In [1]:

```
import numpy as np
from tqdm.notebook import tqdm
```

In [2]:

```
# CFAR version 2, in this slidin window is created
#on the basis of value of the pixel

class CFAR_v2(object):

    #initializing the values

    def __init__(self,img,tw,gw,bw,pfa):
        self.img = img
        self.tw = tw
        self.gw = gw
        self.bw = bw
        self.pfa = pfa
        print("Kernel Ready.")

    #checking if the pixel exists
    def isPixelexists(self,size_img,a,b):
        r,c = size_img
        #print(r,c)
        if (a>=0 and a<r) and (b>=0 and b<c) :
            return True
        else:
            return False

    #Computing 4 buffer values.TOP,BOTTOM,LEFT and RIGHT
    def get_topBuffer(self,u,v,size_t,size_g):
        top_buffer = []
        radius_t = int(size_t/2)
        radius_g = int(size_g/2)
        #we have considered the target_window pixels too.
        for p in range(radius_t+1,radius_g+1):

            x = u-p
            for m in range(-p,p+1):
                y = v+m
                #print(x,y)
                if self.isPixelexists(self.img.shape,x,y):
                    #print("Found")
                    top_buffer.append(self.img[x][y])
                else:
                    #print("Not found")
                    top_buffer.append(0)
```

```python
        return top_buffer

    def get_bottomBuffer(self,u,v,size_t,size_g):
        bottom_buffer = []
        radius_t = int(size_t/2)
        radius_g = int(size_g/2)
        for p in range(radius_t+1,radius_g+1):

            x = u+p
            for m in range(-p,p+1):
                y = v+m
                #print(x,y)
                if self.isPixelexists(self.img.shape,x,y):
                    #print("Found")
                    bottom_buffer.append(self.img[x][y])
                else:
                    #print("Not found")
                    bottom_buffer.append(0)

        return bottom_buffer

    def get_leftBuffer(self,u,v,size_t, size_g):
        left_buffer = []
        radius_t = int(size_t/2)
        radius_g = int(size_g/2)
        for p in range(radius_t+1,radius_g+1):
            y = v-p
            for m in range(-p,p+1):
                x = u+m
                #print(x,y)
                if self.isPixelexists(self.img.shape,x,y):
                    #print("Found")
                    left_buffer.append(self.img[x][y])
                else:
                    #print("Not found")
                    left_buffer.append(0)

        return left_buffer

    def get_rightBuffer(self,u,v,size_t,size_g):
        right_buffer = []
        radius_t = int(size_t/2)
        radius_g = int(size_g/2)
        for p in range(radius_t+1,radius_g+1):
            y = v+p
            for m in range(-p,p+1):
                x = u+m
                #print(x,y)
                if self.isPixelexists(self.img.shape,x,y):
                    #print("Found")
                    right_buffer.append(self.img[x][y])
                else:
                    #print("Not found")
                    right_buffer.append(0)

        return right_buffer


    def compute_DV(self):
        dvi = []
        print("Computing DVi..")

        for i in tqdm(range(self.img.shape[0])):
            for j in (range(self.img.shape[1])):
                #print("hello")
                win_top_buffer = self.get_topBuffer(i,j,self.tw,self.gw)
                win_bottom_buffer = self.get_bottomBuffer(i,j,self.tw,self.gw)
                win_left_buffer = self.get_leftBuffer(i,j,self.tw,self.gw)
                win_right_buffer = self.get_rightBuffer(i,j,self.tw,self.gw)

                guard_buffer = np.array(

                    [win_top_buffer,win_bottom_buffer,win_left_buffer,win_right_buffer]

                )

                #print(guard_buffer)
                #print(guard_buffer.mean())
                #print(guard_buffer.std())
                #print((img[i][j] - guard_buffer.mean())/guard_buffer.std())
                dvi.append(abs(self.img[i][j] - guard_buffer.mean())/guard_buffer.std())

        dvi = np.array(dvi).reshape(self.img.shape)
        print("Process completed, DV image succesfully Computed.\n")
        return dvi
    def compute_noise(self):
        noise_data = []

        print("Computing P...")

        for i in tqdm(range(self.img.shape[0])):
            for j in range(self.img.shape[1]):

                win_top_buffer = self.get_topBuffer(i,j,self.tw,self.bw)
                win_bottom_buffer = self.get_bottomBuffer(i,j,self.tw,self.bw)
                win_left_buffer = self.get_leftBuffer(i,j,self.tw,self.bw)
                win_right_buffer = self.get_rightBuffer(i,j,self.tw,self.bw)

                background_buffer = np.array(

                    [win_top_buffer,win_bottom_buffer,win_left_buffer,win_right_buffer]

                )

                #print(guard_buffer)
                #print(guard_buffer.mean())
                noise_data.append(float(background_buffer.mean()))

        noise_data = (np.array(noise_data))
        #print(noise_data)
        P = np.array(self.compute_scaleFactor()*noise_data).reshape(self.img.shape)
        print("Process Completed, P image succesfully computed.\n")
        return P

    def compute_scaleFactor(self):
        N = 0
        for b in range(self.tw,self.bw+1):
            if b%2 != 0:
                N += 4*b -4
        return (N*(self.pfa**(-1/N) -1))

    def shipDetection(self):
        final_image = []

        T = self.compute_noise()
        #T= 30
        DV = self.compute_DV()
        for i in range(self.img.shape[0]):
            for j in range(self.img.shape[1]):

                if DV[i][j] > T[i][j]:

                    final_image.append(0)
                else:
                    final_image.append(1) #valid Ships

        final_image = np.array(final_image).reshape(self.img.shape)
        print("Binary Image of Ships is Succesfully Generated.\n")
        return final_image,DV,T
```