MA576 HW5

Aidan Fischer

November 2023

I pledge my honor that I have abided by the Stevens Honor System.

Q1

X is a random variable that takes the values $z_1, z_2, ..., z_n$ with $P(z_i) = p_i$. $p_i \geq 0$ and $\sum_i p_i = 1$.

Want to find the minimizer x^* of $f(x) = \mathbb{E}[(X - x)^2]$

Expanding the expected value function,

$f(x) = \mathbb{E}[(X - x)^2]$

$= \sum_i (z_i - x)^2 p(z_i)$

$= \sum_i (z_i - x)^2 p_i$

$f'(x) = \sum_i -2(z_i - x)p_i$

$= \sum_i (-2z_i p_i + 2xp_i)$

$= \sum_i -2z_i p_i + \sum_i 2xp_i$

$= \sum_i -2z_i p_i + 2x \sum_i p_i$

$= -2(\sum_i z_i p_i) + 2x$

Finding critical points

$0 = -2(\sum_i z_i p_i) + 2x$

$2(\sum_i z_i p_i) = 2x$

$\sum_i z_i p_i = x$

Since the second derivative is 2, which is positive, we know that this critical point is a minimizer. Because f is a degree 2 polynomial in the single variable x, we know it is coercive, therefore this critical point is a global minimizer.

So, $x^* = z^T p$, where z and p are the vectors consisting of each z_i and p_i respectively.

Q2

$$\underset{x \in \mathbb{R}^n}{minimize} \frac{1}{2} ||Ax - b||_2^2$$

1)

FONC

$\nabla \frac{1}{2}||Ax-b||_2^2$
$= \nabla \frac{1}{2}(Ax-b)^T(Ax-b)$
$= \nabla \frac{1}{2}(x^TA^T - b^T)(Ax-b)$
$= \nabla \frac{1}{2}(x^TA^TAx - b^TAx - x^TA^Tb + b^Tb)$
$= \nabla \frac{1}{2}(x^TA^TAx - 2b^TAx + b^Tb)$
$= \frac{1}{2}(2A^TAx - 2A^Tb)$
$= A^TAx - A^Tb$
$A^TAx - A^Tb = 0 => A^TAx = A^Tb => x^* = (A^TA)^{-1}A^Tb$

SONC

$\nabla^2 \frac{1}{2}||Ax-b||_2^2 = \nabla A^TAx - A^Tb = A^TA$
A^TA is positive definite only if it is nonsingular (A must have full rank)
x^* is

```
[-0.01574095  0.12101415  0.04518362  0.04764731 -0.1623104   0.00908704
 -0.0947885  -0.15907534  0.10031668  0.09054754  0.11128148  0.1171073
 -0.01742716  0.16248451  0.06777838  0.05863189 -0.06047807  0.07110943
  0.0825872   0.04114662]
```

See Appendix 2.1 for code.

2-4

At the end of this part, I will display 5 legended plots comparing the combinations of stepsizes and stopping conditions. Prior to that, I will show the stepsizes

For each, a superscript k refers to iteration k

Fixed step: $\alpha = \frac{1}{\lambda_{max}(A^TA)}$

$\lambda_{max}(A^TA) = 409.70723906911905 => \alpha = 0.002440767222644305$

Exact Line Search: $\alpha_k \in arg\min_{\alpha \geq 0} f(x^k + \alpha d^k)$

For quadratic functions, it is possible to compute α exactly.

This problem is quadratic (we have $f(x) = \frac{1}{2}(x^TA^TAx - 2b^TAx + b^Tb$ from FONC derivation). Here, $Q = \frac{1}{2}A^TA$ and $b = -\frac{1}{2}A^Tb$ ("b" from quadratic form in terms of "b" from our problem)

The formula is $\alpha_k = -\frac{d^{kT}\nabla f(x^k)}{2d^{kT}Qd^k} = -\frac{d^{kT}2(Qx^k+b)}{2d^{kT}Qd^k}$

So, $\alpha_k = -\frac{d^{kT}(A^TAx^k - A^Tb)}{d^{kT}A^TAd^k}$

Armijo's rule is

$f(x^k + \alpha d^k) - f(x^k) < \sigma\alpha\nabla f(x^k)^T d^k$

With parameters $s > 0, \sigma \in (0,1), \beta \in (0,1)$

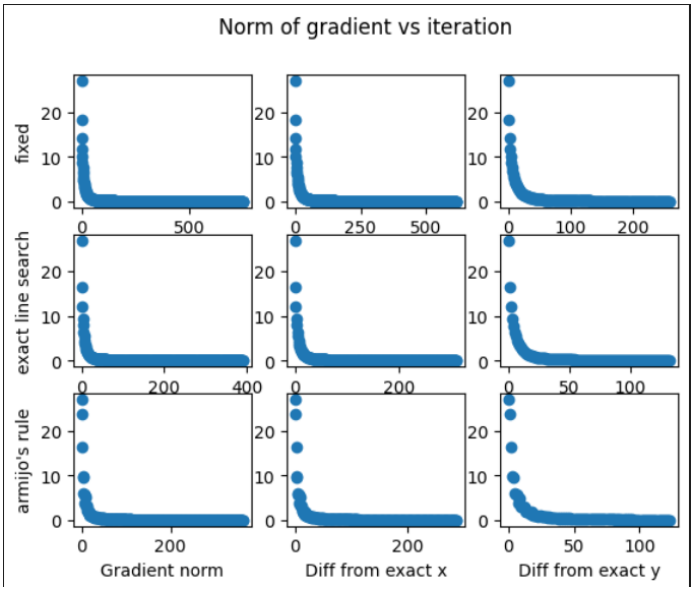
α_k is set to initial guess s, and updated to $\beta\alpha_k$ until the inequality is satisfied.

For this problem, I will choose $s = 0.5, \sigma = 1/4, \beta = 3/4$

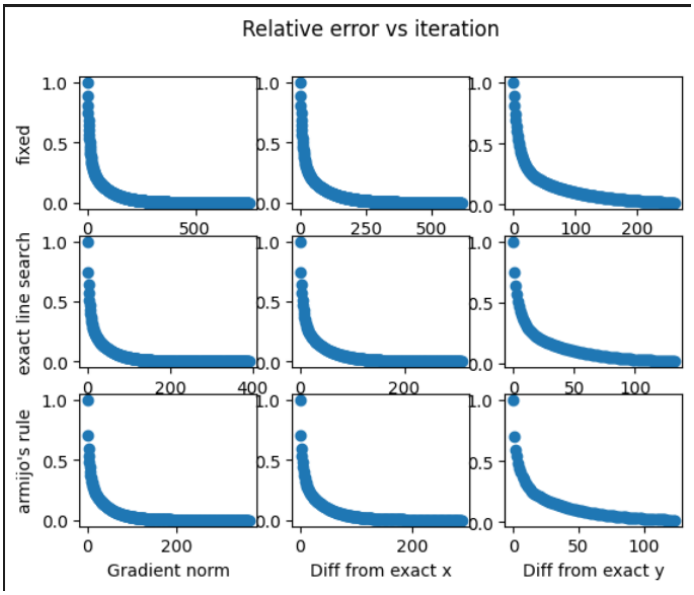
See Appendices 2.2 for setup code, 2.3 for algorithm running code, and 2.4 for plot generation code.

The main title of the plots gives the axes. The first part gives the x axis values, and the second part gives y axis values (i.e. a vs b means a is on the x axis and b is on the y axis)

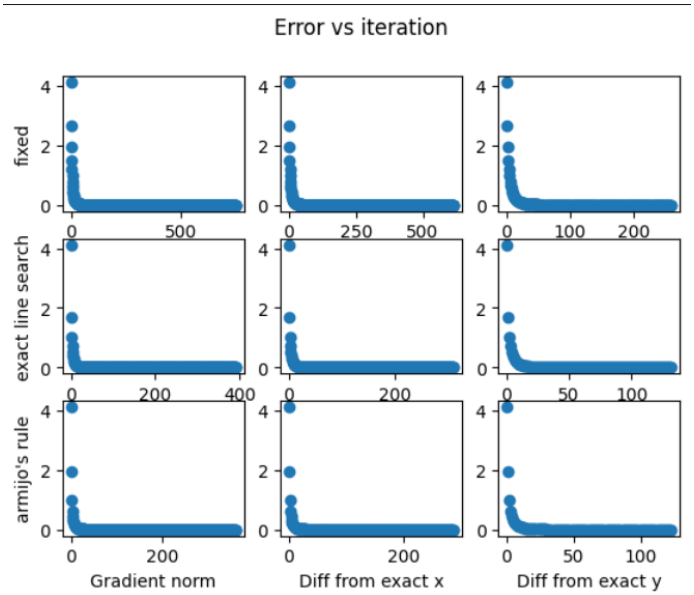
The different step sizes and stop conditions are paired up in a grid.



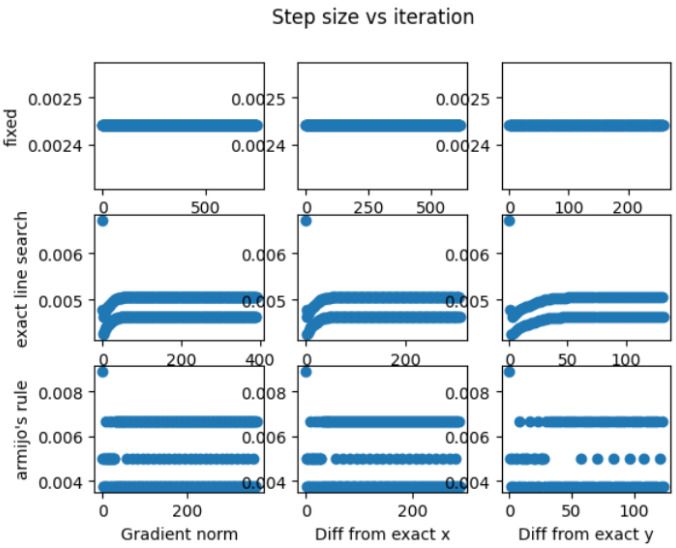
The norm of the gradient tapers off quickly in all cases. It takes a few iterations to reach close to the answer, but a lot more to get within epsilon.



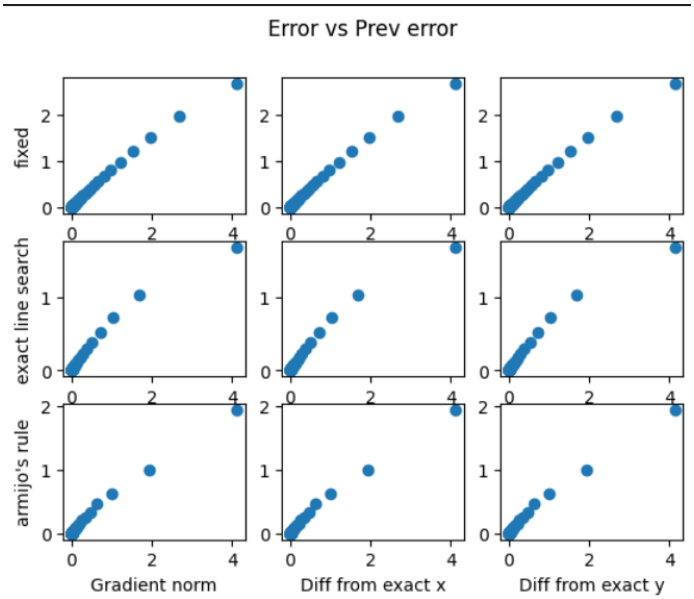
While relative error still tapers off, it does so slower than the gradient does.



Error tapers off very quickly.



Step size varies a ton in the two methods where it actually changes. It seems to cycle between a few different values. For exact line search, it takes a few iterations but settles between 2 values. For armijo's rule, it cycles between 3 values.



The distancing between points shows how fast convergence happens. The closeness of the points near 0,0 show that the error decreases slower over time, but the fact they stay near the line y=x means the error usually decreases.

Q3

Code for descent is similar to problems 2 and 6-7.

For exact line search, since we don't have an analytical solution for the stepsize, I will be using Golden Search to minimize $\varphi(\alpha) = f(x^k + \alpha d^k)$

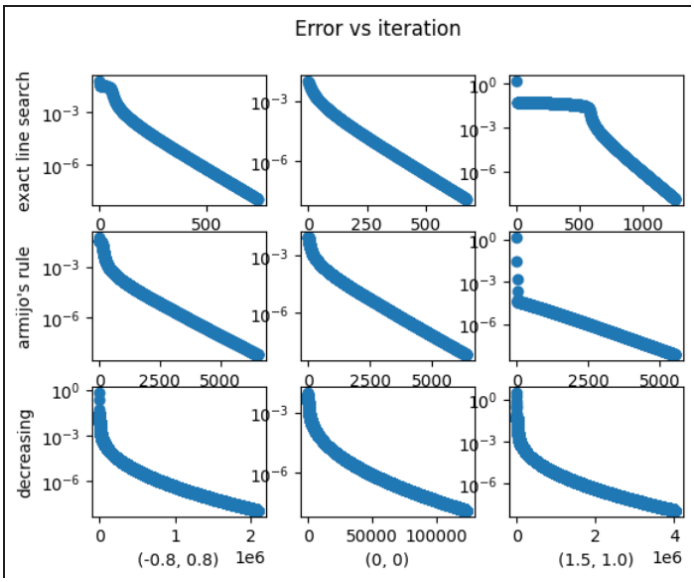
See Appendex 3.1 for golden search code.

For decreasing, since c has to be tuned, I start c at 4 and run a binary search. If the algorithm takes too long, increase c to move further. If it balloons to infinity (i.e. jumps too far and the gradient grows exponentially), then decrease c. The c change is divided by 2 each time.

a)

Steepest Descent

For $x^0 = (-0.8, 0.8)$, c=0.5 For $x^0 = (0, 0)$, c=2 For $x^0 = (1.5, 1)$, c=0.375



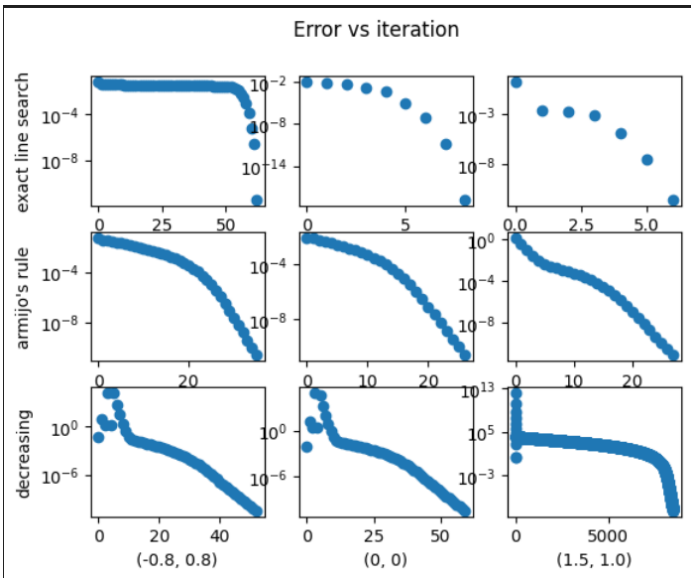
b)

Newton's Method

I used hybrid Newton's because the Hessian is not guaranteed to be pd

c is 2 in all cases

In both Newton's and Steepest, the rate of convergence is what I expect.



Q4

1)

$$f(x) = -\sum_{j=1}^{m} log(1 - a_j^T x) - \sum_{i=1}^{n} log(1 + x_i) - \sum_{i=1}^{n} log(1 - x_i)$$

$$\nabla f(x) = \sum_{j=1}^{m} \frac{a_j^T}{1 - a_j^T x} - [\frac{1}{1 + x_1}, \frac{1}{1 + x_2}, ...] + [\frac{1}{1 - x_1}, \frac{1}{1 - x_2}, ...]$$

$$\nabla^2 f(x) = \sum_{j=1}^{m} \frac{a_j a_j^T}{(1 - a_j^T x)^2} + \begin{pmatrix} \frac{1}{(1+x_1)^2} & 0 & ... \\ 0 & \frac{1}{(1+x_2)^2} & 0 \\ ... & 0 & ... \end{pmatrix} + \begin{pmatrix} \frac{1}{(1-x_1)^2} & 0 & ... \\ 0 & \frac{1}{(1-x_2)^2} & 0 \\ ... & 0 & ... \end{pmatrix}$$

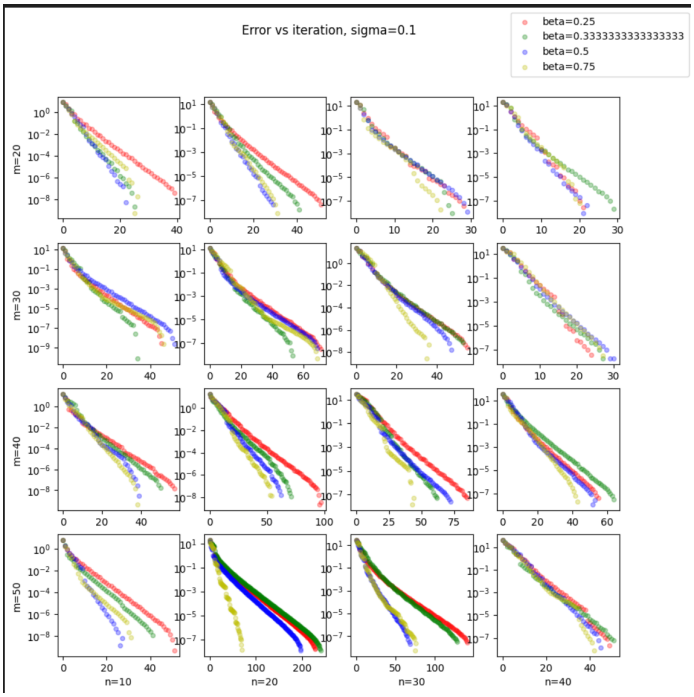
2-3

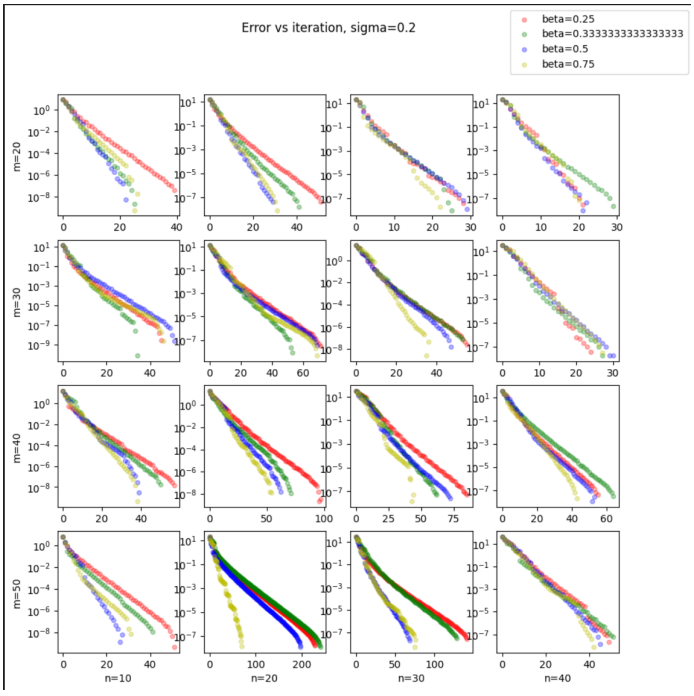
For this problem, I selected 4 different values each for sigma, beta, m, and n.

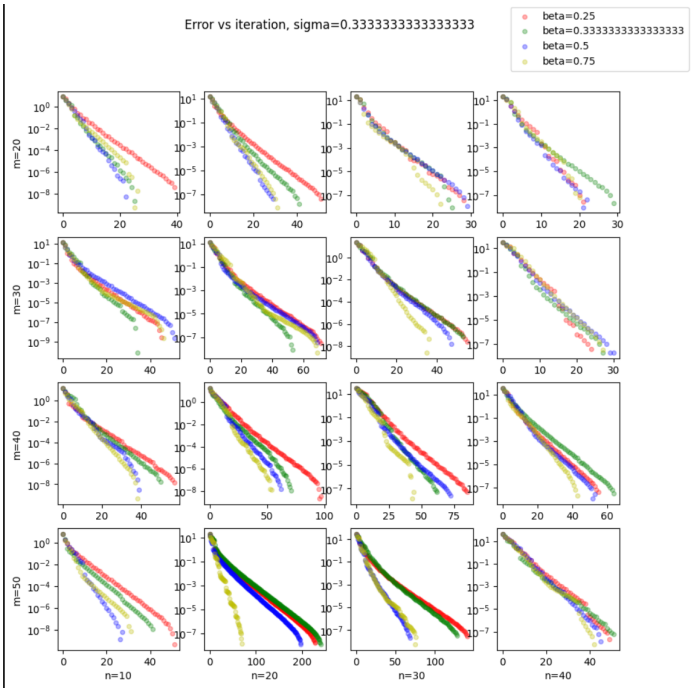
Since this results in 256 unique combinations of these values, I had to be a bit smart with how I constructed the graphs.

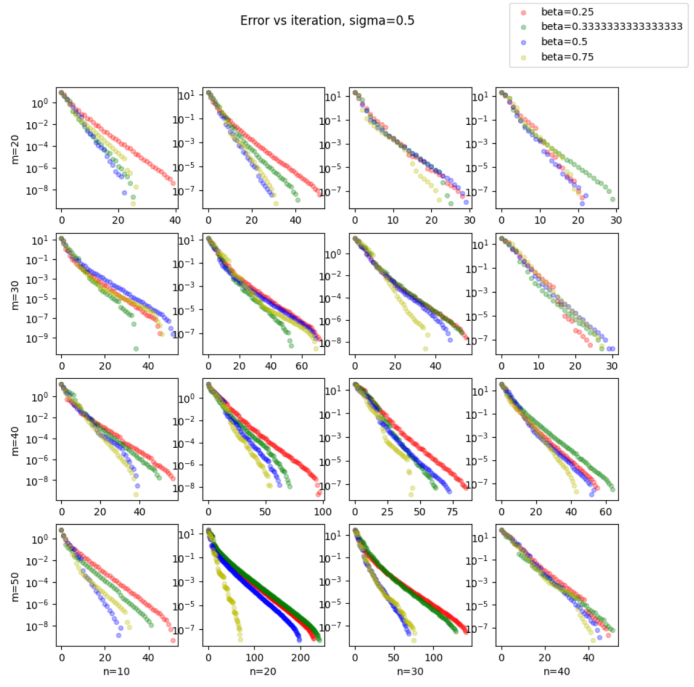
As before, I plot multiple graphs in one figure. M and N values form the plot grid. Each plot has four graphs, one for each beta value. There are four figures, one for each sigma value.

Here are the graphs for steepest descent


Error vs iteration, sigma=0.1
beta=0.25
beta=0.3333333333333333
beta=0.5
beta=0.75
m=20
m=30
m=40
m=50
n=10
n=20
n=30
n=40


Error vs iteration, sigma=0.2
beta=0.25
beta=0.3333333333333333
beta=0.5
beta=0.75
m=20
m=30
m=40
m=50
n=10
n=20
n=30
n=40


Error vs iteration, sigma=0.3333333333333333
beta=0.25
beta=0.3333333333333333
beta=0.5
beta=0.75
m=20
m=30
m=40
m=50
n=10
n=20
n=30
n=40

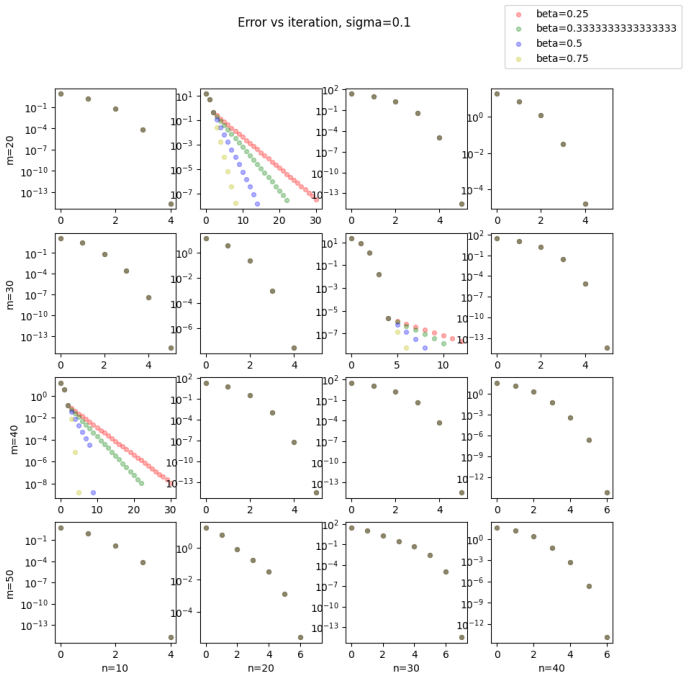

Error vs iteration, sigma=0.5
beta=0.25
beta=0.3333333333333333
beta=0.5
beta=0.75
m=20
m=30
m=40
m=50
n=10
n=20
n=30
n=40

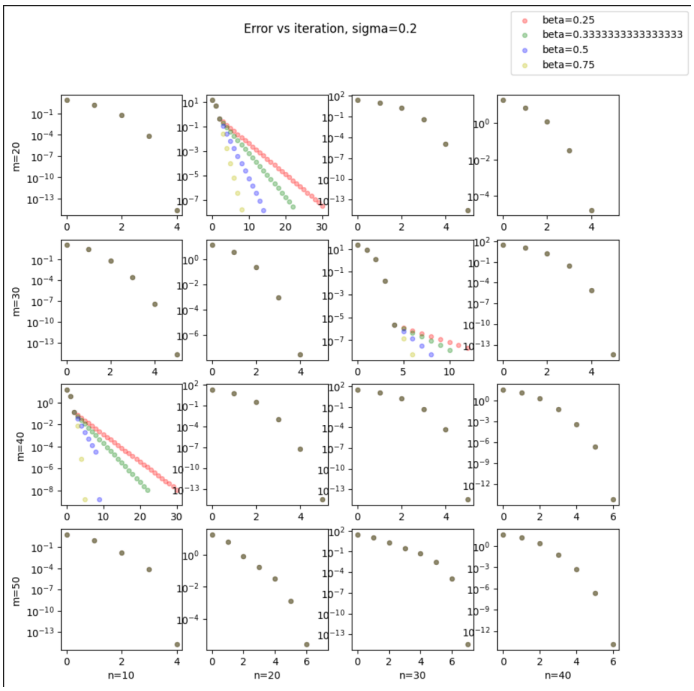
The shortest algorithms took about 30 iterations. Regardless of sigma, and with beta=0.75, these happen at m=20 and n=30, m=20 and n=40, m=30, n=30

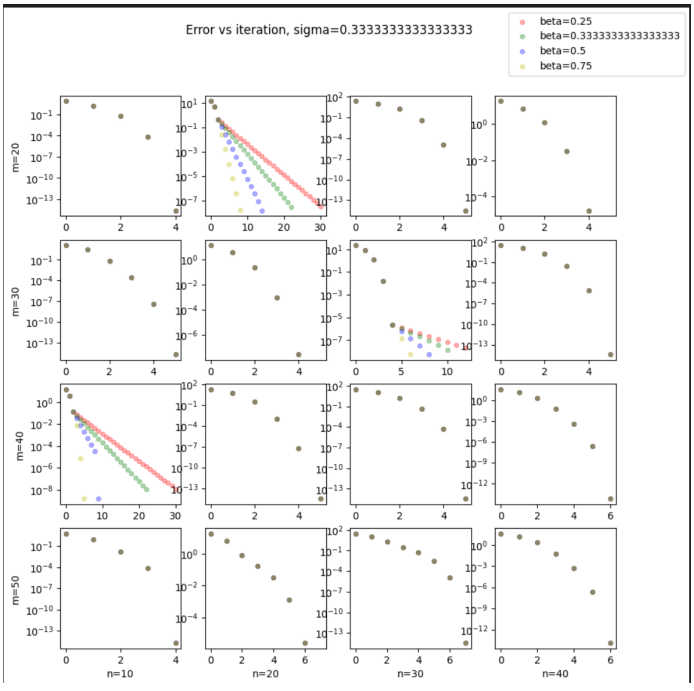
The longest took around 200 iterations at m=50, n=20

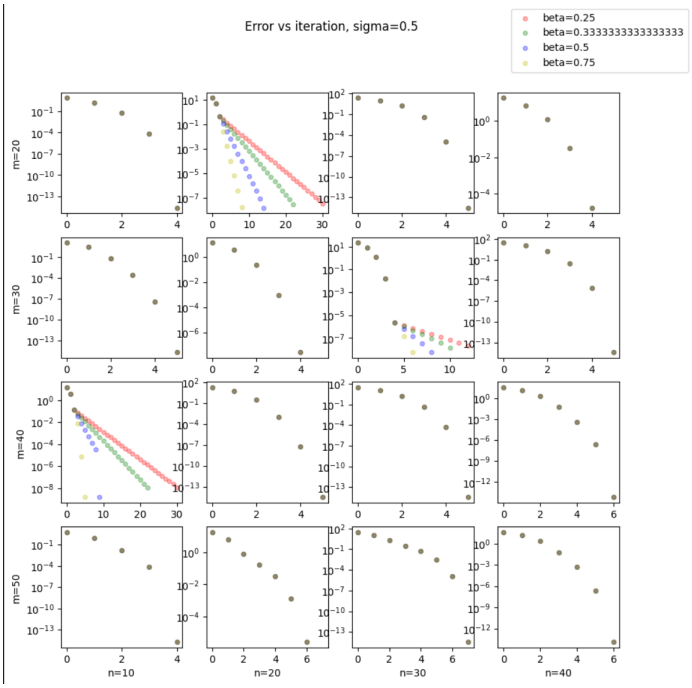
4

Here are the graphs for Newton's method




Error vs iteration, sigma=0.2
beta=0.25
beta=0.3333333333333333
beta=0.5
beta=0.75
m=20
m=30
m=40
m=50
n=10
n=20
n=30
n=40


Error vs iteration, sigma=0.3333333333333333
beta=0.25
beta=0.3333333333333333
beta=0.5
beta=0.75
m=20
m=30
m=40
m=50
n=10
n=20
n=30
n=40



As you can see, Newton's method performs significantly better, completing in less than 10 iterations in most cases, however in a couple cases it takes up to 30. Beta and sigma do not seem to affect the results in most cases, but do have some variation in the cases where Newton's takes 30 iterations.

Overall, Newton's method's convergence rate is better than steepest descent in this problem.

I am guessing the specific A matrices being generated would affect these results somewhat.

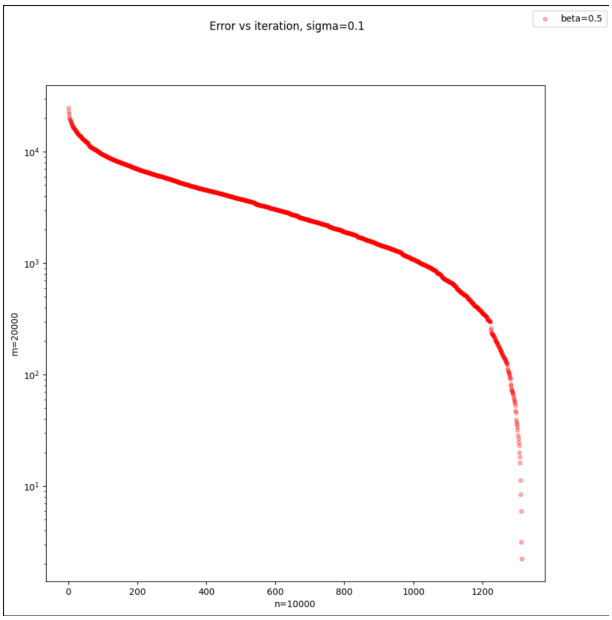
iteration code is similar to other problems so not including in appendix. Only difference is which values I loop over to gather plotting data.

See appendix 4.1 for function generators (Generators for A and x0, f, df, and hf). See appendix 4.2 for plot generation code (has some differences to prior plot code).

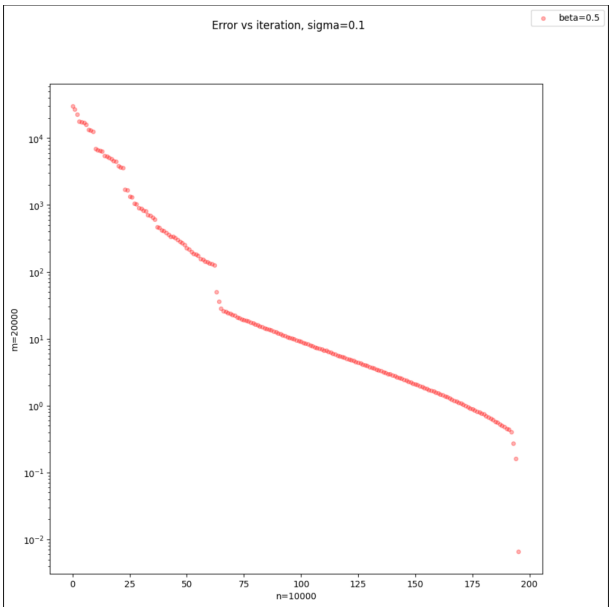
Q5

During coding for this problem, I realized that my functions in python were extremely poorly optimized. It would not be feasible to run at the given m and n. See Appendix 5.1 for optimized code (and see comments in Hessian function for explanation of the optimization). This optimization reduced the approximate time for each Newton iteration from 3 hours to 40 seconds.

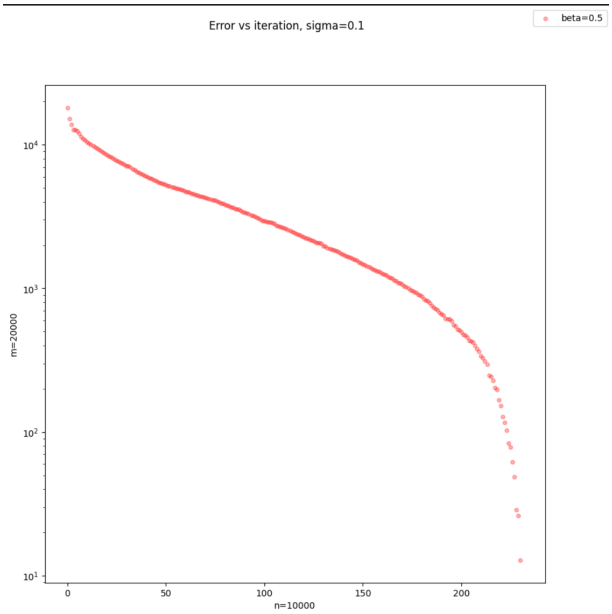
Here is the error graph for steepest descent.



Steepest descent failed to converge after 1.5 hours, so I stopped it early. Here is the error graph for Newton's method



Here is the error graph for Diagonal approximation Newton's



The convergence rate of Newton's was the fastest, but it's iteration time was far longer than the other methods.

The diagonal approximate newton's has a similar convergence path to steepest descent but over fewer iterations. However, these iterations were still far slower than steepest descent.

Q6

1-2

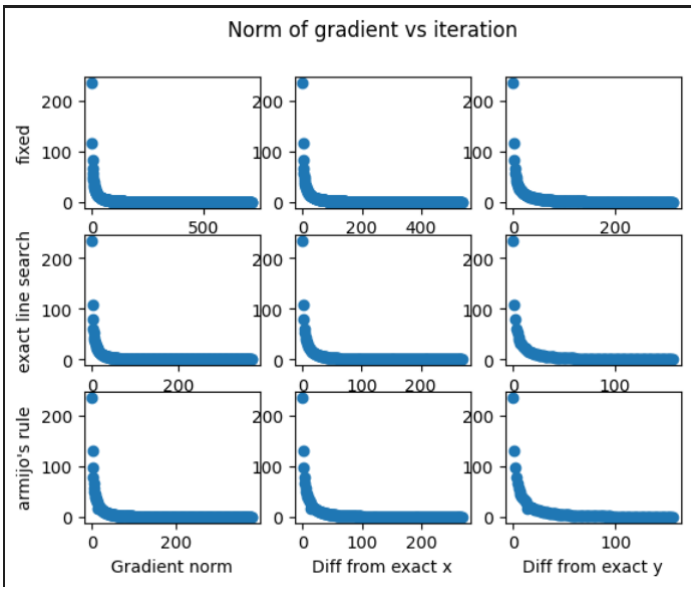
See Appendix 6.1 for conjugate method code. See Appendix 6.2 for A,b random generation code

Let Q=I for the purposes of Q-conjugacy

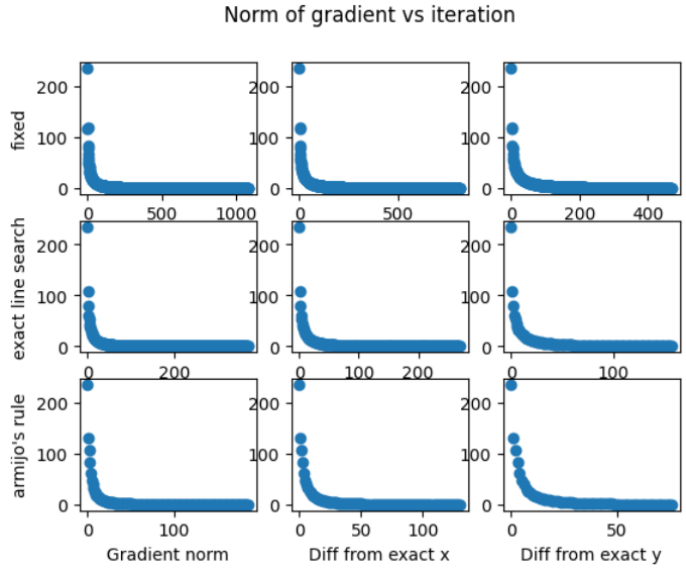
The conjugate method uses Q-conjugate directions with the descent method. For analysis of the results, in addition than the one graph asked for in part 3 of this problem, I will include the gradient norm vs iteration number graph for the random A and b for both steepest descent and conjugate method and compare.

Rather than just setting the next direction to the negative of the gradient, it is set to a linear combination of the negative gradient and itself ($d^{k+1} = -\nabla f(x^k) + \beta^k d^k$) where β^k depends on a formula that makes the new direction Q-conjugate to the old one.

Steepest descent



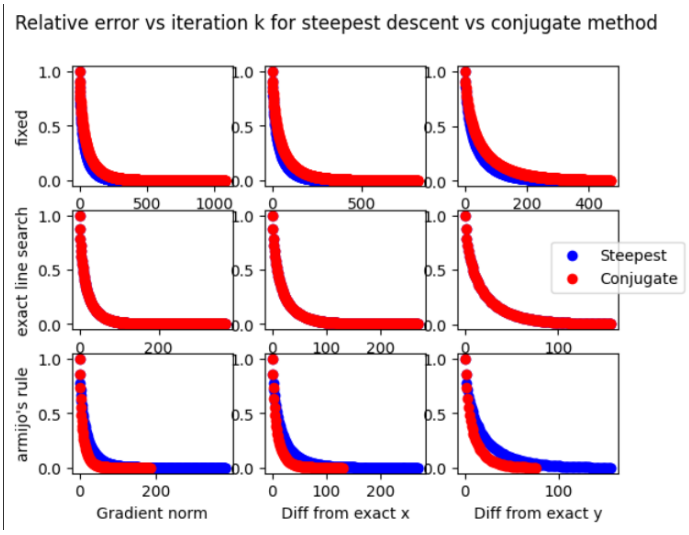
Conjugate method



As you can see, the conjugate method has a massive effect on the efficiency of the descent algorithm. For fixed stepsize, the conjugate method is actually slower. I theorize this happens because the stepsize is not tuned. The guessed x^k dances around the answer for a while before it can reach a stopping condition. For exact line search, the number of iterations is about he same. However,

Armijo's rule converges in about half as many iterations when using the conjugate method.

3



For fixed stepsize and exact line search, the relative error remains about the same overall, but as seen in the previous plot, conjugate takes longer to reach the stopping condition. For the other armijo's rule, conjugate method's relative error converges to 0 faster than steepest descent.

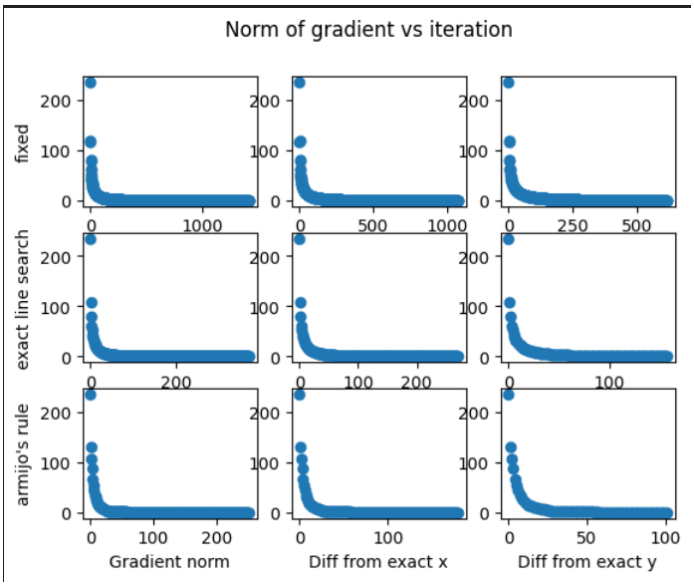
Since the plot code is similar to prior plot code (just with an extra scatter plot per subplot and with a legend added), I do not feel the need to add it to the appendix.

4

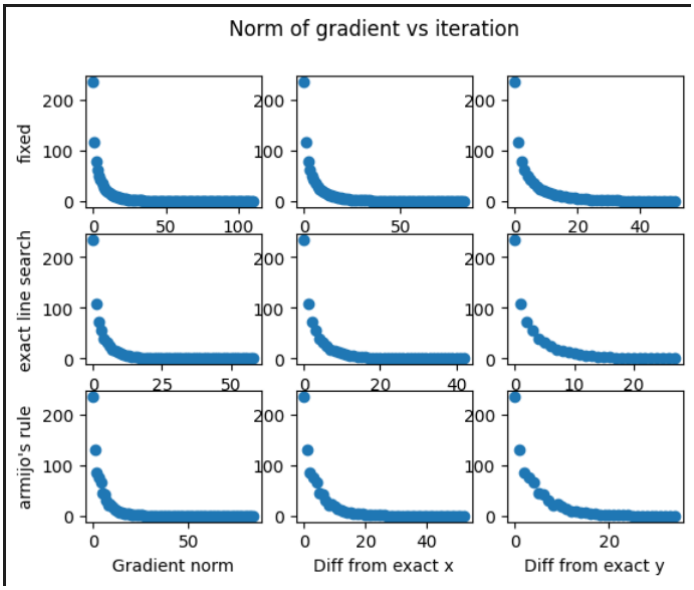
Since the code for this problem is similar to the prior code for conjugate method (all I did was replace the beta line with the appropriate formulas), I do not see the need to add it the appendix as a whole. See Appendix 6.3 for the lines that apply the Polak-Ribierre and Fletcher-Reeves formulae.

Polak-Ribierre is defined as $\beta = \frac{(g^{k+1})^T Q d^k}{(g^k)^T g^k}$

Norm graph for Polak-Ribierre



Fletcher-Reeves is defined as $\beta = \frac{(g^{k+1})^T g^{k+1}}{(g^k)^T g^k}$
Norm graph for Fletcher-Reeves



Fletcher-Reeves converges far faster than Polak-Ribierre FOR THIS PROBLEM. I doubt this is the case for all problems, else Polak-Ribierre wouldn't be useful. Fletcher-Reeves is fast even for fixed stepsize in this problem.

Polak-Ribierre is about as fast as the default conjugate gradient beta formula

for quadratics that I used earlier.

Q7

Quasi-Newton methods select a direction using $d^k = -V^k \nabla f(x^k)$, where V^k is a symmetric matrix constructed during each iteration.

V^k is constructed to be a computationally-cheap approximation of $\nabla^2 f(x^k)^{-1}$, the inverse of the Hessian of f at x^k.

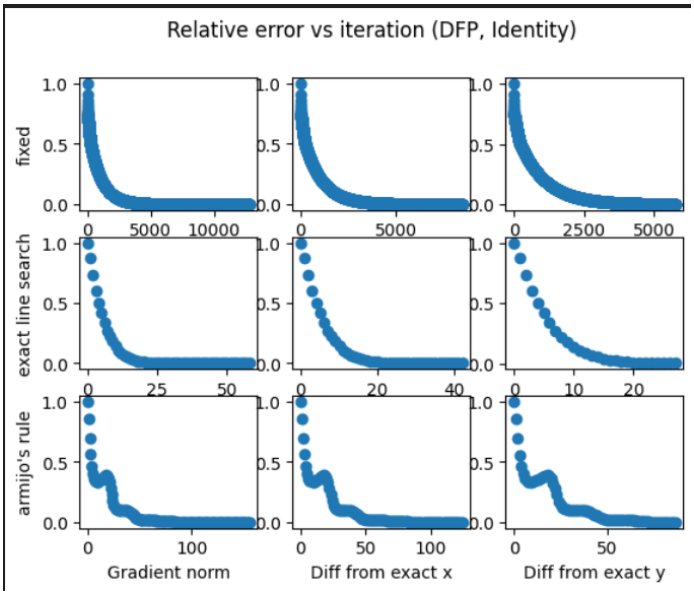
The starting approximation V^0 has no specific formula and usually requires an educated guess (Multiple of identity and approximate Hessian at x^0 are common choices)

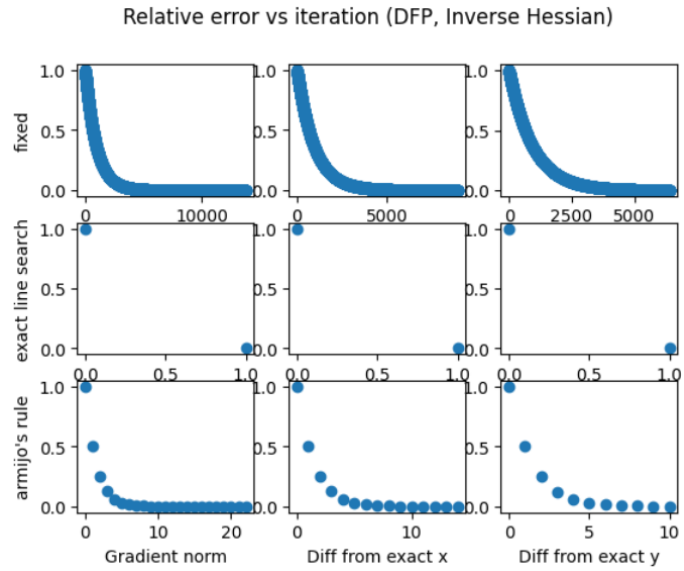
For this problem, we will try $V^0 = I$ and $V^0 = \nabla^2 f(x^0)^{-1}$. Because this problem is quadratic, we know the Hessian. The Hessian is $2Q = A^T A$ (from prior questions), and thus the initial guess for V^0 is $(A^T A)^{-1}$

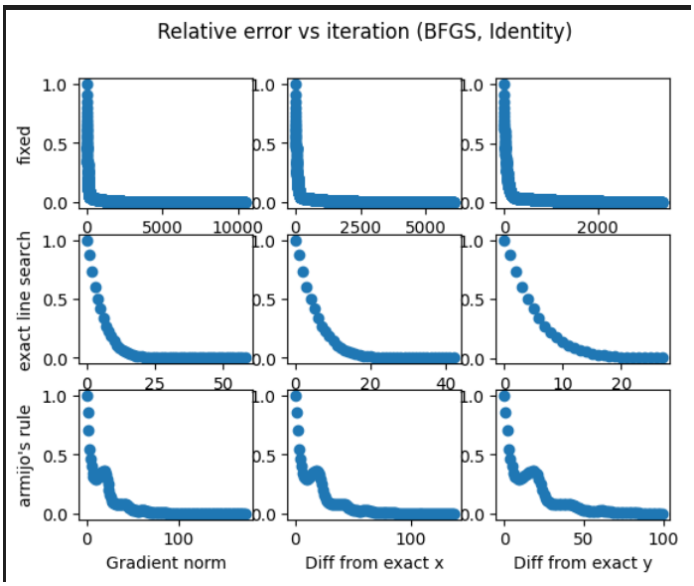
See Appendix 7.1 for full code (iteration code and plot code).

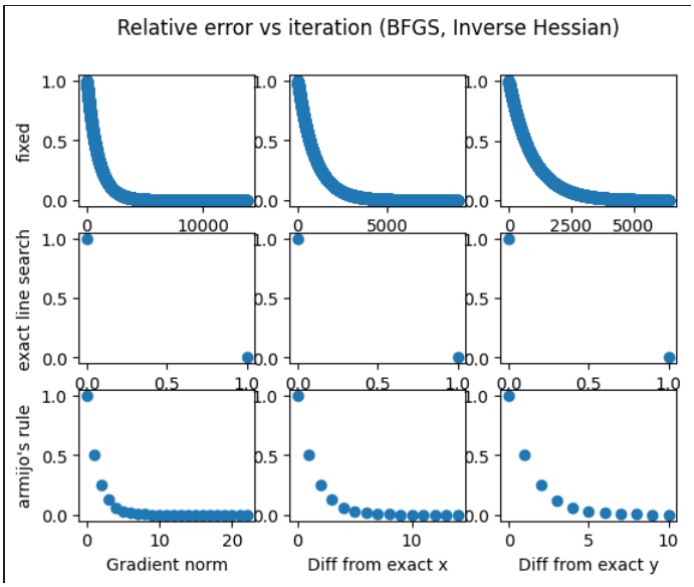
I will display the same plot for question 6 part 3 (relative error) for each V^k, V^0 pair and analyze the results with respect to steepest descent and conjugate descent and with respect to themselves.

Note that since I am using the same random seed as in problem 6, this problem is using the exact same A and b as problem six. Therefore it makes sense for me to directly compare the performance of Quasi Newton with the graphs produced for problem 6.




Relative error vs iteration (DFP, Inverse Hessian)
fixed
exact line search
armijo's rule
Gradient norm
Diff from exact x
Diff from exact y


Relative error vs iteration (BFGS, Identity)
fixed
exact line search
armijo's rule
Gradient norm
Diff from exact x
Diff from exact y



In all cases, Quasi-Newton performs significantly worse than steepest and conjugate descent for a fixed stepsize. It is obvious that Quasi-Newton must use a stepsize that adjusts based on the problem conditions.

For exact line search and Armijo's rule, the situation is more interesting.

Exact line search performs extremely well in all cases (even completing in a single iteration for all stop conditions whenever V^0 is selected to be the inverse Hessian!). It performs better than Armijo's rule in all cases here. It also performs better than steepest and conjugate descent in all cases.

Armijo's rule, while performing worse when compared to exact line search for Quasi Newton, is also interesting in its own right. Whenever V^0 is selected to be the Identity matrix, armijo's rule actually performs worse than conjugate method (ONLY when beta is calculated using Fletcher-Reeves), and better than conjugate method when V^0 is the inverse hessian. It performs better than steepest descent in all cases.

Selecting V^0 to be the inverse hessian worked significantly in this case, but this isn't always possible. We won't always know the hessian. However, approximating the Hessian at x^0 is possible, but isn't as fast as just choosing an appropriate multiple of I.

Appendix

2.1

```
A = np.loadtxt("A.txt",delimiter=",")
b = np.loadtxt("b.txt",delimiter=",")
xstar = np.linalg.inv(A.T @ A) @ A.T @ b
```

```
print(xstar)
```

2.2

```
# Part 2 - Steepest descent
f = lambda x: 1/2 * np.linalg.norm(A @ x - b,ord = 2) ** 2
df = lambda x: A.T @ A @ x - A.T @ b

lambdas = np.linalg.eigvals(A.T@A)
lambda_max = np.max(lambdas[np.isreal(lambdas)])
print(lambda_max)

# Set stepsizes in terms of lambda expressions that take
# in descent directions and current x
stepsize_fixed = lambda dk, xk: 1/lambda_max
print(1/lambda_max)
stepsize_exactline=lambda dk, xk: -(dk.T @ (A.T @ A @ xk - A.T @ b)) / \
                                    (2 * dk.T @ A.T @ A @ dk)
s = 0.5
sigma = 1/4
beta = 3/4
def stepsize_armijo(dk, xk):
    alpha = s
    while f(xk+alpha*dk) - f(xk) >= sigma * alpha * df(xk).T @ dk:
        alpha *= beta
    return alpha

# Define stopping conditions.

epsilon = 10 ** -4

stop_gradnorm = lambda xk: np.norm(A.T @ A @ xk - A.T @ b, ord=2) <= epsilon
stop_exactdiff = lambda xk: np.norm(xstar - xk) <= epsilon
stop_valdiff = lambda xk: np.norm(f(xstar)-f(xk), ord=2) <= epsilon

stepsizes = {"fixed": stepsize_fixed,
             "exact line search": stepsize_exactline,
             "armijo's rule": stepsize_armijo}
stopconds = {"Gradient norm": stop_gradnorm,
             "Diff from exact x": stop_exactdiff,
             "Diff from exact y": stop_valdiff}
```

2.3

```
# Run each stepsize, stopcond pair, storing data necessary for plotting

plotdata = {}
x0=np.zeros((A.shape[1],))

for stepname, stepsize in tqdm(stepsizes.items()):
    for stopname, stopcond in tqdm(stopconds.items()):
        xk = x0
        curalg = (stepname, stopname)
        plotdata.update({curalg: {"gradnorm": [],
                                  "relerr": [],
                                  "err": [],
                                  "stepsize": [],
                                  "k": 0}})
        xks = []
        while True:
            grad = df(xk)
            dk = -grad
            step = stepsize(dk, xk)
            plotdata[curalg]["gradnorm"].append(np.linalg.norm(grad))
            plotdata[curalg]["relerr"].append(
                np.linalg.norm(xstar - xk, ord=2)/ \
                np.linalg.norm(xstar, ord=2))
            plotdata[curalg]["err"].append(abs(f(xstar) - f(xk)))
            plotdata[curalg]["stepsize"].append(step)
            if stopcond(xk):
                break
            plotdata[curalg]["k"] += 1
            xks.append(xk)
            xk = xk + step * dk
```

2.4

```
def data_graph(yname, plotname):
    fig, axs = plt.subplots(3, 3)
    for i, stepname in enumerate(stepsizes.keys()):
        for j, stopname in enumerate(stopconds.keys()):
            alg = (stepname, stopname)
            y = plotdata[alg][yname]
            x = range(plotdata[alg]["k"]+1)
            axs[i,j].scatter(x,y)
            if j == 0:
                axs[i,j].set(ylabel=stepname)
            if i == 2:
                axs[i,j].set(xlabel=stopname)
```

```
    fig.suptitle(plotname)
data_graph("gradnorm","Norm of gradient vs iteration")
data_graph("relerr","Relative error vs iteration")
data_graph("err","Error vs iteration")
data_graph("stepsize","Step size vs iteration")
fig, axs = plt.subplots(3, 3)
for i, stepname in enumerate(stepsizes.keys()):
    for j, stopname in enumerate(stopconds.keys()):
        alg = (stepname, stopname)
        y = plotdata[alg]["err"][1:]
        x = plotdata[alg]["err"][:-1]
        axs[i,j].scatter(x,y)
        if j == 0:
            axs[i,j].set(ylabel=stepname)
        if i == 2:
            axs[i,j].set(xlabel=stopname)
fig.suptitle("Error vs Prev error")
```

3.1

```
def stepsize_exactline(dk, xk):
    # Golden Search
    x1, x2 = xk.flatten()
    d1, d2 = dk.flatten()
    phi = lambda alpha: bf(x1 + alpha * d1, x2 + alpha * d2)
    gr = (math.sqrt(5)-1)/2
    k = 0
    a = 0
    b = 4
    while b - a > tol and k < max_iter:
        l = b - (b-a) * gr
        r = a + (b-a) * gr
        if phi(l) < phi(r):
            b = r
        else:
            a = l
        k += 1
    return a
```

4.1

```
def generate_problem(m,n):
    rand = np.random.default_rng(seed=1)
    return rand.normal(size=(m,n)), np.zeros((n,))[np.newaxis].T
```

```
def sum_func(n, f):
    acc = 0
    for i in range(n):
        acc += f(i)
    return acc

def gen_f(A):
    m, n = A.shape
    return lambda xk: -sum_func(m, lambda j: math.log(1 \
    - (A[j,:] @ xk).item())) \
                      -sum_func(n, lambda i: math.log(1 \
                      + xk.flatten()[i])) \
                      -sum_func(n, lambda i: math.log(1 \
                      - xk.flatten()[i]))

def gen_df(A):
    m, n = A.shape
    return lambda xk: sum_func(m, lambda j: A[j,:] \
    / (1 - A[j,:] @ xk))[np.newaxis].T \
                      -np.vectorize(lambda xi: 1 / (1 + xi))(xk) \
                      +np.vectorize(lambda xi: 1 / (1 - xi))(xk)

def gen_hf(A):
    m, n = A.shape
    return lambda xk: sum_func(m, lambda j: A[j,:][np.newaxis].T \
    @ A[j,:][np.newaxis] \
    / ((1 - A[j,:] @ xk) ** 2)) \
                     +np.diag(np.vectorize(lambda xi: 1 \
                     / ((1 + xi) ** 2))(xk).flatten()) \
                     +np.diag(np.vectorize(lambda xi: 1 \
                     / ((1 - xi) ** 2))(xk).flatten())


def gen_stepsize(A, sigma, beta, f, df):
    def helper(dk, xk):
        s = 1
        alpha = s
        while np.any(A @ (xk + alpha * dk) >= 1) or \
        np.any(np.abs(xk + alpha * dk) >= 1) or \
        f(xk + alpha * dk) - f(xk) >= sigma * alpha * df(xk).T @ dk:
            alpha *= beta
        return alpha
    return helper
```

4.2

```
colors = ['r', 'g', 'b', 'y']
def data_graph(plotname, sigma):
    fig, axs = plt.subplots(4,4, figsize=(10,10))
    for i, m in enumerate(ms):
        for j, n in enumerate(ns):
            # select xstar as average over all values of this
            xstar = 0
            for beta in betas:
                for sigma in sigmas:
                    alg = (sigma, beta, m, n)
                    xstar += plotdata_steepest[alg][-1]
            xstar /= 16
            for k, beta in enumerate(betas):
                alg = (sigma, beta, m, n)
                y = abs(np.array(plotdata_steepest[alg]) - xstar)
                x = range(len(plotdata_steepest[alg]))
                axs[i,j].scatter(x, y, c=colors[k],
                label=f"beta={beta}",
                s=plt.rcParams['lines.markersize'] ** 2 * 2,
                marker = '.', alpha=0.3)
            if j == 0:
                axs[i,j].set(ylabel=f"m={m}")
            if i == 3:
                axs[i,j].set(xlabel=f"n={n}")
            axs[i,j].set_yscale("log")
    handles, labels = fig.gca().get_legend_handles_labels()
    by_label = dict(zip(labels, handles))
    fig.legend(by_label.values(), by_label.keys(), loc="upper right")
    fig.suptitle(plotname)
for sigma in sigmas:
    data_graph(f"Error vs iteration, sigma={sigma}", sigma)
```

5.1

```
def gen_f(A):
    m, n = A.shape
    return lambda xk: -np.sum(np.log(1 - (A @ xk))) \
                      -np.sum(np.log(1 + xk)) \
                      -np.sum(np.log(1 - xk))

def gen_df(A):
    m, n = A.shape
    return lambda xk: np.sum(A / (1 - A @ xk), axis=0)[np.newaxis].T \
```

```
                    -1 / (1 + xk) \
                    +1 / (1 - xk)

def gen_hf(A):
    m, n = A.shape
    def hf(xk):
        # The sum of outer products of rows is
        # equivalent to matrix multiplication
        # (i.e, the sum of the outer product
        # of column i of A and row i of B)
        # Therefore, A^TA can be written as
        # the sum of the outer products of
        # its rows with themselves

        # The numerator of the fractional
        # part of the sum in the Hessian
        # formula is exactly the outer
        # product of the rows of A. However,
        # the division makes this a bit more complicated.

        # It is exactly this
        # division that makes the hessian function
        # so slow. If we can optimize
        # this to one matrix-wide division
        # and a matrix multiplication, it
        # should be must faster.

        # If we take a copy of A, name
        # it B and divide it each row j
        # by (1-A[j] @ xk), then perform
        # B^TA, that should give the same output.

        # Get the denominators into a vector
        denom = 1 - A @ xk
        # Divide each row of A by the denominators.
        # Because of numpy behaviour, dividing an m x n
        # matrix by an m x 1 vector has the
        # effect of dividing each row in the matrix
        # by the corresponding vector element
        B = A.copy() / denom
        # Perform the sum of outer products
        # by converting to the corresponding
        # matrix multiply
        return B.T @ A
    return hf
```

6.1

```
plotdata_conjugate = {}
x0=np.zeros((A.shape[1],))

for stepname, stepsize in tqdm(stepsizes.items()):
    for stopname, stopcond in tqdm(stopconds.items()):
        xk = x0
        curalg = (stepname, stopname)
        plotdata_conjugate.update({curalg: {"gradnorm": [],
                                            "relerr": [],
                                            "err": [],
                                            "stepsize": [],
                                            "k": 0}})
        grad = df(xk)
        dk = -grad
        while True:
            step = stepsize(dk, xk)
            plotdata_conjugate[curalg]["gradnorm"].append(np.linalg.norm(grad))
            plotdata_conjugate[curalg]["relerr"].append(
                np.linalg.norm(xstar - xk, ord=2)/ \
                np.linalg.norm(xstar, ord=2))
            plotdata_conjugate[curalg]["err"].append(abs(f(xstar) - f(xk)))
            plotdata_conjugate[curalg]["stepsize"].append(step)
            if stopcond(xk):
                break
            plotdata_conjugate[curalg]["k"] += 1
            xk = xk + step * dk
            grad = df(xk)
            if np.linalg.norm(grad,ord=2)<=10e-8:
                break
            beta = (grad.T @ dk)/(dk.T @ dk)
            dk = -grad + beta * dk
```

6.2

```
rand = np.random.default_rng(seed=1)
A = rand.normal(size=(m,n))
b = rand.normal(size=(m,))
```

6.3

```
oldgrad = grad # Before updating grad.
beta = (grad.T @ dk)/(oldgrad.T @ oldgrad) # Polak-Ribiere
```

```
beta = (grad.T @ grad)/(oldgrad.T @ oldgrad) # Fletcher-Reeves
```

7.1

```
plotdata_quasinewton = {}
x0=np.zeros((A.shape[1],))

formulas = {
    "DFP": lambda pk, qk, Vk: Vk + (pk @ pk.T)/(pk.T @ qk) - \
    (Vk @ qk @ qk.T @ Vk)/(qk.T @ Vk @ qk),
    "BFGS": lambda pk, qk, Vk: Vk + (1 + (qk.T @ Vk @ qk)/(pk.T @ qk)) *\
    (pk @ pk.T)/(pk.T @ qk) - (pk @ qk.T @ Vk + Vk @ qk @ pk.T)/(pk.T @ qk)
}

V0s = {
    "Identity": np.identity(n),
    "Inverse-Hessian": np.linalg.inv(A.T @ A)
}

for stepname, stepsize in tqdm(stepsizes.items()):
    for stopname, stopcond in tqdm(stopconds.items()):
        curalg = (stepname, stopname)
        plotdata_quasinewton.update({curalg: {}})
        for formname, formula in tqdm(formulas.items()):
            for vname, V0 in tqdm(V0s.items()):
                curquasi = (formname, vname)
                plotdata_quasinewton[curalg].update({curquasi: {"gradnorm": [],
                                    "relerr": [],
                                    "err": [],
                                    "stepsize": [],
                                    "k": 0}})
                Vk = V0
                xk = x0
                grad = df(xk)
                while True:
                    dk = -Vk @ grad
                    step = stepsize(dk, xk)
                    plotdata_quasinewton[curalg][curquasi]["gradnorm"].append(
                    np.linalg.norm(grad))
                    plotdata_quasinewton[curalg][curquasi]["relerr"].append(
                        np.linalg.norm(xstar - xk, ord=2)/ \
                        np.linalg.norm(xstar, ord=2))
                    plotdata_quasinewton[curalg][curquasi]["err"].append(
                    abs(f(xstar) - f(xk)))
```

```
                    plotdata_quasinewton[curalg][curquasi]["stepsize"].append(
                    step)
                    if stopcond(xk):
                        break
                    plotdata_quasinewton[curalg][curquasi]["k"] += 1
                    oldxk = xk
                    xk = xk + step * dk
                    oldgrad = grad
                    grad = df(xk)
                    pk = xk - oldxk
                    qk = grad - oldgrad
                    Vk = formula(pk[np.newaxis].T, qk[np.newaxis].T, Vk)

colormap = {("DFP","Identity"): 'b', ("DFP","Inverse-Hessian"): 'r',
            ("BFGS","Identity"): 'y', ("BFGS","Inverse-Hessian"): 'g'}
for formname in formulas.keys():
    for vname in V0s.keys():
        fig, axs = plt.subplots(3, 3)
        for i, stepname in enumerate(stepsizes.keys()):
            for j, stopname in enumerate(stopconds.keys()):
                alg = (stepname, stopname)
                quasi = (formname, vname)
                y = plotdata_quasinewton[alg][quasi]["relerr"]
                x = range(plotdata_quasinewton[alg][quasi]["k"]+1)
                axs[i,j].scatter(x,y)
                if j == 0:
                    axs[i,j].set(ylabel=stepname)
                if i == 2:
                    axs[i,j].set(xlabel=stopname)
        fig.suptitle("Relative-error-vs-iteration-(" +
        formname + ",-" + vname +")")
```