# 数字图像理论基础

周洋

上海大学机自学院无人艇工程研究院 & 人工智能研究院

版本：3.14

更新：2025 年 2 月 13 日

# 目录

# 1 数字图像基础代码

## 1.1 一维数据转二维可视化

**以手写数字识别数据为例：**如果想要可视化图片看看具体内容，可以采用 Python 的模块 matplotlib.pyplot 中的 imshow() 函数快速进行。需要注意可视化的时候，灰度图必须按照二维的格式进行输入。实例代码如下，WIN 10 平台，Python 3.7 下调试通过：

```
#1 加载包和数据集
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

train=pd.read_csv('train.csv',index_col=False)

x_train=train.iloc[0,1:].values  # 提取出数据，从第二列开始
x_train=x_train.astype(np.float)
x_label=train.iloc[0:,0].values  # 提取出标签

imgs_test = x_train.reshape(28, 28)  # 转换成二维矩阵

# 显示图片，squeeze()是把维度1的去掉（这里也可不加），gray灰度显示
plt.imshow(imgs_test.squeeze(), cmap='gray')
plt.title('%i' % x_label[0])  # 提取出对应的标签进行验证
plt.show()  # 显示
```

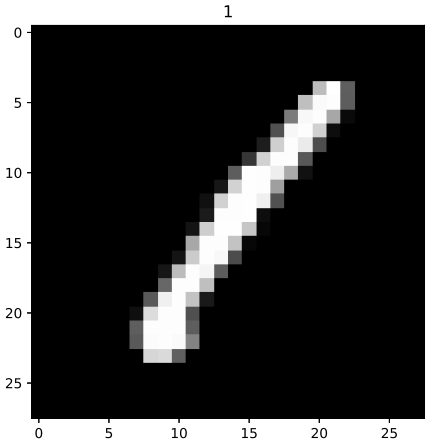



图 1: 手写数字可视化

## 1.2 PIL 库

首先是 PIL(python image library) 图像处理库。如果需要进行安装，虽然库的名字叫 PIL，但是要 pip install pillow。接下来简单了解下 PIL 库中 Image 类的作用，该类的调用可基本覆盖一般的图像处理内容。

假设打开测试集中的一张猫片如下：

图 2: 凶猛的噬元兽

通过 PIL 库中的 Image 类可以实现许多对图像的基本处理，实例代码如下，WIN 10 平台，Python 3.7 下调试通过：

```
# Pillow >= 1.0 no longer supports “import Image”. Please use “from PIL import Image” instead.
from PIL import Image
from PIL import ImageFilter
from PIL import ImageEnhance

img = Image.open("cat1.jpg")
print(img.format)  # JPEG
print(img.mode)  # RGB
print(img.size)  # (500, 374) 注意是(宽,高)，跟array的(行,列)相反

img_resize = img.resize((256,256))  # 调整尺寸

img_resize.save("catresize.jpg")
img_rotate = img.rotate(45)  # 旋转
img_rotate.save("catrotate.jpg")
om=img.convert('L')  # 灰度处理
om.save('catgray.jpg')
```

```
om = img.filter(ImageFilter.CONTOUR)  # 图片的轮廓
om.save('catcontour.jpg')
om = ImageEnhance.Contrast(img).enhance(20)  # 对比度为初始的10倍
om.save('catencontrast.jpg')

#更改图片格式:
import os

filelist =["cat1.jpg",
           "catcontour.jpg",
           "catencontrast.jpg",
           "catgray.jpg",
           "catresize.jpg",
           "catrotate.jpg",
           ]

for infile in filelist:
  outfile = os.path.splitext(infile)[0] + ".png"
  if infile != outfile:
    try:
      Image.open(infile).save(outfile)  # 不是png的话，转换成png
    except IOError:
      print ("cannot convert", infile)  #  否则说不能转换（如果文件找不到也一
          样）
```

运行后保存了 10 张图片，jpg 和 png 同样内容，分别 5 张。最终效果如下：


图 3: 被操纵的可怜噬元兽

如果想要读取一张图片里的内容，即将图片转换成矩阵，处理完毕后，再将矩阵数

据转换成图片，可以用如下操作，WIN 10 平台，Python 3.7 下调试通过：

```
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt

def ImageToMatrix(filename):
    # 读取图片
    im = Image.open(filename)
    # 显示图片
    # im.show()
    width,height = im.size
    im = im.convert("L")
    data = im.getdata()
    data = np.matrix(data,dtype='float')/255.0  # 也可用np.array。但是matrix不能是3维的，只能输出灰度图
    new_data = np.reshape(data,(height,width))  # 因为转成array后维度也变了
    return new_data

def MatrixToImage(data):
    data = data*255
    new_im = Image.fromarray(data.astype(np.uint8))
    return new_im

filename = 'cat1.jpg'
data = ImageToMatrix(filename)
print(data)

# plt.imshow(data,cmap=plt.cm.gray,interpolation='nearest')
# plt.show()  # 这两句可以把data转换成图片显示出来，等同于下面的new_im
new_im = MatrixToImage(data)
new_im.show()
new_im.save('cat1_1.bmp')
```

上面先对图片去除颜色，变成黑白的，再转换成二维数据矩阵进行操作。而一般的图片拥有 RGB 三通道，一个简单的测试方法如下，WIN 10 平台，Python 3.7 下调试通过：

```
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt

```

```
x = np.random.random((600,800,3))*255
# 注意需要使用2次random，同时注意random/rand/randn的区别
# 注意用双括号，否则出现错误random_sample() takes at most 1 positional argument(3 given)
print(x)

new_im = Image.fromarray(x.astype(np.uint8))
new_im.show()

# or
# plt.imshow(x)  # 该方法只能显示(H,W)-灰度图，或者(H,W,3)-彩色图
# plt.show()
```

显示出来为下图：

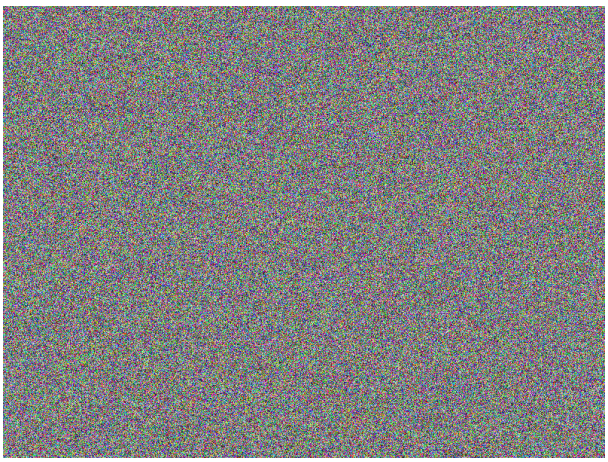


图 4: random 图像

如果想把图片转换成 array，等处理完毕后，再以图片的形式输出，可以采取以下代码。还是以上述猫片为例，WIN 10 平台，Python 3.7 下调试通过，最终会重新生成同一张猫片：

```
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt

im = Image.open('cat1.jpg')
width,height = im.size
data = im.getdata()
data = np.array(data,dtype='float')  # 不能用matrix，且这条不能省略。因为'ImagingCore' object has no attribute 'shape'/'reshape'
print(data.shape)  # (187000,3)  # 读取图片转换成array后变成2X2的矩阵了
```

```
data = data.reshape((height,width,3))  # 刚好reshape成原图片的array格式
# print(data)

# 注意上面的数据是float，转换成图片时要按照int输出
plt.imshow(data.astype(np.uint8))  # 该方法只能输出(H,W,3) or (H,W)，连(H,W,1)都不行
plt.show()
# or
# new_im = Image.fromarray(data.astype(np.uint8))
# new_im.show()
```

注意这里，利用 PIL 读取图片信息转换成 array 后，保存成了一个 (187000,3) 的矩阵，实际上就是把图片的 RGB 每个通道的平面数据信息以列向量的形式表达出来（一行一行读取）。而且 array 的显示是将最后一个维度的内容作为列来输出，所以将 (187000,3)reshape 成 (height,width,3) 时，最后一个维度的内容不会动，只会将 187000 个数据调整成 (height,width) 的矩阵（也是一行一行重组），所以重组后进行输出就刚好是原来的图像。一个简单验证实例如下，WIN 10 平台，Python 3.7 下调试通过：

```
import numpy as np

x = np.random.random((8,3))
print(x)
y = x.reshape(4,2,3)
print(y,y[0,1,2])
```

输出结果为：

```
[[0.26591368 0.40916516 0.89403719]
 [0.10377064 0.43995609 0.07389684]
 [0.6780993  0.53525621 0.87553874]
 [0.86170012 0.81410495 0.88396116]
 [0.76881033 0.05837419 0.13190795]
 [0.02823422 0.16532255 0.75404448]
 [0.11855252 0.10464019 0.92393298]
 [0.24752051 0.53557813 0.03448807]]
[[[0.26591368 0.40916516 0.89403719]
  [0.10377064 0.43995609 0.07389684]]

 [[0.6780993  0.53525621 0.87553874]
  [0.86170012 0.81410495 0.88396116]]
```

```
14
15  [[0.76881033  0.05837419  0.13190795]
16   [0.02823422  0.16532255  0.75404448]]
17
18  [[0.11855252  0.10464019  0.92393298]
19   [0.24752051  0.53557813  0.03448807]]] 0.07389684436676058
```

上述代码可以清楚地看到 array 在 rshape 后对应的元素都在什么位置。

## 1.3 opencv 库

其次介绍下 opencv 库。OpenCV 是个很强大的图像处理库，性能也很好，在 Python 里有直接可供使用的调用库，直接 pip install opencv 或者去 PyPI 下载 whl 文件即可。其功能与 PIL 有点类似，但更为强大。一般来说，用 opencv 读取图片需要使用到的函数是 cv2.imread()，读取完毕后将直接转换成 numpy.ndarray 的格式，可以类似 PIL 一样进行各种处理。另外，opencv 库还可以处理视频，可以创建一个 VideoCapture 对象，读取视频文件，获取相机/视频的各种属性，以及视频保存等。

下面这个例子会载入一张图片（还是使用上面那张猫片），打印出图像大小，并对图像进行转换保存为.png 格式，同时 reshape 该图像并显示出来。WIN 10 平台，Python 3.7 下调试通过：

```
1  import cv2
2
3  # 读取图像
4  im = cv2.imread('cat1.jpg')
5  h,w =im.shape[:2]
6  print(h,w) # 374 500
7
8  # 保存图像
9  cv2.imwrite('result.png',im)
10
11 # reshape图像
12 reim = cv2.resize(im,(374,500))
13 cv2.imshow('resize',reim)
14 # cv2.imwrite('dealcat2.jpg',reim)
```

函数 imread() 返回图像为一个标准的 Numpy 数组，其 shape 的组成是：**（高，宽，通道数）**，对应 Numpy 矩阵的行和列。所以当查看 0:2 之间的 shape 时（即上面的代码

[:2])，会返回图像的高和宽。但是当 reshape 时却不一样，原因在于对于数据处理常用矩阵的“**行和列**”，但图像处理却一般使用“**长和宽**”，所以上面 reshape 的代码中的组成为：（**宽，高**），与 shape 的前两个参数值相反，需注意。上述代码最后会显示经过默认的**双线性插值法**计算得到的 reshape 猫片：

**图 5:** 被 reshape 的可怜噬元兽

另外，从代码中可以看到，在对图像进行处理时，括号中是先指明对象；而在 imread()、imwrite() 和 imshow() 里是首先指明文件名，再指明对象，这里也需要注意。

在 OpenCV 中，图像不是按传统的 RGB 颜色通道，而是按 BGR 顺序（即 RGB 的倒序）储存的。所以这时候需要转换函数 cvtColor() 来实现颜色空间的转换。例如，可以通过下面的方式将原图像转换成灰度图像，WIN 10 平台，Python 3.7 下调试通过：

```
1 import cv2
2
3 # 读取图像
4 im = cv2.imread('cat1.jpg')
5 gray = cv2.cvtColor(im,cv2.COLOR_BGR2GRAY)
6
7 # 显示图像
8 cv2.imshow('graycat1',gray)
9 cv2.waitKey()
```

代码中 cv2.waitKey(delay)，delay = NONE & 0 表示一直显示，除此之外表示延时切换到下一帧图像的毫秒数。

转换灰度图时主要是通过将 RGB 按比例计算得到灰度数值。另外一些颜色转换的参数包括：

- cv2.COLOR_BGR2RGB

- cv2.COLOR_GRAY2BGR

在通过灰度图回转为彩色图时必须知晓之前转换灰度图时的比例。当 RGB 相等时，图像会显示为灰值，不相等时颜色会偏向于较大的基色。

接下来探讨视频处理。单纯使用 Python 来处理视频有些困难，因为需要考虑速度、编解码器、摄像机、操作系统和文件格式等，所以一般视频处理还是采用 OpenCV 的 Python 接口。

OpenCV 可以很好地支持从摄像头或文件捕获视频，下面给出一个捕获视频帧并在 OpenCV 窗口中显示这些视频帧的例子，WIN 10 平台，Python 3.7 下调试通过：

```
import cv2

# 设置视频捕获
# cv2.VideoCapture(0)即从摄像头捕获
cap = cv2.VideoCapture('Forza.mp4')

while True:
    ret,frame = cap.read()
    cv2.imshow('ideo test',frame)
    key = cv2.waitKey(10)
    if key == 27:
        break
    if key == ord(' '):
        cv2.imwrite('vid_result.jpg',frame)
```

捕获对象 VideoCapture 从摄像头或文件捕获。read() 方法解码并返回下一视频帧，第一个变量 ret 是一个判断视频帧是否成功读入的标志，第二个变量则是实际读入的图像数组。函数 waitkey() 等待用户按键：如果按下的是 Esc 键（ASCII 码是 27），则退出应用；如果按下的是空格键（ord() 函数返回'' 内对应的 ASCII 码），就保存该视频帧。如果什么都不做，frame 会一直以 **waitkey() 中的数值作为间隔时间进行顺序切换**，直到结束。

另外也可以使用 if cv2.waitKey(1) & 0xFF == ord('q'): break 来实现 1ms 之内的正常退出。其中，0xFF 是一个位掩码，十六进制常数，二进制值为 11111111，它将左边的 24 位设置为 0，只保留原始的最后 8 位，把返回值限制在在 0 和 255 之间。此处是防止 BUG。

**注意！** OpenCV 处理视频时，假设视频可以正常播放，但是等视频播放完却产生报错：error: (-215:Assertion failed) size.width>0 && size.height>0 in function 'cv::imshow'。而如果在中途中断了视频播放，比如上面代码中按下 Esc 键，则不会报错（比如上面的代

码）。这种错误主要是视频末尾读取帧不正确导致的（因为循环在一直进行没有 break）。

于是在显示图像前面加了一个判断，变为：

```
if ret == True:
    cv.imshow('frame',frame)
else:
    break
```

则不会报错了。

所以上面显示视频的代码可以进一步完善修改为下面形式，WIN 10 平台，Python 3.7 下调试通过：

```
import cv2

# 设置视频捕获
# cv2.VideoCapture(0)即从摄像头捕获
cap = cv2.VideoCapture('Forza.mp4')

while True:
    ret,frame = cap.read()
    if ret == True:
        cv2.imshow('ideo test',frame)
    else:
        break  # 视频读取完退出循环
    if cv2.waitKey(1) == 27:
        break
    if cv2.waitKey(1) == ord(' '):
        cv2.imwrite('vid_result.jpg',frame)

cap.release()  # 释放视频数据
cv2.destroyAllWindows()  # 关闭视频窗口
```

如果想要保存视频也很简单。保存视频的接口为：

<VideoWriter object> = cv.VideoWriter(filename, fourcc, fps, frameSize[, isColor])

参数说明：

- filename：要保存的视频名称和路径（例如 data/outVideo.mp4）
- fourcc: 视频编码器
- fps: 帧率

- framesize: 帧数大小
- sColor:True 彩色，False 灰度视频，默认 True

实例如下，WIN 10 平台，Python 3.7 下调试通过：

```
import cv2

cap = cv2.VideoCapture('Forza.mp4')  # 要读取的视频  0、1 本地相机或外接相机

# 创建VideoWriter类对象
fourcc = cv2.VideoWriter_fourcc(*'XVID')
fps = cap.get(cv2.CAP_PROP_FPS)
size = (int(cap.get(cv2.CAP_PROP_FRAME_WIDTH)), int(cap.get(cv2.
    CAP_PROP_FRAME_HEIGHT)))
out = cv2.VideoWriter('outVideo.mp4', fourcc, fps, size)

# 读取视频流
while cap.isOpened():
    ret,frame = cap.read()  # 获取一帧图像
    if ret:
        frame = cv2.flip(frame, 1)  # 测试调整方向
        out.write(frame)  # 写入视频对象
        cv2.imshow('frame',frame)  # 显示读取视频
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break  # q键关闭
    else:
        break

# 关闭流
cap.release()
out.release()
cv2.destroyAllWindows()
```

除此之外，使用 OpenCV 可以从一个文件读取视频帧，并将其转换成 Numpy 数组。实例如下，WIN 10 平台，Python 3.7 下调试通过：

```
from numpy import *
import cv2

# 设置视频捕获
# cv2.VideoCapture(0)即从摄像头捕获
```

```
cap = cv2.VideoCapture('Forza.mp4')

frames = []  # 创建list
# 获取帧，存储到数组中
while True :
    ret,frame = cap.read()
    if ret:
        cv2.imshow('video',frame)
        frames.append(frame)  # 只有list可以被append
        cache = frame  # 缓存当前帧，防止被清空
        if cv2.waitKey(1) == 27:
            break
    else:
        break
frames = array(frames)  # 把list转换成array

# 检查尺寸
print(cache.shape)  # (1080, 1920, 3)
print(frames.shape)  # (903, 1080, 1920, 3)
```

上述代码将每一帧数组添加到列表末，直到捕获结束。最终得到的数组会有帧数、帧高、帧宽及颜色通道数（3 个）。这里共记录了 903 帧。

上述代码有一些值得注意。首先在使用 append() 函数时要注意只有 list 可以被 append（传递参数可以是 array），所以如果 frames = array(frames) 这条代码写在 while 循环里了就会报错，因为第二次循环 frames 就转换成了数组不能被 append 了；在检查尺寸的时候不能直接调用 frame，因为跳出循环后 frame 被置空了会提示'NoneType'（但是如果在视频播放时按下 Esc 键却可以显示，因为当前帧被保存下来了），所以这时候需要一个缓存变量来储存当前帧，这样哪怕在循环结束后，最后一帧信息也会被保存在 cache 变量中，以供调用。

接下来，介绍两类利用 OpenCV 对图像进行处理的例子以供参考。

### 1.3.1 简单边缘检测

边缘检测通常用于理解图像中的对象，帮助机器做出更好的预测。编写边缘检测程序是了解机器如何看待外界的好方法。接下来使用 python 进行边缘检测，主要使用 Numpy，Matplotlib 和 OpenCV 库。

实现过程非常简单，主要包括三个部分：边缘检测，可视化，最后保存结果。WIN 10

平台，Python 3.7 下调试通过：

```
import numpy as np
import cv2
import matplotlib.pyplot as plt

def simple_edge_detection(image):
    edgesdetected = cv2.Canny(image, 100, 200)
    images = [image, edgesdetected]

    location = [121, 122]  # give index to subplot - 1*2 form

    for loc, edge_image in zip(location, images):
        plt.subplot(loc)
        plt.imshow(edge_image, cmap = 'gray')

    ''' or
    for i in range(len(location)):
        plt.subplot(location[i])
        plt.imshow(images[i], cmap = 'gray')
    '''

    cv2.imwrite('edge_detected.png', edgesdetected)
    plt.savefig('edge_plot.png')  # can save current fig,  i.e. both subfigs
    plt.show()

img = cv2.imread('0001.png', 0)  # import a figure
simple_edge_detection(img)  # call the function
```

简单解释：

1. Canny 是调用的使用 OpenCV 进行边缘检测的方法；
2. image 是函数的参数，这意味着将在调用函数时传递图像。这样，可以轻松地用不同的图像测试程序；
3. 100 和 200 是磁滞阈值 (确定哪些边缘全部是真正的边缘，哪些不是) 的最小值和最大值；
4. 绘图部分需要位置数组；
5. 然后，可以用循环同时可视化原始图像和边缘检测图像 (利用 zip 或直接用 index)；
6. cmap 参数用于更改图像的颜色。这里将它们转换为灰色。

最后是常规的保存和显示图片。显示结果如下：

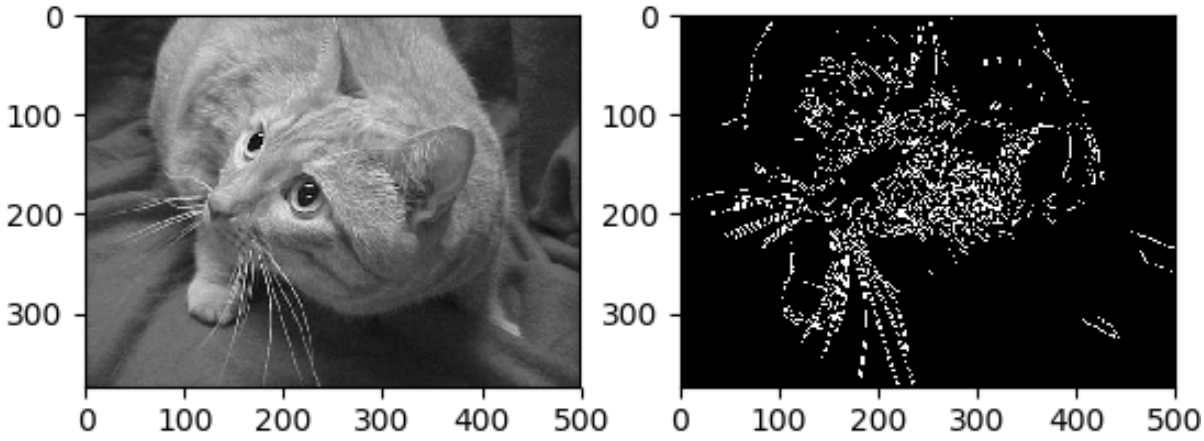



图 6: 被边缘检测的可怜噬元兽

### 1.3.2 图像阴影去除

图像阴影去除，可以通过 Numpy 和 OpenCV 基本函数来实现。

删除阴影时，有两件事要注意。假设图像是灰度图像，如果图像背景较浅且对象较暗，则必须先执行最大滤波，然后再执行最小滤波，最后要通过两者相减来中和最大最小滤波所带来的负面效果（这注意理解）。如果图像背景较暗且物体较亮，反过来先执行最小滤波再进行最大滤波即可。

**最大滤波**：假设有一定大小的图像 $I$。编写的算法应该逐个遍历 $I$ 的像素，并且对于每个像素 $(x, y)$，它必须找到该像素周围的邻域（大小为 $N \times N$ 的窗口）中的最大灰度值，并进行写入 $A$ 相应像素位置 $(x, y)$ 的最大灰度值。所得图像 $A$ 称为输入图像 $I$ 的最大滤波图像。

- max_filtering() 函数接受输入图像和窗口大小 $N$；
- 它最初在输入数组周围创建一个“墙”（带有-1 的填充），当遍历边缘像素时会有所帮助；
- 然后，创建一个“temp”变量，将计算出的最大值复制到其中；
- 然后，遍历该数组并围绕大小为 $N \times N$ 的当前像素创建一个窗口；
- 然后，使用“amax()”函数在该窗口中计算最大值，并将该值写入 temp 数组；
- 将该临时数组复制到主数组 $A$ 中，并将其作为输出返回；
- $A$ 是输入 $I$ 的最大滤波图像。

**最小滤波**：此算法与最大滤波完全相反，即在该像素周围的 $N \times N$ 邻域中找到了最小值，并将该最小灰度值写入 $B$ 中的 $(x, y)$。所得图像 $B$ 称为图像 $I$ 的经过最小滤波的图像。

另外，执行最小-最大滤波后，获得的值不在 $0-255$ 的范围内。因此，必须归一化使用背景减法获得的最终阵列。

注 对于归一化，这里需要做进一步说明。大部分资料和大部分人的认知都容易混淆一些概念，这将会导致算法的完全不同。下面列出概念，分别表示四种 Feature scaling(特征缩放) 方法：

**定义 1.1 (Feature scaling(特征缩放))** 几类特征缩放的定义如下：

- Rescaling

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)} \tag{1}$$

- Mean normalization

$$x' = \frac{x - \text{mean}(x)}{\max(x) - \min(x)} \tag{2}$$

- Standardization

$$x' = \frac{x - \bar{x}}{\sigma} \tag{3}$$

- Scaling to unit length

$$x' = \frac{x}{\|x\|} \tag{4}$$

前面说的归一化，采用的为第一种算法，但又不完全是。因为需要把一组数据重新平移缩放到一个固定的区间内。上面定义中的归一化特指归到 $(0,1)$ 区间内。如果要转换到一个特定的区间内，假设令该区间的下界与上界分别为 inf 和 sup，则改进后的归一化公式为：

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}(\text{sup} - \text{inf}) + \text{inf} \tag{5}$$

有一些情况会用到第二种特征缩放。比如本文档在讲神经网络初探的例子里，通过平均归一化把身高体重转化到一个相对平均的范围内，唯一的区别在于例子里并未除以最大最小的区间范围（将会只在 $(0,1)$ 区间内）。

第三种特征缩放即所谓的“标准化”。而第四种在矢量分析里会经常遇到，也就是所谓的单位化。

值得注意的是还有一种类似的特征缩放，是把每个数值除以所有数值的总和来得到。这样最终所有数值的结果之和等于 1，而也实现了不影响数据权重的特征缩放。比如高斯滤波的卷积核，需要所有数值之和等于 1，这时就会采用这类特征缩放。

变量 $N$（用于过滤的窗口大小）将根据图像中粒子或内容的大小进行更改。对于测试图像，这里选择大小 $N = 20$。

具体测试代码如下，WIN 10 平台，Python 3.7 下调试通过：

```python
import cv2
import numpy as np
import matplotlib.pyplot as plt

def max_filtering(N, I_temp):
    wall = np.full((I_temp.shape[0]+(N//2)*2, I_temp.shape[1]+(N//2)*2), -1)
    wall[(N//2):wall.shape[0]-(N//2), (N//2):wall.shape[1]-(N//2)] = I_temp.copy()
    temp = np.full((I_temp.shape[0]+(N//2)*2, I_temp.shape[1]+(N//2)*2), -1)
    for y in range(0,wall.shape[0]):
        for x in range(0,wall.shape[1]):
            if wall[y,x]!=-1:
                window = wall[y-(N//2):y+(N//2)+1,x-(N//2):x+(N//2)+1]
                num = np.amax(window)
                temp[y,x] = num
    A = temp[(N//2):wall.shape[0]-(N//2), (N//2):wall.shape[1]-(N//2)].copy()
    return A

def min_filtering(N, A):
    wall_min = np.full((A.shape[0]+(N//2)*2, A.shape[1]+(N//2)*2), 300)
    wall_min[(N//2):wall_min.shape[0]-(N//2), (N//2):wall_min.shape[1]-(N//2)] = A.copy()
    temp_min = np.full((A.shape[0]+(N//2)*2, A.shape[1]+(N//2)*2), 300)
    for y in range(0,wall_min.shape[0]):
        for x in range(0,wall_min.shape[1]):
            if wall_min[y,x]!=300:
                window_min = wall_min[y-(N//2):y+(N//2)+1,x-(N//2):x+(N//2)+1]
                num_min = np.amin(window_min)
                temp_min[y,x] = num_min
    B = temp_min[(N//2):wall_min.shape[0]-(N//2), (N//2):wall_min.shape[1]-(N//2)].copy()
    return B

def background_subtraction(I, B):
    O = I - B
    norm_img = cv2.normalize(O, None, 0,255, norm_type=cv2.NORM_MINMAX)
    return norm_img

def min_max_filtering(M, N, I):
```

```
    if M == 0:  # first max_filtering, then min_filtering
        A = max_filtering(N, I)
        B = min_filtering(N, A)
        normalised_img = background_subtraction(I, B)
    elif M == 1:  # first min_filtering, then max_filtering
        A = min_filtering(N, I)
        B = max_filtering(N, A)
        normalised_img = background_subtraction(I, B)
    return normalised_img

P = cv2.imread('cat1.jpg',0)
plt.subplot(121)
plt.imshow(P,cmap='gray')
plt.title("original image")

# can edit the N and M values here for P and C images
O_P = min_max_filtering(M = 0, N = 20, I = P)

#Display final output
plt.subplot(122)
plt.imshow(O_P, cmap = 'gray')
plt.title("Final output")

plt.show()
```

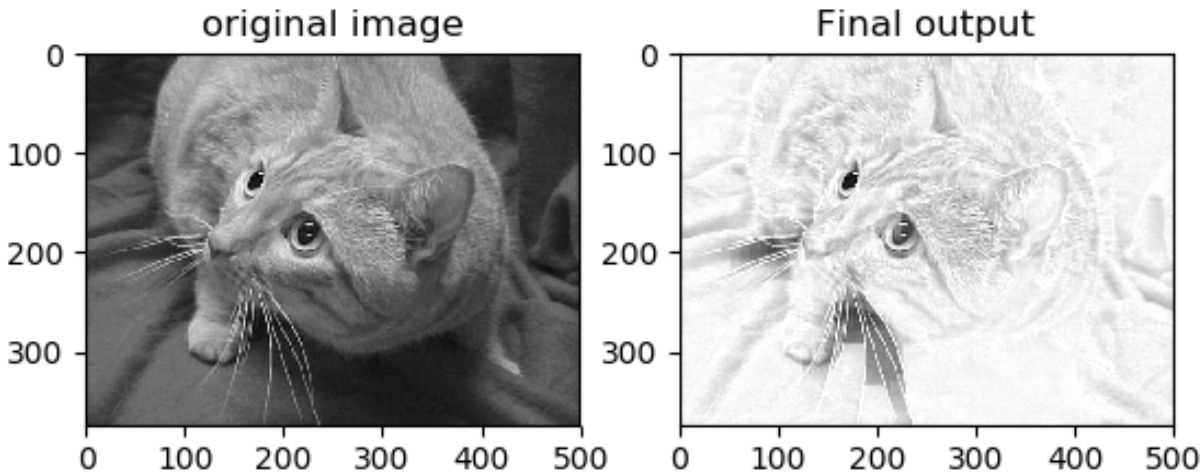



图 7: 被去除阴影的可怜噬元兽

输出图像相较于原始图像的阴影已经去除。

代码里一些简要说明：

1. 最大最小滤波里用到了整除符号//，是因为考虑到填充的区域与框选大小的奇偶性有关；
2. copy() 函数就是一种简单的复制；
3. cv2.normalize 函数中 cv2.NORM_MINMAX 的参数就代表采用的是上述第一种特征缩放方法的变体；
4. 最后这里的 subplot 是最简单的一种罗列方法，如果想要使用前面的 for 循环也可。

## 1.4 检测摄像头或者视频

检测摄像头或者视频的代码如下：

```
#-------------------------------------#
#       调用摄像头检测
#-------------------------------------#
from ssd import SSD  # 把SSD算法的SSD类调进来，也可以是别的算法
from PIL import Image
import numpy as np
import cv2

ssd = SSD()  # 把SSD类实体化

# 调用摄像头
capture=cv2.VideoCapture(0)  # 调用摄像头
# capture=cv2.VideoCapture("C:/1.mp4")  # 读取视频文件，注意是斜杠不是反斜杠

while(True):
    # 读取某一帧
    ref,frame=capture.read()
    if ref == True:
        # 格式转变，BGRtoRGB
        frame = cv2.cvtColor(frame,cv2.COLOR_BGR2RGB)
        # 转变成Image
        frame = Image.fromarray(np.uint8(frame))

        # 进行检测
        frame = np.array(ssd.detect_image(frame))
        # RGBtoBGR满足opencv显示格式
        frame = cv2.cvtColor(frame,cv2.COLOR_RGB2BGR)
        cv2.imshow("video",frame)
```

```
        c= cv2.waitKey(30) & 0xff
        if c==27:  # 该段语句只要在while循环语句即可
            capture.release()  # 关闭视频文件或相机设备
            break
    else:
        break
```

## 1.5 获取视频检测的 FPS

FPS 简单来理解就是图像的刷新频率，也就是每秒多少帧。假设目标检测网络处理 1 帧要 0.02s，此时 FPS 就是 50。

下面是一个简单的实现把 FPS 显示在视频上的代码：

```
# ...
import time  # 用于计时
# ...

# 调用摄像头
capture=cv2.VideoCapture(0)
fps = 0.0  # 设计一个计数器起点
while(True):
    t1 = time.time()  # 记录当前时间（处理前的时间）
    # ...
    # 处理某一帧
    # ...
    fps  = ( fps + (1./(time.time()-t1)) ) / 2  # 把处理一帧前后计算得来的FPS
        求个平均
    print("fps=␣%.2f"%(fps))  # 在terminal中打印fps数值
    frame = cv2.putText(frame, "fps=␣%.2f"%(fps), (0, 40), cv2.
        FONT_HERSHEY_SIMPLEX, 1, (0, 255, 0), 2)  # 把fps写在处理完一帧后的
        frame里

    cv2.imshow("video",frame)  # 把frame显示出来
    # ...
```

# 2 数字图像数学基础

## 2.1 图像微分算子

### 2.1.1 方向导数

**定义 2.1 (方向导数)** 若函数 $f=f(x,y,z)$ 在点 $M_0(x_0,y_0,z_0)$ 处可微，$\cos\alpha,\cos\beta,\cos\gamma$ 为任意单位方向 $\boldsymbol{e}$ 相对于 Cartesian 坐标系的方向余弦，即 $\boldsymbol{e}=(\cos\alpha,\cos\beta,\cos\gamma)$，则函数 $f$ 在点 $M_0$ 处沿 $\boldsymbol{e}$ 方向的方向导数存在，且有

$$\left.\frac{\partial f}{\partial e}\right|_{M_0}=\frac{\partial f}{\partial x}\cos\alpha+\frac{\partial f}{\partial y}\cos\beta+\frac{\partial f}{\partial z}\cos\gamma \tag{6}$$

注意，$\boldsymbol{e}$ 应为向量，但在方向导数的定义中使用的是标量，所以这里公式中仅用非黑体的 $e$ 表示其模，即 $\|e\|$。

**证明.** 由假设 $f(x,y,z)$ 在点 $M_0(x_0,y_0,z_0)$ 可微分，故根据全微分的定义有

$$\begin{aligned}f(M_0+\Delta M)-f(M_0)=&\frac{\partial f}{\partial x}(M_0)\Delta x+\frac{\partial f}{\partial y}(M_0)\Delta y+\frac{\partial f}{\partial z}(M_0)\Delta z\\&+o\left(\sqrt{(\Delta x)^2+(\Delta y)^2+(\Delta z)^2}\right)\end{aligned} \tag{7}$$

其中 $o\left(\sqrt{(\Delta x)^2+(\Delta y)^2+(\Delta z)^2}\right)$ 的意义为 $\sqrt{(\Delta x)^2+(\Delta y)^2+(\Delta z)^2}$ 的高阶小量。

由于

$$\begin{aligned}&\sqrt{(\Delta x)^2+(\Delta y)^2+(\Delta z)^2}=\|e\|\\&\Delta x=\|e\|\cdot\cos\alpha,\ \Delta y=\|e\|\cdot\cos\beta,\ \Delta z=\|e\|\cdot\cos\gamma\end{aligned} \tag{8}$$

代入上式，两边除以 $\|e\|$，再令 $\|e\|\to 0$ 取极限即可证明公式 (6)。

### 2.1.2 梯度

根据方向导数的定义，可以演化出梯度的概念。引入**哈密顿（Hamilton）算子** $\nabla$（念作 Nabla，所以也称为 Nabla 算子），其在 Cartesian 坐标系下具有如下形式：

$$\nabla=\frac{\partial}{\partial x}\boldsymbol{i}+\frac{\partial}{\partial y}\boldsymbol{j}+\frac{\partial}{\partial z}\boldsymbol{k} \tag{9}$$

即运算符 $\nabla$ 兼具向量和微分运算两者的功能。

考虑到

$$\nabla f(M_0) = \frac{\partial f}{\partial x}(M_0)\boldsymbol{i} + \frac{\partial f}{\partial y}(M_0)\boldsymbol{j} + \frac{\partial f}{\partial z}(M_0)\boldsymbol{k} \tag{10}$$

则方向导数可以表示为两个向量的内积：

$$\left.\frac{\partial f}{\partial e}\right|_{M_0} = \nabla f(M_0)\cdot \boldsymbol{e} \tag{11}$$

其中，$\nabla f(M_0)$ 称为 $f$ 在点 $M_0$ 处的梯度，也可记为 $\mathrm{grad} f(M_0)$。可以看到，梯度为**向量**。即，梯度的作用是**把标量场变为向量，描述方向**。

根据方向导数的内积形式(11)可知，只有当单位方向 $\boldsymbol{e}$ 指向梯度向量 $\nabla f(M_0)$ 的方向时，方向导数 $\left.\frac{\partial f}{\partial e}\right|_{M_0}$ 取最大值 $\|\nabla f(M_0)\|$。即梯度具有如下性质：

注 梯度的方向是使得方向导数取得最大值的方向（指向导数增加最大的方向），梯度的模就是方向导数的最大值；同时梯度方向也正好沿着 $f$ 的等值面（使得函数值相同的坐标点的集合）在这点的法线方向，即梯度向量与等值面垂直。

### 2.1.3 图像梯度原理

梯度在图像处理中有明确的数学含义，主要用来刻画图像灰度的变化率。对于图像的数学模型来说，即一个二维函数 $f(x,y)$，是按照像素来离散的，其微分表达式为：

$$\begin{aligned}\frac{\partial f(x,y)}{\partial x} &= f(x+1,y) - f(x,y) = gx \\ \frac{\partial f(x,y)}{\partial y} &= f(x,y+1) - f(x,y) = gy\end{aligned} \tag{12}$$

这分别是图像在 $(x,y)$ 点处 $x$ 方向和 $y$ 方向上的梯度，从该表达式可以看出，图像的梯度相当于 2 个相邻像素之间的差值。

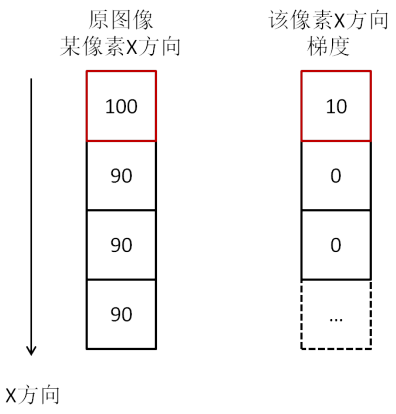



图 8: 图像梯度示意图 1

图像梯度可以用来增强图像的清晰度。先考虑 $x$ 方向，选取某个像素，假设其像素值是 100，沿 $x$ 方向的相邻像素分别是 90，90，90，则可以计算其 $x$ 方向的梯度分别为 10，0，0。这里只取变化率的绝对值，表明变化的大小即可，如上图8所示。

可以看到，100 和 90 之间亮度相差 10，并不是很明显，与一群 90 的连续灰度值在一起，轮廓必然是模糊的。但如果相邻像素灰度值有变化，梯度就有值，相邻像素灰度值没有变化，梯度就为 0。这时候如果把梯度值与对应的像素相加，那么灰度值没有变化的，像素值不变，而有梯度值的，灰度值将会增加，如图9所示。

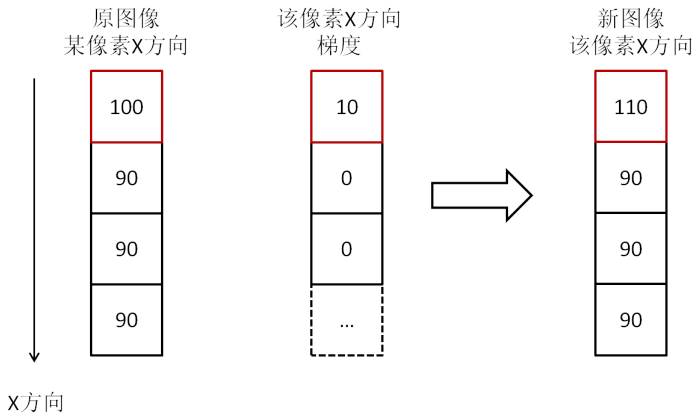



**图 9:** 图像梯度示意图 2

可以看到，相加后的新图像，原图像像素点 100 与 90 亮度只相差 10，现在是 110 与 90，亮度相差 20，对比度进行了增强。反映在图像上，即会带来物体的轮廓边缘与背景增强区别的效应，这就是用梯度来增强图像的原理。

上面只说了 $x$ 方向，$y$ 方向是一样的。更进一步，可以将 $x$ 方向和 $y$ 方向的梯度结合起来，用 $L1$ 或 $L2$ 范数来刻画。出于对计算量的考虑，一般来说会采用绝对值来进行计算：

$$M(x,y) = |gx| + |gy| \tag{13}$$

接下来，用一段 Python 代码来实现上述图像增强的方法，依旧还是采用之前的猫片，WIN 10 平台，Python 3.7 下调试通过：

```
1 import cv2
2 import numpy as np
3
4 cat = cv2.imread("cat1.jpg", 0)
5 row, column = cat.shape
6 cat_f = np.copy(cat)
7 cat_f = cat_f.astype("float")  # 图像像素值为ubyte类型，需要转换
8
```

```
gradient = np.zeros((row, column))

for x in range(row - 1):
    for y in range(column - 1):
        gx = abs(cat_f[x + 1, y] - cat_f[x, y])
        gy = abs(cat_f[x, y + 1] - cat_f[x, y])
        gradient[x, y] = gx + gy

sharp = cat_f + gradient
sharp = np.where(sharp < 0, 0, np.where(sharp > 255, 255, sharp))

gradient = gradient.astype("uint8")  # 需要再转回ubyte类型
sharp = sharp.astype("uint8")
cv2.imshow("cat", cat)  # or use cv2.imwrite to save the figures
cv2.imshow("gradient", gradient)
cv2.imshow("sharp", sharp)
cv2.waitKey()
```

关于代码中的解释：

- opencv 中 imshow 内部的参数类型可以分为两种：当输入矩阵是 uint8 类型的时候，此时 imshow 显示图像的时候，会认为输入矩阵的范围在 0-255 之间；如果 imshow 的参数是 double 类型的时候，那么 imshow 会认为输入矩阵的范围在 0-1；
- 用 python 处理图像，涉及两幅图像像素值之间的加减运算时，需要注意图像像素值是 ubyte 类型，数据范围为 0 255，若做运算出现负值或超出 255，则会抛出异常RuntimeWarning: overflow encountered in ubyte_scalars，所以需要转换为 int 或 float，处理完数据后还需再转换回 ubyte 类型，才能正常显示（不然根据上一条，显示图像会呈现过曝的白色）；
- 在 NumPy 中，where() 函数可看作判断表达式的数组版本：x = where(condition,y,z)，其中 condition、y 和 z 都是数组，它的返回值是一个形状与 condition 相同的数组。当 condition 中的某个元素为 True 时，x 中对应下标的值从数组 y 获取，否则从数组 z 获取。由于运算是在 C 语言级别完成的，所以计算效率比较高。这里使用 where() 函数目的在于把数据都转换到 (0,255) 区间内，不使用“归一化”公式的原因在于不希望改变已经在 (0,255) 区间内的数值。

代码运行的结果如下图10，可以看到较为明显的锐化效果：

(a) cat (b) gradient (c) sharp

图 10: 用梯度做图像增强

### 2.1.4 图像梯度算子

前面介绍的图像梯度，其定义是根据微积分原理在二维离散函数中推导出来的。但是梯度的核心目的是得到像素点与其相邻像素的灰度值变化情况，并通过这种变化来增强图像。再回顾一下，假设某像素与其 8 领域用如下矩阵表示：

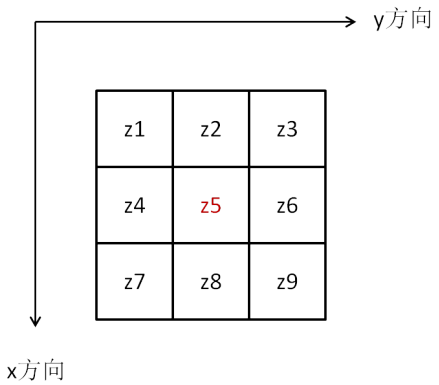



图 11: 图像梯度示意图 3

那么，根据图像梯度的定义：

$$\begin{aligned} gx &= z8 - z5 \\ gy &= z6 - z5 \end{aligned} \tag{14}$$

这种原始定义的梯度只是灰度值变化情况的度量工具，而这种度量工具显然有多种，这些计算方法统称为**梯度算子**。根据不同的相邻像素差值计算得到的效果可能有所不同，但基本原理是一致的。

例如罗伯特（Roberts）交叉梯度算子，定义的是：

$$\begin{aligned} gx &= z9 - z5 \\ gy &= z8 - z6 \end{aligned} \tag{15}$$

命名为交叉算子的原因如下图12所示：

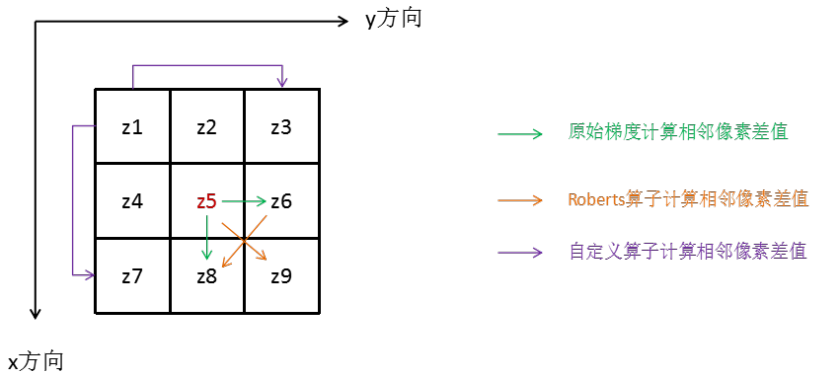



图 12: 图像梯度示意图 4

可以看到，不管是原始梯度也好，Roberts 算子也好，都只是利用了 $z5, z6, z8, z9$ 的像素值。下面可以自定义一个算子，比如：

$$\begin{aligned} gx &= (z7 + z8 + z9) - (z1 + z2 + z3) \\ gy &= (z3 + z6 + z9) - (z1 + z4 + z7) \end{aligned} \tag{16}$$

事实上，这个自定义的算子和著名的 Sobel 算子已经非常接近，只是 Sobel 算子增加了距离权重而已。Sobel 算子的定义如下（与中心点 $Z5$ 更近的点 $Z3, Z4, Z6, Z8$ 的权重为 2，其它对角线上的权重为 1）：

$$\begin{aligned} gx &= (z7 + 2 * z8 + z9) - (z1 + 2 * z2 + z3) \\ gy &= (z3 + 2 * z6 + z9) - (z1 + 2 * z4 + z7) \end{aligned} \tag{17}$$

下面分别用 Roberts 交叉算子和自定义算子进行图像增强，WIN 10 平台，Python 3.7 下调试通过：

```
1  import cv2
2  import numpy as np
3
4  cat = cv2.imread("cat1.jpg", 0)
5  row, column = cat.shape
6  cat_f = np.copy(cat)
7  cat_f = cat_f.astype("float")
8
9  def Roberts_operator(fig, fig_row, fig_column):
10     Roberts = np.zeros((fig_row, fig_column))
11     for x in range(fig_row - 1):
12         for y in range(fig_column - 1):
13             gx = abs(fig[x + 1, y + 1] - fig[x, y])
14             gy = abs(fig[x + 1, y] - fig[x, y + 1])
```

```
            Roberts[x, y] = gx + gy
    sharp_Roberts = fig + Roberts
    sharp_Roberts = np.where(sharp_Roberts < 0, 0, np.where(sharp_Roberts >
        255, 255, sharp_Roberts))
    sharp_Roberts = sharp_Roberts.astype("uint8")
    return sharp_Roberts

def user_defined_operator(fig, fig_row, fig_column):
    user_defined = np.zeros((fig_row, fig_column))
    for x in range(1, fig_row - 1):
        for y in range(1, fig_column - 1):
            gx = abs((fig[x + 1, y - 1] + fig[x + 1, y] + fig[x + 1, y + 1]) -
                     (fig[x - 1, y - 1] + fig[x - 1, y] + fig[x - 1, y + 1]))
            gy = abs((fig[x - 1, y + 1] + fig[x, y + 1] + fig[x + 1, y + 1]) -
                     (fig[x - 1, y - 1] + fig[x, y - 1] + fig[x + 1, y - 1]))
            user_defined[x, y] = gx + gy
    sharp_user_defined = fig + user_defined
    sharp_user_defined = np.where(sharp_user_defined < 0, 0, np.where(
        sharp_user_defined > 255, 255, sharp_user_defined))
    sharp_user_defined = sharp_user_defined.astype("uint8")
    return sharp_user_defined

cv2.imshow("cat", cat)  # or use cv2.imwrite to save the figures
cv2.imshow("sharp_Roberts", Roberts_operator(cat_f, row, column))
cv2.imshow("sharp_user_defined", user_defined_operator(cat_f, row, column))
cv2.waitKey()
```

上述代码将两种算子写成函数进行调用，最终计算结果如下图13，可以明显看出几类算子的区别，邻域用得越多图像增强的感觉越明显：

(a) cat (b) Roberts (c) user_defined

**图 13:** 用两种梯度算子做图像增强

注 上述代码中，第二个函数在定义 $gx, gy$ 时长度过长，理应使用 \ 符号进行换行，但

由于这里有括号将内部内容进行了包裹，所以此时在想换行的地方直接回车，Python 能够自动识别为一条语句而进行合理的缩进，此时就不需要换行符了。

### 2.1.5 图像二阶微分算子

前面介绍的图像梯度以及图像的几个梯度算子，这些本质上都是一阶导数，或一阶微分。其实还有更进一步的二阶算子，将会对灰度变化强烈的地方会更敏感。图像的二阶微分可以通过以下推导建立：

首先还是由于图像是按照像素来离散的，则有：

$$\frac{\partial f}{\partial x} = f'(x) = f(x+1) - f(x) \tag{18}$$

那么二阶微分就是：

$$\begin{aligned} \frac{\partial^2 f}{\partial x^2} = \frac{\partial f'(x)}{\mathrm{d}x} &= f'(x+1) - f'(x) \\ &= f(x+2) - f(x+1) - f(x+1) + f(x) \\ &= f(x+2) - 2f(x+1) + f(x) \end{aligned} \tag{19}$$

如果采用中心差分，则：

$$\frac{\partial^2 f}{\partial x^2} = f(x+1) + f(x-1) - 2f(x) \tag{20}$$

于是，在 $x$ 和 $y$ 方向上，有：

$$\begin{aligned} \frac{\partial^2 f}{\partial x^2} &= f(x+1,y) + f(x-1,y) - 2f(x,y) \\ \frac{\partial^2 f}{\partial y^2} &= f(x,y+1) + f(x,y-1) - 2f(x,y) \end{aligned} \tag{21}$$

如果把 $x$ 方向和 $y$ 方向的二阶导数结合在一起：

$$\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} = f(x+1,y) + f(x-1,y) + f(x,y+1) + f(x,y-1) - 4f(x,y) \tag{22}$$

这实质上就是著名的拉普拉斯二阶微分算子（Laplacian）。可以通过代码看一下实际效果，WIN 10 平台，Python 3.7 下调试通过：

```
1 import cv2
2 import numpy as np
3
4 cat = cv2.imread("cat1.jpg", 0)
```

```
row, column = cat.shape
cat_f = np.copy(cat)
cat_f = cat_f.astype("float")

two = np.zeros((row, column))

for x in range(1, row - 1):
    for y in range(1, column - 1):
        two[x, y] = cat_f[x + 1, y] + cat_f[x - 1, y] + \
                    cat_f[x, y + 1] + cat_f[x, y - 1] - 4 * cat_f[x, y]

sharp = cat_f - two
sharp = np.where(sharp < 0, 0, np.where(sharp > 255, 255, sharp))
sharp = sharp.astype("uint8")

cv2.imshow("cat", cat)
cv2.imshow("sharp_Laplacian", sharp)
cv2.waitKey()
```

计算后的结果如下图14，可以看到，图像增强的效果比前面的一阶微分要好很多。

(a) cat (b) Laplacian

**图 14:** 用 Laplacian 算子做图像增强

需要注意，将原图像与拉普拉斯二阶导数图像合并的时候，必须考虑符号上的差别。可以看到上面的代码中用的是减号，而不是一阶导数中用的加号。到底用加号还是减号，与中心点 $f(x,y)$ 的系数有关. 这个定义的拉普拉斯二阶导数中，$f(x,y)$ 的系数是-4，是负的，原图像就要减去拉普拉斯二阶导数图像。

拉普拉斯二阶导数还有其它的形式，例如：

$$\text{Laplacian} = 4f(x,y) - f(x+1,y) - f(x-1,y) - f(x,y+1) - f(x,y-1) \tag{23}$$

这时 $f(x,y)$ 的系数是正的，原图像就要加上拉普拉斯二阶导数图像。

注意到前面介绍图像一阶导数时用的是绝对值，而二阶导数就没有使用绝对值，且需要考虑系数的正负符号问题，才能决定最后的图像合并是用原图像加上还是减去二阶导数图像。其原因要从图像二阶导数的本质说起。

仍然简化问题，考虑下 $x$ 方向，选取某个像素，如下图15所示：

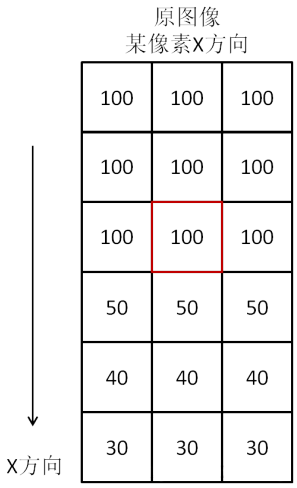



**图 15:** 图像梯度示意图 5

可以看出，在图中标红色框框的像素附近是一个明显的分界线，上面是一片平坦的灰度区域，下面是灰度缓慢变化的区域。而且有着明显的灰度突变：从 100 突变到 50，可以把这个看作图像中物体的轮廓边缘。

根据之前的介绍，可以计算出这个像素 $x$ 方向上的一阶导数和二阶导数，如下图16所示：

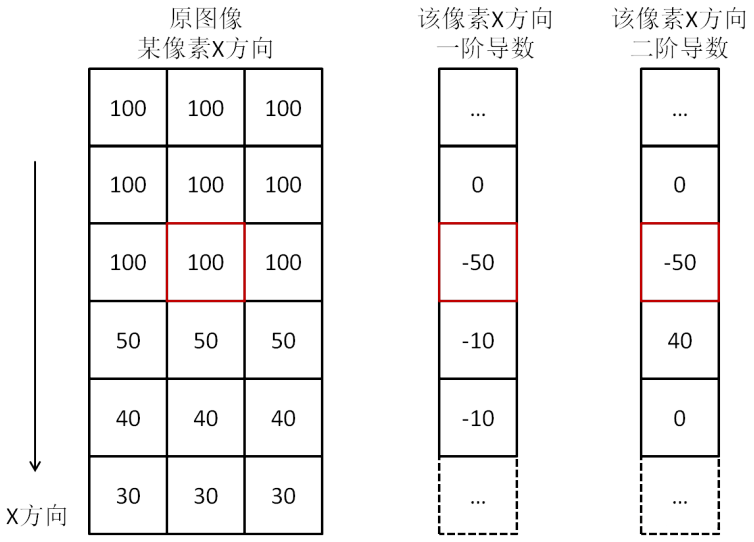



**图 16:** 图像梯度示意图 6

注意到：对于一阶导数，除了灰度突变的地方，其它灰度缓慢变化的地方数值相同，而且符号也相同。而二阶导数在灰度缓慢变化的地方数值为 0，在灰度突变的地方有符

号相反的 2 个数值，即二阶导数产生了一个**像素宽**的**双边缘**。

前面提到，求一阶导数时，用的是绝对值，而二阶导数并没有用绝对值，因为在边缘处，有符号相反的二阶导数值，可以强化这个边缘的对比度。如下图17所示：

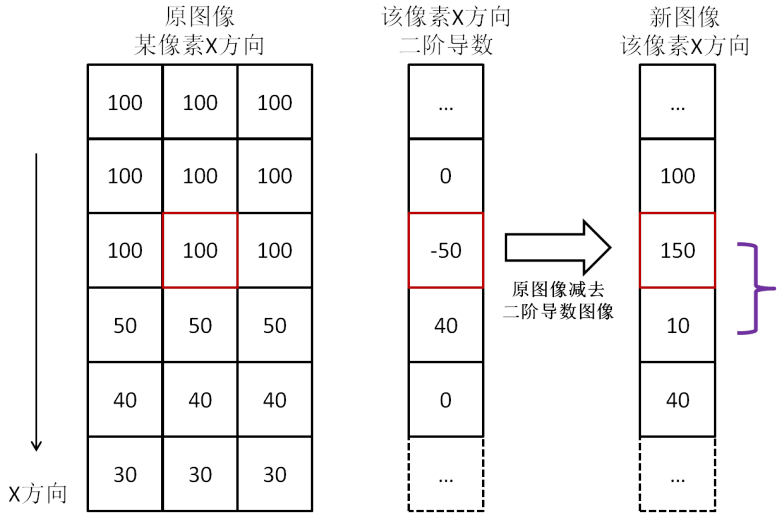



**图 17:** 图像梯度示意图 7

可以看到，原图像与二阶导数图像合并后，在灰度均匀或灰度缓慢变化的地方，图像并没有任何改变；但在灰度突变的边缘处，原来是 100 和 50 的灰度差别，现在是 150 和 10 的灰度差别，对比度增强了很多。因此在合并图像时，就要考虑符号的问题，不然就会适得其反，使得双边缘的对比反差消失。

一般来说，二阶导数比一阶导数获得的物体边界更加细致。但是，显而易见的，二阶导数对噪声点也更加敏感，会放大噪声的影响。下图18可以明显说明这个问题：

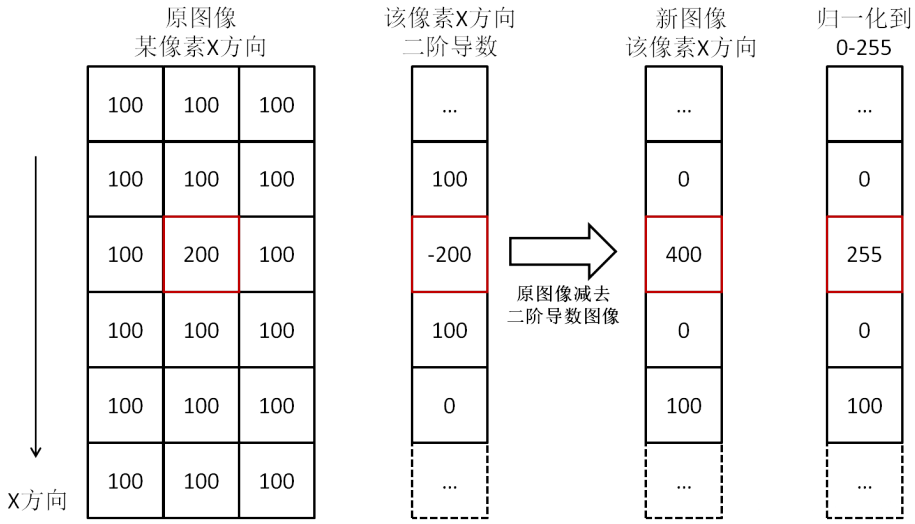



**图 18:** 图像梯度示意图 8

在一片灰度均匀的区域，有一个噪声点，经过二阶导数处理后，噪声点更加孤立明显了，尤其在这些灰度平滑区域更加的显眼，噪声被放大了。

## 2.2 图像滤波器

图像滤波，即在尽量保留图像细节特征的条件下对目标图像的噪声进行抑制，是图像预处理中不可缺少的操作，其处理效果的好坏将直接影响到后续图像处理和分析的有效性和可靠性。

由于成像系统、传输介质和记录设备等的不完善，数字图像在其形成、传输记录过程中往往会受到多种噪声的污染。另外，在图像处理的某些环节当输入的像对象并不如预想时也会在结果图像中引入噪声。这些噪声在图像上常表现为一引起较强视觉效果的孤立像素点或像素块。一般，噪声信号与要研究的对象不相关它以无用的信息形式出现，扰乱图像的可观测信息。对于数字图像信号，噪声表为或大或小的极值，这些极值通过加减作用于图像像素的真实灰度值上，对图像造成亮、暗点干扰，极大降低了图像质量，影响图像复原、分割、特征提取、图像识别等后继工作的进行。要构造一种有效抑制噪声的滤波器必须考虑两个基本问题：**能有效地去除目标和背景中的噪声；同时，能很好地保护图像目标的形状、大小及特定的几何和拓扑结构特征**。

传统的图像滤波手段包括线性卷积滤波、中值滤波、双边滤波等。图像滤波既可以在实域进行，也可以在频域进行。图像滤波可以更改或者增强图像。通过滤波，可以强调一些特征或者去除图像中一些不需要的部分。**滤波是一个邻域操作算子，利用给定像素周围的像素的值决定此像素的最终的输出值**。

图像滤波可以通过公式：

$$O(i,j)=\sum_{m,n} I(i+m,j+n)\cdot K(m,n) \tag{24}$$

其中，$K$ 为滤波器，在很多文献中也称之为**核（kernel）**。常见的应用包括去噪、图像增强、检测边缘、检测角点、模板匹配等。

下面对几大类经典滤波技术作一简单介绍：

### 2.2.1 盒子滤波

盒子滤波是一种非常有用的线性滤波，也叫方框滤波，最简单的均值滤波就是盒子滤波归一化的情况。

在原理上，和均值滤波一样，用一个内核和图像进行卷积：

$$K=\alpha\begin{bmatrix}1 & 1 & \cdots & 1\\ 1 & 1 & \cdots & 1\\ \vdots & \vdots & \ddots & \vdots\\ 1 & 1 & \cdots & 1\end{bmatrix} \tag{25}$$

其中，

$$\alpha = \begin{cases} \frac{1}{\text{ksize.width} * \text{ksize.height}} & \text{when normalize = true} \\ 1 & \text{otherwise} \end{cases} \tag{26}$$

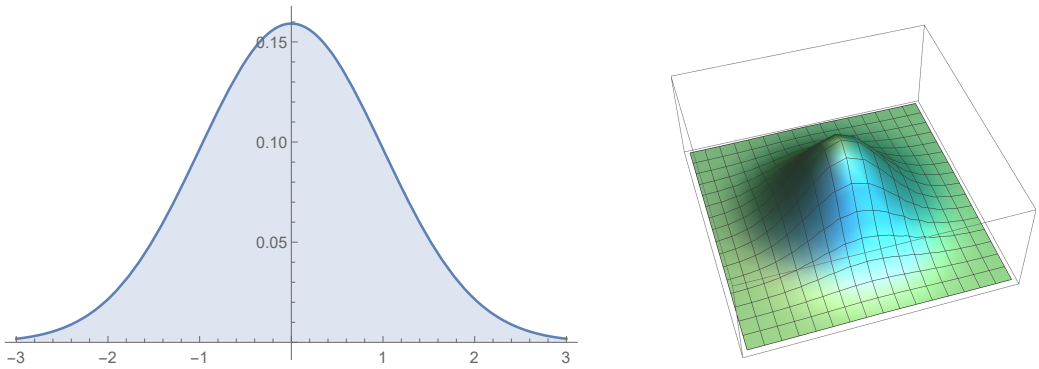
可见，归一化了就是均值滤波；不归一化则可以计算每个像素邻域上的各种积分特性，方差、协方差，平方和等等。

### 2.2.2 高斯滤波

高斯滤波像均值滤波，是简单的取平均值，模板系数都是 1。而图像上的像素实际上是坐标离散但是值却连续的，因为越靠近的点关系越密切，越远离的点关系越疏远。因此，加权平均更合理，距离越近的点权重越大，距离越远的点权重越小。

加权平均的方式就是通过一维高斯函数，亦即传统的正态分布的概率密度函数：

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(x-\mu)^2}{2\sigma^2}} \tag{27}$$




(a) Gaussian 函数，$\mu = 0, \sigma = 1$　　(b) 二维 Gaussian 函数

**图 19:** Gaussian 函数图像

如图19(a)，一维高斯函数离原点距离越近，得到的权重越高，越远离原点，得到的权重越小。

由于图像是二维矩阵，则采用二维高斯函数（如图19(b)），有了这个函数即可计算滤波模板中每个点的权重：

$$f(x,y) = \frac{1}{2\pi\sigma^2} e^{\frac{-(x^2+y^2)}{2\sigma^2}} \tag{28}$$

下面介绍具体的操作步骤：

1. **权重矩阵计算：**
   假定中心点是 (0,0)，那么距离它最近的 8 个点的坐标如下：

| | | |
|---|---|---|
| (-1,1) | (0,1) | (1,1) |
| (-1,0) | **(0,0)** | (1,0) |
| (-1,-1) | (0,-1) | (1,-1) |

更远的点以此类推。将这模板中的坐标 $x, y$ 代入到二维高斯函数中，假定 $\mu = 0, \sigma = 1.5$，则模糊半径为 1 的权重矩阵如下：

| | | |
|---|---|---|
| 0.0454 | 0.0566 | 0.0454 |
| 0.0566 | **0.0707** | 0.0566 |
| 0.0454 | 0.0566 | 0.0454 |

这 9 个点的权重总和等于 0.4787，如果只计算这 9 个点的加权平均，还必须让它们的权重之和等于 1，因此上面 9 个值还要分别除以 0.4787，得到最终的权重矩阵：

| | | |
|---|---|---|
| 0.0947 | 0.1183 | 0.0947 |
| 0.1183 | **0.1478** | 0.1183 |
| 0.0947 | 0.1183 | 0.0947 |

2. **计算高斯滤波的结果：**

   假设现在 3 * 3 高斯滤波器覆盖的图像的像素灰度值为：

| | | |
|---|---|---|
| 14 | 15 | 16 |
| 24 | **25** | 26 |
| 34 | 35 | 36 |

每个点与上边计算得到的 9 个权重值相乘，得到：

| | | |
|---|---|---|
| 14 × 0.0947 | 15 × 0.1183 | 16 × 0.0947 |
| 24 × 0.1183 | **25 × 0.1478** | 26 × 0.1183 |
| 34 × 0.0947 | 35 × 0.1183 | 36 × 0.0947 |

$\longrightarrow$

| | | |
|---|---|---|
| 1.3264 | 1.7748 | 1.5159 |
| 2.8396 | **3.6940** | 3.0763 |
| 3.2212 | 4.1411 | 3.4107 |

将最后这 9 个值加起来，就是中心点的高斯模糊的值。

对所有点重复这个过程，就得到了高斯模糊后的图像。如果原图是彩色图片，可以对 RGB 三个通道分别做高斯模糊。

3. **边界计算：**

   值得注意的是，如果一个点处于边界，周边没有足够的点，一个变通方法是，把已有的点拷贝到另一面的对应位置，模拟出完整的矩阵，或者用零元素来填充。

### 2.2.3 双边滤波

高斯滤波只考虑了周边点与中心点的空间距离来计算得到权重。首先，对于图像滤波来说，一个通常的 intuition 是：**（自然）图像在空间中变化缓慢，因此相邻的像素点会更相近。但是这个假设在图像的边缘处变得不成立。如果在边缘处也用这种思路来进行滤波的话，即认为相邻相近，则得到的结果必然会模糊掉边缘，这是不合理的**。因此考虑再利用像素点的值的大小进行补充，因为边缘两侧的点的像素值差别很大，因此会使得其加权的时候权重具有很大的差别。可以理解成**先根据像素值对要用来进行滤波的邻域做一个分割或分类，再给该点所属的类别相对较高的权重，然后进行邻域加权求和，得到最终结果**。

双边滤波与高斯滤波相比，对于图像的边缘信息能够更好的保留，其原理为一个与空间距离相关的高斯核函数与一个灰度距离相关的高斯函数相乘。

空间距离：

$$e^{-\frac{(x_i-x_c)^2+(y_i-y_c)^2}{2\sigma^2}} \tag{29}$$

其中，$(x_c, y_c)$ 是中心点坐标，比如 $(0,0)$；$(x_i, y_i)$ 为当前点的坐标，$\sigma$ 为空间域标准差。

灰度距离：

$$e^{-\frac{[\text{gray}(x_i,y_i)-\text{gray}(x_c,y_c)]^2}{2\sigma^2}} \tag{30}$$

其中，$\text{gray}(x_i, y_i)$ 是当前像素点的灰度值；$\text{gray}(x_c, y_c)$ 为是模板中覆盖图片区域的中心点像素的灰度值，也就是 $(0,0)$ 处的灰度值，$\sigma$ 为值域标准差。

对于高斯滤波，仅用空间距离的权值系数核与图像卷积后确定中心点的灰度值。即认为离中心点越近，其权值系数越大。

双边滤波中加入了对灰度信息的权重，即在邻域内，灰度值越接近中心点灰度值的点的权值更大，灰度值相差大的点权重越小。其权重大小则由值域高斯函数确定。

两者权重系数相乘，得到最终的卷积模板，由于双边滤波需要每个中心点邻域的灰度信息来确定其系数，所以速度比一般的滤波慢得多，而且计算量增长速度为核的大小的平方。

参数的选择：空间域 $\sigma$ 的选取，和值域 $\sigma$ 的选取。

结论：$\sigma$ 越大，边缘越模糊；$\sigma$ 越小，边缘越清晰。

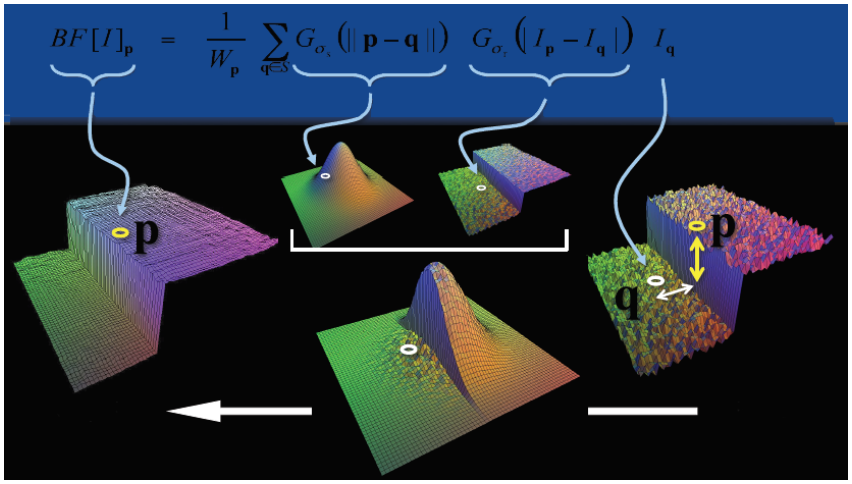



图 20: 双边滤波示意图（来自网络）

## 2.3 微分算子与滤波器

前面提到，可以用一阶微分算子和二阶微分算子来增强图像，由于是增强了图像中的物体边缘轮廓，起到了锐化图像的效果，因此这些算子操作可用于图像锐化。

同时前面介绍的滤波器可以用来滤波，其实微分算子也可以使用滤波器的形式。

这里分别选择一个一阶微分算子和一个二阶微分算子作为示例：Sobel 算子和 Laplacian 算子。首先，还是根据之前图11中所示的图像像素 8 领域，对于 Laplacian 算子，其二阶梯度表示是：

$$\text{Laplacian} = z8 + z2 + z6 + z4 - 4 * z5 \tag{31}$$

如果把 8 邻域都考虑进去，形成一个系数矩阵，就是：

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

那么对图像实施 Laplacian 算子操作，实质上就是用这个系数矩阵与图像中的任意一点的邻域区域矩阵进行矩阵点乘，然后求点乘后矩阵的和。写出公式就是：

$$g(x,y) = \sum_{a=-n}^{n} \sum_{b=-n}^{n} w(a,b) f(x+a, y+b) \tag{32}$$

其中 $w(a,b)$ 就是上面的 Laplacian 算子矩阵。所以，像 Laplacian 算子这样的二阶微分算子，可以使用滤波器（图像降噪、图像模糊）的形式，这为统一计算模型带来了极大方便。

关于图像微分算子，最后简单说一下 Canny 边缘检测，前面有使用过 OpenCV 中的 Canny 算法但没有解释其原理。Canny 边缘检测是一种非常流行的边缘检测算法，是 John Canny 在 1986 年提出的。它是一个多阶段的算法，即由多个步骤构成，所以比一般单独只使用微分算子效果要更好。

这种边缘检测算法的步骤如下：

1. **图像降噪**：第一步就需要先去除噪声，因为噪声就是灰度变化很大的地方，所以容易被识别为伪边缘；
2. **计算图像梯度**：计算图像梯度，得到可能边缘；
3. **非极大值抑制**：通常灰度变化的地方都比较集中，将局部范围内的梯度方向上，灰度变化最大的保留下来，其它的不保留，这样可以剔除掉一大部分的点；
4. **阈值筛选**：通过非极大值抑制后，仍然有很多的可能边缘点，进一步的设置一个双阈值，即低阈值（low），高阈值（high）。灰度变化大于 high 的，设置为强边缘像素，低于 low 的，剔除。在 low 和 high 之间的设置为弱边缘。进一步判断，如果其领域内有强边缘像素，保留，如果没有，剔除。这样做的目的是只保留强边缘轮廓的话，有些边缘可能不闭合，需要从满足 low 和 high 之间的点进行补充，使得边缘尽可能的闭合。

在 OpenCV 中，Canny 算法并没有将图像降噪包含在内，需要在 Canny 函数外面执行图像降噪的过程，例如使用 GaussianBlur 函数。在 Canny 函数中只有 low 和 high 双阈值参数需要手动设置。

演示代码如下，WIN 10 平台，Python 3.7 下调试通过：

```
1 import cv2
2
3 cat = cv2.imread("cat1.jpg", 0)  # 读入图像
4 catBlur = cv2.GaussianBlur(cat, (5, 5), 0)  # 通过调整高斯模糊的核大小进行降噪控制
5 # Canny边缘检测，50为低阈值low，150为高阈值high
6 canny = cv2.Canny(catBlur, 50, 150)  # 调整low和high双阈值，能够得到不同的边缘效果
7 cv2.imshow("cat", cat)
8 cv2.imshow("canny", canny)
9 cv2.waitKey()
```

结果如下图21。可以看到通过调整高斯模糊的参数，边缘检测的效果比之前直接使用 Canny 函数要好很多。

(a) cat (b) Laplacian

图 21: Canny 边缘检测

# 3 图像频域处理

## 3.1 一元函数的 Taylor 展开

在微积分中，Taylor 展开是应用较为广泛的一种数学手段。但实际使用时，常容易陷入概念模糊、公式复杂难记等问题中，所以这里还是对其作一简要介绍。另外，Taylor 展开确实可以很复杂，但涉及到相关领域，一般只会用到一元函数的 Taylor 展开。

**定理 3.1 (泰勒（Taylor）展开式)** *若函数 $f(x)$ 连续且具有 $n$ 阶导数，则有*

$$f(x)=f(x_0)+\frac{f'(x_0)}{1!}(x-x_0)+\frac{f''(x_0)}{2!}(x-x_0)^2+\ldots+\frac{f^{(n)}(x_0)}{n!}(x-x_0)^n \tag{33}$$

**证明.** 首先对于任意连续函数，可以考虑 $n$ 次多项式拟合（这也是 Taylor 展开动机的精髓），如下：

$$f(x)=a_0+a_1x+a_2x^2+\ldots+a_nx^n \tag{34}$$

对上式进行连续求导得：

$$\begin{aligned}
&f'(x)=a_1+2\cdot a_2x+\ldots+n\cdot a_nx^{n-1}\\
&f''(x)=1\cdot 2\cdot a_2+\ldots+(n-1)(n)\cdot a_nx^{n-2}\\
&\ldots\ldots\\
&f^{(n)}(x)=1\cdot 2\cdot 3\cdot\ldots(n-2)\cdot(n-1)\cdot(n)a_n
\end{aligned} \tag{35}$$

令 $x=0$，可得下面被称为 $n$ 次多项式的麦克劳林（Maclaurin）公式：

$$f(x)=f(0)+\frac{f'(0)}{1!}x+\frac{f''(0)}{2!}x^2+\ldots+\frac{f^{(n)}(0)}{n!}x^n \tag{36}$$

显然，多项式拟合也可以以 $(x-x_0)$ 的幂次展开，即写为

$$f(x)=A_0+A_1(x-x_0)+A_2(x-x_0)^2+\ldots+A_n(x-x_0)^n \tag{37}$$

再按上述方法进行求解，便可得到 Taylor 展开的表达式。

取 Taylor 展开的一阶展开式，会发现其与下述**一元微分**的概念很相似：

$$\Delta y=f(x+\Delta x)-f(x)=f'(x)\cdot\Delta x+o(\Delta x) \tag{38}$$

其中，$o(\Delta x)$ 为 $\Delta x$ 的高阶小量。所以，对于 Taylor 展开而言，当 $x$ 与 $x_0$ 较为接近时，可用下述近似来替代：

$$f(x)\approx f(x_0)+f'(x_0)\cdot(x-x_0) \tag{39}$$

## 3.2 Fourier 变换

与 Taylor 展开类似，对于任意周期函数 $f(t)$，一个直观的念头是也希望可以采用某种类似的方式对其进行多项式拟合。考虑到周期函数的特殊性，其多项式的组成必然也需要用到周期函数。

### 3.2.1 Fourier 级数

**定理 3.2 (傅里叶（Fourier）级数)** 在区间 $\left[-\frac{T}{2},\frac{T}{2}\right]$ 内可积的周期函数 $f(t)$ 可以进行如下三角函数形式的展开：

$$f(t)=\frac{a_0}{2}+\sum_{n=1}^{\infty}\left[a_n\cos(n\omega_0 t)+b_n\sin(n\omega_0 t)\right] \tag{40}$$

其中，常值分量

$$a_0=\frac{2}{T}\int_{-\frac{T}{2}}^{\frac{T}{2}}f(t)\mathrm{d}t \tag{41}$$

余弦分量的幅值

$$a_n=\frac{2}{T}\int_{-\frac{T}{2}}^{\frac{T}{2}}f(t)\cos(n\omega_0 t)\mathrm{d}t \tag{42}$$

正弦分量的幅值

$$b_n=\frac{2}{T}\int_{-\frac{T}{2}}^{\frac{T}{2}}f(t)\sin(n\omega_0 t)\mathrm{d}t \tag{43}$$

式中，$T$ 为函数 $f(t)$ 的周期；$\omega_0$ 为角频率，满足 $\omega_0=\frac{2\pi}{T}$；$n$ 的取值为 $n=1,2,3,\cdots$。

**证明.** 考察形如 $1,\cos(\omega_0 t),\sin(\omega_0 t),\cos(2\omega_0 t),\sin(2\omega_0 t),\cdots,\cos(n\omega_0 t),\sin(n\omega_0 t),\cdots$ 的一系列函数系，会发现其中任意两个函数均具有“正交性”的特征，即满足

$$\begin{cases} \displaystyle\int_{-\frac{T}{2}}^{\frac{T}{2}} 1\cdot\cos(n\omega_0 t)\mathrm{d}t = 0, \quad \int_{-\frac{T}{2}}^{\frac{T}{2}} 1\cdot\sin(n\omega_0 t)\mathrm{d}t = 0 \\ \displaystyle\int_{-\frac{T}{2}}^{\frac{T}{2}} \cos(m\omega_0 t)\cdot\cos(n\omega_0 t)\mathrm{d}t = 0, \quad m\neq n \\ \displaystyle\int_{-\frac{T}{2}}^{\frac{T}{2}} \sin(m\omega_0 t)\cdot\sin(n\omega_0 t)\mathrm{d}t = 0, \quad m\neq n \\ \displaystyle\int_{-\frac{T}{2}}^{\frac{T}{2}} \cos(m\omega_0 t)\cdot\sin(n\omega_0 t)\mathrm{d}t = 0 \end{cases} \tag{44}$$

其中，$m=1,2,3,\cdots$，$n=1,2,3,\cdots$。上式(44)的中间两式，当 $m=n$ 时积分为 $\frac{T}{2}$。另外对于上式(44)中的后三式需要用到几个积化和差公式，如下：

$$\begin{cases} \cos\alpha\cos\beta = \dfrac{1}{2}\left[\cos\left(\alpha+\beta\right)+\cos\left(\alpha-\beta\right)\right] \\ \sin\alpha\sin\beta = \dfrac{1}{2}\left[\cos\left(\alpha+\beta\right)-\cos\left(\alpha-\beta\right)\right] \\ \cos\alpha\sin\beta = \dfrac{1}{2}\left[\sin\left(\alpha+\beta\right)-\sin\left(\alpha-\beta\right)\right] \end{cases} \tag{45}$$

有了函数系的正交性特征，则可对公式(40)两边进行积分，得到 $a_0$ 的表达式：

$$a_0 = \frac{2}{T}\int_{-\frac{T}{2}}^{\frac{T}{2}} f(t)\mathrm{d}t \tag{46}$$

再对公式(40)两边同时乘以 $\cos(n\omega_0 t)$ 后进行积分，可得到 $a_n$ 的表达式：

$$a_n = \frac{2}{T}\int_{-\frac{T}{2}}^{\frac{T}{2}} f(t)\cos(n\omega_0 t)\mathrm{d}t \tag{47}$$

最后对公式(40)两边同时乘以 $\sin(n\omega_0 t)$ 后进行积分，可得到 $b_n$ 的表达式：

$$b_n = \frac{2}{T}\int_{-\frac{T}{2}}^{\frac{T}{2}} f(t)\sin(n\omega_0 t)\mathrm{d}t \tag{48}$$

从以上推导可以看出，若想要进行上述三角函数的展开，只需 $f(t)$ 在周期 $T$ 内满足可积条件即可。

**注** 值得说明的是，上述 Fourier 级数展开的前置条件是不完备的。若要满足 Fourier 级数的逐点收敛性，需要符合狄利克雷（Dirichlet）条件。该部分内容一般包含于数学分析书籍中，这里不做具体论述。由于本书探讨的绝大部分声学信号均符合 Dirichlet 条件，所以这里仅用“周期内可积”来简化描述满足 Fourier 级数展开的条件。

**例 3.1** 试分析方波、锯齿波和三角波的 Fourier 级数展开。

首先给出方波、锯齿波和三角波的解析表达式，分别用函数 $f_{\text{square}}(t)$、$f_{\text{sawtooth}}(t)$ 和 $f_{\text{triangle}}(t)$ 表示。为简化说明可令振幅为 1，在一个周期 $T$ 内有

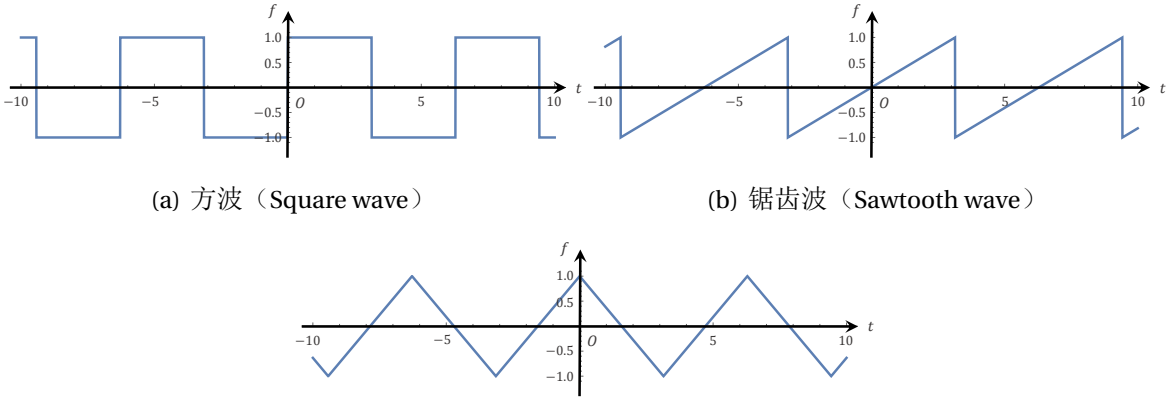
$$f_{\text{square}}(t)=\begin{cases}-1, & t\in\left[-\dfrac{T}{2},0\right]\\ 1, & t\in\left[0,\dfrac{T}{2}\right]\end{cases} \tag{49}$$

$$f_{\text{sawtooth}}(t)=\frac{2}{T}t,\quad t\in\left[-\frac{T}{2},\frac{T}{2}\right] \tag{50}$$

$$f_{\text{triangle}}(t)=\begin{cases}1+\dfrac{4}{T}t, & t\in\left[-\dfrac{T}{2},0\right]\\ 1-\dfrac{4}{T}t, & t\in\left[0,\dfrac{T}{2}\right]\end{cases} \tag{51}$$

注意上述三式中，不必纠结分段函数的间断点处取值如何。在计算 Fourier 级数展开时，是分别对区间内进行积分，所以均可看作是闭区间。

令周期 $T=2\pi$（角频率 $\omega_0$ 则简化为 1），三者分别的函数图如下图22所示：




(a) 方波（Square wave）　(b) 锯齿波（Sawtooth wave）

(c) 三角波（Triangle wave）

**图 22:** 方波 $f_{\text{square}}(t)$、锯齿波 $f_{\text{sawtooth}}(t)$ 和三角波 $f_{\text{triangle}}(t)$ 的函数图

首先对 $f_{\text{square}}(t)$ 进行 Fourier 级数展开的求解。根据图22所示，$f_{\text{square}}(t)$ 为奇函数，即满足 $f_{\text{square}}(t)=-f_{\text{square}}(-t)$，代回 Fourier 级数展开式(40)有 $a_n=0$，同时 $f_{\text{square}}(t)$ 在一个周期内的积分面积为零，得到 $a_0=0$，所以 $f_{\text{square}}(t)$ 的 Fourier 级数展开仅存在正弦级数项，且被积函数为偶函数，仅需计算周期 $T$ 一半的积分再乘以 2 即可。此时求得：

$$\begin{aligned}b_n&=\frac{2}{T}\int_{-\frac{T}{2}}^{\frac{T}{2}}f_{\text{square}}(t)\sin(n\omega_0 t)\mathrm{d}t=\frac{2}{\pi}\int_0^{\pi}f_{\text{square}}(t)\sin(nt)\mathrm{d}t\\&=\frac{2}{\pi}\int_0^{\pi}\sin(nt)\mathrm{d}t=\frac{2}{n\pi}\left[-\cos(nt)\Big|_0^{\pi}\right]=\frac{2}{n\pi}\left[1-\cos(n\pi)\right]\end{aligned} \tag{52}$$

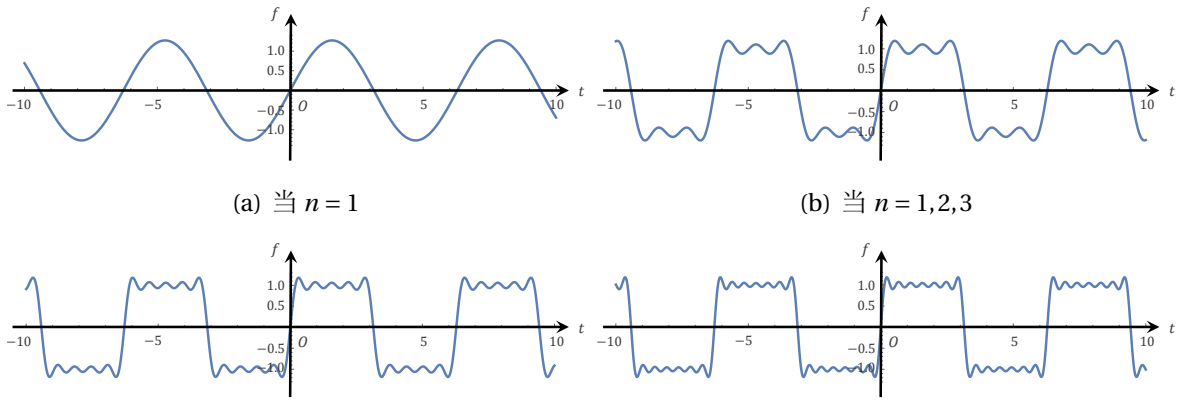
可以看到，此时 $b_n$ 的取值与 $n$ 相关，即

$$b_n = \begin{cases} \dfrac{4}{n\pi}, & n\text{为奇数} \\ 0, & n\text{为偶数} \end{cases} \tag{53}$$

于是 $f_{\text{square}}(t)$ 的 Fourier 级数展开可表示为如下形式：

$$\begin{aligned} f_{\text{square}}(t) &= \sum_{n=1}^{\infty} \frac{4}{(2n-1)\pi} \sin[(2n-1)t] = \frac{4}{\pi} \sum_{n=1}^{\infty} \frac{\sin[(2n-1)t]}{2n-1} \\ &= \frac{4}{\pi}\left(\sin t + \frac{1}{3}\sin 3t + \frac{1}{5}\sin 5t + \cdots\right) \end{aligned} \tag{54}$$

不断增加 Fourier 级数的项数，可以看到上述表达式的波形逐渐趋向于 $f_{\text{square}}(t)$ 的形状（间断点处取值为左极限与右极限的平均值），如下图23所示：




(a) 当 $n = 1$　　(b) 当 $n = 1, 2, 3$

(c) 当 $n = 1, 2, \cdots, 5$　　(d) 当 $n = 1, 2, \cdots, 7$

**图 23:** 不同项数下 $f_{\text{square}}(t)$ 的 Fourier 级数展开波形

接下来对 $f_{\text{sawtooth}}(t)$ 进行 Fourier 级数展开的求解。根据图22所示，$f_{\text{sawtooth}}(t)$ 与 $f_{\text{square}}(t)$ 类似，也为奇函数，同时 $f_{\text{sawtooth}}(t)$ 在一个周期内的积分面积也为零，所以其 Fourier 级数展开也仅存在正弦级数项，且被积函数也为偶函数，仅需计算周期 $T$ 一半的积分再乘以 2 即可。此时求得：

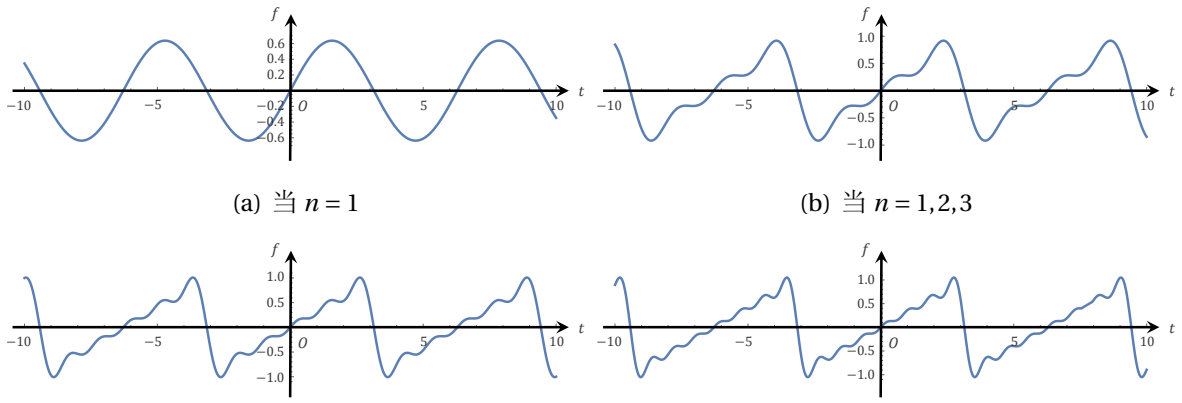
$$\begin{aligned} b_n &= \frac{2}{T}\int_{-\frac{T}{2}}^{\frac{T}{2}} f_{\text{sawtooth}}(t)\sin(n\omega_0 t)\mathrm{d}t = \frac{2}{\pi}\int_0^{\pi} \frac{1}{\pi} t\sin(nt)\mathrm{d}t \\ &= -\frac{2}{n\pi^2}\left[\int_0^{\pi} t\mathrm{d}(\cos nt)\right] = -\frac{2}{n\pi^2}\left[t\cos(nt)\Big|_0^{\pi} - \int_0^{\pi}\cos(nt)\mathrm{d}t\right] \\ &= -\frac{2}{n\pi^2}\cdot\pi\cos(n\pi) = (-1)^{n+1}\frac{2}{n\pi} \end{aligned} \tag{55}$$

上式第二行的推导，用到了微积分中的分部积分法。另外，考虑到此时 $\cos(n\pi)$ 可以替换为 $(-1)^n$，所以可以采取最后一步的化简方式。

于是 $f_{\text{sawtooth}}(t)$ 的 Fourier 级数展开可表示为如下形式：

$$f_{\text{sawtooth}}(t)=\sum_{n=1}^{\infty}(-1)^{n+1}\frac{2}{n\pi}\sin(nt)=\frac{2}{\pi}\left(\sin t-\frac{1}{2}\sin 2t+\frac{1}{3}\sin 3t-\cdots\right) \tag{56}$$

不断增加 Fourier 级数的项数，可以看到上述表达式的波形逐渐趋向于 $f_{\text{sawtooth}}(t)$ 的形状（间断点处取值为左极限与右极限的平均值），如下图24所示：




(a) 当 $n=1$　　(b) 当 $n=1,2,3$

(c) 当 $n=1,2,\cdots,5$　　(d) 当 $n=1,2,\cdots,7$

**图 24:** 不同项数下 $f_{\text{sawtooth}}(t)$ 的 Fourier 级数展开波形

最后对 $f_{\text{triangle}}(t)$ 进行 Fourier 级数展开的求解。根据图22所示，$f_{\text{triangle}}(t)$ 为偶函数，即满足 $f_{\text{triangle}}(t)=f_{\text{triangle}}(-t)$，代回 Fourier 级数展开式(40)有 $b_n=0$，同时 $f_{\text{triangle}}(t)$ 在一个周期内的积分面积也为零，得到 $a_0=0$，所以 $f_{\text{triangle}}(t)$ 的 Fourier 级数展开仅存在余弦级数项，且也为偶函数，仅需计算周期 $T$ 一半的积分再乘以 2 即可。此时求得：

$$\begin{aligned}a_n&=\frac{2}{T}\int_{-\frac{T}{2}}^{\frac{T}{2}}f_{\text{triangle}}(t)\cos(n\omega_0 t)\mathrm{d}t=\frac{2}{\pi}\int_0^{\pi}f_{\text{triangle}}(t)\cos(nt)\mathrm{d}t\\&=\frac{2}{\pi}\int_0^{\pi}\cos(nt)\mathrm{d}t-\frac{2}{\pi^2}\int_0^{\pi}2t\cos(nt)\mathrm{d}t=-\frac{4}{n\pi^2}\left[\int_0^{\pi}t\mathrm{d}(\sin nt)\right]\\&=-\frac{4}{n\pi^2}\left[t\sin(nt)\Big|_0^{\pi}-\int_0^{\pi}\sin(nt)\mathrm{d}t\right]=\frac{4}{n^2\pi^2}\left[\cos(n\pi)-1\right]\end{aligned} \tag{57}$$

上式第二行到第三行的推导，同样用到了微积分中的分部积分法。

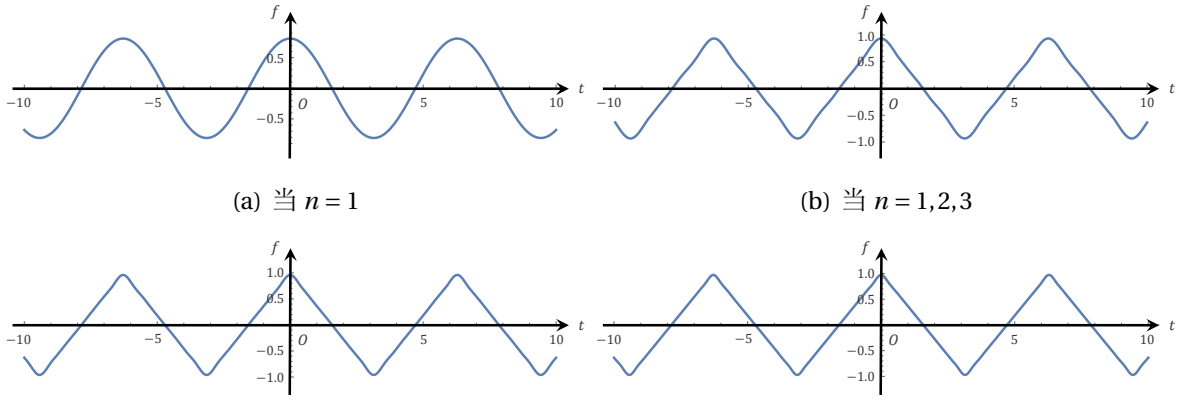
可以看到，此时 $a_n$ 的取值也与 $n$ 相关，即

$$a_n=\begin{cases}\dfrac{8}{n^2\pi^2}, & n\text{为奇数}\\ 0, & n\text{为偶数}\end{cases} \tag{58}$$

于是 $f_{\text{triangle}}(t)$ 的 Fourier 级数展开可表示为如下形式：

$$\begin{aligned}f_{\text{triangle}}(t)&=\sum_{n=1}^{\infty}\frac{8}{(2n-1)^2\pi^2}\cos[(2n-1)t]=\frac{8}{\pi^2}\sum_{n=1}^{\infty}\frac{\cos[(2n-1)t]}{(2n-1)^2}\\&=\frac{8}{\pi^2}\left(\cos t+\frac{1}{3^2}\cos 3t+\frac{1}{5^2}\cos 5t+\cdots\right)\end{aligned} \tag{59}$$

不断增加 Fourier 级数的项数，可以看到上述表达式的波形逐渐趋向于 $f_{\text{triangle}}(t)$ 的形状，如下图25所示：




(a) 当 $n=1$　　(b) 当 $n=1,2,3$

(c) 当 $n=1,2,\cdots,5$　　(d) 当 $n=1,2,\cdots,7$

**图 25:** 不同项数下 $f_{\text{triangle}}(t)$ 的 Fourier 级数展开波形

以上例题3.1仅给出了方波、锯齿波和三角波所对应的简化函数表达式。若波形振幅与角频率不做归一化处理，也并不需要重新计算便可直接通过上述结果得到各自最终的 Fourier 级数展开式。对于振幅来说，仅需整体乘以振幅的系数；对于角频率来说，则需要在正弦或余弦级数项中增加角频率系数 $\omega_0$。事实上，随着波形在选定坐标系中的位置不同（上下左右平移），或进行镜像翻转（锯齿波的锯齿可以“向右”也可以“向左”），其 Fourier 级数展开也会略有不同，但同样不需要重新计算，直接通过函数关系进行替换即可（例如对 $y$ 轴镜像只需用 $f(-t)$ 替换原来的 $f(t)$）。不过，无论如何变化，方波、锯齿波和三角波 Fourier 级数展开的整体性质仍然保持不变：

注 针对 Fourier 级数展开，方波和三角波仅存在奇数频率的无穷级数项，而锯齿波存在全整数频率的无穷级数项。

### 3.2.2 Fourier 积分

上一节考虑了任意周期函数 $f(t)$ 的 Fourier 级数展开。一个自然的想法即是将函数周期 $T$ 扩展至无穷，此时区间 $\left[-\frac{T}{2},\frac{T}{2}\right]\to[-\infty,\infty]$，函数 $f(t)$ 则化为更一般的**非周期函数**。这便是本节要介绍的 **Fourier 积分**。

在此之前，重新梳理下 Fourier 级数中“区间内可积”的概念。当区间 $\left[-\frac{T}{2},\frac{T}{2}\right]\to[-\infty,\infty]$ 时，函数 $f(t)$ 仍然可积，需要满足

$$\int_{-\infty}^{\infty}\left|f(t)\right|\mathrm{d}t<\infty \tag{60}$$

注 上述定义引出如下两条性质：

1. 根据公式(60)，进一步有

$$\int_{-\infty}^{\infty}\left|f(t)\right|\mathrm{d}t \geqslant \int_{-\infty}^{\infty} f(t)\mathrm{d}t \to \int_{-\infty}^{\infty} f(t)\mathrm{d}t = \text{有限值} \tag{61}$$

2. 当 $t \to \pm\infty$ 时，$f(t) \to 0$。

以上说明与微积分中广义积分（反常积分）的收敛条件类似。

所以，若想将周期函数 $f(t)$ 的 Fourier 级数展开推广至 $[-\infty,\infty]$ 的非周期情形，实际上对 $f(t)$ 提出了更加严苛的要求。

**定理 3.3 (Fourier 积分)** 在整个区间 $[-\infty,\infty]$ 内可积的函数 $f(t)$ 可以以连续频谱 $\omega$ 的形式进行积分表示：

$$f(t) = \int_0^{\infty} [A(\omega)\cos\omega t + B(\omega)\sin\omega t]\mathrm{d}\omega \tag{62}$$

其中，

$$A(\omega) = \frac{1}{\pi}\int_{-\infty}^{\infty} f(t)\cos\omega t\mathrm{d}t \tag{63}$$

$$B(\omega) = \frac{1}{\pi}\int_{-\infty}^{\infty} f(t)\sin\omega t\mathrm{d}t \tag{64}$$

**证明.** 首先，对于定理3.2中 Fourier 级数的表达式和各个参数，当 $T \to \infty$ 时，$n\omega_0$ 趋向于连续频谱 $\omega$，$\Delta\omega = (n+1)\omega_0 - n\omega_0 = \omega_0$ 趋向于微元 $\mathrm{d}\omega$，且有

$$\sum_{n=1}^{\infty}\cdots\Delta\omega \xrightarrow{T\to\infty} \int_0^{\infty}\cdots\mathrm{d}\omega \tag{65}$$

此时，若函数 $f(t)$ 在整个区间 $[-\infty,\infty]$ 内可积，结合关系式 $\omega_0 = \frac{2\pi}{T}$ 可求得：

$$a_0 = \frac{2}{T}\int_{-\frac{T}{2}}^{\frac{T}{2}} f(t)\mathrm{d}t \xrightarrow{T\to\infty} 0 \tag{66}$$

$$\begin{aligned}\sum_{n=1}^{\infty} a_n\cos(n\omega_0 t) &= \sum_{n=1}^{\infty}\frac{\Delta\omega}{\pi}\left[\int_{-\frac{T}{2}}^{\frac{T}{2}} f(t)\cos(n\omega_0 t)\mathrm{d}t\right]\cos(n\omega_0 t)\\ &\xrightarrow{T\to\infty} \int_0^{\infty}\underbrace{\left[\frac{1}{\pi}\int_{-\infty}^{\infty} f(t)\cos\omega t\mathrm{d}t\right]}_{A(\omega)}\cos\omega t\mathrm{d}\omega\end{aligned} \tag{67}$$

$$\begin{aligned}\sum_{n=1}^{\infty} b_n\sin(n\omega_0 t) &= \sum_{n=1}^{\infty}\frac{\Delta\omega}{\pi}\left[\int_{-\frac{T}{2}}^{\frac{T}{2}} f(t)\sin(n\omega_0 t)\mathrm{d}t\right]\sin(n\omega_0 t)\\ &\xrightarrow{T\to\infty} \int_0^{\infty}\underbrace{\left[\frac{1}{\pi}\int_{-\infty}^{\infty} f(t)\sin\omega t\mathrm{d}t\right]}_{B(\omega)}\sin\omega t\mathrm{d}\omega\end{aligned} \tag{68}$$

以上各式代回 Fourier 级数的表达式(40)，则公式(62)得证。

**例 3.2** *试求解矩形（rect）函数的 Fourier 积分。*

首先给出 rect 函数的解析表达式（$\tau$ 为一固定参数）：

$$\mathrm{rect}(t)=\begin{cases}\dfrac{1}{\tau}, & |t|\le\dfrac{\tau}{2}\\ 0, & |t|>\dfrac{\tau}{2}\end{cases} \tag{69}$$

对 rect 函数进行全定义域的积分，会发现结果恒为 1，满足可积条件：

$$\int_{-\infty}^{\infty}\mathrm{rect}(t)\mathrm{d}t=\int_{-\frac{\tau}{2}}^{\frac{\tau}{2}}\frac{1}{\tau}\mathrm{d}t\equiv 1 \tag{70}$$

考虑到 $\mathrm{rect}(t)$ 为偶函数，对于公式(64)，得到 $B(\omega)=0$。而

$$A(\omega)=\frac{1}{\pi}\int_{-\frac{\tau}{2}}^{\frac{\tau}{2}}\frac{1}{\tau}\cos\omega t\mathrm{d}t=\frac{2}{\omega\pi\tau}\sin\omega t\Big|_0^{\frac{\tau}{2}}=\frac{2}{\omega\pi\tau}\sin\frac{\omega\tau}{2} \tag{71}$$

代入公式(62)，则 rect 函数的 Fourier 积分为

$$\mathrm{rect}(t)=\frac{1}{\pi}\int_0^{\infty}\frac{\sin(\omega\tau/2)\cos\omega t}{\omega\tau/2}\mathrm{d}\omega \tag{72}$$

特别对上式令 $\tau=2$，取 $t=0$，可以得到一个特殊的积分关系式：

$$\mathrm{rect}(0)=\frac{1}{\pi}\int_0^{\infty}\frac{\sin\omega}{\omega}\mathrm{d}\omega=\frac{1}{2}\rightarrow\int_0^{\infty}\frac{\sin\omega}{\omega}\mathrm{d}\omega=\frac{\pi}{2} \tag{73}$$

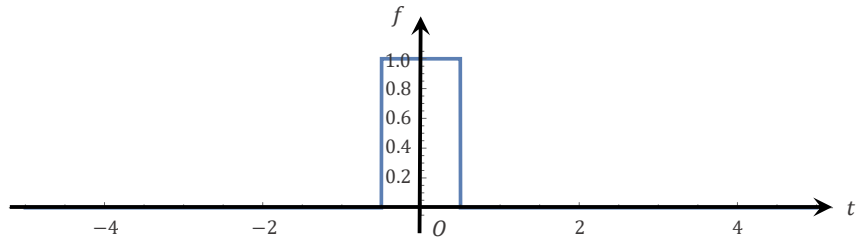
由于上式中被积函数为偶函数，若再采用变量替换 $\omega=\pi t$（也可只移动参数 $\pi$，但做变量替换更能保持函数形式的整体性），则可得到归一化处理后的积分构造式：

$$\int_{-\infty}^{\infty}\frac{\sin\pi t}{\pi t}\mathrm{d}t=1 \tag{74}$$

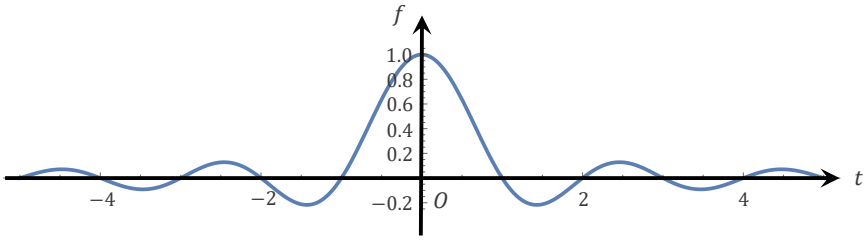
其中，$\frac{\sin\pi t}{\pi t}$ 一般被定义为 sinc 函数，是一个在整个区间 $[-\infty,\infty]$ 内有值且满足可积条件（积分为 1）的特殊函数。

下图26分别给出了 rect 函数（取 $\tau=1$）和 sinc 函数的函数图：

注 *例题3.2中得到 sinc 函数表达式的方式较为巧妙。事实上，若想要证明 sinc 函数在全区间 $[-\infty,\infty]$ 上的积分为 1，还有复变函数中的留数定理、构造重积分变换式等多种方法。不过借助 Fourier 积分，可以很好地利用其收敛性的先决条件（Dirichlet 条件），证明过程更加完备，而且思路较为直观。*




(a) rect 函数（$\tau = 1$）




(b) sinc 函数

图 26: rect 函数和 sinc 函数的函数图

### 3.2.3 连续时间 Fourier 变换

Fourier 级数展开给出了任意周期函数可以表示为一系列三角函数叠加的数学结论，而 Fourier 积分将周期函数的限制进行延拓，推广至非周期函数。从原理上看，这种表述较为清晰，但整体公式形式偏繁琐，而且无穷级数和广义积分的表达式也不太容易看出其具体含义。根据 Euler 公式，可以进一步化简，首先将 Fourier 级数的三角展开式以**复指数形式**表示。

**定理 3.4 (Fourier 级数复指数展开)** 在区间 $\left[-\frac{T}{2}, \frac{T}{2}\right]$ 内可积的周期函数 $f(t)$ 可以进行如下复指数形式的展开：

$$f(t) = \sum_{n=-\infty}^{\infty} F(n\omega_0) \mathrm{e}^{\mathrm{j}n\omega_0 t} \tag{75}$$

其中，

$$F(n\omega_0) = \frac{1}{T} \int_{-\frac{T}{2}}^{\frac{T}{2}} f(t) \mathrm{e}^{-\mathrm{j}n\omega_0 t} \mathrm{d}t \tag{76}$$

式中，$n$ 的取值为 $n = 0, \pm 1, \pm 2, \cdots$。

**证明.** 根据 Euler 公式，可以得到：

$$\begin{cases} \cos n\omega_0 t = \dfrac{1}{2}\left(\mathrm{e}^{\mathrm{j}n\omega_0 t} + \mathrm{e}^{-\mathrm{j}n\omega_0 t}\right) \\ \sin n\omega_0 t = -\dfrac{1}{2}\mathrm{j}\left(\mathrm{e}^{\mathrm{j}n\omega_0 t} - \mathrm{e}^{-\mathrm{j}n\omega_0 t}\right) \end{cases} \tag{77}$$

此时公式(40)可以改写为

$$f(t)=\frac{a_0}{2}+\sum_{n=1}^{\infty}\left[\frac{1}{2}\left(a_n+\mathrm{j}b_n\right)\mathrm{e}^{-\mathrm{j}n\omega_0 t}+\frac{1}{2}\left(a_n-\mathrm{j}b_n\right)\mathrm{e}^{\mathrm{j}n\omega_0 t}\right] \tag{78}$$

代入 $a_0$, $a_n$, $b_n$ 的积分计算式可知：

$$\frac{a_0}{2}=\frac{1}{T}\int_{-\frac{T}{2}}^{\frac{T}{2}}f(t)\mathrm{d}t \tag{79}$$

$$\begin{aligned}\frac{1}{2}\left(a_n+\mathrm{j}b_n\right)&=\frac{1}{T}\int_{-\frac{T}{2}}^{\frac{T}{2}}f(t)\left[\cos(n\omega_0 t)+\mathrm{j}\sin(n\omega_0 t)\right]\mathrm{d}t\\&=\frac{1}{T}\int_{-\frac{T}{2}}^{\frac{T}{2}}f(t)\mathrm{e}^{\mathrm{j}n\omega_0 t}\mathrm{d}t\end{aligned} \tag{80}$$

$$\begin{aligned}\frac{1}{2}\left(a_n-\mathrm{j}b_n\right)&=\frac{1}{T}\int_{-\frac{T}{2}}^{\frac{T}{2}}f(t)\left[\cos(n\omega_0 t)-\mathrm{j}\sin(n\omega_0 t)\right]\mathrm{d}t\\&=\frac{1}{T}\int_{-\frac{T}{2}}^{\frac{T}{2}}f(t)\mathrm{e}^{-\mathrm{j}n\omega_0 t}\mathrm{d}t\end{aligned} \tag{81}$$

上述三式之间存在一定规律，可以用一个以基频倍数 $n\omega_0$ 为自变量的复值离散函数 $F(n\omega_0)$ 来统一描述（为作区分，频域函数一般用相对应的大写符号表示），即

$$F(n\omega_0)=\frac{1}{T}\int_{-\frac{T}{2}}^{\frac{T}{2}}f(t)\mathrm{e}^{-\mathrm{j}n\omega_0 t}\mathrm{d}t=\begin{cases}\dfrac{a_0}{2}, & n=0\\ \dfrac{1}{2}\left(a_n+\mathrm{j}b_n\right), & n=-1,-2,-3,\cdots\\ \dfrac{1}{2}\left(a_n-\mathrm{j}b_n\right), & n=1,2,3,\cdots\end{cases} \tag{82}$$

代回公式(78)，则公式(75)得证。

定理3.4中的两个公式(75)和(76)以一种高度浓缩的形式提供了“连续时间域-离散频域”的双向变换，构成了一组 **Fourier 变换对**。$F(n\omega_0)$ 可以看作周期函数 $f(t)$ 经展开后的无穷级数系数，也可以看作时间域函数经过变换后的频域函数。

对于 Fourier 级数来说，其直观意义在于可以看到到一系列三角函数叠加后是如何逼近于原本的周期函数；而 Fourier 级数复指数展开更适合考察时间域函数和经变换后频域函数的相关性质。

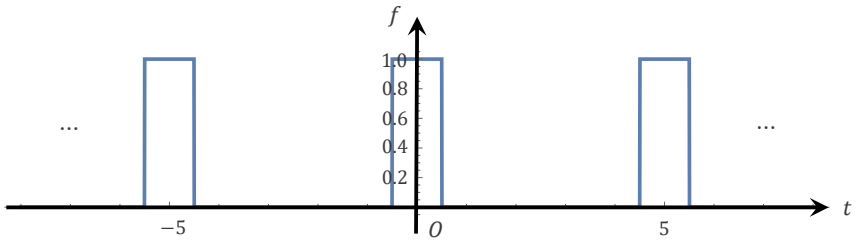
**例 3.3** 试求解周期 rect 函数的 Fourier 级数复指数展开。

周期 rect 函数即以 rect 函数为基础，以周期 $T$ 进行循环所构成的周期函数。首先其在区间 $[-\frac{T}{2},\frac{T}{2}]$ 内明显满足可积条件，通过公式(76)，再结合公式(77)可得：

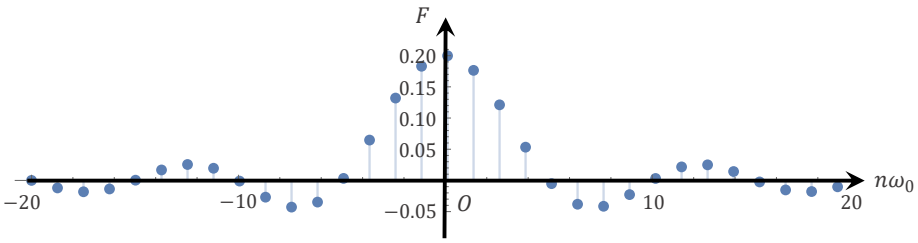
$$F(n\omega_0)=\frac{1}{T}\int_{-\frac{\tau}{2}}^{\frac{\tau}{2}}\frac{1}{\tau}\mathrm{e}^{-\mathrm{j}n\omega_0 t}\mathrm{d}t=\frac{1}{T\tau}\cdot\frac{\mathrm{j}}{n\omega_0}\,\mathrm{e}^{-\mathrm{j}n\omega_0 t}\Big|_{-\frac{\tau}{2}}^{\frac{\tau}{2}}=\frac{1}{T}\frac{\sin(n\omega_0\tau/2)}{n\omega_0\tau/2} \tag{83}$$

可以看到，周期 rect 函数所求得的频域函数 $F(n\omega_0)$ 具有某种 sinc 函数的形态，只不过是离散形式的。

下图27分别给出了周期 rect 函数（取 $\tau=1,T=5\tau$）和经变换后的频域函数图：




(a) 周期 rect 函数（$\tau=1,T=5\tau$）




(b) 频域函数 $F(n\omega_0)$

**图 27:** 周期 rect 函数和其频域函数图

通过定理3.4，有了形式更加简洁的 Fourier 级数复指数展开，则可以继续根据 Fourier 积分的思路，**将周期函数推广至非周期函数**。

**定理 3.5 (Fourier 积分复指数展开)** *在整个区间 $[-\infty,\infty]$ 内可积的函数 $f(t)$ 可以以连续频谱 $\omega$ 的形式进行如下复指数形式的展开：*

$$f(t)=\frac{1}{2\pi}\int_{-\infty}^{\infty}F(\omega)\mathrm{e}^{\mathrm{j}\omega t}\mathrm{d}\omega \tag{84}$$

*其中，*

$$F(\omega)=\int_{-\infty}^{\infty}f(t)\mathrm{e}^{-\mathrm{j}\omega t}\mathrm{d}t \tag{85}$$

**证明.** 需要说明的是，以上定理也可以通过 Fourier 积分的表达式(62)结合 Euler 公式进行推导证明，但计算过程会略显繁琐。

有了定理3.4，这里只需要效仿定理3.3中得到 Fourier 积分的推导思路，将无穷级数项求和将变为连续函数的积分，直接一步得到：

$$f(t)=\sum_{-\infty}^{\infty}\frac{\Delta\omega}{2\pi}\left[\int_{-\frac{T}{2}}^{\frac{T}{2}}f(t)\mathrm{e}^{-\mathrm{j}n\omega_0 t}\mathrm{d}t\right]\mathrm{e}^{\mathrm{j}n\omega_0 t}\xrightarrow{T\to\infty}\frac{1}{2\pi}\int_{-\infty}^{\infty}\underbrace{\left[\int_{-\infty}^{\infty}f(t)\mathrm{e}^{-\mathrm{j}\omega t}\right]}_{F(\omega)}\mathrm{e}^{\mathrm{j}\omega t}\mathrm{d}\omega \tag{86}$$

注 从以上证明过程可以看出，系数 $\frac{1}{2\pi}$ 可以自由选择放在 $f(t)$ 或 $F(\omega)$ 的表达式中。不同于 Fourier 级数与 Fourier 积分的形式，定理3.5将系数置于 $f(t)$ 中，只是约定俗成的一种写法。

定理3.5中的两个公式(84)和(85)同样提供了“时间域-频域”的双向变换，不过这次是通过非周期信号实现连续时间域到连续频域的转换，也可以反过来实现连续频域到连续时间域的转换。

由于 $\omega$ 为三角函数中的角频率，在实际使用中，常采用频率 $f=\frac{\omega}{2\pi}$ 进行替换。将该式代入公式(84)和(85)中，可最终得到如下通常意义下的 Fourier 变换，也被称为**连续时间 Fourier 变换**，简称 FT（Fourier Transform）。

**定义 3.1 (Fourier 变换)** 在 $(-\infty,\infty)$ 范围内满足 Dirichlet 条件的函数，存在如下 Fourier 变换表达式：

$$\begin{cases}F(f)=\mathcal{F}[f(t)]=\displaystyle\int_{-\infty}^{\infty}f(t)\mathrm{e}^{-\mathrm{j}2\pi ft}\mathrm{d}t\\ f(t)=\mathcal{F}^{-1}[F(f)]=\displaystyle\int_{-\infty}^{\infty}F(f)\mathrm{e}^{\mathrm{j}2\pi ft}\mathrm{d}f\end{cases} \tag{87}$$

其中，$f(t)$ 为函数在时间域上的表达式，$F(f)$ 则为在频域上的表达式；$\mathcal{F}[f(t)]$ 被称为 Fourier 正变换，$\mathcal{F}^{-1}[F(f)]$ 则为 Fourier 逆变换。

由于时间域上的函数最常见的即为**有限声音信号**，满足 $(-\infty,\infty)$ 范围内的可积条件，所以理论上可以通过上述 Fourier 变换将任意时域信号 $f(t)$ 转化为频域信号 $F(f)$，对音频信号的频谱进行分析。

进一步考察下此时 $F(f)$ 的物理含义。由于 $f(t)$ 可视作某种信号的“强度”，而根据 Fourier 正变换的表达式可知，$F(f)\mathrm{d}f$ 与 $f(t)$ 量纲相同，所以 $F(f)$ 可看作是信号 $f(t)$ 的**“频率密度函数”**。

**例 3.4** 试证明：rect 函数和 sinc 函数互为 Fourier 变换对。具体来讲，即满足

$$\mathrm{rect}(t)\xleftrightarrow{\mathrm{FT}}\mathrm{sinc}(\tau f),\quad \mathrm{sinc}(\tau t)\xleftrightarrow{\mathrm{FT}}\mathrm{rect}(f) \tag{88}$$

**证明.** 首先推导时域函数 $\mathrm{rect}(t)$ 的 Fourier 正变换：

$$F(f)=\mathcal{F}[f(t)]=\int_{-\frac{\tau}{2}}^{\frac{\tau}{2}}\frac{1}{\tau}\mathrm{e}^{-\mathrm{j}2\pi ft}\mathrm{d}t=\frac{\sin(\tau\pi f)}{\tau\pi f}=\mathrm{sinc}(\tau f) \tag{89}$$

根据本节 Fourier 变换的推导过程可知，求得了 Fourier 正变换即同时得到了 Fourier 逆变换。这里利用 $\mathrm{sinc}(\tau f)$ 的 Fourier 逆变换做一验证：

$$\begin{aligned} f(t)=\mathcal{F}^{-1}[F(f)]&=\int_{-\infty}^{\infty}\frac{\sin(\tau\pi f)}{\tau\pi f}\left[\cos\left(2\pi ft\right)+\mathrm{j}\sin\left(2\pi ft\right)\right]\mathrm{d}f \\ &=2\int_{0}^{\infty}\frac{\sin(\tau\pi f)\cos\left(2\pi ft\right)}{\tau\pi f}\mathrm{d}f \end{aligned} \tag{90}$$

上式最后一步的推导，用到了被积函数中奇函数与偶函数的积分特性。

若对公式(90)进行变量替换 $f=\frac{\omega}{2\pi}$，则可得到与公式(72)完全一样的 rect 函数 Fourier 积分表达式。

接下来推导时域函数 $\mathrm{sinc}(\tau t)$ 的 Fourier 正变换：

$$F(f)=\mathcal{F}[f(t)]=\int_{-\infty}^{\infty}\frac{\sin(\tau\pi t)}{\tau\pi t}\left[\cos\left(2\pi ft\right)-\mathrm{j}\sin\left(2\pi ft\right)\right]\mathrm{d}t \tag{91}$$

可以看到，虽然上式与公式(90)略有区别，但结合被积函数中相同的积分特性，还是会得到形式相同的积分结果（注意此时参数 $f$ 与 $t$ 进行了替换）。

频域函数 $\mathrm{rect}(f)$ 的 Fourier 逆变换证明从略，计算过程与公式(89)类似。

以上巧合，主要源于函数本身为偶函数，以及 $\mathrm{e}^{\pm\mathrm{j}2\pi ft}$ 特殊的积分特性。

进一步，考虑到**周期函数**其实并不符合定义3.1的 Fourier 变换条件（在 $(-\infty,\infty)$ 范围内不可积），尽管按照定理3.4的 Fourier 级数复指数展开可以对周期函数进行频域求解，但最好还是将其改写为类似定义3.1的统一变换形式。

**引理 3.6 (周期函数的 Fourier 变换)** *对于周期函数 $f(t)$，存在如下 Fourier 变换表达式：*

$$F(f)=\mathcal{F}[f(t)]=\sum_{n=-\infty}^{\infty}F(n)\delta(f-nf_0) \tag{92}$$

*其中，*

$$F(n)=\frac{1}{T}\int_{-\frac{T}{2}}^{\frac{T}{2}}f(t)\mathrm{e}^{-\mathrm{j}2\pi nf_0t}\mathrm{d}t \tag{93}$$

*$\delta$ 函数为 Dirac 函数，$f_0$ 为周期函数的频率，满足 $f_0=\frac{1}{T}$，$n$ 的取值为 $n=0,\pm1,\pm2,\cdots$。*

**证明.** 结合 Dirac 函数的性质，计算 $\delta(t)$ 的 Fourier 正/逆变换（满足可积条件），可以得到如下特殊的 Fourier 变换对：

$$\delta(t) \xleftrightarrow{\text{FT}} 1, \quad 1 \xleftrightarrow{\text{FT}} \delta(f) \tag{94}$$

同时还可从理论上给出 Dirac 函数一种特殊的积分定义（$\tau$ 为任意积分参数）：

$$\delta(x) = \int_{-\infty}^{\infty} \mathrm{e}^{\pm \mathrm{j}2\pi\tau x} \mathrm{d}\tau \tag{95}$$

根据定理3.4的 Fourier 级数复指数展开，有

$$f(t) = \sum_{n=-\infty}^{\infty} F(n\omega_0)\mathrm{e}^{\mathrm{j}n\omega_0 t} = \sum_{n=-\infty}^{\infty} F(n)\mathrm{e}^{\mathrm{j}2\pi n f_0 t} \tag{96}$$

其中，

$$F(n) = F(n\omega_0) = \frac{1}{T}\int_{-\frac{T}{2}}^{\frac{T}{2}} f(t)\mathrm{e}^{-\mathrm{j}n\omega_0 t} = \frac{1}{T}\int_{-\frac{T}{2}}^{\frac{T}{2}} f(t)\mathrm{e}^{-\mathrm{j}2\pi n f_0 t}\mathrm{d}t \tag{97}$$

此时，直接对 $f(t)$ 进行 Fourier 变换，结合公式(95)的形式，可将本不具备可积条件的函数变换为利用 Dirac 函数描述的形式：

$$\begin{aligned} F(f) = \mathcal{F}[f(t)] &= \int_{-\infty}^{\infty}\left[\sum_{n=-\infty}^{\infty} F(n)\mathrm{e}^{\mathrm{j}2\pi n f_0 t}\right]\mathrm{e}^{-\mathrm{j}2\pi f t}\mathrm{d}t \\ &= \sum_{n=-\infty}^{\infty} F(n)\left[\int_{-\infty}^{\infty}\mathrm{e}^{-\mathrm{j}2\pi(f-nf_0)t}\mathrm{d}t\right] = \sum_{n=-\infty}^{\infty} F(n)\delta(f-nf_0) \end{aligned} \tag{98}$$

以上推导还可进一步延伸归纳出如下的 Fourier 变换对：

$$\mathrm{e}^{\mathrm{j}2\pi f_0 t} \xleftrightarrow{\text{FT}} \delta(f-f_0), \quad \delta(t-t_0) \xleftrightarrow{\text{FT}} \mathrm{e}^{-\mathrm{j}2\pi f_0 t} \tag{99}$$

**例 3.5** 试求解周期 rect 函数的 Fourier 变换表达式，并分析其与例题3.3的关系。

根据引理3.6，先计算 $F(n)$，形式与公式(83)相同（只需将 $\omega_0$ 转为 $f_0$），最终有

$$F(f) = \mathcal{F}[f(t)] = \frac{1}{T}\sum_{n=-\infty}^{\infty}\frac{\sin(\pi n f_0\tau)}{\pi n f_0\tau}\cdot\delta(f-nf_0) = f_0\,\mathrm{sinc}(\tau f)\cdot\sum_{n=-\infty}^{\infty}\delta(f-nf_0) \tag{100}$$

即 $F(f)$ 可表示为经过一串周期 Dirac 函数“抽样”后的某种 sinc 函数。对 $F(f)$ 的每一项进行积分，刚好等于例题3.3中 $F(n\omega_0)$ 的对应项，符合其“频率密度”的概念。

## 3.3 离散时间 Fourier 变换（DTFT）

**定理 3.7 (离散时间 Fourier 级数)** 周期离散函数 $x[n]$ 可以进行如下形式的展开：

$$x[n]=\sum_{k=0}^{N-1}X[k\Omega_0]\mathrm{e}^{\mathrm{j}k\Omega_0 n} \tag{101}$$

其中，

$$X[k\Omega_0]=\frac{1}{N}\sum_{k=0}^{N-1}x[n]\mathrm{e}^{-\mathrm{j}k\Omega_0 n} \tag{102}$$

式中，$N$ 为函数 $x[n]$ 的周期；$\Omega_0$ 为角频率，满足 $\Omega_0=\frac{2\pi}{N}$；$n$ 的取值为 $n=1,2,3,\cdots$。

**证明.**（待补全!!）对连续时间进行抽样，

$$t=nT_\mathrm{s},\quad T=NT_\mathrm{s},\quad \omega_0 t=\frac{2\pi}{T}nT_\mathrm{s}=\frac{2\pi}{N}n=\Omega_0 n \tag{103}$$

**定理 3.8 (离散时间 Fourier 积分)** 非周期离散函数 $x[n]$ 可以进行如下形式的展开：

$$x[n]=\frac{1}{2\pi}\int_{-\pi}^{\pi}X[\Omega]\mathrm{e}^{\mathrm{j}\Omega n}\mathrm{d}\Omega \tag{104}$$

其中，

$$X(\Omega)=\sum_{n=-\infty}^{\infty}x[n]\mathrm{e}^{-\mathrm{j}\Omega n} \tag{105}$$

**证明.**（待补全!!）当 $N\to\infty$ 时，$k\Omega_0$ 趋向于连续频谱 $\Omega$，$\Delta\Omega=(k+1)\Omega_0-k\Omega_0=\Omega_0$ 趋向于微元 $\mathrm{d}\Omega$，且有

$$\sum_{k=1}^{\infty}\cdots\Delta\Omega\xrightarrow{N\to\infty}\int_0^{\infty}\cdots\mathrm{d}\Omega \tag{106}$$

直接一步得到：

$$x[n]=\sum_{k=0}^{N-1}\frac{\Delta\Omega}{2\pi}\left[\frac{1}{N}\sum_{k=0}^{N-1}x[n]\mathrm{e}^{-\mathrm{j}k\Omega_0 n}\right]\mathrm{e}^{\mathrm{j}k\Omega_0 n}\xrightarrow{N\to\infty}\frac{1}{2\pi}\int_{-\pi}^{\pi}\underbrace{\left[\sum_{n=-\infty}^{\infty}x[n]\mathrm{e}^{-\mathrm{j}\Omega n}\right]}_{X(\Omega)}\mathrm{e}^{\mathrm{j}\Omega n}\mathrm{d}\Omega \tag{107}$$

**定义 3.2 (离散时间 Fourier 变换)** 存在如下 Fourier 变换表达式：

$$\begin{cases} X(F) = \mathcal{F}[x[n]] = \sum\limits_{n=-\infty}^{\infty} x[n]\mathrm{e}^{-\mathrm{j}2\pi Fn} \\ x[n] = \mathcal{F}^{-1}[X(F)] = \int_{-\pi}^{\pi} X(f)\mathrm{e}^{\mathrm{j}2\pi Fn}\mathrm{d}F \end{cases} \tag{108}$$

其中，$x[n]$ 为函数在时间域上的表达式，$X(F)$ 则为在频域上的表达式；$\mathcal{F}[x[n]]$ 被称为 Fourier 正变换，$\mathcal{F}^{-1}[X(F)]$ 则为 Fourier 逆变换。

**证明.**（待补全!!）

**引理 3.9 (周期函数的离散时间 Fourier 变换)** 对于周期函数 $x[n]$，存在如下 Fourier 变换表达式：

$$X(F) = \mathcal{F}[x[n]] = \sum_{k=-\infty}^{\infty} X[k]\delta(F - kF_0) \tag{109}$$

其中，

$$X[k] = \frac{1}{N}\sum_{k=0}^{N-1} x[n]\mathrm{e}^{-\mathrm{j}k\Omega_0 n} \tag{110}$$

$\delta$ 函数为 Dirac 函数，$F_0$ 为周期函数的频率，满足 $F_0 = \frac{1}{N}$，$k$ 的取值为 $n = 0, \pm 1, \pm 2, \cdots$。

**证明.**（待补全!!）

## 3.4 离散 Fourier 变换（DFT）

**定义 3.3 (离散 Fourier 变换)** 存在如下 Fourier 变换表达式：

$$\begin{cases} X_k = \mathcal{F}[x_n] = \sum\limits_{n=0}^{N-1} x_n\mathrm{e}^{-\mathrm{j}k\Omega_0 n} \\ x_n = \mathcal{F}^{-1}[X_k] = \dfrac{1}{N}\sum\limits_{n=0}^{N-1} X_k\mathrm{e}^{\mathrm{j}k\Omega_0 n} \end{cases} \tag{111}$$

其中，$x[n]$ 为函数在时间域上的表达式，$X(F)$ 则为在频域上的表达式；$\mathcal{F}[x[n]]$ 被称为 Fourier 正变换，$\mathcal{F}^{-1}[X(F)]$ 则为 Fourier 逆变换。

**证明.**（待补全!!）

## 3.5 多维 Fourier 变换

**定义 3.4 (多维 Fourier 变换)** 对于满足条件的定义在 $\mathbb{R}^n$ 上的函数 $f(x)$，其 Fourier 变换被定义为

$$F(\boldsymbol{\omega}) = \mathcal{F}[f(\boldsymbol{x})] = \int_{\mathbb{R}^n} f(\boldsymbol{x}) \mathrm{e}^{-\mathrm{j}2\pi(\boldsymbol{\omega}\cdot\boldsymbol{x})} \mathrm{d}\boldsymbol{x} \tag{112}$$

其中 j 是虚数单位；$\boldsymbol{\omega}\cdot\boldsymbol{x}$ 为向量内积，满足 $\boldsymbol{\omega}\cdot\boldsymbol{x} = \omega_1 x_1 + \omega_2 x_2 + \cdots + \omega_n x_n$。

特别地，针对数字图像，有如下二维 Fourier 变换的形式：

**定义 3.5 (二维 Fourier 变换)** 二维 Fourier 变换被定义为

$$\begin{cases} F(\mu,\nu) = \mathcal{F}[f(x,y)] = \displaystyle\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x,y)\mathrm{e}^{-\mathrm{j}2\pi(\mu x+\nu y)}\mathrm{d}x\mathrm{d}y \\ f(x,y) = \mathcal{F}^{-1}[F(\mu,\nu)] = \displaystyle\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} F(\mu,\nu)\mathrm{e}^{\mathrm{j}2\pi(\mu x+\nu y)}\mathrm{d}\mu\mathrm{d}\nu \end{cases} \tag{113}$$

其中相对应于多维 Fourier 变换的形式，有 $\boldsymbol{x} = (x, y)$, $\boldsymbol{\omega} = (\mu, \nu)$。

进一步离散化，可得：

**定义 3.6 (二维离散 Fourier 变换)** 存在如下 Fourier 变换表达式：

$$\begin{cases} F(\mu,\nu) = \mathcal{F}[f(x,y)] = \displaystyle\sum_{m=0}^{M-1}\sum_{n=0}^{N-1} f(x,y)\mathrm{e}^{-\mathrm{j}2\pi\left(\frac{\mu x}{M}+\frac{\nu y}{N}\right)} \\ f(x,y) = \mathcal{F}^{-1}[F(\mu,\nu)] = \dfrac{1}{MN}\displaystyle\sum_{m=0}^{M-1}\sum_{n=0}^{N-1} f(\mu,\nu)\mathrm{e}^{\mathrm{j}2\pi\left(\frac{\mu x}{M}+\frac{\nu y}{N}\right)} \end{cases} \tag{114}$$

# 4 偏微分方程图像处理

通过前面的介绍，可以发觉传统的图像处理模型具有各向同性，在减弱噪声的同时也会模糊边界。而偏微分方程的优势就在于可以实现各向异性扩散。该领域数学概念较多，专业术语难懂，所以首先对一些接下来要用到的基本概念和算法做一个基本阐述。

## 4.1 微分几何基础

偏微分方程图像处理与微分几何关系密切，具体的原因将在后续进行阐述，这里首先把将会用到的微分几何基础知识做一简单罗列。

### 4.1.1 预备知识

在三维欧式空间中，坐标系的构建主要用于对点进行定位。以坐标 $(\boldsymbol{i},\boldsymbol{j},\boldsymbol{k})$ 进行表示的方式也可以看作是从坐标系原点向该点指向的一个向量。若坐标维度之间并无牵连，则该点为任意自由移动的点；若坐标维度之间有关系，则该点的运动将受到束缚，其活动轨迹将引申出曲线与曲面的概念：当该点仅依赖于一个参数（例如弧长 $s$）时，其轨迹描述为**曲线**；当该点依赖于两个参数时，则该轨迹将描绘出**曲面**。

**例 4.1** 以下形式代表曲线：

$$(1,1,t),(x(s),y(s),z(s))$$

以下形式代表曲面：

$$(x,y,z(x,y)),(x(u,v),y(u,v),z(u,v))$$

### 4.1.2 参数曲线分析

**定义 4.1 (参数化可微曲线)** 参数化可微曲线为一个区间 $I=(a,b)\in\mathbb{R}$ 向三维空间 $\mathbb{R}^3$ 的连续可微映射 $\boldsymbol{r}:I\to\mathbb{R}^3$

假设映射 $\boldsymbol{r}$ 是把任意参数 $t\in I$ 映射到 $\boldsymbol{r}(t)=(x(t),y(t),z(t))\in\mathbb{R}^3$，则这里的可微表示函数 $x(t),y(t),z(t)$ 都是可微的。其中，变量 $t$ 称为该曲线的参数。

**定义 4.2 (曲线的弧长)** 给定 $t\in I$，从点 $t_0$ 开始计算的正则参数化曲线 $\boldsymbol{r}:I\to\mathbb{R}^3$ 的弧长定义为

$$s(t)=\int_{t_0}^{t}\left|\frac{\mathrm{d}}{\mathrm{d}t}\boldsymbol{r}(t)\right|\mathrm{d}t \tag{115}$$

其中，

$$\left|\frac{\mathrm{d}}{\mathrm{d}t}\boldsymbol{r}(t)\right| = |\dot{\boldsymbol{r}}(t)| = \sqrt{(\dot{x}(t))^2 + (\dot{y}(t))^2 + (\dot{z}(t))^2} \tag{116}$$

以上定义中的**正则**表示：

1. $\boldsymbol{r}(t)$ 至少是自变量 $t$ 的三次以上连续可微的向量函数（因为曲线的几何不变量涉及 $\boldsymbol{r}(t)$ 的三次倒数）；
2. 处处是正则点（曲线在该点的切线是完全确定的），即对于任意的 $t$ 有 $\dot{\boldsymbol{r}}(t) \neq 0$（曲线的切向量）。

由于 $\dot{\boldsymbol{r}}(t) \neq 0$，则弧长 $s$ 也可作为参数 $t$ 的可微函数，且

$$\frac{\mathrm{d}s}{\mathrm{d}t} = |\dot{\boldsymbol{r}}(t)| \tag{117}$$

由于

$$\boldsymbol{r}'(s) = \frac{\mathrm{d}\boldsymbol{r}(t)}{\mathrm{d}t} \cdot \frac{\mathrm{d}t}{\mathrm{d}s} \tag{118}$$

而

$$|\boldsymbol{r}'(s)| = \frac{\mathrm{d}t}{\mathrm{d}s} \cdot \left|\frac{\mathrm{d}\boldsymbol{r}(t)}{\mathrm{d}t}\right| = \frac{1}{|\dot{\boldsymbol{r}}(t)|} \cdot \left|\frac{\mathrm{d}\boldsymbol{r}(t)}{\mathrm{d}t}\right| = 1 \tag{119}$$

说明切向量 $\boldsymbol{r}'(s)$ 为**单位长度**。

注 在这里为了区分，使用 $\dot{\boldsymbol{r}}(t)$ 表示 $\frac{\mathrm{d}\boldsymbol{r}(t)}{\mathrm{d}t}$，而 $\boldsymbol{r}'(t)$ 表示 $\frac{\mathrm{d}\boldsymbol{r}(s)}{\mathrm{d}s}$。

另外由于

$$\left(\boldsymbol{r}'(s) \cdot \boldsymbol{r}'(s)\right)' = 0 = \boldsymbol{r}'(s) \cdot \boldsymbol{r}''(s) \tag{120}$$

说明 $\boldsymbol{r}''(s) \perp \boldsymbol{r}'(s)$。

将弧长 $s$ 作为参数曲线的唯一参数，$\boldsymbol{r}'(s)$ 代表切向量，可用 $\boldsymbol{t}(s)$ 表示，而向量 $\boldsymbol{r}''(s)$ 与其垂直，所以可知 $\boldsymbol{r}''(s)$ 与**曲线的法向量方向相同**。

**定理 4.1 (刻画曲线弯曲程度的方式)** 设 $\boldsymbol{t}(s)$ 是曲线 $\boldsymbol{r}(s)$ 的单位切向量，$s$ 是弧长参数，用 $\Delta\theta$ 表示切向量 $\boldsymbol{r}(s+\Delta s)$ 与 $\boldsymbol{r}(s)$ 之间的夹角，则

$$\lim_{\Delta s \to 0}\left|\frac{\Delta\theta}{\Delta s}\right| = \left|\frac{\mathrm{d}\boldsymbol{t}}{\mathrm{d}s}\right| \tag{121}$$

**证明.** 把曲线上的单位切向量 $\boldsymbol{t}(s)$ 平行移动，使其起点咋一起，于是切向量 $\boldsymbol{r}(s+\Delta s)$ 与 $\boldsymbol{r}(s)$ 之间的夹角 $\Delta\theta$ 是在单位球面上从 $\boldsymbol{r}(s+\Delta s)$ 到 $\boldsymbol{r}(s)$ 的大圆弧的弧长，而 $|\boldsymbol{r}(s+\Delta s)-\boldsymbol{r}(s)|$ 正好是该角所对应的弦长，所以

$$\begin{aligned}\left|\frac{\mathrm{d}\boldsymbol{t}}{\mathrm{d}s}\right| &= \lim_{\Delta s\to 0}\frac{|t(s+\Delta s)-t(s)|}{|\Delta s|}\\ &= \lim_{\Delta s\to 0}\frac{2\left|\sin\frac{\Delta\theta}{2}\right|}{|\Delta s|}\\ &= \lim_{\Delta s\to 0}\left|\frac{\Delta\theta}{\Delta s}\right|\end{aligned} \tag{122}$$

证毕。

上述定理正是刻画曲线弯曲程度的方式，根据其描述，可以引出如下**曲率**的定义。

**定义 4.3 (曲线的曲率)** *设曲线的方程是 $\boldsymbol{r}(s)$，其中 $s$ 是曲线的弧长参数，令*

$$\kappa(s)=\left|\frac{\mathrm{d}\boldsymbol{t}}{\mathrm{d}s}\right|=\left|\boldsymbol{r}''(s)\right| \tag{123}$$

*称 $\kappa(s)$ 为曲线 $\boldsymbol{r}(s)$ 在 $s$ 处的曲率。*

再结合上述分析，令曲线的单位法向量为 $\boldsymbol{n}(s)$，可得出，

$$\boldsymbol{n}(s)=\frac{\boldsymbol{r}''(s)}{|\boldsymbol{r}''(s)|}=\frac{1}{\kappa(s)}\cdot\boldsymbol{t}'(s) \tag{124}$$

更进一步，可推导出著名的 ***Frenet-Serret* 标架运动方程**，令 $\boldsymbol{b}(s)$ 为曲线的副法向量，定义为 $\boldsymbol{b}(s)=\boldsymbol{t}(s)\times\boldsymbol{n}(s)$，则有：

$$\begin{aligned}\boldsymbol{t}'(s)&=\kappa(s)\cdot\boldsymbol{n}(s)\\ \boldsymbol{n}'(s)&=-\kappa(s)\boldsymbol{t}(s)-\tau(s)\cdot\boldsymbol{b}(s)\\ \boldsymbol{b}'(s)&=\tau(s)\cdot\boldsymbol{n}(s)\end{aligned} \tag{125}$$

该证明从略，因为比较好证，但书写较为繁琐。

上式可用矩阵形式重新书写：

$$\frac{\mathrm{d}}{\mathrm{d}s}\begin{bmatrix}t\\ n\\ b\end{bmatrix}=\begin{bmatrix}0 & \kappa & 0\\ -\kappa & 0 & -\tau\\ 0 & \tau & 0\end{bmatrix}\begin{bmatrix}t\\ n\\ b\end{bmatrix} \tag{126}$$

特别地，针对笛卡尔直角坐标系下的平面曲线而言，$\boldsymbol{r}(s)$ 可表示为 $\boldsymbol{r}(s)=(x(s),y(s))$，其切向量为 $\boldsymbol{t}(s)=(x'(s),y'(s))$，法向量很明显可表示为 $\boldsymbol{n}(s)=(-y'(s),x'(s))$，则其曲率可

通过如下方式进行计算：

$$\begin{aligned}\kappa_r(s) &= \boldsymbol{t}'(s)\cdot\boldsymbol{n}(s) = -x''(s)y'(s)+y''(s)x'(s)\\ &= \begin{vmatrix} x'(s) & y'(s) \\ x''(s) & y''(s) \end{vmatrix}\end{aligned} \tag{127}$$

注意，这里求得的只是**相对曲率**，用 $\kappa_r(s)$ 表示。平面曲线中，曲率有正负两种，分别对应曲线朝 $\boldsymbol{n}(s)$ 所指方向弯曲，和 $\boldsymbol{n}(s)$ 所指相反方向弯曲（曲线的法向量按公式推导可以看出，都是**朝着弯曲方向的反向**）。

若平面曲线 $\boldsymbol{r}=\boldsymbol{r}(t)$ 的参数方程是 $\boldsymbol{t}(t)=(x(t),y(t))$。其中 $t$ 未必是弧长参数，则曲线的弧长为，

$$\mathrm{d}s = \left|\boldsymbol{r}'(t)\right|\mathrm{d}t = \sqrt{(x')^2+\left(y'\right)^2}\mathrm{d}t \tag{128}$$

其单位切向量为，

$$\boldsymbol{t}(t) = \frac{r'(t)}{|r'(t)|} = \left(\frac{x'}{\sqrt{(x')^2+\left(y'\right)^2}},\frac{y'}{\sqrt{(x')^2+\left(y'\right)^2}}\right) \tag{129}$$

法向量为，

$$\boldsymbol{n}(t) = \left(-\frac{y'}{\sqrt{(x')^2+\left(y'\right)^2}},\frac{x'}{\sqrt{(x')^2+\left(y'\right)^2}}\right) \tag{130}$$

则其相对曲率可求得：

$$\begin{aligned}\kappa_r(t) &= \frac{\mathrm{d}\boldsymbol{t}(t)}{\mathrm{d}s}\cdot\boldsymbol{n}(t) = \frac{\mathrm{d}t}{\mathrm{d}s}\cdot\boldsymbol{t}'(t)\cdot\boldsymbol{n}(t)\\ &= \frac{x'(t)y''(t)-x''(t)y'(t)}{\sqrt{\left((x')^2+\left(y'\right)^2\right)^3}}\end{aligned} \tag{131}$$

上式就是一般比较熟知的**平面曲线曲率计算表达式**。

### 4.1.3 参数曲面分析

参数曲面相较于曲线，其分析难度直线上升。所以这里仅列出必要的概念，以及简要说明。

**定义 4.4 (正则曲面)** 一个集合 $\Sigma \subset \mathbb{R}^3$ 为正则曲面，当且仅当存在一个映射，满足以下形式：

$$\boldsymbol{\Sigma}(\boldsymbol{x}_\Sigma) : \mathbb{R}^2 \ni \boldsymbol{x}_\Sigma = \begin{bmatrix} u \\ v \end{bmatrix} \mapsto \boldsymbol{\Sigma}(\boldsymbol{x}_\Sigma) = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \in \mathbb{R}^3 \tag{132}$$

其中，矢量形式 $\boldsymbol{\Sigma}(\boldsymbol{x}_\Sigma) = \boldsymbol{r}(u,v) = (x(u,v), y(u,v), z(u,v))$ 都是 3 次以上连续可微的，处处是正则点的参数曲面。

可以明显看出，曲面的参数化模型为**向量值映照**（$\mathbb{R}^2 \mapsto \mathbb{R}^3$）。

参数曲面有一种特殊形式如下，

$$z = z(u,v) = z(u(x,y), v(x,y)) \tag{133}$$

该形式下曲面的参数方程可表示为，

$$\boldsymbol{r} = (x, y, z(x,y)) \tag{134}$$

这种曲面形式称为 ***Monge* 形式**，可以明显看出，用 *Monge* 形式给出的曲面都是正则的。

对于曲面来说，在其上任意一点 $\boldsymbol{r}(u,v)$ 的任意一个切向量为，

$$\mathrm{r}(u,v) = \boldsymbol{r}_u(u,v)\mathrm{d}u + \boldsymbol{r}_v(u,v)\mathrm{d}v \tag{135}$$

其中，$\boldsymbol{r}_u$ 和 $\boldsymbol{r}_v$ 分别是曲面上沿着参数网格 $(u,v)$ 的切向量。

**定义 4.5 (曲面的第一基本形式)** 曲面的第一基本形式为如下表达式：

$$\begin{aligned} \boldsymbol{I} = |\mathrm{d}\boldsymbol{r}|^2 &= \mathrm{d}\boldsymbol{r} \cdot \mathrm{d}\boldsymbol{r} = (\boldsymbol{r}_u\,\mathrm{d}u + \boldsymbol{r}_v\,\mathrm{d}v) \cdot (\boldsymbol{r}_u\,\mathrm{d}u + \boldsymbol{r}_v\,\mathrm{d}v) \\ &= \boldsymbol{r}_u \cdot \boldsymbol{r}_u(\mathrm{d}u)^2 + 2\boldsymbol{r}_u \cdot \boldsymbol{r}_v\,\mathrm{d}u\,\mathrm{d}v + \boldsymbol{r}_v \cdot \boldsymbol{r}_v(\mathrm{d}v)^2 \end{aligned} \tag{136}$$

如果令

$$E = \boldsymbol{r}_u \cdot \boldsymbol{r}_u, \quad F = \boldsymbol{r}_u \cdot \boldsymbol{r}_v, \quad G = \boldsymbol{r}_v \cdot \boldsymbol{r}_v \tag{137}$$

则有

$$\boldsymbol{I} = E(\mathrm{d}u)^2 + 2F\mathrm{d}u\mathrm{d}v + G(\mathrm{d}v)^2 \tag{138}$$

其中，$E, F, G$ 称为曲面的第一类基本量。

曲面的第一基本形式还可以写成矩阵形式：

$$\boldsymbol{I} = (\mathrm{d}u,\ \mathrm{d}v)\begin{pmatrix} E & F \\ F & G \end{pmatrix}\begin{pmatrix} \mathrm{d}u \\ \mathrm{d}v \end{pmatrix} \tag{139}$$

可以看到，曲面的第一基本量被写成了对称矩阵的形式。

**第一基本形式完全描述了曲面的度量性质**，并不关心这个曲面是如何嵌入在 $\mathbb{R}^3$，从而使得可以计算曲面上曲线的长度与区域的面积。

**例 4.2** 对于曲面，其面积近似等于在点 $\boldsymbol{r}(u_0, v_0)$ 处的切平面上由向量 $(\Delta u)\boldsymbol{r_u}$ 和 $(\Delta v)\boldsymbol{r_v}$ 所张成的平行四边形的面积：

$$\begin{aligned} |(r_u\Delta u)\times(r_v\Delta v)| &= |\boldsymbol{r}_u\times\boldsymbol{r}_v|\Delta u\Delta v \\ &= |r_u\|\boldsymbol{r}_v|\sin\angle(\boldsymbol{r}_u,\boldsymbol{r}_v)\,\Delta u\Delta v \\ &= \sqrt{EG-F^2}\Delta u\Delta v \end{aligned} \tag{140}$$

则可以定义曲面的面积元 $\mathrm{d}\sigma$：

$$\mathrm{d}\sigma = \sqrt{EG-F^2}\mathrm{d}u\mathrm{d}v \tag{141}$$

对其积分即可求得曲面在某区域内的面积：

$$A = \iint_D \sqrt{EG-F^2}\,\mathrm{d}u\,\mathrm{d}v \tag{142}$$

上述求面积元的方式也可用于求解积分变元的重积分，面积元中绝对值内的量刚好可以写成所谓的 *Jacob* 矩阵。

**定义 4.6 (曲面的第二基本形式)** 令 $\boldsymbol{N}(u,v)$ 为曲面在某一点的法向量，则曲面的第二基本形式为如下表达式：

$$\begin{aligned} \boldsymbol{II} &= 2\mathrm{d}\boldsymbol{r}\cdot\boldsymbol{N} = 2(\boldsymbol{r}(u+\mathrm{d}u, v+\mathrm{d}v)-\boldsymbol{r}(u,v))\cdot\boldsymbol{N} \\ &= \boldsymbol{r}_{uu}\cdot\boldsymbol{N}(\mathrm{d}u)^2 + 2\boldsymbol{r}_{uv}\cdot\boldsymbol{N}\mathrm{d}u\mathrm{d}v + \boldsymbol{r}_{vv}\cdot\boldsymbol{N}(\mathrm{d}v)^2 \end{aligned} \tag{143}$$

如果令

$$L = \boldsymbol{r}_{uu}\cdot\boldsymbol{N},\quad M = \boldsymbol{r}_{uv}\cdot\boldsymbol{N},\quad N = \boldsymbol{r}_{vv}\cdot\boldsymbol{N} \tag{144}$$

则有

$$\boldsymbol{II} = L(\mathrm{d}u)^2 + 2M\mathrm{d}u\mathrm{d}v + N(\mathrm{d}v)^2 \tag{145}$$

其中，$L, M, N$ 称为曲面的第二类基本量。

所以曲面的第二基本形式也可以写成类似第一基本形式的矩阵形式。

可以看到，第二基本形式关心的是**曲面在某处的弯曲程度**，用以衡量曲面上邻近的点与切平面的距离。

由于 $\boldsymbol{r}_u \cdot \boldsymbol{N} = 0, \boldsymbol{r}_v \cdot \boldsymbol{N} = 0$，

$$\begin{aligned} \boldsymbol{r}_{uu} \cdot \boldsymbol{N} + \boldsymbol{r}_u \cdot \boldsymbol{N}_u = 0 \quad \boldsymbol{r}_{uv} \cdot \boldsymbol{N} + \boldsymbol{r}_u \cdot \boldsymbol{N}_v = 0 \\ \boldsymbol{r}_{vu} \cdot \boldsymbol{N} + \boldsymbol{r}_v \cdot \boldsymbol{N}_u = 0 \quad \boldsymbol{r}_{vv} \cdot \boldsymbol{N} + \boldsymbol{r}_v \cdot \boldsymbol{N}_v = 0 \end{aligned} \tag{146}$$

所以第二类基本量还可以定义为：

$$L = -\boldsymbol{r}_u \cdot \boldsymbol{N}_u, \quad M = -\boldsymbol{r}_u \cdot \boldsymbol{N}_v = -\boldsymbol{r}_v \cdot \boldsymbol{N}_u, \quad N = -\boldsymbol{r}_v \cdot \boldsymbol{N}_v \tag{147}$$

所以曲面的第二基本形式也可另外表示为：

$$\boldsymbol{II} = L(\mathrm{d}u)^2 + 2M\mathrm{d}u\,\mathrm{d}v + N(\mathrm{d}v)^2 = -\mathrm{d}\boldsymbol{r} \cdot \mathrm{d}\boldsymbol{N} = -(\mathrm{d}\boldsymbol{N}, \mathrm{d}\boldsymbol{r}) \tag{148}$$

### 4.1.4 法曲率、主曲率、高斯曲率与平均曲率

参数曲面不同于曲线，其曲率只有一种度量方式。在曲面中，任意一点都会引申出多种曲率（假想通过该点沿任意方向一刀切下去，都会产生不同的曲线切面）。其中，最能够度量曲面性质的有**法曲率**、**主曲率**、**高斯曲率**与**平均曲率**。下面分别进行介绍。

考虑参数曲面上的一根曲线，如下图：

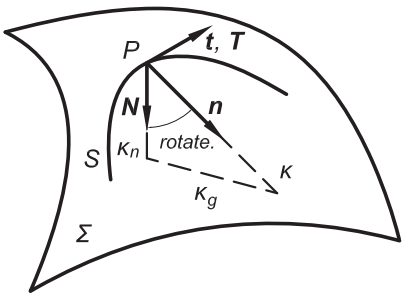



图 28: 参数曲面上的曲线示意图

可以明显看到，曲面在 $P$ 点的法向量 $\boldsymbol{N}$（一般来说，曲面法向量指向曲面弯曲的反向，这里取同向仅做示意）与嵌入该曲面的曲线法向量 $\boldsymbol{n}$，方向是不同的，存在一个旋转关系。

这个很好理解，假设一刀沿着曲面法向量 $\boldsymbol{N}$ 将曲面切开，会有无数种选择。再假定该切割平面与曲线 $S$ 相切，则只会有一种可能。切下去后会留下一条曲线痕迹，而该曲线与曲线 $S$ 并不相同，原因在于曲线 $S$ 还会有其自身的弯曲和扭转性质。

而切下去的这条曲线，其曲率就称为曲面的**法曲率**。所以法曲率在曲面上要给定出切割的具体方向，才会产生固定的数值，如下图所示：

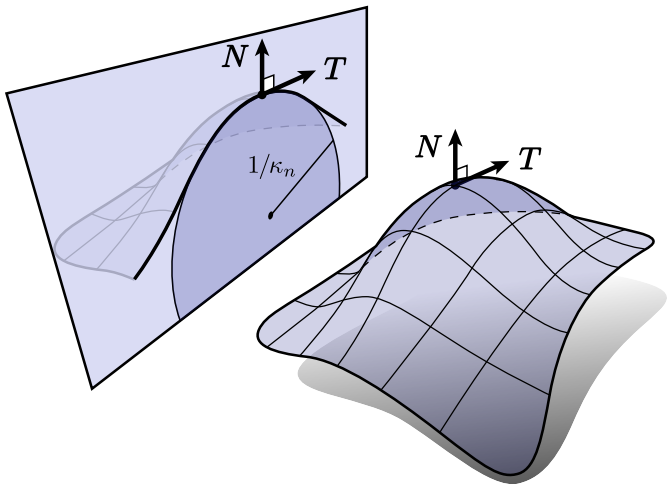



图 29: 法曲率几何意义示意图

法曲率的计算如下：

假设该条曲线 $S$ 的参数方程为

$$\boldsymbol{r} = \boldsymbol{r}(u(s), v(s)) \tag{149}$$

则曲线 $S$ 的切向量 $\boldsymbol{t}$ 可用曲线的曲率 $\kappa$ 和法向量 $\boldsymbol{n}$ 来表示：

$$\dot{\boldsymbol{t}} = \kappa \cdot \boldsymbol{n} \tag{150}$$

与此同时，切割平面切下来的曲线也存在类似关系，如下：

$$\dot{\boldsymbol{T}} = \kappa_n \cdot \boldsymbol{N} \tag{151}$$

这里为突出曲线和曲面的区别，特地用大写进行区分。上式里的 $\kappa_n$ 即为法曲率，两边同时乘以曲面的法向量 $\boldsymbol{N}$，即可求得最后的法曲率值：

$$\kappa_n = \frac{L\mathrm{d}u^2 + 2M\mathrm{d}u\mathrm{d}v + N\mathrm{d}v^2}{E\mathrm{d}u^2 + 2F\mathrm{d}u\mathrm{d}v + G\mathrm{d}v^2} = \frac{\boldsymbol{II}}{\boldsymbol{I}} \tag{152}$$

除了在曲面法向量的部分，当然还有剩下的部分，而这个部分当然就应该在曲面的切平面上，因为它处于与法向量垂直的平面上，一般称之为**测地曲率**。

首先有如下结论：

$$\kappa^2 = \kappa_g^2 + \kappa_n^2 \tag{153}$$

测地曲率也可以直接写出矢量计算公式：

$$\kappa_g = \dot{\boldsymbol{T}} \cdot (\boldsymbol{N} \times \boldsymbol{T}) \tag{154}$$

另外，法曲率和测地曲率有一些其他表达式，例如计算法曲率的 ***Euler* 公式**，和计算测地曲率的 ***Liouville* 公式**，将在介绍完主曲率后进行介绍。

简单总结下法曲率和测地曲率的几何意义：

1. 曲面上的曲线有一个曲率向量 $\dot{\boldsymbol{T}}$。这个向量往曲面的法线做投影，得到的投影向量就是法曲率向量；往曲面的切平面做投影，得到向量就是测地曲率向量，这个向量的大小就是**测地曲率**；
2. 从定义上看，测地曲率刻画了**曲线在曲面内蕴的弯曲程度**，而法曲率刻画了曲线在嵌入空间的弯曲程度。比如一张平面上的直线的测地曲率为 0，法曲率为 0，如果把这张纸弯曲成圆柱，纸上的直线在三维空间就弯曲了，但是测地曲率还是为 0。

所以，法曲率是为了维持曲线在曲面上而有的曲率。测地曲率 $\kappa_g$ 则是偏离测地线的弯曲程度。

以下示意图可更方便进行理解：

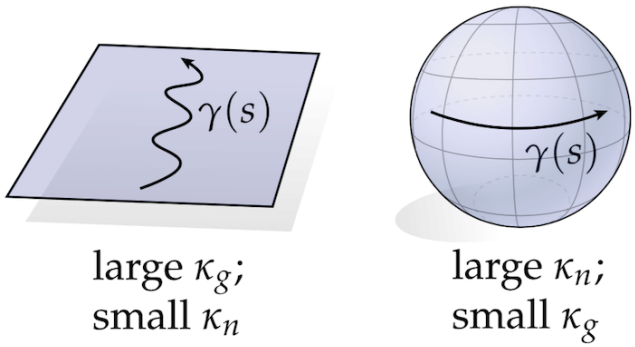



**图 30:** 法曲率和测地曲率几何意义示意图

可以看出，在弯曲程度很低的曲面上画一根曲线，不管如何弯曲，其法曲率都很小，此时测地曲率可以很大；而针对球面，沿着比如赤道切开，其法曲率的倒数就是球体的半径，而此时测地曲率为零（**此时曲面上两点之间距离最短**）。

前面说到，曲面上一点可以有无数个法曲率，这些曲率中，自然会产生一个最大值和最小值，称为**主曲率**，记作 $\kappa_1$ 与 $\kappa_2$。当曲率取最大与最小值的两个法平面方向总是**垂直**的，这是 Euler 在 1760 年的一个结论，称之为**主方向**，一般用 $\{\boldsymbol{e}_1, \boldsymbol{e}_2\}$ 表示，如下图所示：

光滑曲面上任意一点的 2 个主曲率方向一定满足正交关系的证明如下：

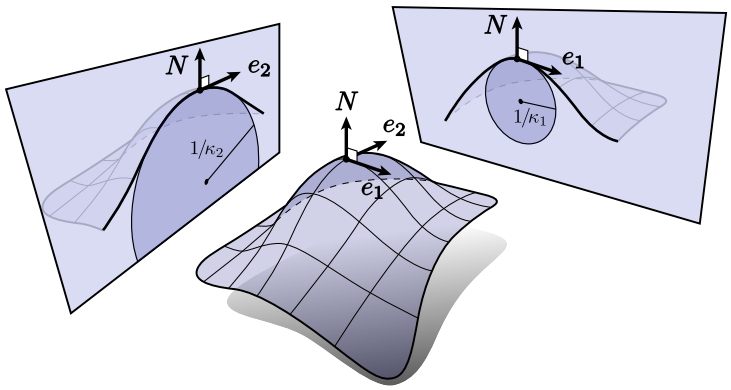



图 31: 主曲率几何意义示意图

**证明.** 考虑上述法曲率的计算表达式，引入 $\lambda = \mathrm{d}v/\mathrm{d}u$，则有

$$k_n = \frac{L + 2M\lambda + N\lambda^2}{E + 2F\lambda + G\lambda^2} \tag{155}$$

可以看到，此时法曲率 $\kappa_n$ 写成了参数 $\lambda$ 的一元函数。法曲率有无穷多解的原因也在于 $\lambda$ 的取值不同。所以想要求得主曲率，直接对上式求导并令其为零即可：

$$\frac{\mathrm{d}k_n}{\mathrm{d}\lambda} = \frac{(2M\lambda + 2N\lambda)\left(E + 2F\lambda + G\lambda^2\right) - (2F\lambda + 2G\lambda)\left(L + 2M\lambda + N\lambda^2\right)}{\left(E + 2F\lambda + G\lambda^2\right)^2} = 0 \tag{156}$$

化简得：

$$(NF - GM)\lambda^2 + (EN - LG)\lambda + EM - FL = 0 \tag{157}$$

对于光滑曲面，很明显此时会存在两个实根 $\lambda_1, \lambda_2$（也有可能两者相同），根据 *Vieta* 定理：

$$\lambda_1 + \lambda_2 = \frac{LG - EN}{NF - GM} \quad \lambda_1\lambda_2 = \frac{EM - FL}{NF - GM} \tag{158}$$

所对应的主方向为 $\boldsymbol{e}_1, \boldsymbol{e}_2$，根据切向量的计算表达式，有：

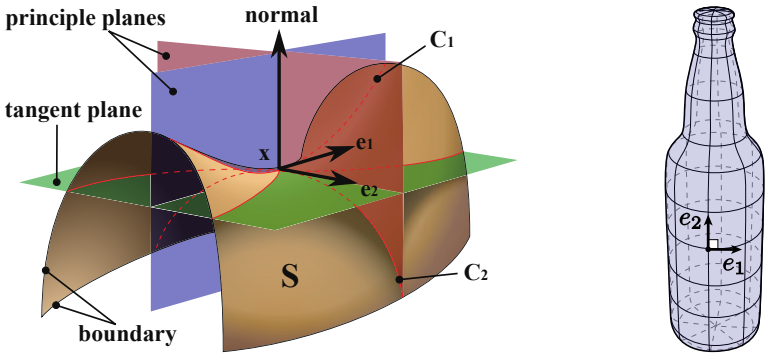
$$\boldsymbol{e}_1\boldsymbol{e}_2 = \boldsymbol{r}_u^2 \mathrm{d}u_1\mathrm{d}u_2 + \boldsymbol{r}_u\boldsymbol{r}_v\left(\mathrm{d}u_1\mathrm{d}v_2 + \mathrm{d}u_2\mathrm{d}v_1\right) + \boldsymbol{r}_v^2\mathrm{d}v_1\mathrm{d}v_2 \tag{159}$$

上式除以 $\mathrm{d}u_1\mathrm{d}u_2$ 并代入 $\lambda_1, \lambda_2$ 可有：

$$\frac{\boldsymbol{e}_1\boldsymbol{e}_2}{\mathrm{d}u_1\mathrm{d}u_2} = E + F(\lambda_1 + \lambda_2) + G\lambda_1\lambda_2 = 0 \tag{160}$$

证毕。

说明主曲率和主方向最经典的曲面就是“**马鞍面**”（如下图32(a)），但其实生活中这样经典的曲面还有许多，例如汽水瓶表面（如下图32(b)）：




(a) (b)

图 32: 主方向相互正交示意图

有了主曲率的定义，可以把法曲率写成主曲率的形式，假设 $\boldsymbol{e}$ 作为曲面上的一个切向量，其与主方向 $\boldsymbol{e}_1$ 的夹角为 $\theta$，则有：

$$\boldsymbol{e} = \cos\theta \boldsymbol{e}_1 + \sin\theta \boldsymbol{e}_2 \tag{161}$$

此时法曲率可以计算得到：

$$\kappa_n = \kappa_1 \cos^2\theta + \kappa_2 \sin^2\theta \tag{162}$$

这就是前面说到的著名的 ***Euler* 公式**。

证明从略，因为要引入 *Weingarten* 映射（可参考其他微分几何教材），但其他证法又没思路。

接下来考虑主曲率的计算方式，同样需要根据 *Weingarten* 映射进行计算。这里为求简洁，直接给出最后结果：

$$\begin{aligned} k_1 k_2 &= \frac{LN-M^2}{EG-F^2} \\ k_1 + k_2 &= \frac{LG-2MF+NE}{EG-F^2} \end{aligned} \tag{163}$$

其中，

$$K = \kappa_1 \kappa_2 \tag{164}$$

被定义为**高斯曲率**。

$$H = \frac{1}{2}(k_1 + k_2) \tag{165}$$

定义为**平均曲率**。

高斯曲率和平均曲率有许多几何特性，这里仅总结其中最关键的点：

1. **高斯曲率**：属于曲面曲率概念，它度量了曲面内在的弯曲程度。一个曲面做任何非拉伸的变换都不会改变它的高斯曲率，如平面，高斯曲率为0，把它弯曲成圆柱，其高斯曲率也还是为0。高斯曲率为0的曲面也叫**可展曲面**，它展平到平面不会产生扭曲。高斯曲率也等于两个主曲率的乘积，但是它本身不依赖主曲率；
2. **平均曲率**：属于曲面曲率概念，它度量了曲面在空间中的弯曲程度。比如平面弯曲成圆柱后，其平均曲率就不为0了。它等于主曲率的平均值。平均曲率为0的曲面也叫**极小曲面**，如肥皂薄膜。

注 数学上，“极小曲面”是指微小变化下面积最小的曲面，而不是所有可能曲面中最小的曲面，因此被称为“极小曲面”而不是“最小曲面”。

下图可以很好地展示高斯曲率和平均曲率在不同曲面上的直观感觉：

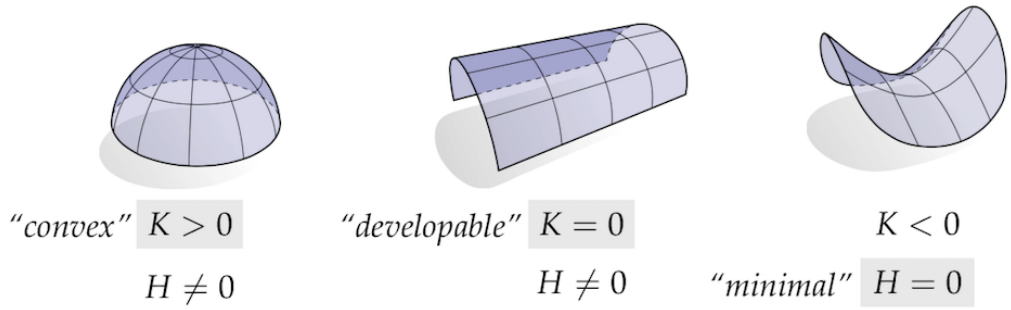



图 33: 不同曲面所对应的高斯曲率和平均曲率

更进一步，事实上，对三维空间中的曲面来说，平均曲率和高斯曲率其实都可以由**曲面的单位法向量场**来定义，这个在一般的微分几何书籍里都很少看到：

$$\begin{aligned} H &= -\frac{1}{2}\operatorname{div}(\boldsymbol{N}) \\ K &= \frac{1}{2}\operatorname{div}[\boldsymbol{N}\cdot\operatorname{div}(\boldsymbol{N}) + \boldsymbol{N}\times\operatorname{curl}(\boldsymbol{N})] \end{aligned} \tag{166}$$

上述公式证明从略，因为不会证，看了参考书籍也看不懂。

这里法向量的选取影响曲率的正负号。曲率的符号取决于法向量的方向：如果曲面“远离”法向量则曲率是正的。上面的公式对三维空间中任何方式定义的曲面都成立，只要能够计算单位法向量的散度。

当曲面为显式曲面时，曲面 $z=f(x,y)$ 的单位法向量可通过下式计算：

$$\boldsymbol{N} = \frac{\boldsymbol{r}_x \times \boldsymbol{r}_y}{|\boldsymbol{r}_x \times \boldsymbol{r}_y|} \tag{167}$$

由于

$$\boldsymbol{r}_x = (1,0,f_x) \quad \boldsymbol{r}_y = (0,1,f_y) \quad \boldsymbol{r}_x \times \boldsymbol{r}_y = (-f_x,-f_y,1) \tag{168}$$

所以

$$\boldsymbol{N}=\frac{\{-\nabla f,1\}}{\sqrt{1+|\nabla f|^2}}=\left(\frac{-f_x}{\sqrt{1+f_x^2+f_y^2}},\frac{-f_y}{\sqrt{1+f_x^2+f_y^2}},\frac{1}{\sqrt{1+f_x^2+f_y^2}}\right) \tag{169}$$

显式曲面的平均曲率最终可以表示为：

$$H=-\frac{1}{2}\operatorname{div}_{(x,y,z)}\left(\frac{\{-\nabla f,1\}}{\sqrt{1+|\nabla f|^2}}\right)=\frac{1}{2}\operatorname{div}_{(x,y)}\left(\frac{\nabla f}{\sqrt{1+|\nabla f|^2}}\right) \tag{170}$$

当曲面为隐式曲面时，曲面 $F(x,y,z)=0$ 的单位法向量可根据“等高线”与梯度的关系求得：

$$\boldsymbol{N}=\frac{\nabla F}{|\nabla F|}=\left(\frac{F_x}{\sqrt{F_x^2+F_y^2+F_z^2}},\frac{F_y}{\sqrt{F_x^2+F_y^2+F_z^2}},\frac{F_z}{\sqrt{F_x^2+F_y^2+F_z^2}}\right) \tag{171}$$

隐式曲面的平均曲率最终可以表示为：

$$H=-\frac{1}{2}\operatorname{div}_{(x,y,z)}\left(\frac{\nabla F}{|\nabla F|}\right) \tag{172}$$

通过第一第二基本形式，或上述矢量表达式，可以推导出展开后的平均曲率表达式：

- 显式曲面 $z=f(x,y)$：

$$H=\frac{\left(1+f_x^2\right)f_{yy}-2f_xf_yf_{xy}+\left(1+f_y^2\right)f_{xx}}{2\left(1+f_x^2+f_y^2\right)^{\frac{3}{2}}} \tag{173}$$

- 隐式曲面 $F(x,y,z)=0$ 平均曲率的展开式太长，且用处不大，但推导过程并不复杂，这里从略。

## 4.2 其余预备基础

### 4.2.1 病态问题

**定义 4.7 (病态问题)** 指输出结果对输入数据非常敏感，如果输入数据有微小误差，则引起解的误差会非常大。

在偏微分方程领域，最后的落脚点基本都是**数值计算**，毕竟常微分方程想寻求通解已然很难，更不用说偏微分方程了。在构建数值算法的时候，很明显算法有”优劣”之分，问题也有”好坏”之别。在数值分析的问题中，如线性代数方程组、矩阵特征值、非线性方程及方程组都存在病态问题。

一般来说，考察数值计算方法的好坏主要从**收敛性**和**稳定性**两方面来考虑。

收敛性很好理解，**在构建离散格式的时候，如果假设离散的步长越来越小时，数值解能够越接近理论解，即说明算法具有收敛性**。

而稳定性则是与病态问题相关的，由于一般数值计算方法都是逐层进行的，计算第 $n+1$ 层上的函数值时，要用到第 $n$ 层上计算所得的结果。因此上一层的计算误差必然会影响到下一层，从而就要分析这种误差传播的情况。**如果误差的影响越来越大，则此种数值格式是不稳定的，病态的；相反，如果误差的影响可以控制，则是稳定的**。

下面可以用数学化的方式对该概念进行描述：

**定义 4.8 (数值格式的稳定性)** *对于定解问题的某个数值格式，设初始层上引入误差 $e_0$，记 $e_n$ 是第 $n$ 层上的误差，如果存在常数 $K$ 使得*

$$\|e_n\| \leq K\|e_0\| \tag{174}$$

*那么称该数值格式是稳定的。其中 $\|\cdot\|$ 为某种顶一下的范数。*

**例 4.3** *考虑流体力学中对流方程*

$$\frac{\partial u}{\partial t}+\frac{\partial u}{\partial x}=0 \tag{175}$$

*的差分格式*

$$\frac{u_{j,n+1}-u_{j,n}}{\tau}+\frac{u_{j,n}-u_{j-1,n}}{h}=0 \tag{176}$$

*的稳定性。其中 $t$ 为时间，$x$ 为空间（一维），$j$ 和 $n$ 分别是两个参数离散后的节点，$\tau$ 和 $h$ 分别为相对应的步长。*

设在第 0 层上的网格节点上的 $u_{j,0}$ 有误差 $e_{j,0}$，即初值为 $u_{j,0}+e_{j,0}$ 而不是 $u_{j,0}$。用 $u_{j,0}+e_{j,0}$ 为初值进行计算，得到第 $n$ 层上的结果为 $u_{j,n}+e_{j,n}$。假定在这一计算过程中没有引进其他误差，那么 $u_{j,n}+e_{j,n}$ 满足差分方程(176)，即

$$\frac{\left(u_{j,n+1}+e_{j,n+1}\right)-\left(u_{j,n}+e_{j,n}\right)}{\tau}+\frac{\left(u_{j,n}+e_{j,n}\right)-\left(u_{j-1,n}+e_{j-1,n}\right)}{h}=0 \tag{177}$$

用上式减去(176)，可得

$$\frac{e_{j,n+1}-e_{j,n}}{\tau}+\frac{e_{j,n}-e_{j-1,n}}{h}=0 \tag{178}$$

这就是误差所满足的方程，或者另写为

$$e_{j,n+1}=(1-\lambda)e_{j,n}+\lambda e_{j-1,n} \tag{179}$$

其中 $\lambda = \tau / h$ 为步长比。

如果 $\lambda \leq 1$，则

$$\begin{aligned} \left|e_{j,n+1}\right| &\leq (1-\lambda)\left|e_{j,n}\right| + \lambda\left|e_{j-1,n}\right| \\ &\leq (1-\lambda)\sup_j\left|e_{j,n}\right| + \lambda\sup_j\left|e_{j,n}\right| = \sup_j\left|e_{j,n}\right| \end{aligned} \tag{180}$$

故

$$\sup_j\left|e_{j,n+1}\right| \leq \sup_j\left|e_{j,n}\right| \tag{181}$$

以及

$$\sup_j\left|e_{j,n+1}\right| \leq \sup_j\left|e_{j,n}\right| \leq \cdots \leq \sup_j\left|e_{j,0}\right| \tag{182}$$

也就是说，误差是不增长的，或者说差分格式(176)在条件 $\lambda \leq 1$ 下式稳定的。

### 4.2.2 欧拉-拉格朗日方程

欧拉-拉格朗日方程（*Euler-Lagrange* Equation）是针对变分问题的一类通用型解法，描述了一个泛函（函数的函数）取到极值的条件。在微积分中，对于一个连续函数来说，取得极指的必要条件是它的导函数等于 0，也就是驻点（stationary point）。但若想求得泛函的极值和极值点，就不是那么容易了。

变分法的基本内容就是确定泛函的极值及极值点（但是是特定形式下的泛函），在一定条件下，确定泛函的极值点和确定微分方程边值问题的解这两个问题是可以相互转化的。也就是说，**微分方程的边值问题常常可以化为变分问题来研究**。这种方法，就其本质而言，是将求解的基本微分方程的问题，化为求解某些泛函的极值（或驻值）问题。而在具体求解问题的近似解时，又进而可将求泛函的极值（或驻值）问题变为函数的极值（或驻值问题），最后将问题归结为求解线性（或非线性）代数方程组。

在介绍变分问题前，先引出以下引理，对微分的定义做一补充：

**引理 4.2 (微分的两种定义)** *对于函数 $y(x)$ 的微分，一般定义为函数的增量，即*

$$\Delta y = y(x+\Delta x) - y(x) = A(x)\Delta x + o(\Delta x) \tag{183}$$

*其中 $A(x)$ 与 $\Delta x$ 无关，且有 $\Delta x \to 0$ 时 $o(\Delta x) \to 0$，于是就称函数 $y(x)$ 是可微的，其线性部分称为函数的微分 $\mathrm{d}y = A(x)\Delta x = y'(x)\Delta x$。*

*除此之外，函数微分的定义还可以通过引入一微小参量 $\varepsilon$，对 $y(x+\varepsilon\Delta x)$ 关于 $\varepsilon$ 求导，并令 $\varepsilon \to 0$ 的途径得到，即*

$$\left.\frac{\mathrm{d}y(x+\varepsilon\Delta x)}{\mathrm{d}\varepsilon}\right|_{\varepsilon\to 0} = \left.y'(x+\varepsilon\Delta x)\Delta x\right|_{\varepsilon\to 0} = y'(x)\Delta x = \mathrm{d}y \tag{184}$$

上式说明 $y(x+\varepsilon\Delta x)$ 在 $\epsilon=0$ 处关于 $\epsilon$ 的导数就是函数 $f(x)$ 在 $x$ 处的微分。

相应地，如果考虑泛函 $\mathscr{F}[f(x)]$，变量函数 $f(x)$ 的增量在其很小时称为变分，一般可用 $\delta f(x)$ 来表示。根据以上引理的介绍，泛函的变分也可有类似的两个定义，令泛函 $\mathscr{F}$ 在函数 $f(x)$ 处的变分为 $\delta\mathscr{F}$，则第一种定义可为：

$$\delta\mathscr{F}=\left\{\mathscr{F}\left[f(x)+\delta f(x)\right]-\mathscr{F}\left[f(x)\right]\right\}_{\delta f\to 0} \tag{185}$$

另外一种定义，被称为**拉格朗日泛函变分**，是 $\mathscr{F}\left[f(x)+\varepsilon\delta f(x)\right]$ 对 $\varepsilon$ 的导数在 $\varepsilon=0$ 时的值，即

$$\delta\mathscr{F}=\frac{\partial}{\partial\varepsilon}\mathscr{F}\left[f(x)+\varepsilon\delta f(x)\right]_{\varepsilon\to 0} \tag{186}$$

下面考虑从一种最常见的函数到一个数的运算：定积分运算，来构建欧拉-拉格朗日方程。假设自变量集合中含有 $x, f(x), f'(x)$ 等，定义泛函

$$\mathscr{F}(f)=\int_{x_1}^{x_2}\mathscr{L}\left(x,f(x),f'(x)\right)\mathrm{d}x \tag{187}$$

现在考虑如何计算该定积分的极值。

**证明.** 根据拉格朗日泛函变分的定义，上述泛函的变分可表示为：

$$\delta\mathscr{F}=\frac{\partial}{\partial\varepsilon}\left[\int_{x_1}^{x_2}\mathscr{L}\left(x,f(x)+\varepsilon\delta f(x),f'(x)+\varepsilon\delta f'(x)\right)\mathrm{d}x\right]_{\varepsilon\to 0} \tag{188}$$

这个表达式非常巧妙地将原本不能求极值点的情况，变成了可以求极值点的情况。利用链式求导法则和积分运算的线性可进一步求得：

$$\delta\mathscr{F}=\int_{x_1}^{x_2}\left(\frac{\partial\mathscr{L}}{\partial f}\delta f+\frac{\partial\mathscr{L}}{\partial f'}\delta f'\right)\mathrm{d}x \tag{189}$$

注意，积分内右部分可以通过分部积分进行转换（原理：$\mathrm{d}(uv)=u\mathrm{d}v+v\mathrm{d}u$），

$$\int_{x_1}^{x_2}\left(\frac{\partial\mathscr{L}}{\partial f'}\delta f'\right)\mathrm{d}x=\int_{x_1}^{x_2}\left(\frac{\partial\mathscr{L}}{\partial f'}\right)\mathrm{d}\left(\delta f\right)=\delta f\frac{\partial\mathscr{L}}{\partial f'}\bigg|_{x_1}^{x_2}-\int_{x_1}^{x_2}\delta f\mathrm{d}\left(\frac{\partial\mathscr{L}}{\partial f'}\right) \tag{190}$$

考虑到存在固定边界条件，即增量扰动需要保证起始点和结束点为 0，则 $\delta f(x_1)=\delta f(x_2)=0$，所以上式左边等于 0，

$$\int_{x_1}^{x_2}\left(\frac{\partial\mathscr{L}}{\partial f'}\eta'\right)\mathrm{d}x=-\int_{x_1}^{x_2}\eta\mathrm{d}\left(\frac{\partial\mathscr{L}}{\partial f'}\right)=-\int_{x_1}^{x_2}\eta\frac{\mathrm{d}}{\mathrm{d}x}\left(\frac{\partial\mathscr{L}}{\partial f'}\right)\mathrm{d}x \tag{191}$$

最终公式(189)变化为：

$$\delta\mathscr{F}=\int_{x_1}^{x_2}\delta f\left(\frac{\partial\mathscr{L}}{\partial f}-\frac{\mathrm{d}}{\mathrm{d}x}\left(\frac{\partial\mathscr{L}}{\partial f'}\right)\right)\mathrm{d}x \tag{192}$$

考虑到泛函 $\mathscr{F}$ 的极值条件为 $\delta\mathscr{F}=0$，同时保证对任意 $\delta f$ 成立，必须有：

$$\frac{\partial \mathscr{L}}{\partial f}-\frac{\mathrm{d}}{\mathrm{d}x}\left(\frac{\partial \mathscr{L}}{\partial f'}\right)=0 \tag{193}$$

上式就是 $\mathscr{F}(f)$ 取得极值的条件，也即 *Euler-Lagrange* Equation。

更进一步，考虑多元函数的变分问题，可定义泛函如下：

$$\mathscr{F}(f)=\int_{\Omega}\mathscr{L}(\boldsymbol{x},f(\boldsymbol{x}),\nabla f(\boldsymbol{x}))\mathrm{d}\boldsymbol{x} \tag{194}$$

其中，$f(\boldsymbol{x})$ 为多元连续函数（还不能称之为向量值映照，因为函数值还是一维的），$\boldsymbol{x}=[x_1,x_2,\ldots,x_n]$，积分区域 $\Omega\subset\mathbb{R}^n$。

可依据上述类似的推导过程推导出最终的极值条件：

$$\frac{\partial \mathscr{L}}{\partial f}(\boldsymbol{x},f(\boldsymbol{x}),\nabla f(\boldsymbol{x}))-\frac{\partial}{\partial x^i}\left(\frac{\partial \mathscr{L}}{\partial \nabla_i f}\right)(\boldsymbol{x})=0 \tag{195}$$

上式中的指标形式满足 ***Einstein* 求和约定**，一对**哑指标**在一项里同时出现，表示**遍历其取值范围求和**。

根据梯度的定义，上式也可进一步记为

$$\frac{\partial \mathscr{L}}{\partial f}(\boldsymbol{x},f(\boldsymbol{x}),\nabla f(\boldsymbol{x}))-\nabla\cdot\left(\frac{\partial \mathscr{L}}{\partial \boldsymbol{\nabla} f}\right)(\boldsymbol{x})=0 \tag{196}$$

这就是多元函数情形下的 *Euler-Lagrange* Equation。

### 4.2.3 扩散（热传导）方程

热传导方程在图像处理中应用广泛，原因在于图像去噪的过程本质上就是一个高频向低频扩散的过程，这与热传导中高温向低温扩散的原理不谋而合。本小节对热传导方程做一介绍，以期更好地理解扩散原理。

**定义 4.9 (傅里叶定律)** 假设垂直通过微元面积 $\mathrm{d}A$ 上的导热热流量为 $\mathrm{d}\Phi$，则 $\mathrm{d}A$ 上的导热热流密度 $q$ 为

$$q=\frac{\mathrm{d}\Phi}{\mathrm{d}A} \tag{197}$$

注意，热流密度具有方向，其流向必定指向温度降低的方向。傅里叶在进行大量实验研究后发现了导热热流密度与温度梯度之间的关系，提出了以下著名的导热基本定律——傅里叶定律。对于物性参数不随方向变化的各向同性物体，其数学表达式为：

$$\boldsymbol{q}=-\lambda\nabla T \tag{198}$$

即，导热热流密度的大小与温度梯度的绝对值成正比，其方向与温度梯度的方向相反。其中，$T$ 为温度大小，$\lambda$ 为导热率，为常数。

有了傅里叶定律，接下来就可以很方便地推导出**导热微分方程**。

在导热过程中，微元体的热平衡可表述为：单位时间内，**净导入微元体的热流量**与**微元体内热源的生成热**之和，等于微元体热力学能的增加。

一般来说，物体不会凭空产生热量（图像亦是如此），这里为方便简化，便不考虑微元体内热源的生成热。而净导入微元体的热流量，假设用 $\mathrm{d}\Phi_V$ 指代，导热的总方向矢量为 $\boldsymbol{l}$，则 $\boldsymbol{l}$ 方向净导入微元体的热量为：

$$\mathrm{d}\Phi_V = \mathrm{d}\Phi_V(\boldsymbol{l}) - \mathrm{d}\Phi_V(\boldsymbol{l}+\mathrm{d}\boldsymbol{l}) = \left[\boldsymbol{q}(\boldsymbol{l}) - \boldsymbol{q}(\boldsymbol{l}+\mathrm{d}\boldsymbol{l})\right]\mathrm{d}A \tag{199}$$

根据泰勒展开，保留其一阶导数项，同时注意热流密度 $\boldsymbol{q}$ 的方向就是 $\boldsymbol{l}$ 指向的方向，则上式进一步变为

$$\mathrm{d}\Phi_V = \left(\frac{\partial \boldsymbol{q}}{\partial l}\mathrm{d}\boldsymbol{l}\right)\mathrm{d}A = -\lambda\nabla\cdot(\nabla T)\,\mathrm{d}V \tag{200}$$

其中，梯度的表达式借鉴了方向导数的定义，并考虑到梯度作为矢量本身指向 $\boldsymbol{q}$ 和 $\boldsymbol{l}$ 的反向，所以可用梯度的内积来替换热流密度的方向导数。另外，$\mathrm{d}V = \mathrm{d}\boldsymbol{l}\mathrm{d}A$ 刚好指代微元体的体积。

与此同时，单位时间内，微元体热力学能的增加 $\mathrm{d}U$ 为

$$\mathrm{d}U = \rho c\frac{\partial T}{\partial t}\mathrm{d}V \tag{201}$$

其中，$t$ 为时间，$\rho$ 为物体的密度，$c$ 为物体的比热容。

最后，根据微元体的热平衡关系，可以最终得到：

$$\frac{\partial T}{\partial t} = a\nabla^2 T \tag{202}$$

这就是**热传导方程**的最终形式。式中，$\nabla^2$ 为拉普拉斯算子，也可用 $\Delta$ 指代。在直角坐标系中，

$$\nabla^2 T = \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2 T}{\partial z^2} \tag{203}$$

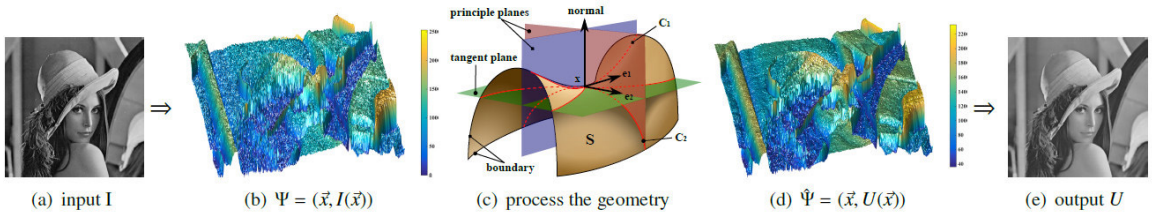
$a = \frac{\lambda}{\rho c}$，称为**热扩散率**，也叫**导温系数**。

注意，热传导方程是**二阶偏微分方程**，若要想求解，需要**两个边界条件**和**一个初始条件**。

## 4.3 图像与曲率

本节探讨图像与曲率之间的联系。前面微分几何工具终于派上用场。

给定图像 $I(x,y)$，可以将其当做一个三维曲面 $\big(x,y,I(x,y)\big)$。其实这就是一个典型的 *Monge* 形式，所以可以利用经典的微分几何工具来处理该三维曲面。假设处理后的曲面为 $\big(x,y,U(x,y)\big)$，则可以提取出处理后的图像 $U(x,y)$，这也是经过图像处理后的目标图像。具体演示过程如下图34所示：




(a) input I　(b) $\Psi=(\vec{x},I(\vec{x}))$　(c) process the geometry　(d) $\hat{\Psi}=(\vec{x},U(\vec{x}))$　(e) output $U$

图 34: 图像处理与曲面之间的联系

以这种方式进行图像处理的论文非常多，基本的思路都是把图像嵌入三维空间、计算各种曲率（高斯曲率、平均曲率、主曲率等等），然后根据曲率的大小进行各向异性扩散。而且这种扩散方程在数学上是非常完善的，也就是经典的几何流，如下图35所示。

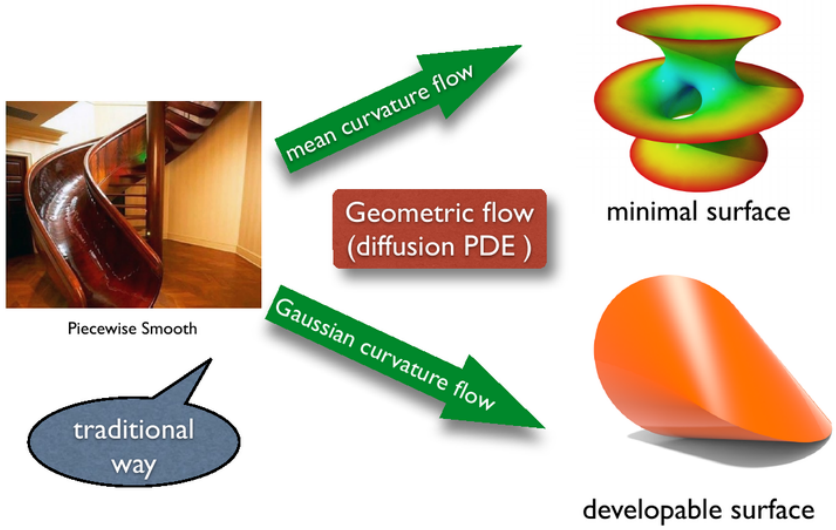



图 35: 传统的几何流算法

所以这就是图像跟微分几何有十分密切关系的原因。为了更直观描述，接下来通过一段 Python 代码来演示如何计算图像的平均曲率（利用公式(173)）并用图像展示其数值变化，WIN 10 平台，Python 3.7 下调试通过：

```
1 import numpy as np
2 import cv2
3
4 def cal_curvature(img):
```

```
    x, y = np.gradient(img)  #一阶导数
    xx, xy = np.gradient(x)  #二阶偏导数
    yx, yy = np.gradient(y)  #二阶偏导数

    Iup =  (1+x*x)*yy - 2*x*y*xy + (1+y*y)*xx  # 公式的分子
    Idown = 2*np.power(((1 + x*x + y*y)),1.5)   #公式的分母
    final = Iup/Idown
    final = (final-final.min())/(final.max()-final.min())  #将结果归一化
    final = final * 255  #将像素值扩展为0-255
    final = final.astype(np.uint8)
    return final

img = cv2.imread("cat1.jpg", 0)
img = img.astype("float")
cv2.imshow("meancurvature", cal_curvature(img))  # or use cv2.imwrite to save
    the figure
```

依旧用的是之前的猫片，最后呈现出的平均曲率如下图：

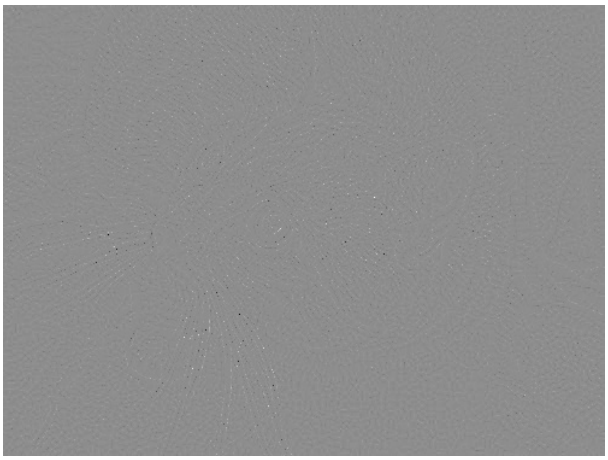


**图 36:** 图像的平均曲率

仔细看图片，可以看出一点猫的轮廓，特别是胡须部分，说明图像边沿处的平均曲率更大，符合基本认知。

## 4.4 经典算法介绍

### 4.4.1 P-M 方程

首先是 1990 年的著名图像去噪模型：P-M 方程，他们的研究开辟了偏微分方程在图像处理领域中应用的很多新方向，因此他们的工作被认为是该领域最有影响的工作。

基本上只要做这个方向的都会引用该篇论文。

前面有提到过热扩散方程，应用在图像上，处理后的结果即为下述线性各向同性扩散方程的解：

$$\begin{cases} \frac{\partial I}{\partial t} = \Delta I = \operatorname{div}(\nabla I) \\ I(x,y,0) = I_0(x,y) \end{cases} \quad (x,y) \in R^2 \tag{204}$$

可以证明，这种处理方式等同于用 ***Gauss* 滤波**来进行图像去噪，即利用如下 *Gauss* 函数 $G_\sigma$

$$G_\sigma(x,y) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2+y^2}{2\sigma^2}\right) \tag{205}$$

对 $I_0$ 卷积：

$$I(x,y,t) = G_\sigma(x,y) * I_0(x,y) \tag{206}$$

得到 $t$ 时刻去噪图像，其中 $t = 0.5\sigma^2$。

简要证明如下，为简洁这里仅利用一维扩散（热传导）方程来说明：

**证明.** 考虑如下的一维扩散问题（热扩散率 $a = 1$）：

$$\begin{cases} \frac{\partial I}{\partial t} = \frac{\partial^2 I}{\partial x^2}, & -\infty < x < +\infty, t > 0 \\ I(x,0) = I_0(x) & -\infty < x < +\infty \end{cases} \tag{207}$$

对上式两边同时关于变量 $x$ 求傅里叶变换可得：

$$\begin{cases} \frac{\mathrm{d}\bar{I}}{\mathrm{d}t} = \lambda^2 \tilde{I} \\ \tilde{I}\big|_{t=0} = \tilde{I}_0(\lambda) \end{cases} \tag{208}$$

其中，$\tilde{I}$ 和 $\tilde{I}_0$ 分别为 $I$ 和 $I_0$ 的傅里叶变换。通过积分求解上式可得如下解的表达式：

$$\tilde{I}(\lambda,t) = \tilde{I}_0 \exp\left(\lambda^2 t\right) \tag{209}$$

对上式两边求傅里叶逆变换可得：

$$I(x,t) = \frac{1}{2\sqrt{\pi t}} \int_{-\infty}^{+\infty} I_0(\xi) \exp\left(\frac{-(\xi-x)^2}{4t}\right) \mathrm{d}\xi = G(x,t) * I_0(x) \tag{210}$$

其中，

$$G(x,t) = \begin{cases} \frac{1}{2\sqrt{\pi t}} \exp\left(\frac{-x^2}{4t}\right), & t > 0 \\ 0, & t = 0 \end{cases} \tag{211}$$

显然 $G(x,t)$ 是 *Gauss* 函数，在此式中令 $4t = 2\sigma^2$ 就可得到方差为 $\sigma^2$ 的 *Gauss* 函数。再进一步推广到二维情况，即可得证。

所以可以发现，扩散方程里的时间 $t$ 实际上就是 *Gauss* 滤波的尺度因子，但是这是一个各向同性的扩散过程，即**在图像边缘处沿切向和法向是同等扩散的**，虽然能去除图像噪声但不能保护图像边缘。

相比于热扩散方程，P-M 方程构造出了一个拟各向异性的扩散方程如下：

$$\begin{cases} \frac{\partial I(x,y,t)}{\partial t} = \mathrm{div}(g(|\nabla I|)\cdot\nabla I) \\ I_0 = I(x,y,0) \end{cases} \tag{212}$$

其中 $g(|\nabla I|) \in [0,1]$ 为**扩散系数**，或称为**边缘停止函数**（Egde-Stopping Function），它是一个关于梯度的单调递减函数。这个参数的引入非常巧妙地减少了在图像边缘处法向的扩散，因为那是梯度增大的方向，此时该参数会起到抑制扩散的作用。

论文里提出了两个典型的扩散系数：

$$\begin{aligned} g(\nabla u) &= \exp\left(-\left(\frac{|\nabla I|}{k}\right)^2\right) \\ g(\nabla u) &= 1\Big/\left(1+\frac{|\nabla I|^2}{k^2}\right) \end{aligned} \tag{213}$$

其中常数 $k$ 为阈值，可以预先设定，也可以随着图像每次迭代的结果变化而变化。

论文中采用了数值解法来求解上述偏微分方程，主要利用如下差分格式：

$$\frac{I_{x,y}^{n+1}-I_{x,y}^{n}}{\Delta t} = g\left(\left|\Delta_x^+\right|\right)\Delta_x^+ - g\left(\left|\Delta_x^-\right|\right)\Delta_x^- + g\left(\left|\Delta_y^+\right|\right)\Delta_y^+ - g\left(\left|\Delta_y^-\right|\right)\Delta_y^- \tag{214}$$

其中，

$$\begin{cases} \Delta_x^+ = I_{x+1,y}^n - I_{x,y}^n \\ \Delta_x^- = I_{x,y}^n - I_{x-1,y}^n \end{cases}, \begin{cases} \Delta_y^+ = I_{x,y+1}^n - I_{x,y}^n \\ \Delta_y^- = I_{x,y}^n - I_{x,y-1}^n \end{cases} \tag{215}$$

如果都以中心点为被减数，上述差分格式的第 2 项和第 4 项也可以改成 + 号。另外，除了前后左右可以差分外，斜对角线的元素也可以差分，这就可以演化成八向扩散的差分格式。

除此之外，该差分格式中有三个参数需要设置，迭代次数 $t$，根据情况设置；扩散系数里的 $k$，取值越大越平滑，越不容易保留边缘；还有步长 $\Delta t$，取值越小越精细。

下面是代码实现，WIN 10 平台，Python 3.7 下调试通过：

```
1 import cv2
2 import numpy as np
3 import math
4
5 class anisodiff2D(object):
```

```
6       def __init__(self, num_iter=5, delta_t=1/7, kappa=30, option=2):
7           super(anisodiff2D, self).__init__()
8           self.num_iter = num_iter
9           self.delta_t = delta_t
10          self.kappa = kappa
11          self.option = option
12          self.hN = np.array([[0, 1, 0], [0, -1, 0], [0, 0, 0]])
13          self.hS = np.array([[0, 0, 0], [0, -1, 0], [0, 1, 0]])
14          self.hE = np.array([[0, 0, 0], [0, -1, 1], [0, 0, 0]])
15          self.hW = np.array([[0, 0, 0], [1, -1, 0], [0, 0, 0]])
16          self.hNE = np.array([[0, 0, 1], [0, -1, 0], [0, 0, 0]])
17          self.hSE = np.array([[0, 0, 0], [0, -1, 0], [0, 0, 1]])
18          self.hSW = np.array([[0, 0, 0], [0, -1, 0], [1, 0, 0]])
19          self.hNW = np.array([[1, 0, 0], [0, -1, 0], [0, 0, 0]])
20
21      def fit(self, img):
22          diff_im = img.copy()
23          dx = 1; dy = 1; dd = math.sqrt(2)
24
25          for i in range(self.num_iter):
26              nablaN = cv2.filter2D(diff_im, -1, self.hN)
27              nablaS = cv2.filter2D(diff_im, -1, self.hS)
28              nablaW = cv2.filter2D(diff_im, -1, self.hW)
29              nablaE = cv2.filter2D(diff_im, -1, self.hE)
30              nablaNE = cv2.filter2D(diff_im, -1, self.hNE)
31              nablaSE = cv2.filter2D(diff_im, -1, self.hSE)
32              nablaSW = cv2.filter2D(diff_im, -1, self.hSW)
33              nablaNW = cv2.filter2D(diff_im, -1, self.hNW)
34              cN = 0; cS = 0; cW = 0; cE = 0; cNE = 0; cSE = 0; cSW = 0; cNW = 0
35
36              if self.option == 1:
37                  cN = np.exp(-(nablaN/self.kappa)**2)
38                  cS = np.exp(-(nablaS/self.kappa)**2)
39                  cW = np.exp(-(nablaW/self.kappa)**2)
40                  cE = np.exp(-(nablaE/self.kappa)**2)
41                  cNE = np.exp(-(nablaNE/self.kappa)**2)
42                  cSE = np.exp(-(nablaSE/self.kappa)**2)
43                  cSW = np.exp(-(nablaSW/self.kappa)**2)
44                  cNW = np.exp(-(nablaNW/self.kappa)**2)
```

```
45                elif self.option == 2:
46                    cN = 1/(1+(nablaN/self.kappa)**2)
47                    cS = 1/(1+(nablaS/self.kappa)**2)
48                    cW = 1/(1+(nablaW/self.kappa)**2)
49                    cE = 1/(1+(nablaE/self.kappa)**2)
50                    cNE = 1/(1+(nablaNE/self.kappa)**2)
51                    cSE = 1/(1+(nablaSE/self.kappa)**2)
52                    cSW = 1/(1+(nablaSW/self.kappa)**2)
53                    cNW = 1/(1+(nablaNW/self.kappa)**2)
54
55                diff_im = diff_im + self.delta_t * (
56                    (1/dy**2)*cN*nablaN + (1/dy**2)*cS*nablaS +
57                    (1/dx**2)*cW*nablaW + (1/dx**2)*cE*nablaE +
58                    (1/dd**2)*cNE*nablaNE + (1/dd**2)*cSE*nablaSE +
59                    (1/dd**2)*cSW*nablaSW + (1/dd**2)*cNW*nablaNW)
60            return diff_im
61
62    cat = cv2.imread("cat1.jpg", 0)
63    cat = cat.astype("float")  # 图像像素值为ubyte类型，需要转换
64    pm = anisodiff2D()
65    pmm = pm.fit(cat).astype("uint8")  # 需要再转回ubyte类型
66    cv2.imshow("pmm",pmm)  # or use cv2.imwrite to save the figures
67    cv2.waitKey()
```

上述代码封装了一个八向扩散的类，用 opencv 里的 filter2D 函数来进行差分计算。如果仅用四向扩散，将上述代码里包含 NE、SE、SW 和 NW 的部分均删除即可。最终计算结果如下图，可以看到非常明显的效果，并且八向扩散的效果更明显。

(a) 四向扩散　　(b) 八向扩散

**图 37:** P-M 方程图像去噪

### 4.4.2 ROF 模型

ROF 模型在 92 年由 Rudin，Osher，Fatemi 提出，他们提出“最小化‘全变差’”可以有更好的去除图像中噪声的效果。

首先在原论文中，全变差（TV，Total Variation）是图像差分的 $L1$ 范数，即

$$\begin{aligned} \mathrm{TV}(I) &= \sum_{x,y} \sqrt{\big(I(x+1,y)-I(x,y)\big)^2 + \big(I(x,y+1)-I(x,y)\big)^2} \\ &= \sum_{x,y} \sqrt{I_x^2 + I_y^2} = \iint_{\Omega} |\nabla I| \mathrm{d}x\mathrm{d}y \end{aligned} \tag{216}$$

本质上还是一种针对图像梯度进行优化的算法（其中 $\Omega$ 域即图像域）。另外，为了保证图像与原图像的一致性，该模型还引入了保真项，保证解 $I$ 与初值 $I_0$ 的偏差不会太大，最终演化成**泛函求极值问题，即变分问题**。

根据上述分析，重新阐述 ROF 模型如下：

该模型是一个能量最小化问题 $\min E(I)$，其中：

$$E(I) = \iint_a I(x,y)\mathrm{d}x\mathrm{d}y \tag{217}$$

且满足以下两个约束条件：

$$\begin{aligned} &\iint_{\Omega} I(x,y)\mathrm{d}x\mathrm{d}y = \iint_{\Omega} I_0(x,y)\mathrm{d}x\mathrm{d}y \\ &\frac{1}{\Omega}\int_{\Omega} \big(I(x,y)-I_0(x,y)\big)^2 \mathrm{d}x\mathrm{d}y = \sigma^2 \end{aligned} \tag{218}$$

其中，$I$ 为原始图像，$I_0$ 为加入均值为 0，方差为 $\sigma^2$ 的噪声后图像。上式中第一个等式可以保证**求出的图像与原图像在整体灰度上相等**，另外第二个等式中的 $\sigma$ 在实际问题中并不能提前得知。引入参数 $\lambda$，将式217转化成一个条件极值问题 $\min E(I)$，其中：

$$E(I) = \iint_{\Omega} |\nabla I| \mathrm{d}x\mathrm{d}y + \frac{\lambda}{2}\iint_{\Omega} \big|I(x,y)-I_0(x,y)\big|^2 \mathrm{d}x\mathrm{d}y \tag{219}$$

$\lambda$ 为依赖于噪声水平的尺度参数，对平滑和去噪起到重要的平衡作用。$\lambda$ 过大，很明显上式中第二项的权重就会增大，会更倾向于保真而不能很好地去噪；反过来 $\lambda$ 太小，会造成过度平滑现象。

式(219)就是著名的 ROF 模型，其右边第一项称为图像问题泛函的正则项，其作用是在能量泛函极小化的过程中，去除噪声平滑图像；第二项称为图像泛函保真项，它的作用是控制演化图像 $I$ 和带噪观测图像 $I_0$ 之间的差异程度，起到保护图像边缘等几何结构的信息以及降低失真度的作用。

可以证明，ROF 模型实质上也是一个各向异性扩散模型，简要证明如下：

**证明.** 根据多元函数情形下的 *Euler-Lagrange* 方程，式(219)求得极值的条件为

$$\lambda(I-I_0)-\nabla\cdot\left(\frac{(\nabla I)}{|\nabla I|}\right)=0 \tag{220}$$

可以将此类比为函数的驻点（凸优化问题），利用等式左端的表达式（类比为一阶导数或梯度）建立梯度下降流：

$$\frac{\partial I}{\partial t}=\operatorname{div}\left(\frac{(\nabla I)}{|\nabla I|}\right)-\lambda(I-I_0) \tag{221}$$

可以看到，上式拥有类似 P-M 方程的形式，而图像梯度 $\nabla I$ 前的系数 $|\nabla I|^{-1}$ 刚好是梯度的单调递减函数，所以跟 P-M 方程一样，ROF 模型也可以抑制梯度方向的扩散。

根据梯度下降流的方法，ROF 模型的代码实现如下，WIN 10 平台，Python 3.7 下调试通过：

```
import cv2
import numpy as np

def denoise(I_init,delta_t=0.5,tau=0.03,max_iteration=40):
    # I_init为原始图像,tau与步长有关，tau为全变差所占的权重，max_iteration为最大迭代次数
    m,n = I_init.shape
    I = I_init

    count = 0
    while(count<max_iteration):
        Iold = I
        GradIx = np.roll(I,-1,axis=1)-I  # 差分Ix
        GradIy = np.roll(I,1,axis = 0)-I  # 差分Iy
        # roll()函数，相当于将矩阵移动，类似于无界模式的贪吃蛇，在最边缘位置的数会移动到对面

        NormGrad = np.maximum(1,np.sqrt(GradIx**2+GradIy**2))  # 为了防止出现分母为0
        GGradIx = np.roll(GradIx/NormGrad,-1,axis=1)-GradIx/NormGrad
        GGradIy = np.roll(GradIy/NormGrad,-1,axis=0)-GradIy/NormGrad
        DivGradI = GGradIx+GGradIy
        step = delta_t*(np.exp(-count*2/(max_iteration))+0.1)  # 有点模拟退火的意思

```

```
22          I = Iold - step*(DivGradI-tau*(Iold-I_init))  # 梯度下降
23          error = np.linalg.norm(I-Iold)/np.sqrt(n*m)
24
25          if (count%10==0):
26              print('step:',step)
27              print('error:',error)
28
29          count+=1
30      return I
31
32  cat = cv2.imread("cat1.jpg", 0)
33  cat = cat.astype("float")  # 图像像素值为ubyte类型，需要转换
34  I = denoise(cat)
35  ROF = I.astype("uint8")  # 需要再转回ubyte类型
36  cv2.imshow("ROF",ROF)  # or use cv2.imwrite to save the figures
37  cv2.waitKey()
```

计算中需注意参数的选取，保证差分格式可以收敛（通过 step 和 error 的数值来监控），最后结果如下图所示：

图 38: ROF 图像去噪结果

ROF 去噪模型的一个弊端在于会带来“**阶梯效应**”，即图像处理后某些区域内灰度相同。区域内灰度相同，表示该区域任意一点其灰度值的一阶导数为 0，这也是 ROF 模型里太过依赖对图像梯度的优化导致的。

另外，可以通过人为构造噪声图像来检验 ROF 去噪的量化效果，只需在函数 denoise 中增加对输入图像与输出图像差值的输出，然后再将其与设定好加入的噪声进行比对分析即可，WIN 10 平台，Python 3.7 下调试通过：

```python
import numpy as np
import matplotlib.pyplot as plt

def denoise(I_init,delta_t=0.5,tau=30,max_iteration=100):
    # I_init为原始图像,tau与步长有关，tau为全变差所占的权重，max_iteration为最大迭代次数
    m,n = I_init.shape
    I = I_init

    count = 0
    while(count<max_iteration):
        Iold = I
        GradIx = np.roll(I,-1,axis=1)-I  # 差分Ix
        GradIy = np.roll(I,1,axis = 0)-I  # 差分Iy
        # roll()函数，相当于将矩阵移动，类似于无界模式的贪吃蛇，在最边缘位置的数会移动到对面

        NormGrad = np.maximum(1,np.sqrt(GradIx**2+GradIy**2))  # 为了防止出现分母为0
        GGradIx = np.roll(GradIx/NormGrad,-1,axis=1)-GradIx/NormGrad
        GGradIy = np.roll(GradIy/NormGrad,-1,axis=0)-GradIy/NormGrad
        DivGradI = GGradIx+GGradIy
        step = delta_t*(np.exp(-count*2/(max_iteration))+0.1)  # 有点模拟退火的意思

        I = Iold - step*(tau*DivGradI+(Iold-I_init))  # 梯度下降
        error = np.linalg.norm(I-Iold)/np.sqrt(n*m)

        if (count%10==0):
            print('step:',step)
            print('error:',error)

        count+=1
    return I,I_init-I

im = np.zeros((500,500))  # 建立一个图形，外黑，中灰，内白
im[100:400,100:400] = 128
im[200:300,200:300] = 255
noise = 30 *np.random.standard_normal((500,500))  # 噪声
```

```
36 im = im + noise
37 U,T = denoise(im)
38 plt.gray()
39
40 plt.subplot(121)
41 plt.imshow(im)
42 plt.subplot(122)
43 plt.imshow(U)
44 plt.show()
45 print(np.corrcoef(T.flatten(),noise.flatten()))#相关度检验
```

首先计算结果如下图所示：

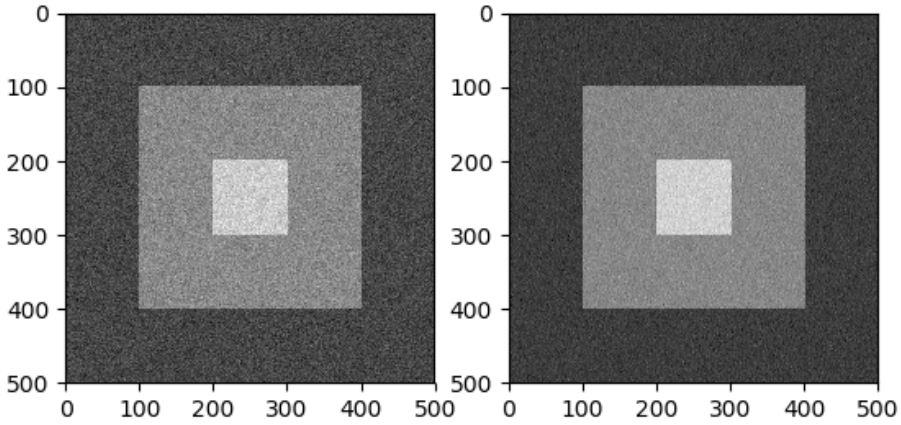



图 39: ROF 图像去噪量化结果

另外该段代码使用了 Numpy 库中的 numpy.corrcoef() 函数，进行 ***Pearson* 乘积矩阵相关系数**的输出。

这里将 T 和 noise 两个矩阵展开成行向量，考察他们的相关性，最终形成的是一个 $2\times2$ 的矩阵。其中矩阵中的行和列分别指代两个矩阵，例如 1 行 1 列的数值代表 T 向量自己跟自己的相关性，所以为 1。为考察 T 和 noise 的相关性，只需看 1 行 2 列或 2 行 1 列的数值即可。最终结果如下：

```
1 [[1.         0.80323148]
2  [0.80323148 1.        ]]
```

80% 多的相关性，说明去噪效果较为明显。

另外，为了避免 ROF 模型带来的图像“阶梯效应”，可以采用高阶偏微分方程进行图像的平滑处理，即将梯度 $\nabla I$ 换成 $\nabla^2 I$，用拉普拉斯算子来优化，这就变成了一个四阶偏微分方程。

### 4.4.3 CDD 模型

前面讲的都是图像去噪相关的模型和算法，其实在图像修复领域所采用的也是类似的方法。早在 01 年就有人将上述的 TV 模型引入到图像修复领域，然而 TV 修复模型的扩散强度 $1/|\nabla I|$ 仅依赖于等照度线的梯度值，而不依赖于等照度线的几何信息，总是倾向于用最短的直线来连接断裂的线性结构物体，所以 TV 模型不能满足**视觉连通性原则**，如下图所示：

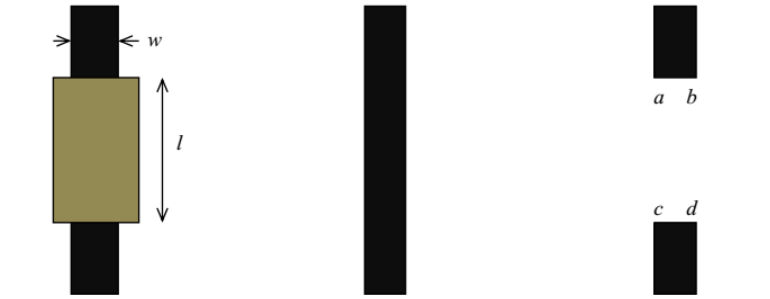



图 40: TV 模型修复结果示意图

对于待修复的图像，按照一般人的视觉感受都应该得到图40(b) 的修复结果，然而当修复区的比例 $l > w$ 时却得到图40(c) 的错误修复结果，因为 TV 修复模型的原理就是连接最短的直线。

本节介绍的 CDD 模型刚好就是为了解决这一问题，该模型利用**曲率**来控制扩散强度，当图像修复过程到达图40(c) 状态时（大曲率处），增大扩散强度，从而使图像信息进一步扩散，而在小曲率处扩散逐渐消失，达到满足连通性效果。

例如在图40(c) 中，拐角处 $a, b, c, d$ 四点的曲率为 $\infty$，而在理想输出图40(b) 中，边缘上各点的曲率为 0。因此可以用曲率调节扩散强度。

具体 CDD 修复模型如下：

$$\begin{aligned} &\frac{\partial \mathrm{I}}{\partial t} = \nabla \cdot \left( \frac{g(|k|)}{|\nabla I|} \nabla I \right) \\ &I = I^0 \end{aligned} \tag{222}$$

其中 $I_0$ 表示原始图像。

原论文中曲率的计算式为

$$\kappa = \nabla \cdot \left( \frac{\nabla I}{|\nabla I|} \right) \tag{223}$$

**这很明显是错误的！**用的是类似平均曲率的表达式，但却是隐式的，很明显这里应该用显式。但大家都这样用，将错就错吧.. 确实也差不太多。

另外原论文中使用

$$g(s) = s^p, s > 0, p \geq 1 \tag{224}$$

在实际操作中，需要先构造一个 mask，然后利用 mask 作为阈值对原图像进行各向异性扩散，这就是完整的 CDD 修复过程。

具体在计算的时候，论文中采用的是**半点格式离散化的中心差分法**，为保持统一，这里也采用该方法构建数值格式。

在以图像中某一像素为中点的四邻域中，由如下关系：

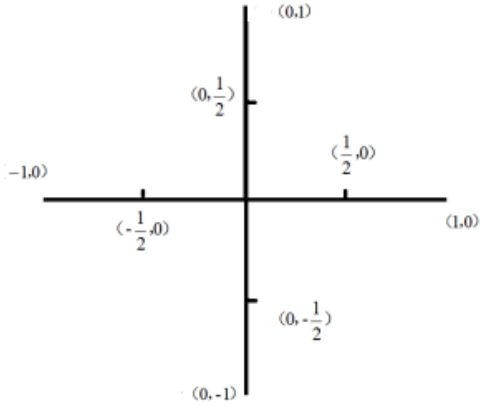



图 41: 半点格式离散化中心差分法（CDD）

令

$$\boldsymbol{j} = -\frac{g(|k|)}{|\nabla I|}\nabla I \tag{225}$$

则数值迭代式如下：

$$I^{n+1} = I^n - \Delta t\left(\nabla \cdot j^n\right) \tag{226}$$

其中上指标表示迭代次数，$\triangle t$ 为时间步长。

由于 $\boldsymbol{j}$ 为向量，如果令

$$\boldsymbol{j} = (j^1, j^2) \tag{227}$$

其中括号里的上指标表示向量的分量。则可以推导出：

$$\nabla \cdot \boldsymbol{j}_{(0,0)} = \frac{j^1_{(\frac{1}{2},0)} - j^1_{(-\frac{1}{2},0)}}{h} + \frac{j^2_{(0,\frac{1}{2})} - j^2_{(0,-\frac{1}{2})}}{h} \tag{228}$$

其中 $j$ 的下指标表示 $j$ 的位置，$h$ 为采点步长，这里刚好为一个像素。

在 $\boldsymbol{j}$ 中，有 $\nabla I$ 和 $\kappa$ 两个量，下面分别进行构造：

- 在半点处的梯度求法如下，以 $(1/2,0)$ 处的梯度为例：

$$\begin{aligned}\nabla I_{(\frac{1}{2},0)} &= \left(\left.\frac{\partial I}{\partial x}\right|_{(\frac{1}{2},0)}, \left.\frac{\partial I}{\partial y}\right|_{(\frac{1}{2},0)}\right) = \left(\frac{I_{(1,0)}-I_{(0,0)}}{h}, \frac{I_{(\frac{1}{2},1)}-I_{(\frac{1}{2},-1)}}{2h}\right) \\ &= \left(\frac{I_{(1,0)}-I_{(0,0)}}{h}, \frac{I_{(1,1)}+I_{(0,1)}-I_{(1,-1)}-I_{(0,-1)}}{4h}\right)\end{aligned} \tag{229}$$

- 曲率的数值构造方式类似，由于

$$\kappa = \nabla\cdot\left[\frac{\nabla I}{|\nabla I|}\right] = \frac{\partial}{\partial x}\left[\frac{I_x}{|\nabla I|}\right] + \frac{\partial}{\partial y}\left[\frac{I_y}{|\nabla I|}\right] \tag{230}$$

例如第一项可以如下构造：

$$\frac{\partial}{\partial x}\left[\frac{I_x}{|\nabla I|}\right]_{(\frac{1}{2},0)} \simeq \left[\frac{I_x}{|\nabla I|}\right]_{(1,0)} - \left[\frac{I_x}{|\nabla I|}\right]_{(0,0)} \tag{231}$$

里面的新项例如 $(I_x/|\nabla I|)_{(1,0)}$ 直接使用中心差分公式即可，比如该项可以写成

$$\left[\frac{I_x}{|\nabla I|}\right]_{(1,0)} \simeq \frac{I_{(2,0)}-I_{(0,0)}}{\sqrt{\left(I_{(2,0)}-I_{(0,0)}\right)^2+\left(I_{(1,1)}-I_{(1,-1)}\right)^2}} \tag{232}$$

而第二项就麻烦了，展开后异常繁琐，但计算过程清晰明了，并不复杂，这里从略。

这就是整体的数值构造方法。

为揭示 CDD 算法的优势所在，这里把 TV 模型的图像修复原理也简要说明下。其基本的算法就是利用 ROF 模型（没有曲率参数），其他的部分基本保持一致。这里将原论文中对数值格式的定义再简要描述下，采用的也是半点格式离散化的中心差分法，但用的是方位来进行表示的，如下图所示：

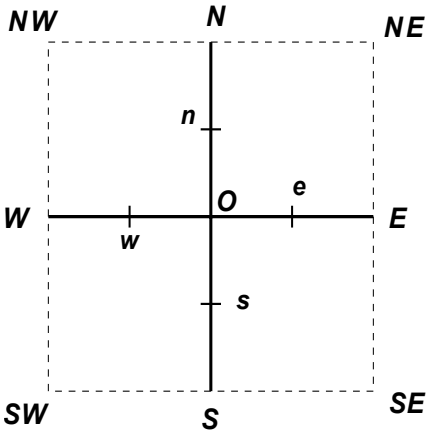



图 42: 半点格式离散化中心差分法（TV）

由于相比于 CDD 模型，TV 模型没有曲率参数，所以更为简便。令

$$\boldsymbol{v} = \left(v^1, v^2\right) = \nabla I/|\nabla I| \tag{233}$$

这里的 $\boldsymbol{v}$ 就相当于 CDD 模型中的 $\boldsymbol{j}$。接下来对其求散度，与 CDD 模型类似：

$$\begin{aligned}\nabla\cdot\boldsymbol{v}&=\frac{\partial v^1}{\partial x}+\frac{\partial v^2}{\partial y}\\&\simeq\frac{v_e^1-v_w^1}{h}+\frac{v_n^2-v_s^2}{h}\end{aligned}\tag{234}$$

其中的变量，以 $v_e^1$ 为例，求解方式如下：

$$v_e^1=\frac{1}{|\nabla I_e|}\left[\frac{\partial I}{\partial x}\right]_e\simeq\frac{1}{|\nabla I_e|}\frac{I_E-I_O}{h}\tag{235}$$

其中，

$$|\nabla I_e|\simeq\frac{1}{h}\sqrt{(I_E-I_O)^2+\left[\left(I_{NE}+I_N-I_S-I_{SE}\right)/4\right]^2}\tag{236}$$

在原论文中，上式被简化成如下形式：

$$|\nabla I_e|\simeq\frac{1}{h}\sqrt{(I_E-I_O)^2+\left[\left(I_{NE}-I_{SE}\right)/2\right]^2}\tag{237}$$

并表示实验表明，针对修复 sharp 边缘有时候该式更加行之有效。

TV 模型图像修复使用的就是 ROF 模型中的式(220)，根据上述离散思路，可进一步设计计算方式：

令

$$\Lambda_O=\{E,N,W,S\}\tag{238}$$

则式(220)可离散化为

$$0=\sum_{P\in\Lambda_O}\frac{1}{\left|\nabla I_p\right|}\left(I_O-I_P\right)+\lambda(O)\left(I_O-I_O^0\right)\tag{239}$$

该式很巧妙，刚好可以用一种统一的方式来书写，方便接下来的算法设计。

令

$$\begin{aligned}w_P&=\frac{1}{\left|\nabla I_p\right|},\quad P\in\Lambda_O\\h_{OP}&=\frac{w_P}{\sum_{Q\in\Lambda_O}w_Q+\lambda(O)}\\h_{OO}&=\frac{\lambda(O)}{\sum_{Q\in\Lambda_O}w_Q+\lambda(O)}\end{aligned}\tag{240}$$

则式239可以最终简化为

$$I_O = \sum_{P \in \Lambda_O} h_{OP} I_P + h_{OO} I_O^0 \tag{241}$$

在原论文中，并没有采取梯度下降的方式来构建迭代格式，而是直接使用上式来迭代：

$$I_O^{(n)} = \sum_{P \in \Lambda_O} h_{OP}^{(n-1)} I_P^{(n-1)} + h_{OO}^{(n-1)} I_O^{(n-1)} \tag{242}$$

其中，$h^{(n-1)} = h(I^{(n-1)})$。

下面用代码来实现，首先是 TV 模型，WIN 10 平台，Python 3.7 下调试通过：

```
import cv2
import numpy as np

I = cv2.imread("inpainting.png", 0)
I = I.astype("float")  # 图像像素值为ubyte类型，需要转换
m,n = I.shape
mask = np.zeros((m, n))

# 定义mask，需要根据修复地方的特征进行调整
for i in range(m):
    for j in range(n):
        if I[i,j]>5:
            mask[i,j]=255
        else:
            mask[i,j]=0

mask = mask.astype("uint8")  # 需要再转回ubyte类型
cv2.imshow("inpaintmask",mask)  # or use cv2.imwrite to save the figures

l = 0.2
a = 0.5  # 避免分母为0
count = 0
while(count<300):
    for i in range(1,m-1):
        for j in range(1,n-1):
            if mask[i+1,j]==0 or mask[i-1,j]==0 or mask[i,j-1]==0 or mask[i,j
                +1]==0:  # 如果当前像素是被污染的像素，则进行处理
                absIe=np.sqrt((I[i+1,j]-I[i,j])**2+((I[i+1,j+1]-I[i+1,j-1])/2)
                    **2)
```

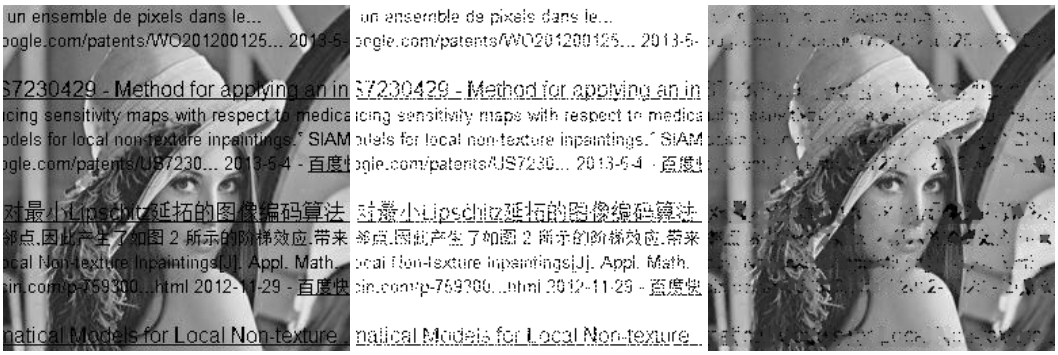
```
                absIw=np.sqrt((I[i,j]-I[i-1,j])**2+((I[i-1,j+1]-I[i-1,j-1])/2)
                    **2)
                absIn=np.sqrt(((I[i+1,j+1]-I[i-1,j+1])/2)**2+(I[i,j+1]-I[i,j])
                    **2)
                absIs=np.sqrt(((I[i+1,j-1]-I[i-1,j-1])/2)**2+(I[i,j]-I[i,j-1])
                    **2)
                we = 1/np.sqrt(absIe**2+a**2)
                ww = 1/np.sqrt(absIw**2+a**2)
                wn = 1/np.sqrt(absIn**2+a**2)
                ws = 1/np.sqrt(absIs**2+a**2)
                hoe=we/((we+ww+wn+ws)+1)
                how=ww/((we+ww+wn+ws)+1)
                hon=wn/((we+ww+wn+ws)+1)
                hos=ws/((we+ww+wn+ws)+1)
                hoo=1/((we+ww+wn+ws)+1)

                I[i,j] = hoe*I[i+1,j]+how*I[i-1,j]+hon*I[i,j+1]+hos*I[i,j-1]+
                    hoo*I[i,j]

    count=count+1

I = I.astype("uint8")  # 需要再转回ubyte类型
cv2.imshow("inpaintI",I)  # or use cv2.imwrite to save the figures
cv2.waitKey()
```

这里没有选择之前的猫片，因为细节纹理太多，修复起来能累死人，而且效果也不好。首先采用经典的 luna 加噪声图进行修复，迭代 300 次效果如下：



(a) inpainting (b) mask (c) CDD

图 43: TV 图像修复结果（luna）

接下来尝试对圆圈图进行修复，该修复由于边界较宽而较为困难，需要较高的迭代

次数，这里尝试 1000 次迭代：

(a) inpainting (b) mask (c) CDD

图 44: TV 模型图像修复结果（circle）

在修复过程中发现，迭代次数会影响修复效果，中间的白条会随着迭代次数的增加而不断缩短，这里迭代 1000 次也只能修复到如此地步。

下面考虑 CDD 模型的实现，相比 TV 模型多了曲率定义的部分，而且原论文中算法设计部分也并非完全与 TV 模型一致。另外对于曲率的定义，公式实在是太冗长了，代码写的过于繁琐，此时再纠结于具体的实现已是得不偿失。

除此之外，网上也有现成的 CDD 模型的代码实现（一般是基于 Matlab），但大多代码写的都是错的（甚至有的就是 TV 模型在挂羊头卖狗肉，有的连 TV 模型写的也是错的..），还有的挂在 CSDN 上，需要天价购买..

之前有尝试过自己重新写，后来发现实在写不动，算了，这里贴一下论文里的图以及网上的实现图作为简单示意即可。

首先是原论文中的实现结果，如下图，可以很明显看到图像修复效果明显，细节纹理都保护得很好。

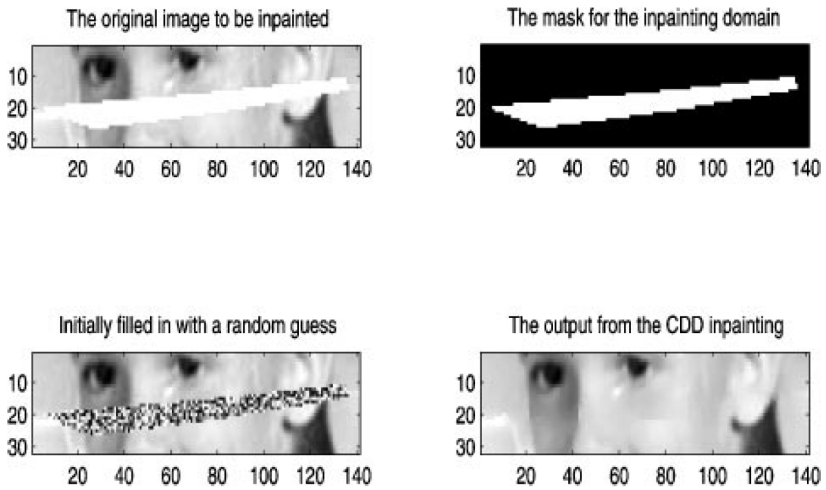



图 45: CDD 模型原论文图像修复结果

接下来是网上看到的实现结果，实话说这个结果真的有被震惊到了。但评论区里大多数人都表示不能复现到此种效果，也是未解之谜。

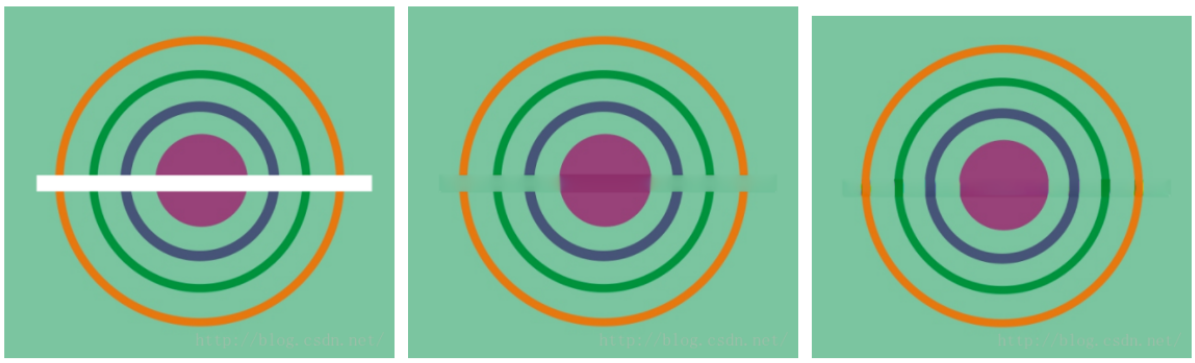



图 46: CDD 模型图像修复结果

一般来说，CDD 图像修复最大的弊端还是在于修复时间太长，但通过引入**图像曲率**满足了“连通性原理”，是该论文最大的亮点。

# 5 习题集

**例 5.1 (*Frenet-Serret* 标架运动方程)** 试证明参数曲线的 *Frenet-Serret* 标架运动方程：

$$\begin{aligned}\boldsymbol{t}'(s) &= \kappa(s)\cdot\boldsymbol{n}(s)\\ \boldsymbol{n}'(s) &= -\kappa(s)\boldsymbol{t}(s)-\tau(s)\cdot\boldsymbol{b}(s)\\ \boldsymbol{b}'(s) &= \tau(s)\cdot\boldsymbol{n}(s)\end{aligned}\tag{243}$$

**例 5.2 (* 曲面曲率与法向量的关系)** 试证明参数曲面的平均曲率和高斯曲率与曲面单位法向量存在如下恒等式关系：

$$\begin{aligned}H &= -\frac{1}{2}\operatorname{div}(\boldsymbol{N})\\ K &= \frac{1}{2}\operatorname{div}[\boldsymbol{N}\cdot\operatorname{div}(\boldsymbol{N})+\boldsymbol{N}\times\operatorname{curl}(\boldsymbol{N})]\end{aligned}\tag{244}$$

# A 数学基础

## A.1 特殊函数

**定义 A.1 (狄拉克（Dirac）函数)** 一维狄拉克（Dirac）函数定义为 $\delta(x)$，满足

$$\begin{cases} \int_{-\infty}^{\infty} \delta(x)\mathrm{d}x = 1 \\ \delta(x) = 0, \qquad x \neq 0 \end{cases} \tag{245}$$

Dirac 函数是一种广义函数，它在自变量不为零处全为零，而在自变量为零处又没有明确的数值定义，但在整个定义域区间内却满足积分为 1。

Dirac 函数具有如下基本数学性质：

- Dirac 函数为偶函数，满足 $\delta(x) = \delta(-x)$，进一步推广得到：$\delta(x - x_0) = \delta(x_0 - x)$。其中，$x_0$ 为一维空间的实数取值；
- 对于任意一元函数 $f(x)$，必有 $f(x)\delta(x) = f(0)\delta(x)$，进一步得到如下 Dirac 函数的“筛选抽取”性质：

$$f(x_0) = \int_{-\infty}^{\infty} f(x)\delta(x - x_0)\mathrm{d}x = \int_{-\infty}^{\infty} f(x)\delta(x_0 - x)\mathrm{d}x \tag{246}$$