19CS10039
Kaushal Bauthuka
(1)

4. (a) $x^{(k)} \approx x^{(k+1)}$ as $k \to \infty$

The equation becomes

$A^T(Ax^{(k)} - b) = 0$

$\Rightarrow A^T A x^{(k)} = A^T b$ (Normal equation)

Thus, $x^{(k)} = \hat{x}$

The given equation resembles gradient descent in such a way that,

minimize $\|Ax - b\|_2^2$ $\hookrightarrow (Ax-b)^T(Ax-b)$

Now $\frac{\partial}{\partial x}(\|Ax-b\|_2^2) = 2A^T(Ax-b)$

Thus, it can be further written that,

$x^{(k+1)} = x^{(k)} - \alpha\, 2A^T(Ax-b)$ (Updation step)

Let $\alpha = \frac{1}{2\|A\|^2}$

$\therefore x^{(k+1)} = x^{(k)} - \frac{2A^T}{2\|A\|^2}(Ax-b)$

$\Rightarrow x^{(k+1)} = x^{(k)} - \frac{A^T(Ax-b)}{\|A\|^2}$ $\Rightarrow$ This is the given equation

(b) Time complexity of multiplication Ax = $O(mn)$ time

Time complexity of subtraction $Ax - b$ = $O(m)$ time

Time complexity of multiplication $A^T(Ax-b)$ = $O(mn)$ time

Time complexity of division $\frac{A^T(Ax-b)}{\|A\|^2}$ = $O(m)$ time

Time Complexity of subtraction $x^{(k)} - \frac{A^T(Ax-b)}{\|A\|^2}$ = $O(m)$ time

$\therefore$ Total time complexity = $O(mn)$ time.

Since this is run for k steps, Time complexity = $O(mnk)$ time

(c) The direct method consists of using the QR decomposition. It involves an expensive step that has a time complexity of $O(mn^2)$ time

If n (number of features) is large, then this is very slow.

Thus, the iterative method is beneficial, since we can control the number of iterations (k). Also, we can cause early stopping if increment in consecutive values of $x^{(k)}$ is negligible.

In [1]:
```
# Kaushal Banthia
# 19CS10039
# Question 7
```

In [2]:
```
import numpy as np
```

In [3]:
```
def algorithm(A, b, iter):
    x = np.zeros((10, 1))
    actual = np.linalg.inv(A.transpose() @ A) @ A.transpose() @ b
    norm = np.linalg.norm(A, 2)
    for i in range(iter):
        temp = x - ((A.transpose() @ ((A @ x) - b)) / np.square(norm))
        x = temp
    print("Actual x is: " + str(actual))
    print("Iterative x is: " + str(x))
    print("2 norm of difference after " + str(iter) + " iterations is: " + str(np.li
```

In [4]:
```
A = np.random.rand(30,10)
b = np.random.rand(30,1)
rank = np.linalg.matrix_rank(A)
print("Rank(A): " + str(rank))
if rank==10:
    print("A is full rank")
    algorithm(A, b, 100)
    algorithm(A, b, 1000)
else:
    print("A is not full rank")
```

```
Rank(A): 10
A is full rank
Actual x is: [[ 0.24988416]
 [ 0.19057976]
 [-0.23528952]
 [-0.17585296]
 [-0.06039597]
 [-0.37317122]
 [ 0.54833551]
 [ 0.19659172]
 [ 0.56898956]
 [-0.01360586]]
Iterative x is: [[ 0.25728869]
 [ 0.14997773]
 [-0.18175885]
 [-0.1028233 ]
 [ 0.00763066]
 [-0.33004629]
 [ 0.43742638]
 [ 0.19869833]
 [ 0.53365844]
 [-0.02306954]]
2 norm of difference after 100 iterations is: 0.17329950658665813
Actual x is: [[ 0.24988416]
 [ 0.19057976]
 [-0.23528952]
 [-0.17585296]
 [-0.06039597]
 [-0.37317122]
 [ 0.54833551]
 [ 0.19659172]
 [ 0.56898956]
 [-0.01360586]]
```

```
Iterative x is: [[ 0.24987738]
 [ 0.19056965]
 [-0.23528909]
 [-0.17583652]
 [-0.06038942]
 [-0.37317683]
 [ 0.5483312 ]
 [ 0.19659991]
 [ 0.56898932]
 [-0.01360651]]
2 norm of difference after 1000 iterations is: 2.4060953424038222e-05
```

In [5]:

```
# As it can be seen from the above code's output, the values of from both iterative
# If we increase the number of iterations from 100 to 1000, then they come even more
# difference reducing even further. This numerically verifies that the algorithm con
```