# Comparative Study on the Generation and the Efficiency of Two Types of Laser Marking Hatch Contour

## summary

The main problem of laser printing technology is how to quickly and automatically generate a laser trace with high efficiency and uniform distribution according to the product shape and material in laser printing. In this paper, we use MATLAB to design several functions such as *buffering*, *interpolating* and *sketching* to simulate laser printing and study the generalization of our model. We finally create a model with high efficiency and automatic fast printing.

For the first problem, we design a universal algorithm by designing interpolation functions *buffering*, *interpolating* and *sketching*, and reasonably using MATLAB's own function *polyshape* to construct laser printing border, function *polybuffer* to create buffer, function *perimeter* to get the perimeter.

As for problem 2, problem 2 and problem 1 are essentially a plane image problem, i.e. using parallel lines or contour lines to fill the whole plane image, but the difference is that small circles are added in the frame of problem 2, which makes the image more complex, but there is no difference in essence. So, we use the first question procedure and complete the title requirements.

For the third question: by laser marking a variety of automatically generated graphics and the graphics in the topic, we find our models have good stability and we conclude that:

1、Number of points are too small or sparse: Zigzag hatching

2、With several holes: Contour hatching

3、Number of points are big and diverse, the division interval is small: Contouring

**Keywords: Buffering, Interpolating, Sketching, Zigzag hatching, Contour hatching**

# Contents

# 1 Introduction

## 1.1 Background

Laser printing is an important invention in the 20th century and is widely used in industrial processing. Laser marking refers to Laser Drawing on Product Surface, with high processing efficiency, no contact operation, no consumables, small impact on the surface deformation of the product, marking content solid and other advantages. In order to meet the requirements of high efficiency laser marking, the shadow curve should be parallel and evenly distributed with the boundary line of the figure, and generated automatically and quickly.

Efficiency is an important index when generating shadow graphics. Therefore, it is very important to select and optimize the performance and efficiency of hatching algorithm before printing, which can reduce the printing cost and increase the printing efficiency.

## 1.2 Restatement of the Problem

Parallel hatch in direction and parallel hatch in outline are two basic methods of marking hatch. The hatch curves are required to be uniform and regular and to be basically parallel to each other. Filling omissions are not allowed in a certain area, and repeated filling is not allowed, either. Laser marking should generate contours immediately. In order to meet the requirements of high efficiency laser marking, the shadow curve should be parallel and evenly distributed with the boundary line of the figure, and generated automatically and quickly.

According to the above conditions, we are supposed to solve the following three questions:

- Set internal contraction boundary distance equals to 1mm, hatch spacing equals to 1mm and internal contraction boundary distance equals to 0.1mm, hatch spacing equals to 0.1mm, respectively. Implement zigzag parallel hatch and contour parallel hatches with single-layer profile pattern in annex 1. Calculate the total length of the curve, the number of horizontal lines and the number of cycles. Then calculate the running time ratio of the two hatch ways.

- Set internal contraction boundary distance equals to 1mm, hatch spacing equals to 1mm and internal contraction boundary distance equals to 0.1mm, hatch spacing equals to 0.1mm, respectively. Implement zigzag parallel hatch and contour parallel hatches with single-layer profile pattern in annex 2. Calculate the total length of the curve, the number of horizontal lines and the number of cycles. Then calculate the running time ratio of the two hatch ways.

- Check the running time of the hatching algorithm and analyse its performance. In addition, provide a strategy or direction for optimizing the performance and efficiency of the hatching algorithm so that it can meet the efficiency requirements of practical industrial applications.

# 2 Problem Analysis

## 2.1 Zigzag Parallel Hatch

The essence of the problem is a plane filling problem. The given data are scattered points, we attempt to scan and sketch the graph by interval through ring interpolation (algorithm 1), buffer shrinkage, point set divisions (algorithm 2).

## 2.2 Contour parallel hatch

The essence of the problem is a plane image problem, that is, fill the whole plane image with equidistant contour line. We consider to call the function *polybuffer*（in MATLAB） to realize it. The given data are scattered points, so we first need to form a closed curve by connecting the scattered points, and then calculate the total length, number of circles and average elapsed time (unit: ms) of hatched curves subject of contour parallel hatch.

We divided our work into four steps:

- Loading the two columns of the provided data in $x$ and $y$ respectively.
- Using *polyshape* to sketch the graph.

The loop calls the function *polybuffer* until $polygon.NumRegions\ = 0$, so we could sketch the graph of shrinking contour parallel. Then we apply the function *perimeter* to calculate the perimeter and we could obtain the total perimeter by Repeated accumulation.

- output cycle-index, that is, the number of circles.
- Add *tic*, *toc* before and after the main function to calculate the time.

# 3 Zigzag Parallel Hatch

The *ZigZagMain* function aims at drawing, which can be divided into three subfunctions:

use *buffering* function to divide the whole datasets into several disconnected closed curve which are stored in several datasets.

Independently perform Interpolate function to obtain new datasets corresponding to their original datasets, followed by combining the several new data into a whole new database.

Use the *sketching* function to sketch the curves step by step. Viewers can set parameter '*pausetime*' from 0.1-0.5 if they want to see the simulation of virtual hatching.

Obtain the number of parallel horizontal lines, total length of the lines and several running times of multiple runs at the end of this *ZigZagMain* function.

We first load the two columns of data into x, y to form the coordinates of each point, set the step size spans=1 mm, sketch the border of laser printing target image. Then we call the *buffering* function to sketch the top line of the closed curve and divide the data into many segments. After that, we call the *interpolate* and *sketching* function, then we get the graph:

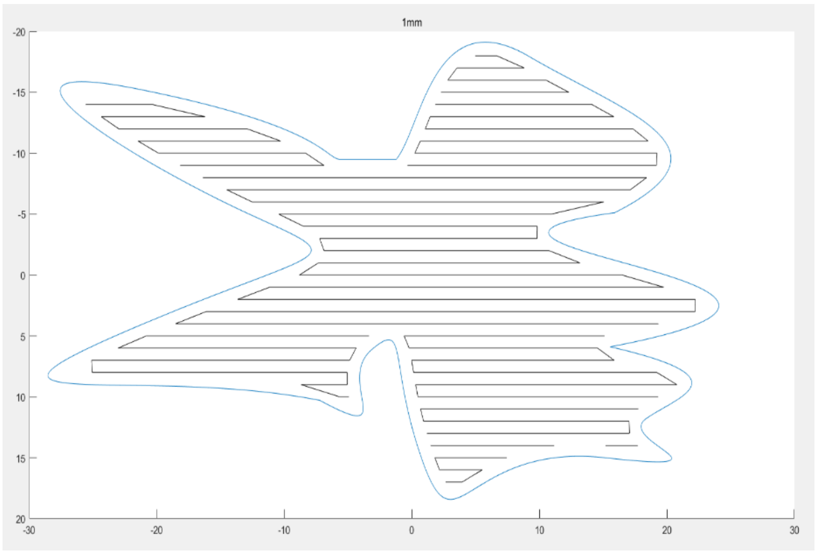


Figure 1

Finally, we output circle-index (the number of circles) and the total circumference. In order to count the running time of the program, we add *tic*, *toc* before and after the main function to calculate the running time of the program. We ran five times and took the average. (see 5 Comparisons)

The flow graph of zigzag parallel hatch is as follow:

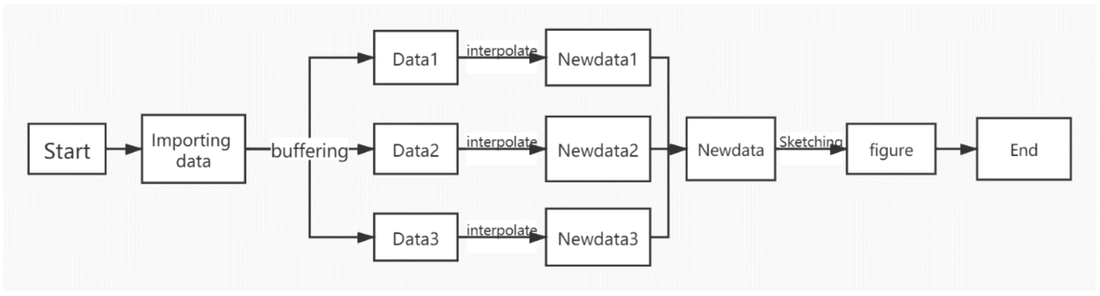


Figure 2

## 3.1 Ring Interpolation Method (Algorithm 1)

### 3.1.1 Step 1-Data Pre-processing on x-axis

We know that the function *interp1*(in MATLAB) can only interpolate a ascending or descending function. Then how to interpolate to get the interpolation of overlapping segments remains a question.

We first deal with the value on x-axis. There exist two adjacent points which are adjacent, but their x-coordinates are the same, which leads to the error of the interpolation function. We decide deal with these points first. The method is to detect whether the x-coordinates of the front point is the same as that of the following point. If so, then we add 0.0001 to the x-coordinate of the preceding point. In this case, so long as the segmentation of the x is not dense enough, we can preliminarily consider that there is no interpolation element between these two points, and there is roughly no deviation from the original straight line.

### 3.1.2 Step 2-Compute the Position Relationship

Calculate the position relationship between the prior point and the following point on x-axis and store this relationship in the $\alpha$ matrix. If $x_i > x_{i+1}$, then $a_i = 0$; if $x_i > x_{i+1}$, then $a_i = 1$. Notably, since the data has been pre-processed in step 1, it is impossible to have $x_i = x_{i+1}$. Calculate all the $a_i$. The total number of $a_i$ should be less than the total number of x, and then we make a comparison between them.

The algorithm we designed tells us, if $a_i = a_{i+1}$, then we do not think they have a steering in the direction of the x-axis; If $a_i \neq a_{i+1}$, then we conclude that there will be a diverse of direction. We need to record the position of the steering point with $j = i + 1$ (the initial value of the $j$ is 1, representing the first interpolation from the point $x_1$), making it convenient in the next interpolation. Then we apply interpolation to the vectors from $x_j$ to $x_i$, the x-coordinates of the points we obtain are in the order that from the lower side to the upper side. Finally, because steering can no longer be detected, we need to manually add an interpolation, from $j$ to *end*, making this interpolation completed.

### 3.1.3 Step 3-Constructing Matrices Storing Interpolated Points

Before each interpolation, we need to construct a matrix to store the interpolated points. We do not need to arrange the interpolation points in order but x-coordinates should correspond to the y coordinates one by one. The new interpolated points are then placed behind the matrix.

### 3.1.4 Step 4-Deleting Unusable Data

We need to delete the corresponding coordinates by finding the location where the *NaN* points appear, indicating that the interpolation points are beyond x-y range and cannot be interpolated.

Note: we know that the system built-in functions have *ceil* and *floor* for upper and lower integration. So, what if you truncate one decimal place? If you use *ceil(x/0.1)*0.1*. For example, for 15.99*10=159.9, we take 160 and divide by 10 so we get 16.0.

## 3.2 Buffering Function (Algorithm 2)

The self-defined function *buffering* can be divided into two parts, one of which is constraining the polygon into a given region and the other is dividing all points into several disconnected closed curve. To implement the first function, create a polyshape object， employing internal function *polyshape* and obtain a new region with *polybuffer*. Then we obtaining vertices of this region. According to the proposition of internal function *polyshape*. Vertices, perform the following tasks on the vertices' set:

If there is only one closed curve, there will be no *NaN* in the vertex's matrix, we solely add the coordinates of the initial point of that curve in the end of that matrices. Else if there are more than one curves, they will be divided by *NaN*. For the first occurring *NaN*, supplant it with the first coordinates of the whole matrices. For the following NaN, supplant it with the next one coordinate of the last coordinate. Add the next one coordinate of the last occurring *NaN*.

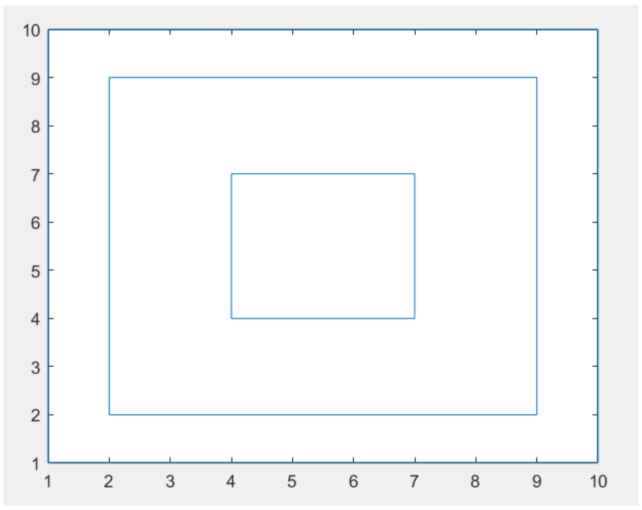


Figure 3

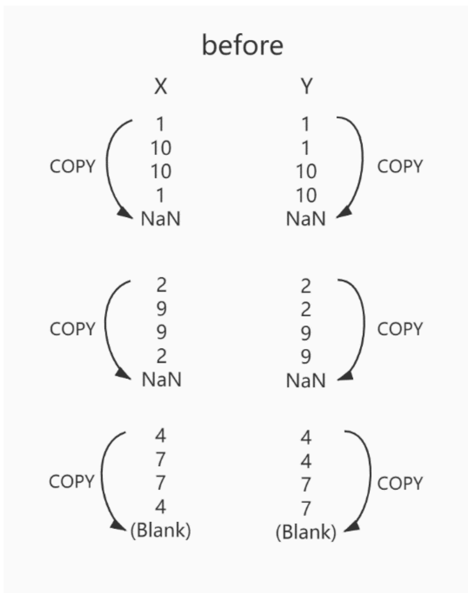


Figure 5

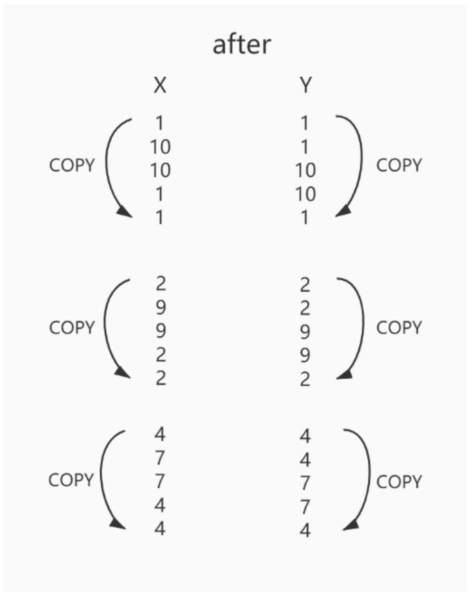


Figure 4

## 3.3 Sketching Function (Algorithm 3)

We define a new function. *Sketching* function is used to sketch the graph. First, we get the smallest and the largest x-coordinates. Since we have obtained the coordinates of all points through the interpolate function, now we begin to classify these points, *find* function can help us find the coordinates of all points where x-coordinates equal to a certain value. On the basis of the number of indexes returned, we temporarily store the coordinates in the *X1* matrix and the *Y1* matrix, then rearrange them in a descending order. Since the x coordinates of all return points are the same, we will no longer need to store X points. Then we store all sequential Y in the cell and calculate the number of times interpolated at the x coordinates, and store them in *points.* As all interpolated points appear in pairs, we will introduce the following algorithm：

1. Creating a list, and add min(x) in the first position of the list.
2. Whenever detecting the change of points, we add the series number of the two points into list.
3. Adding max(x) in the last position of the list.

Example 1:

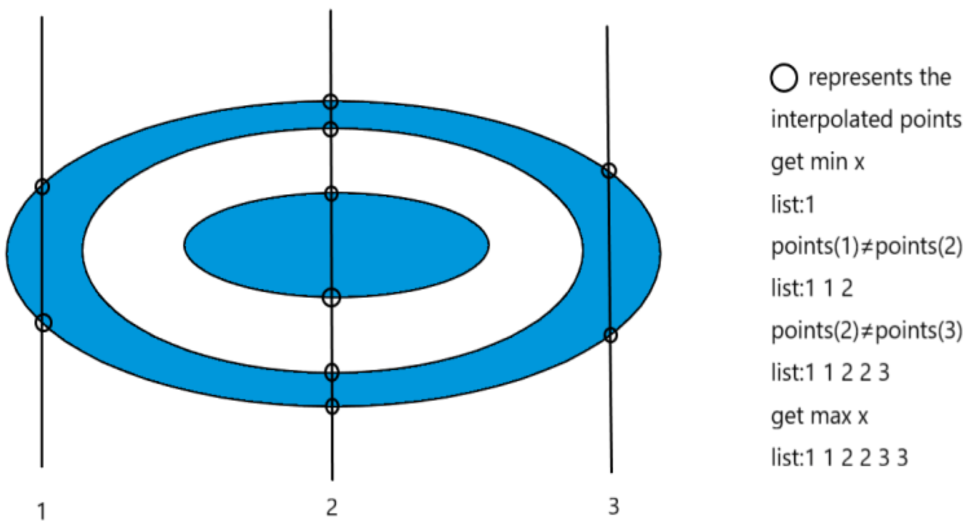


Figure 6

Example 2:

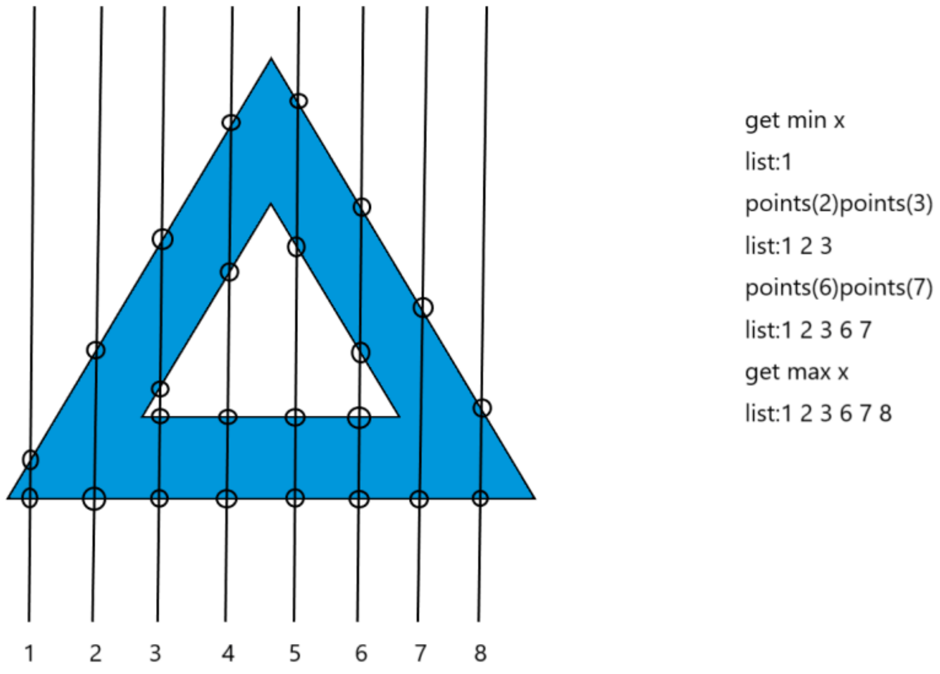


Figure 7

We find that in this case, we could divide any polygon into many small polygons, which could be connected by a zigzag line. After that, we sketch the graph of the two adjacent values from the list and all the points between them, as follows:

Connect the two largest points in the same x coordinates. When x is even, connect the maximum point in the x-coordinate system with the maximum point in the x+1-coordinate system; When x is odd, connect the second largest point in the x-coordinate system with the second largest point in the x+1-coordinate system. We continue this step until reaching the breakpoint in the list. Then we check if there is a third and fourth point, if so, repeat the above, if not, jump into the next interval to sketch the graph.

We also need to calculate the length of the line by calculating the Euclidean distance between the two points. It is also necessary to calculate the number of parallel lines. To simulate a real drawing, we input the pause time at the parameter *pausetime*.

# 4 Contour Parallel Hatch

To implement contour parallel, we use three built-in functions in MATLAB: *polyshape*, *polybuffer*, *perimeter*. We explain them as follows:

- *polyshape* function creates a polygon defined by 2-D vertices, and returns a polyshape object with properties describing its vertices, solid regions, and holes. For example, *pgon = polyshape ([0 0 1 1], [1 0 0 1])* creates the solid square defined by the four points (0,1), (0,0), (1,0), and (1,1).

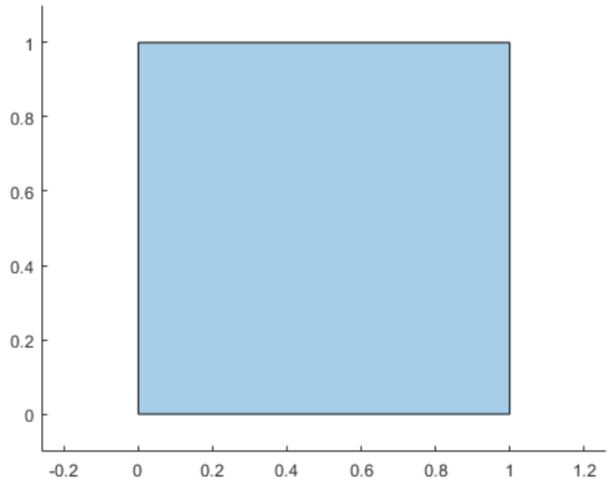


Figure 8

- *polybuffer* function creates a polygon with a solid boundary and a hole boundary and then creates a buffer at a distance of spans from the boundaries. By default, the buffer has rounded joints.

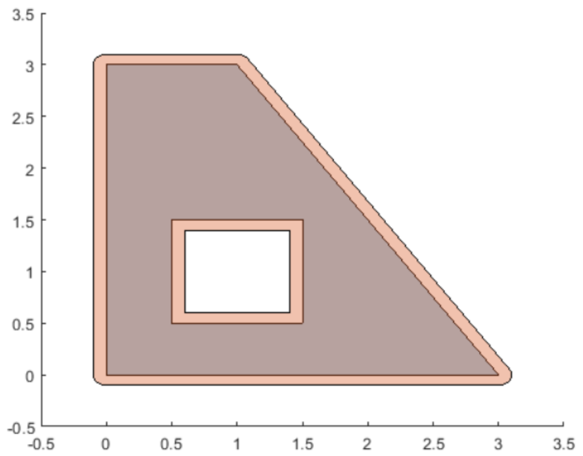


Figure 9

- *Perimeter* function: $P$ = *perimeter(polyin)* returns the perimeter of a polyshape object, which is the sum of the lengths of its boundaries.

## 4.1 Solution to problem 1

The flow chart of our work is shown as follow:

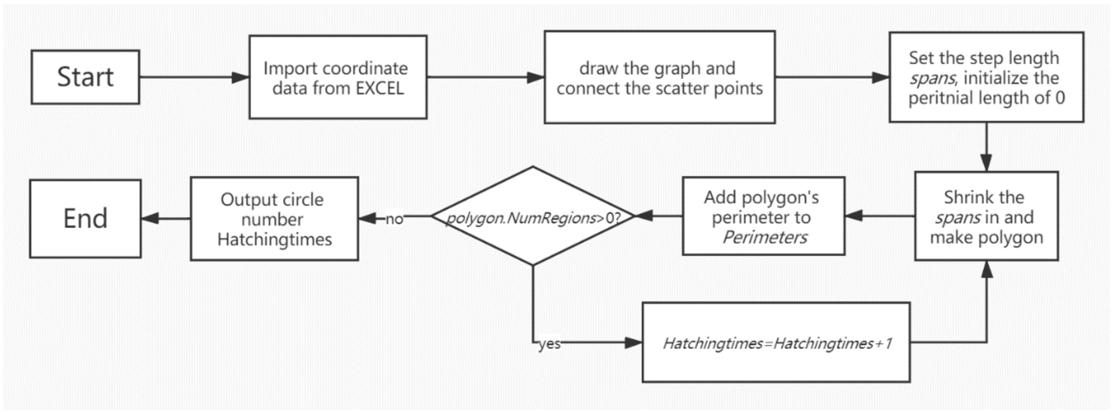

Start
Import coordinate data from EXCEL
draw the graph and connect the scatter points
Set the step length *spans*, initialize the peritnial length of 0
Shrink the *spans* in and make polygon
Add polygon's perimeter to *Perimeters*
*polygon.NumRegions>0?*
no
yes
Output circle number Hatchingtimes
End
*Hatchingtimes=Hatchingtimes+1*

Figure 10

We first load the two columns of data（from graph 1）into $x, y$ to form the coordinates of each point. We set the step size $span = 1mm$, and then call the build-in function *polyshape* (in MATLAB) to sketch of laser printing target image (as shown in the graph).

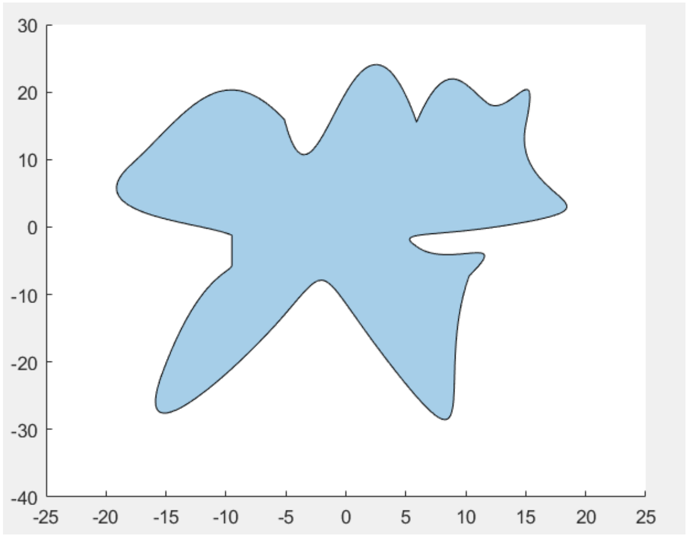


Figure 11

Next, we call the build-in function *polybuffer* to get the next contour parallel. Then we call the function *perimeter* to calculate its perimeter (as shown in the graph).

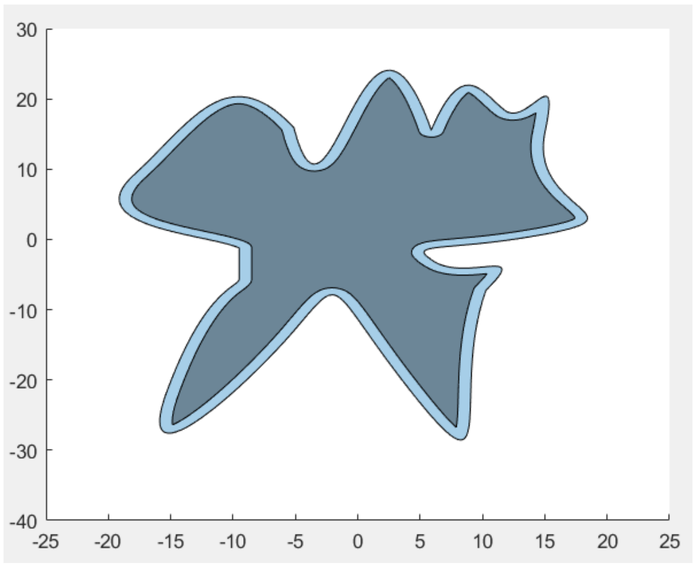


Figure 12

Then, we repeat the previous step until *polygon.NumRegions*=0 and add the circumference of each circle and we can get the total circumference Perimeters(as shown in the graph).

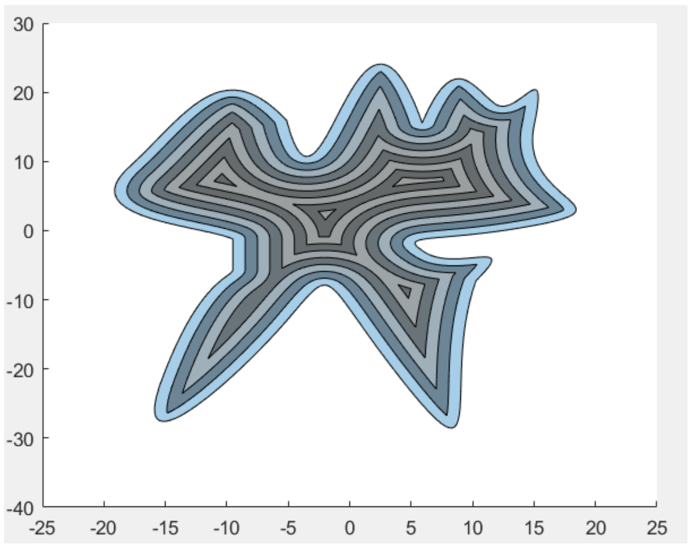


Figure 13

Finally, we output the cycle-index (the number of circles). We get

Cycle-index=9
Perimeter=896.82mm.

In order to count the running time of the program, we add *tic*, *toc* before and after the main function to calculate the running time of the program. We have run ten times and then took the average.

T=(232.928+243.392+242.021+232.562+256.766+232.981+231.775+254.642+241.7

65+239.580)/10=240.841（ms）

Then we set the step size $span = 0.1mm$, applying the same method to sketch the graph and calculate the time (as shown in the graph).

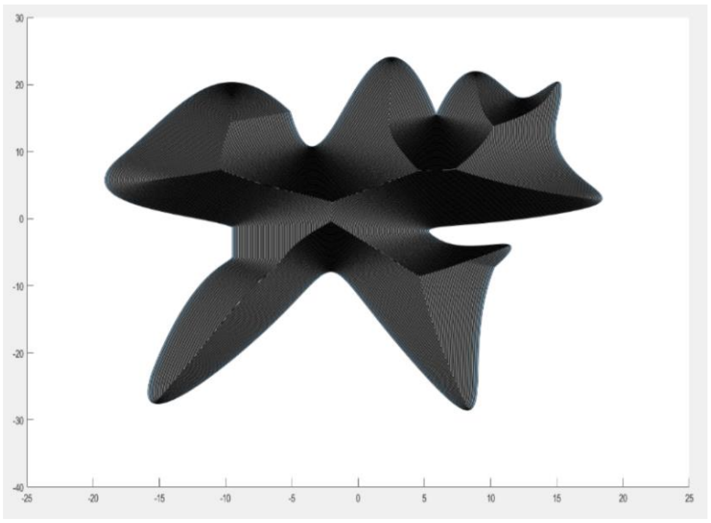


Figure 14

Circle-index: 86
Circumference :1001.71(mm)
T=(1930.140+1961.109+1924.476+1963.719+1945.641+1922.202+1962.207+1952.135+1955.216+1915.281)/10=1943.213（ms）

## 4.2 Solution to problem 2

Both the contour parallel of question two and question one are essentially a plane image problem, that is, filling the whole plane image with equidistant contours, but the difference is that small circles are added to the boarder to make the image more complex, but there is no difference in nature. So we follow the steps in the first problem.

Loading the data in the graph2 into *x* and *y* to form the coordinates of each point, and the spans is set to be 1 mm. Applying the algorithm, we have:

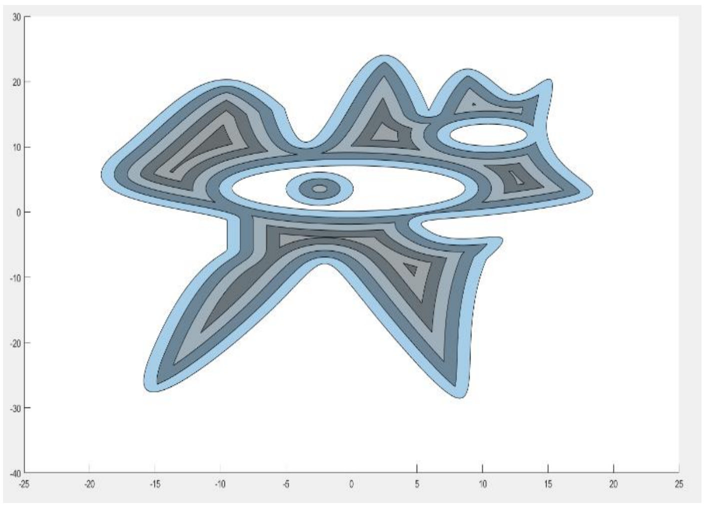


Figure 15

Circle-index=6
Perimeter=773.05

T=(187.548+190.625+180.138+182.871+184.797+183.247+181.536+183.387+185.681+214.610)/10=187.444（ms）

Set *spans* to be 0.1mm, applying the algorithm, we have:

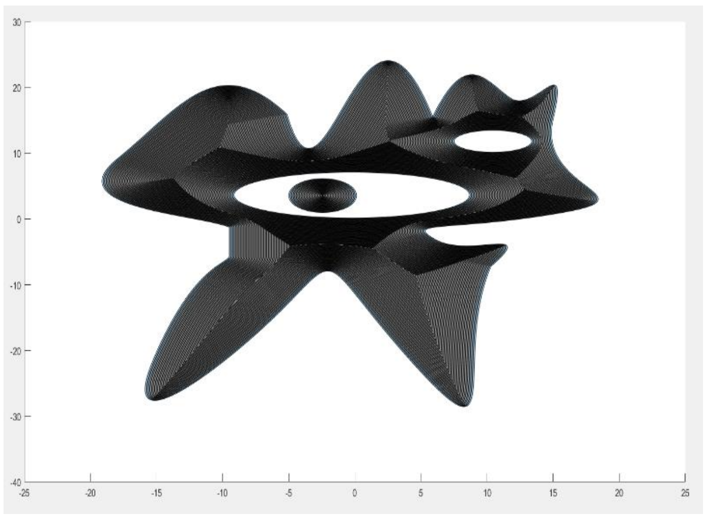


Figure 16

Circle-index=60
Perimeter=9060.12mm

T=(1728.301+1673.761+1705.029+1662.201+1652.007+1721.213+1816.685+1744.420+1713.214+1831.593)/10=1724.842（ms）

# 5 Comparisons

We make comparisons between these two ways.

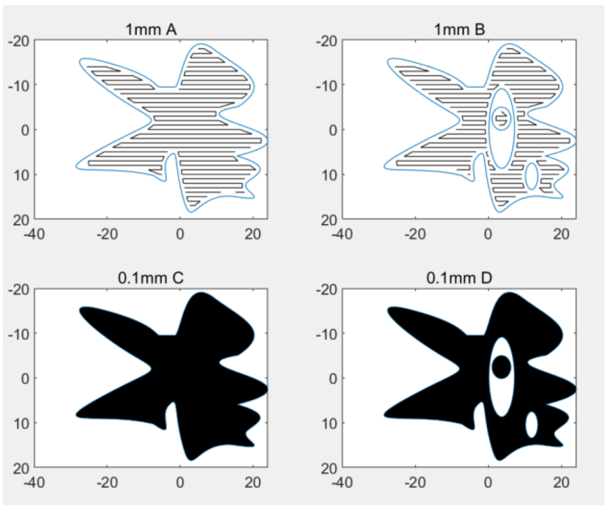


Figure 17

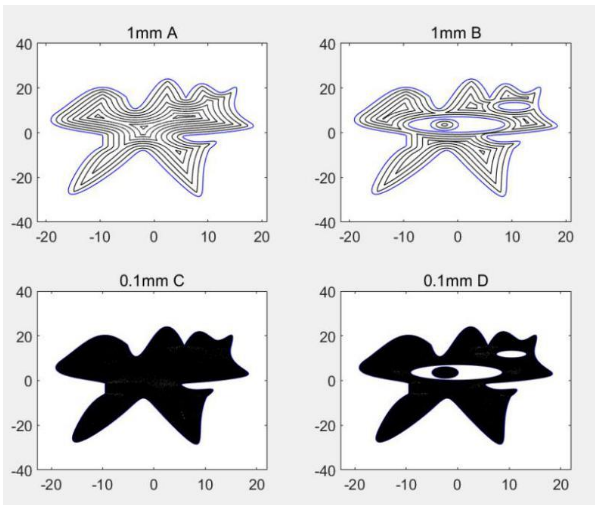


Figure 18

We provide several videos in supporting materials.
Zigzag Parallel Hatch：

| | Length(mm) | lines | Time 1(ms) | Time 2(ms) | Time 3(ms) | Time 4(ms) | Time 5（ms） | Time(average) |
|---|---|---|---|---|---|---|---|---|
| **A** | 867.2227 | 49 | 9223.93 | 9025.72 | 9416.69 | 9565.16 | 9557.71 | 9357.84 |
| **B** | 100.1310 | 504 | 13764.17 | 13964.98 | 13880.26 | 14134.51 | 14878.58 | 14124.50 |
| **C** | 713.3562 | 80 | 16889.17 | 15305.69 | 13716.23 | 27744.85 | 13104.83 | 17352.15 |
| **D** | 904.8230 | 794 | 41239.57 | 42068.01 | 38938.10 | 57207.24 | 60989.86 | 48088.56 |

Figure 19

Contour Parallel Hatch：

| | Length(mm) | circles | Time 1(ms) | Time 2(ms) | Time 3(ms) | Time 4(ms) | Time 5（ms） | Time(average) |
|---|---|---|---|---|---|---|---|---|
| **A** | 895.92 | 10 | 2849.66 | 2971.85 | 2959.20 | 1037.65 | 2732.33 | 2510.64 |
| **B** | 772.77 | 7 | 317.76 | 316.36 | 318.45 | 436.44 | 262.67 | 330.34 |
| **C** | 100.05 | 87 | 51605.46 | 51766.65 | 52826.01 | 52165.61 | 52201.82 | 52113.11 |
| **D** | 905.83 | 61 | 29049.67 | 29602.98 | 29588.86 | 29104.65 | 30136.38 | 29496.51 |

Figure 20

# 6 Generalization of the Model

By concluding several results, we conclude the following statements:

1. In terms of the figures with several holes, ZigZag performs much less efficient than Contouring. The reason for it is that the holes in the figures considerably increase

the division times (that is the number of divided intervals) in sketching functions, which results in a longer calculation time. On the other side, contour function greatly benefits from holes since they can decrease its circles, an effective calculation as a result.

2. For those points lie sparse in the plane, the given points are few, or the distance between lines is too small.

Comparing the following examples:

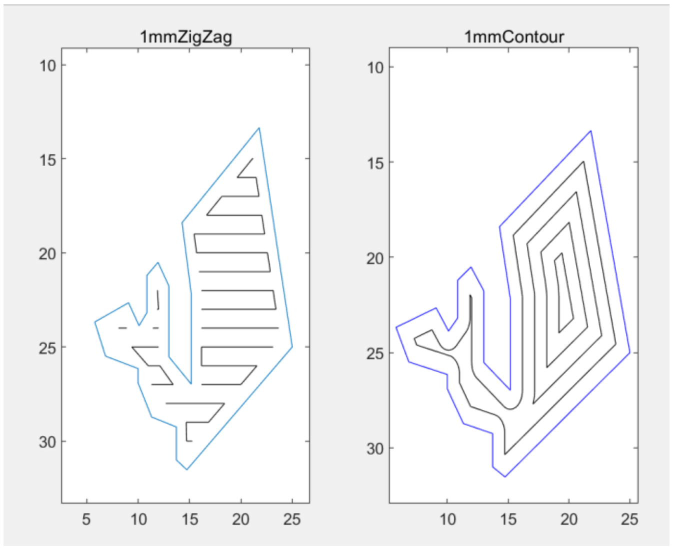


Figure 22

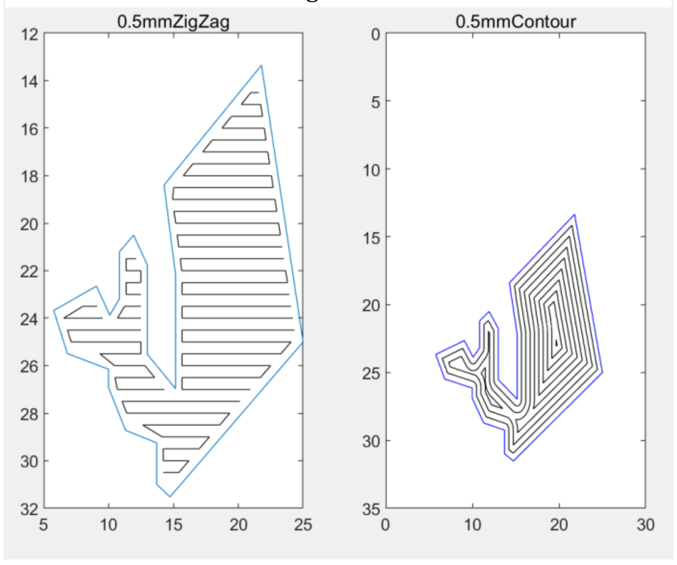


Figure 21

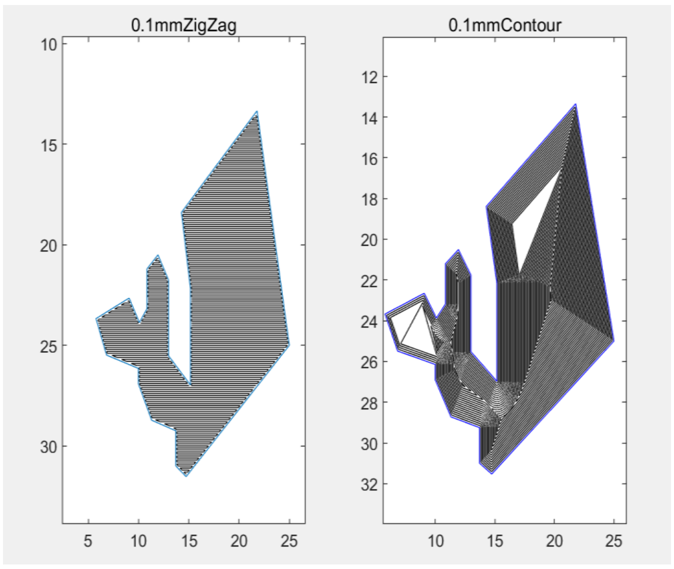


Figure 23

With the increase of division precision, the Contour tends to make mistakes. The reason originates from the fact that the some given points are far from centroid.

Comparing B and D in chapter 5 (figures with holes) in two hatching ways, we discover that the time cost by the Contour is much less than Zigzag.

Comparing C in two hatching ways, we discover that the time cost by the Contour is still less than ZigZag, from which we conclude that for a large given point set and a small division interval, Contouring is still better than Zigzag.

To summarize:

**Given a set of points:**

1. **Number of points are too small or sparse: Zigzag hatching**
2. **With several holes: Contour hatching**
3. **Number of points are big and diverse; the division interval is small: Contouring**

# 7 Conclusion

By designing interpolation functions *buffering*, *interpolate* and *sketching* functions and reasonably using MATLAB's build-in function (*polyshape* to construct laser printing border, polybuffer function to create buffer, perimeter function to obtain perimeter, etc), we designed a model which can generate contour line, maximize uniform distribution, and automatically print. The comparison and optimization of hatch efficiency provide the strategy of optimization hatching algorithm.

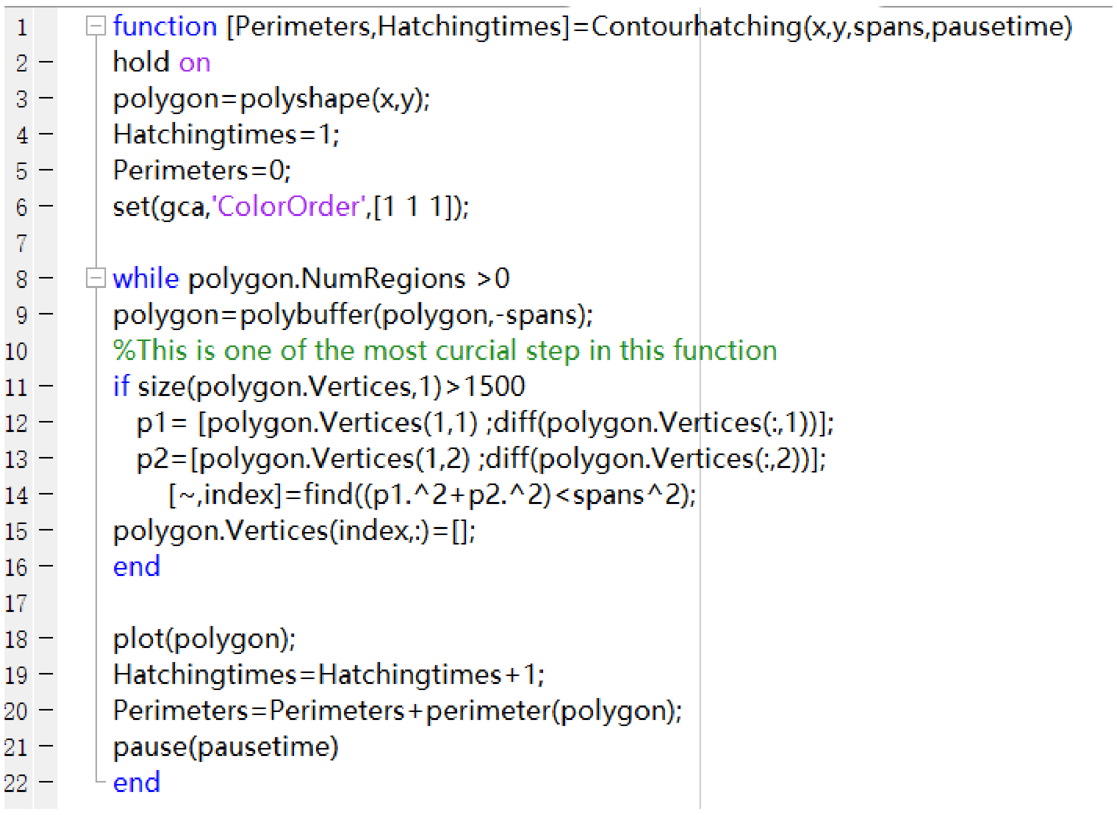
At the same time, we also generated a variety of complex models to detect the stability of the algorithm, and found that our models have good stability. It is of great significance to optimize the laser printing algorithm to meet the efficiency requirements of practical industrial applications.

# 8 Reference

1、CAO Zheng, LU Ying etc. Study of Laser-Marking Properties and Mechanism of PBT/SnO2 [J]. Journal of Changzhou University, 2016, 2095-0411.
2、Qiang Wang. Research on dynamic laser marking [J]. Huazhong University of Science & Technology, 2008.
3、FENG Chi, WU Wenjing etc. Research Progress in Laser Marking on Plastics [J]. CHINA PLASTICS, 2011, 1001-9278.
4、ZHAO Wei, SA Yu. Research of laser mark system [J]. Journal of Tianjin University of Technology and Education, 2006, 1673-1018.
5、Cao Ronghua, Qiu Huadong etc. Quality Assessment and Technical Parameter Optimization of Laser Direct Part Marking [J], APPLIED LASER, 2011, 10.3788.
6、ZHU Fang-yuan, ZHU Xing-long, ZHOU Ji-ping. A Method of Fitting Laser Dot by Least Square Circle [J]. Yangzhou University, 2006, 1671-5276.
7、WAN Chenghui, CHENG Xiaojun, CHENG Xiaolong. Contours Generated by Rapid Prototyping Technology [J]. JOURNAL OF TONGJI UNIVERSITY, 2013, 0253-374.
8、ZHU Chongxun, DENG Honggui. Design of 2D Curve Graphic in VB [J]. Zhongnan University, 2000.

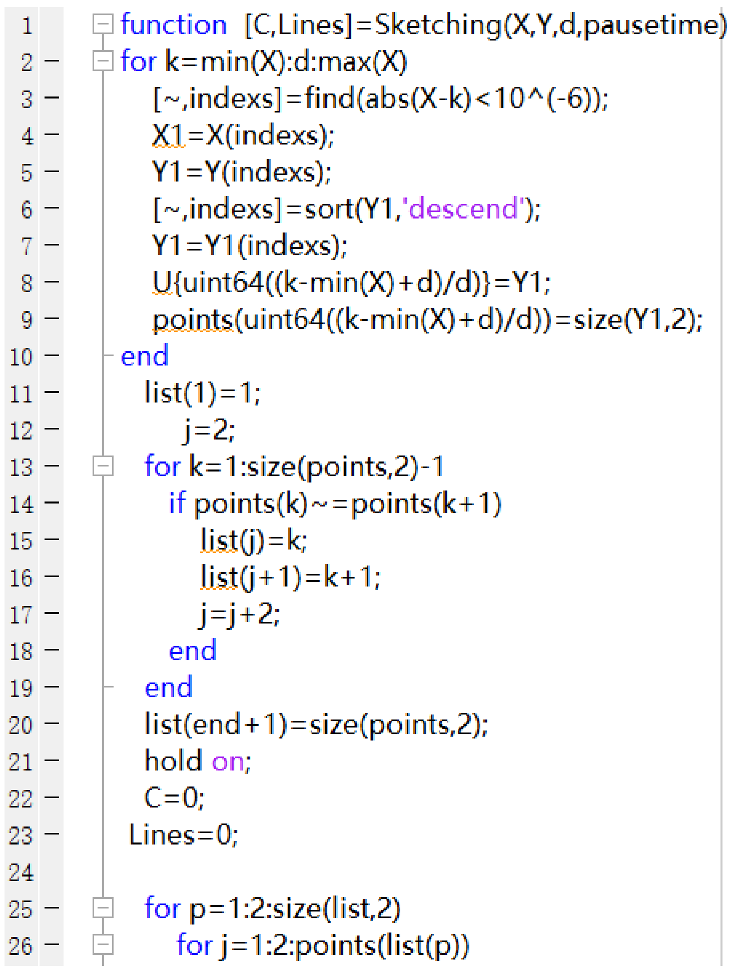
# 9 Appendix

Contour hatching

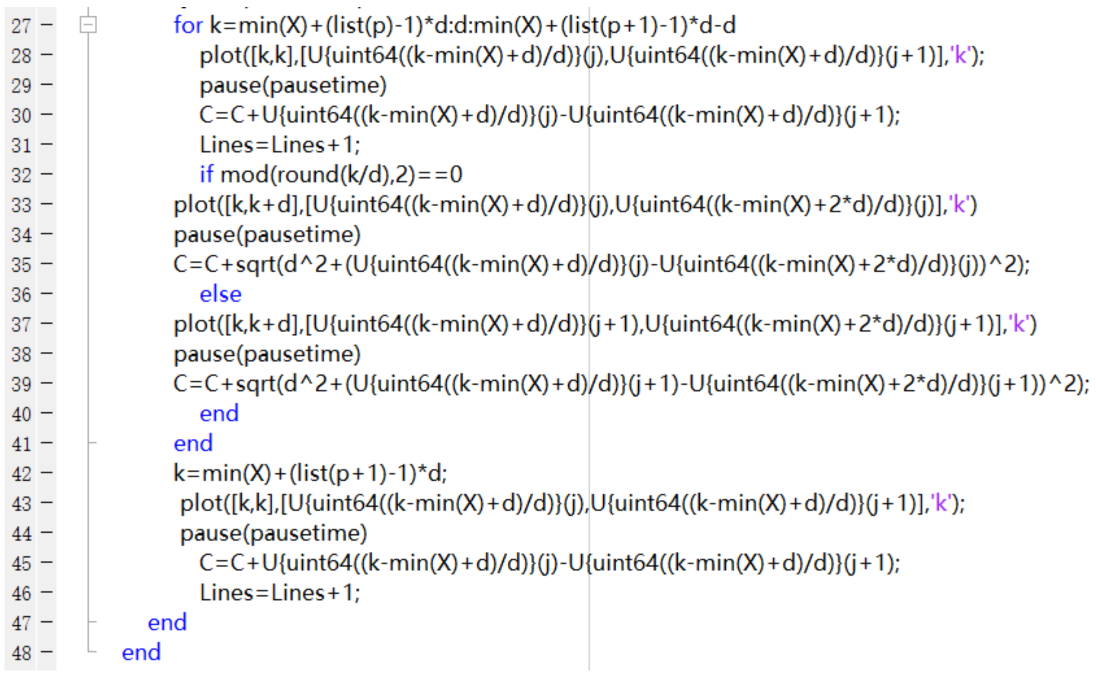


```matlab
function [Perimeters,Hatchingtimes]=Contourhatching(x,y,spans,pausetime)
hold on
polygon=polyshape(x,y);
Hatchingtimes=1;
Perimeters=0;
set(gca,'ColorOrder',[1 1 1]);

while polygon.NumRegions >0
polygon=polybuffer(polygon,-spans);
%This is one of the most curcial step in this function
if size(polygon.Vertices,1)>1500
  p1= [polygon.Vertices(1,1) ;diff(polygon.Vertices(:,1))];
  p2=[polygon.Vertices(1,2) ;diff(polygon.Vertices(:,2))];
     [~,index]=find((p1.^2+p2.^2)<spans^2);
polygon.Vertices(index,:)=[];
end

plot(polygon);
Hatchingtimes=Hatchingtimes+1;
Perimeters=Perimeters+perimeter(polygon);
pause(pausetime)
end
```

## Sketching

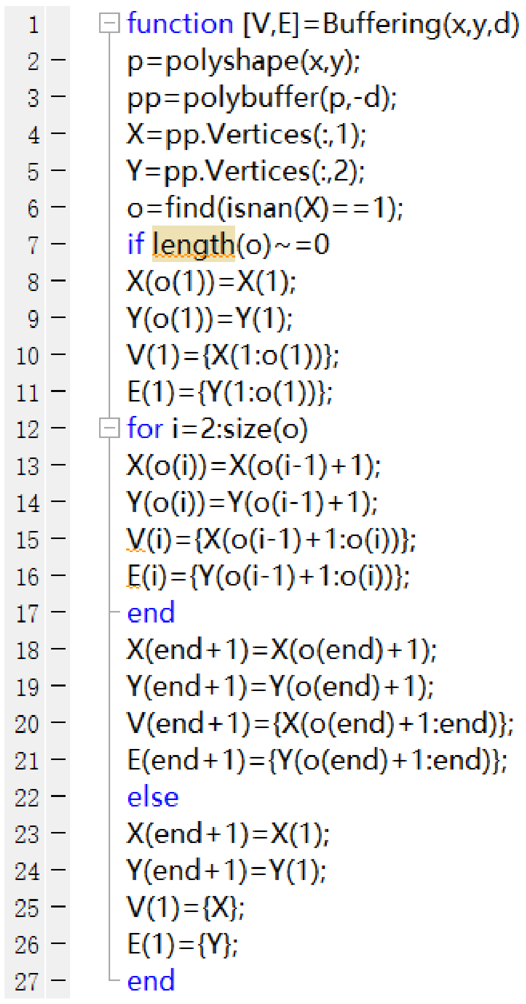


```
function  [C,Lines]=Sketching(X,Y,d,pausetime)
for k=min(X):d:max(X)
    [~,indexs]=find(abs(X-k)<10^(-6));
    X1=X(indexs);
    Y1=Y(indexs);
    [~,indexs]=sort(Y1,'descend');
    Y1=Y1(indexs);
    U{uint64((k-min(X)+d)/d)}=Y1;
    points(uint64((k-min(X)+d)/d))=size(Y1,2);
end
  list(1)=1;
     j=2;
  for k=1:size(points,2)-1
     if points(k)~=points(k+1)
        list(j)=k;
        list(j+1)=k+1;
        j=j+2;
     end
  end
  list(end+1)=size(points,2);
  hold on;
  C=0;
 Lines=0;

  for p=1:2:size(list,2)
      for j=1:2:points(list(p))
```



```
          for k=min(X)+(list(p)-1)*d:d:min(X)+(list(p+1)-1)*d-d
             plot([k,k],[U{uint64((k-min(X)+d)/d)}(j),U{uint64((k-min(X)+d)/d)}(j+1)],'k');
             pause(pausetime)
             C=C+U{uint64((k-min(X)+d)/d)}(j)-U{uint64((k-min(X)+d)/d)}(j+1);
             Lines=Lines+1;
             if mod(round(k/d),2)==0
          plot([k,k+d],[U{uint64((k-min(X)+d)/d)}(j),U{uint64((k-min(X)+2*d)/d)}(j)],'k')
          pause(pausetime)
          C=C+sqrt(d^2+(U{uint64((k-min(X)+d)/d)}(j)-U{uint64((k-min(X)+2*d)/d)}(j))^2);
             else
          plot([k,k+d],[U{uint64((k-min(X)+d)/d)}(j+1),U{uint64((k-min(X)+2*d)/d)}(j+1)],'k')
          pause(pausetime)
          C=C+sqrt(d^2+(U{uint64((k-min(X)+d)/d)}(j+1)-U{uint64((k-min(X)+2*d)/d)}(j+1))^2);
             end
          end
          k=min(X)+(list(p+1)-1)*d;
           plot([k,k],[U{uint64((k-min(X)+d)/d)}(j),U{uint64((k-min(X)+d)/d)}(j+1)],'k');
           pause(pausetime)
             C=C+U{uint64((k-min(X)+d)/d)}(j)-U{uint64((k-min(X)+d)/d)}(j+1);
             Lines=Lines+1;
      end
  end
```

Buffering



```
function [V,E]=Buffering(x,y,d)
p=polyshape(x,y);
pp=polybuffer(p,-d);
X=pp.Vertices(:,1);
Y=pp.Vertices(:,2);
o=find(isnan(X)==1);
if length(o)~=0
X(o(1))=X(1);
Y(o(1))=Y(1);
V(1)={X(1:o(1))};
E(1)={Y(1:o(1))};
for i=2:size(o)
X(o(i))=X(o(i-1)+1);
Y(o(i))=Y(o(i-1)+1);
V(i)={X(o(i-1)+1:o(i))};
E(i)={Y(o(i-1)+1:o(i))};
end
X(end+1)=X(o(end)+1);
Y(end+1)=Y(o(end)+1);
V(end+1)={X(o(end)+1:end)};
E(end+1)={Y(o(end)+1:end)};
else
X(end+1)=X(1);
Y(end+1)=Y(1);
V(1)={X};
E(1)={Y};
end
```