Gradient Descent Method

Tamas Kis | tamas.a.kis@outlook.com | https://tamaskis.github.io

CONTENTS

1 GRADIENT DESCENT METHOD FOR UNCONSTRAINED OPTIMIZATION

1.1 Definition

Consider a multivariate, scalar-valued function $f(\mathbf{x})$, where $f : \mathbb{R}^n \to \mathbb{R}$. Often, $f(\mathbf{x})$ will obtain a **local minimum**. We denote this local minimum as

$$\boxed{f_{\text{min}} = f(\mathbf{x}_{\text{min}})} \tag{1}$$

where f_{min} is the local minimum, and where $\mathbf{x}_{\text{min}}$ is the **local minimizer**. Effectively, $\mathbf{x}_{\text{min}}$ is the location at which f attains its local minimum value. Consider an **unconstrained optimization** problem, where the goal is to find a local minimum of this multivariate, scalar-valued function $f(\mathbf{x})$:

$$\boxed{\underset{\mathbf{x}}{\text{minimize}} \quad f(\mathbf{x})} \tag{2}$$

In this context, $f(\mathbf{x})$ is known as the **objective function**. This optimization problem is an unconstrained optimization problem since there are no constraints on what values $\mathbf{x}$ can take on (i.e. $\mathbf{x}$ is not bounded by some region). The **gradient descent method** is an iterative method for unconstrained optimization (i.e. for minimizing an objective function). Like its name implies, it utilizes gradients in its approach to minimization. The basic motivation is that if a function is decreasing, then it will decrease the fastest in the direction of its *negative* gradient.

Let's consider the function $f(\mathbf{x})$ where $\mathbf{x} \in \mathbb{R}^n$. We make an initial guess, $\mathbf{x}_0$, for the local minimizer ($\mathbf{x}_{\text{min}}$) of f. To update our estimate of $\mathbf{x}_{\text{min}}$, we can move in the direction of the negative gradient[1] of f. But how much do we move by? We pick a value γ and scale $\nabla f(\mathbf{x}_0)$ by γ. Therefore, our first estimate $\mathbf{x}_1$ for $\mathbf{x}_{\text{min}}$ will be

$$\mathbf{x}_1 = \mathbf{x}_0 - \gamma \nabla f(\mathbf{x}_0)$$

Subsequent updates can be performed as

$$\mathbf{x}_{i+1} = \mathbf{x}_i - \gamma \nabla f(\mathbf{x}_i)$$

In its most general form, we can let γ vary every iteration i. Therefore, we write

$$\boxed{\mathbf{x}_{i+1} = \mathbf{x}_i - \gamma_i \nabla f(\mathbf{x}_i)} \tag{3}$$

The gradient descent method can only find *local* extrema reliably. For the gradient descent method to find the *global* minimum of a function, the function must be convex. For example, let's consider the case where you are looking for the global minimum of a function with multiple local minima. Then for different initial guesses, the gradient descent algorithm will likely converge to different local minima, and rarely converge to the global minimum.

1.2 Basic Implementation for a Differentiable, Univariate Function

For the most basic implementation, we consider a differentiable, univariate, scalar-valued function $f(x)$. The gradient of such a function is just its derivative, $f'(x)$. Therefore, the gradient descent method (assuming a constant learning rate γ) gives

$$x_{i+1} = x_i - \gamma f'(x) \tag{4}$$

Additionally, we have made an initial guess x_0, have set a tolerance (convergence criteria) TOL, and a maximum number of iterations i_{max}. Given an initial guess x_0, we can keep coming up with new estimates of the local minimizer. But how do we know when to stop? To resolve this issue, we define the **error**[2] as

$$\boxed{\varepsilon = |x_{i+1} - x_i|} \tag{5}$$

[1] To gain more intuition behind the gradient descent method, one can read the analogy in [1].

[2] Note that ε is an *approximate* error. The motivation behind using this definition of ε is that as i gets large (i.e. $i \to \infty$), $x_{i+1} - x_i$ approaches $x_{i+1} - x^*$ (*assuming* this sequence is convergent), where x^*_{min} is the true local minimizer (and therefore $x_{i+1} - x^*_{\text{min}}$ represents the *exact* error).

Once ε is small enough, we say that the estimate of the local minimizer has **converged** to the true local minimizer, x^*_{min}, within some **tolerance** (which we denote as TOL). Therefore, if we predetermine that, at most, we can *tolerate* an error of TOL, then we will keep iterating Eq. (4) until $\varepsilon < \text{TOL}$. In some cases, the error may never decrease below TOL, or take too long to decrease to below TOL. Therefore, we also define the **maximum number of iterations** (i_{max}) so that the algorithm does not keep iterating forever, or for too long of a time.

Algorithm 1:

Gradient descent algorithm for a differentiable, univariate, scalar-valued function.

Given:

• $f(x)$	- univariate, scalar-valued function ($f : \mathbb{R} \to \mathbb{R}$)
• $f'(x)$	- derivative of $f(x)$
• $x_0 \in \mathbb{R}$	- initial guess for local minimizer
• $\gamma \in \mathbb{R}$	- learning rate
• $\text{TOL} \in \mathbb{R}$	- tolerance
• $i_{\text{max}} \in \mathbb{Z}$	- maximum number of iterations

Procedure:

1. Initialize the error so that the loop will be entered.

   $\varepsilon = (2)(\text{TOL})$

2. Manually set the local minimizer estimate at the first iteration.

   $x_{\text{old}} = x_0$

3. Initialize x_{new} so its scope will not be limited to within the while loop.

   $x_{\text{new}} = 0$

4. Find the local minimizer using the gradient descent method.

```
$i = 1$
while ($\varepsilon > \text{TOL}$) and ($i < i_{\text{max}}$)
    (a) Update local minimizer estimate.
            $x_{\text{new}} = x_{\text{old}} - \gamma f'(x_{\text{old}})$
    (b) Calculate error.
            $\varepsilon = |x_{\text{new}} - x_{\text{old}}|$
    (c) Store current local minimizer estimate for next iteration.
            $x_{\text{old}} = x_{\text{new}}$
    (d) Increment loop index.
            $i = i + 1$
end
```

5. The converged local minimizer is assumed to be the most recent estimate, x_{new}.

   $x_{\text{min}} = x_{\text{new}}$

6. Find the local minimum.

$$f_{\min} = f(x_{\min})$$

Return:

- $x_{\min} \in \mathbb{R}$ - converged local minimizer
- $f_{\min} \in \mathbb{R}$ - converged local minimum

An implementation of this algorithm for finding the local minimum of $f(x) = (x-2)^2$ in MATLAB is shown below. Note that we can easily find the derivative of $f(x)$; it is just $f'(x) = 2(x-2)$.

```
% given parameters
f = @(x) (x-2)^2;    % f(x)
df = @(x) 2*(x-2);   % f'(x)
x0 = 2.1;            % initial guess
gamma = 0.0001;      % learning rate
TOL = 1e-9;          % tolerance
imax = 1e6;          % maximum number of iterations

% initializes the error so the loop will be entered
err = 2*TOL;

% sets local minimizer estimate at the first iteration of the gradient
% descent method as the initial guess
x_old = x0;

% initializes x_new so its scope isn't limited to the while loop
x_new = 0;

% gradient descent method
i = 1;
while (i < imax) && (err > TOL)

    % updates estimate of local minimizer
    x_new = x_old-gamma*df(x_old);

    % calculates error
    err = abs(x_new-x_old);

    % stores current local minimizer estimate for next iteration
    x_old = x_new;

    % increments loop index
    i = i+1;

end

% converged local minimizer and local minimum
x_min = x_new
f_min = f(x_min)
```

From a starting guess of 2.1, the algorithm converges (within tolerance) to $x_{\min} = 2$ in 49515 iterations.

1.3 Barzilai-Borwein Learning Rate

In the example in the previous section (Section 1.2), we used a learning rate of $\gamma = 0.0001$. With this γ, it took 49515 iterations to converge to a tolerance of $\text{TOL} = 10^{-9}$. What if we used $\gamma = 0.1$? Then the algorithm converges in just 78 iterations! Therefore, we could suspect that increasing γ will help us converge faster. However, if we increase γ to 1, then the solution never converges, and after the maximum number of iterations, x_{min} is approximated as 1.9000, which we know is clearly wrong. If we increase γ to 3, the algorithm diverges completely and results in `NaN`.

If we implement the same algorithm as before but for $f(x) = (x-2)^3$ (whose derivative is $f'(x) = 3(x-2)^2$), the algorithm will still converge even with $\gamma = 3$. Therefore, we know that even if a certain γ demonstrates convergence for one objective function, it does not guarantee convergence for another objective function. So how can we define γ for an arbitrary function? For a certain class of functions (more information can be found at [1]), we can define γ at every iteration using the **Barzilai-Borwein learning rate**, which is defined by Eq. (6) below.

$$\boxed{\gamma_i = \frac{\left|\left(\mathbf{x}_i - \mathbf{x}_{i-1}\right)^T \left[\nabla f\left(\mathbf{x}_i\right) - \nabla f\left(\mathbf{x}_{i-1}\right)\right]\right|}{\left\|\nabla f\left(\mathbf{x}_i\right) - \nabla f\left(\mathbf{x}_{i-1}\right)\right\|^2}} \tag{6}$$

For a univariate function $f(x)$, this becomes

$$\gamma_i = \frac{\left|\left(x_i - x_{i-1}\right)\left[f'(x_i) - f'(x_{i-1})\right]\right|}{\left[f'(x_i) - f'(x_{i-1})\right]^2}$$

The updated algorithm (for a differentiable, univariate function) is shown below. Note that we also have to set the local minimizer estimate at the first iteration (i.e. x_1) to a value slightly different than x_0 – otherwise, γ_1 will be undefined (its denominator would be 0); we can think of this as "kick-starting" the algorithm [1].

Algorithm 2:
Gradient descent algorithm for a differentiable, univariate, scalar-valued function using the Barzilai-Borwein learning rate.

Given:

• $f(x)$	- univariate, scalar-valued function ($f : \mathbb{R} \to \mathbb{R}$)
• $f'(x)$	- derivative of $f(x)$
• $x_0 \in \mathbb{R}$	- initial guess for local minimizer
• $\text{TOL} \in \mathbb{R}$	- tolerance
• $i_{\text{max}} \in \mathbb{Z}$	- maximum number of iterations

Procedure:

1. Initialize the error so that the loop will be entered.

$$\varepsilon = (2)(\text{TOL})$$

2. Manually set the local minimizer estimates at the first and second iterations based on the initial guess.

$$x_{\text{old}} = x_0$$
$$x_{\text{int}} = x_0 + 0.001$$

3. Initialize x_{new} so its scope will not be limited to within the while loop.

$$x_{\text{new}} = 0$$

4. Find the local minimizer using the gradient descent method with the Barzilai-Borwein learning rate.

```
$i = 2$
while $(\varepsilon > \text{TOL})$ and $(i < i_{\max})$
    (a) Calculate the learning rate.
            $\gamma_i = \dfrac{|(x_{\text{int}} - x_{\text{old}})\,[f'(x_{\text{int}}) - f'(x_{\text{old}})]|}{[f'(x_{\text{int}}) - f'(x_{\text{old}})]^2}$
    (b) Update local minimizer estimate.
            $x_{\text{new}} = x_{\text{int}} - \gamma_i f'(x_{\text{int}})$
    (c) Calculate error.
            $\varepsilon = |x_{\text{new}} - x_{\text{int}}|$
    (d) Store current and previous estimates for next iteration.
            $x_{\text{old}} = x_{\text{int}}$
            $x_{\text{int}} = x_{\text{new}}$
    (e) Increment loop index.
            $i = i + 1$
end
```

5. The converged local minimizer is assumed to be the most recent estimate, x_{new}.

$$x_{\min} = x_{\text{new}}$$

6. Find the local minimum.

$$f_{\min} = f(x_{\min})$$

Return:

- $x_{\min} \in \mathbb{R}$ - converged local minimizer
- $f_{\min} \in \mathbb{R}$ - converged local minimum

Note that this time around, we start with a loop index of 2, and we also initialize the *second* guess for $x_{\min}$, i.e. x_{int} for the first iteration. This is because γ_i (at the i^{th} iteration) is calculated using values for both the i^{th} and the $(i-1)^{\text{th}}$ iterations. If we started at $i = 1$, then we could not calculate γ_i, because there is no 0^{th} iteration. Additionally, the reason why we initialize x_1 to be $x_0 + 0.001$ is because we need x_0 and x_1 (as well as the gradient, or derivative in the univariate case, of f at those points) to be different, or else γ_1 would be undefined.

Algorithm 2 is implemented for $f(x) = (x-2)^2$ in the code below. This code converges in just 4 iterations (even with an initial guess that is wildly off)!

```
% given parameters
f = @(x) (x-2)^2;    % f(x)
df = @(x) 2*(x-2);   % f'(x)
x0 = 1000000;        % initial guess
TOL = 1e-9;          % tolerance
```

```
imax = 1e6;          % maximum number of iterations

% initializes the error so the loop will be entered
err = 2*TOL;

% sets local minimizer estimates for 1st iteration of the gradient descent
% method
x_old = x0;
x_int = x0+0.001;

% initializes x_new so its scope isn't limited to the while loop
x_new = 0;

% gradient descent method using the Barzilai-Borwein learning rate
i = 2;
while (i < imax) && (err > TOL)

    % calculates learning rate
    gamma = abs((x_int-x_old)*(df(x_int)-df(x_old)))/(df(x_int)-...
        df(x_old))^2;

    % updates estimate of local minimizer
    x_new = x_int-gamma*df(x_int);

    % calculates error
    err = abs(x_new-x_int);

    % stores current and previous local minimizer estimates for next
    % iteration
    x_old = x_int;
    x_int = x_new;

    % increments loop index
    i = i+1;

end

% converged local minimizer and local minimum
x_min = x_new
f_min = f(x_min)
```

1.4 General Implementation (Assuming an Unknown Gradient)

In the most general case, we want to minimize a multivariate function whose gradient we do not know. In such a case, we can approximate the gradient at a point $\mathbf{x}$ using the `igradient` function from the *Numerical Differentiation Toolbox* [2].

$$\nabla f(\mathbf{x}) \approx \texttt{igradient}(f, \mathbf{x})$$

Additionally, we use the Barzilai-Borwein learning rate since for an arbitrary f, it is difficult to determine what the appropriate learning rate should be. However, to add a degree of flexibility with the learning rate, we define the

learning rate scaling factor, λ, to modify the Barzilai-Borwein learning rate:

$$\boxed{\gamma_i = \frac{\lambda \left|(\mathbf{x}_i - \mathbf{x}_{i-1})^T \left[\nabla f(\mathbf{x}_i) - \nabla f(\mathbf{x}_{i-1})\right]\right|}{\|\nabla f(\mathbf{x}_i) - \nabla f(\mathbf{x}_{i-1})\|^2}} \tag{7}$$

For the general case, there are two basic algorithms for implementing the gradient descent method. The first implementation, shown in Algorithm 3 below, does *not* store the result of each iteration. On the other hand, the second implementation, shown in Algorithm 4, *does* store the result of each iteration. `fmingrad` implements both of these algorithms. Also note that `fmingrad` allows for the gradient, $\nabla f(\mathbf{x})$, to be specified directly. If it is not input, `fmingrad` approximates the gradient using the `igradient` function discussed earlier.

Since Algorithm 4 first needs to preallocate a potentially huge array to store all of the intermediate solutions, Algorithm 3 is often noticeably faster. Even if i_{max} (determines size of the preallocated array) is set to be a small number (for example, 10), Algorithm 3 is still faster. The reason we still consider and implement Algorithm 4 is so that convergence studies may be performed.

Algorithm 3:

Gradient descent method ("fast" implementation).

Given:

- $f(\mathbf{x})$ - multivariate, scalar-valued function ($f : \mathbb{R}^n \to \mathbb{R}$)
- $\mathbf{x}_0 \in \mathbb{R}^n$ - initial guess for local minimizer
- $\nabla f(\mathbf{x})$ - *(OPTIONAL)* gradient of $f(\mathbf{x})$
- $\lambda \in \mathbb{R}$ - *(OPTIONAL)* learning rate scaling factor
- $\text{TOL} \in \mathbb{R}$ - tolerance
- $i_{\text{max}} \in \mathbb{Z}$ - maximum number of iterations

Procedure:

1. Define the gradient using the `igradient` function if it is not input.

```
if $\nabla f(\mathbf{x})$ not specified
    $\nabla f(\mathbf{x}) \approx \texttt{igradient}(f, \mathbf{x})$
end
```

2. Manually set the local minimizer estimates at the first and second iterations based on the initial guess.

$$\mathbf{x}_{\text{old}} = \mathbf{x}_0$$
$$\mathbf{x}_{\text{int}} = \mathbf{x}_0 + 0.001$$

3. Calculate the gradient for the initial guess.

$$\mathbf{g}_{\text{old}} = \nabla f(\mathbf{x}_{\text{old}})$$

4. Initialize $\mathbf{x}_{\text{new}}$ so its scope will not be limited to within the while loop.

$$\mathbf{x}_{\text{new}} = \mathbf{0}$$

5. Initialize the error so that the loop will be entered.

$$\varepsilon = (2)(\text{TOL})$$

6. Find the local minimizer using the gradient descent method.

```
$i = 2$
while $(\varepsilon > \text{TOL})$ and $(i < i_{\max})$
    (a) Gradient at the current iteration.
        $\mathbf{g}_{\text{int}} = \nabla f(\mathbf{x}_{\text{int}})$
    (b) Calculate the learning rate.
        $\gamma = \dfrac{\lambda \left|(\mathbf{x}_{\text{int}} - \mathbf{x}_{\text{old}})^T (\mathbf{g}_{\text{int}} - \mathbf{g}_{\text{old}})\right|}{\left\|\mathbf{g}_{\text{int}} - \mathbf{g}_{\text{old}}\right\|^2}$
    (c) Update local minimizer estimate.
        $\mathbf{x}_{\text{new}} = \mathbf{x}_{\text{int}} - \gamma \mathbf{g}_{\text{int}}$
    (d) Calculate error.
        $\varepsilon = \|\mathbf{x}_{\text{new}} - \mathbf{x}_{\text{int}}\|$
    (e) Store current and previous estimates for next iteration.
        $\mathbf{x}_{\text{old}} = \mathbf{x}_{\text{int}}$
        $\mathbf{x}_{\text{int}} = \mathbf{x}_{\text{new}}$
    (f) Store gradient for next iteration.
        $\mathbf{g}_{\text{old}} = \mathbf{g}_{\text{int}}$
    (g) Increment loop index.
        $i = i + 1$
end
```

7. The converged local minimizer is assumed to be the most recent estimate, $\mathbf{x}_{\text{new}}$.

   $$\mathbf{x}_{\min} = \mathbf{x}_{\text{new}}$$

8. Find the local minimum.

   $$f_{\min} = f(\mathbf{x}_{\min})$$

Return:

- $\mathbf{x}_{\min} \in \mathbb{R}^n$ - converged local minimizer
- $f_{\min} \in \mathbb{R}$ - converged local minimum

Algorithm 4:

Gradient descent method ("return all" implementation).

Given:

• $f(\mathbf{x})$	- multivariate, scalar-valued function ($f:\mathbb{R}^n\to\mathbb{R}$)
• $\mathbf{x}_0\in\mathbb{R}^n$	- initial guess for local minimizer
• $\nabla f(\mathbf{x})$	- *(OPTIONAL)* gradient of $f(\mathbf{x})$
• $\lambda\in\mathbb{R}$	- *(OPTIONAL)* learning rate scaling factor
• $\mathrm{TOL}\in\mathbb{R}$	- tolerance
• $i_{\max}\in\mathbb{Z}$	- maximum number of iterations

Procedure:

1. Define the gradient using the `igradient` function if it is not input.

```
if $\nabla f(\mathbf{x})$ not specified
    $\nabla f(\mathbf{x})\approx \texttt{igradient}(f,\mathbf{x})$
end
```

2. Preallocate $\mathbf{x}\in\mathbb{R}^{n\times i_{\max}}$ to store the estimates of the local minimizer at each iteration.
3. Preallocate $\mathbf{g}\in\mathbb{R}^{n\times i_{\max}}$ to store the gradient at each iteration.
4. Manually set the local minimizer estimates at the first and second iterations based on the initial guess.

$$\mathbf{x}_1=\mathbf{x}_0$$
$$\mathbf{x}_2=\mathbf{x}_0+0.001$$

5. Calculate the gradient for the initial guess.

$$\mathbf{g}_1=\nabla f(\mathbf{x}_0)$$

6. Initialize the error so that the loop will be entered.

$$\varepsilon=(2)(\mathrm{TOL})$$

7. Find the local minimizer using the gradient descent method.

```
$i=2$
while $(\varepsilon>\mathrm{TOL})$ and $(i<i_{\max})$
```

```
        (a) Gradient at the current iteration.

                $\mathbf{g}_i = \nabla f(\mathbf{x}_i)$

        (b) Calculate the learning rate.

                $\gamma = \frac{\lambda \left|(\mathbf{x}_i - \mathbf{x}_{i-1})^T (\mathbf{g}_i - \mathbf{g}_{i-1})\right|}{\|\mathbf{g}_i - \mathbf{g}_{i-1}\|^2}$

        (c) Update local minimizer estimate.

                $\mathbf{x}_{i+1} = \mathbf{x}_i - \gamma \mathbf{g}_i$

        (d) Calculate error.

                $\varepsilon = \|\mathbf{x}_{i+1} - \mathbf{x}_i\|$

        (e) Increment loop index.

                $i = i + 1$
end
```

8. Preallocate a vector, $\mathbf{f} \in \mathbb{R}^i$, to store the local minimum estimates at each iteration.
9. Find the local minimum estimates at each iteration.

```
for $j = 1$ to $i$
    $\mathbf{f}_j = f(\mathbf{x}_j)$
end
```

Return:

- $\mathbf{x} \in \mathbb{R}^{n \times i}$ - matrix where the first column is the initial guess for the local minimizer, the subsequent columns are the intermediate local minimizer estimates, and the final column is the converged local minimizer
- $\mathbf{f} \in \mathbb{R}^i$ - vector where the first element is the initial guess for the local minimum, the subsequent elements are the intermediate local minimum estimates, and the final element is the converged local minimum

Note:

- i is the number of iterations it took for the solution to converge.

2 OTHER APPLICATIONS

2.1 Constrained Optimization

Consider a constrained optimization problem of the form [3]

$$\boxed{\begin{array}{ll} \underset{\mathbf{x}}{\text{minimize}} & f_c(\mathbf{x}) \\ \text{subject to} & \mathbf{g}(\mathbf{x}) \leq \mathbf{0} \\ & \mathbf{h}(\mathbf{x}) = \mathbf{0} \end{array}} \tag{8}$$

The gradient descent method solves unconstrained optimization problems of the form

$$\underset{\mathbf{x}}{\text{minimize}} \quad f(\mathbf{x})$$

However, as a workaround, we can modify the objective function with penalty terms to turn a constrained optimization problem into an unconstrained optimization problem.

Let's consider the case where we want to minimize a function $f_c(\mathbf{x})$ subject to the **inequality constraint**

$$\mathbf{g}(\mathbf{x}) \leq \mathbf{0}$$

and the **equality constraint**

$$\mathbf{h}(\mathbf{x}) = \mathbf{0}$$

Our goal is to define the objective function

$$\boxed{f(\mathbf{x}) = f_c(\mathbf{x}) + p_g(\mathbf{x}) + p_h(\mathbf{x})} \tag{9}$$

where $f_c(\mathbf{x})$ is the objective function for the constrained problem, $f(\mathbf{x})$ is the objective function for the equivalent unconstrained problem, $p_g(\mathbf{x})$ is a penalty term that accounts for the inequality constraint, and $p_h(\mathbf{x})$ is a penalty term that accounts for the equality constraint.

Let's begin with the inequality constraint. For this constraint, we only want to penalize $\mathbf{x}$ when $\mathbf{g}(\mathbf{x}) > \mathbf{0}$. Therefore, we define the penalty term piecewise as

$$\boxed{p_g(\mathbf{x}) = \begin{cases} 0, & \mathbf{g}(\mathbf{x}) \leq \mathbf{0} \\ w_g \left\|\mathbf{g}(\mathbf{x})\right\|^2, & \mathbf{g}(\mathbf{x}) > \mathbf{0} \end{cases}} \tag{10}$$

where w_g is some scalar weight. Similarly, for the equality constraint, we only want to penalize $\mathbf{x}$ when $\mathbf{h}(\mathbf{x}) \neq \mathbf{0}$. Therefore, we define the penalty term piecewise as

$$\boxed{p_h(\mathbf{x}) = \begin{cases} 0, & \mathbf{h}(\mathbf{x}) = \mathbf{0} \\ w_h \left\|\mathbf{h}(\mathbf{x})\right\|^2, & \mathbf{h}(\mathbf{x}) \neq \mathbf{0} \end{cases}} \tag{11}$$

where w_h is some scalar weight.

This is a simple workaround and provides no assurances on convergence. Additionally, the scalar weights w_g and w_h are parameters that need to be tuned manually.

2.2 Solution of Linear Systems

Consider the linear system

$$\boxed{\mathbf{A}\mathbf{x} = \mathbf{b}} \tag{12}$$

Our goal is to solve for the $\mathbf{x}^*$ that solves this linear system. Rearranging,

$$\mathbf{A}\mathbf{x} - \mathbf{b} = \mathbf{0}$$

We know that $\|\mathbf{A}\mathbf{x} - \mathbf{b}\| \geq 0\ \forall \mathbf{x}$ (and by extension, $\|\mathbf{A}\mathbf{x} - \mathbf{b}\|^2 \geq 0\ \forall \mathbf{x}$). Additionally, we know that for the solution, $\mathbf{x}^*$, we have $\mathbf{A}\mathbf{x}^* - \mathbf{b} = \mathbf{0}$. Therefore, the minimizer of $\|\mathbf{A}\mathbf{x} - \mathbf{b}\|^2$ is our solution for $\mathbf{A}\mathbf{x} = \mathbf{b}$. Thus, we define the objective function as

$$\boxed{f(\mathbf{x}) = \|\mathbf{A}\mathbf{x} - \mathbf{b}\|^2} \tag{13}$$

The gradient of this objective function (for real $\mathbf{A}$ and $\mathbf{b}$) is

$$\boxed{\nabla f(\mathbf{x}) = 2\mathbf{A}^T(\mathbf{A}\mathbf{x} - \mathbf{b})} \tag{14}$$

The method above works for a general matrix $\mathbf{A}$. If $\mathbf{A}$ is real, symmetric, and positive-definite, the objective function $f(\mathbf{x}) = \mathbf{x}^T\mathbf{A}\mathbf{x} - 2\mathbf{x}^T\mathbf{b}$ (with gradient $\nabla f(\mathbf{x}) = 2(\mathbf{A}\mathbf{x} - \mathbf{b})$) can be used instead [1].

2.3 Solution of Nonlinear Systems

Consider a nonlinear system of the form

$$\boxed{\mathbf{g}(\mathbf{x}) = \mathbf{0}} \tag{15}$$

We know that $\|\mathbf{g}(\mathbf{x})\|^2 \geq 0\ \forall \mathbf{x}$. Additionally, we know that for the solution, $\mathbf{x}^*$, we have $\mathbf{g}(\mathbf{x}^*) = \mathbf{0}$. Therefore, the minimizer of $\|\mathbf{g}(\mathbf{x})\|^2$ is our solution for $\mathbf{g}(\mathbf{x}) = \mathbf{0}$. Thus, we define the objective function as

$$\mathbf{f}(\mathbf{x}) = \|\mathbf{g}(\mathbf{x})\|^2$$

Recall that for an arbitrary (real) vector $\mathbf{v}$, $\|\mathbf{v}\|^2 = \mathbf{v}^T\mathbf{v}$. Therefore, our objective function becomes [1]

$$\boxed{\mathbf{f}(\mathbf{x}) = \mathbf{g}(\mathbf{x})^T\mathbf{g}(\mathbf{x})} \tag{16}$$

REFERENCES

[1] *Gradient descent*. Wikipedia. Accessed: April 17, 2020. URL: https://en.wikipedia.org/wiki/Gradient_descent.

[2] Tamas Kis. *Numerical Differentiation Toolbox*. 2021. URL: https://github.com/tamaskis/Numerical_Differentiation_Toolbox-MATLAB.

[3] *Optimization problem*. Wikipedia. Accessed: April 17, 2020. URL: https://en.wikipedia.org/wiki/Optimization_problem.