(1) Gradient descent is a foundational optimization technique used in machine learning to minimize a cost function iteratively.
The three gradient descent which is explored in the assignment are as:

Vanilla Gradient Descent: It is the simplest form of gradient descent
ii) It computes the gradient of ~~each~~ cost function w.r.t all ~~each~~ data points in each iteration
iii) The update rule for parameter (weights) θ is given by.

$$\theta\_new = \theta\_old - learningrate \times \nabla J(\theta\_old)$$

where $\nabla J(\theta\_old)$ is the gradient of cost function with respect to θ_old.

Mini-Batch Gradient decent:

i) This is a compromise between Vanila and Stochastic Gradient decent.
ii) It divides dataset into small batch (mini-batches) & computes gradient using one mini batch at a time.
iii) Update rule is similar to

Vanila Gradient descent, but its sums for the gradients of the mini-batch.

$$\theta\_new = \theta\_old - \text{Learning rate} \times (1/\text{batch\_size}) \times \Sigma(\nabla J(\theta\_old) \text{ for data in mini-batch})$$

Stochastic Gradient Descent (SGD):

→ In stochastic gradient descent the gradient is computed and parameters are updated for each individual data point in the dataset.

→ The update rule is similar to Vanila Gradient descent but it uses a single data point at a time:

$$\theta\_new = \theta.old - \text{learning rate} \times \nabla J(\theta.old) \text{ for a randomly selected data point.}$$

→ Observations from Graph:

1 a) Vanilla Gradient descent converges more smoothly.

b) Vanilla gradient descent looks computationally more intensive as it uses whole dataset, so we can see cost is very high at the start.

(2) a) Stochastic Gradient descent exhibits faster convergence.

b) SGD's path looks more erratic as introduces significant noise into parameter updates.

c) Its less computationally intensive ~~& often~~ as it processes one data point at a time.

(3) a) Mini-Batch Gradient descent looks like a mid-path as far as convergence is concerned.

b) In case of mini batch path is less erratic as it introduces controlled noise.

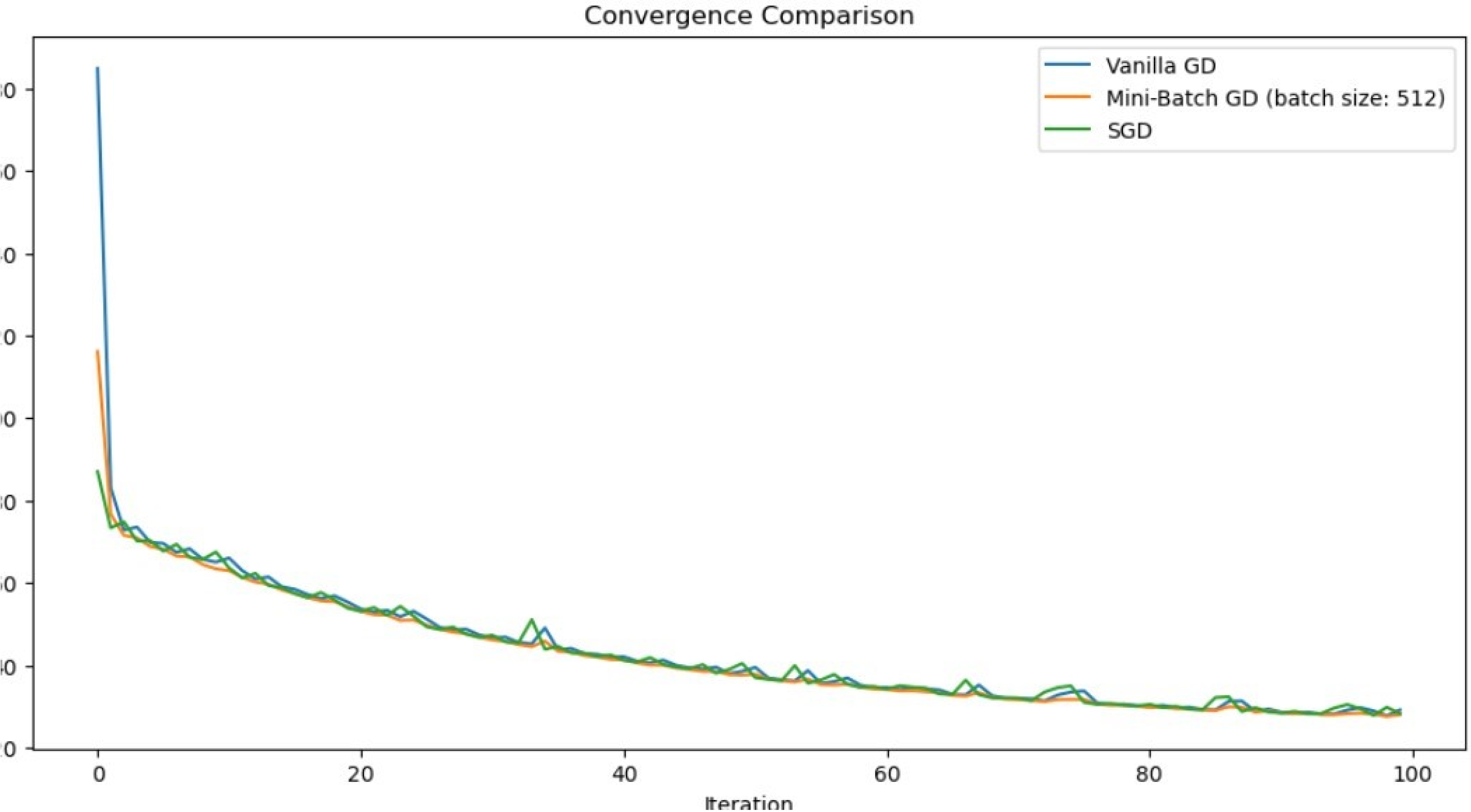
c) Its versatile choice as it balances the trade off between ~~faster~~ convergence ~~and~~ smoothness and ~~faster~~ computational efficiency.

Overall observation:

SGD is useful when computational resources are limited and faster convergence is needed ~~with~~ even with noise.

Vanila works best when computational resources are abundant & smooth convergence path is needed.

Mini-Batch Gradient descent is a versatile choice that balances the trade off b/w SGD & Vanilla.



Convergence Comparison
Vanilla GD
Mini-Batch GD (batch size: 512)
SGD
0
20
40
60
80
100
Iteration

Experiment with mini-batch Gradient descent

a) Load the dataset with at least 1500 rows and 15 columns. (Transportation dataset in exp)

b) Carefully pre-process by removing nulls, missing value & added scaling feature.

c) Split data into training set & validation set to assess performance.

Initialization: Initialize model parameters (θ) and set hyperparameters including learning rate, number of iterations and convergence criteria.: In our case, we have kept no. of iterations = 1000, learning rate = 0.01.

Batch size selection: 1) Define batch size to experiment with. In our case, we have used batch size of 16, 32, 64, 128 and 256.

2) The optimal size depends on a number of factors such as dataset size, hardware resources ~~and~~ and convergence speed etc. Smaller batch size is generally used when fast convergence is a priority while large batch size is needed when

computational efficiency is critical or when dealing with noisy data. (6)

Training loops:

→ Implement the loop for each variant.

→ for each batch size; perform following steps

a) Divide training data into mini-batches of selected ~~data~~ batch size.

b) Iterate thru mini-batches, update parameters using appropriate gradient descent algo.

c) Record & track (plot on graph) the cost (or loss) on the validation set.

Analysis:

→ A graph is plotted ~~for~~ by plotting the cost (or loss) on validation set for each batch size over iterations. We see best batch size is 64 for given experiment with minimum loss.

Conclusion:

) Batch size there is always a trade off between faster convergence & loss.

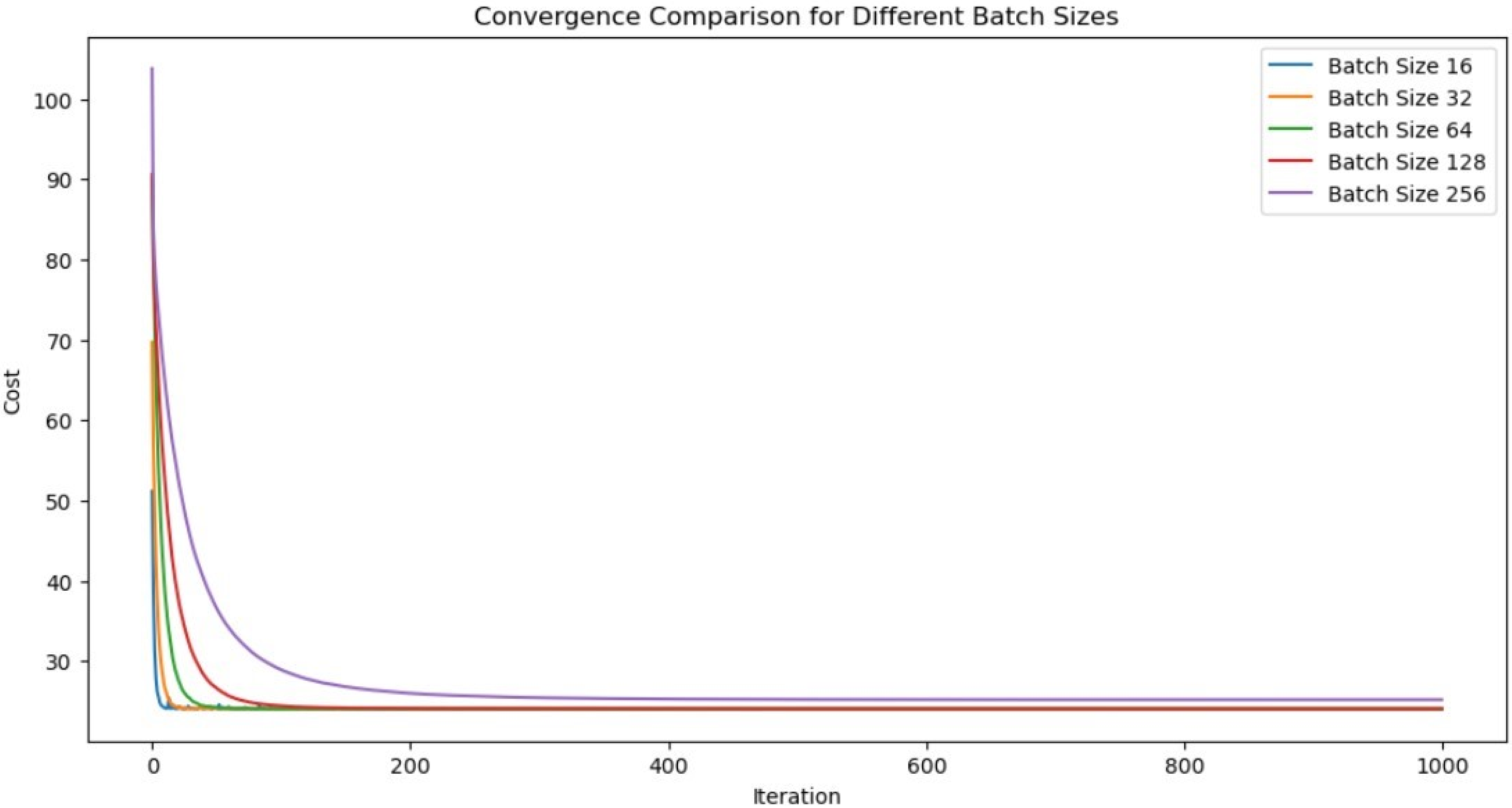
Large batch size → Smoother & stable convergence

Smaller batch size → Faster convergence but can lead to unstable convergence

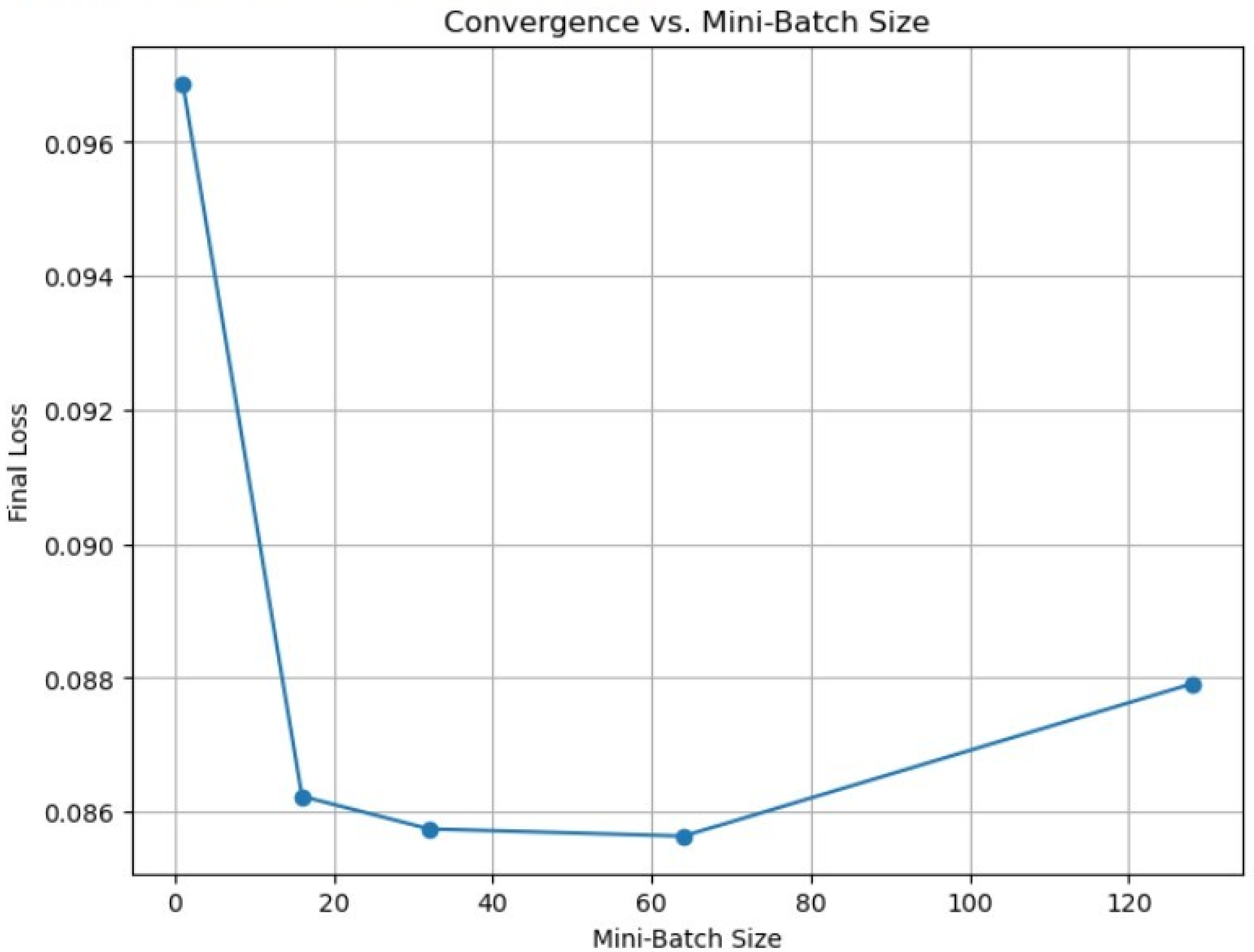
→ A small batch size can introduce more noise but may lead to faster convergence while a large batch size can reduce noise but might converge more slowly.

→ Mini-batch is suitable for a wide range of dataset size, can leverage parallelism for faster training on multi-core processors or GPUs.

(All the graphs are in next pages)


Convergence Comparison for Different Batch Sizes
Batch Size 16
Batch Size 32
Batch Size 64
Batch Size 128
Batch Size 256
Cost
Iteration
100
90
80
70
60
50
40
30
0
200
400
600
800
1000

```
Best Batch Size: 64
Final Loss with Best Batch Size: 0.08563883618900707
```



(2) In optimization line search strategy is one of the two basic iterative approaches to find local minimum of an function. The method starts at a point x_0 and then move in a direction d by a step α. The step length α is chosen so that to minimize $f(x_0 + \alpha d)$.

Bisection Search: This is a simple and robust method that works out by repeatedly bisecting intervals of possible sizes until a step size is found ~~gap~~ ~~the condition~~ which satisfies the condition. Here $c = (a+b)/2$

Golden Search: This is more efficient variant of bisection search that uses golden ratio instead of ~~mid point~~ bi-section to find the suitable step size.

Here mid point is found such that

$$\frac{b}{a} = \psi$$

when $\psi = \frac{1+\sqrt{5}}{2} = 1.6180$

You start with interval $[a, b]$ and calculate two points

$$x_1 = a + (1-\psi)(b-a)$$

$$x_2 = a + \psi(b-a)$$

Evaluate function at x_1 & x_2 & compare function values:

Depending on function value, update the interval & continue until its sufficiently small.

<u>Armijo rule</u>: This is more sophisticated method that uses the gradient of the objective function to ~~find the~~ guide the search for a suitable size. Its based on the idea of making sure that the function decreases sufficiently during each iteration.

→ Rule involves a parameter typically a small number between (0 & 1) and a decrease factor.

→ At each iteration, check if new fun value is less than ~~previous one~~ or equal to current value minima a certain prop. of gradient times the

step size.

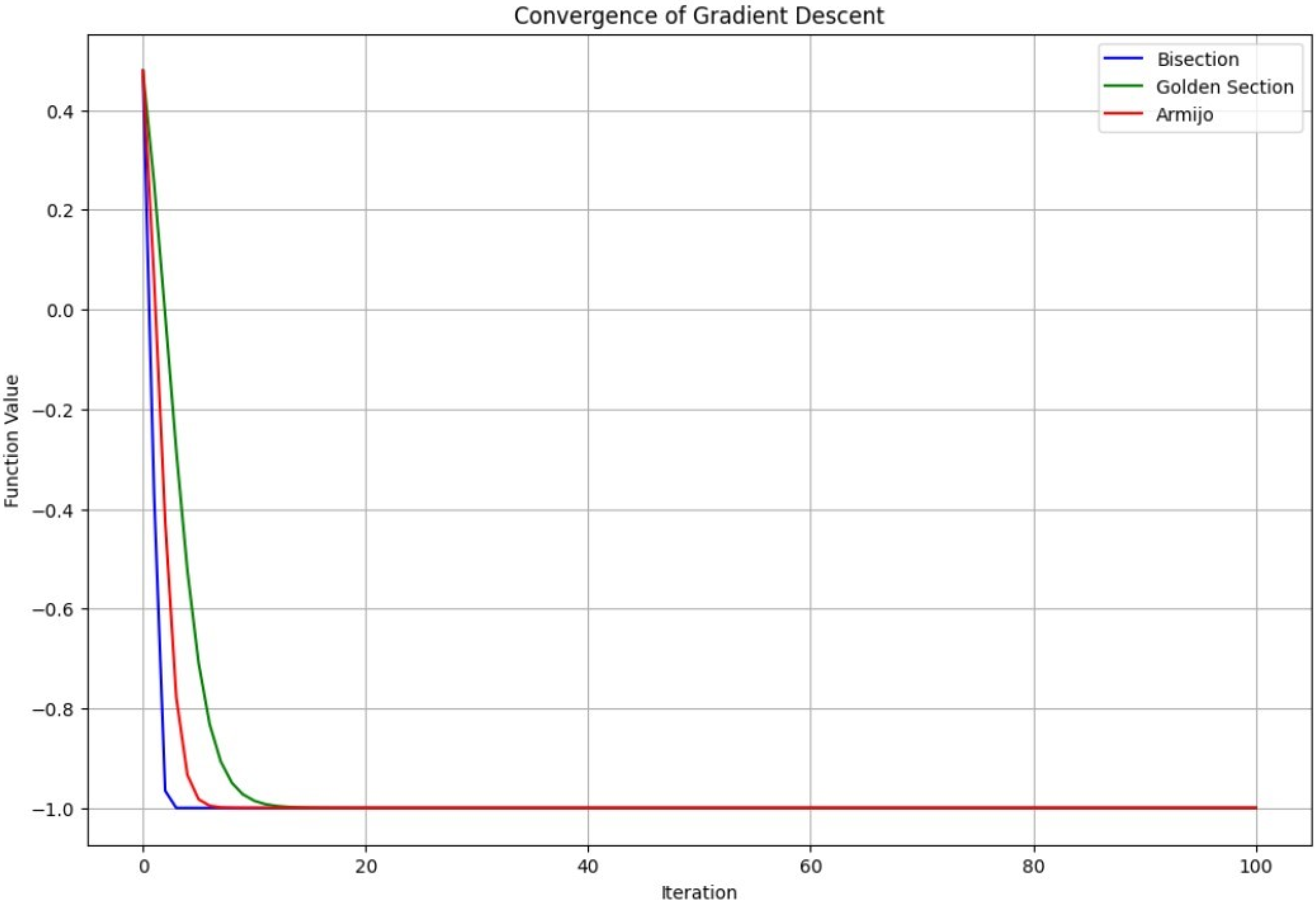
⇒ If condition is met, step size is accepted. Otherwise its reduced and the check is repeated until the condition is satisfied.

These line searches are essential tools in optimization algorithms for finding optimal solⁿ. Choice of ~~ab~~ search depends on problem statement.

Bisection & Golden search are suitable for one dimensional & Armijo is more suitable for multi-dimensional optimization setting.

In this case, Bisection works the best, then Armijo & last is golden search.

Method	Adv.	Disadv.
Bisection Search.	Simple, Robust	Slow for large prob.
Golden.	More effi. than bisecti (generally)	Less robust than bisectn
Armijo	Most sophisticated & works best for complicated problem.	May not converge & expensive computationally



Convergence of Gradient Descent
Bisection
Golden Section
Armijo
Function Value
Iteration
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
0
20
40
60
80
100

Function $f(x) = (x^2 \cos(x) + \sin(x) - x)$

Gradient descent is a first order iterative optimization algorithm for finding a local minimum of a differentiable function.

Its the learning rate which determines how much we adjust our current position which greatly affects the convergance of algorithm.

$$f(x) = x^2 \cos(x) + \sin(x) - x.$$

$$f'(x) = 2x \cos(x) - x^2 \sin(x) + \cos(x) - 1$$

Constant Step Size: In this case, we simply use a fixed step throughout the optimization process. This is simple but it leads to slow convergance or overshooting the minima.

Mathematically ~~x_{old}=~~

$$x_{new} = x_{old} - \gamma f'(x_{old}).$$

Choice of γ is crucial here. If its too large, it will overshoot the minimum and diverge. If its too ~~large~~ small, it will converge too slowly.

Decaying step size: In this case, we start with an initial step size α_0 and decrease it over time using a schedule like $\alpha_K = \alpha_0 / K$ where K is positive constant

A common strategy of decaying is

$$\gamma_t = \frac{\alpha_0}{1+Kt}$$

where $\gamma_t \rightarrow$ step size at iteration t

$\alpha_0 \rightarrow$ initial step size

$K \rightarrow$ decay constant

Choice of α_0 and K will determine how quickly step size decays. A smaller K will result in smaller decay while large K will result in faster decay.

Bold driver Algorithm: The bold driver algorithm dynamically adjusts the size of step if progress is good and decreases if its not. Exact rule of step size can vary. One of the way is to multiply step size by a factor greater than 1 after a successful iteration & a factor less than 1 after unsuccessful run method

Conclusion : The best choice of step size will vary depend on specific characteristi of function and initial condition. Its often good to use diff. methods and step size to see what works best.

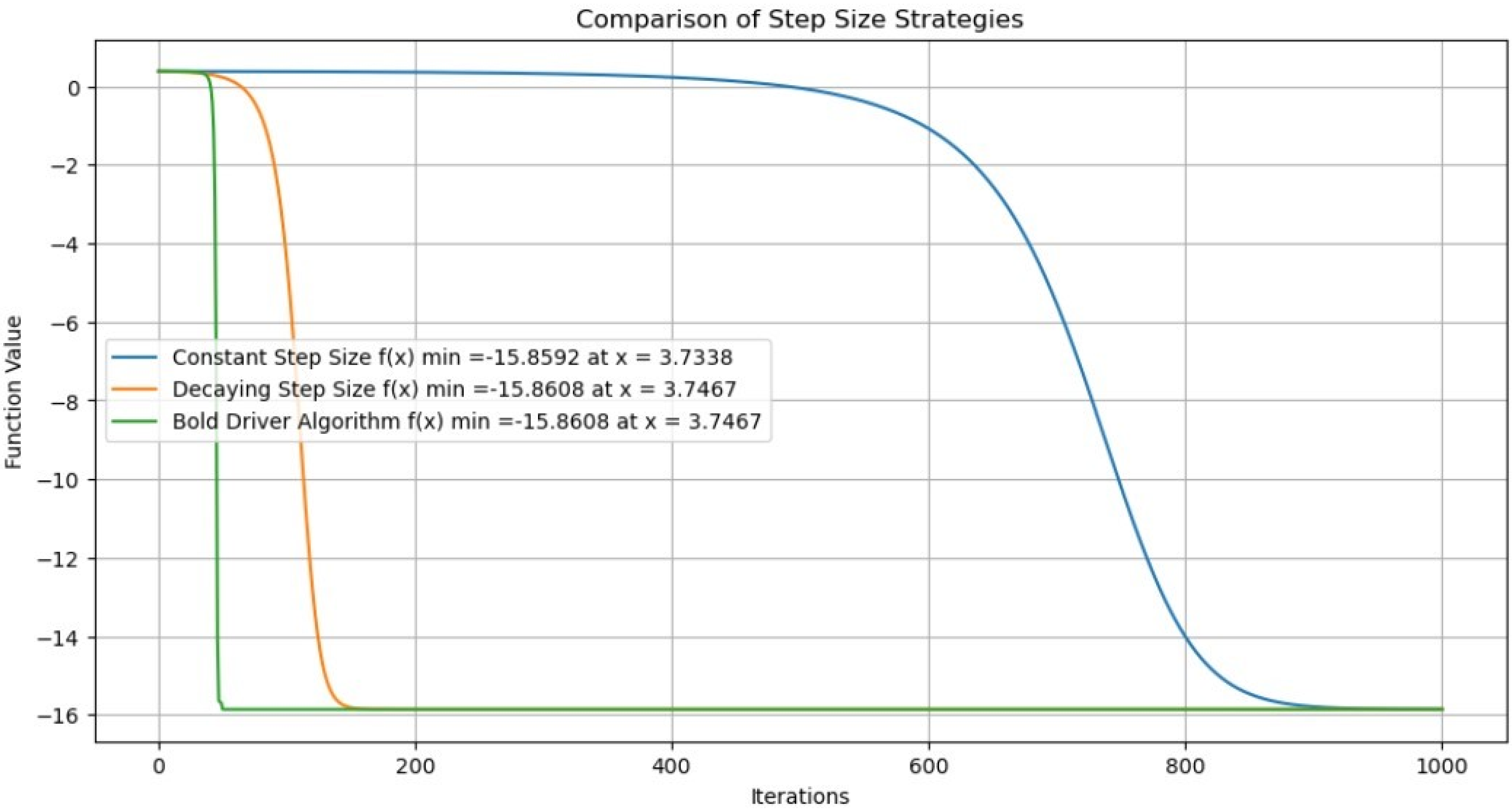
Overall:

Constant Step Size: Its simplest approach but might not be most efficient. Here, convergance depends on gamma (γ) & it doesn't adapt to function.

Decaying step Size: It can be more efficent as it allows for larger step initially and then smaller steps later on. However choice of α & k is crucial.

Bold driver: Its adaptive approach & here step size varies based on function's behaviour. Its far more sensetive to initial step size & choice of increase / decrease factors

Clearly in this example bold driver's algorithm worked best.



Comparison of Step Size Strategies
Constant Step Size f(x) min =-15.8592 at x = 3.7338
Decaying Step Size f(x) min =-15.8608 at x = 3.7467
Bold Driver Algorithm f(x) min =-15.8608 at x = 3.7467
Function Value
Iterations
0
−2
−4
−6
−8
−10
−12
−14
−16
0
200
400
600
800
1000