1 Q1 Gradient Based Optimization

a.

The Gradient-Combination (GC) algorithm involves gradually improving the model prediction, based on the gradients of the loss function with respect to the predictions of the current model.

Given the loss function is $\mathcal{L}(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2$, we can derive the pseudo-residuals as follows:

The pseudo-residual $r_{t,i}$ is given by the negative derivative of the total loss with respect to the function $f(x_i)$ at each data point (x_i, y_i).

So, we have:

$$r_{t,i} = -\frac{\partial}{\partial f(x_i)} \sum_{j=1}^{n} \mathcal{L}(y_j, f(x_j))\Big|_{f(x_j)=f_{t-1}(x_j),\ j=1,\ldots,n}$$

Substitute the squared error loss function into this:

$$r_{t,i} = -\frac{\partial}{\partial f(x_i)} \sum_{j=1}^{n} \frac{1}{2}(y_j - f(x_j))^2\Big|_{f(x_j)=f_{t-1}(x_j),\ j=1,\ldots,n}$$

We can simplify this to:

$$r_{t,i} = -(y_i - f_{t-1}(x_i))$$

Hence, we have shown that $r_{t,i} = y_i - f_{t-1}(x_i)$

For the optimization problem in step (GC3) specific to this setting, we substitute $r_{t,i} = y_i - f_{t-1}(x_i)$ into the expression for h_t:

$$h_t = \arg\min_{f \in F} \sum_{i=1}^{n} \mathcal{L}(r_{t,i}, f(x_i))$$

which results in:

$$h_t = \arg\min_{f \in F} \sum_{i=1}^{n} (y_i - f_{t-1}(x_i) - f(x_i))^2$$

This is the expression for the optimization problem in step (GC3) specific to the regression setting with the squared error loss function.

b.

The adaptive step-size approach (SS2) is designed to find the optimal step size that minimizes the loss function at each iteration.

The step-size (α_t) is given by the following expression:

$$\alpha_t = \arg\min_{\alpha} \sum_{i=1}^{n} \mathcal{L}(y_i, f_{t-1}(x_i) + \alpha h_t(x_i))$$

Substitute the squared error loss function into this expression, we get:

$$\alpha_t = \arg\min_{\alpha} \sum_{i=1}^{n} \frac{1}{2}(y_i - (f_{t-1}(x_i) + \alpha h_t(x_i)))^2$$

To find the optimal α_t, we take the derivative of this function with respect to α and set it equal to zero:

$$0 = \frac{\partial}{\partial \alpha} \sum_{i=1}^{n} (y_i - f_{t-1}(x_i) - \alpha h_t(x_i))^2$$

Solving this, we get:

$$0 = \sum_{i=1}^{n} -2(y_i - f_{t-1}(x_i) - \alpha h_t(x_i)) h_t(x_i)$$

Since we cannot guarantee that $h_t(x_i)$ is never zero, we cannot cancel out the $h_t(x_i)$ term. Therefore, we rearrange to solve for α:

$$\alpha_t = \frac{\sum_{i=1}^{n} (y_i - f_{t-1}(x_i)) h_t(x_i)}{\sum_{i=1}^{n} h_t^2(x_i)}$$

This is the expression for the adaptive step-size (SS2) specific to the regression setting with the squared error loss function.

c.

The adaptive step size:

step-size to be α = 0.1:

Listing 1: Q1_c.py

```
# true function
def f(x):
    t1 = np.sqrt(x * (1-x))
    t2 = (2.1 * np.pi) / (x + 0.05)
    t3 = np.sin(t2)
    return t1*t3
# def f_sampler(f, n=100, sigma=0.05):
def f_sampler(f, n=100, sigma=0.05, alpha=0.1): #  alpha
    # sample points from function f with Gaussian noise (0,sigma**2)
    xvals = np.random.uniform(low=0, high=1, size=n)
    # yvals = f(xvals) + sigma * np.random.normal(0,1,size=n)
    yvals = f(xvals) + sigma * np.random.normal(0,1,size=n) + alpha #  alpha
    return xvals, yvals
np.random.seed(123)
# X, y = f_sampler(f, 160, sigma=0.2)
X, y = f_sampler(f, 160, sigma=0.2, alpha=0.1) #  alpha
```

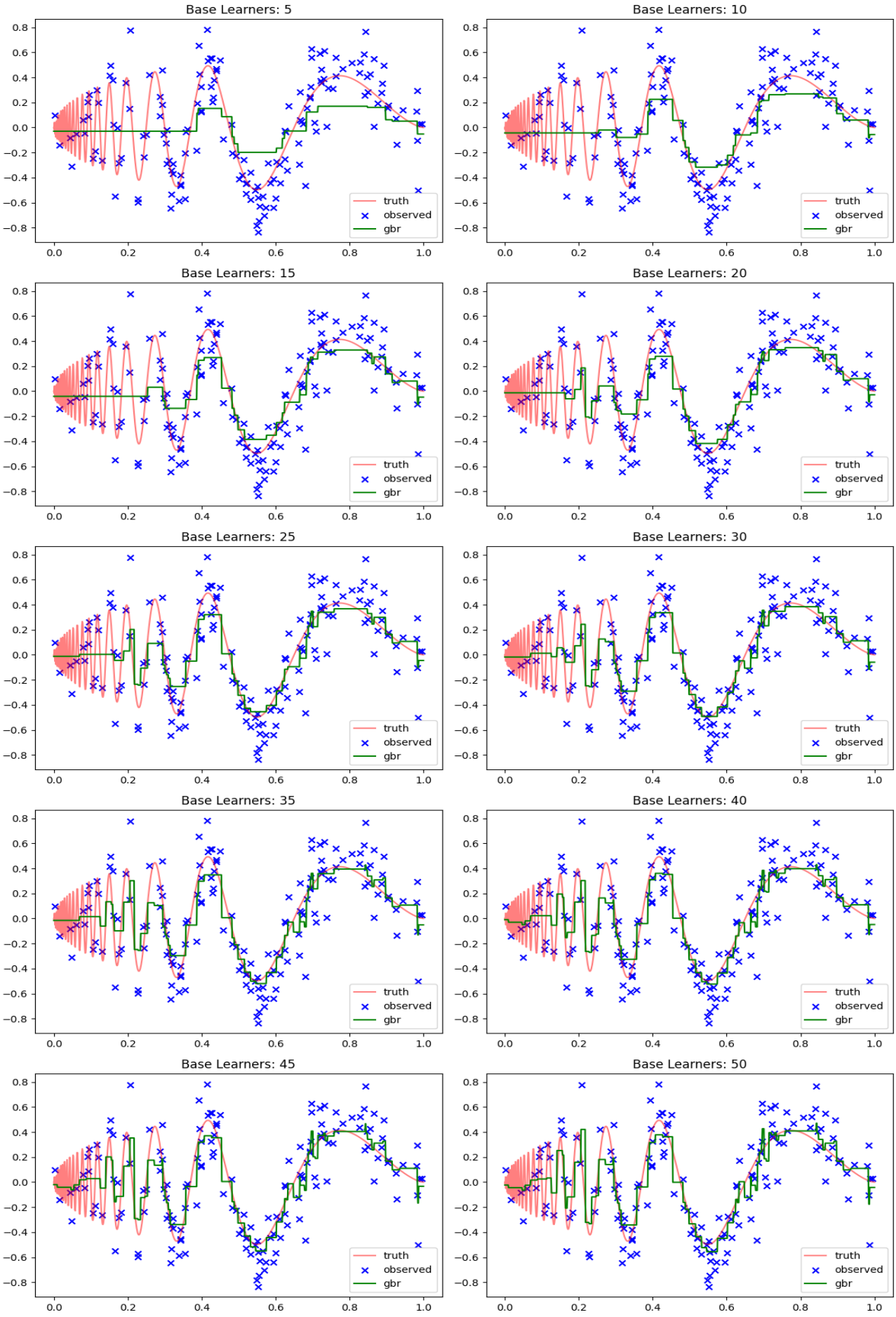


Figure 1: q1AdaptiveStepSize

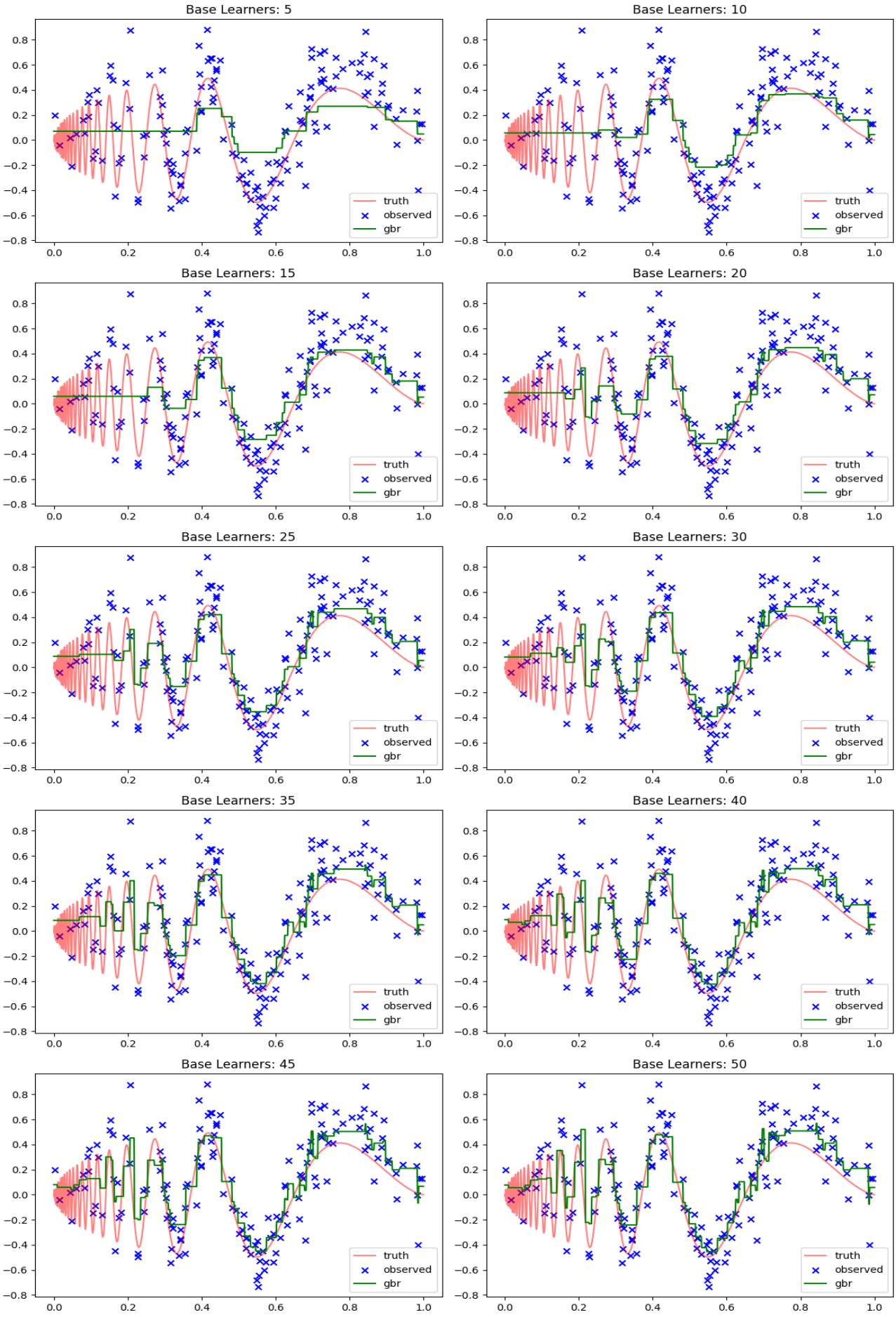


Figure 2: q1Alpha=0.1

```
X = X.reshape(-1,1)
fig, axes = plt.subplots(5, 2, figsize=(12, 20))
base_learners = [5, 10, 15, 20, 25, 30, 35, 40, 45, 50]
for i, ax in enumerate(axes.flatten()):
    n_learners = base_learners[i]
    gbr = GradientBoostingRegressor(n_estimators=n_learners).fit(X,y)
    xx = np.linspace(0,1,1000)
    ax.plot(xx, f(xx), alpha=0.5, color='red', label='truth')
    ax.scatter(X,y, marker='x', color='blue', label='observed')
    ax.plot(xx, gbr.predict(xx.reshape(-1,1)), color='green', label='gbr')
    ax.set_title(f"Base Learners: {n_learners}")
    ax.legend()
plt.tight_layout()
# plt.savefig('q1_alpha_0.1.png')
plt.show()
```

Commenting on the results:

- As the number of base learners increases, the gradient-combination model becomes more flexible and better approximates the true function. The predictions converge to the true function with more base learners.
- The adaptive step-size implementation adjusts the step size at each iteration based on the data, leading to better convergence and potentially faster learning. In contrast, the fixed step-size implementation uses a constant step size throughout, which may result in slower convergence or overshooting of the optimal solution.
- The performance of the model on different x-ranges of the data depends on the nature of the true function and the distribution of the data. If the true function varies significantly across different x-ranges, the model may struggle to accurately predict in those regions. However, with a sufficient number of base learners, the model can capture complex patterns and improve prediction accuracy in all regions of the data.

d.

maxdepth=2,The adaptive step size:
maxdepth=2,The adaptive step size:

Listing 2: Q1_d.py

```
for i, ax in enumerate(axes.flatten()):
    n_learners = base_learners[i]
    gbr = GradientBoostingRegressor(n_estimators=n_learners, max_depth=2).fit(X,y)
    xx = np.linspace(0,1,1000)
    ax.plot(xx, f(xx), alpha=0.5, color='red', label='truth')
    ax.scatter(X,y, marker='x', color='blue', label='observed')
```

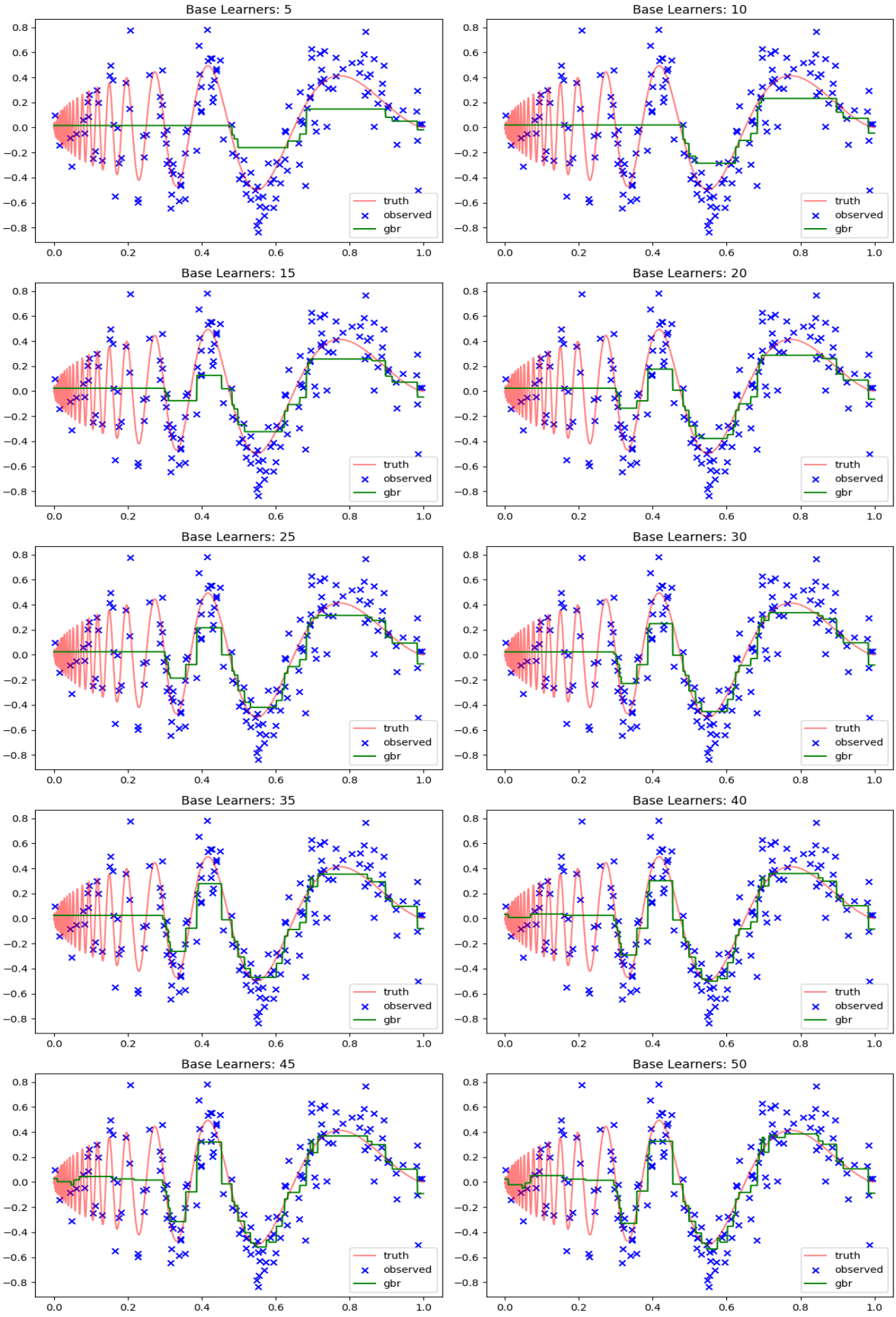


Figure 3: maxdepth=2,AdaptiveStepSize

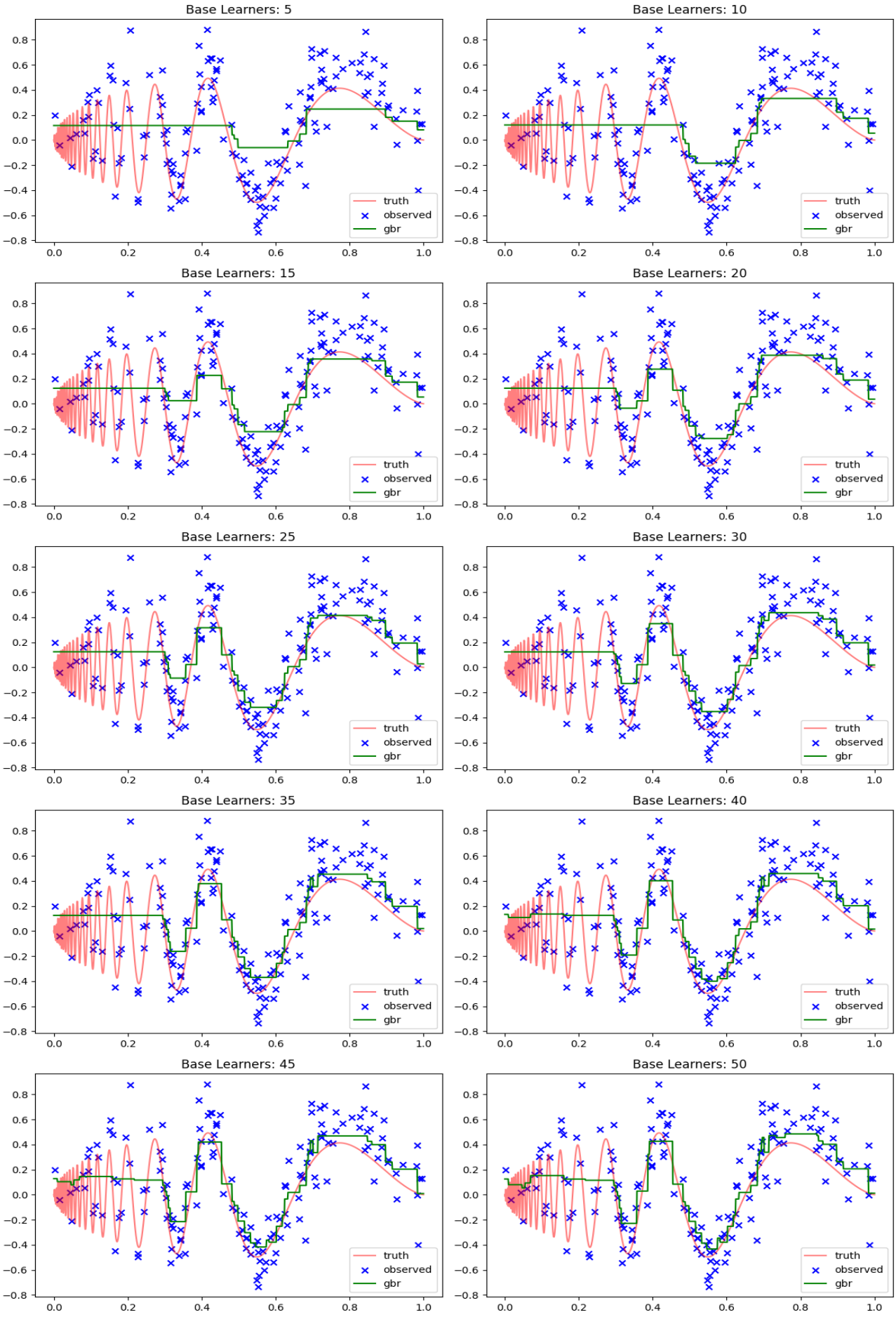


Figure 4: maxdepth=2,alpha=0.1

```
    ax.plot(xx, gbr.predict(xx.reshape(-1,1)), color='green', label='gbr')
    ax.set_title(f"Base Learners: {n_learners}")
    ax.legend()
```

Using depth 2 decision trees as base learners will allow the model to capture more complex patterns in the data compared to using depth 1 decision trees (decision stumps). This is because depth 2 decision trees can model interactions between features, whereas decision stumps cannot.

For the adaptive case, where the step size is adjusted at each iteration based on the data:

The model will likely converge faster towards the optimal solution as it can take larger steps when far away from the optimum and smaller steps when it is close. This can lead to better performance, especially when the number of base learners is high, as the model can effectively adjust the influence of each learner to minimize the loss function. For the non-adaptive case, where the step size is fixed:

The model might take longer to converge or might not converge to the optimal solution at all, especially if the step size is not appropriately chosen. The performance of the model could be worse compared to the adaptive case, especially for larger numbers of base learners, as each learner has the same influence regardless of its effectiveness at reducing the loss. These are general observations and the actual results can vary depending on the specific dataset and true function. Therefore, it is always recommended to experiment with different settings and validate the performance of the model using a held-out validation set or cross-validation.

e.

Let's consider the classification setting where y is an element of -1, 1. The classification loss function is defined as:

$$L(y, \hat{y}) = \log(1 + e^{-y\hat{y}})$$

For round t of the algorithm, the expression for $r_{t,i}$, the pseudo-residual for the i-th data point at round t, can be calculated as:

$$r_{t,i} = -\frac{\partial L(y_i, f_t(x_i))}{\partial f_t(x_i)}$$

Since we are using gradient boosting, the base learner $f_t(x_i)$ at round t is the negative gradient of the loss function with respect to the predicted value at round $t-1$:

$$f_t(x_i) = -\frac{\partial L(y_i, \hat{y}_{t-1}(x_i))}{\partial \hat{y}_{t-1}(x_i)}$$

Substituting the expression for $f_t(x_i)$ into the expression for $r_{t,i}$, we have:

$$r_{t,i} = -\frac{\partial L(y_i, -\frac{\partial L(y_i, \hat{y}_{t-1}(x_i))}{\partial \hat{y}_{t-1}(x_i)})}{\partial \hat{y}_{t-1}(x_i)}$$

Simplifying this expression, we obtain:

$$r_{t,i} = \frac{\partial^2 L(y_i, \hat{y}_{t-1}(x_i))}{(\partial \hat{y}_{t-1}(x_i))^2}$$

This is the expression for $r_{t,i}$ in the classification setting.

Now, let's write down the expression for the optimization problem in step (GC3) specific to this setting. In step (GC3), we need to find the base learner h_t that minimizes the loss function $L(y, f_{t-1}(x)+ h_t(x))$. In the classification setting, this can be expressed as:

$$h_t = \text{argmin}(h) \sum_i L(y_i, f_{t-1}(x_i) + h(x_i))$$

We need to find the base learner h_t that minimizes the sum of the classification loss over all data points, given the current model $f_{t-1}(x)$. This optimization problem can be solved using various algorithms, such as gradient descent or decision tree algorithms.

f.

We have the loss function $L(y, \hat{y}) = \log(1 + e^{-y\hat{y}})$ for classification, where y is the true label and $\hat{y}$ is the predicted value.

In step (SS2), the weight α_t is determined by minimizing the weighted loss function with respect to α:

$$\alpha_t = \arg\min_{\alpha} \sum_i w_i \cdot L(y_i, f_{t-1}(x_i) + \alpha \cdot h_t(x_i))$$

where w_i represents the weight of the i-th data point and $f_{t-1}(x_i)$ is the predicted value at round $t-1$.

Expanding the weighted loss function, we have:

$$\sum_i w_i \cdot L(y_i, f_{t-1}(x_i) + \alpha \cdot h_t(x_i)) = \sum_i w_i \cdot \log(1 + e^{-y_i(f_{t-1}(x_i)+\alpha \cdot h_t(x_i))})$$

To find the value of α_t that minimizes this expression, we can take the derivative with respect to α and set it to zero:

$$\frac{\partial}{\partial \alpha} \left(\sum_i w_i \cdot \log(1 + e^{-y_i(f_{t-1}(x_i)+\alpha \cdot h_t(x_i))}) \right) = 0$$

Simplifying the expression, we have:

$$\sum_i w_i \cdot y_i \cdot h_t(x_i) \cdot \frac{e^{-y_i(f_{t-1}(x_i)+\alpha \cdot h_t(x_i))}}{1 + e^{-y_i(f_{t-1}(x_i)+\alpha \cdot h_t(x_i))}} = 0$$

Unfortunately, this expression does not have a closed-form solution. It is not possible to solve for α_t analytically. Instead, iterative optimization techniques, such as gradient descent or Newton's method, are typically used to find an approximate solution for α_t.

These iterative optimization techniques involve updating the value of α_t in each iteration until

convergence is reached, based on the gradient or Hessian of the loss function with respect to α.

g.

When it is not possible to solve for a_t exactly, an iterative optimization algorithm can be used to approximate its value. One common approach is to use a gradient descent algorithm.

Here's how you might implement the algorithm:

1. Initialize a_t to an initial value, such as $a_t = 0$.
2. Calculate the gradient of the loss function with respect to a at the current a_t value.
3. Update a_t using the gradient descent update rule: $a_t = a_t - \text{learning\_rate} \cdot \text{gradient}$.
4. Repeat steps 2 and 3 until convergence is reached, typically by monitoring the change in a_t between iterations or the value of the loss function.

Additional considerations for implementing the algorithm:

- The learning rate determines the step size in each iteration. It needs to be chosen carefully to balance convergence speed and stability.
- Techniques such as line search or adaptive learning rates (e.g., using AdaGrad or Adam) can be used to adaptively adjust the learning rate during optimization.
- Regularization techniques, such as L1 or L2 regularization, can be applied to prevent overfitting or improve generalization performance.
- Depending on the size of the dataset and complexity of the model, it may be necessary to use mini-batch gradient descent or stochastic gradient descent to efficiently compute the gradients and update a_t.
- Convergence criteria, such as a maximum number of iterations or a desired threshold for the change in a_t, should be defined to terminate the optimization process.

The additional computational costs of using this iterative optimization approach compared to using a constant step size are primarily due to the repeated computation of gradients and the updates to a_t in each iteration. Overall, the iterative optimization approach provides a practical solution when it is not possible to find a_t exactly. It allows us to approximate the optimal a_t value and improve the performance of the algorithm.

2 Question 2. Test Set Linear Regression

a.

To construct a hypothesis g from the elements of H such that rMSE(g) = 0, we can use a naive, brute-force algorithm that iterates through all possible combinations of hypotheses in H and checks if the combined predictions match the true labels for all data points in the test set. Here is a description of the algorithm:

1. Initialize an empty list of hypotheses, g_list.
2. For each hypothesis h in H:
   a. Add h to g_list.
   b. Query rMSE(g_list) to get the current rMSE value.
   c. If rMSE(g_list) = 0, terminate the algorithm as we have found the desired hypothesis g.
   d. Otherwise, continue to the next hypothesis.
3. If none of the combinations of hypotheses in H yield an rMSE value of 0, return that it is not possible to construct a hypothesis g with rMSE(g) = 0.

The number of queries required for this algorithm is T, the total number of hypotheses in H. This is because we query rMSE(g_list) for each combination of hypotheses that we construct.

In the worst case, this algorithm has a time complexity of O(T * n), where T is the number of hypotheses and n is the size of the test set. This is because we iterate through all possible combinations of hypotheses and for each combination, we query rMSE(g_list) which requires computing the predictions for all data points in the test set.

Note: This brute-force algorithm may not be efficient for large values of T or n. If the size of H or the test set becomes prohibitively large, more efficient algorithms, such as gradient-based optimization or model averaging techniques, can be used to find a hypothesis g with rMSE(g) = 0.

b.

To compute the inner product $y > h(X)$, where $y = (y1, y2, ..., yn)$ is the true label vector and $h(X) = (h(x1), h(x2), ..., h(xn))$ is the predicted label vector, we need to find the smallest number of queries required.

Since we can query rMSE(h) for any hypothesis h, we can use this to indirectly estimate the inner product $y > h(X)$. Here is an approach to compute $y > h(X)$ with the fewest number of queries:

1. Initialize the inner product result, $y > h(X)$, to 0.
2. For each data point xi in the test set:
   a. Query rMSE(h_i), where h_i is a hypothesis that returns yi for $xi (i.e., h_i(xi) = yi)$.
   b. Update $y > h(X)$ by adding the squared value of rMSE(h_i) to the current result: $y > h(X) = y > h(X) + (rMSE(h_i))^2$.
3. The final value of $y > h(X)$ is the sum of squared rMSE values for all hypotheses h_i that correctly predict the true labels.

The number of queries required for this approach is equal to the number of data points in the test set, n. This is because we query rMSE(h_i) for each data point, and each query provides information

about the squared difference between the predicted and true labels for that specific point.

In summary, the smallest number of queries required to compute $y > h(X)$ is n, the number of data points in the test set. This approach allows us to estimate the inner product without explicitly knowing the true labels, by leveraging the rMSE values obtained from querying the model's predictions.

c.

To solve the optimization problem and obtain the optimal weights $\alpha_1, \alpha_2, ..., \alpha_K$, we can leverage the previous result of computing $y > h(X)$ using rMSE queries. Here is an approach to find the optimal weights:

1. Initialize a vector of weights $\alpha = [\alpha_1, \alpha_2, ..., \alpha_K]$ to some initial values, such as $\alpha = [1/K, 1/K, ..., 1/K]$ (equal weights for each hypothesis).
2. For each hypothesis h_i in H:
   (a) Calculate the predictions for the test set using h_i: $h_i(X) = (h_i(x_1), h_i(x_2), ..., h_i(x_n))$.
   (b) Compute the inner product between the true label vector y and the prediction vector $h_i(X)$: $y > h_i(X)$.
   (c) Store the value of $y > h_i(X)$ in a list or array, denoted as y_i.
3. Calculate the normalization factor Z as the square root of the sum of squared y_i values: $Z = \sqrt{y_1^2 + y_2^2 + ... + y_K^2}$.
4. Update the weights α_i by dividing each y_i by Z: $\alpha_i = \frac{y_i}{Z}$.
5. The updated weights $\alpha_1, \alpha_2, ..., \alpha_K$ are the optimal weights for the given set of hypotheses H.

The number of queries needed for this approach is K, which is the number of hypotheses in H. For each hypothesis, we need to query rMSE to compute $y > h_i(X)$.

The exact values of the weights $\alpha_1, \alpha_2, ..., \alpha_K$ are given by the normalized y_i values divided by the normalization factor Z. In terms of X, y, and the elements of H:

$$\alpha_i = \frac{y_i}{Z} = \frac{(\text{rMSE}(h_i))^2}{\sqrt{\sum_{k=1}^{K} (\text{rMSE}(h_k))^2}}$$

These weights represent the optimal blending of the hypotheses in H, taking into account their individual rMSE values and their predictions on the test set.

In summary, this approach requires K queries of rMSE to compute the optimal weights $\alpha_1, \alpha_2, ..., \alpha_K$. The values of α_i are obtained by normalizing the y_i values, which are computed by taking the inner product between the true label vector y and the prediction vector $h_i(X)$. The exact values of α_i are given by the ratio of the squared rMSE values to the square root of the sum of squared rMSE values for all hypotheses in H.

3 Question 3. Newton's Method

a.

The formula (3) for Newton's method in the case of a function $f : \mathbb{R}^n \to \mathbb{R}$ with vector inputs is a generalization of equation (1) to functions with multiple variables.

In equation (1), we have a single variable function $g(x)$, where x is a scalar. The update rule in equation (1) uses the first derivative $g'(x)$ and the second derivative $g''(x)$ to iteratively update the value of x.

In equation (3), we have a multivariable function $f(x)$, where x is a vector in $\mathbb{R}^n$. The update rule in equation (3) uses the gradient vector $\nabla f(x)$, which is the vector of partial derivatives of f with respect to each component of x, and the Hessian matrix $H(x)$, which is the matrix of second partial derivatives of f with respect to each pair of components of x.

The update rule in equation (3) can be seen as a generalization of equation (1) because it uses the gradient vector $\nabla f(x)$ to guide the search direction and the Hessian matrix $H(x)$ to determine the step size and direction in each iteration. By taking into account the first and second derivatives of the function with respect to each component of the input vector, Newton's method in equation (3) can effectively navigate the multidimensional space to find the minimum of the function.

b.

Listing 3: Q3_b.py

```
def f(x, y):
    return 100 * (y - x**2)**2 + (1 - x)**2
# Create a meshgrid of x and y values
x = np.linspace(-2, 2, 100)
y = np.linspace(-1, 3, 100)
X, Y = np.meshgrid(x, y)
# Calculate the corresponding z values using the function
Z = f(X, Y)
# Create a 3D plot
fig = plt.figure()
ax = plt.axes(projection='3d')
ax.plot_surface(X, Y, Z, cmap='viridis')
# Set labels and title
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('f(x, y)')
ax.set_title('3D Plot of f(x, y)')
# Display the plot
plt.show()
```

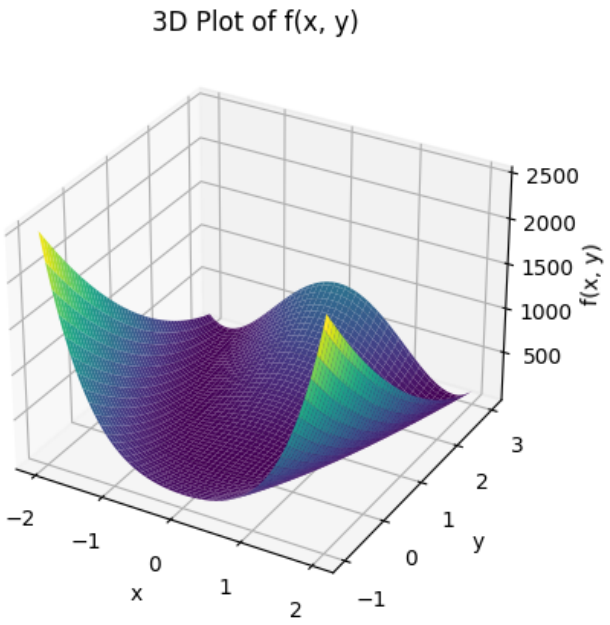


Figure 5: q3_b

Listing 4: Q3_b.py

```python
# Define the symbols
x, y = sp.symbols('x y')
# Define the function
f = 100 * (y - x**2)**2 + (1 - x)**2
# Calculate the gradient
grad = [sp.diff(f, var) for var in (x, y)]
print("Gradient: ", grad)
# Calculate the Hessian
hessian = [[sp.diff(var2, var1) for var1 in (x, y)] for var2 in (x, y)]
print("Hessian: ", hessian)
```

```
Gradient:  [-400*x*(-x**2 + y) + 2*x - 2, -200*x**2 + 200*y]
Hessian:  [[1, 0], [0, 1]]
```

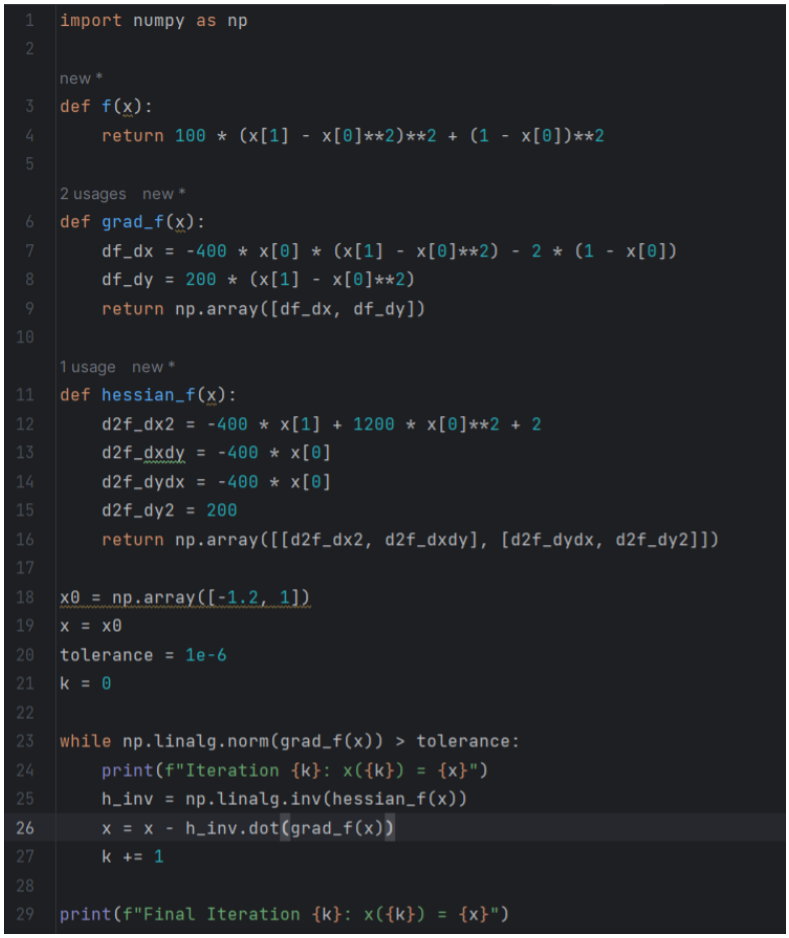
Figure 6: q3_b_Gradient_Hessian

c.

Listing 5: Q3_c.py

```python
def f(x):
    return 100 * (x[1] - x[0]**2)**2 + (1 - x[0])**2
```

```python
def grad_f(x):
    df_dx = -400 * x[0] * (x[1] - x[0]**2) - 2 * (1 - x[0])
    df_dy = 200 * (x[1] - x[0]**2)
    return np.array([df_dx, df_dy])
def hessian_f(x):
    d2f_dx2 = -400 * x[1] + 1200 * x[0]**2 + 2
    d2f_dxdy = -400 * x[0]
    d2f_dydx = -400 * x[0]
    d2f_dy2 = 200
    return np.array([[d2f_dx2, d2f_dxdy], [d2f_dydx, d2f_dy2]])
x0 = np.array([-1.2, 1])
x = x0
tolerance = 1e-6
k = 0
while np.linalg.norm(grad_f(x)) > tolerance:
    print(f"Iteration {k}: x({k}) = {x}")
    h_inv = np.linalg.inv(hessian_f(x))
    x = x - h_inv.dot(grad_f(x))
    k += 1
print(f"Final Iteration {k}: x({k}) = {x}")
```



```python
import numpy as np

new *
def f(x):
    return 100 * (x[1] - x[0]**2)**2 + (1 - x[0])**2

2 usages  new *
def grad_f(x):
    df_dx = -400 * x[0] * (x[1] - x[0]**2) - 2 * (1 - x[0])
    df_dy = 200 * (x[1] - x[0]**2)
    return np.array([df_dx, df_dy])

1 usage  new *
def hessian_f(x):
    d2f_dx2 = -400 * x[1] + 1200 * x[0]**2 + 2
    d2f_dxdy = -400 * x[0]
    d2f_dydx = -400 * x[0]
    d2f_dy2 = 200
    return np.array([[d2f_dx2, d2f_dxdy], [d2f_dydx, d2f_dy2]])

x0 = np.array([-1.2, 1])
x = x0
tolerance = 1e-6
k = 0

while np.linalg.norm(grad_f(x)) > tolerance:
    print(f"Iteration {k}: x({k}) = {x}")
    h_inv = np.linalg.inv(hessian_f(x))
    x = x - h_inv.dot(grad_f(x))
    k += 1

print(f"Final Iteration {k}: x({k}) = {x}")
```

Figure 7: q3_code

Iteration 0: x(0) = [-1.2 1.]
Iteration 1: x(1) = [-1.1752809 1.38067416]
Iteration 2: x(2) = [0.76311487 -3.17503385]
Iteration 3: x(3) = [0.76342968 0.58282478]
Iteration 4: x(4) = [0.99999531 0.94402732]
Iteration 5: x(5) = [0.9999957 0.99999139]
Final Iteration 6: x(6) = [1. 1.]

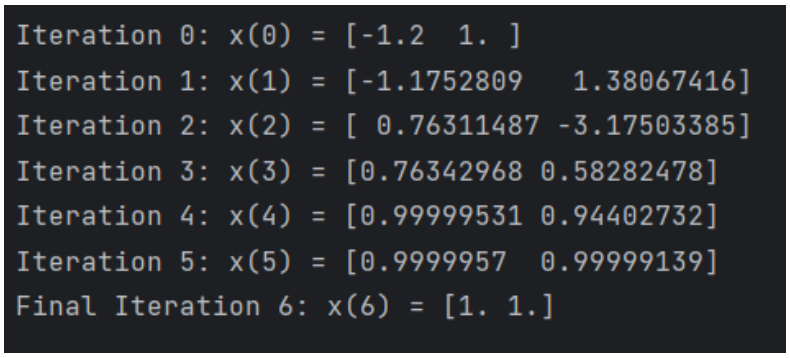


Figure 8: q3_Iteration