ECE 457B Assignment 3

Tzu-Chun Chou
20718966
t6chou@uwaterloo.ca

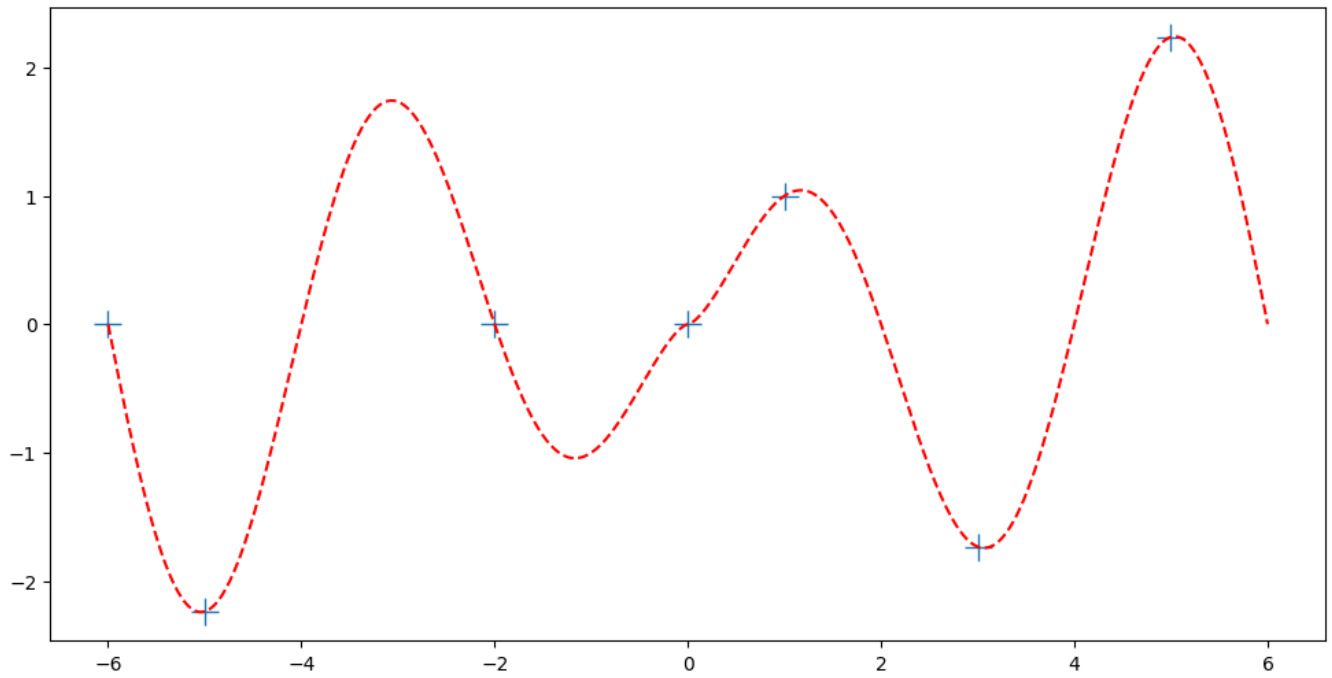
Question 1

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
```

Setup

In [ ]:
```
# original function
def f(x):
    return np.sqrt(np.abs(x))*np.sin(((np.pi)/2)*x)

# centroid
centroid = np.array([-6, -5, -2, 0, 1, 3, 5])
# dataset
x = np.linspace(-6, 6, 1000)
# target
y = np.zeros(1000)
for i in range(len(x)):
    y[i] = f(x[i])

plt.figure(figsize=(12, 6))
plt.plot(centroid ,f(centroid), '+', markersize=15)
plt.plot(x, y, '--r')
```

Out[ ]:
```
[<matplotlib.lines.Line2D at 0x1ee4bcdb880>]
```



RBF Interporlator on Guassian Kernel

In [ ]:
```
class RBFInterpolator():
    # inititalization function, numCenters is the number of Centers, centers if given, and sp
    def __init__(self, inpdim, outpdim, numCenters, centers, spread):
        self.inpdim = inpdim
        self.outpdim = outpdim
        self.numCenters = numCenters
        #initialize random weight function
        self.W = np.random.random((self.numCenters, self.outpdim))
        #initialize centers
```

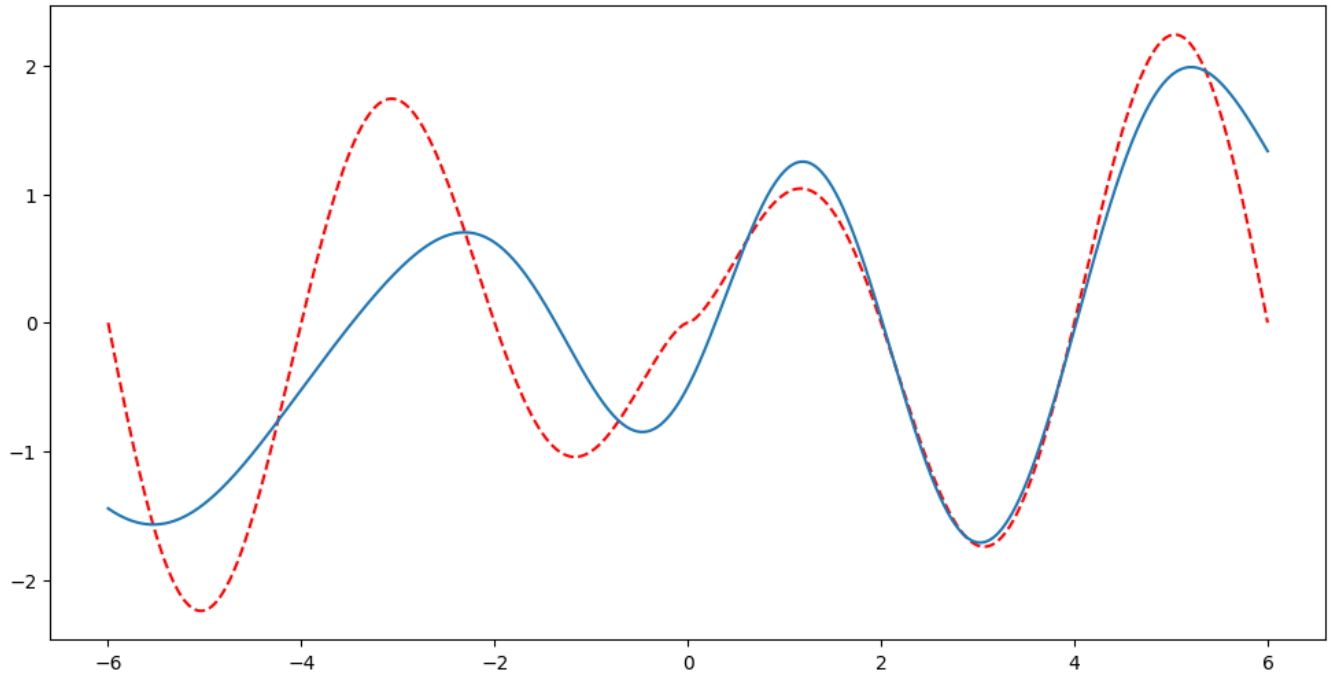
```
        self.C = centers
        #calculate initial spread (sigma)
        self.sigma = spread

    # gaussian basis function
    def _gaussBasisFunc(self, center, x):
        return np.exp(-((np.linalg.norm(x-center)**2) / 2*(self.sigma**2)))

    # function for calculating activation
    def _calculateActivation(self, X):
        # create the matrix of activations.
        # First initialize with zeros
        # Activation matrix will have dimension (number of rows in train * num_centers)
        G = np.zeros((X.shape[0], self.numCenters), float)

        # loop over all the centers and inputs
        for center_index, center in enumerate(self.C):
            for x_index, x in enumerate(X):

                # calculate activations and store in appropriate (input, center) cell
                G[x_index, center_index] = self._gaussBasisFunc(center, x)

        return G

    # function for training
    def fit(self, X, y):

        # calculating the activations for input data
        G = self._calculateActivation(X)

        # Calculate the weights
        self.W = np.dot(np.linalg.pinv(G), y)

        return self


    # function for testing
    def predict(self, X):

        # do the forward pass : calculate activations and dot product with weights to get out
        G = self._calculateActivation(X)
        output = np.dot(G, self.W)

        return output
```

1.a

I assumed the 6 datapoints given as the centroid of RBFs and generated 1000 random data points using f(x) and used them as the training samples.

In [ ]:
```
# sigma = 1
interp = RBFInterpolator(inpdim=1, outpdim=1, numCenters=len(centroid), centers=centroid, spr
interp.fit(x, y)
interp.predict(X = np.linspace(-6, 6, 1000))

plt.figure(figsize=(12,6))
plt.plot(x, f(x), '--r')
plt.plot(x, interp.predict(X = np.linspace(-6, 6, 1000)))
```

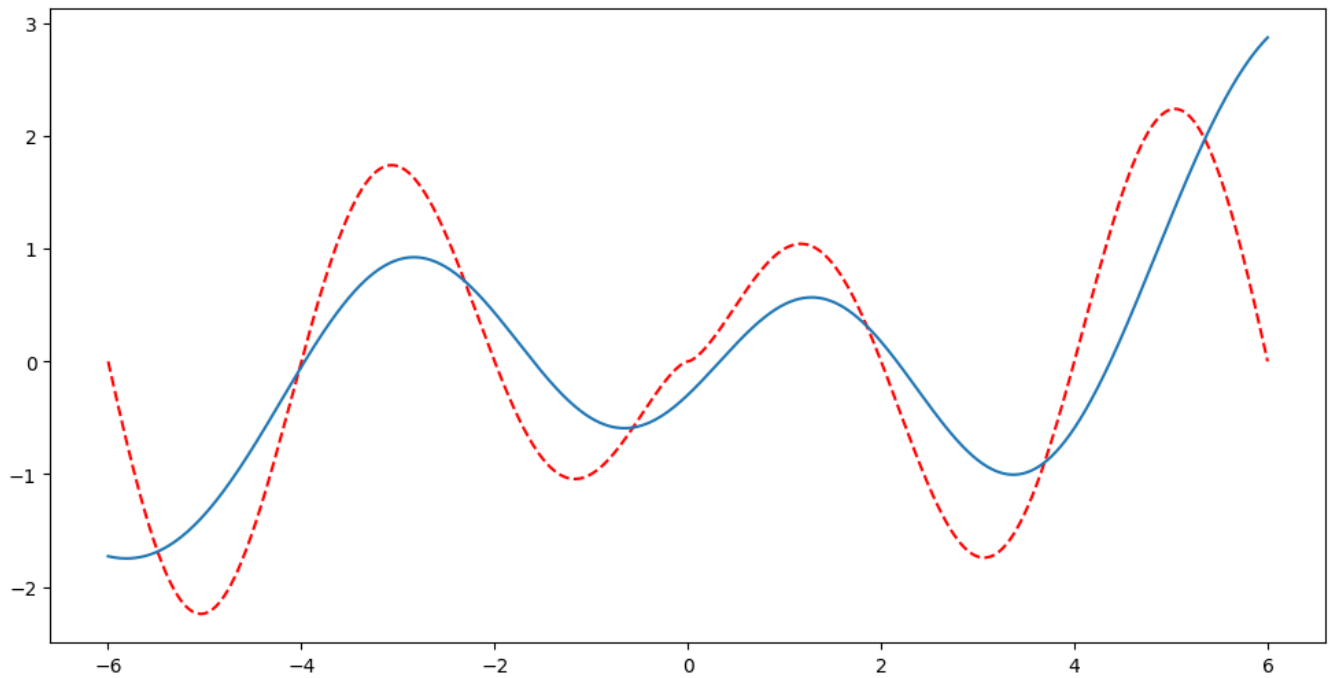
Out[ ]:
```
[<matplotlib.lines.Line2D at 0x1ee4bd4a4d0>]
```



1.b

In [ ]:
```
# sigma = 0.5
interp = RBFInterpolator(inpdim=1, outpdim=1, numCenters=len(centroid), centers=centroid, spr
interp.fit(x, y)
interp.predict(X = np.linspace(-6, 6, 1000))

plt.figure(figsize=(12,6))
plt.plot(x, f(x), '--r')
plt.plot(x, interp.predict(X = np.linspace(-6, 6, 1000)))
```

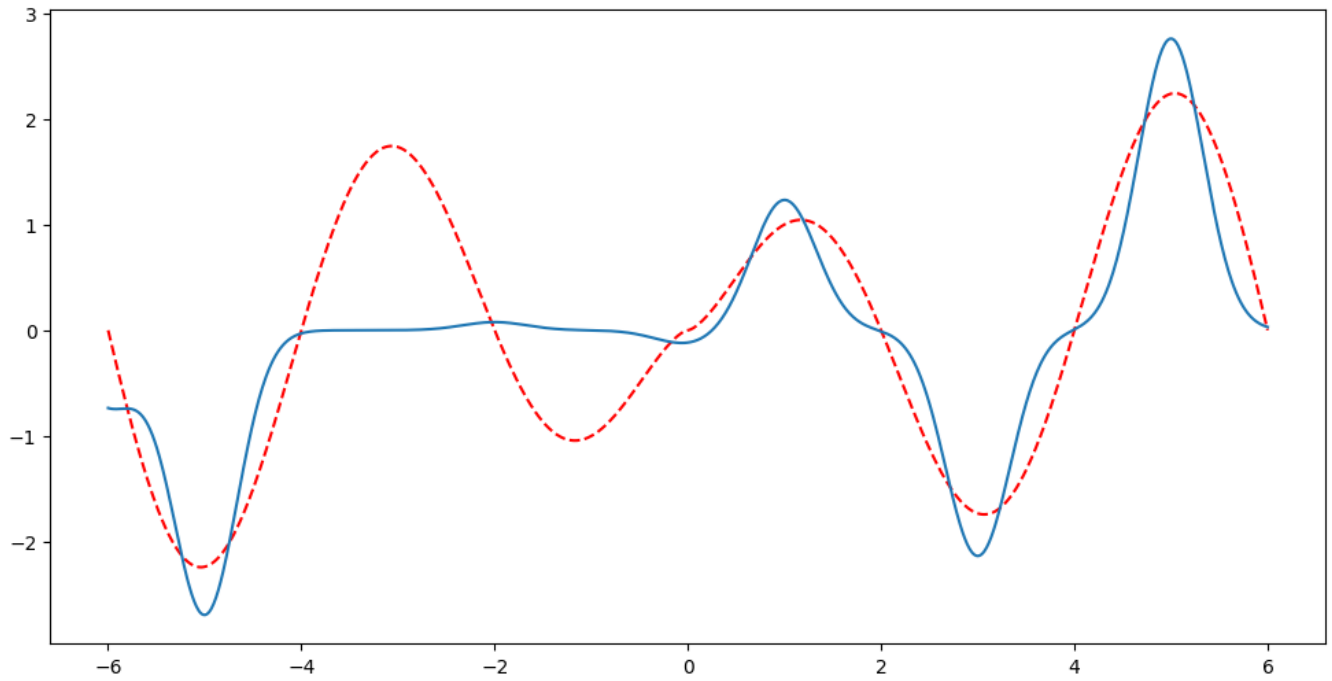
Out[ ]:
```
[<matplotlib.lines.Line2D at 0x1ee4c081510>]
```



In [ ]:
```
# sigma = 3
interp = RBFInterpolator(inpdim=1, outpdim=1, numCenters=len(centroid), centers=centroid, spr
interp.fit(x, y)
interp.predict(X = np.linspace(-6, 6, 1000))

plt.figure(figsize=(12,6))
```

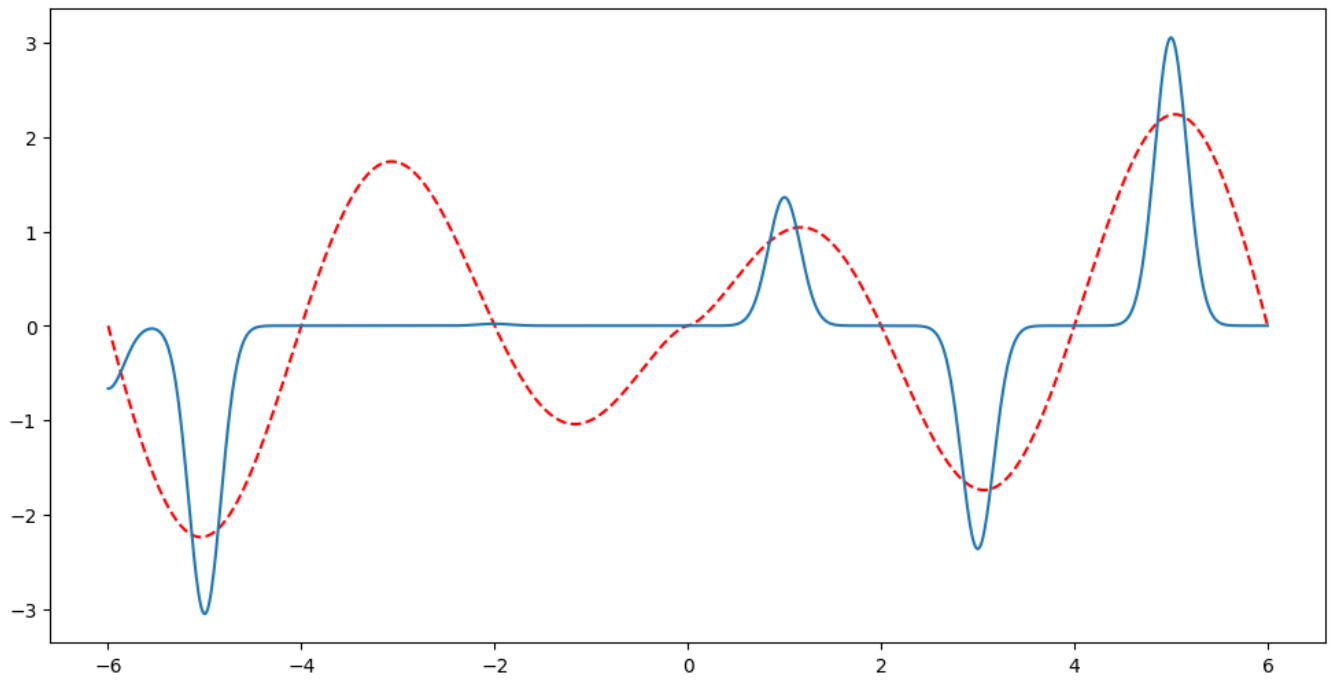
```
plt.plot(x, f(x), '--r')
plt.plot(x, interp.predict(X = np.linspace(-6, 6, 1000)))
```

Out[ ]:
```
[<matplotlib.lines.Line2D at 0x1ee4c0e4b50>]
```



In [ ]:
```
# sigma = 6
interp = RBFInterpolator(inpdim=1, outpdim=1, numCenters=len(centroid), centers=centroid, spr
interp.fit(x, y)
interp.predict(X = np.linspace(-6, 6, 1000))

plt.figure(figsize=(12,6))
plt.plot(x, f(x), '--r')
plt.plot(x, interp.predict(X = np.linspace(-6, 6, 1000)))
```

Out[ ]:
```
[<matplotlib.lines.Line2D at 0x1ee4c150220>]
```



1.c

From the above trial we can see that the width of the function plays a major importance in the accuracy of the interpolator. Overall we see that using the sigma value of 1 gives the most accurate interpolation

of the original function. In one hand using the value of 0.5 shows slightly less accuracy likely because the spread is too small to doesn't provide for a good interpolation of the function in between the sample data. On the other hand using the value of 3 and 6 provides very inaccurate result. likely because many information is being lost when the range of the radial function are larger than the original range of the function which only rangs from around -2 and 2.

Question 2

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
import math
from IPython import display
```

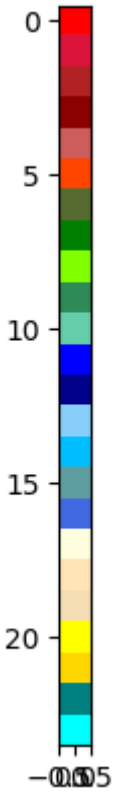
RGB Color Table

In [ ]:
```
inputRGB = np.array([
    [255,0,0],
    [220,20,60],
    [178,34,34],
    [139,0,0],
    [205,92,92],
    [255,69,0],
    [85,107,47],
    [0,128,0],
    [127,255,0],
    [46,139,87],
    [102,205,170],
    [0,0,255],
    [0,0,139],
    [135,206,250],
    [0,191,255],
    [95,158,160],
    [65,105,225],
    [255,255,224],
    [255,228,181],
    [245,222,179],
    [255,255,0],
    [255,215,0],
    [0,128,128],
    [0,255,255]
])
```

calibrate the color codes to values between 0 and 1, instead of being between 0 and 255

In [ ]:
```
normRGB = inputRGB/255
plt.imshow(np.reshape(normRGB,(normRGB.shape[0],1,3)))
print(normRGB.shape)
```

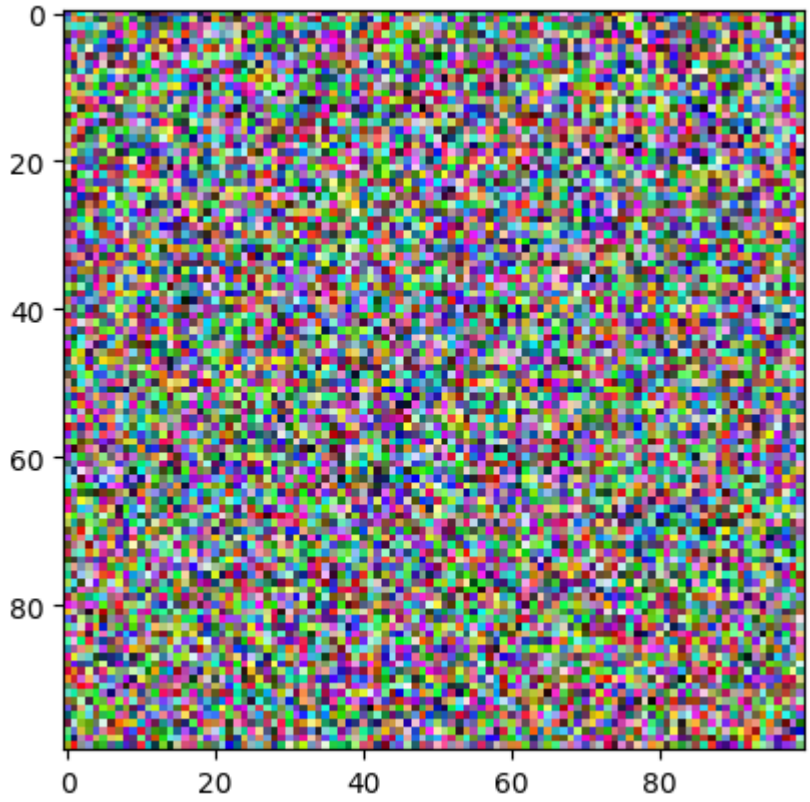
```
(24, 3)
```



Initialize all parameters according to the specification except sigma(0) which is set to 10 instead of 1 for better performance.

In [ ]:
```
space_size = 100 # 100 x 100 grid of neurons
alpha_0 = 0.8
sigma_0 = 10
max_epochs = 1000

# Initialize random weights
w = np.random.random((space_size,space_size,3))

# Generate a figure of the original grid (randomly selected)
plt.imshow(w)
```

Out[ ]:
```
<matplotlib.image.AxesImage at 0x1d93cb417e0>
```

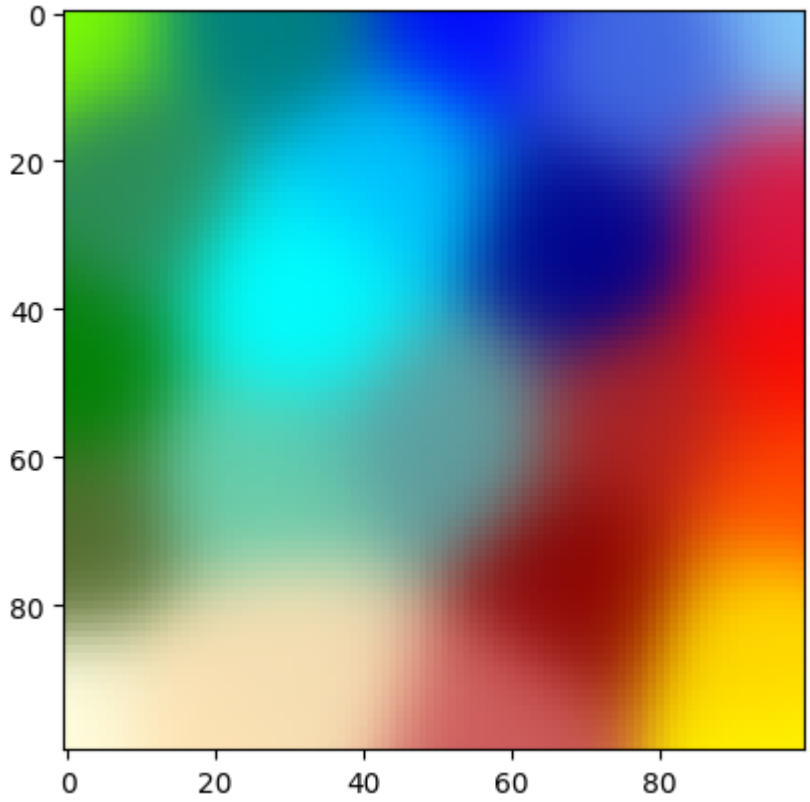


2.a

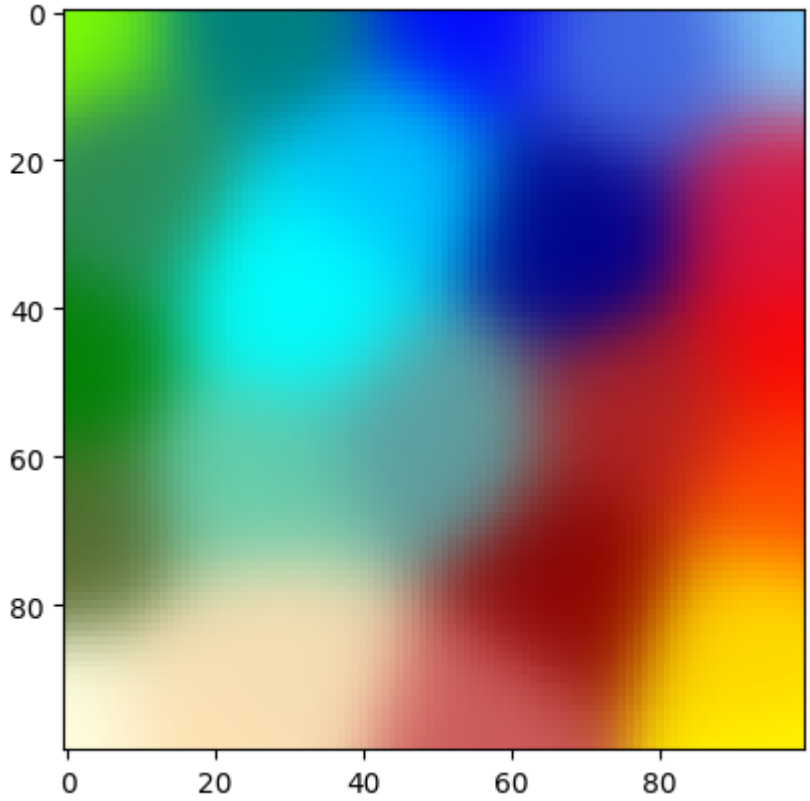
In [ ]:

```
epoch = 1
alpha = alpha_0
sigma = sigma_0

while epoch <= max_epochs:
    for x in normRGB:
        # calculate performance index
        diff = np.linalg.norm(x - w, axis = 2)
        # find index of winning node (I = min||x-w||)
        ind = np.unravel_index(np.argmin(diff, axis=None), diff.shape)

        # Update weights for neighbourhood
        for i in range(0, space_size):
            for j in range(0 ,space_size):
                # di,j is the euclidean distance between winning node i and surrounding node j
                d = math.sqrt((ind[0]-i)**2 + (ind[1]-j)**2)
                # Ni,j is the topolgical neighborhood of the winner node
                N = math.exp(-d**2/(2*(sigma**2)))
                # w(new) = w(old) + alpha*N*(x-w(old))
                w[i][j] += alpha*N*(x-w[i][j])

    # Decrease the learning rate and sigma by the given scheme
    alpha = alpha_0 * math.exp(-epoch/max_epochs)
    sigma = sigma_0 * math.exp(-epoch/max_epochs)

    # Generate figures of the SOM grid after 20, 40, 100, 1000 epochs.
    plot_ind = [20, 40, 100, 1000]
    if epoch in plot_ind:
        print("Epoch Number: {}".format(epoch))
        plt.imshow(w)
        display.display(plt.gcf())

    epoch += 1
```

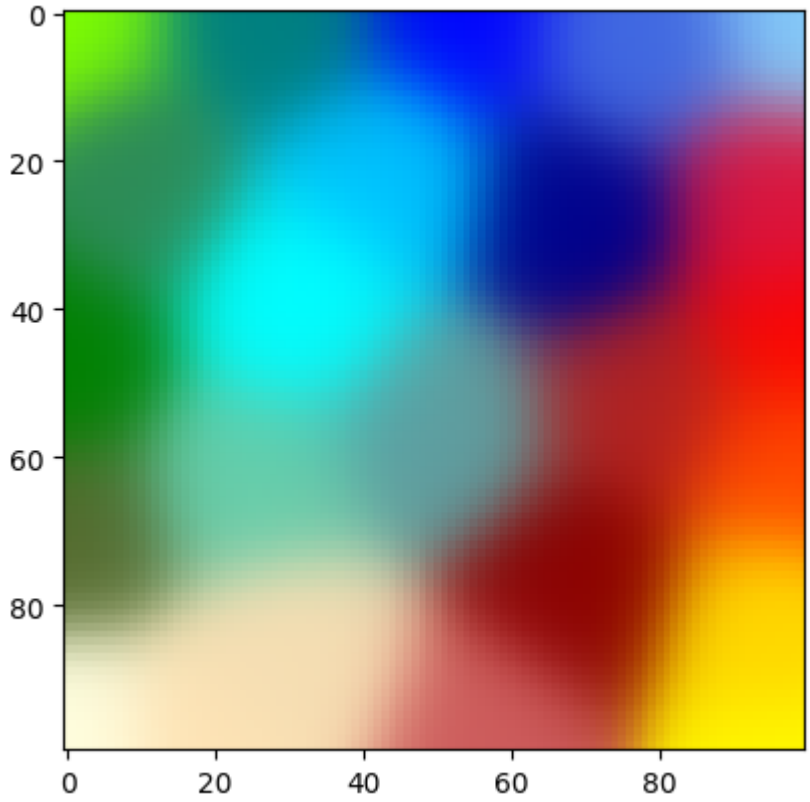
```
Epoch Number: 20
```



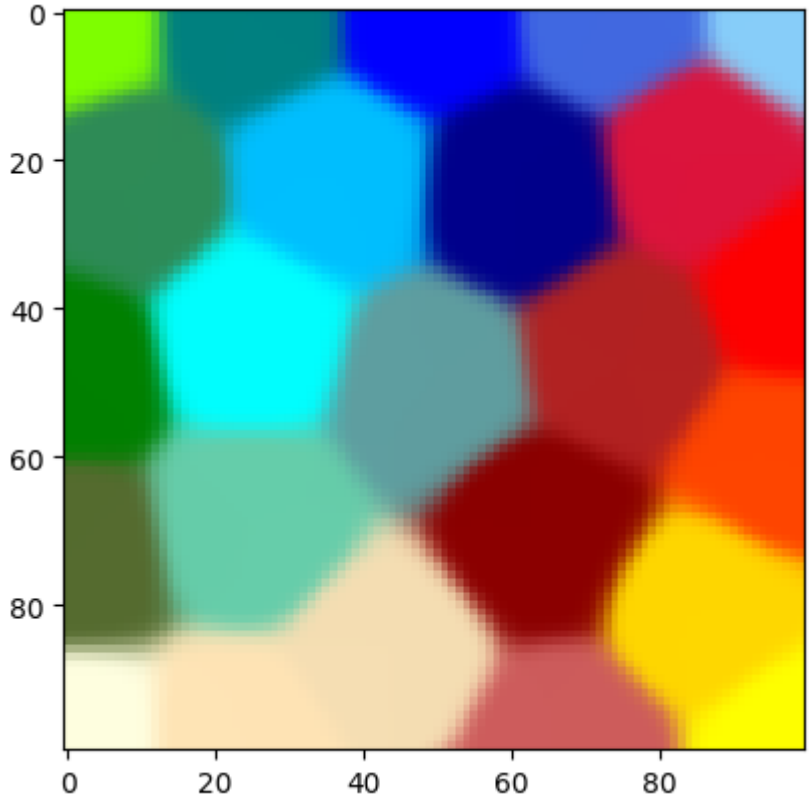

```
Epoch Number: 40
```

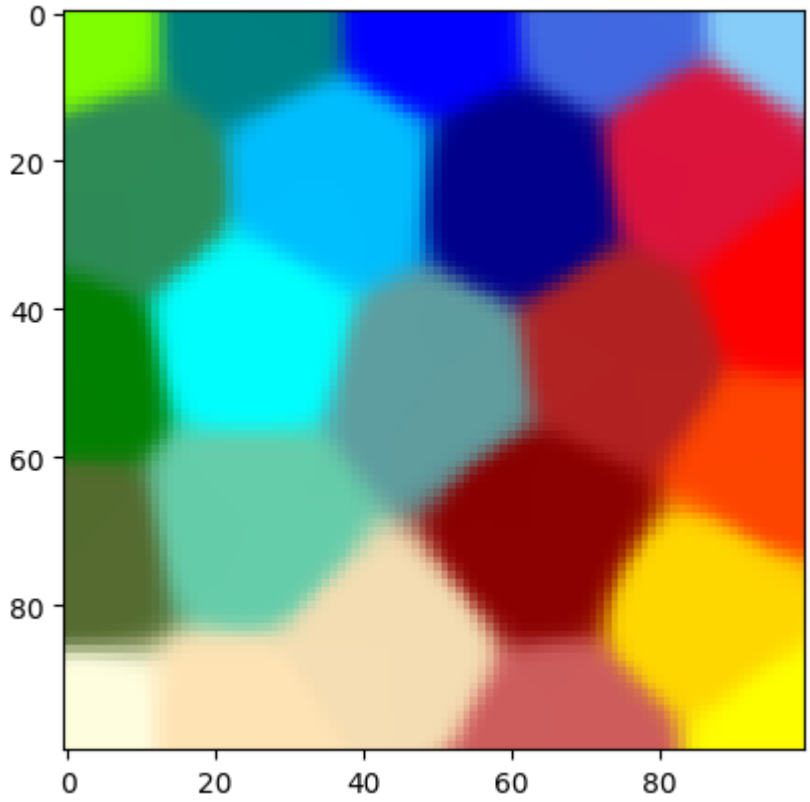



```
Epoch Number: 100
```




```
Epoch Number: 1000
```

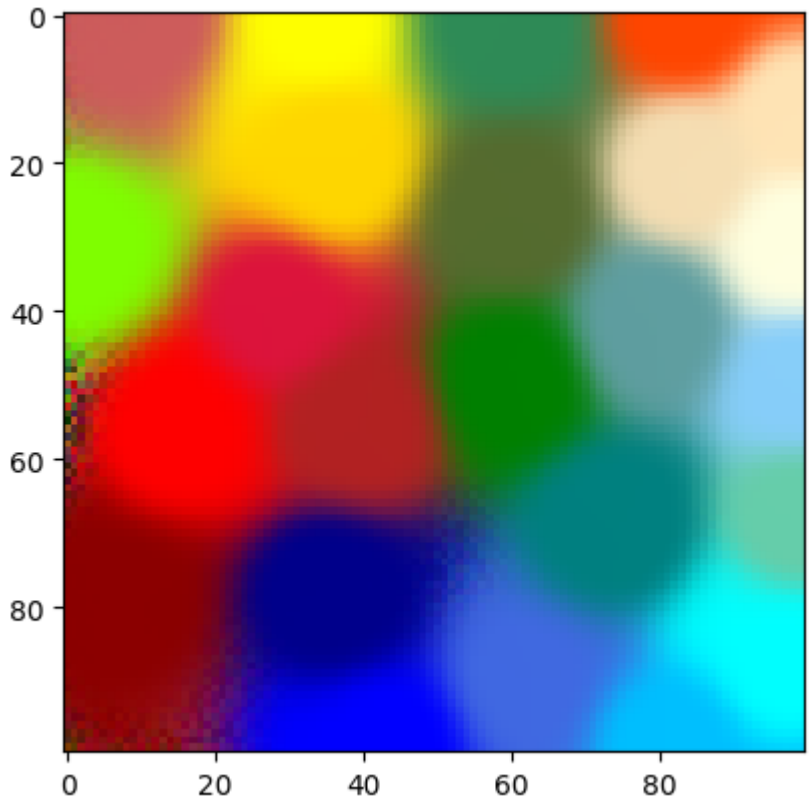




In [ ]:
```
from minisom import MiniSom
som = MiniSom(100, 100, 3, sigma=10.,
              learning_rate=0.8,
              neighborhood_function='gaussian')
som.train_random(normRGB, 1000, verbose=True)
plt.imshow(abs(som.get_weights()), interpolation='none')
```

```
 [ 1000 / 1000 ] 100% - 0:00:00 left
 quantization error: 6.149552678818614e-07
```

Out[ ]: <matplotlib.image.AxesImage at 0x1d93cdc5600>



2.b

There are 2 conclusions that we can draw from the here. First, as we increase the value of sigma(0) we can see that the each genereated cluster increase in radius. This is because this value is directly tied to the range of the topological neigbourhood, meaning a small sigma value will lead to only a small neighbourhood having its weight updated. This is why I used an initial sigma value of 10 instead of 1 since the original value is only showing clustering in a very limited range that cannot cover the entire 100x100 grid. Furthermore, as we increase in epoch counts, we can see an increase in clarity of the color segmentation, this is especially apparent between 100 and 1000 epochs. This is expected as a higher training count leads to higher accuracy of the model in general, especially for KSOM since the decrease in learning rate and sigma value as epoch increases allows for fine tuning the boundary between the clusters.