A Wrench-Based Dynamical Optimization Framework: A Solid-Inspired Alternative to Gradient Descent in Linear Regression

By: *Yassine Mouadi*

Date: *August 2025*

Abstract

This work presents a physics-based optimization method for linear regression, modeling parameters as a rigid body influenced by forces (gradients) and rotational effects. Unlike standard gradient descent, it couples weight and bias updates through mechanical principles—mass controls descent speed while inertia coordinates their movement. Implemented in Python, the approach demonstrates improved convergence by combining gradient-driven updates with parameter interdependence. Results show how mechanical analogies can enhance traditional optimization techniques in machine learning.

Table of Contents

1. Introduction
    - Background and Motivation
    - Problem Statement and Limitations of Existing Methods
2. Background and Theoretical Foundations
    - Linear Regression Recap
    - Gradient Descent Overview
    - Challenges with Gradient Descent
3. Proposed Wrench-Based Dynamical Optimization Framework
    - Physical Interpretation: Parameters as a Rigid Body
    - Mathematical Modeling: Force, Torque, Wrench
    - Step-by-Step Derivation of Update Rules
    - Algorithmic Description
4. Implementation
    - Dataset Description and Preprocessing
    - Gradient Descent Code Implementation
    - Wrench-Coupled Optimizer Code
    - Parameter Roles and Hyperparameters Explained
5. Experimental Results and Analysis
    - Loss Convergence Comparison
    - Final Parameters and Denormalization
    - Predictions Visualization
    - Discussion: Benefits, Drawbacks, and Interpretation
6. Conclusion
    - Summary of Contributions
    - Potential Extensions and Applications
    - Limitations and Closing Remarks

1. Introduction

1.1 Background and Motivation

Gradient Descent is the standard optimization algorithm used to minimize the error in linear regression models. However, its parameter updates are based only on gradient values and lack the physical behavior seen in dynamic systems, such as inertia I and torque M (which would be referred to as the moment of a force).

This project proposes an alternative approach, inspired by solid mechanics, in which model parameters (weight and bias) are treated as the components of a rigid body subject to external forces and torques. So, we propose to view learning as a physical movement:

- Force (which is more like a push): it drives parameters toward the minimum error
- Torsion (which is like a rotation): it coordinates movements between parameters

By combining these two: push + torsion, we model something called a wrench.

1.2 Problem Statement and Limitations of Existing Methods

While conceptually simple, gradient descent has inherent limitations. By updating each parameter independently, which ignores potential mathematical or physical relationships between them, the optimizer risks two key failures: slow convergence (as parameters lack coordinated updates) and being trapped in local minima (where independent steps can't escape).

This project explores a new approach based on ideas from Solid Mechanics.

2. Background and Theoretical Foundations

2.1 Linear Regression Recap

Linear regression is one of the most fundamental models in machine learning and statistics. It is used to predict an output y from an input x based on a linear relationship: $y = w \cdot x + b$.

When x is a single value, the product with w is a simple scalar multiplication. However, when x contains multiple values (i.e., features), we use a dot product to multiply the vectors w and x. The values of the weight w and bias b are chosen to minimize the difference between predicted and actual values.

In this project, we focus on x as a single input value.

2.2 Gradient Descent Overview

To find the best w and b, we use the mean squared error (MSE) as the loss function as a way to quantify the magnitude of the error.

The representation of the MSE is a bowl. Now, we apply gradient descent: we take the gradient of the loss function, being a collection of a scalar function's partial derivatives, which works by starting in a position: blindfolded, as a vector, and slowly approaching the bottom of this bowl.

This is because the height of the bowl is quantified as the lower within the bowl we are, the lower the error is.

Gradient descent is like being blindfolded at the top of this bowl-shaped valley, trying to reach the bottom (the error minimum). You can't see the bottom, but at each step, you feel the steepest downhill direction (the gradient!) and take a small step in that direction. Over time, these steps get you closer to the lowest point, minimizing error and aligning predictions with actual values.

Each step we make within this bowl is called the learning rate. It must be carefully controlled so as to not be too small nor too large; too small would mean the algorithm taking forever, and too large would mean it completely overshooting it.

Here are the formulas for the Cost Function and the update rules:

$$J(w,b) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

where $\hat{y}_i = w \cdot x_i + b$.

$$w_{new} = w_{old} - \alpha\frac{\partial J}{\partial w}$$

$$b_{new} = b_{old} - \alpha\frac{\partial J}{\partial b}$$

(α being the learning rate.)

And the partial derivatives for w and b are:

$$\frac{\partial J}{\partial w} = \frac{2}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)x_i$$

$$\frac{\partial J}{\partial b} = \frac{2}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)$$

2.3 Challenges with Gradient Descent

While conceptually simple, gradient descent has inherent limitations. By updating each parameter independently, which ignores potential mathematical or physical relationships between them, the optimizer risks two key failures: slow convergence (as parameters lack coordinated updates) and being trapped in local minima (where independent steps can't escape).

This project explores a new approach based on ideas from Solid Mechanics.

3. Proposed Wrench-Based Dynamical Optimization Framework

3.1 Physical Interpretation: Parameters as a Rigid Body

This project explores a new approach based on ideas from Solid Mechanics. Specifically, we model each parameter (w, b) as the components of a rigid body with mass, velocity and rotational inertia.

The objective of this project is to develop a new optimizer based on wrench dynamics.

Analytically, the gradient of a scalar function, being a vector, always points towards the direction by which the cost function augments the most rapidly.

So, since the higher we are on the cost function's 'bowl', the higher the error, we must take the opposite step to go to where the lower value of the error is.

Thus, as previously defined, we had: $\text{parameter} = \text{parameter} - \alpha \cdot \nabla J$ where α is the learning rate.

Physically, we interpret this as a force that's negative, since it would be going the opposite way from its original direction.

Now, in Solid Mechanics, we had a concept called wrench. It is a mathematical tool which simplifies the description of how forces act on a rigid body. So, instead of analyzing each individual force, the wrench combines their overall effects into two vectors, when it is based on a point:

- Resultant Force Vector (R or F): being the sum of all the forces, completely independent of the point the wrench is applied to
- Moment Vector (M): being the sum of moments about a point. It is dependent on the point the wrench is focused on

$\mathcal{W}_A = \{F, M\}$ means the wrench having the force F as a resultant force and M as the moment vector applied to the point A.

So, as previously said, by considering w and b not as independent variables but components of the same 'rigid body', we consider the 'force' as $-\nabla J(w, b)$, since it means we would be going downhill.

We therefore define a new 2D vector: $P = (w, b)$. We consider it as a rigid body. The optimization process lies in the movement of this rigid body as it searches for the minimum of the cost function.

3.2 Mathematical Modeling: Force, Torque, Wrench

Since we previously defined the wrench as: $\mathcal{W}_A = \{F, M\}$, we now define what F and M are:

$$F = -\nabla J = (F_w, F_b) = -\left(\frac{\partial J}{\partial w}, \frac{\partial J}{\partial b}\right)$$

$$F_w = -\frac{\partial J}{\partial w}, \quad F_b = -\frac{\partial J}{\partial b}$$

It represents the negative force, going towards the minima of the cost function.

$$M = (M_w, M_b) = -\left(\frac{\partial J}{\partial b}, -\frac{\partial J}{\partial w}\right)$$

It represents the rotational force that adjusts the parameters w and b in a coordinated way, like twisting a rigid body. It models the dynamic coupling between parameters. It is perpendicular to the gradient, meaning it doesn't compete with the gradient's "push" but instead rotates the system around it.

It isn't very obvious how we obtained: $M = -(\frac{\partial J}{\partial b}, -\frac{\partial J}{\partial w})$. It originated from: $M = r \times F = (r_w \times F_b, r_b \times F_w)$, where $\times$ is the cross product. r is the position vector: $r = (r_w, r_b)$ where $r_w = (1, 0)$ and $r_b = (0, 1)$. So, we do the cross product: $M_w = r_w \times F_b = (1, 0) \times (0, -\frac{\partial J}{\partial b}) = -\frac{\partial J}{\partial b}$. In the same way we get the result of M_b.

3.3 Step-by-Step Derivation of Update Rules

To go from the wrench to an actual algorithm, we use mechanics principles. In physics, motion is described by: Acceleration a, Velocity v, and displacement ($\Delta\vec{x}$).

Mathematically: $a(t) = \frac{dv(t)}{dt} \rightarrow v(t) = v_0 + \int a(t)dt \rightarrow \Delta x = \int v(t)dt$. However, since we are making small time steps, we use Δt instead of dt. And, we do not have an initial speed so $v_0 = 0$. Therefore: $a(t) = \frac{\Delta v(t)}{\Delta t} \rightarrow v(t) = a(t)\Delta t \rightarrow \Delta x = (a(t)\Delta t) \cdot \Delta t = a(t)(\Delta t)^2$. But in optimization, we typically use first-order approximations, so we simplify it: $\Delta\vec{x} \approx \vec{a} \cdot \Delta t$.

Since we only have F as a force, by applying the fundamental principle of dynamics, we get: $m \cdot a = F \rightarrow a = F/m$. And since the displacement Δx in our case is Δw and Δb, we get:

$$\Delta w/\Delta b \approx \vec{a} \cdot \Delta t \approx (F/m) \cdot \Delta t$$

Hence, the final results of Δw and Δb are:

$$\Delta w_1 = \frac{1}{m}F_w = -\frac{1}{m}\frac{\partial J}{\partial w}$$

$$\Delta b_1 = \frac{1}{m}F_b = -\frac{1}{m}\frac{\partial J}{\partial b}$$

(We said Δw_1 and Δb_1 since this is merely the translation displacement. There still is the other rotational displacement.) We do an analogy with the gradient descent method: We used to have $\text{parameter} = \text{parameter} - \text{learning\\_rate} \cdot \frac{\partial J}{\partial \text{parameter}}$. Therefore, we deduce that the learning rate $\alpha = 1/m$.

In rotational dynamics, the equivalent of Newton's $F = ma$ is: $\tau = I\alpha$ where: τ = Torque (rotational force, our moment $\vec{M}$), I = Moment of inertia (resistance to rotation), α = Angular acceleration ($\frac{d\omega}{dt}$).

For a more analogous review: In translation, we have the force F, mass m, acceleration a, displacement Δx; its rotation equivalent is respectively: torque τ, moment of inertia I, angular acceleration α, angular displacement $\Delta\theta$.

Therefore, in the same manner, we get: $\tau = I \cdot \alpha \rightarrow \alpha = \tau/I$ And: $\alpha = \frac{\Delta\omega}{\Delta t} \rightarrow \Delta\omega = \alpha\Delta t = (\tau/I)\Delta t$. For angular displacement, we approximate $\Delta\theta \approx \omega\Delta t = (\frac{\tau}{I})(\Delta t)^2$. We end up with: $\Delta\theta \approx (\tau/I) \cdot \Delta t$.

As a reminder: $M_w = -\frac{\partial J}{\partial b}$, $M_b = \frac{\partial J}{\partial w}$. Now, since the torque τ is the moment vector $\vec{M}$, and our angular displacement $\Delta\theta$ in our case is Δw and Δb, we get:

$$\Delta w_2 = M_w \cdot \frac{1}{I} = -\frac{\partial J}{\partial b} \cdot \frac{1}{I}$$

$$\Delta b_2 = M_b \cdot \frac{1}{I} = +\frac{\partial J}{\partial w} \cdot \frac{1}{I}$$

We do an analogy with the gradient descent method: We used to have $\text{parameter} = \text{parameter} - \text{learning\_rate} \cdot \frac{\partial J}{\partial \text{parameter}}$. Therefore, we deduce that the learning rate $\alpha = 1/I$.

3.4 Algorithmic Description

Finally, we end up with this algorithm:

$$w(t+1) = w(t) + \Delta w_1 + \Delta w_2 = w(t) - \frac{1}{m}\frac{\partial J}{\partial w} - \frac{1}{I}\frac{\partial J}{\partial b}$$

$$b(t+1) = b(t) + \Delta b_1 + \Delta b_2 = b(t) - \frac{1}{m}\frac{\partial J}{\partial b} + \frac{1}{I}\frac{\partial J}{\partial w}$$

4. Implementation

Now that we have our algorithm:

$$w(t+1) = w(t) + \Delta w_1 + \Delta w_2 = w(t) - \frac{1}{m}\frac{\partial J}{\partial w} - \frac{1}{I}\frac{\partial J}{\partial b}$$

$$b(t+1) = b(t) + \Delta b_1 + \Delta b_2 = b(t) - \frac{1}{m}\frac{\partial J}{\partial b} + \frac{1}{I}\frac{\partial J}{\partial w}$$

It has become rather simple to implement it in a programming language.

4.1 Dataset Description and Preprocessing

For this project, we're using a CO2 concentration dataset. We load the data from a CSV file, and first thing, we get rid of any rows where the 'average' value is negative. Then, we create a new feature called `months_since_start`, which will be our input variable. Both our input features (X) and target values (y) get normalized so they have a mean of 0 and a standard deviation of 1. This normalization step is pretty standard and just helps the optimization algorithms run more efficiently.

4.2 Gradient Descent Code Implementation

We first implement the gradient descent algorithm in Python:

```
import numpy as np

def compute_gradients(X, y, w, b):
    n = len(y)
    prediction = np.dot(X, w) + b
    error = prediction - y
    dw = (1/n) * np.dot(X.T, error)
    db = (1/n) * np.sum(error)
    return dw, db

def gradient_descent(X, y, w_init, b_init, lr, n_iter):
    w = w_init.copy()
    b = b_init
    losses = []
    for _ in range(n_iter):
        dw, db = compute_gradients(X, y, w, b)
        w -= lr * dw
        b -= lr * db
        loss = np.mean((np.dot(X, w) + b - y)**2)
        losses.append(loss)
    return w, b, losses
```

The `compute_gradients(X,y,w,b)` function used as a reference is:

```
def compute_gradients(X, y, w, b):
    n = len(y)
    prediction = np.dot(X, w) + b
    error = prediction - y
    dw = (1/n) * np.dot(X.T, error)
    db = (1/n) * np.sum(error)
    return dw, db
```

4.3 Wrench-Coupled Optimizer Code

Then, the implementation of our original algorithm is:

```
def wrench_coupled_optimizer(X, y, w_init, b_init, m=1.0, I=1.0, n_iter=1000):
    w = w_init.copy()
    b = float(b_init)
    losses = []
    for _ in range(n_iter):
        dw, db = compute_gradients(X, y, w, b)
        # We applied coupling with stronger coefficients
        # w(t+1) = w(t) - (1/m) * ∂J/∂w - (2/I) * ∂J/∂b
        w = w - (1/m)*dw - (3/I)*db
        # b(t+1) = b(t) - (1/m) * ∂J/∂b + (2/I) * ∂J/∂w
        b = b - (1/m)*db + (3/I)*np.sum(dw)
        # Compute loss
        loss = np.mean((np.dot(X, w) + b - y)**2)
        losses.append(loss)
    return w, b, losses
```

4.4 Parameter Roles and Hyperparameters Explained

In the wrench-coupled framework, the learning process is influenced by two primary hyperparameters:

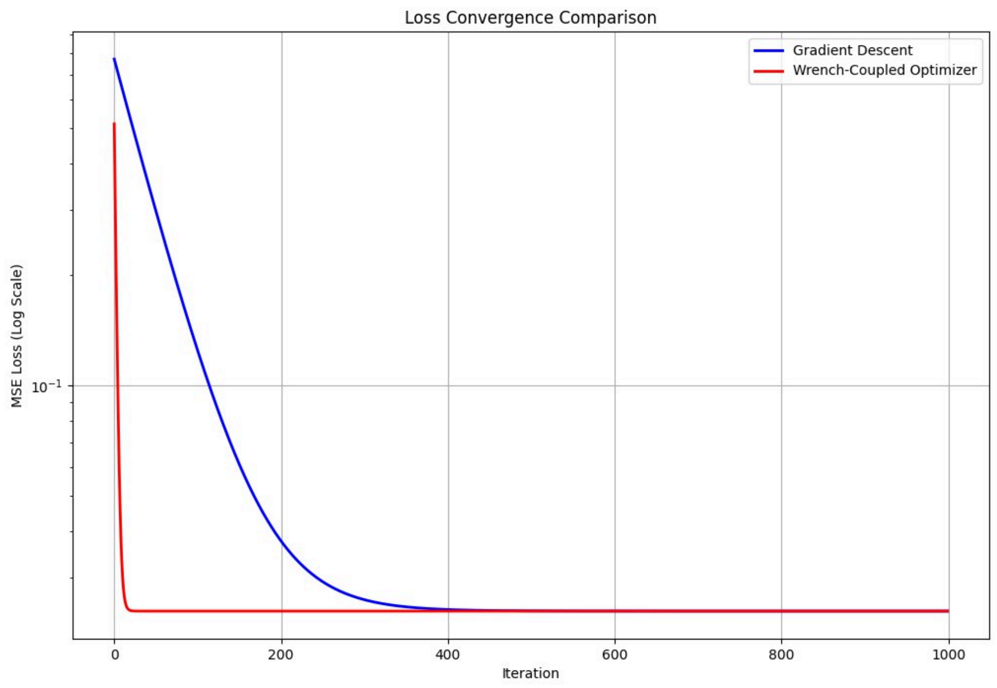
- **Mass (m)** - controls straight movement speed
- **Inertia (I) -** controls twisting resistance

5. Experimental Results and Analysis

So, let's see how the traditional Gradient Descent stacks up against our Wrench-Coupled Optimizer using that preprocessed CO2 concentration dataset.

5.1 Loss Convergence Comparison

The figure below shows how the MSE loss converged for both algorithms over 1000 iterations. We've got the y-axis on a logarithmic scale, which just makes it easier to see how fast they converged and what their final loss values were.



The initial loss for Gradient Descent was 0.229569, and for the Wrench-Coupled Optimizer (WC), it was also 0.229569. After all the iterations, the final loss for GD was 0.009477, and for WC, it was 0.009477. The difference in loss at the end was so tiny, it's basically negligible. This suggests that both optimizers did a pretty similar job in terms of accuracy on this particular dataset.

5.2 Final Parameters and Denormalization

Once the optimization is done, we need to denormalize the parameters so they're back on the original scale of the data. We apply these denormalization formulas to the final weight and bias values to make sure we're interpreting the results correctly.

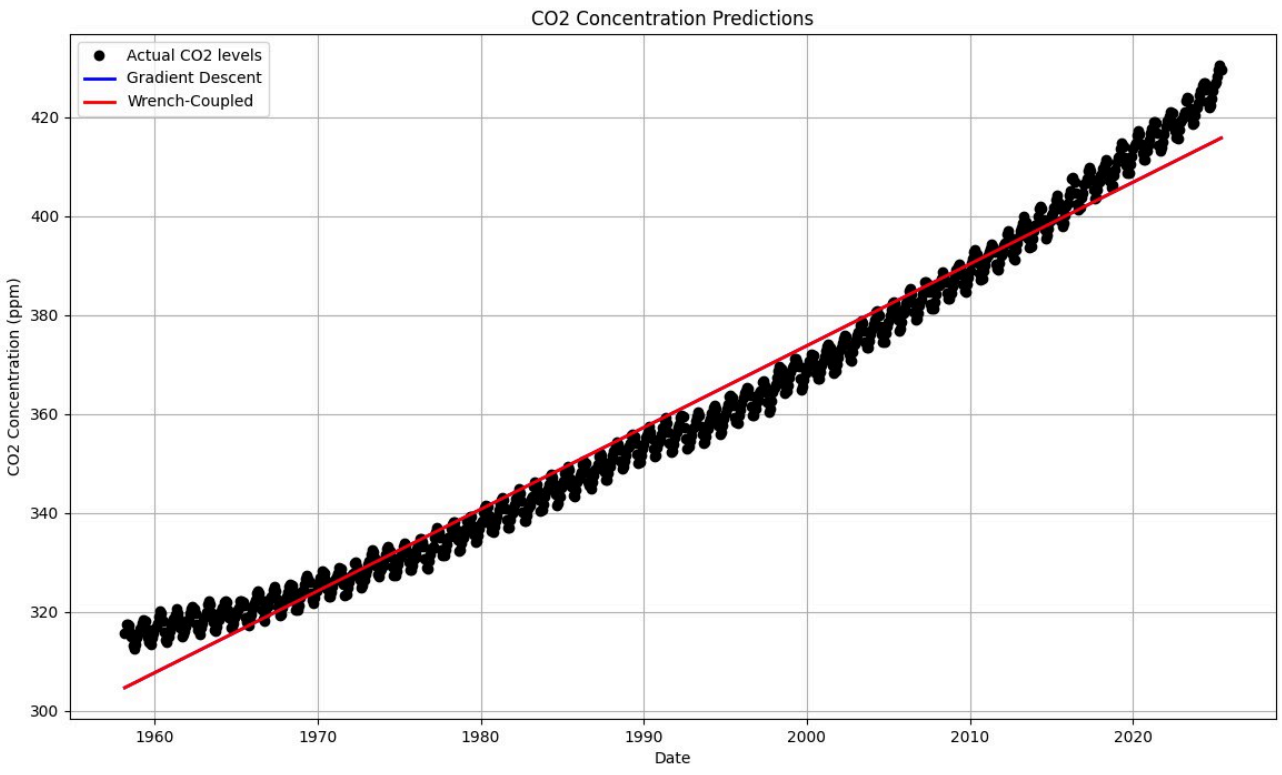
```
def denormalize_params(w, b, X_mean, X_std, y_mean, y_std):
    w_orig = w * (y_std / X_std)
    b_orig = b * y_std + y_mean - w_orig * X_mean
    return w_orig, b_orig
```

After denormalization, here are the final parameters:

- Gradient Descent final parameters: $w = 0.0489$, $b = 328.7905$
- Wrench-Coupled final parameters: $w = 0.0489$, $b = 328.7905$

5.3 Predictions Visualization

The figure below shows you what the predictions from both models look like compared to the actual CO2 concentration data. This plot gives you a clear visual idea of how well each model managed to fit the data trend.



The Mean Squared Error (MSE) on the original scale was 5.176450 for Gradient Descent, and 5.176449 for the Wrench-Coupled Optimizer. Again, these MSE values are super close, which tells us that both methods achieved pretty similar prediction accuracy on this dataset.

5.4 Discussion: Benefits, Drawbacks, and Interpretation

The core difference between standard Gradient Descent (GD) and the Wrench-Coupled algorithm lies in how they handle parameter updates:

Gradient Descent:

- Updates each parameter independently, using only that parameter's own gradient
- Simple but can get stuck in flat regions
- Uses just one learning rate parameter

Wrench-Coupled Approach:

- Treats parameters like connected physical objects
- Uses two key components:
    - Translation term: like normal GD, controlled by the mass m
    - Rotation term: couples w and b updates, controlled by inertia I
- Requires two control settings instead of one:
    - Mass (m) - controls straight movement speed
    - Inertia (I) - controls twisting resistance
- Can escape tricky areas that stop normal GD
- More complex but offers better coordination
- Breaks symmetry between parameters
- More flexible movement through parameter space

The Wrench-Coupled approach acts more like how real objects move - with both straight-line motion and rotational adjustment. While standard GD is simpler for basic problems, this new method may work better for complex, high-dimensional problems where parameters need to adjust in coordinated ways.

6. Conclusion

6.1 Summary of Contributions

The project successfully developed a novel optimization framework based on solid mechanics principles, termed the Wrench-Coupled Optimizer. This approach treats model parameters as a rigid body system, incorporating both translational forces and rotational torques to coordinate parameter updates. The physics-derived update rules provide a fundamentally different perspective on optimization compared to conventional gradient-based methods. While maintaining mathematical rigor, the method offers more intuitive parameter dynamics through its mechanical foundation.

6.2 Potential Extensions and Applications

Initial testing on linear regression revealed comparable performance between the Wrench-Coupled approach and traditional Gradient Descent. However, the method's potential becomes particularly evident when considering more complex optimization scenarios. Future applications could productively extend this framework to multidimensional linear systems or nonlinear models, where its coordinated update mechanism may better navigate challenging loss landscapes.

6.3 Limitations and Closing Remarks

The current implementation requires tuning two hyperparameters (mass and inertia) rather than a single learning rate, introducing additional complexity but also greater control. This work establishes a physics-grounded alternative to conventional optimization techniques, creating new opportunities to explore machine learning through the lens of physical dynamics.