图像处理实验报告

210810508-彭珂

May 19, 2024

Contents

1 CV 模型

1.1 模型介绍

CV 模型是一个非常著名的基于区域的活动轮廓模型，相比其它活动轮廓模型，CV 模型具有如下几个优点。CV 模型是基于 Mumford-Shah 分割技巧和水平集方法，而不是基于边界函数使得演变曲线停在想要的边缘上。而且即使初始图像含有噪声，也不需要光滑初始图像，边缘的位置仍可以很好地被检测和保留。CV 模型可以检测不是由梯度定义的边缘或者是非常光滑的边缘，而对于这些边缘，经典的活动轮廓模型是不适用的。最后 CV 模型可以仅从一条初始曲线自动地检测内部轮廓，而且初始曲线的位置不必环绕待检测的物体，可以在图像的任意位置。这是 CV 模型一个非常重要的优点。

但是 CV 模型主要对于具有同质区域的图像有较为显著的分割效果，而不可以处理具有不均匀强度的图像。在 CV 模型里，常量 c1 和 c2 用来近似区域 inside(C) 和 outside(C) 内的图像强度，很显然，如果原始图像的图像强度在 inside(C) 或者 outside(C) 内并不是均匀的，这种全局的拟合将会是不准确的。也就是说 CV 模型仅考虑图像强度的全局信息，而不包含图像任何局部强度信息，而图像的局部强度信息对于具有图像强度不均匀性质的图像分割是至关重要的，因此 CV 模型不可以分割具有图像强度不均匀性质的图像。

1.2 数学原理

CV 模型的能量泛函为:

$$
\begin{aligned}
F^{CV}(c_1,c_2,C)=&\lambda_1\int_{inside(C)}\left|u_0(x,y)-c_1\right|^2\mathrm{d}x\mathrm{d}y\\
&+\lambda_2\int_{outside(C)}\left|u_0(x,y)-c_2\right|^2\mathrm{d}x\mathrm{d}y\\
&+\nu|C|
\end{aligned}
$$

其中 ν,λ_1,λ_2 为参数,$\nu|C|$ 项为边界长度项,$\lambda_1\int_{inside(C)}\left|u_0(x,y)-c_1\right|^2\mathrm{d}x\mathrm{d}y$,
$\lambda_2\int_{outside(C)}\left|u_0(x,y)-c_2\right|^2\mathrm{d}x\mathrm{d}y$ 项为数据拟合项。

我们设 ϕ 为一个水平集函数，使得 $\phi(x,y)=0$ 为边界 C，$\phi(x,y)>0$ 为内部，$\phi(x,y)<0$ 为外部。则 CV 模型的能量泛函可以表示为:

$$
\begin{aligned}
F^{CV}(c_1,c_2,\phi)=&\lambda_1\int_{\Omega}\left|u_0(x,y)-c_1\right|^2H(\phi(x,y))\,\mathrm{d}x\mathrm{d}y\\
&+\lambda_2\int_{\Omega}\left|u_0(x,y)-c_2\right|^2[1-H(\phi(x,y))]\,\mathrm{d}x\mathrm{d}y\\
&+\nu\int_{\Omega}\left|\nabla H(\phi(x,y))\right|\mathrm{d}x\mathrm{d}y
\end{aligned}
$$

其中 ν,λ_1,λ_2 为参数,Ω 为图像区域,H 为 Heaviside 函数。$\nu\int_{\Omega}\left|\nabla H(\phi(x,y))\right|\mathrm{d}x\mathrm{d}y$ 为边界长度项，$\lambda_1\int_{\Omega}\left|u_0(x,y)-c_1\right|^2H(\phi(x,y))\,\mathrm{d}x\mathrm{d}y,\lambda_2\int_{\Omega}\left|u_0(x,y)-c_2\right|^2[1-H(\phi(x,y))]\,\mathrm{d}x\mathrm{d}y$ 项为数据拟合项。为了极小化能量泛函，我们采用交替极小化的方式，即轮流优化 c_1,c_2,ϕ。

在确定一个初始值之后，我们轮流优化 c_1,c_2,ϕ

首先优化 c_1:

$$
\begin{aligned}
&\frac{\partial}{\partial c_1}F^{CV}=0\\
\Rightarrow&\frac{\partial}{\partial c_1}\lambda_1\int_{\Omega}\left|u_0(x,y)-c_1\right|^2H(\phi(x,y))\,\mathrm{d}x\mathrm{d}y=0\\
\Rightarrow&\frac{\partial}{\partial c_1}\int_{\Omega}\left(u_0^2(x,y)-2c_1u_0(x,y)+c_1^2\right)H(\phi(x,y))\,\mathrm{d}x\mathrm{d}y=0\\
\Rightarrow&-2\int_{\Omega}u_0(x,y)H(\phi(x,y))\,\mathrm{d}x\mathrm{d}y+2c_1\int_{\Omega}H(\phi(x,y))\,\mathrm{d}x\mathrm{d}y=0\\
\Rightarrow&c_1=\frac{\int_{\Omega}u_0(x,y)H(\phi(x,y))\,\mathrm{d}x\mathrm{d}y}{\int_{\Omega}H(\phi(x,y))\,\mathrm{d}x\mathrm{d}y}
\end{aligned}
$$

故令:

$$
c_1=\frac{\int_{\Omega}u_0(x,y)H(\phi(x,y))\,\mathrm{d}x\mathrm{d}y}{\int_{\Omega}H(\phi(x,y))\,\mathrm{d}x\mathrm{d}y}
$$

然后优化 c_2:

$$
\begin{aligned}
&\frac{\partial}{\partial c_2}F^{CV}=0\\
\Rightarrow&\frac{\partial}{\partial c_2}\lambda_2\int_\Omega|u_0(x,y)-c_1|^2[1-H(\phi(x,y))]\,\mathrm{d}x\mathrm{d}y=0\\
\Rightarrow&\frac{\partial}{\partial c_2}\int_\Omega\left(u_0^2(x,y)-2c_2u_0(x,y)+c_2^2\right)[1-H(\phi(x,y))]\,\mathrm{d}x\mathrm{d}y=0\\
\Rightarrow&-2\int_\Omega u_0(x,y)[1-H(\phi(x,y))]\,\mathrm{d}x\mathrm{d}y+2c_2\int_\Omega[1-H(\phi(x,y))]\,\mathrm{d}x\mathrm{d}y=0\\
\Rightarrow&c_2=\frac{\int_\Omega u_0(x,y)[1-H(\phi(x,y))]\,\mathrm{d}x\mathrm{d}y}{\int_\Omega[1-H(\phi(x,y))]\,\mathrm{d}x\mathrm{d}y}
\end{aligned}
$$

故令:

$$
c_2=\frac{\int_\Omega u_0(x,y)[1-H(\phi(x,y))]\,\mathrm{d}x\mathrm{d}y}{\int_\Omega[1-H(\phi(x,y))]\,\mathrm{d}x\mathrm{d}y}
$$

最后优化 ϕ:

令:

$$
\begin{aligned}
f(\boldsymbol{x},\phi(\boldsymbol{x}),\nabla\phi(\boldsymbol{x}))=&\lambda_1|u_0(\boldsymbol{x})-c_1|^2H(\phi(\boldsymbol{x}))\\
&+\lambda_2|u_0(\boldsymbol{x})-c_2|^2[1-H(\phi(\boldsymbol{x}))]\\
&+\nu\delta(\phi(\boldsymbol{x}))|\nabla\phi(\boldsymbol{x})|
\end{aligned}
$$

则:

$$
F^{CV}(\phi)=\int_\Omega f(\boldsymbol{x},\phi(\boldsymbol{x}),\nabla\phi(\boldsymbol{x}))\,\mathrm{d}\boldsymbol{x}
$$

故 F^{CV} 在 ϕ 处的梯度:

$$
\begin{aligned}
\frac{\delta}{\delta\phi}F^{CV}=&\frac{\partial f}{\partial\phi}-\nabla\cdot\frac{\partial f}{\partial\nabla\phi}\\
=&\lambda_1|u_0(\boldsymbol{x})-c_1|^2\delta(\phi(\boldsymbol{x}))\\
&-\lambda_2|u_0(\boldsymbol{x})-c_1|^2\delta(\phi(\boldsymbol{x}))\\
&+\nabla\cdot\left(\nu\delta(\phi(\boldsymbol{x}))\frac{\nabla\phi(\boldsymbol{x})}{|\nabla\phi(\boldsymbol{x})|}\right)\\
=&\delta(\phi(\boldsymbol{x}))\left[\lambda_1\left(u_0(\boldsymbol{x})-c_1\right)^2-\lambda_2\left(u_0(\boldsymbol{x})-c_1\right)^2-\nu\nabla\cdot\left(\frac{\nabla\phi(\boldsymbol{x})}{|\nabla\phi(\boldsymbol{x})|}\right)\right]
\end{aligned}
$$

然后 ϕ 向负梯度方向更新。

1.3 代码实现

为了用程序实现 CV 模型，我们定义一个 CV_model 类，把 c_1, c_2, ϕ 作为其类属性，然后在其内部定义一个 fit 方法，用来优化 c_1, c_2, ϕ。在使用要处理的图像实例化一个 CV_model 类之后，我们调用 fit 方法之后，就会不断更新迭代 c_1, c_2, ϕ，直到达到设置的迭代次数上限。

具体代码实现如下:

```
import math
import numpy as np
from PIL import Image
```

```python
import matplotlib.pyplot as plt
from scipy.io import loadmat
from scipy.signal import convolve2d

epsilon = 0.01
sigma = 25

def set_sigma(a: float):
    global sigma
    sigma = a
def H_pointwise(x: float) -> float:
    return (1/2) * (1 + (2/math.pi) * math.atan(x/epsilon))

H = np.vectorize(H_pointwise)

def delta_pointwise(x: float) -> float:
    return (1/math.pi)*(epsilon/(x**2 + epsilon**2))

def gradient_length(image: np.array) -> float:
    gradient = np.gradient(image)
    return np.sqrt(gradient[0]**2 + gradient[1]**2)

delta = np.vectorize(delta_pointwise)

def K_pointwise(u: np.array) -> float:
    return np.exp(-(np.linalg.norm(u)**2)/(2*sigma**2))/(2*math.pi*
        sigma**2)

K = np.vectorize(K_pointwise)

def divergence_after_normalized_gradient(image: np.array) -> np.array:
    gradient = np.gradient(image)
    magnitude = np.sqrt(gradient[0]**2 + gradient[1]**2)
    magnitude[magnitude == 0] = 0.01
    normalized_gradient = gradient / magnitude
    divergence = np.gradient(normalized_gradient[0])[0] + np.gradient(
        normalized_gradient[1])[1]

    return divergence

def laplace(image: np.array) -> np.array:
```

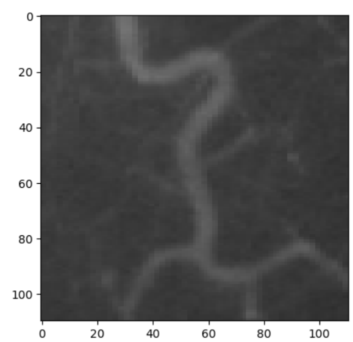
```
    gradient = np.gradient(image)
    laplacian = np.gradient(gradient[0])[0] + np.gradient(gradient[1])[1]
    return laplacian

class CV_model:
    def __init__(self, figure: np.array, c1: float = 0, c2: float = 255, lambda1: float = 1, lambda2: float = 1, nu: float = 1):
        self.figure = figure
        self.c1 = c1
        self.c2 = c2
        self.phi = np.ones_like(self.figure)
        self.phi[0, :] = self.phi[-1, :] = self.phi[:, 0] = self.phi[:, -1] = -1
        self.lambda1 = lambda1
        self.lambda2 = lambda2
        self.nu = nu

    def update_phi(self, learning_rate: float):
        flow  = delta(self.phi) * (self.nu * divergence_after_normalized_gradient(self.phi) - self.lambda1 * (self.figure - self.c1)**2 + self.lambda2 * (self.figure - self.c2)**2)
        self.phi = self.phi + learning_rate * flow

    def update_c1(self, H_phi: np.array):
        self.c1 = np.sum(self.figure * H_phi) / np.sum(H_phi)

    def update_c2(self, H_phi: np.array):
        self.c2 = np.sum(self.figure * (1 - H_phi)) / np.sum(1 - H_phi)

    def fit(self, learning_rate: float, max_iter: int = 100):

        for i in range(max_iter):
            H_phi = H(self.phi)
            self.update_c1(H_phi)
            self.update_c2(H_phi)
            self.update_phi(learning_rate)

    def draw(self, i: int):
```

```python
        colored_figure = np.repeat(self.figure[:, :, np.newaxis], 3, axis=2)
        plt.imshow(colored_figure, cmap='gray')
        plt.contour(self.phi, [0], colors='r')
        plt.savefig(fname=f"out/img/output{i+1}_CV.png", bbox_inches="tight", pad_inches=0.1)
        plt.show()

# 合成图像强度不均匀的图像
def generate_synthetic_image() -> np.array:
    Img = np.zeros((101, 101))
    for i in range(101):
        for j in range(101):
            Img[i, j] = 30 * (1 + np.sin(0.01 * np.pi * (i - j)))
    Img[50, 50] = 100
    for i in range(101):
        for j in range(101):
            if np.sqrt((i - 50) ** 2 + (j - 50) ** 2) <= (35 + 7 * np.cos(8 * np.arctan((j - 50)*(i - 50)))):
                Img[i, j] = 100 * (1 + 0.2 * np.sin(0.01 * np.pi * (i - j)))
    return Img

def get_img()->list:
    imgs = []
    for i in range(1, 6):
        img = Image.open(f"fig/{i}.bmp")
        img = np.array(img)
        imgs.append(img)
    img = Image.open("fig/6.png")
    img = np.array(img)
    img_gray = img[:, :, 0]
    imgs.append(img_gray)
    data = loadmat('fig/brain_img75.mat')
    img = data['img']
    imgs.append(img)
    img = Image.open("fig/myBrain_axial.bmp")
    img = np.array(img)
    imgs.append(img)
    img = generate_synthetic_image()/120
    imgs.append(img)
```

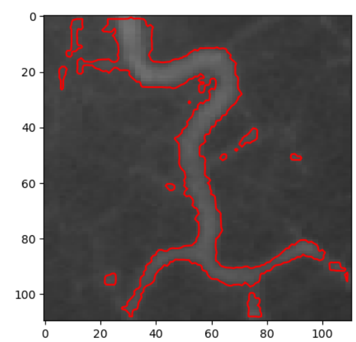
```python
    return imgs



if __name__ == '__main__':
    for i, img in enumerate(imgs):
        if i != 8:
            model = CV_model(img, lambda1=1, lambda2=1, nu=1)
        else:
            model = CV_model(img, lambda1=256, lambda2=256, nu=1)
        model.fit(0.1, 500)
        model.draw(i)
```

1.4 实验结果

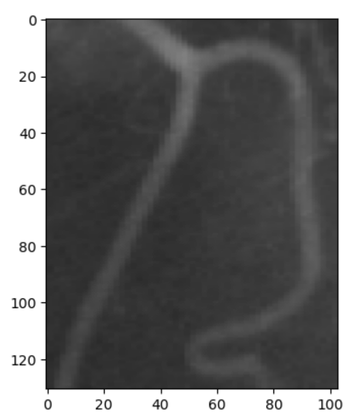
CV 模型的实验结果如下：



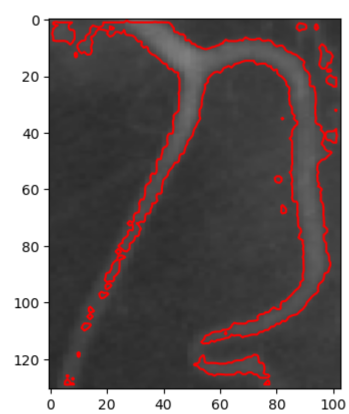
(1) 原图



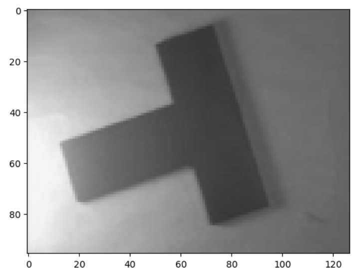
(2) CV 模型处理后



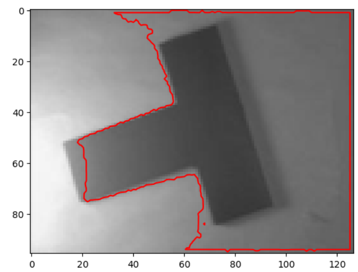
(3) 原图



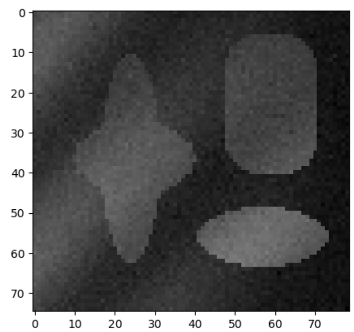
(4) CV 模型处理后



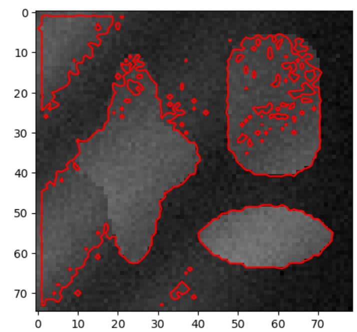
(5) 原图



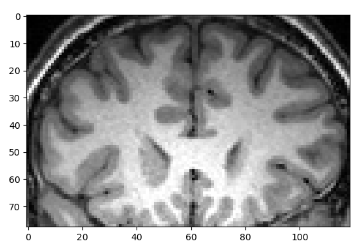
(6) CV 模型处理后



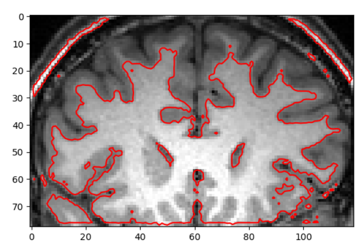
(7) 原图



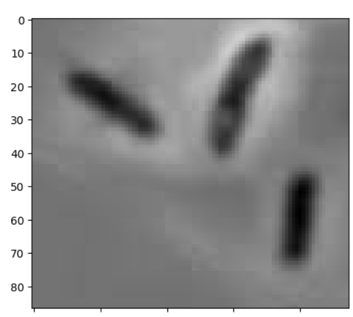
(8) CV 模型处理后



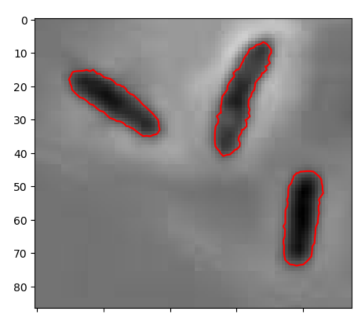
(9) 原图



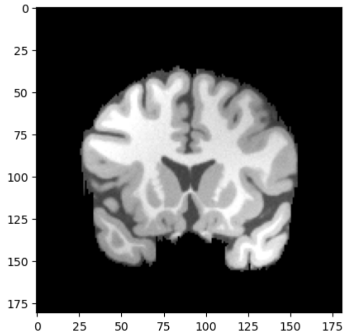
(10) CV 模型处理后



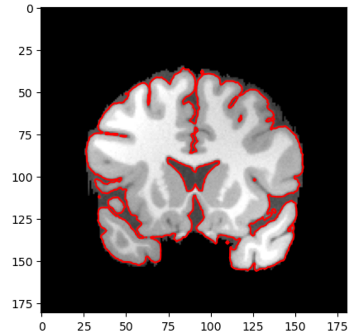
(11) 原图



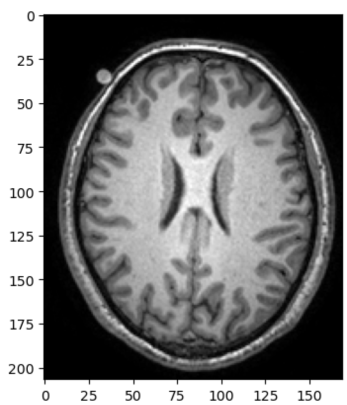
(12) CV 模型处理后



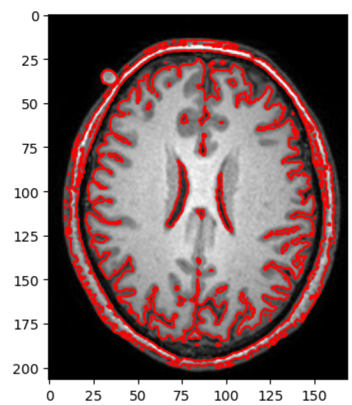
(13) 原图



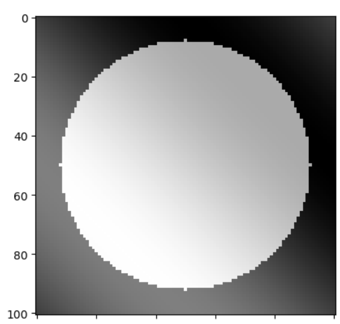
(14) CV 模型处理后



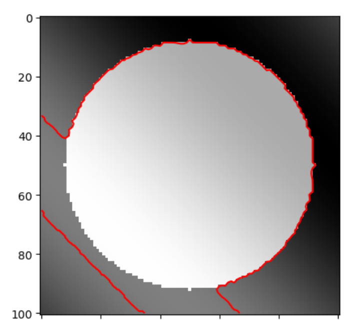
(15) 原图



(16) CV 模型处理后



(17) 原图



(18) CV 模型处理后

1.5 结论与讨论

可以看到，对于图像强度均匀的图片的分割，CV 模型具有优良的效果，但是对于图像强度不均匀的图片，比如上面的最后一张图，CV 模型就不能给出令人满意的结果。

2 RSF 模型

2.1 模型介绍

不同于 CV 模型用两个常量 c1 和 c2 来近似轮廓线 C 两侧的区域 inside(C) 和 outside(C) 内的图像强度。RSF 模型用两个拟合函数 $f_1(x)$ 和 $f_2(x)$ 来拟合 C 两侧的区域内的图像强度。注意到 f1 和 f2 的值是随着中心点 x 的变化而变化的。f_1 和 f_2 的这种空间变化性质使得 RSF 模型从本质上区别于 CV 模型，这种空间变化性质来源于空间变化的核函数 K_σ 的局部化性质。RSF 模型的区域可伸缩性也来源于核函数 K_σ，尺度参数 σ 可以控制局部区域的大小，从小的邻域到整个定义域，这样就可以在一个可控制尺度的区域内充分利用图像的强度信息用于引导活动轮廓线的移动。RSF 模型通过使用核函数 K_σ，充分利用图像的局部强度信息，因此该模型可以分割具有图像强度不均匀性质的图像，而且对于一些具有弱边界的物体如血管等的分割有很好的效果。但是 RSF 模型仅仅利用图像的局部信息可能会导致能量泛函的局部极小，因此 RSF 模型的分割结果会更加依赖于轮廓线的初始化。此外，因为 RSF 模型是非凸的，这也是导致局部极小解存在的一个原因。

2.2 数学原理

RSF 模型的能量泛函为：

$$
\begin{aligned}
F^{RSF}(f_1, f_2, C) =&\lambda_1 \int_\Omega \left(\int_{inside(C)} K(\boldsymbol{x}-\boldsymbol{y}) |u_0(\boldsymbol{y}) - f_1(\boldsymbol{x})|^2 \,\mathrm{d}\boldsymbol{y} \right) \mathrm{d}\boldsymbol{x} \\
&+\lambda_2 \int_\Omega \left(\int_{outside(C)} K(\boldsymbol{x}-\boldsymbol{y}) |u_0(\boldsymbol{y}) - f_2(\boldsymbol{x})|^2 \,\mathrm{d}\boldsymbol{y} \right) \mathrm{d}\boldsymbol{x} \\
&+\nu |C|
\end{aligned}
$$

其中 $\nu, \lambda_1, \lambda_2$ 为参数,$\nu|C|$ 项为边界长度项,$\lambda_1 \int_\Omega \left(\int_{inside(C)} K(\boldsymbol{x}-\boldsymbol{y}) |u_0(\boldsymbol{y}) - f_1(\boldsymbol{x})|^2 \,\mathrm{d}\boldsymbol{y} \right) \mathrm{d}\boldsymbol{x}$, $\lambda_2 \int_\Omega \left(\int_{outside(C)} K(\boldsymbol{x}-\boldsymbol{y}) |u_0(\boldsymbol{y}) - f_2(\boldsymbol{x})|^2 \,\mathrm{d}\boldsymbol{y} \right) \mathrm{d}\boldsymbol{x}$ 项为数据拟合项。

我们设 ϕ 为一个水平集函数，使得 $\phi(\boldsymbol{x}) = 0$ 为边界 C，$\phi(\boldsymbol{x}) > 0$ 为内部，$\phi(\boldsymbol{x}) < 0$ 为外部。则 RSF 模型的能量泛函可以表示为：

$$
\begin{aligned}
F^{RSF}(f_1, f_2, \phi) =&\mathcal{E}^{RSF}(f_1, f_2, \phi) + \nu \mathcal{L}^{RSF}(\phi) + \mu \mathcal{P}^{RSF}(\phi) \\
=&\lambda_1 \int_\Omega \left(\int_\Omega K(\boldsymbol{x}-\boldsymbol{y}) |u_0(\boldsymbol{y}) - f_1(\boldsymbol{x})|^2 H(\phi(\boldsymbol{y})) \,\mathrm{d}\boldsymbol{y} \right) \mathrm{d}\boldsymbol{x} \\
&+\lambda_2 \int_\Omega \left(\int_\Omega K(\boldsymbol{x}-\boldsymbol{y}) |u_0(\boldsymbol{y}) - f_2(\boldsymbol{x})|^2 [1 - H(\phi(\boldsymbol{y}))] \,\mathrm{d}\boldsymbol{y} \right) \mathrm{d}\boldsymbol{x} \\
&+\nu \int_\Omega |\nabla H(\phi(\boldsymbol{x}))| \,\mathrm{d}\boldsymbol{x} \\
&+\mu \int_\Omega \frac{1}{2} \left(|\nabla \phi(\boldsymbol{x})| - 1 \right)^2 \,\mathrm{d}\boldsymbol{x}
\end{aligned}
$$

其中 $\nu, \lambda_1, \lambda_2, \mu$ 为参数,$\nu\mathcal{L}^{RSF}(\phi)$ 项为边界长度项,$\mathcal{E}^{RSF}(f_1, f_2, \phi)$ 项为数据拟合项,$\mu\mathcal{P}^{RSF}(\phi)$ 为水平集正则项。为了极小化能量泛函，我们采用交替极小化的方式，即轮流优化 f_1, f_2, ϕ。

在确定一个初始值之后，我们轮流优化 f_1, f_2, ϕ

首先优化 f_1:

$$\begin{aligned}
&\frac{\delta}{\delta f_1}F^{RSF}=0\\
\Rightarrow&\frac{\delta}{\delta f_1}\lambda_1\int_\Omega\left(\int_\Omega K(\boldsymbol{x}-\boldsymbol{y})|u_0(\boldsymbol{y})-f_1(\boldsymbol{x})|^2H(\phi(\boldsymbol{y}))\,\mathrm{d}\boldsymbol{y}\right)\mathrm{d}\boldsymbol{x}=0\\
\Rightarrow&\frac{\delta}{\delta f_1}\int_\Omega\left(\int_\Omega K(\boldsymbol{x}-\boldsymbol{y})\left(-2u_0(\boldsymbol{y})f_1(\boldsymbol{x})+f_1^2(\boldsymbol{x})\right)H(\phi(\boldsymbol{y}))\,\mathrm{d}\boldsymbol{y}\right)\mathrm{d}\boldsymbol{x}=0\\
\Rightarrow&-2\int_\Omega K(\boldsymbol{x}-\boldsymbol{y})u_0(\boldsymbol{y})H(\phi(\boldsymbol{y}))\,\mathrm{d}\boldsymbol{y}+2f_1\int_\Omega K(\boldsymbol{x}-\boldsymbol{y})H(\phi(\boldsymbol{y}))\,\mathrm{d}\boldsymbol{y}=0\\
\Rightarrow&f_1=\frac{\int_\Omega K(\boldsymbol{x}-\boldsymbol{y})u_0(\boldsymbol{y})H(\phi(\boldsymbol{y}))\,\mathrm{d}\boldsymbol{y}}{\int_\Omega K(\boldsymbol{x}-\boldsymbol{y})H(\phi(\boldsymbol{y}))\,\mathrm{d}\boldsymbol{y}}\\
\Rightarrow&f_1=\frac{K(\boldsymbol{x})*[u_0(\boldsymbol{x})H(\phi(\boldsymbol{x}))]}{K(\boldsymbol{x})*H(\phi(\boldsymbol{x}))}
\end{aligned}$$

故令:

$$f_1=\frac{K(\boldsymbol{x})*[u_0(\boldsymbol{x})H(\phi(\boldsymbol{x}))]}{K(\boldsymbol{x})*H(\phi(\boldsymbol{x}))}$$

然后优化 f_2:

$$\begin{aligned}
&\frac{\delta}{\delta f_2}F^{RSF}=0\\
\Rightarrow&\frac{\delta}{\delta f_2}\lambda_2\int_\Omega\left(\int_\Omega K(\boldsymbol{x}-\boldsymbol{y})|u_0(\boldsymbol{y})-f_2(\boldsymbol{x})|^2[1-H(\phi(\boldsymbol{y}))]\,\mathrm{d}\boldsymbol{y}\right)\mathrm{d}\boldsymbol{x}=0\\
\Rightarrow&\frac{\delta}{\delta f_2}\int_\Omega\left(\int_\Omega K(\boldsymbol{x}-\boldsymbol{y})\left(-2u_0(\boldsymbol{y})f_2(\boldsymbol{x})+f_2^2(\boldsymbol{x})\right)[1-H(\phi(\boldsymbol{y}))]\,\mathrm{d}\boldsymbol{y}\right)\mathrm{d}\boldsymbol{x}=0\\
\Rightarrow&-2\int_\Omega K(\boldsymbol{x}-\boldsymbol{y})u_0(\boldsymbol{y})[1-H(\phi(\boldsymbol{y}))]\,\mathrm{d}\boldsymbol{y}+2f_2\int_\Omega K(\boldsymbol{x}-\boldsymbol{y})[1-H(\phi(\boldsymbol{y}))]\,\mathrm{d}\boldsymbol{y}=0\\
\Rightarrow&f_2=\frac{\int_\Omega K(\boldsymbol{x}-\boldsymbol{y})u_0(\boldsymbol{y})[1-H(\phi(\boldsymbol{y}))]\,\mathrm{d}\boldsymbol{y}}{\int_\Omega K(\boldsymbol{x}-\boldsymbol{y})[1-H(\phi(\boldsymbol{y}))]\,\mathrm{d}\boldsymbol{y}}\\
\Rightarrow&f_2=\frac{K(\boldsymbol{x})*[u_0(\boldsymbol{x})[1-H(\phi(\boldsymbol{x}))]]}{K(\boldsymbol{x})*[1-H(\phi(\boldsymbol{x}))]}
\end{aligned}$$

故令:

$$f_2=\frac{K(\boldsymbol{x})*[u_0(\boldsymbol{x})[1-H(\phi(\boldsymbol{x}))]]}{K(\boldsymbol{x})*[1-H(\phi(\boldsymbol{x}))]}$$

最后优化 ϕ:

令:

$$\begin{aligned}
f(\boldsymbol{x},\phi(\boldsymbol{x}),\nabla\phi(\boldsymbol{x}))=&\lambda_1\int_\Omega K(\boldsymbol{x}-\boldsymbol{y})|u_0(\boldsymbol{y})-f_1(\boldsymbol{x})|^2H(\phi(\boldsymbol{y}))\,\mathrm{d}\boldsymbol{y}\\
&+\lambda_2\int_\Omega K(\boldsymbol{x}-\boldsymbol{y})|u_0(\boldsymbol{y})-f_2(\boldsymbol{x})|^2[1-H(\phi(\boldsymbol{y}))]\,\mathrm{d}\boldsymbol{y}\\
&+\nu\delta(\phi(\boldsymbol{x}))|\nabla\phi(\boldsymbol{x})|\\
&+\mu\frac{1}{2}\left(|\nabla\phi(\boldsymbol{x})|-1\right)^2
\end{aligned}$$

则:

$$F^{RSF}(\phi)=\int_\Omega f(\boldsymbol{x},\phi(\boldsymbol{x}),\nabla\phi(\boldsymbol{x}))\,\mathrm{d}\boldsymbol{x}$$

记:

$$e_1(\boldsymbol{x}) = \int_\Omega K(\boldsymbol{y}-\boldsymbol{x})\left(u_0(\boldsymbol{x}) - f_1(\boldsymbol{y})\right)^2 \mathrm{d}\boldsymbol{y}$$
$$e_2(\boldsymbol{x}) = \int_\Omega K(\boldsymbol{y}-\boldsymbol{x})\left(u_0(\boldsymbol{x}) - f_2(\boldsymbol{y})\right)^2 \mathrm{d}\boldsymbol{y}$$

则 F^{RSF} 在 ϕ 处的梯度:

$$\begin{aligned}
\frac{\delta}{\delta\phi}F^{RSF} =& \frac{\partial f}{\partial \phi} - \nabla \cdot \frac{\partial f}{\partial \nabla \phi} \\
=& \lambda_1 \int_\Omega K(\boldsymbol{y}-\boldsymbol{x})\left(u_0(\boldsymbol{x}) - f_1(\boldsymbol{y})\right)^2 \mathrm{d}\boldsymbol{y}\delta(\phi(\boldsymbol{x})) \\
&- \lambda_2 \int_\Omega K(\boldsymbol{y}-\boldsymbol{x})\left(u_0(\boldsymbol{x}) - f_2(\boldsymbol{y})\right)^2 \mathrm{d}\boldsymbol{y}\delta(\phi(\boldsymbol{x})) \\
&+ \nu \nabla \cdot \left(\delta(\phi(\boldsymbol{x}))\frac{\nabla\phi(\boldsymbol{x})}{|\nabla\phi(\boldsymbol{x})|}\right) \\
&+ \mu \nabla \cdot \left(\nabla\phi(\boldsymbol{x}) - \frac{\nabla\phi(\boldsymbol{x})}{|\nabla\phi(\boldsymbol{x})|}\right) \\
=& \delta(\phi(\boldsymbol{x}))\left[\lambda_1 e_1(\boldsymbol{x}) - \lambda_2 e_2(\boldsymbol{x}) - \nu\nabla\cdot\left(\frac{\nabla\phi(\boldsymbol{x})}{|\nabla\phi(\boldsymbol{x})|}\right)\right] - \mu\left(\nabla^2\phi(\boldsymbol{x}) - \nabla\cdot\left(\frac{\nabla\phi(\boldsymbol{x})}{|\nabla\phi(\boldsymbol{x})|}\right)\right)
\end{aligned}$$

然后令 ϕ 向负梯度方向更新。

2.3 代码实现

实现思路与 CV 模型相似，我们定义一个 RSF_model 类，把 f_1, f_2, ϕ 作为其类属性，然后在其内部定义一个 fit 方法，用来优化 f_1, f_2, ϕ。在使用要处理的图像实例化一个 RSF_model 类之后，我们调用 fit 方法之后，就会不断更新迭代 f_1, f_2, ϕ，直到达到设置的迭代次数上限。具体代码实现如下:

```python
import math
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
from scipy.io import loadmat
from scipy.signal import convolve2d

epsilon = 0.01
sigma = 25

def set_sigma(a: float):
    global sigma
    sigma = a
def H_pointwise(x: float) -> float:
    return (1/2) * (1 + (2/math.pi) * math.atan(x/epsilon))

H = np.vectorize(H_pointwise)

```

```python
def delta_pointwise(x: float) -> float:
    return (1/math.pi)*(epsilon/(x**2 + epsilon**2))

def gradient_length(image: np.array) -> float:
    gradient = np.gradient(image)
    return np.sqrt(gradient[0]**2 + gradient[1]**2)

delta = np.vectorize(delta_pointwise)

def K_pointwise(u: np.array) -> float:
    return np.exp(-(np.linalg.norm(u)**2)/(2*sigma**2))/(2*math.pi*sigma**2)

K = np.vectorize(K_pointwise)


def divergence_after_normalized_gradient(image: np.array) -> np.array:
    gradient = np.gradient(image)
    magnitude = np.sqrt(gradient[0]**2 + gradient[1]**2)
    magnitude[magnitude == 0] = 0.01
    normalized_gradient = gradient / magnitude
    divergence = np.gradient(normalized_gradient[0])[0] + np.gradient(normalized_gradient[1])[1]
    return divergence

def laplace(image: np.array) -> np.array:
    gradient = np.gradient(image)
    laplacian = np.gradient(gradient[0])[0] + np.gradient(gradient[1])[1]
    return laplacian

class RSF_model:
    def __init__(self, figure: np.array, lambda1: float = 1, lambda2: float = 1, nu: float = 1, mu: float = 1, left: int = 10, right: int = 10, up: int = 10, down: int = 10, difference: int = 8, mode: bool = False, sigma: float = 25):
        set_sigma(sigma)
        self.figure = figure
        self.f1 = np.zeros_like(self.figure)
        self.f2 = np.ones_like(self.figure)
        if mode:
```

```python
            self.phi = np.ones_like(self.figure)
        else:
            self.phi = np.zeros_like(self.figure)
        self.phi[:up, :] = self.phi[-down:, :] = self.phi[:, :left] =
            self.phi[:, -right:] = difference
        self.lambda1 = lambda1
        self.lambda2 = lambda2
        self.nu = nu
        self.mu = mu
        self.omega = np.empty(self.figure.shape, dtype=object)
        for index, _ in np.ndenumerate(self.omega):
            self.omega[index] = index
        self.K_omega = K(self.omega)

    def K_omega_mines_index(self, index: tuple) -> np.array:
        i, j = index
        m, n = self.omega.shape
        K_00 = np.fliplr(np.flipud(self.K_omega[:i + 1, :j + 1]))
        K_01 = np.flipud(self.K_omega[:i + 1, 1: n - j])
        K_10 = np.fliplr(self.K_omega[1:m - i, :j + 1])
        K_11 = self.K_omega[1:m - i, 1:n - j]
        Komega_minus_index = np.concatenate([np.concatenate([K_00,
            K_01], axis=1), np.concatenate([K_10, K_11], axis=1)],
            axis=0)
        return Komega_minus_index


    def e1_e2_pointwise(self, index: tuple) -> tuple:
        i, j = index
        K_y_minus_index = self.K_omega_mines_index(index)
        return np.sum((K_y_minus_index)*((self.figure[i, j] - self.f1)
            **2)), np.sum((K_y_minus_index)*((self.figure[i, j] - self
            .f2)**2))


    def update_phi(self, learning_rate: float):
        compute_e1_e2 = np.vectorize(self.e1_e2_pointwise)
        e1, e2 = compute_e1_e2(self.omega)
        div = divergence_after_normalized_gradient(self.phi)
        delta_phi = delta(self.phi)
        flow = -delta_phi * (self.lambda1 * e1 - self.lambda2 * e2) +
```

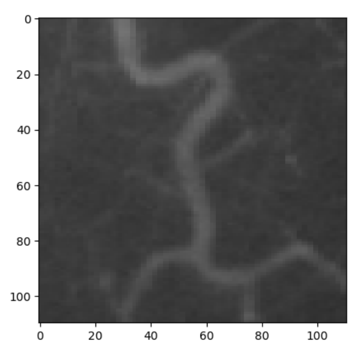
```
            self.nu * delta_phi * div + self.mu * (laplace(self.phi) - div)

        self.phi = self.phi + learning_rate * flow

    def update_f1(self, H_phi: np.array):
        denominator = convolve2d(self.K_omega, H_phi, mode='same')
        denominator[denominator == 0] = 0.001
        self.f1 = convolve2d(self.K_omega, H_phi*self.figure, mode='same') / denominator

    def update_f2(self, H_phi: np.array):
        denominator = convolve2d(self.K_omega, 1-H_phi, mode='same')
        denominator[denominator == 0] = 0.001
        self.f2 = convolve2d(self.K_omega, (1-H_phi)*self.figure, mode='same') / denominator

    def fit(self, learning_rate: float, max_iter: int = 100):
        for i in range(max_iter):
            H_phi = H(self.phi)
            self.update_f1(H_phi)
            self.update_f2(H_phi)
            self.update_phi(learning_rate)
            print(i)

    def draw(self, i: int):
        colored_figure = np.repeat(self.figure[:, :, np.newaxis], 3, axis=2)
        plt.imshow(colored_figure, cmap='gray')
        plt.contour(self.phi, [0], colors='r')
        plt.savefig(fname=f"out/img/output{i+1}_RSF.png", bbox_inches="tight", pad_inches=0.1)
        plt.show()


# 合成图像强度不均匀的图像
def generate_synthetic_image() -> np.array:
    Img = np.zeros((101, 101))
    for i in range(101):
        for j in range(101):
            Img[i, j] = 30 * (1 + np.sin(0.01 * np.pi * (i - j)))
```

```python

    Img[50, 50] = 100

    for i in range(101):
        for j in range(101):
            if np.sqrt((i - 50) ** 2 + (j - 50) ** 2) <= (35 + 7 * np.cos(8 * np.arctan((j - 50)*(i - 50)))):
                Img[i, j] = 100 * (1 + 0.2 * np.sin(0.01 * np.pi * (i - j)))
    return Img

def get_img()->list:
    imgs = []
    for i in range(1, 6):
        img = Image.open(f"fig/{i}.bmp")
        img = np.array(img)
        imgs.append(img)
    img = Image.open("fig/6.png")
    img = np.array(img)
    img_gray = img[:, :, 0]
    imgs.append(img_gray)
    data = loadmat('fig/brain_img75.mat')
    img = data['img']
    imgs.append(img)
    img = Image.open("fig/myBrain_axial.bmp")
    img = np.array(img)
    imgs.append(img)
    img = generate_synthetic_image()/120
    imgs.append(img)
    return imgs


if __name__ == '__main__':
    imgs = get_img()
    max_iter_list_RSF = [30, 30, 6, 100, 30, 30, 1, 1, 1]
    left_list = [10, 1, 5, 10, 1, 10, 10, 10, 5]
    right_list = [1, 1, 20, 4, 1, 8, 10, 5, 5]
    up_list = [1, 1, 8, 5, 1, 5, 10, 10, 5]
    down_list = [1, 1, 10, 10, 1, 10, 10, 5, 7]
    sigma_list = [75, 25, 25, 25, 25, 20, 25, 25, 25]
    learning_rate_list = [.1, .1, .1, .2, .1, .1, .1, .1, .1]
```

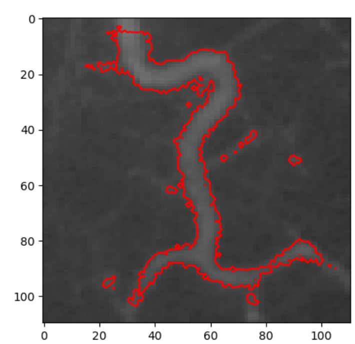
```python
    difference_list = [255, 255, 8, 255, 255, 8, 8, 8, 8]
    nu_list = [.5, .5, 1, 100, 1, 5, 1, 1, 1]
    mu_list = [.5, .5, 1, 0, .5, .1, 1, 1, 1]
    mode_list = [True, True, False, True, True, True, False, False,
        False]


    for i in range(9):
        img = imgs[i]
        model = RSF_model(img, lambda1=1, lambda2=1, nu=nu_list[i],
            mu=mu_list[i], left=left_list[i], right=right_list[i], up
            =up_list[i], down=down_list[i], difference=
            difference_list[i], mode=mode_list[i], sigma=sigma_list[i
            ])
        model.fit(learning_rate_list[i], max_iter_list_RSF[i])
        model.draw(i)
```

2.4 实验结果

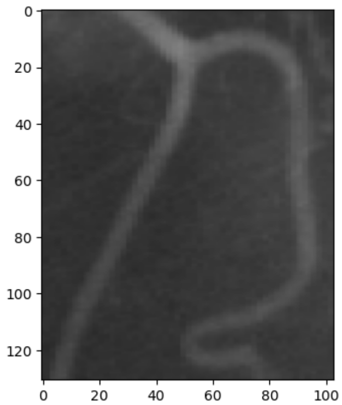
RSF 模型的实验结果如下:



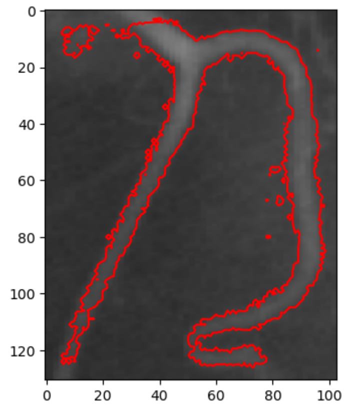
(19) 原图



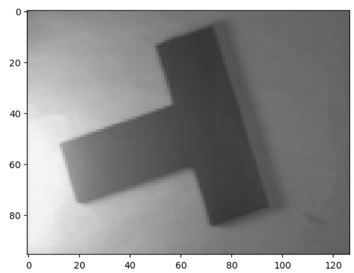
(20) RSF 模型处理后



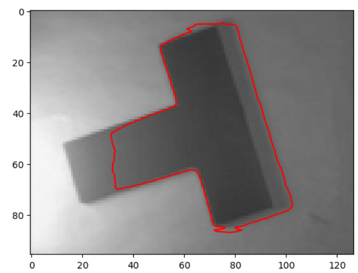
(21) 原图



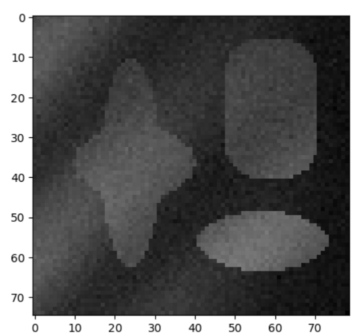
(22) RSF 模型处理后



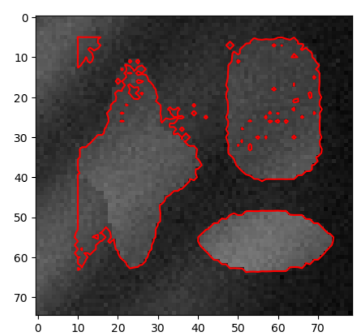
(23) 原图



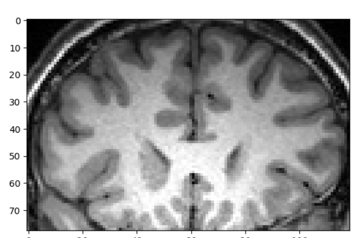
(24) RSF 模型处理后



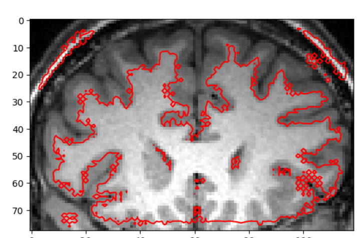
(25) 原图



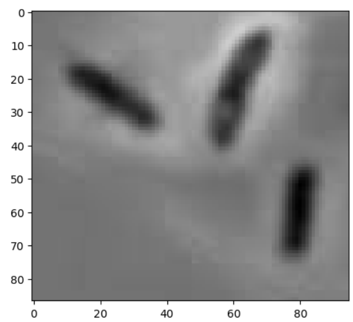
(26) RSF 模型处理后



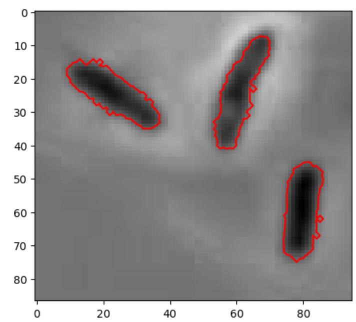
(27) 原图



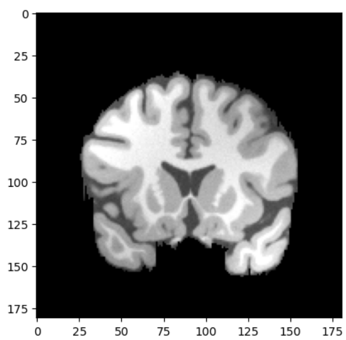
(28) RSF 模型处理后



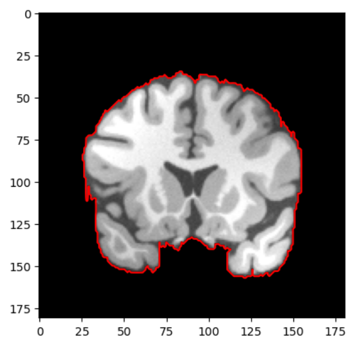
(29) 原图



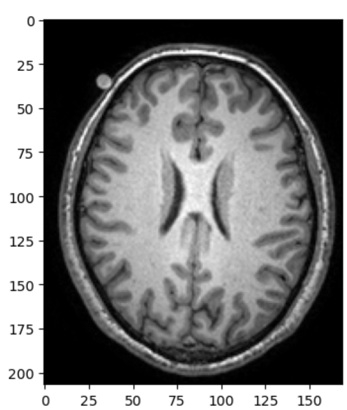
(30) RSF 模型处理后



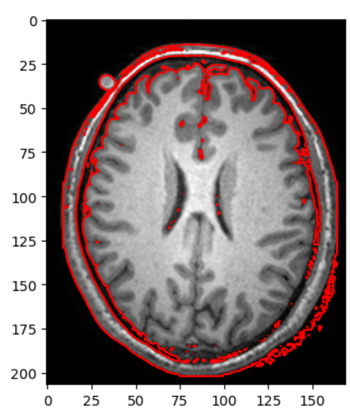
(31) 原图



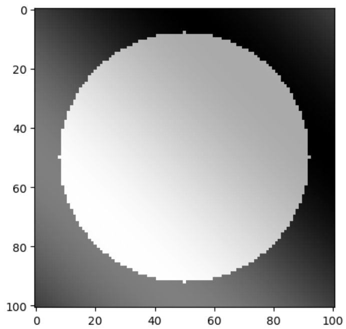
(32) RSF 模型处理后



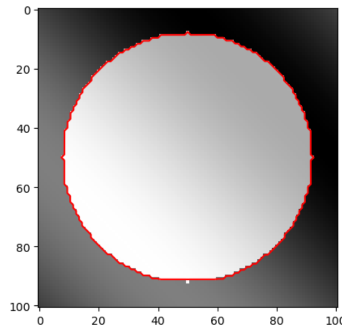
(33) 原图



(34) RSF 模型处理后



(35) 原图



(36) RSF 模型处理后

2.5 结论与讨论

可以看到，对于图像强度不均匀的情况，RSF 模型的效果优于 CV 模型，比如最后一张图，RSF 就把图里的圆完美地分出来了，但可能是因为初始值和超参数的选取不是很恰当，有些图像的分割效果不是很理想。