Math 124 - Programming for Mathematical Applications

UC Berkeley, Spring 2023

Project 4 - Image segmentation

Due Friday, April 14

Description

Image segmentation is a technique for partitioning an image into multiple segments, in order to identify objects and boundaries. It has a wide range of applications, in fields such as computer vision, medical imaging, and face recognition.

In this project you will implement a simplified version of the so-called *Chan-Vese* levelset based image segmentation method. If you are interested, you can learn more about the method at https://www.ipol.im/pub/art/2012/g-cv/article.pdf (https://www.ipol.im/pub/art/2012/g-cv/article.pdf). But all you need to know for the project will be described below.

Preliminaries

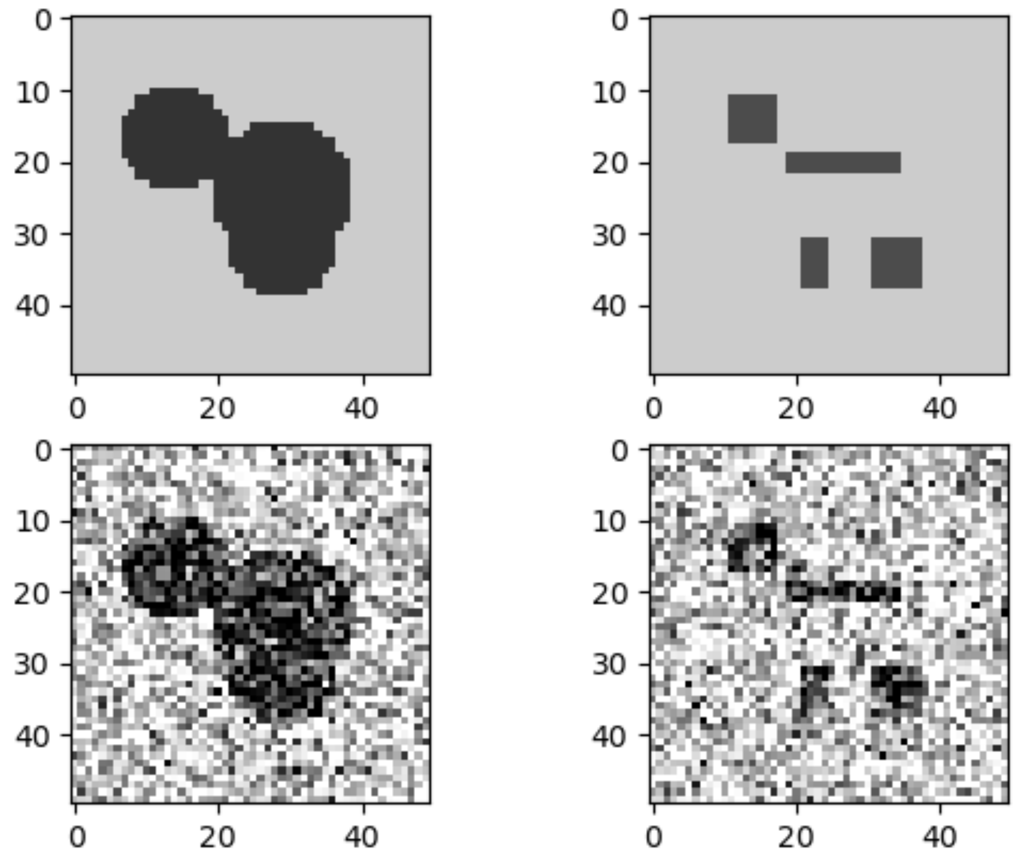
First we will define the images that we will use to test our method. The function below implements two test problems of size `m` -by- `m` , and it has an option to add a given amount of Gaussian noise:

In [1]:
```
using PyPlot

function test_image(ver, m=50, noise=0)
    A = 0.8*ones(Float64, m, m)
    if ver == 1
        i = 1:m
        sc = m/100
        for c in [[50,60,20], [65,60,15], [35,30,15]]
            A = @. max(0.2, A - 0.6*Float64((i - sc*c[1])^2 +
                        (i' - sc*c[2])^2 < (sc*c[3])^2))
        end
    elseif ver == 2
        is = [[25,35,25,35], [65,75,65,75], [65,75,45,50], [40,45,
        for i in is
            i = round.(Int, i*m/100)
            A[i[1]:i[2], i[3]:i[4]] .= 0.3
        end
    else
        error("Unknown image version")
    end

    A += noise*randn(size(A))
    A = min.(max.(A, 0), 1)
end
```

Out[1]: `test_image (generic function with 3 methods)`

These two test images are shown below, with no noise (top row) and with noise of magnitude 0.3 (bottom row).

In [2]:
```
count = 0
m = 50
for noise = [0, 0.3], ver = 1:2
    subplot(2,2,count+=1)
    A = test_image(ver, m, noise)
    imshow(A[:,:,[1,1,1]])
end
```



Clearly, it appears much more difficult to identify the objects and the boundaries with a large amount of noise. The method we will implement here is particularly good at handling these cases.

Level sets and contour plotting

The Chan-Vese method is based on the *levelset method*. A function $\Phi(x, y)$ is used to represent an interface as a zero contour, that is, the points x, y where $\Phi(x, y) = 0$. For example, a circle centered at x_0, y_0 with radius r can be represented by the function

$$\Phi(x, y) = \sqrt{(x - x_0)^2 + (y - y_0)^2} - r$$

This is implemented in the function below, which creates a matrix Φ of given size `sz` and initializes it to values that represent a large circle.

In [3]:

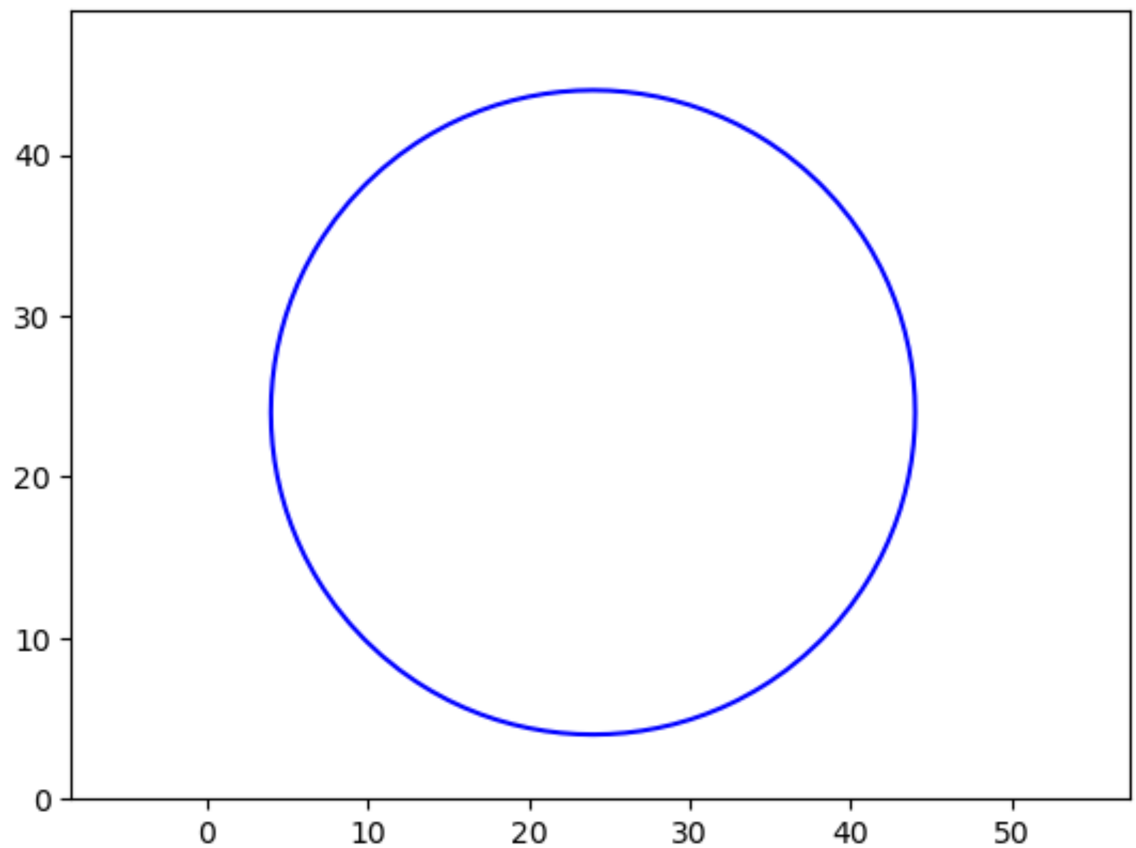
```julia
function initial_value(sz)
    m,n = sz
    Φ = [sqrt((i - m/2)^2 + (j - n/2)^2) - 0.4n for i = 1:m, j = 1
end
```

Out[3]:

```
initial_value (generic function with 1 method)
```

The `contour` function can be used to plot the zero contour for this matrix Φ:

In [4]:
```
Φ = initial_value([50,50])
contour(Φ, [0.0], colors="b")
axis("equal");
```



Algorithm

The segmentation method is based on starting from an initial matrix Φ, and evolving the interface using the expressions below. With certain assumptions on the image matrix A, the zero contour $\Phi(x, y) = 0$ will align with the boundaries of the objects in the image.

First, we define so-called smoothed Heaviside and delta functions:

$$H(t) = \frac{1}{2}\left(1 + \frac{2}{\pi}\arctan(t)\right)$$

$$\delta(t) = \frac{d}{dt}H(t) = \frac{1}{\pi(t^2 + 1)}$$

For an image matrix A and a levelset matrix Φ, both of size m-by-n, we define the following scalars:

$$c_1 = \frac{\sum_{i=1}^m \sum_{j=1}^n A_{ij} H(\Phi_{ij})}{\sum_{i=1}^m \sum_{j=1}^n H(\Phi_{ij})}$$
$$c_2 = \frac{\sum_{i=1}^m \sum_{j=1}^n A_{ij}(1 - H(\Phi_{ij}))}{\sum_{i=1}^m \sum_{j=1}^n (1 - H(\Phi_{ij}))}$$

Next we define an *update matrix* $\Delta\Phi$ of size m-by-n with the following entries:

$$\Delta\Phi_{ij} = 100\delta(\Phi_{ij}) \left(0.2\kappa_{ij} - (A_{ij} - c_1)^2 + (A_{ij} - c_2)^2\right)$$

Here, the curvature κ_{ij} is defined by the following expressions:

$$\begin{aligned}
\Phi_{ij}^{xx} &= \Phi_{i+1,j} - 2\Phi_{ij} + \Phi_{i-1,j} \\
\Phi_{ij}^{yy} &= \Phi_{i,j+1} - 2\Phi_{ij} + \Phi_{i,j-1} \\
\Phi_{ij}^{xy} &= (\Phi_{i+1,j+1} - \Phi_{i-1,j+1} - \Phi_{i+1,j-1} + \Phi_{i-1,j-1})/4 \\
\Phi_{ij}^{x} &= (\Phi_{i+1,j} - \Phi_{i-1,j})/2 \\
\Phi_{ij}^{y} &= (\Phi_{i,j+1} - \Phi_{i,j-1})/2 \\
\kappa_{ij}^0 &= \frac{\Phi_{ij}^{xx}(\Phi_{ij}^{y})^2 - 2\Phi_{ij}^{x}\Phi_{ij}^{y}\Phi_{ij}^{xy} + \Phi_{ij}^{yy}(\Phi_{ij}^{x})^2}{((\Phi_{ij}^{x})^2 + (\Phi_{ij}^{y})^2)^{3/2} + 10^{-6}} \\
\kappa_{ij} &= \max(\min(\kappa_{ij}^0, 5), -5)
\end{aligned}$$

Finally, the algorithm performs the following steps iteratively:

- Compute c_1, c_2
- Compute the update matrix $\Delta\Phi$
- Update $\Phi \to \Phi + \Delta\Phi$
- Repeat until $\max_{ij} |\Delta\Phi_{ij}| < 2 \cdot 10^{-2}$

Problem 1 - A type hierarchy for stencil operations

If you consider the operations in the Image Filtering section of the lecture notes, you can see they all fit the following pattern: Loop over all the (internal) image pixels, apply some function to a *local* 3-by-3 submatrix around each pixel, which determines the new filtered image pixel value. This structure is also called a *stencil operation*, and the function that maps a 3-by-3 matrix to a value is called *the stencil*.

To demonstrate how to implement this using a Julia type hierarchy, we will first define an abstract stencil type:

In [5]:

```julia
abstract type AbstractStencil end
```

We can then define a struct for the actual stencils, as a *subtype* of the abstract stencil. For example, for the mean filter:

In [6]:

```julia
struct AverageStencil <: AbstractStencil end
```

This allows us to define functions that are different depending on the subtype, but still write general functions that can operate on any stencil of subtype `AbstractStencil`. For example, the average stencil is defined by the following function on each 3-by-3 submatrix:

In [7]:

```julia
apply_to_3x3(s::AverageStencil, A33) = sum(A33) / length(A33)    #
```

Out[7]:

```
apply_to_3x3 (generic function with 1 method)
```

Other functions can now be written in a way that accepts any stencil, or more precisely any struct object which is a subtype of `AbstractStencil`. The following function demonstrates the syntax for doing this, note how the input stencil `s` is passed to a function `apply_stencil` that you will implement next.

In [8]:

```julia
function stencil_demo(s::AbstractStencil)
    count = 0
    plot_next(A) = subplot(1,4,count+=1), imshow(A[:,:,[1,1,1]])
    for noise = [0, 0.3]
        A = test_image(1, 50, noise)
        plot_next(A)
        plot_next(apply_stencil(s,A))
    end
end
```

Out[8]:

```
stencil_demo (generic function with 1 method)
```

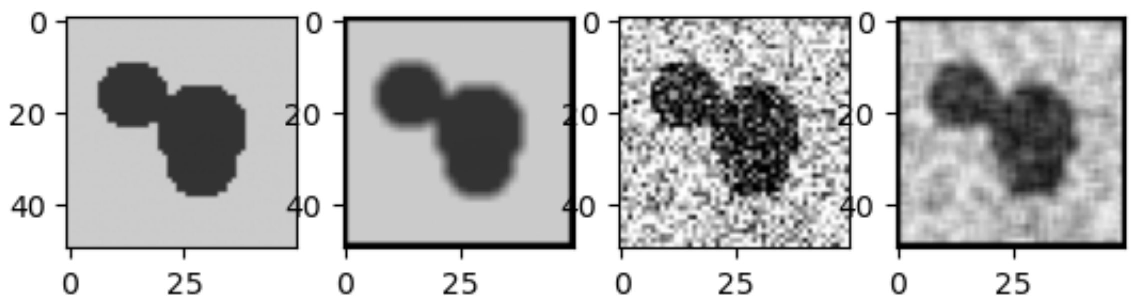
Problem 1(a)

Complete the `apply_stencil` function below, which applies the stencil `s` on the image `A` and returns the resulting image. *Hint*: This is exactly like e.g. the `image_avgfilter` in the lecture notes, except the actual stencil operation is optained by called `apply_to_3x3`.

In [9]:
```
function apply_stencil(s::AbstractStencil, A)
    B = 0*A
    for i = 2:size(A, 1)-1
        for j = 2:size(A, 2)-1
            B[i,j] = apply_to_3x3(s, A[i-1:i+1, j-1:j+1])
        end
    end
    return B
end
```

Out[9]:
```
apply_stencil (generic function with 1 method)
```

In [10]:
```
# Test code
stencil_demo(AverageStencil())
```



Problem 1(b)

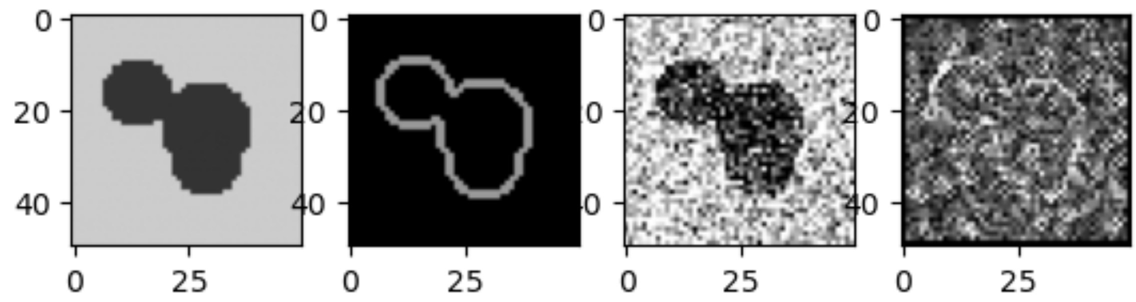
Similarly, define a new subtype `EdgeStencil` which applies the same operation as `maxabsgradfilter` in the lecture notes.

In [11]:
```
struct EdgeStencil <: AbstractStencil end
```

In [12]:
```
function apply_to_3x3(s::EdgeStencil, A33)
    return max(abs(A33[3,2] - A33[1,2]), abs(A33[2,3] - A33[2,1]))
end
```

Out[12]:
```
apply_to_3x3 (generic function with 2 methods)
```

In [13]:
```julia
# Test code
stencil_demo(EdgeStencil())
```



Note that with the high level of noise, the edge detection essentially cannot identify the object in the image. The goal of the rest of this problem set is to implement the better levelset segmentation algorithm.

Problem 2 - Utilities

Problem 2(a)

Note that the curvature κ is also a (more complicated) stencil operation of the same form as the previous ones.

Define a new subtype `KappaStencil` which implements this function.

In [14]:
```julia
struct KappaStencil <: AbstractStencil end
```

Problem 2(b)

Implement the functions `H(t)` and `δ(t)` .

In [15]:
```julia
H(t) = (1 + 2/π * atan(t))/2
δ(t) = 1/(π*(t^2 + 1))
```

Out[15]:
```
δ (generic function with 1 method)
```

Problem 2(c)

Implement a function `coefficients(Φ, A)` which computes and returns c_1, c_2 for input matrices Φ and A.

In [16]:

```
function coefficients(Φ, A)
    sum1 = 0
    for i = 1:size(A, 1), j = 1:size(A, 2)
        sum1 += A[i,j] * H(Φ[i,j])
    end
    sum2 = 0
    for i = 1:size(A, 1), j = 1:size(A, 2)
        sum2 += H(Φ[i,j])
    end
    c1 = sum1/sum2

    sum1 = 0
    for i = 1:size(A, 1), j = 1:size(A, 2)
        sum1 += A[i,j] * (1 - H(Φ[i,j]))
    end
    sum2 = 0
    for i = 1:size(A, 1), j = 1:size(A, 2)
        sum2 += 1 - H(Φ[i,j])
    end
    c2 = sum1/sum2
    return c1, c2
end
```

Out[16]:

```
coefficients (generic function with 1 method)
```

Problem 2(d)

Implement a function `update(Φ, A)` which computes and returns the update matrix $\Delta\Phi$ for input matrices Φ and A (using the functions implemented above for computing c_1, c_2 and κ).

In [17]:
```
function update(Φ, A)
    ΔΦ = 0*Φ
    c1, c2 = coefficients(Φ, A)
    for i = 2:size(Φ, 1)-1, j = 2:size(Φ, 2)-1
        Φ_ijxx = Φ[i + 1,j] - 2*Φ[i,j] + Φ[i - 1,j]
        Φ_ijyy = Φ[i,j + 1] - 2*Φ[i,j] + Φ[i,j - 1]
        Φ_ijxy = (Φ[i+1, j+1] - Φ[i-1, j+1] - Φ[i+1, j-1] + Φ[i-1,
        Φ_ijx = (Φ[i+1, j] - Φ[i-1, j])/2
        Φ_ijy = (Φ[i, j+1] - Φ[i, j-1])/2
        κ_ij0 = (Φ_ijxx*(Φ_ijy)^2 - 2*Φ_ijx*Φ_ijy*Φ_ijxy + Φ_ijyy*
        κij = max(min(κ_ij0, 5), -5)
        ΔΦ[i,j] = 100 * δ(Φ[i,j]) * (0.2*κij - (A[i,j] - c1)^2 + (
    end
    return ΔΦ
end
```

Out[17]: update (generic function with 1 method)

Problem 3 - Final Image Segmentation function

Implement a function `image_segment(A; maxiter=100000)` which implements the overall algorithm, more precisely:

- Start by initializing Φ using the `initial_value` function
- Iterate at most `maxiter` times
- Compute updates $\Delta\Phi$ and add to Φ
- Terminate if $\max_{ij} |\Delta\Phi_{ij}| < 2 \cdot 10^{-2}$

The function finally returns Φ (whether it terminated early or not).

In [18]:
```
function image_segment(A; maxiter = 100000)
    Φ = initial_value([50,50])
    count = 0
    while true
        ΔΦ = update(Φ, A)
        if count == maxiter || maximum(abs.(ΔΦ)) < 2 * 10^(-2)
            break
        end
        count += 1
        Φ += ΔΦ
    end
    return Φ
end
```

Out[18]: image_segment (generic function with 1 method)

In [19]:
```
# Test code:
count = 0
for noise = [0, 0.3], ver = 1:2
    subplot(2,2,count+=1)
    A = test_image(ver, 50, noise)
    Φ = image_segment(A)
    imshow(A[:,:,[1,1,1]])
    contour(Φ, [0.0], colors="b")
end
```