APPM 5360, Spring 2023 - Written Homework 6

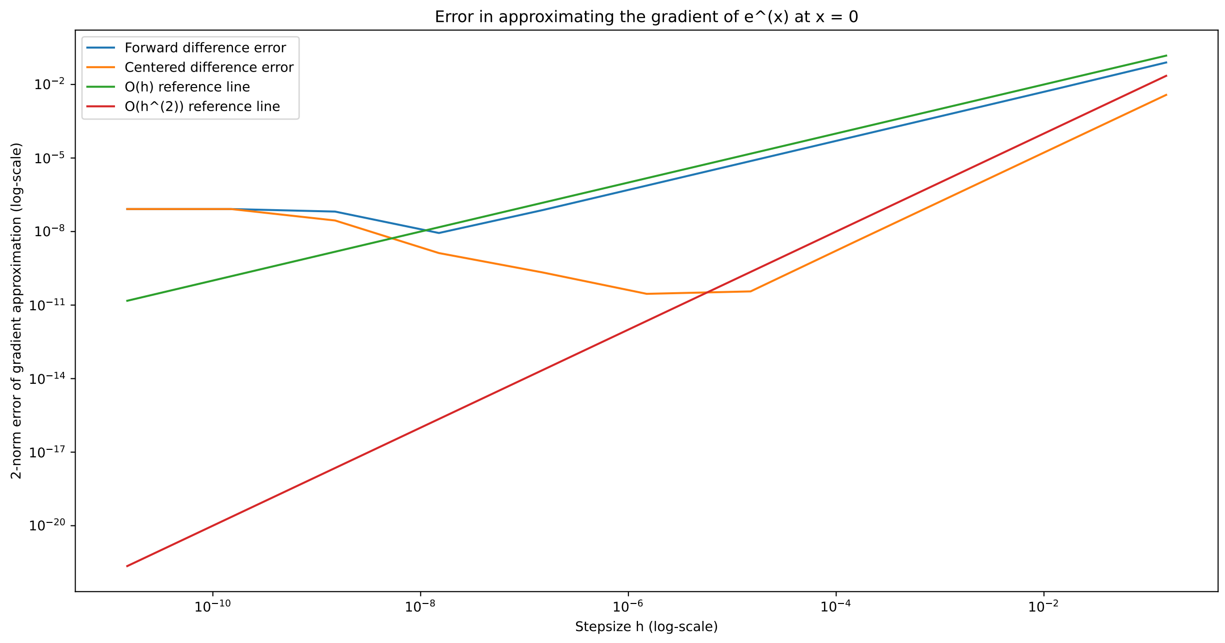
Eappen Nelluvelil; Collaborators: Jack, Bisman, Logan, Tyler

March 3, 2023

1. Problem 1, part (b)

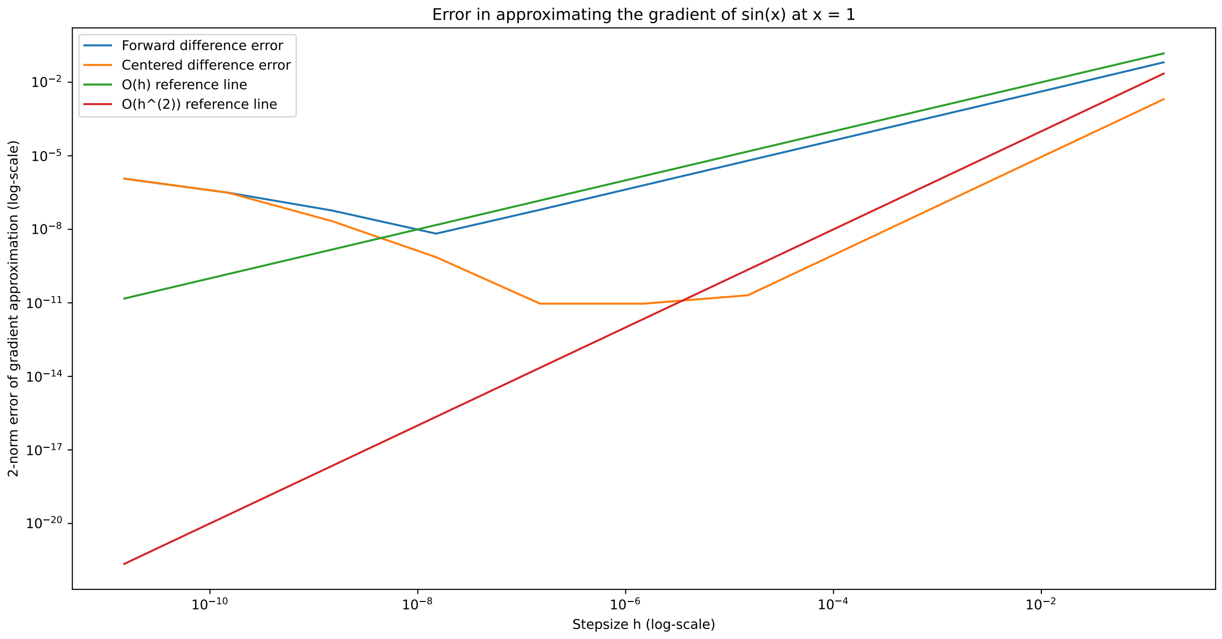
   I ran the `gradientCheck` function on the following pairs of functions and known gradients, and attached the corresponding gradient approximation error plots:

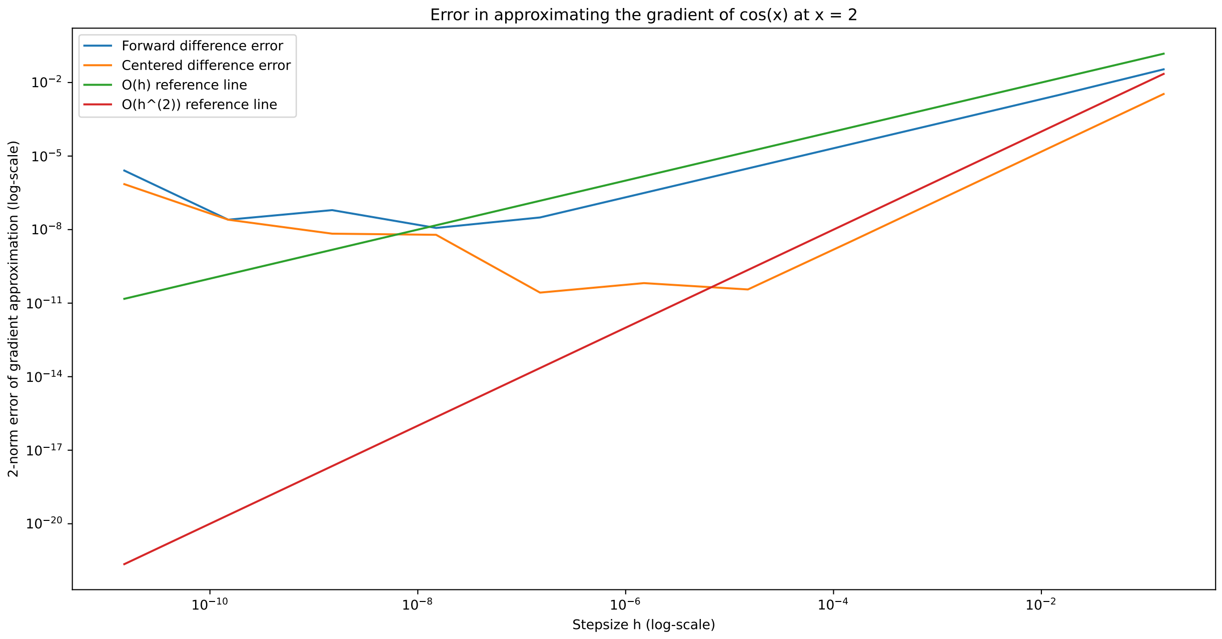
   (a) $f(x) = e^x$, $f'(x) = e^x$, $x_0 = 0$



   (b) $g(x) = \sin(x)$, $g'(x) = \cos(x)$, $x_0 = 1$

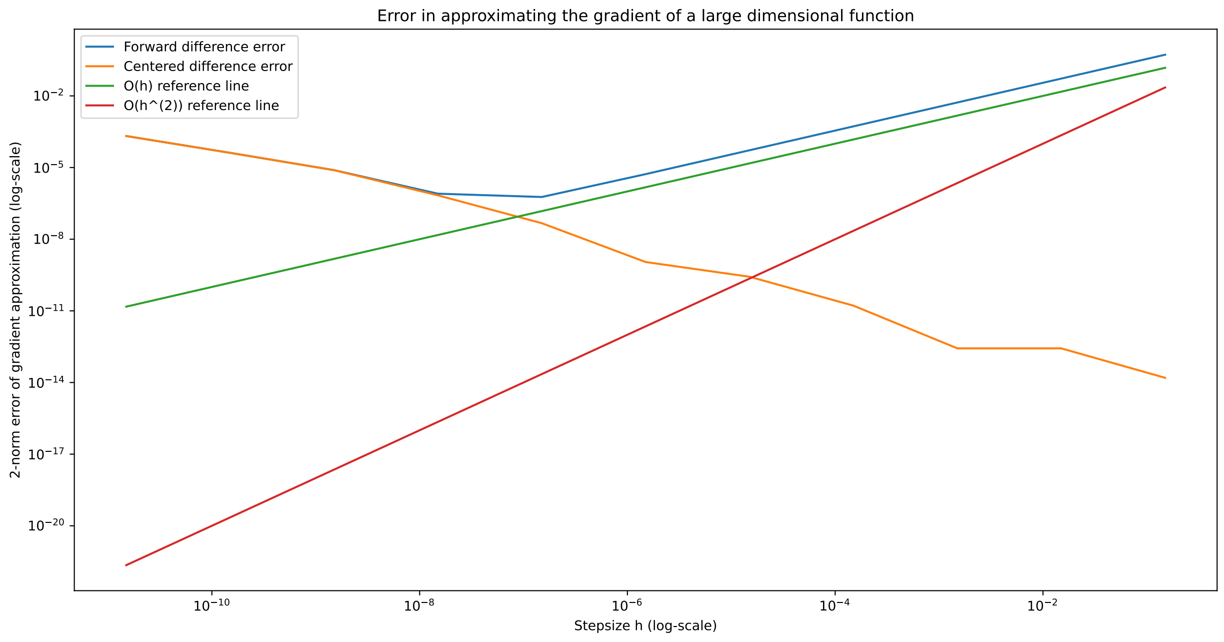
   (c) $h(x) = \cos(x)$, $h'(x) = -\sin(x)$, $x_0 = 2$


Error in approximating the gradient of sin(x) at x = 1
Forward difference error
Centered difference error
O(h) reference line
O(h^(2)) reference line
2-norm error of gradient approximation (log-scale)
Stepsize h (log-scale)
10^{-2}
10^{-5}
10^{-8}
10^{-11}
10^{-14}
10^{-17}
10^{-20}
10^{-10}
10^{-8}
10^{-6}
10^{-4}
10^{-2}


Error in approximating the gradient of cos(x) at x = 2
Forward difference error
Centered difference error
O(h) reference line
O(h^(2)) reference line
2-norm error of gradient approximation (log-scale)
Stepsize h (log-scale)
10^{-2}
10^{-5}
10^{-8}
10^{-11}
10^{-14}
10^{-17}
10^{-20}
10^{-10}
10^{-8}
10^{-6}
10^{-4}
10^{-2}

In all three cases, the forward and centered finite difference approximations to the gradients perform as expected, i.e., the forward difference approximation errors are behave like $\mathcal{O}\left(h\right)$ and the centered difference approximation errors behave like $\mathcal{O}\left(h^2\right)$ We see that for stepsizes h smaller than 10^{-5}, numerical roundoff issues cause the centered difference approximations to diverge from the true gradient, which is to be expected.

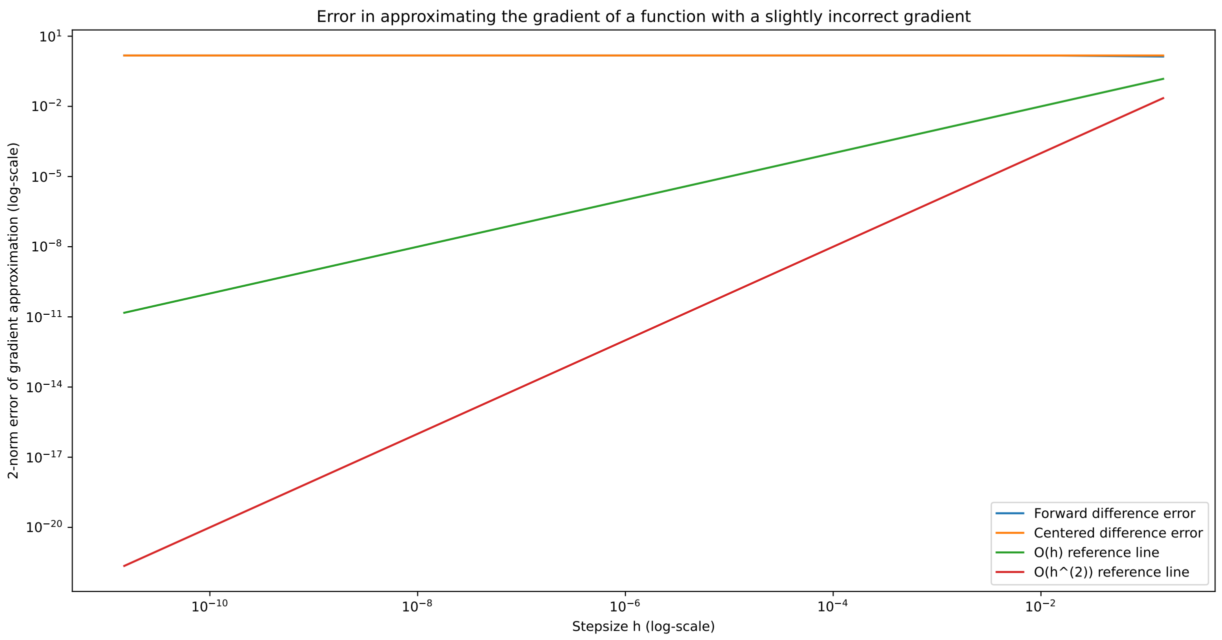
I also tested the `gradientCheck` function on a large dimensional quadratic, $k\left(x\right) = \frac{1}{2}x^T x$, where $x \in \mathbb{R}^{50}$, and attached the plot below:



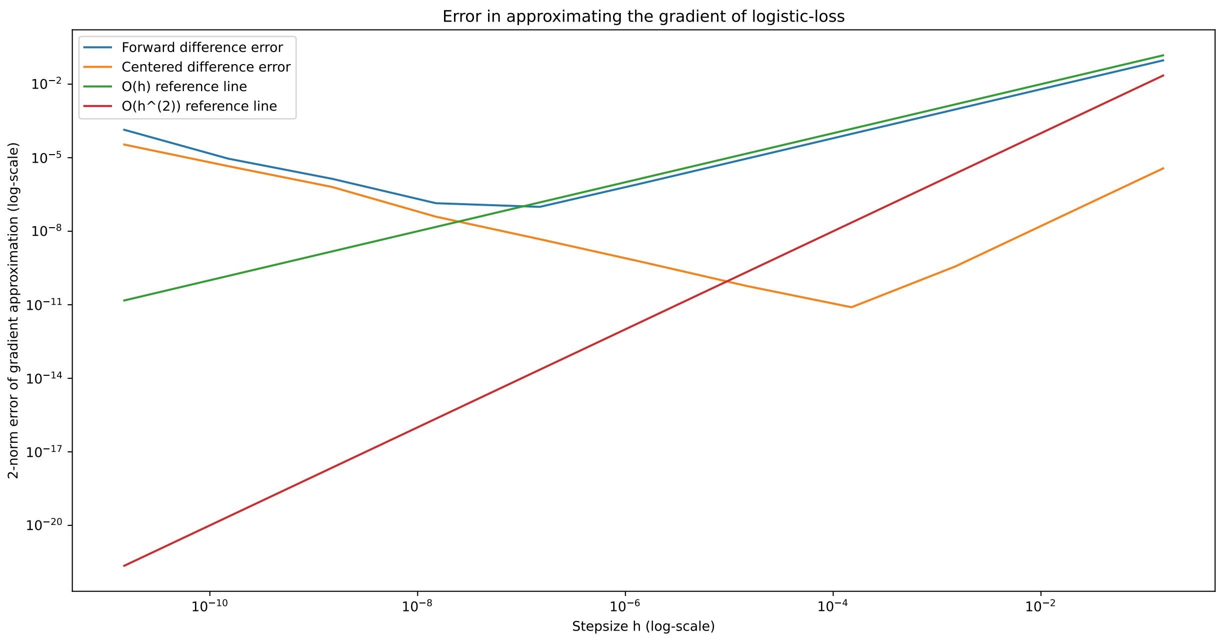
The forward difference approximations behave as expected, and the centered difference approximations are very good approximations to the true gradient for large step sizes h, but numerical roundoff issues cause the errors to immediately increase in size as h gets smaller.

To check if `gradientCheck` function can reliably distinguish correct gradients, I used the function $j\left(x\right) = x^2$ and passed in the slightly incorrect gradient of $j'\left(x\right) = \frac{5}{2}x$.

The `gradientCheck` function, in this case, can reliably distinguish correct gradients as we can see that the errors in the approximation stay constant and do not decrease as h gets smaller.



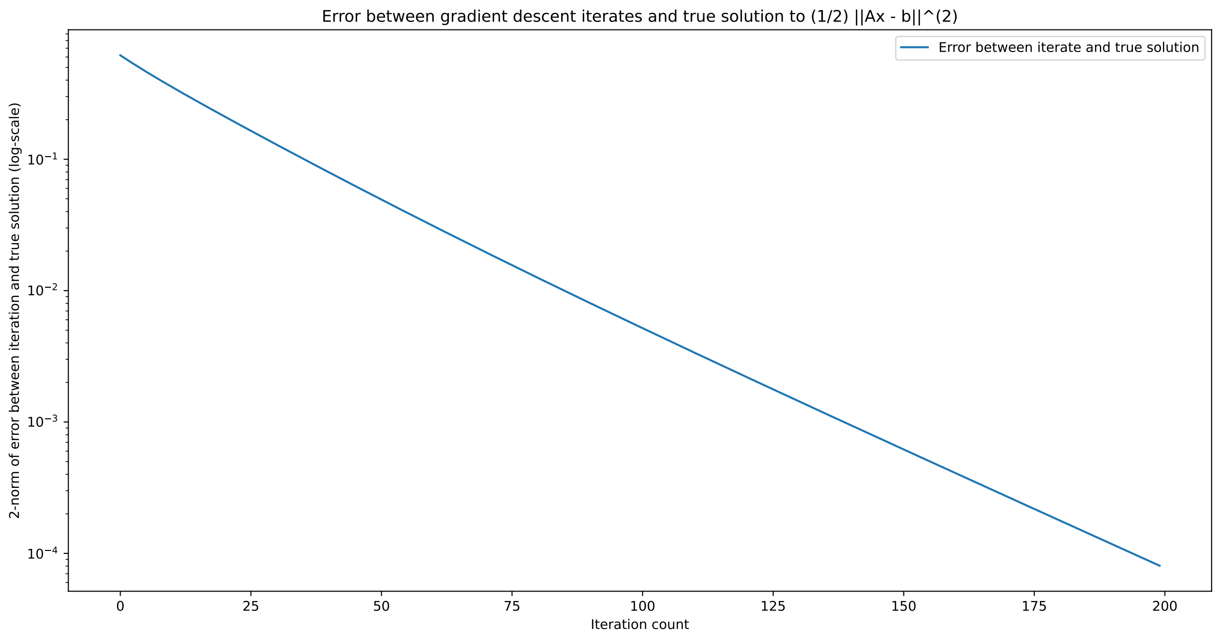
I then ran the `gradientCheck` function on the logistic-loss/gradient pair using spam data, i.e., $n = 10$ and $p = 5$, and attached the resulting error plot below.



From the above plot, we see the forward and centered difference approximations behave as expected, which indicates that I correctly implemented the gradient of the logistic-loss function.

2. Problem 2

   I implemented a function `gradientDescent`, which implements the gradient descent method and takes in inputs the function, its gradient, a max iteration count, stepsize, and stopping tolerance (if successive iterates are close enough), along with other parameters to label the plots. I then applied the function to the quadratic $f(x) = \frac{1}{2}||Ax - b||^2$, where $A \in \mathbb{R}^{10\times5}$ and $b \in \mathbb{R}^{10}$ have entries drawn from a uniform distribution. I chose the stepsize to be $t = \frac{1}{||A||^2}$. I also verified that the matrix A has full rank, and attached a plot of the error between each iterate and the true solution to the minimizer of the f below.



The y-axis is on a log-scale, and we see that my gradient descent solver performs as expected. The relevant code is given below:

Listing 1: `gradientDescent` implementation

```
def gradientDescent(f, grad_f, x0, x_star, title,
max_iters=100, stepsize=1, tol=1e-7, compute_error=False):
num_iters    = 0
errors       = []
xn           = np.zeros(x0.size)

while (num_iters < max_iters):
    num_iters += 1
    xn = x0 - stepsize*grad_f(x0)

    # Compute the error between the new iterate and true solution,
    # if provided
    if compute_error:
        errors.append(np.linalg.norm(xn - x_star, 2))

    if (np.linalg.norm(xn - x0, 2) <= tol):
        break

    x0 = xn

errors = np.array(errors)

# Make a plot of error between each iterate and the true solution
# as a function of the iteration count
if compute_error:
    plt.figure()
    plt.plot(np.arange(num_iters), errors,
    label="Error between iterate and true solution")
    # plt.xscale("log", base=10)
    plt.yscale("log", base=10)
    plt.xlabel("Iteration count")
    plt.ylabel("2-norm of error between iteration and true solution (log-scale)")
    plt.legend()
    plt.title(title)
    plt.show()

return xn
```

Listing 2: Code that calls the gradient descent solver to find the minimizer of the earlier quadratic

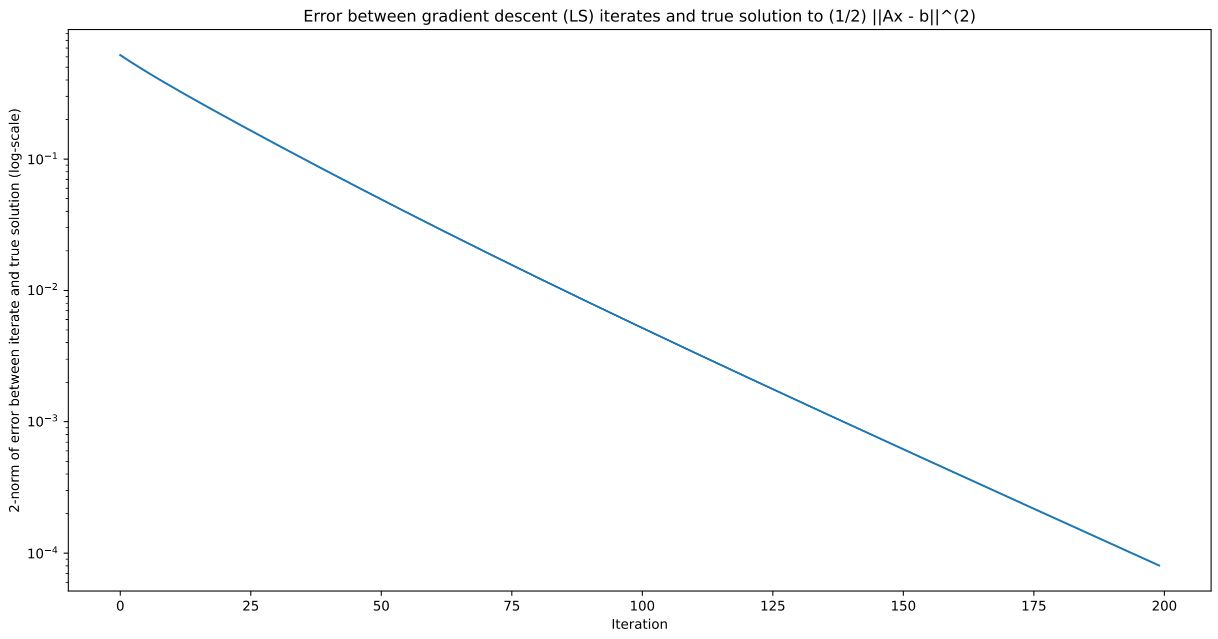
```
# Problem 2:
# Apply the gradient descent function to the quadratic
# f(x) = (1/2)*||Ax - b||^(2), where A is over-determined
# and has full column rank
m = 10
n = 5
A = np.random.rand(m, n)
b = np.random.rand(m)

print("Does A have full column rank? ", np.linalg.matrix_rank(A) == n)

f = lambda x: (1/2) * np.linalg.norm(A @ x - b, 2)**2
grad_f = lambda x: A.T @ (A @ x - b)

# Compute the least-squares solution to the above problem
x_star, residuals, rank, s = np.linalg.lstsq(A, b, rcond=None)

# Use the zero vectors as our initial guess
x0 = np.zeros((n, ))

# Compute an approximate solution using our gradient descent function
stepsize = 1/(np.linalg.norm(A, ord=2)**2)
error_title = "Error between gradient descent iterates
```

```
and true solution to (1/2) ||Ax - b||^(2)"
xn = gradientDescent(f, grad_f, x0, x_star,
error_title, max_iters=200, stepsize=stepsize,
 tol=1e-7, compute_error=True)
```

3. Problem 3

   I then implemented another gradient descent solver that took the same inputs as the previous gradient descent solver, but also included a backtracking line search procedure that refines the stepsize by checking the Armijo condition. The implementation also accounts for the Armijo condition being met in one iteration so as to not make the stepsize too small from iteration to iteration.

   I applied this gradient descent solver to the same quadratic as from before, and attached the resulting error plot below.



Listing 3: Implementation of the gradient descent method with a backtracking line search procedure

```
def gradientDescentLS(f, grad_f, x0, x_star, error_title,
eval_title, max_iters=100, stepsize=1, tol=1e-7, compute_error=False):
    num_iters  = 0
    func_evals = []
    errors     = []
    xn         = np.zeros(x0.size)

    # Typical line search parameters
    c = 1e-4
    rho = 0.9
    t = stepsize
    while (num_iters < max_iters):
        num_iters += 1

        # Perform a line search to refine the stepsize
        # Our descent direction will be the negative gradient at the current iterate
        armijo_cond_iters = 0
        pn = -grad_f(x0)
        while (f(x0 + t*pn) > f(x0) - c*t*np.vdot(grad_f(x0), pn)):
            t *= rho
            armijo_cond_iters += 1

        xn = x0 - t*grad_f(x0)
        func_evals.append(f(xn))

        # Compute the error between the new iterate and the true solution,
        # if provided
        if compute_error:
            errors.append(np.linalg.norm(xn - x_star, 2))

        if (np.linalg.norm(xn - x0, 2) < tol):
            break

        x0 = xn

        # Increase the stepsize by a factor of two
        # if we decrease the stepsize only once
        if armijo_cond_iters == 1:
            t *= 2

    # Make a plot of error between each iterate and the true solution
    # as a function of the iteration count
    if compute_error:
        plt.figure()
        plt.plot(np.arange(num_iters), errors,
        label="Error between iterate and true solution")
        # plt.xscale("log", base=10)
        plt.yscale("log", base=10)
        plt.xlabel("Iteration")
        plt.ylabel("2-norm of error between iterate and true solution (log-scale)")
        plt.title(error_title)
        plt.show()

    # Make a plot of the function evaluations at each iterate as a
    # function of the iteration count
    plt.figure()
    plt.plot(np.arange(num_iters), func_evals)
    plt.xlabel("Iteration")
    plt.ylabel("Function value at each iterate")
    plt.title(eval_title)
    plt.show()
    return xn
```

Listing 4: Code that calls the gradient descent solver with backtracking line search procedure on the same quadratic from earlier

```
# Apply the gradient descent function to the same quadratic as above, but
# now use a line search
error_title_LS = "Error between gradient descent (LS) iterates and true solution to (1
eval_title     = "Evaluation of (1/2) ||Ax - b||^(2) at each iterate"
xn_LS = gradientDescentLS(f, grad_f, x0, x_star, error_title_LS, eval_title, max_iters
                          tol=1e-7, compute_error=True)

# Problem 3
# Compute the 2-norm of the difference between the approximate solutions
# computed via gradient descent and gradient descent with line search
print("2-norm of difference between GD and GD with linesearch solutions: ",
      np.linalg.norm(xn - xn_LS, 2))
```

The 2-norm of the difference of the approximate solutions computed by the two gradient descent implementations is 0 (`2-norm of difference between GD and GD with linesearch solutions: 0.0`).

4. Problem 4, part (a)

   Below are the fast implementations of the logistic-loss and its corresponding gradient functions.

Listing 5: Fast implementations of $\ell(\mathbf{w})$ and $\nabla\ell(\mathbf{w})$

```
# Problem 4
# Load the spam data
spamData = sio.loadmat("spamData")

# Pre-process the training and test data
Xtrain = np.log(spamData["Xtrain"] + 0.1)
Xtest  = np.log(spamData["Xtest"]  + 0.1)

print(Xtrain[0, :])
print(Xtest[0, :])

# Load the training and test labels, and change any 0's to -1's
ytrain = np.array(np.reshape(spamData["ytrain"], newshape=(spamData["ytrain"].size, ))
ytrain[ytrain == 0] = -1

ytest  = np.array(np.reshape(spamData["ytest"], newshape=(spamData["ytest"].size, )),
ytest[ytest   == 0] = -1

def lr(w):
    return np.sum(np.log(1 + np.exp(-ytrain * np.matmul(Xtrain, w))))

def grad_lr(w):
    mu = 1/(1 + np.exp(-ytrain * np.matmul(Xtrain, w)))
    return (np.matmul(-Xtrain.T, ytrain*(1 - mu)))
```

```
sigmoid = lambda a: np.exp(a)/(1 + np.exp(a))

# Problem 4, part (a)
# Time the fast implementation of the gradient of the negative log-likelihood functior
# and make sure it takes less than 1 second
w = np.random.rand(Xtrain.shape[1])
print("Time taken to evaluate the fast implementation of the gradient of loss functior
      timeit.timeit('"grad_lr(w)"', number=1000000))
```

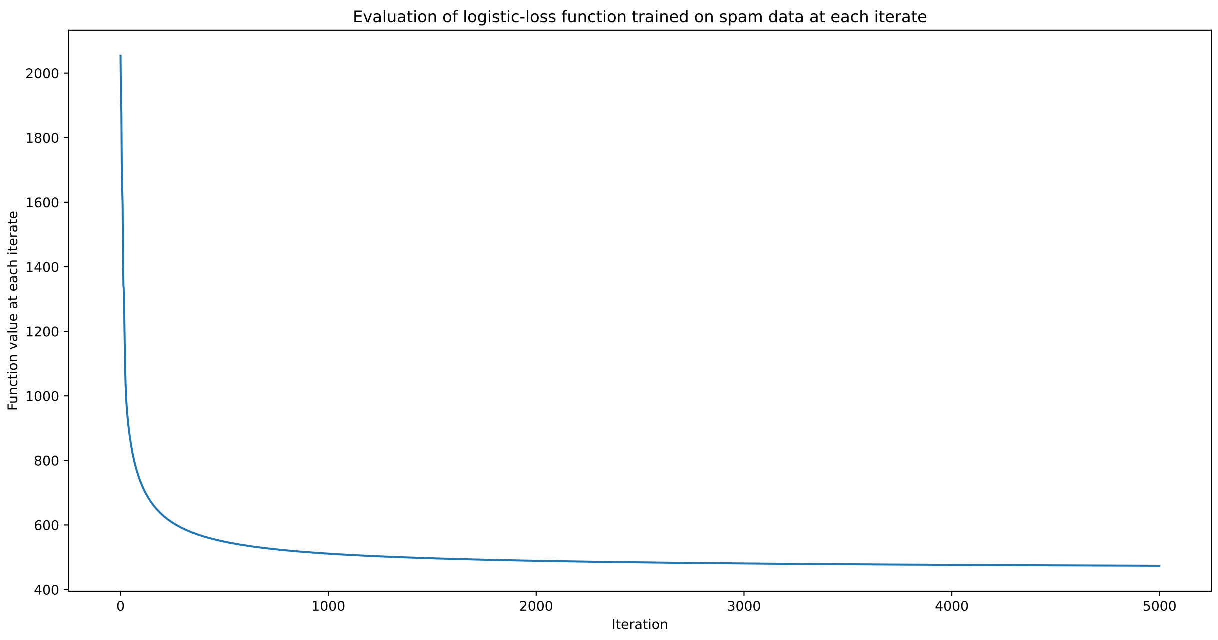
On my machine, the timer reports that the averaged time to evaluate the fast implementation of the gradient is roughly 0.047 seconds (`Time taken to evaluate the fast implementation of the gradient of loss function:  0.04732380001223646`).

5. Problem 4, part (b)

   I ran the gradient descent solver with line search on the logistic function (using only the training data) to obtain an approximate classified $\mathbf{w}$. The relevant code is below, and I have also attached a plot of the logistic function evaluations as a function of iteration count.

Listing 6: Code that calls the gradient descent solver with line search on the logistic function trained on the training data

```
# Problem 4, part (b)
# Run the gradient descent solver with line search on the logistic function
# on the training data to obtain a classified w
w0 = (1/1000)*np.ones(Xtrain.shape[1])
eval_title_lr = "Evaluation of logistic-loss function trained on spam data at each ite
w_GDLS = gradientDescentLS(lr, grad_lr, w0, w0, "", eval_title_lr, max_iters=5000, ste
                           tol=1e-7, compute_error=False)

# Perform classification on the training and testing data
training_computed_labels =
np.array([sigmoid(np.vdot(w_GDLS, Xtrain[i, :])) for i in range(Xtrain.shape[0])])
for i in np.arange(training_computed_labels.size):
    if training_computed_labels[i] > 0.5:
        training_computed_labels[i] = 1;
    else:
        training_computed_labels[i] = -1;

testing_computed_labels  =
np.array([sigmoid(np.vdot(w_GDLS, Xtest[i, :])) for i in range(Xtest.shape[0])])
for i in np.arange(testing_computed_labels.size):
    if testing_computed_labels[i] > 0.5:
        testing_computed_labels[i] = 1;
    else:
        testing_computed_labels[i] = -1;

print("Training data misclassification rate: ",
1 - (np.sum(ytrain == training_computed_labels)/ ytrain.size))
print("Testing data misclassification rate: ",
1 - (np.sum(ytest  == testing_computed_labels) / ytest.size))
```



After computing the approximate classifier $\mathbf{w}$ and computing the misclassification rate on the training and testing data, I found that my training data misclassifcation rate was approximately 0.0525 and that my testing data misclassification rate was approximately 0.0559, i.e.,

(a) Training data misclassification rate: 0.052528548123980445

(b) Testing data misclassification rate: 0.05598958333333337

In running the gradient descent solver with line search on the logistic function, I saw that the logistic function evaluated at the final iterate returned from the solver was approximately 473.64, e.g., $\ell\left(\mathbf{w_n}\right) \approx 473.647496340764$. From the plot, I saw the function values stagnate beyond 1000 iterations, but I used 5000 iterations to obtain the desired misclassification rates on the training and testing data sets. For the most part, implementing the code was not difficult, but I did run into several issues with NumPy's shape broadcasting and the logistic function returning extremely large values that caused numerical overflow. However, this did not terribly impact the accuracy of the approximate classifier.