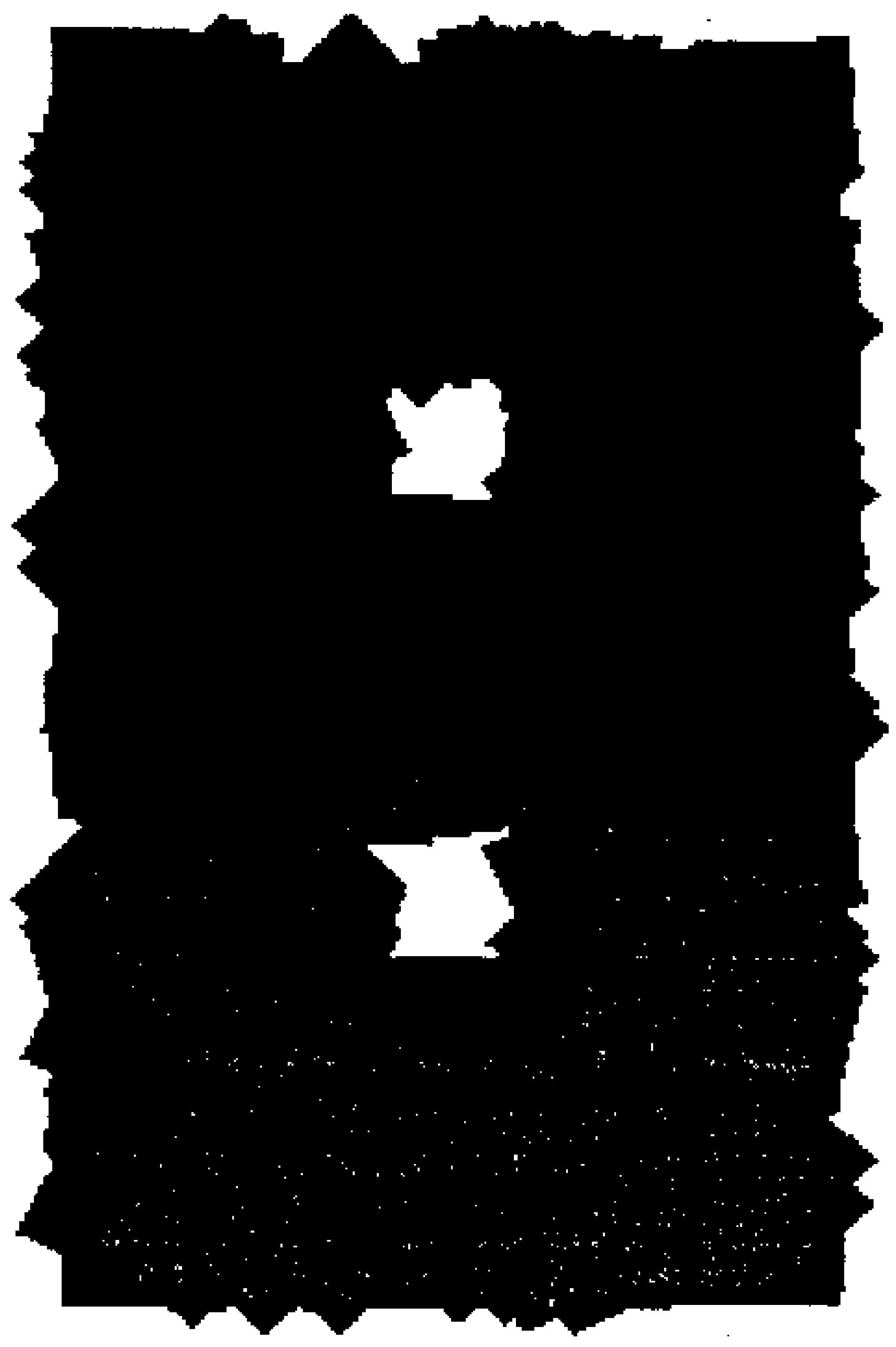
Figure 12: Set **A**(2n) for 1000 points.

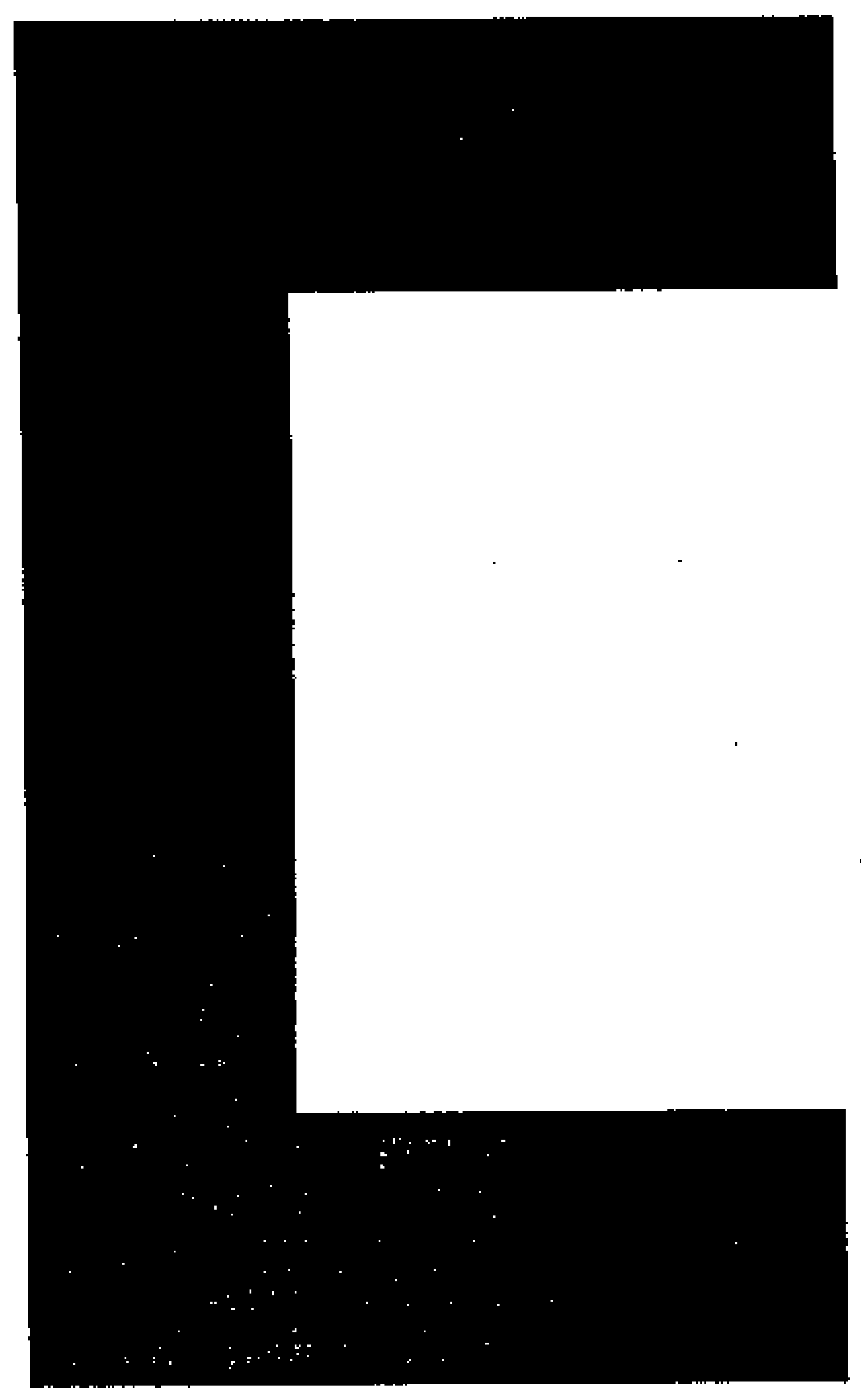

Figure 13: Original set for C.

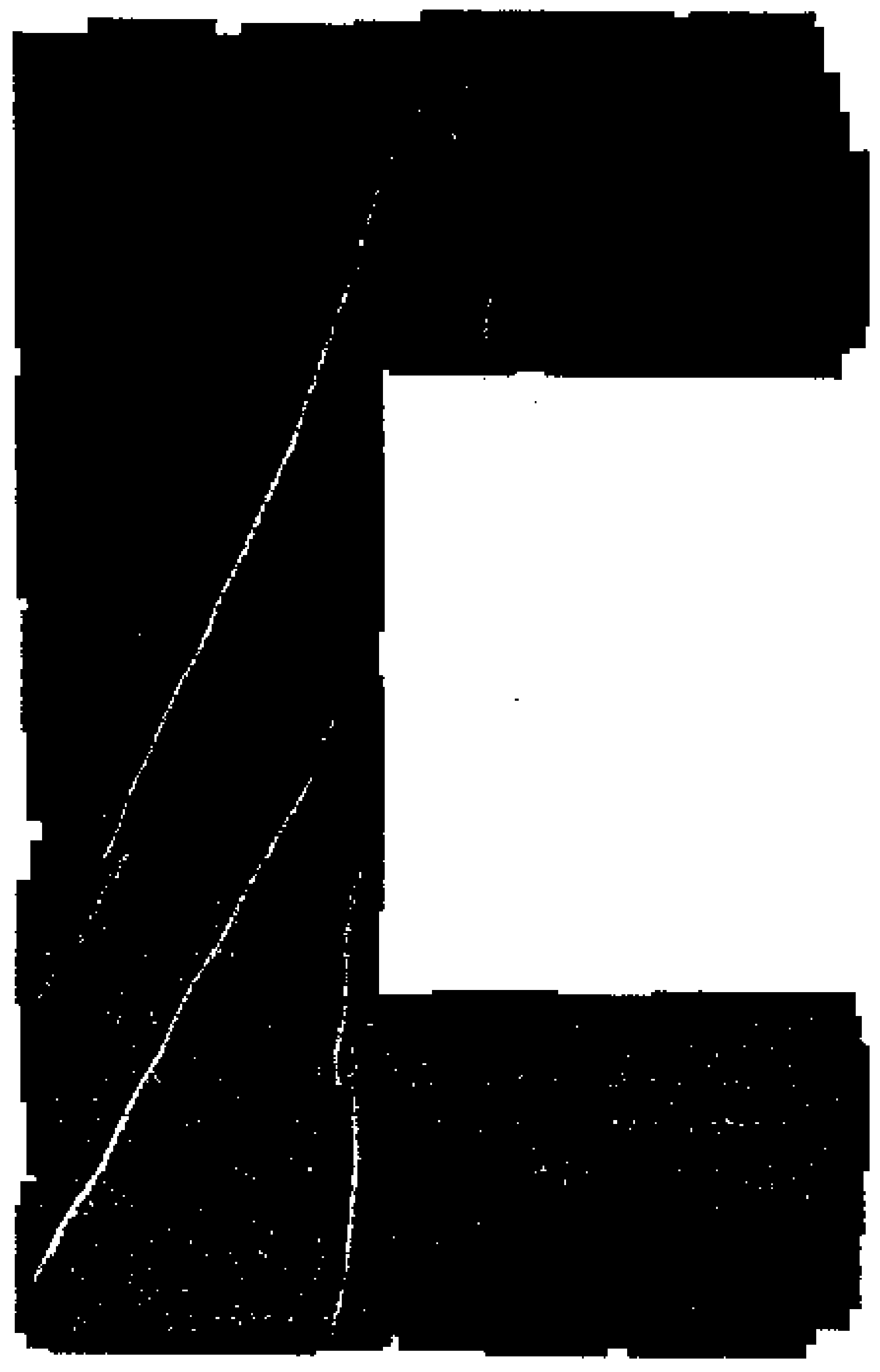
Figure 14: Set **A**(1n) for 1000 points.

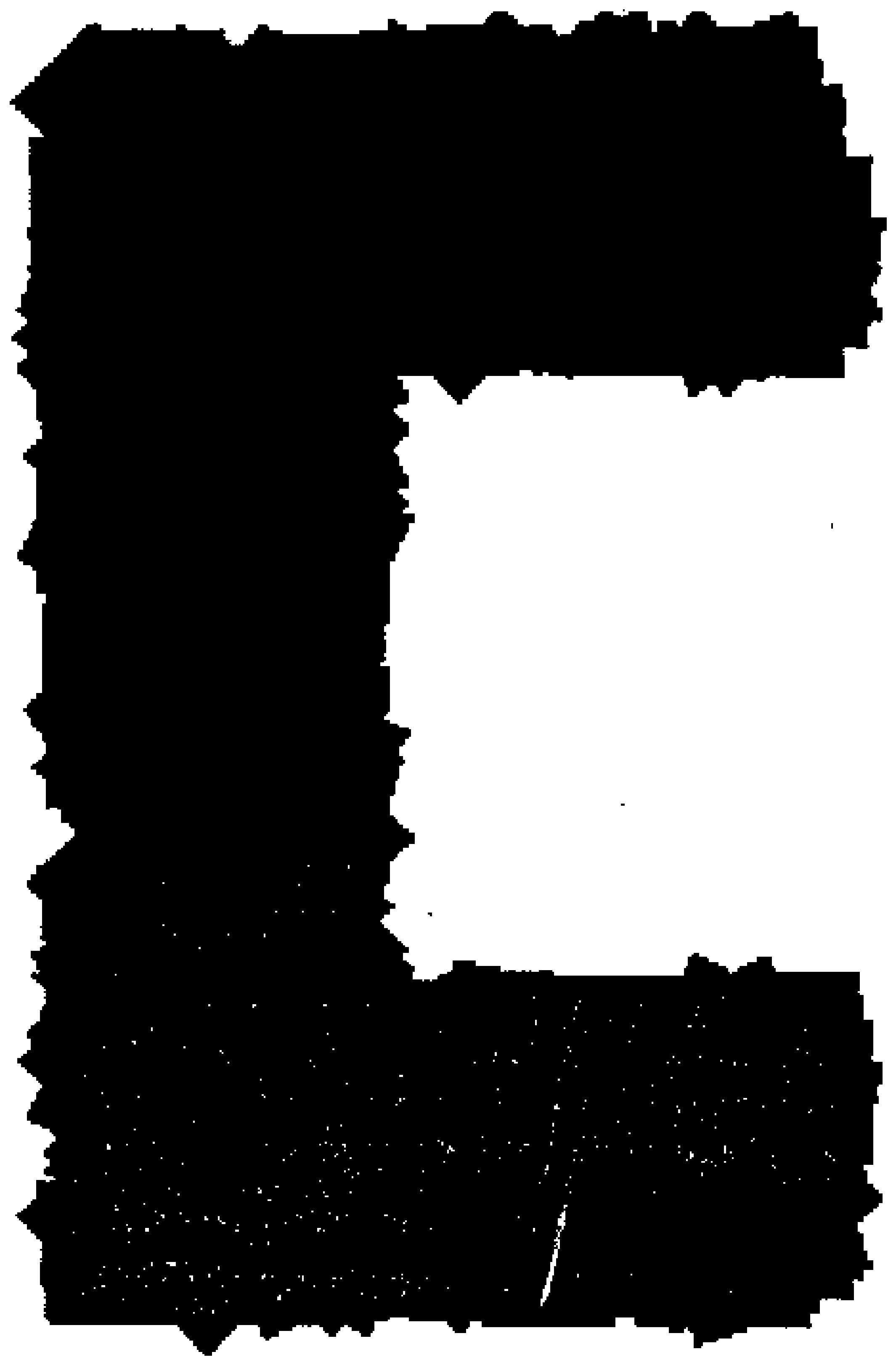
Figure 15: Set **A**(2n) for 1000 points.

TABLE I

No. of Points	h_n	Distance for set A_{1n}	Distance for set A_{2n}
100	0.472	17178	17178
200	0.400	17556	16912
300	0.364	16979	16696
400	0.335	15889	15437
500	0.315	15295	15937
600	0.300	14831	14954
700	0.287	13778	14442
800	0.277	13480	14121
900	0.269	13115	13826
1000	0.262	13156	13203

fig(16) Table for set **P**

TABLE II

No. of Points	h_n	Distance for set A_{1n}	Distance for set A_{2n}
100	0.513	21997	21802
200	0.416	20029	20586
300	0.376	18849	19237
400	0.349	17626	18026
500	0.329	17269	17611
600	0.314	17000	16838
700	0.301	16459	16462
800	0.290	16056	15868
900	0.281	15543	15427
1000	0.274	14551	15042

fig(17) Table for set **8**

TABLE III

No. of Points	h_n	Distance for set A_{1n}	Distance for set A_{2n}
100	0.458	17515	17656
200	0.380	16750	16641
300	0.345	15818	16276
400	0.317	14981	15408
500	0.299	14304	14686
600	0.286	13996	14422
700	0.274	13530	13320
800	0.264	13056	12817
900	0.256	12592	13092
1000	0.247	12186	12627

fig(18) Table for set **C**

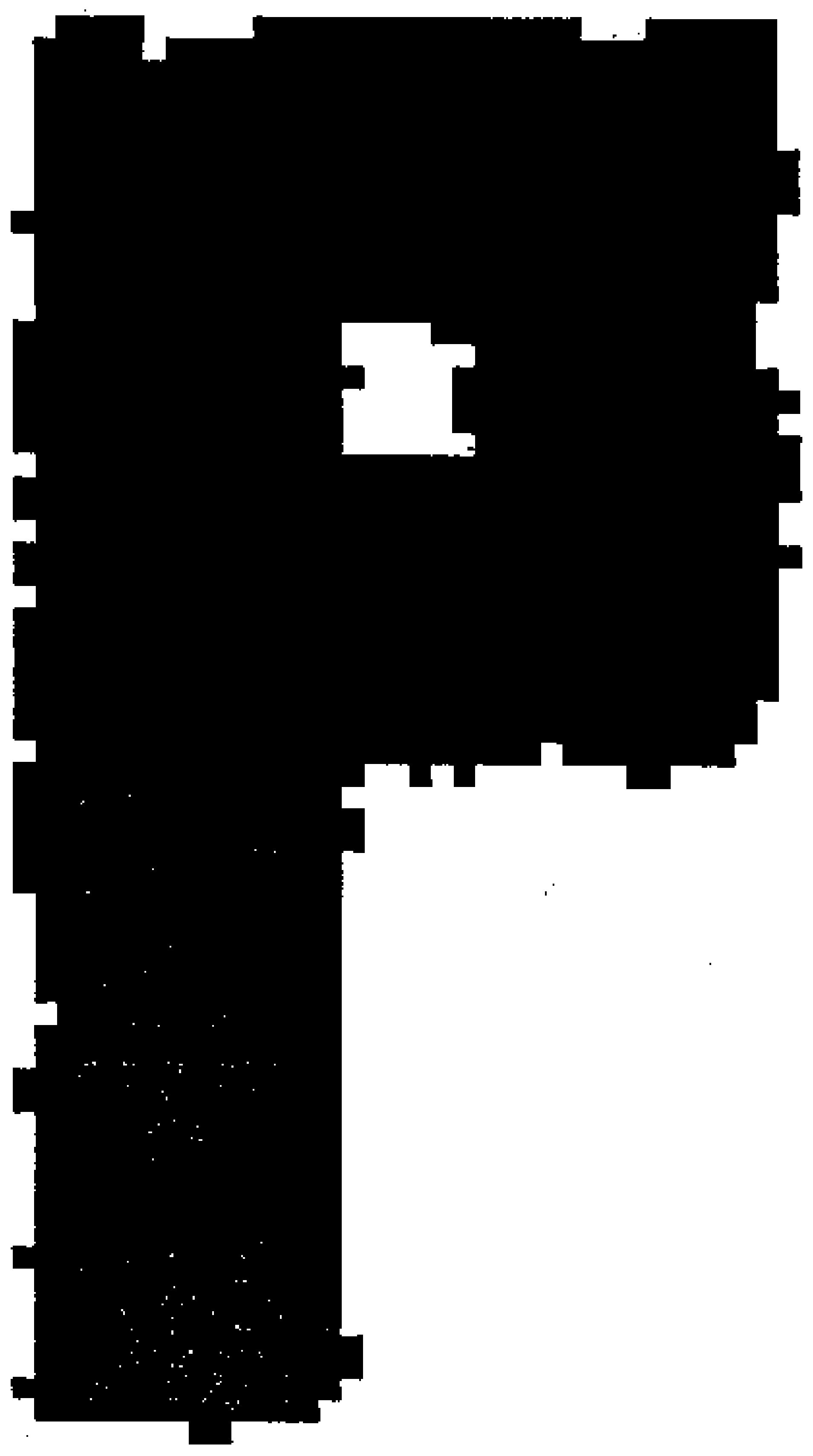
Figure 19: Set **A**(1n) obtained with MLP for **P** with 1000 points

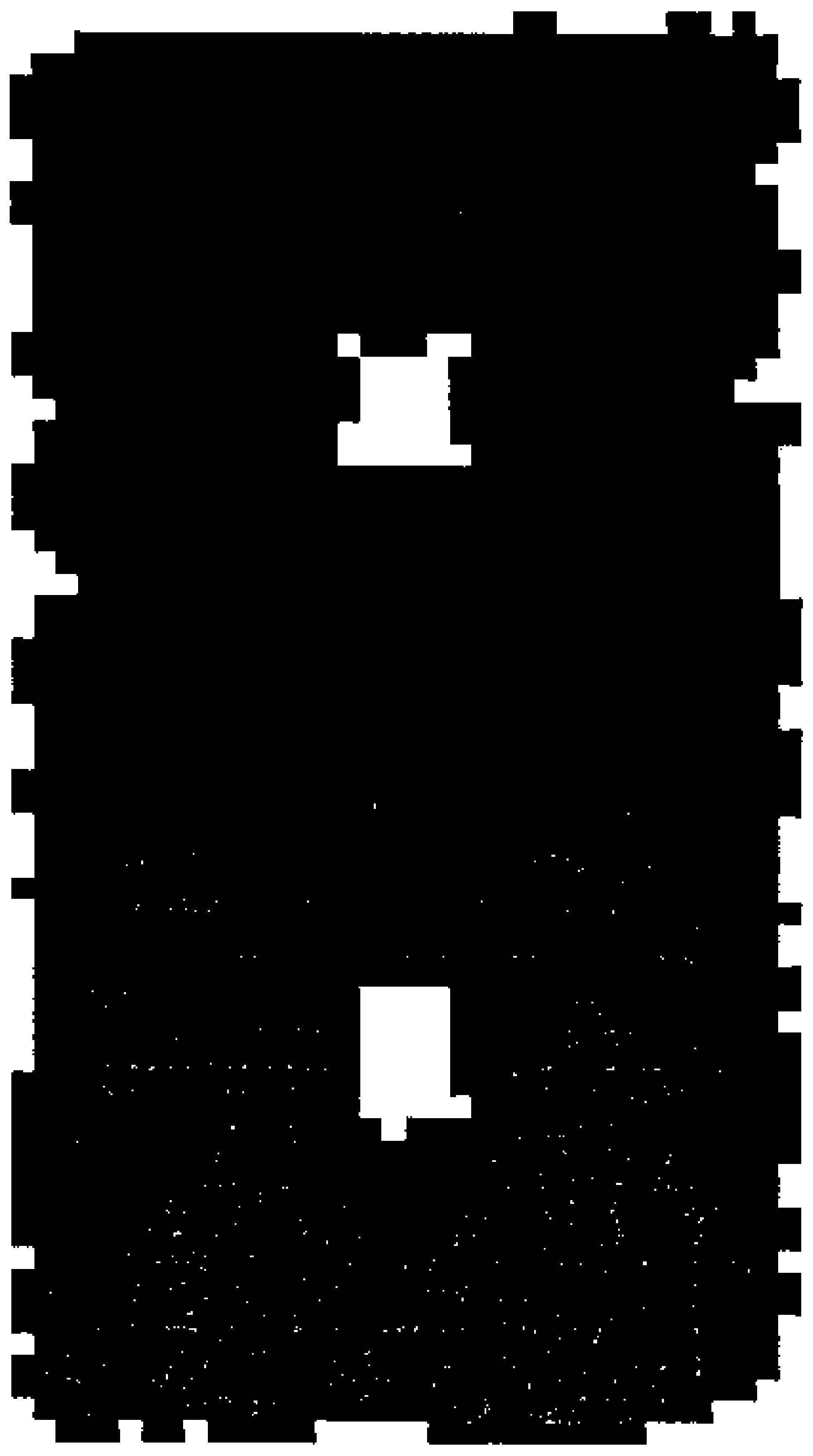
Figure 20: Set **A**(1n) obtained with MLP for **8** with 1000 points.

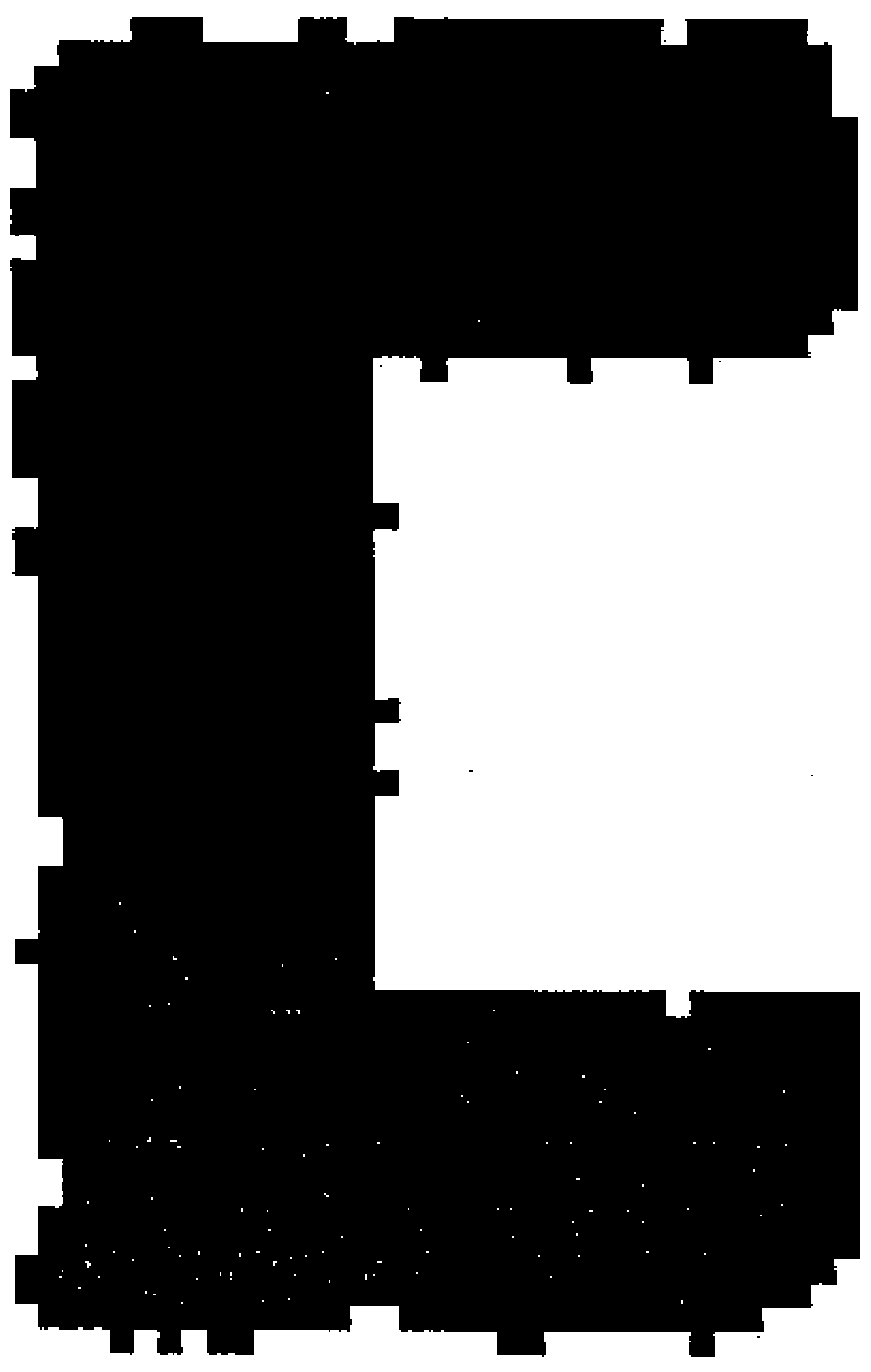

Figure 21: Set **A**(1n) obtained with MLP for **C** with 1000 points.

BIBLIOGRAPHY

1. C A Murthy, *"On Consistent Estimation of Classes in R^2 in the Context of Cluster Analysis* , Ph.D thesis submitted to Indian Statistical Institute,Calcutta, 1988.

2. S G Akl and G T Toussiant, "Efficient convex hull algorithms for pattern recognition applications" , *Proc.* 4th *Intern. Joint Conf on Pattern Recognition*, Kyoto, 1978.

3. R A Jarvis, "Computing the shape hull of a set of points in the plane" , *Proc. of the IEEE Comp. Soc. Conf. on Pattern Recognition and Image Processing*, 1977.

4. J Fairfield, "Contoured shape generation forms that people see in dot patterns" , *Proc. IEEE Conf. on Cybernetics and Society*, 1979.

5. H Edelsbrunner, D G Kirkpatrick and R Seidel, "On the Shape of a Set of Points in the Plane", *IEEE Tr. on Information Theory*, Vol-II-29, July, 1983.

6. F Harary *"Graph Theory"* , Addison Wesley, 1969.

7. R C Lippmann, "An Inrtoduction to Computing with Neural Nets", *IEEE ASSP Magazine* , April, 1987.

8. I Aleksander and H Morton, *An Introduction to Neural Computing*, Chapman and Hall, 1990.

9. J Hertz, A Krough and R G Palmer *Introduction to the Theory of Neural Computation*, Addison-Wesley, 1991.

10. S G Tzafestas and A N Venetsanopoulos (eds), *Fuzzy Reasoning in Information, Decision and Control Systems*, Kluwer Academic Publishers Printed in Netherlands.

11. V Grenander, *Absract Inference*, John Wiley, 1981.