Multidimensional optimization

Release 1.0.0

Victor Barbarich

Mar 29, 2022

CONTENTS:

CHAPTER

ONE

CONCEPTUAL MODEL

1.1 Imagine situation

Factory's management has a function that describes the income from the sale of products. The function depends on many variables, e.g. the salary by each employee, the cost of each product and so on.

The management gave us the function $g(\mathbf{x})$, the running costs, the gradient of the functions in analytical form (Their mathematicians did a good job). But it will better if our solution automatically calculates gradient.

And our goal is to ensure the **best cost distribution**.

* For unifying let's introduce $f(\mathbf{x}) = -g(\mathbf{x})$.

CHAPTER

TWO

MATHEMATICAL MODEL

We have analyzed their problem. The solution is to use the gradient methods.

We have chosen 2 methods:

1. Gradient Descent
2. Nonlinear conjugate gradient method

2.1 Problem statement

1. $f(\mathrm{x}) : \mathbb{R}^n \to \mathbb{R}$
2. $\mathrm{x} \in X \subseteq \mathbb{R}^n$
3. $f \longrightarrow \min\limits_{\mathrm{x} \in X}$
4. The f is defined and differentiable on the X
5. Convergence to a local minimum can be guaranteed. When the function f is convex, gradient descent can converge to the global minima.

2.2 Gradient descent

2.2.1 Equations

1. Function argument:

$$\mathrm{x} = \begin{bmatrix} x_1 & x_2 & \dots & x_n \end{bmatrix}^\top \tag{2.1}$$

2. Gradient:

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} & \dots & \frac{\partial f}{\partial x_n} \end{bmatrix}^\top \tag{2.2}$$

3. Gradient step:

$$\mathrm{x}_{i+1} = \mathrm{x}_i - \gamma_i \cdot \nabla f(\mathrm{x}_i) \tag{2.3}$$

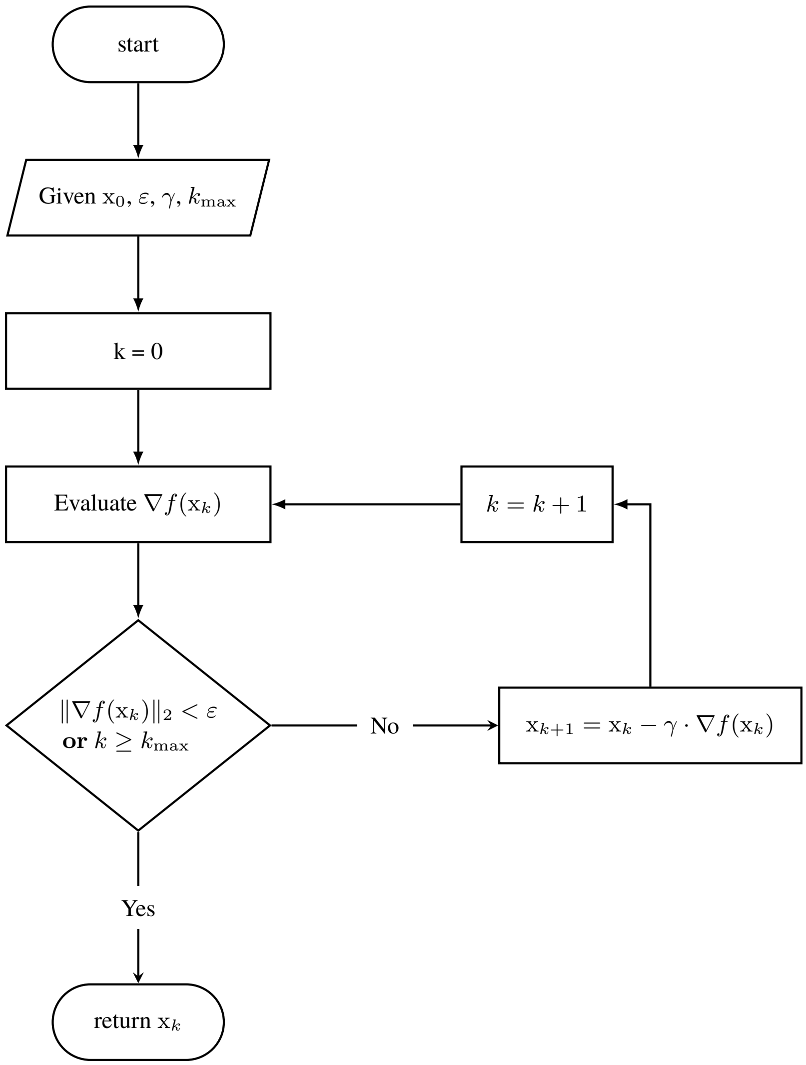
4. Terminate condition:

$$\|\nabla f(\mathrm{x}_i)\|_2 < \varepsilon \tag{2.4}$$

2.2.2 Algorithm with constant step

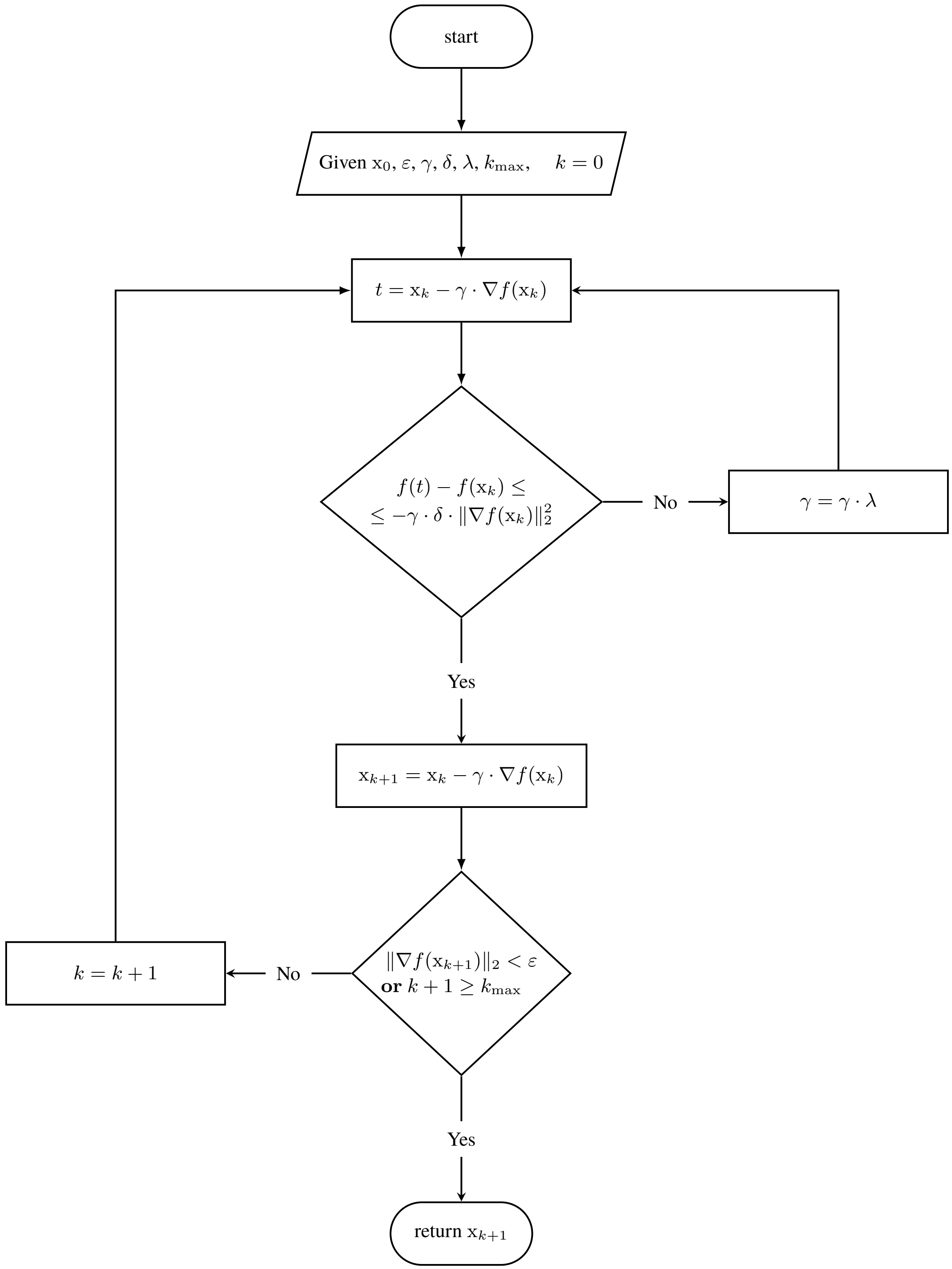
The gradient of the function shows us the direction of increasing the function. The idea is to move in the opposite direction to x_{k+1} where $f(\mathrm{x}_{k+1}) < f(\mathrm{x}_k)$.

But, if we add a gradient to x_k without changes, our method will often diverge. So we need to add a gradient with some weight γ.



2.2.3 Algorithm with descent step

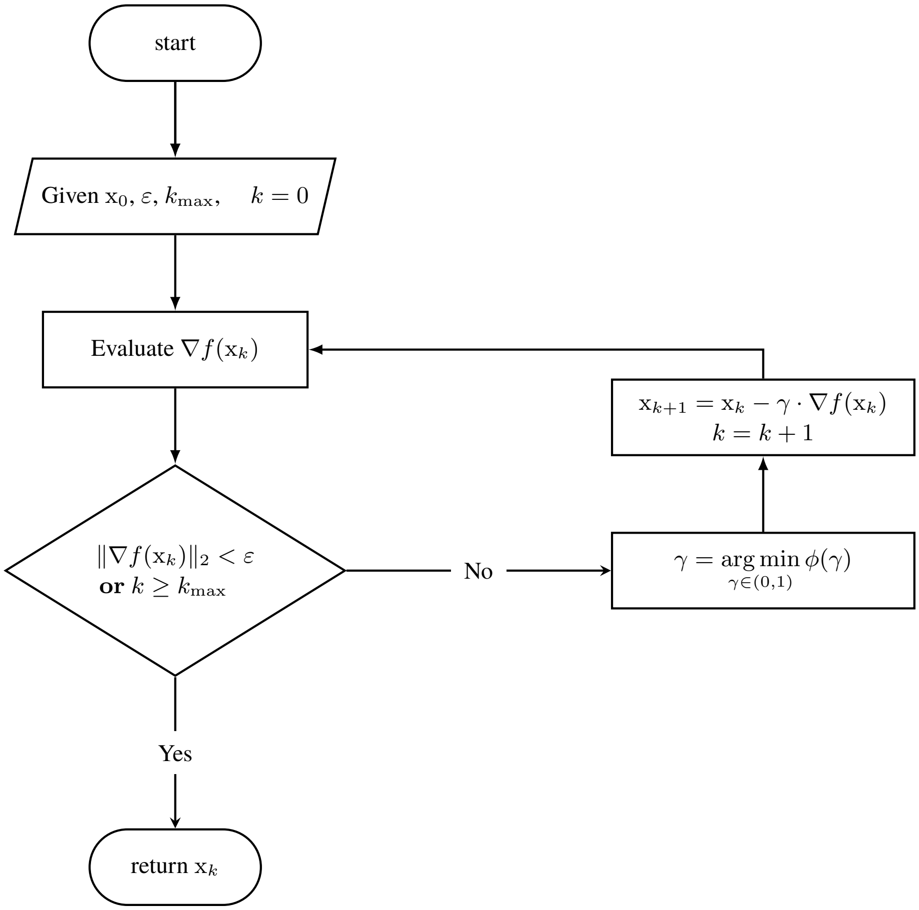
Requirements: $0 < \lambda < 1$ is the step multiplier, $0 < \delta < 1$.



2.2.4 Algorithm with optimal step size

Another good idea is to find a γ that minimizes $\phi(\gamma) = f(\mathrm{x}_k + \gamma \cdot \nabla f(\mathrm{x}_k))$

So we have a task to find the $\gamma_{\min} = \arg\min\limits_{\gamma \in (0,1)} \phi(\gamma)$. We will use Brent's algorithm to search $\gamma_{\min}$.



2.3 Strong Wolfe conditions

The conditions necessary to minimize $\phi(\gamma) = f(\mathrm{x}_k + \gamma p_k)$ and find $\gamma_k = \arg\min\limits_{\gamma} \phi$

$$f(\mathrm{x}_k + \gamma_k p_k) \leq f(\mathrm{x}_k) + c_1 \gamma_k \nabla f_k^\top p_k \tag{2.5}$$

$$|\nabla f(\mathrm{x}_k + \gamma_k p_k)^\top p_k| \leq -c_2 \nabla f_k^\top p_k \tag{2.6}$$

2.4 Nonlinear conjugate gradient method

The Fletcher–Reeves method. p_k is the direction to evaluate x_{k+1}.

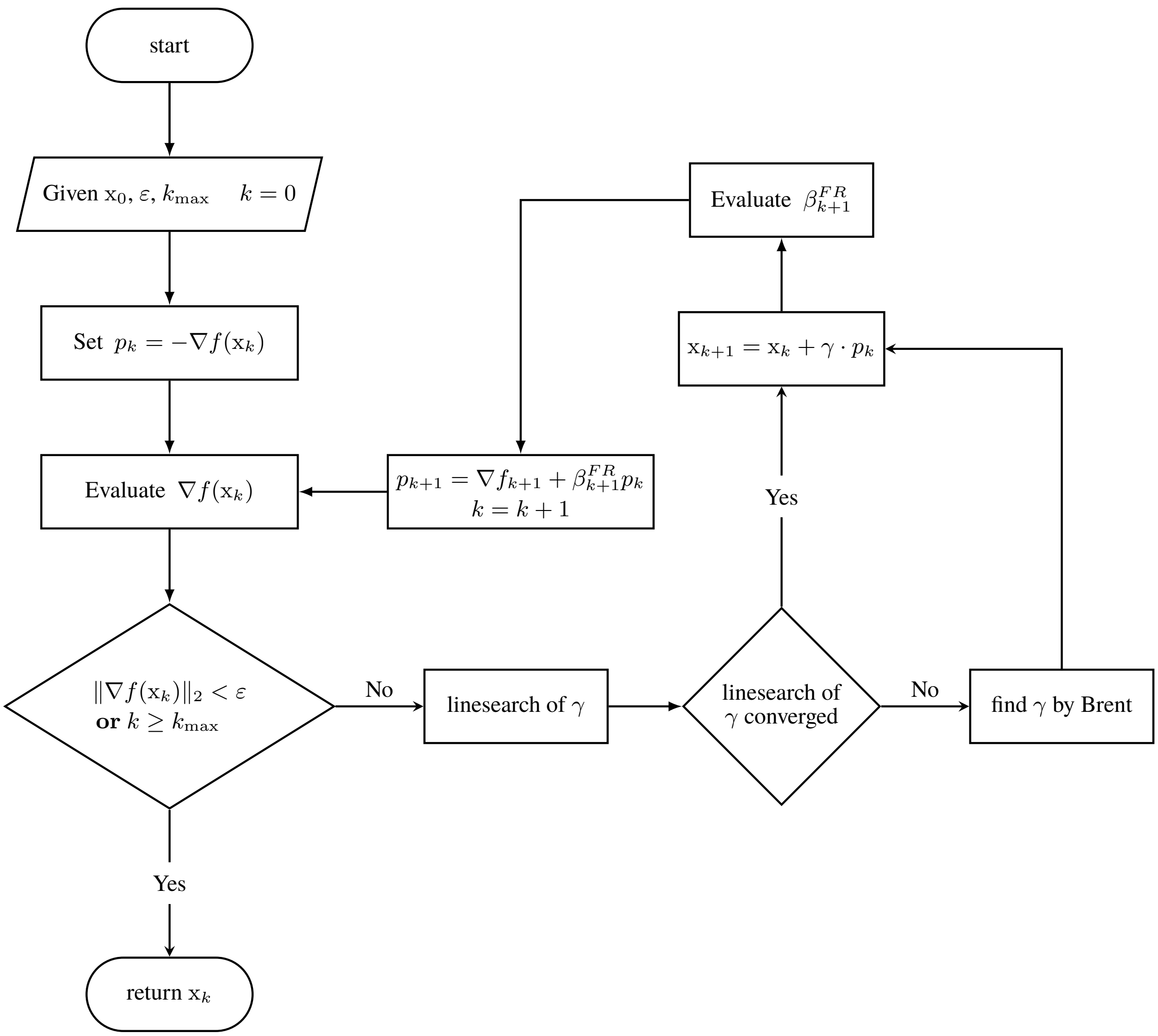
1. $p_0 = -\nabla f_0$
2. $p_{k+1} = \nabla f_{k+1} + \beta_{k+1}^{FR} p_k$

In the RF method, γ is searched using Line Search (Nocedal, Wright (2006) *Numerical Optimization* pp.60-61)

Our modification is that if Line Search does not converge, use Brent's algorithm to search for $\gamma_{\min} = \arg\min_{\gamma} \phi$

$$\beta_{k+1}^{FR} = \frac{\nabla f_{k+1}^\top \nabla f_{k+1}}{\nabla f_k^\top \nabla f_k} \tag{2.7}$$

2.4.1 Algorithm Nonlinear conjugate gradient method



CHAPTER

THREE

REQUIREMENTS

3.1 Required input fields

1. Type of algorithm. (Gradient with constant step, fractional step, optimal step, nonlinear conjugate gradient method and Newton-CG)
2. Field with function input $f(\mathrm{x})$
3. Field with the analytical gradient flag. If analytical gradient flag is True then request the gradient in analytical form. $\nabla f(\mathrm{x})$
4. Field with the start point (The default value is random numbers from a uniform distribution on (-1, 1))
5. Search precision ε
6. Field with a history saving flag
7. Required parameters for each method

3.2 Visualization Required

1. If the function depends on 1 variable, then it is necessary to draw a curve and animate the movement of the point and draw the previous steps.
2. If the function depends on 2 variables, then you need to draw contour lines and animate the movement of the point and draw the previous steps.
3. For functions of larger dimensions, output a graph of the decreasing gradient norm and the function values for each iteration.
4. For 1 and 2 dimensional output a graph of the decreasing gradient norm and the function values for each iteration too.

3.3 Requirements for methods

1. Each method must keep a history of all iterations. Be sure to save every x_k, f_k, $\|\nabla f_k\|$
2. The solution must contain the final point, the value of the function, the number of iterations, and the history.
3. The solution should cause a minimum number of calculations of the function values

CHAPTER

FOUR

MULTI DIMENSIONAL OPTIMIZATION

4.1 Algorithms

4.1.1 gradient_descent_constant_step

gradient_descent_constant_step(*function*, *x0*, *epsilon=1e-05*, *gamma=0.1*, *max_iter=500*, *verbose=False*, *keep_history=False*)

Algorithm with constant step. Documentation: paragraph 2.2.2, page 3. The gradient of the function shows us the direction of increasing the function. The idea is to move in the opposite direction to x_{k + 1} where f(x_{k + 1}) < f(x_{k}). But, if we add a gradient to x_{k} without changes, our method will often diverge. So we need to add a gradient with some weight gamma.

Code example::

```
>>> def func(x): return x[0] ** 2 + x[1] ** 2
>>> x_0 = [1, 2]
>>> solution = gradient_descent_constant_step(func, x_0)
>>> print(solution[0])
{'point': array([1.91561942e-06, 3.83123887e-06]), 'f_value': 1.
↪834798903191018e-11}
```

Parameters

- **function** (*Callable[[numpy.ndarray], numbers.Real]*) – callable that depends on the first positional argument
- **x0** (*Sequence[numbers.Real]*) – numpy ndarray which is initial approximation
- **epsilon** (*numbers.Real*) – optimization accuracy
- **gamma** (*numbers.Real*) – gradient step
- **max_iter** (*numbers.Integral*) – maximum number of iterations
- **verbose** (*bool*) – flag of printing iteration logs
- **keep_history** (*bool*) – flag of return history

Returns tuple with point and history.

Return type *Tuple*[*MultiDimensionalOptimization.algorithms.support.Point*, *MultiDimensionalOptimization.algorithms.support.HistoryMDO*]

4.1.2 gradient_descent_frac_step

gradient_descent_frac_step(*function*, *x0*, *epsilon=1e-05*, *gamma=0.1*, *delta=0.1*, *lambda0=0.1*, *max_iter=500*, *verbose=False*, *keep_history=False*)

Algorithm with fractional step. Documentation: paragraph 2.2.3, page 4 Requirements: 0 < lambda0 < 1 is the step multiplier, 0 < delta < 1.

Code example:

```
>>> def func(x): return x[0] ** 2 + x[1] ** 2
>>> x_0 = [1, 2]
>>> solution = gradient_descent_frac_step(func, x_0)
>>> print(solution[0])
{'point': array([1.91561942e-06, 3.83123887e-06]), 'f_value': 1.834798903191018e-11}
```

Parameters

- **function** (*Callable[[numpy.ndarray], numbers.Real]*) – callable that depends on the first positional argument
- **x0** (*Sequence[numbers.Real]*) – numpy ndarray which is initial approximation
- **epsilon** (*numbers.Real*) – optimization accuracy
- **gamma** (*numbers.Real*) – gradient step
- **delta** (*numbers.Real*) – value of the crushing parameter
- **lambda0** (*numbers.Real*) – initial step
- **max_iter** (*numbers.Integral*) – maximum number of iterations
- **verbose** (*bool*) – flag of printing iteration logs
- **keep_history** (*bool*) – flag of return history

Returns tuple with point and history.

Return type *Tuple*[*MultiDimensionalOptimization.algorithms.support.Point*, *MultiDimensionalOptimization.algorithms.support.HistoryMDO*]

4.1.3 gradient_descent_optimal_step

gradient_descent_optimal_step(*function*, *x0*, *epsilon=1e-05*, *max_iter=500*, *verbose=False*, *keep_history=False*)

Algorithm with optimal step. Documentation: paragraph 2.2.4, page 5 The idea is to choose a gamma that minimizes the function in the direction f'(x_k)

Code example:

```
>>> def func(x): return -np.e ** (- x[0] ** 2 - x[1] ** 2)
>>> x_0 = [1, 2]
>>> solution = gradient_descent_optimal_step(func,x_0)
>>> print(solution[0])
{'point': array([9.21321369e-08, 1.84015366e-07]), 'f_value': -0.9999999999999577}
```

Parameters

- **function** (*Callable[[numpy.ndarray], numbers.Real]*) – callable that depends on the first positional argument
- **x0** (*Sequence[numbers.Real]*) – numpy ndarray which is initial approximation
- **epsilon** (*numbers.Real*) – optimization accuracy
- **max_iter** (*numbers.Integral*) – maximum number of iterations
- **verbose** (*bool*) – flag of printing iteration logs
- **keep_history** (*bool*) – flag of return history

Returns tuple with point and history.

Return type *Tuple*[*MultiDimensionalOptimization.algorithms.support.Point*, *MultiDimensionalOptimization.algorithms.support.HistoryMDO*]

4.1.4 nonlinear_cgm

nonlinear_cgm(*function*, *x0*, *epsilon=1e-05*, *max_iter=500*, *verbose=False*, *keep_history=False*)

Paragraph 2.4.1 page 6 Algorithm works when the function is approximately quadratic near the minimum, which is the case when the function is twice differentiable at the minimum and the second derivative is non-singular there.

Code example::

```
>>> def func(x): return 10 * x[0] ** 2 + x[1] ** 2 / 5
>>> x_0 = [1, 2]
>>> solution = nonlinear_cgm(func, x_0)
>>> print(solution[0])
{'point': array([-1.70693616e-07,  2.90227591e-06]), 'f_value': 1.
↪9760041961386155e-12}
```

Parameters

- **function** (*Callable[[numpy.ndarray], numbers.Real]*) – callable that depends on the first positional argument
- **x0** (*Sequence[numbers.Real]*) – numpy ndarray which is initial approximation
- **epsilon** (*numbers.Real*) – optimization accuracy
- **max_iter** (*numbers.Integral*) – maximum number of iterations
- **verbose** (*bool*) – flag of printing iteration logs
- **keep_history** (*bool*) – flag of return history

Returns tuple with point and history.

Return type *Tuple*[*MultiDimensionalOptimization.algorithms.support.Point*, *MultiDimensionalOptimization.algorithms.support.HistoryMDO*]

4.1.5 support

class HiddenPrints

Bases: `object`

Object hides print. Working with context manager "with":

```
>>> with HiddenPrints():
>>>     print("It won't be printed")
```

class HistoryMDO

Bases: `TypedDict`

Class with an optimization history of gradient descent methods

f_grad_norm: List[numbers.Real]

f_value: List[numbers.Real]

iteration: List[numbers.Integral]

message: AnyStr

x: List[Sequence]

class Point

Bases: `TypedDict`

Class with an output optimization point

f_value: numbers.Real

point: numbers.Real

gradient(*function*, *x0*, *delta_x=1e-08*)

Calculate and return a gradient using a two-side difference

Parameters

- **function** (*Callable*) – callable that depends on the first positional argument
- **x0** (*numpy.ndarray*) – the point at which we calculate the gradient
- **delta_x** – precision of derivation

Returns vector np.ndarray with the gradient at the point

Return type numpy.ndarray

update_history_grad_descent(*history*, *values*)

Update HistoryMDO with values, which contains iteration, f_value, f_grad_norm, x as a list

Parameters

- **history** (`MultiDimensionalOptimization.algorithms.support.HistoryMDO`) – object of HistoryMDO
- **values** (*List*) – new values that need to append in history in order iteration, f_value, f_grad_norm, x

Returns updated history

Return type *MultiDimensionalOptimization.algorithms.support.HistoryMDO*