Image Approximation by means of Error Minimization

Kayla Rockhill

May 2021

Abstract

Function approximation has many uses across many areas of physics, mathematics, and computer science. It can be used to break up problems into more manageable pieces and create solutions that are easier to understand. Function approximation can also be used to simplify data which allows for a faster and easier transmission of the data. This method is also useful for clearing out any noise from data. An image can be represented as a two dimensional function and can therefore be approximated. To visualize the results of function approximation and gradient descent, the methods can be implemented in computing software. The platforms used in this paper were Octave and Python. The one and two dimensional cases of function approximation was implemented in both Octave and Python. The one and two dimensional cases for gradient descent were implemented in Python. Octave is a mathematical based platform while Python is a more general language with a wider range of applications. The function approximation method was implemented first in Octave and was then translated to Python. The translation to Python was done so that a comparison between the two could be made. Implementation into Python also allows for the incorporation of hardware since a serial con-

nection between Python and Arduino can be made. Both Python and Octave provided accurate approximations for various target functions in the one and two dimensional cases. An extension to using hardware was started but further work is needed to fix the issues that arose.

Contents

1 Introduction

An approximation of a function may be necessary for a variety of reasons. The approximation function is usually made up of simple components that are easy to understand and manipulate. This simplification can result in a smaller data set which allows for easier and faster transmission of the data. Function approximation may also be used to clean up data sets that have an abundance of noise. This noise can make the data difficult to analyze so removing it allows for easier readability. There are many methods of function approximation including the method of least squares and gradient descent. The method of least squares aims to minimize the squared difference between the target and approximation functions. This method finds the weights of the approximation function that best minimizes the squared difference. Gradient descent is an optimization algorithm that iteratively minimizes the squared difference between the target and approximation functions. Gradient descent finds new weights of the approximation function with every iteration and provides a better approximation of the target function after each iteration.

These methods of approximation can be tested using software. The methods can be implemented as code and can be used to estimate various functions. The target function, approximation function, and the difference between them can be graphed so the success of each approximation can be seen. These methods can be used in the one and two dimensional cases and both can be implemented into code. When using the two dimensional case, images can be used as a target function for approximation. This can allow for an easy comparison when looking at the target image compared to the approximation when testing the accuracy of the approximation methods.

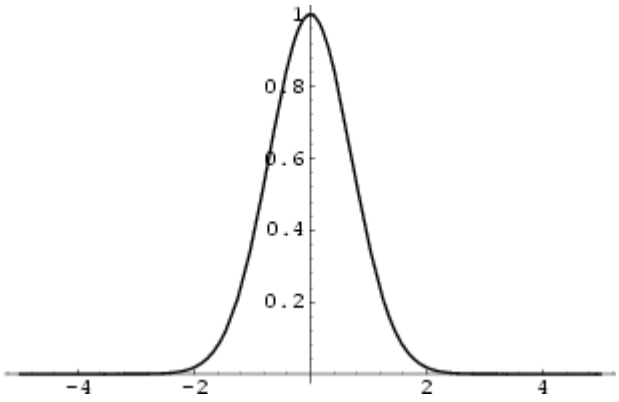

Figure 1: This is an example of a one dimensional Gaussian function located at the origin with an amplitude of one.

1.1 Theory of Function Approximation

Function approximation often arises in mathematics as a method of dividing a problem into smaller sections. The goal is generally to divide a complicated problem into sections that are easier to work with. Function approximation can also be used to estimate a relationship among data points, and therefore be used to develop prediction models. The summation of a particular type of function is often used to develop the approximation. For example, the function approximation can use the summation of Gaussian functions with varying amplitudes to approximate target functions. A Gaussian function can be described by

$$\phi(x) = e^{\frac{-(x-b)^2}{c}} \tag{1}$$

where b is the distance from the origin on the x axis and c is a constant that determines how wide the function is. The height can be adjusted by multiplying the function with different values. In this case, the amplitude is one. Figure 1 shows an example of a one dimensional Gaussian function located at the origin with an amplitude of one. A linear combination of these functions with varying

amplitudes can be used to approximate a target function.

The target function can be any function generally described by

$$f(x) : \mathbb{R} \longrightarrow \mathbb{R} \tag{2}$$

where f is a function of x. The summation function can be defined as

$$g(x) = \sum_{i=1}^{n} a_i \phi_i(x) \tag{3}$$

where ϕ_i is each Gaussian function, and a_i is each amplitude. In the one dimensional case, the Gaussian functions can be distributed along the x axis and the amplitudes to create the closest approximation of the target function can be found. The amplitudes for each of the Gaussian functions are found by doing an error minimization analysis. To find the cumulative error between the target function and the approximation, the integral of the difference between the target function $f(x)$ and the approximation function $g(x)$ is taken which is represented by

$$\text{err} = \frac{1}{2} \int_a^b [f(x) - g(x)]^2 dx \tag{4}$$

where a and b are the horizontal bounds of the function. The difference between $f(x)$ and $g(x)$ is squared the ensure a positive difference and the $\frac{1}{2}$ is used as convention because a derivative will be taken. A visual of this is shown by Figure 2. To find each amplitude, the partial derivative of the error function is taken with respect to each amplitude and set equal to zero since the goal is to minimize the error.

$$\frac{\partial err}{\partial a_i} = \frac{1}{2} \int_a^b \frac{\partial}{\partial a_i} [f(x) - g(x)]^2 dx = 0 \tag{5}$$

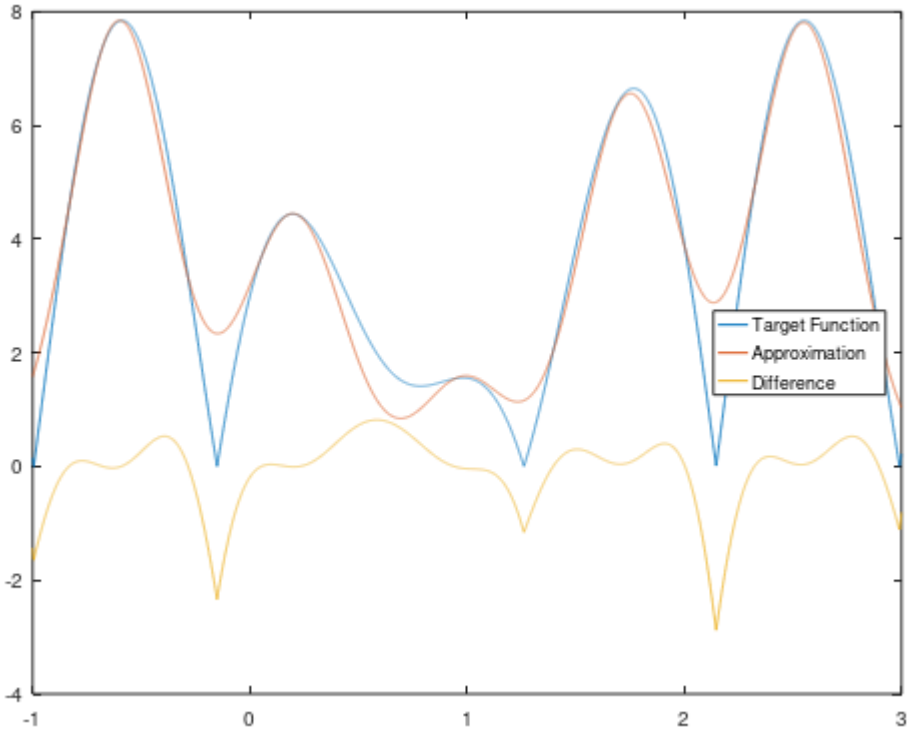

Figure 2: This is a representation of the target function $y = |3\cos(5x)+5\sin(3x)|$ (blue line) being approximated using a summation of Gaussian functions (red line) along with the difference of the two functions. The approximation function was $g = 7.8399 * e^{-(x+0.6)^2*10} + 4.4364 * e^{-(x-0.2)^2*10} + 1.5672 * e^{-(x-0.99)^2*10} + 6.5491 * e^{-(x-1.75)^2*10} + 7.7997 * e^{-(x-2.55)^2*10}$. The difference function remains around zero at the peaks of the target and approximation functions but drops down at the sharp points of the target function. Sharp edges can be difficult to approximate especially when solely using a sum of Gaussian functions. The main goal was to accurately approximate the peaks.

Evaluating partial derivative of Equation 5, the resulting equation is

$$0 = \int_a^b [f(x) - g(x)](-\phi_i(x))dx \tag{6}$$

which can then be rewritten as

$$0 = -\int_a^b f(x)\phi_i(x)dx + \sum_{j=1}^{n} a_j \int_a^b \phi_j(x)\phi_i(x)dx \tag{7}$$

Now let

$$b_i = \int_a^b f(x)\phi_i(x)dx \tag{8}$$

and

$$w_{ij} = \int_a^b \phi_i(x)\phi_j(x)dx \tag{9}$$

Thus, equation 7 becomes

$$0 = -b_i + \sum_{j=1}^{n} a_j w_{ij} \tag{10}$$

Each i represents a partial derivative with respect to a new amplitude. The collection of equations can be written as a matrix expression

$$\begin{pmatrix} w_{11}, w_{12}, ..., w_{1n} \\ w_{21}, w_{22}, ..., w_{2n} \\ ... \\ w_{n1}, w_{n2}, ..., w_{nn} \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ ... \\ a_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ ... \\ b_n \end{pmatrix} \tag{11}$$

with

$$W = \begin{pmatrix} w_{11}, w_{12}, ..., w_{1n} \\ w_{21}, w_{22}, ..., w_{2n} \\ ... \\ w_{n1}, w_{n2}, ..., w_{nn} \end{pmatrix} \tag{12}$$

$$A = \begin{pmatrix} a_1 \\ a_2 \\ ... \\ a_n \end{pmatrix} \tag{13}$$

$$B = \begin{pmatrix} b_1 \\ b_2 \\ ... \\ b_n \end{pmatrix} \tag{14}$$

The amplitudes are then found by the following equation

$$A = W^{-1}B \tag{15}$$

where the result matrix is the approximate amplitudes for each of the Gaussian functions. A very similar process was done for the two dimensional case with the main difference being that a double integral of the difference function was taken. Thus, equation 5 becomes

$$\text{err} = \frac{1}{2}\int_c^d \int_a^b [f(x,y) - g(x,y)]^2 dxdy \tag{16}$$

where a and b are the x bounds and c and d are the y bounds. This results in

$$b_i = \int_c^d \int_a^b f(x,y)\phi_i(x,y)dxdy \tag{17}$$

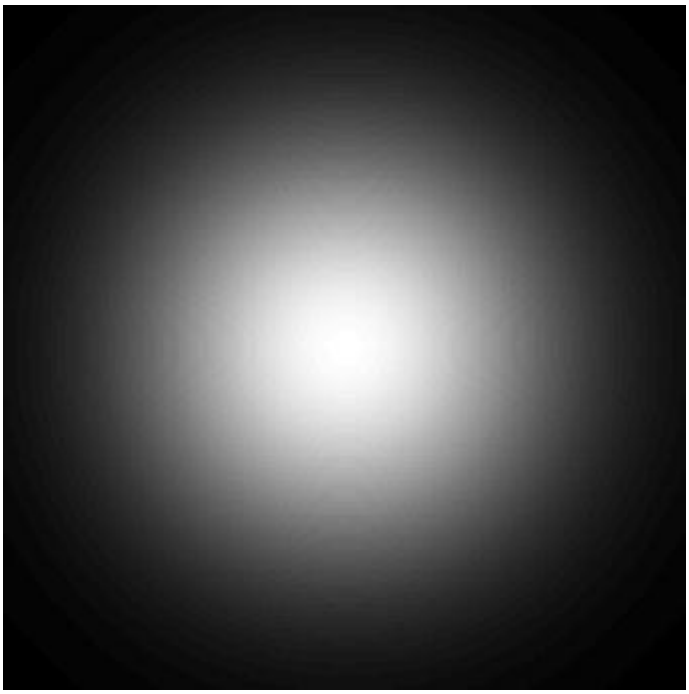

Figure 3: This is an example a two dimensional Gaussian function with an amplitude of one.

and

$$w_{ij} = \int_c^d \int_a^b \phi_i(x,y)\phi_j(x,y)dxdy \tag{18}$$

The rest of the analysis is then the same.

A two dimensional Gaussian function can be written as

$$\phi(x,y) = ae^{\frac{-(x-b)^2-(y-c)^2}{d}} \tag{19}$$

where b is the distance from the origin along the x axis, c is the distance from the origin along the y axis, and d is the width of the function. Once again, the amplitudes in a linear combination of the functions can be adjusted to approximate a target function. Figure 3 shows a two dimensional Gaussian function with an amplitude of one.

1.2 Gradient Descent

Gradient descent is an optimization algorithm that is used to minimize an error function. This algorithm takes small steps in the direction of the negative gradient of the error function until it reaches the minima. This means that it walks downhill along the slope until it reaches the lowest point. The learning rate η is used to determine the size of the step, and as the minima of the function is approached the size of the step decreases. An example of this is shown in Figure 4. An appropriate value for the learning rate must be chosen since a very small value will cause the algorithm to require many iterations to meet the minima while a very large learning rate will cause the algorithm to overshoot the minima. The learning rate is adjusted depending on the function the is being minimized, but typically values of 0.1, 0.01, or 0.001 are used. With every iteration, the parameters of a function are adjusted and the gradient is recalculated. This is repeated until the cost function has reached a minima.

Gradient descent is used to find these coefficients when they can't be solved for analytically, and the function must be differentiable. In this case, radial basis functions are used to approximate a target image. A radial basis function is a function that only depends on its distance from the origin. An example function, as used before, is a Gaussian function. A linear combination of these functions can be used to estimate a target function. The estimation is done by finding the weights on each of the radial basis functions that minimize the error function. Doing this, takes a complex function and makes it more readily understood so the it becomes easier to manipulate. This method can be derived by starting with an error function.

$$E = \sum_{i=1}^{m} \frac{1}{2}(y_i - \hat{y}_i)^2 \tag{20}$$

Here, y is the target function and $\hat{y}$ is the approximation function. Since in

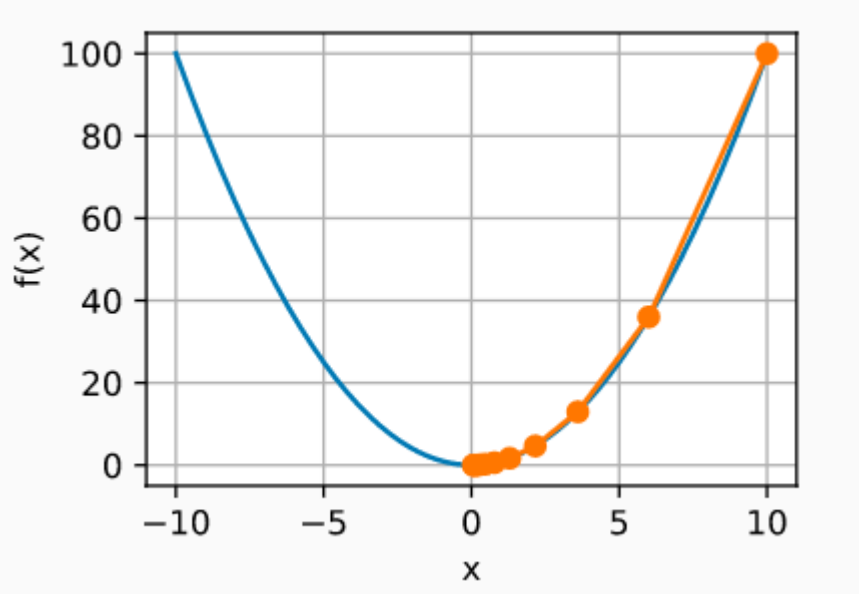

Figure 4: An example graph of gradient descent being used to approach the minima of a cost function f(x). Each point represents another iteration of the algorithm and therefore another step. As the steps approach the minima the step size decreases.

this case the error should always be positive, the difference between the target function and approximation function is squared. This makes the error positive and creates a greater error for outlying points. To get the error for all data points, the sum of the errors for every data point can be found. This equation is referred to as the sum of squared errors. The $\frac{1}{2}$ is used as a convention since the derivative will be taken at a later point. This is the same idea used with the function approximation method except a summation is used instead of an integral. The approximation function can be written more specifically with

$$\hat{y} = \sum_{j=1}^{n} a_j \phi_j(x) \tag{21}$$

Where $\phi_j(x)$ represents a Gaussian function and a_j is the amplitude of the Gaussian function. Equation 20 then becomes

$$E = \sum_{i=1}^{m} \frac{1}{2}(y_i - \sum_{j=1}^{n} a_j \phi_j(x))^2 \tag{22}$$

The goal is to take small steps towards the minima of the function. To do this, each coefficient of $\phi_j(x)$ can be updated by

$$a_j = a_j + \Delta a_j \tag{23}$$

It can be said that

$$\Delta a_j = -\nabla E \tag{24}$$

Meaning that the change in a_j should be in the direction of the negative gradient. Therefore,

$$\Delta a_j \propto -\frac{\partial E}{\partial a_j} \tag{25}$$

The relationship can be rewritten as an equality by multiplying by the size of

the descent steps.

$$\Delta a_j = -\eta \frac{\partial E}{\partial a_j} \tag{26}$$

Where η is known as the learning rate. This derivative can be represented by

$$\frac{\partial E}{\partial a_j} = \frac{\partial}{\partial a_j}(\sum_{i=1}^{m} \frac{1}{2}(y_i - \sum_{j=1}^{n} a_j \phi_j)^2) \tag{27}$$

Taking this derivative with the chain rule results in

$$\frac{\partial E}{\partial a_j} = (y - \hat{y})(\frac{\partial y}{\partial a_j} - \frac{\partial \hat{y}}{\partial a_j}) \tag{28}$$

This becomes

$$\frac{\partial E}{\partial a_j} = (y - \hat{y})(0 - \frac{\partial \hat{y}}{\partial a_j}) \tag{29}$$

Simplifying,

$$\frac{\partial E}{\partial a_j} = -(y - \hat{y}) \left(\frac{\partial \hat{y}}{\partial a_j} \right) \tag{30}$$

The partial derivative of $\hat{y}$ can be written as

$$\frac{\partial \hat{y}}{\partial a_j} = \frac{\partial}{\partial a_j} \sum_{k=1}^{n} a_k \phi_k \tag{31}$$

This can be rewritten as

$$\frac{\partial \hat{y}}{\partial a_j} = \frac{\partial}{\partial a_j}[a_1\phi_1 + a_2\phi_2 + ... + a_n\phi_n] \tag{32}$$

Each partial derivative can be calculated by

$$\frac{\partial}{\partial a_1}[a_1\phi_1 + a_2\phi_2 + ... + a_n\phi_n] = \phi_1 + 0 + ... + 0 = \phi_1 \tag{33}$$

$$\frac{\partial}{\partial a_2}[a_1\phi_1 + a_2\phi_2 + ... + a_n\phi_n] = 0 + \phi_2 + ... + 0 = \phi_2 \tag{34}$$

$$\frac{\partial}{\partial a_n}[a_1\phi_1 + a_2\phi_2 + ... + a_n\phi_n] = 0 + 0 + ... + \phi_n = \phi_n \tag{35}$$

This can be represented as

$$\frac{\partial}{\partial a_j}\sum_{k=1}^{n} a_k\phi_k = \phi_j \tag{36}$$

Rewriting equation 30 using equations 31 and 36 becomes

$$\frac{\partial E}{\partial a_j} = -(y - \hat{y})\phi_j \tag{37}$$

Equation 26 becomes

$$\Delta a_j = \eta(y - \hat{y})\phi_j \tag{38}$$

Thus, equation 23 becomes

$$a_j = a_j + \eta(y - \hat{y})\phi_j \tag{39}$$

And putting the summation notion back in,

$$a_j = a_j + \sum_{i=1}^{m} \eta(y_i - \hat{y_i})\phi_j \tag{40}$$

This is the equation to iteratively find the value of the amplitudes for each of the Gaussian functions in the approximation function. This method can be implemented into Python code so the approximations of functions can be visualized. Figure 5 shows a target function in blue that is being approximated by using gradient descent. The approximation function for 200 iterations is shown in red. Each iteration calculates a new approximation function which is ideally a better approximation than the one before it. The earlier approximations are shown in a lighter red and with each iteration the color darkens. The darkest color is the closest approximation to the target, showing that every iteration

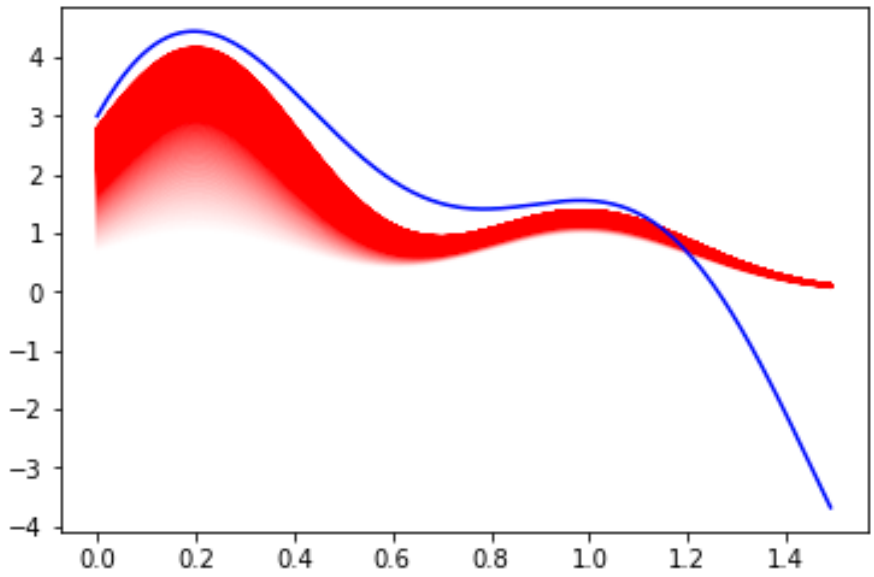

Figure 5: This image shows a target function in blue and each of the 200 estimations of the target function in red. The color of the estimations becomes a darker red with each iteration. The color darkens as the approximation function approaches the target function which shows that the later estimations are a much better approximation than the earlier estimations. The tail ends diverge showing that a perfect approximation isn't always possible and there are limitations to this method.

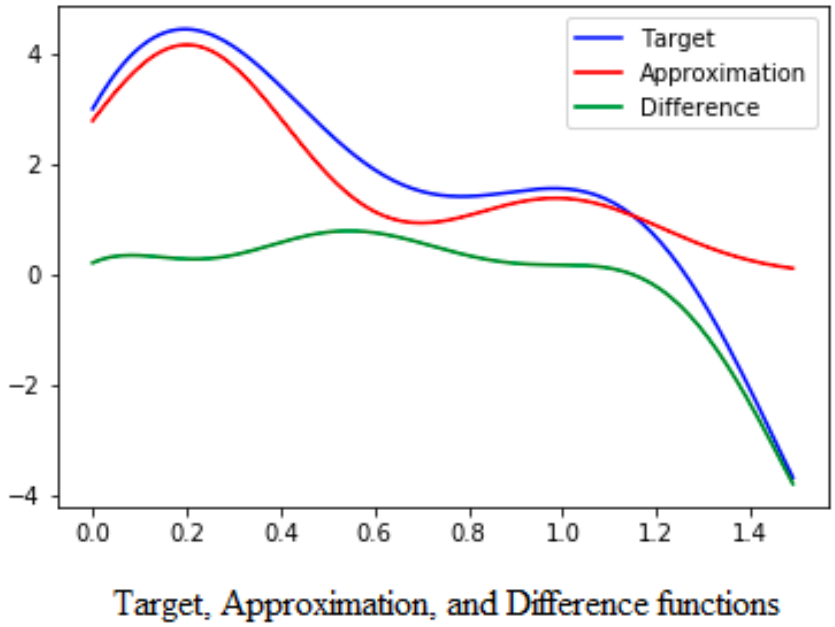

Figure 6: This graph shows the target function and the current approximation along with the difference between the two. The target function used was $y = 3\cos(5x) + 5\sin(3x)$ and the found approximation function was $\hat{y} = 4.1582 * e^{-(x-0.2)^2*10} + 1.3760 * e^{-(x-0.99)^2*10}$. Most of the difference function is approximately showing that the approximation was a good estimation of the target function even though the tail ends diverge.

was a closer approximation. Figure 6 shows the target function with the approximation and the difference between the two functions. The goal is to get the difference function to be zero and the difference function stays right around zero and only diverges at the very end.

This method can be extended to the two dimensional case to approximate approximate images. The theory behind the two dimensional case uses the same methods as described in the one dimensional case. When approximating images, a grid of Gaussian functions can be created with initial amplitudes of one. The gradient descent algorithm can the adjust each amplitude to best approximate

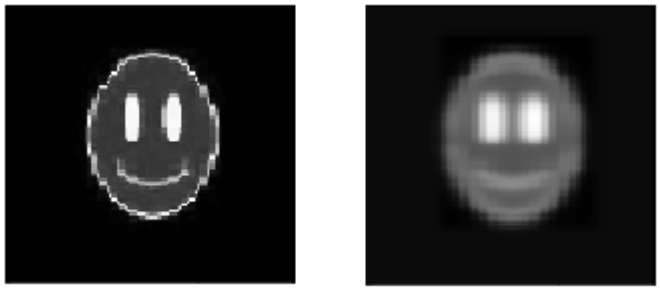

Figure 7: The image on the left is the target image and the image on the right is the approximation of the target image using 700 Gaussian functions after 100 iterations. The approximation image is slightly blurred but is still an accurate representation of the target image.

the target. Figure 7 shows a target image and the approximation after 100 iterations using 700 Gaussian functions.

2 Coding

The graphs shown for each of the figures were developed either in Octave or in Python. Octave is a software that is used primarily for numerical computations. There are many tools for solving mathematical problems and graphing results. The method of function approximation involves a great deal of linear algebra for which the coding process within Octave is intuitive. For these reasons, function approximation codes were written in Octave. The first code used five Gaussian functions to approximate a one-dimensional target function. This code was then extended to the two dimensional case so that images could be approximated. After this point, the Octave code and all other code was written in Python. By implementing the function approximation method into Python, a comparison between the approximations made in Octave and Python could be made. Implementation in Python also allows for the inclusion of hardware. Python can

establish a serial connection with Arduino so any work done in Python can be extended to Arduino. Python has a wider range of applications than Octave and is therefore less mathematically based. Because of this, there are more steps involved when coding function approximation. While the implementation is more challenging, the same goals can still be achieved.

2.1 Octave Code

The Octave code for the one and two dimensional cases are described in Appendix A.

2.1.1 One Dimensional Function Approximation

For the one dimensional case, the first section of code

```
dx = 0.001;
x = [-1:dx:3];
y = abs(3*cos(x*5)+5*sin(x*3)); % target function to approximate.
ea = exp((-(x+0.6).^2)*10);
eb = exp((-(x-0.2).^2)*10);
ec = exp((-(x-0.99).^2)*14);
ed = exp((-(x-1.75).^2)*10);
ee = exp((-(x-2.55).^2)*10);
```

defines the different x values that the functions will be evaluated at, the target function y, and five Gaussian functions along the x-axis. The next section

```
bf = [ea; eb; ec; ed; ee];

W = [];
B = [];
```

puts each of the Gaussian functions into a basis function matrix and defines two empty matrices W and B. These will serve the same purpose as the W and B

matrices described in the function approximation method. The for loops

```
for i = 1:5
  for j = 1:5
    W(i,j) = sum(bf(i,:).*bf(j,:)*dx);
  endfor
endfor

for i = 1:5
   B(i,1) = sum(bf(i,:).*y.*dx);
endfor
```

calculate the values for the W and B matrices by using equations 9 and 8. In the final section,

```
A = inv(W) * B;
plot(x,y, x,(A(1,1)*ea+A(2,1)*eb+ A(3,1)*ec+A(4,1)*ed+A(5,1)*ee),
x, y-(A(1,1)*ea+A(2,1)*eb+ A(3,1)*ec+A(4,1)*ed+A(5,1)*ee))
legend ({"Target Function", "Approximation", "Difference"},
"location", "east");
```

the amplitudes of each Gaussian function are found and the target, approximation, and difference functions are plotted. The resulting graph is seen in Figure 2.

2.1.2 Two Dimensional Function Approximation

The two dimensional case approximates an image. The first section of code

```
clf;
colormap gray;
axis xy;
dx=1;
dy=1;
x = 1:dx:100;
y = 1:dy:75;
[xx,yy]=meshgrid(x,y);
```

defines the length of the x and y axes, the number of points that will be evaluated for each variable, and sets up the two dimensional grid of points. Since an image will be used as a target function,

```
m = imread('test2.jpg');
n = imresize(m,[75,100]);
gr = rgb2gray(n);
c = imcomplement(gr);
```

the image must be imported and resized. For this project, the gray-scale and complement of the image were taken. Many Gaussian functions are needed to accurately represent images, so a grid of evenly distributed Gaussian functions is created with

```
mu = [];
x1 = 30;
k1 = 1;
xnum = 9;
ynum = 10;
total = xnum*ynum;
for i = 1:xnum
    for j = 1:ynum
        mu(1,k1)= x1;
        k1 = k1+1;
    endfor
    x1 = x1+5;
endfor

k2 = 1;
for i = 1:xnum
y1 = 10;
    for j = 1:ynum
        mu(2,k2) = y1;
        k2 = k2+1;
        y1 = y1+5;
    endfor
endfor

mu2 = mu';
```

```
brightness = 10;

bf = [];
for i = 1:total
    bf(i,:) = exp(-((xx(:)-mu2(i,1)).^2)/brightness-((yy(:)
    -mu2(i,2)).^2)/brightness);
endfor
```

Where there are 90 total Gaussian functions that start from the point (30,10) and end at (75,60) forming a grid that is 9 Gaussian functions wide and 10 Gaussian functions high. The first two sets of nested for loops find the x and y coordinates of each Gaussian function and store them in the mu matrix. The last for loop creates a Gaussian function for each point and stores them in a basis function matrix. The next block of code

```
Tr=c(:)';
W = [];
B = [];

for i = 1:total
  for j = 1:total
    W(i,j) = sum(sum(bf(i,:).*bf(j,:)*dx)*dy);
  endfor
endfor

for i = 1:total
   B(i,1) = sum(sum(bf(i,:).*Tr.*dx)*dy);
endfor

A = inv(W) * B;
```

has the same idea as the one dimensional case where the first set of for loops finds the values for the W matrix and the final for loop finds the values for the B matrix. A double sum is now used because the integration is over two dimensions instead of one. The last line finds the amplitudes of each Gaussian function. In the final section of code

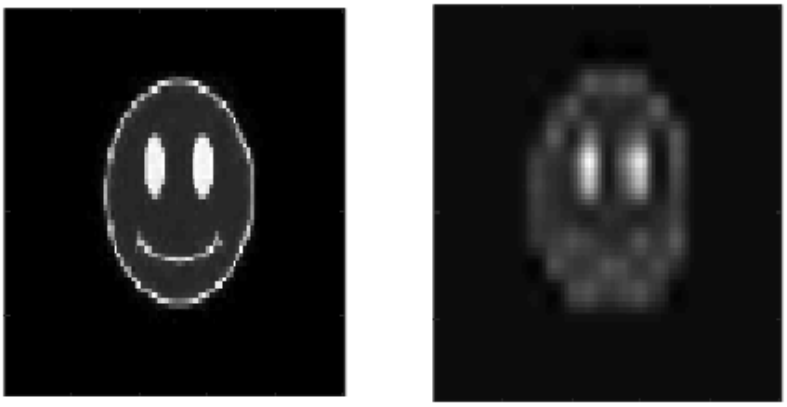

Figure 8: The image on the left is the target image after being edited for approximation. The image on the right is the approximation of the target image using a grid of 90 Gaussian functions. Since only 90 Gaussian functions were used, the approximation image is pixelated but still accurately represents the target image.

```
sum1 = 0;
for i = 1:total
  sum1 = sum1 + bf(i,:)*A(i,1);
endfor

sum1 = reshape(sum1, [75,100]);

subplot(1,2,1)
imagesc(x, y, c)
subplot(1,2,2)
imagesc(x, y, sum1)
colormap gray;
```

the approximation function is created by multiplying each Gaussian function by its corresponding amplitude and summing all the functions together. The target image and approximation image are then plotted and shown in Figure 8.

Using the two dimensional case, the distribution of Gaussian functions was

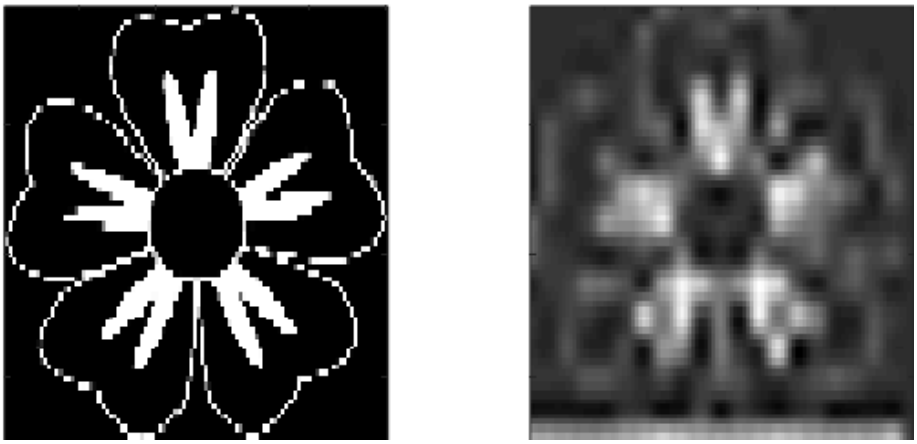

Figure 9: This is another example of an image approximation using Octave.

adjusted from a grid to a random placement. This was done by using a random number generator to determine the placement of Gaussian functions. Figure 10 shows the output of this method using 1000 randomly placed Gaussian functions.

2.2 Python Code

The final Python codes for function approximation in the one and two dimensional cases can be found in Appendix B. The Gradient Descent code is found in Appendix C. To translate the Octave code into the Python language, the same steps that were applied in building the Octave code were used. First, the one dimensional function approximation was translated into Python. An example output of this Python code is shown in Figure 11. The Python packages numpy and matplotlib where used for the development since both function similarly to how Octave functions. All matrices were converted to numpy matrices to allow for manipulation, and all graphs were plotted using matplotlib.

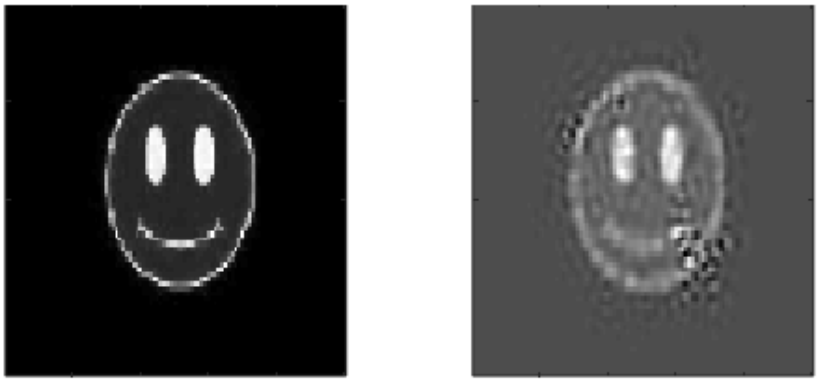

Figure 10: The image on the left is the target image after being edited for approximation. The image on the right is the approximation of the target image using 1000 randomly placed Gaussian functions. Since 1000 Gaussians were used, the approximation image is of high quality and is a very accurate estimate of the target image.

2.2.1 One Dimensional Function Approximation

The first section of code is the same as before

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

dx = 0.01
x = np.arange(-1., 3., dx)
y = 3*np.cos(5*x)+ 5*np.sin(3*x)

ea = np.exp((-(x+0.6)**2)*10)
eb = np.exp((-(x-0.2)**2)*10)
ec = np.exp((-(x-0.99)**2)*10)
ed = np.exp((-(x-1.75)**2)*10)
ee = np.exp((-(x-2.55)**2)*10)

bf = [[ea],
      [eb],
      [ec],
      [ed],
      [ee]]
```

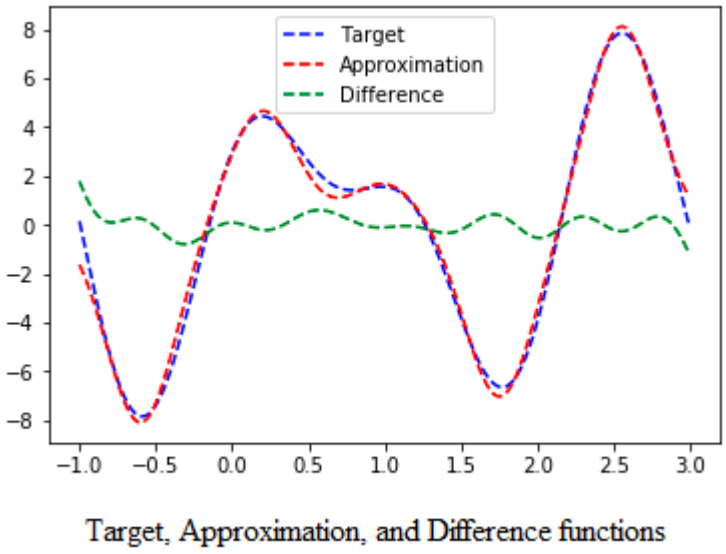

Figure 11: This is an example of a one dimensional function approximation using Python. The target function of $y = 3\cos(5x) + 5\sin(3x)$ is shown in blue, the approximation of the target with function $g = -8.1050 * e^{-(x+0.6)^2*10} + 4.6712 * e^{-(x-0.2)^2*10} + 1.6849 * e^{-(x-0.99)^2*10} - 7.0481 * e^{-(x-1.75)^2*10} + 8.1259 * e^{-(x-2.55)^2*10}$ is shown in red, and the difference between the two is shown in green. The difference function is almost zero for most of the graph and only diverges slightly at the ends showing that the approximation function is a good estimation of the target.

with the exception of a few import statements. Since Python is less mathematically based than Octave, these import statements are needed to run packages that allow for numerical computations and graphing. The rest of the section is essentially the same in that it defines the target function, sets up each of the Gaussian functions, and puts each Gaussian into a basis function matrix. The next step is the same as before

```
bf = np.array(bf)
W = np.empty((5,5))

for i in range(5):
    for j in range(5):
        W[i][j] = np.sum((bf[i]*bf[j]*dx))

W = np.array(W)

B = np.empty((5,1))
for i in range(5):
    B[i][0] = np.sum((bf[i]*y*dx))
B = np.array(B)
```

where the values for the W and B matrices are calculated. Using these,

```
inv = np.linalg.inv(W)
A = inv.dot(B);
print(A)
fig, ax = plt.subplots()
ax.plot(x, y,'b--', label='Target')
ax.plot(x, A[0][0]*ea+A[1][0]*eb+A[2][0]*ec+A[3][0]*ed+A[4][0]
*ee,'r--',
label='Approximation')
ax.plot(x, y-(A[0][0]*ea+A[1][0]*eb+A[2][0]*ec+A[3][0]*ed+A[4][0]
*ee),'g--',
label= 'Difference')
leg = ax.legend();
```

the amplitudes for each of the Gaussians can be found and the results are

plotted. The resulting graph can be seen in Figure 11.

2.2.2 Two Dimensional Function Approximation

For the image approximation, the Pillow and matplotlib packages were used to manipulate the image so it could be approximated. Once again, the first section of code sets up the xy plane and imports the image. The image must then be adjusted to allow for approximation.

```
import matplotlib.image as mpimg
from matplotlib import cm
from PIL import Image
import numpy as np
dx = 1
dy = 1
x = np.arange(0., 75., dx)
y = np.arange(0., 100., dy)
xx, yy = np.meshgrid(x, y)

print(xx.shape, yy.shape, (xx.flatten()).ndim)

def rgb2gray(rgb):
    return np.dot(rgb[...,:3], [0.299, 0.587, 0.144])

m = Image.open("test2.jpg")
n = m.resize((75, 100))
n = np.array(n)

gray = rgb2gray(n)
c=255-gray
n = np.array(c)
print(n.shape)
plt.imshow(c, cmap = plt.get_cmap('gray'))

plt.show()
```

The manipulation of the image is slightly more involved with Python than with Octave but the same results are achieved. A grid of Gaussians is created by

```
xnum = 17
ynum = 18
total = xnum*ynum
mu = np.empty((2,total))
x1 = 20
k1 = 0
for i in range(xnum):
    for j in range(ynum):
        mu[0][k1]= x1
        k1 = k1+1
    x1 = x1+2

k2 = 0;
for i in range(xnum):
    y1 = 10;
    for j in range(ynum):
        mu[1][k2] = y1;
        k2 = k2+1;
        y1 = y1+4;

mu2 = np.transpose(mu)

brightness = 5;
bf = list()

for i in range(total):
    curr_bf = np.exp(-(xx-mu2[i][0])**2/brightness-(yy-mu2[i][1])
    **2/brightness)
    bf.append(curr_bf)
```

This results in a 17x18 grid of Gaussians that starts at the point (20, 10) and ends at (54, 82). The Gaussians are stored in a basis function matrix which is then used to find the values of the W and B matrices.

```
Tr = np.empty((7500,1))
Tr = (c)

W = np.empty((total, total))
B = np.empty((total,1))

for i in range(total):
```

```
    for j in range(total):
        W[i][j] = np.sum((np.sum((bf[i]*bf[j]*dx))*dy))

for i in range(total):
    B[i][0] = np.sum((np.sum((bf[i]*c*dx))*dy))

inv = np.linalg.inv(W)

A = inv.dot(B)
```

The values for the W and B matrices are then used to find the amplitudes and the approximation function can be formed.

```
sum1 = 0;
for i in range(total):
    sum1 = sum1 + bf[i]*A[i][0]

sum1 = np.reshape(sum1, (100,75))
sum1 = np.array(sum1)

#fig, (ax1, ax2) = plt.subplots(2)
#ax1.imshow(c, cmap = plt.get_cmap('gray'))
plt.imshow( sum1, cmap=cm.Greys_r)
plt.show()
```

Once the approximation function is formed, the resulting image is graphed and shown in shown in Figure 12.

2.2.3 One Dimensional Gradient Descent

Gradient Descent was first implemented in the first dimension. The first part of the code is the same as before

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

dx = 0.01
x = np.arange(0, 1.5, dx)
```

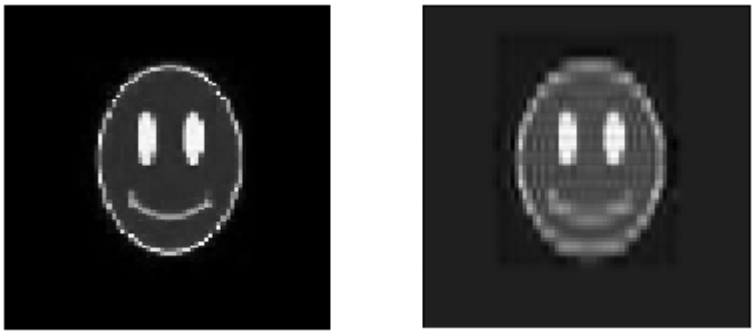

Figure 12: The graph on the left shows the target image and the graph on the right shows the approximation image developed in Python using 306 Gaussian functions.

```
y = 3*np.cos(5*x)+ 5*np.sin(3*x)
plt.plot(x,y,'b-')

phi_1 = np.exp((-(x-0.2)**2)*10)
phi_2 = np.exp((-(x-0.99)**2)*10)

w_1=1
w_2=1

y_hat=(w_1*phi_1)+(w_2*phi_2)

plt.plot(x,y_hat,'r-')
```

where y is the target function, phi_1 and phi_2 are Gaussian functions, w_1 and w_2 are each amplitude, and y_hat is the current approximation function. The next section of code begins the implementation of the Gradient Descent algorithm.

```
learn = 0.4

def summation(y_hat, x_range, y):
    total1 = 0
    total2 = 0
```

```
    total1 = np.sum((((y-y_hat)*phi_1)*0.1))
    total2 = np.sum((((y-y_hat)*phi_2)*0.1))

    return total1 / len(x_range), total2 / len(x_range)
```

The learning rate was set to 0.4 and the summation function calculates part of equation 38. These values can then be used to find the amplitude of each function.

```
for i in range(200):
    s1, s2 = summation(y_hat, x, y)
    w_1 = w_1 + learn * s1
    w_2 = w_2 + learn * s2
    y_hat=(w_1*phi_1)+(w_2*phi_2)
    plt.plot(x,y_hat, color='r', alpha=i/200)
plt.plot(x, y, 'b-')
```

This section implements the function and equation 40 to find the amplitudes for both Gaussian functions. Every iteration of this for loop creates a better approximation of the function and this for loop iterates 200 times to create the graph seen in Figure 7. The lighter red color shows earlier approximations of the function and the darkest red shows the final approximation. The final approximation is much closer to the target function than the first approximation.

2.2.4 Two Dimensional Gradient Descent

Gradient Descent was also used to estimate a target image. The setup of the code was very similar to the other two dimensional python code. The image was first adjusted for approximation and a grid of Gaussian functions was setup and put into a basis function so the approximation function y_hat could be set up. After the initial setup, the algorithm could then be implemented.

```
learn = 0.01
total1 = list()

for i in range(100):
    for j in range(total):
        curr_total = np.sum((((y-y_hat1)*bf[j])*0.1))
        avg = curr_total / len(xx)
        total1.append(avg)

    for k in range(total):
        vals[k][0] = vals[k][0] + learn * total1[k]
        y_hat = y_hat + vals[k][0]*bf[k]
```

The learning rate was set to 0.01 and the for loops are used to calculate part of Equation 38 and Equation 40 so that the target function can be updated and improved. The remainder of the code

```
y_hat = np.reshape(y_hat, (100,75))
y_hat = np.array(y_hat)

fig, (ax1, ax2) = plt.subplots(2)
ax1.imshow( y, cmap=cm.Greys_r)
ax2.imshow( y_hat, cmap=cm.Greys_r)
plt.show()
```

reshapes the target function and forms the graphs shown in Figure 7.

3 Arduino Setup

An Arduino board and a Neopixel grid can be used to display the approximation image. An Arduino is a microcontroller that is used to control various types of hardware. A Neopixel grid is an 8x8 grid of LED lights that can be controlled with an Arduino. The brightness of each LED light can be adjusted to form an image. Each LED would serve the same purpose as a Gaussian function and the brightness of each LED would be similar to the amplitudes on each

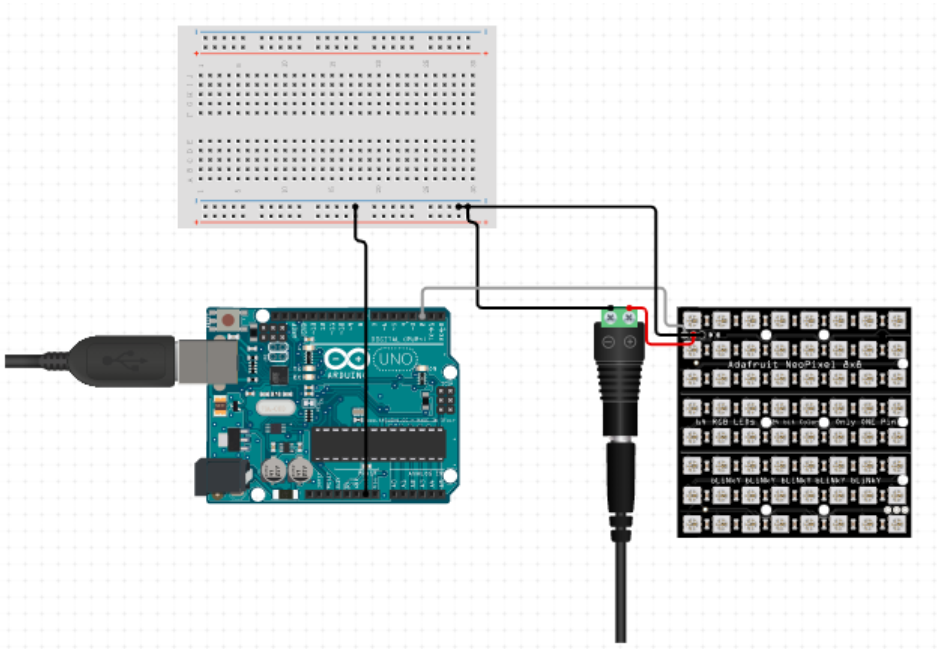

Figure 13: This is a wiring diagram of the NeoPixel grid and the Arduino. A resistor was also used between the data wire of the grid and the Arduino, and a capacitor was used between the power source and the grid.

Gaussian function. The amplitudes found using the gradient descent method can be manipulated and sent to Arduino by means of a serial connection. The amplitudes must be adjusted to correspond to a particular brightness value before they are sent to Arduino.

Figure 13 shows a wiring diagram of the Arduino board and the NeoPixel grid. A resistor were also used between the data wire of the grid and the board along with a capacitor between the grid and the power source. Figure 14 shows a labeled picture of the Ardunio setup. The Neopixel grid has four wires: two ground, one power, and one data. A capacitor is used between the power wires and the power supply to protect the grid from any power surges or fluctuations in power that may occur. An outside power supply is used so that the grid has enough current to power the LEDs without drawing it from the USB port on

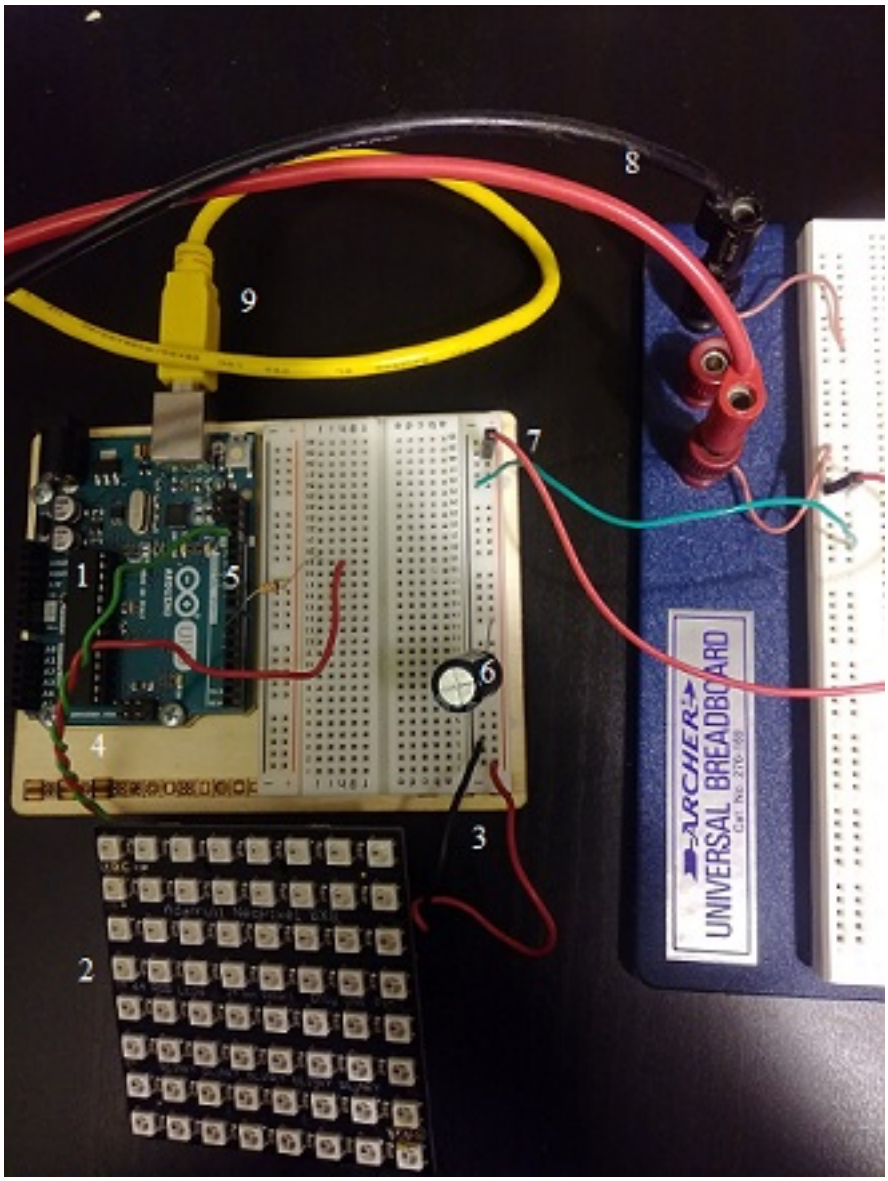

Figure 14: This shows the setup of the Arduino with the Neopixel grid. The labels are as follows 1: Arduino board 2: Neopixel grid 3: Power wires to the grid with red being the 5 volt wire and black being the ground 4: The green wire is another ground and the red wire is for data transmission 5: A 470 ohm resistor that connects the data wire to pin 6 on the Arduino 6: A 1000 microfarad capacitor connecting the power wires of the grid to jumper wires 7: Jumper wires that connect to an outside 5 volt power source 8: Power wires for the outside power source 9: Data cable that connects the Arduino to a computer via USB

the computer which could cause damage. The USB data cord is used to upload code from Arduino IDE to the Arduino board. A resistor is needed between the data wire and the Arduino board to protect the grid from any high current flow from the Arduino. The resistor connects the data wire to pin 6 on the Arduino. The Arduino has 14 data pins that are to transmit information from the board to any hardware that is being used. All Neopixel products are designed to be used on pin 6.

A serial connection was established between Python and Arduino, and Python was used to control basic elements of Arduino. However, issues arose when sending the amplitudes to Arduino to control the LEDs. Further research is still needed to fix these issues and complete this extension to Arduino.

4 Conclusion

Function approximation has many uses and overall allows for easier readability and manipulation of functions. Both the method of function approximation and gradient descent showed accurate approximations when implemented in Octave and Python. Even tough Python requires more steps to implement the methods than Octave it can still achieve the same goal.

Once the gradient descent method was implemented into Python, the next step was to incorporate the use of hardware. An Arduino can be used to control a Neopixel grid to display the result image of gradient descent. Each amplitude found by the algorithm can be manipulated and sent to Arduino through a serial connection. Even though a serial connection was established between Python and Arduino, there were still problems using the amplitudes to control individual LEDs so this extension could not be finished.

5 Appendix A

This is the octave code that approximates a function in the one dimensional case.

```
dx = 0.001;
x = [-1:dx:3];
y = abs(3*cos(x*5)+5*sin(x*3)); % target function to approximate.
ea = exp((-(x+0.6).^2)*10);
eb = exp((-(x-0.2).^2)*10);
ec = exp((-(x-0.99).^2)*14);
ed = exp((-(x-1.75).^2)*10);
ee = exp((-(x-2.55).^2)*10);

bf = [ea; eb; ec; ed; ee];

W = [];
B = [];

for i = 1:5
  for j = 1:5
    W(i,j) = sum(bf(i,:).*bf(j,:)*dx);
  endfor
endfor

for i = 1:5
   B(i,1) = sum(bf(i,:).*y.*dx);
endfor
vals = inv(W) * B;
plot(x,y, x,(vals(1,1)*ea+vals(2,1)*eb+ vals(3,1)*ec+vals(4,1)*ed+vals(5,1)*ee))
legend ({"Target Function", "Approximation"}, "location", "east");
```

This is the octave code that uses the two dimensional case to approximate an image.

```
clf;
colormap gray;
axis xy;
dx=1;
dy=1;
x = 1:dx:100;
```

```
y = 1:dy:75;
[xx,yy]=meshgrid(x,y);

m = imread('test2.jpg');
n = imresize(m,[75,100]);
gr = rgb2gray(n);
c = imcomplement(gr);

mu = [];
x1 = 30;
k1 = 1;
xnum = 9;
ynum = 10;
total = xnum*ynum;
for i = 1:xnum
    for j = 1:ynum
        mu(1,k1)= x1;
        k1 = k1+1;
    endfor
    x1 = x1+5;
endfor

k2 = 1;
for i = 1:xnum
y1 = 10;
    for j = 1:ynum
        mu(2,k2) = y1;
        k2 = k2+1;
        y1 = y1+5;
    endfor
endfor

mu2 = mu';

brightness = 10;

bf = [];
for i = 1:total
    bf(i,:) = exp(-((xx(:)-mu2(i,1)).^2)/brightness-((yy(:)-mu2(i,2)).^2)/brightness);
endfor

Tr=c(:)';
W = [];
B = [];

for i = 1:total
```

```
  for j = 1:total
    W(i,j) = sum(sum(bf(i,:).*bf(j,:)*dx)*dy);
  endfor
endfor

for i = 1:total
   B(i,1) = sum(sum(bf(i,:).*Tr.*dx)*dy);
endfor

A = inv(W) * B;

sum1 = 0;
for i = 1:total
  sum1 = sum1 + bf(i,:)*A(i,1);
endfor

sum1 = reshape(sum1, [75,100]);

subplot(1,2,1)
imagesc(x, y, c)
subplot(1,2,2)
imagesc(x, y, sum1)
colormap gray;
```

6 Appendix B

This is the python code for a one dimensional function approximation.

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

dx = 0.01
x = np.arange(-1., 3., dx)
y = 3*np.cos(5*x)+ 5*np.sin(3*x)

ea = np.exp((-(x+0.6)**2)*10)
eb = np.exp((-(x-0.2)**2)*10)
ec = np.exp((-(x-0.99)**2)*10)
ed = np.exp((-(x-1.75)**2)*10)
ee = np.exp((-(x-2.55)**2)*10)
```

```
bf = [[ea],
      [eb],
      [ec],
      [ed],
      [ee]]

bf = np.array(bf)
W = np.empty((5,5))

for i in range(5):
    for j in range(5):
        W[i][j] = np.sum((bf[i]*bf[j]*dx))

W = np.array(W)


B = np.empty((5,1))
for i in range(5):
    B[i][0] = np.sum((bf[i]*y*dx))
B = np.array(B)

inv = np.linalg.inv(W)
A = inv.dot(B);
print(A)
fig, ax = plt.subplots()
ax.plot(x, y,'b--', label='Target')
ax.plot(x, A[0][0]*ea+A[1][0]*eb+A[2][0]*ec+A[3][0]*ed+A[4][0]*ee,'r--',
label='Approximation')
ax.plot(x, y-(A[0][0]*ea+A[1][0]*eb+A[2][0]*ec+A[3][0]*ed+A[4][0]*ee),'g--',
label= 'Difference')
leg = ax.legend();
```

This is the python code used to approximate a two dimensional image.

```
import matplotlib.image as mpimg
from matplotlib import cm
from PIL import Image
import numpy as np
dx = 1
dy = 1
x = np.arange(0., 75., dx)
y = np.arange(0., 100., dy)
```

```
xx, yy = np.meshgrid(x, y)

print(xx.shape, yy.shape, (xx.flatten()).ndim)

def rgb2gray(rgb):
    return np.dot(rgb[...,:3], [0.299, 0.587, 0.144])

m = Image.open("test2.jpg")
n = m.resize((75, 100))
n = np.array(n)

gray = rgb2gray(n)
c=255-gray
n = np.array(c)
print(n.shape)
plt.imshow(c, cmap = plt.get_cmap('gray'))

plt.show()

xnum = 17
ynum = 18
total = xnum*ynum
mu = np.empty((2,total))
x1 = 20
k1 = 0
for i in range(xnum):
    for j in range(ynum):
        mu[0][k1]= x1
        k1 = k1+1
    x1 = x1+2

k2 = 0;
for i in range(xnum):
    y1 = 10;
    for j in range(ynum):
        mu[1][k2] = y1;
        k2 = k2+1;
        y1 = y1+4;

mu2 = np.transpose(mu)

brightness = 5;
bf = list()

for i in range(total):
    curr_bf = np.exp(-(xx-mu2[i][0])**2/brightness-(yy-mu2[i][1])**2/brightness)
```

```
    bf.append(curr_bf)

Tr = np.empty((7500,1))
Tr = (c)

W = np.empty((total, total))
B = np.empty((total,1))

for i in range(total):
    for j in range(total):
        W[i][j] = np.sum((np.sum((bf[i]*bf[j]*dx))*dy))

for i in range(total):
    B[i][0] = np.sum((np.sum((bf[i]*c*dx))*dy))

inv = np.linalg.inv(W)

A = inv.dot(B)

sum1 = 0;
for i in range(total):
    sum1 = sum1 + bf[i]*A[i][0]

sum1 = np.reshape(sum1, (100,75))
sum1 = np.array(sum1)

#fig, (ax1, ax2) = plt.subplots(2)
#ax1.imshow(c, cmap = plt.get_cmap('gray'))
plt.imshow( sum1, cmap=cm.Greys_r)
plt.show()
```

7 Appendix C

This is the python code that uses gradient descent to approximate a one dimensional graph.

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

dx = 0.01
x = np.arange(0, 1.5, dx)
```

```
y = 3*np.cos(5*x)+ 5*np.sin(3*x)
plt.plot(x,y,'b-')

phi_1 = np.exp((-(x-0.2)**2)*10)
phi_2 = np.exp((-(x-0.99)**2)*10)

w_1=1
w_2=1

y_hat=(w_1*phi_1)+(w_2*phi_2)

plt.plot(x,y_hat,'r-')

learn = 0.4

def summation(y_hat, x_range, y):
    total1 = 0
    total2 = 0
    total1 = np.sum((((y-y_hat)*phi_1)*0.1))
    total2 = np.sum((((y-y_hat)*phi_2)*0.1))

    return total1 / len(x_range), total2 / len(x_range)

for i in range(200):
    s1, s2 = summation(y_hat, x, y)
    w_1 = w_1 + learn * s1
    w_2 = w_2 + learn * s2
    y_hat=(w_1*phi_1)+(w_2*phi_2)
    plt.plot(x,y_hat, color='r', alpha=i/200)
plt.plot(x, y, 'b-')
```

This is the python code that uses gradient descent to approximate an image.

```
    from matplotlib import cm
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.image as mpimg
from PIL import Image
import numpy as np
from random import randint
dx = 1
dy = 1
X = np.arange(0., 75., dx)
```

```
Y = np.arange(0., 100., dy)
xx, yy = np.meshgrid(X, Y)

print(xx.shape, yy.shape, (xx.flatten()).ndim)

def rgb2gray(rgb):
    return np.dot(rgb[...,:3], [0.299, 0.587, 0.144])

m = Image.open("test2.jpg")

n = m.resize((75, 100))
n = np.array(n)

gray = rgb2gray(n)
y=255-gray

plt.imshow(y, cmap = plt.get_cmap('gray'))

plt.show()

xnum = 20
ynum = 35
total = xnum*ynum
mu = np.empty((2,total))
x1 = 20
k1 = 0
for i in range(xnum):
    for j in range(ynum):
        mu[0][k1]= x1
        k1 = k1+1
    x1 = x1+2

k2 = 0;
for i in range(xnum):
    y1 = 10;
    for j in range(ynum):
        mu[1][k2] = y1;
        k2 = k2+1;
        y1 = y1+2;

mu2 = np.transpose(mu)
print(np.shape(mu2))

brightness = 5;
bf = list()
```

```
for i in range(total):
    curr_bf = np.exp(-(xx-mu2[i][0])**2/brightness-(yy-mu2[i][1])**2/brightness)
    bf.append(curr_bf)

vals = np.empty((total,1))
y_hat = 0
for i in range(total):
    vals[i][0] = 0
    y_hat = y_hat + vals[i][0]*bf[i]

learn = 0.01
total1 = list()

for i in range(100):
    for j in range(total):
        curr_total = np.sum((((y-y_hat1)*bf[j])*0.1))
        avg = curr_total / len(xx)
        total1.append(avg)

    for k in range(total):
        vals[k][0] = vals[k][0] + learn * total1[k]
        y_hat = y_hat + vals[k][0]*bf[k]

y_hat = np.reshape(y_hat, (100,75))
y_hat = np.array(y_hat)

fig, (ax1, ax2) = plt.subplots(2)
ax1.imshow( y, cmap=cm.Greys_r)
ax2.imshow( y_hat, cmap=cm.Greys_r)
plt.show()
```

References

[1] M. Buhmann. Radial basis function. *Scholarpedia*, 5(5):9837, 2010. revision #137035.

[2] Jason Lachniet. *Introduction to GNU Octave: A brief tutorial for linear algebra and calculus students.* 2020. http://www.wcc.vccs.edu/sites/default/files/Introduction-to-GNU-Octave.pdf.

[3] Hans Petter Langtangen. *Approximation of Functions.* 2016. http://hplgit.github.io/num-methods-for-PDEs/doc/pub/approx/pdf/approx-4print-A4.pdf.

[4] Amit Saha. *Doing Math with Python.* 2015. http://index-of.es/Varios-2/Doing%20Math%20with%20Python.pdf.

[5] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. *Dive into Deep Learning.* 2020. https://d2l.ai.