《医学图像处理期末实验报告》

年级：11 级生物医学工程

姓名：谢靖伦

学号：11311048

日期：2014 年 6 月 28 日

摘要

实验任务：

a). 对图 12.18(a1)和(a2)提取边界，把边界坐标看作复数生成傅里叶描绘子，通过减小描绘子的个数，寻找能够保证边界能够被辨识的最短描绘子。

b). 就像你在任务 a 中做的一样，选择能够保留两幅图片基本差异的最小数值以进行描述。这么做能够各自为每幅图片产生一个傅里叶描述子。将每个系数集表示为向量，$\boldsymbol{x}_A=(x1, x2,\ldots,xn)^{\mathrm{T}}$ $\boldsymbol{x}_B=(x1, x2,\ldots,xn)^{\mathrm{T}}$

其中 n 是描绘子的长度。**注意，两幅图应该有相同的描绘子长度**

c).对上述两个向量的每个元素添加噪声（参考任务 05-01），以产生两个**模式类**，class A 和 class B。噪声类型为高斯白噪声，均值为 0，标准差为对应向量中最大元素值的 1/10。这样，产生类 A 和 B 的 100 个样本。这 200 个样本集合称为训练集。用这种方法再分别为类 A 和 B 产生 100 个样本，称为测试集。

d).实现感知机分类器

一、技术讨论

1.1 实验原理

1 傅立叶描绘子

图像的目标区域的边界是一条封闭的曲线，因此相对于边界上某一固定的起始点来说，沿边界曲线上的一个动点的坐标变化则是一个周期函数。通过规范化之后，这个周期函数可以展开为傅立叶级数，而傅里叶级数中的一系列系数是直接与边界曲线的形状有关的，可作为形状的描述，称为傅里叶描绘子。

傅立叶描述子，是物体形状边界曲线的傅立叶变换系数，是物体边界曲线信号的频域分析结果。它是一种描述不受起始点移动尺寸变化及旋转影响的曲线的方法。傅立叶描述子的基本思想，是把坐标的序列点看作复数：

$s(k) = s(k) + \mathrm{jy(k)}$ $s(k) = s(k) + \mathrm{jy(k)}$

即 x 轴作为实轴，y 轴作为虚轴，边界的性质不变。这种表示方法的优点，是将一个二维问题简化成一个一维问题。对 s(k)的傅立叶变换为：

$$a(u) = \sum_{k=0}^{N-1} s(k) e^{-j2\pi uk/N} \quad a(u) = \sum_{k=0}^{N-1} s(k) e^{-j2\pi uk/N}$$

傅立叶描述子序列 $\{a(u)\}$ 反映了原曲线的形状特征，同时，由于傅立叶变换具有能量集中性，因此，少量的傅立叶描述子就可以重构出原曲线。少量的傅立叶系数就可以很好地描述轮廓特征。由于傅立叶变换将序列的主要能量集中在了低频系数上，因此，傅立叶描述子的低频系数反映了轮廓曲线的整体形状，而轮廓的细节反映在了高频系数上。

2 感知机分类器

感知机一种人工神经网络。它可以被视为一种最简单形式的前馈式人工神经网络，是一种二元线性分类器。它能够根据每笔资料的特征，把资料判断为不同的类别。

迭代步骤如下：关于感知机的原理过程：click here!

输入：训练数据集 T={(x1, y1), (x2, y2),...,(xN,yN)}，xi 为实数向量，yi 为 1 或-1；
学习率 η（0< η <=1）。

输出：w，b，即感知机模型

（1）选取初值 w0，b0。

（2）在训练数据集中选取数据（xi, yi）；

（3）如果 $y_i(w \cdot x_i + b) \leq 0$，则

$$w \longleftarrow w + \eta y_i x_i$$

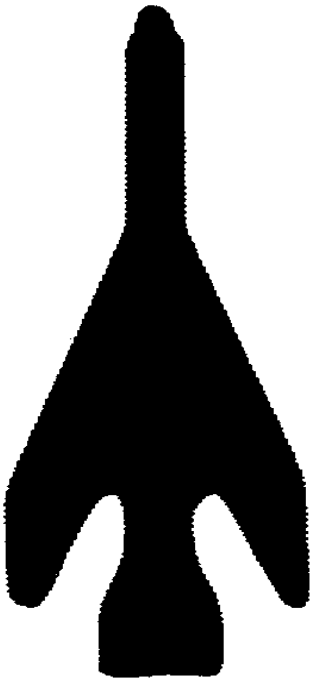
$$b \longleftarrow b + \eta y_i$$

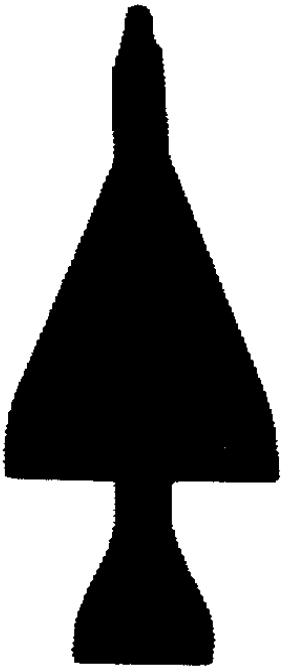
（4）转至（2），直至训练数据集中没有误分类点。

二、结果与讨论

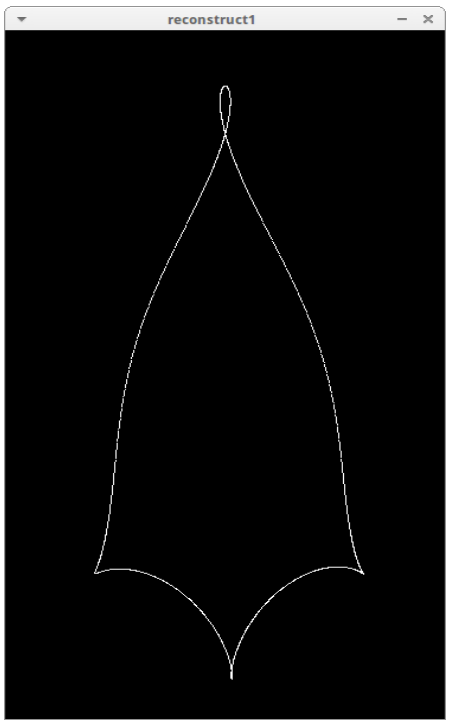
1.傅里叶描绘子：

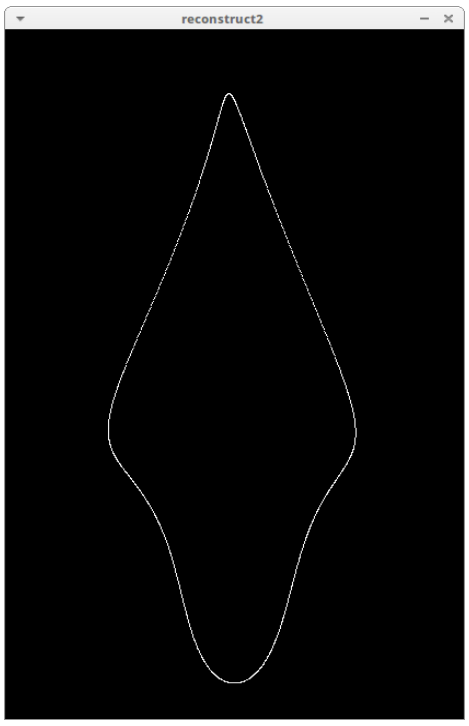
原图　　plane1　　plane2



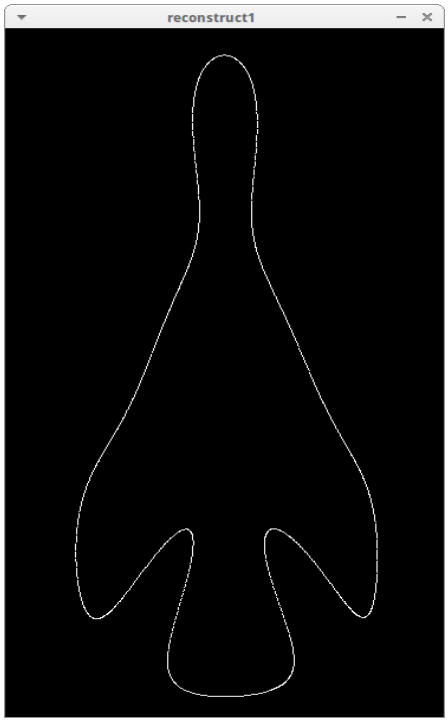


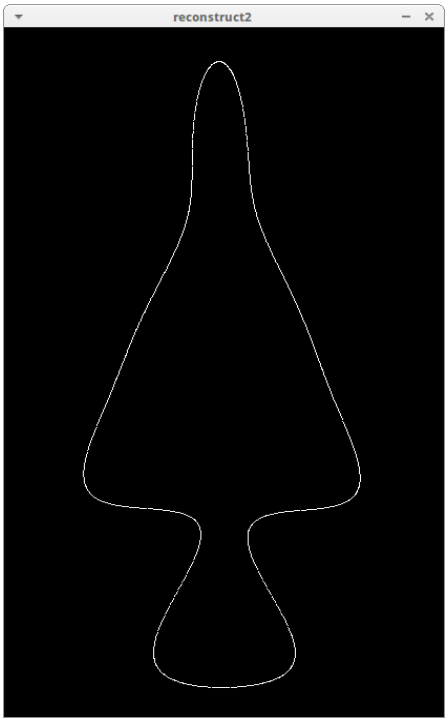
保留 10 个傅里叶描绘子：



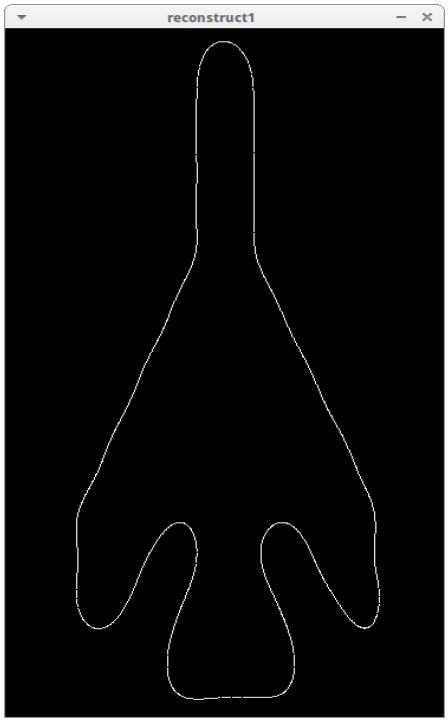


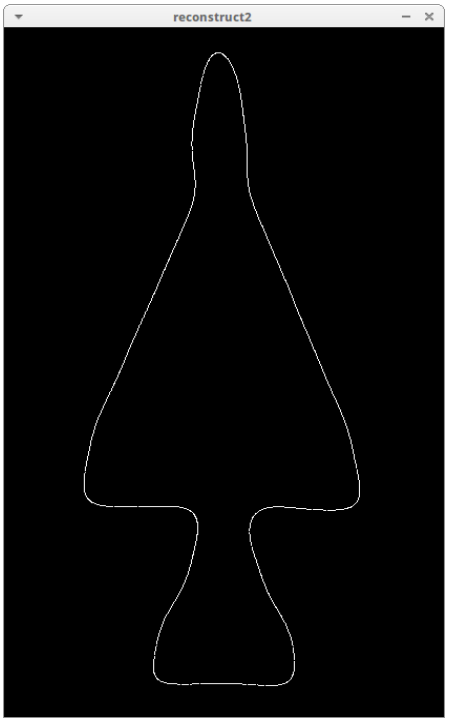
保留 18 个傅里叶描绘子：





保留 50 个傅里叶描绘子：





通过不断的尝试以及比较，我个人认为保证边界能够被辨识的最短描绘子的长度为 17。

Main.cpp

```
#include "MyFunction.h"
int main(int argc, char *argv[] )
{
   Mat src1 ,src2;
   if( argc !=3 || !(src1=imread(argv[1], 0)).data || !(src2=imread(argv[2],0)).data)
      return -1;

   Mat FDsor1 = Mat::zeros(Size (src1.cols, src1.rows),src1.type());
   Mat FDsor2 = Mat::zeros(src2.rows, src2.cols ,src2.type());

   Point b0(0,0);     int rev;
   cout<<"Now input reserve = ";
   cin >> rev ;

   complexVector boundary ,output1,output2;

   //图一的模式生成
   FrDstor::Findbegin(src1, b0);
   FrDstor::Iterator(src1 , boundary , b0);

   FrDstor::Fourier(boundary ,output1);
   complexVector FT1(output1);
   Mat descriptors1 =  Mat::zeros(1,rev,CV_32F);
   FrDstor::idFourier(FT1,descriptors1, rev);
   FrDstor::reconstruct(FT1 ,FDsor1);
   imshow ("reconstruct1",FDsor1);

   // 图二的模式生成
   FrDstor::Findbegin(src2, b0);
   FrDstor::Iterator(src2 , boundary , b0);
   FrDstor::Fourier(boundary ,output2);

   complexVector FT2(output2);
   Mat descriptors2 =  Mat::zeros(1,rev,CV_32F);
   FrDstor::idFourier(FT2, descriptors2, rev);
   FrDstor::reconstruct(FT2 ,FDsor2);
   imshow ("reconstruct2",FDsor2);

//感知机分类器
   Mat W= Mat::zeros(1,rev,CV_32F);
   for (int i(0); i<100;i++) {
      Classifier::Perceptron_Classifier(descriptors1, W ,1);
      Classifier::Perceptron_Classifier(descriptors2, W ,1); }
   cout <<"the w="<<endl<< W<< endl;
```

```
    waitKey(0);
    return 0;
}
```

MyFunction.cpp

```
#include "MyFunction.h"
typedef  unsigned int unit ;
void FrDstor::Findbegin(const Mat &src, Point &b0){

    for (int i(0); i<src.rows; ++i) {
        for (int j(0);j<src.cols;++j) {
            const  uchar* p = src.ptr<uchar>(i);
            if (p[j]==0) {
                b0.x=j;
                b0.y=i;
                return;
            }
        }
    }
}
// Find contours
void FrDstor::Iterator(const Mat &src ,complexVector &boundary, Point &b0) {
    boundary.clear();
    Point ck_1, next ,ck(b0) ,bk(b0);
    complex<float> acomplex(b0.x ,b0.y);
    boundary.push_back (acomplex);
    --ck.x;
    do
    {
        do
        {
            if (ck.y<bk.y&&ck.x==bk.x){
                next.y=ck.y;
                next.x=ck.x+1;  }
            else if (ck.y<bk.y&&ck.x>bk.x){
                next.y=ck.y+1;
                next.x=ck.x;
            }
            else if (ck.y==bk.y&&ck.x>bk.x){
                next.y=ck.y+1;
                next.x=ck.x ;
            }
            else if (ck.y>bk.y&&ck.x>bk.x){
                next.y=ck.y;
                next.x=ck.x-1;
            }
```

```
        else if (ck.y>bk.y&&ck.x==bk.x){
            next.y=ck.y;
            next.x=ck.x-1;
        }
        else if (ck.y>bk.y&&ck.x<bk.x){
            next.y=ck.y-1;
            next.x=ck.x;
        }
        else if (ck.y==bk.y&&ck.x<bk.x){
            next.y=ck.y-1;
            next.x=ck.x;
        }
        else if (ck.y<bk.y&&ck.x<bk.x){
            next.x=ck.x+1;
            next.y=ck.y;
        }
        if (src.at<uchar>(ck)==0){
            acomplex.real(ck.x);
            acomplex.imag(ck.y);
            boundary.push_back(acomplex);
            bk =ck;}
        else {  ck_1=ck;
            ck=next;}
      }
      while (bk!=ck);
      ck=ck_1;
   }
   while(b0!=bk);
   return ;
}
//DFT
void FrDstor::Fourier(const complexVector &boundary, complexVector &output){

   for (unit u(0); u<boundary.size(); ++u)
   {
      complex<float> au;
      for (uint k(0); k<boundary.size(); ++k) {
         {
            complex<float> powder(0,-M_PI*2*k*u/boundary.size());
            complex<float> ep = exp(powder);
            au +=boundary.at(k)*ep;
         }
      }
      output.push_back(au);
   }
```

```
}
//IDFT
void  FrDstor::idFourier(complexVector &DFT_Result ,Mat& descriptors, int P) {
   uint k=0; complexVector temp;
   Mat_<float>::iterator iter= descriptors.begin<float>();
   while(k<DFT_Result.size())
   {
      complex<float> sk;
      for (uint u(0); u<DFT_Result.size(); ++u) {
         if (u<P/2 || u>DFT_Result.size()-P/2-1)
         {
            float it= 0.0;
            magnitude(&DFT_Result.at(u).real(),&DFT_Result.at(u).imag(),&it,1);
            *iter=it;
            iter++;
            complex<float> powder(0,M_PI*2*k*u/DFT_Result.size());
            complex<float> ep =exp(powder);
            sk += DFT_Result.at(u)*ep;
         }
      }
      sk /= DFT_Result.size();
      temp.push_back(sk);
      ++k;
   }
   DFT_Result= temp;
   return ;
}
// 选取部分傅立叶描绘子重建边界
void FrDstor::reconstruct(const complexVector &iDFT_Result, Mat &dst){
   Point po;
   for (unit k(0); k<iDFT_Result.size(); ++k){
      po.x = floor(iDFT_Result.at(k).real());
      po.y = floor(iDFT_Result.at(k).imag());
      dst.at<uchar>(po) = 255;
   }
   return ;
}
//训练集生成
void Classifier::Perceptron_Classifier( Mat&descriptors, Mat& w, bool flag){
   RNG rng;  float a (0);
   Mat noise = Mat::zeros(descriptors.rows, descriptors.cols ,CV_32F);
   double sigma = 0.0;
   minMaxIdx(descriptors, 0, &sigma);   sigma /=10;
   for (int i(0); i<100; ++i) {
      rng.fill(noise,RNG::NORMAL, 0, sigma);
```

```
        add(noise, descriptors, noise);
        for (int j(0);j<noise.cols; ++j){
            a += noise.at<float>(0,j)*w.at<float>(0,j);
        }
        if (a <= 0)  {
            add(noise,w,w) ;}
    }
}
```