# Python Implementation of Subgradient Descent and Nesterov's Accelerated Gradient

Wei Zhao Yiyang Zhang

April 2022

**Abstract**

In this STATS 606 project, we successfully built a python package "ProNfGD" that can implement Subgradient descent and Nesterov's fast gradient method. This package will allow people to easily use these two algorithms in optimization problems from different tasks.

We also give a concrete example of implementing the proposed algorithms on realistic problems and test our python package on it. In the example, we use Subgradient algorithm we built in our package to do the optimization for the mathematical model we built. We want to use this model to construct an Index fund by using only part of its corresponding market Index's components. And based on the result of the optimization, we test the optimal model on the out-of-sample period and found that it is highly effective. Our constructed Index follows the trend of corresponding Market Index pretty well.

**Keywords:** Python, Subgradient descent, Nesterov's accelerated gradient method, Optimization, Index Fund construction.

# 1 Introduction

Python provides general-purpose optimization routines via its scipy.optimize package. However, when encountered with specific problems, the function of the current available python package's effects are limited. Besides, the Subgradient descent method and the Nesterov's fast gradient method are sometimes more efficient than the general optimization methods.

Therefore, in our STATS 606 project, we decided to build a python package "ProNfGD" that can implement two algorithms presented in the course: Subgradient descent and Nesterov's fast gradient method. This package will allow people to easily use the two algorithms in optimization problems from different tasks [Rud16]. We also give a concrete example of implementing the proposed algorithms on realistic problems.

# 2 Subgradient descent

Subgradient method [al] is an iterative algorithm for minimizing convex functions, used predominantly in non-differentiable optimization for cost functions that are convex but non-differentiable. It is often slower than Newton's Method when applied to convex differentiable functions, but is much simpler and can be used on convex non-differentiable functions where Newton's Method will not converge and thus it can be applied to a far wider variety of problems. It was first developed by Naum Z. Shor in the Soviet Union in the 1960's. First, we take a look at the definition of subgradient. $g$ is a subgradient of $f$ at x if, for all y, the following is true:

$$f(y) \geq f(x) + g^T(y - x)$$

Now consider $f$ convex with domain equals $\mathbb{R}^n$, but not necessarily differentiable, to minimize $f$, the subgradient method is like gradient descent, but replacing gradients with subgradients. Initialize $x^{(0)}$, repeat:

$$x^{(k)} = x^{(k-1)} - \alpha_k g^{(k-1)}, k = 1, 2, 3, \ldots$$

where $x^{(k)}$ is the $k$th iterate, $g^{(k-1)} \in \partial f(x^{(k-1)})$, any subgradient of $f$ at $x^{(k-1)}$, and $\alpha_k > 0$ is the $k$th step size. Thus, at each iteration of the subgradient method, we take a step in the direction of a negative subgradient. Recall that a subgradient of $f$ at $x$ is any vector $g$ that satisfies the inequality $f(y) \geq f(x) + g^T(y - x)$ for all $y$. When $f$ is differentiable, the only possible choice for $g^{(k)}$ is $\nabla f(x^{(k)})$, and the subgradient method then reduces to the gradient method except for the choice of step size. Since subgradient method is not necessarily a descent method, thus we keep track of best iterates $x^{(k)}_{\text{best}}$ among $x^0, \ldots, x^{(k)}$ so far, i.e., the one with smallest function value. At each step, we set

$$f(x^{(k)}_{\text{best}}) = \min\{f(x^{(k-1)}_{\text{best}}), f(x^{(k)})\},$$

and set $i^{(k)}_{\text{best}} = k$ if $f(x^{(k)}) = f(x^{(k)}_{\text{best}})$, i.e., if $x^{(k)}$ is the best point found so far. In a descent method there is no need to do so —the current point is always the best one so far. Then we have

$$f(x^{(k)}_{\text{best}}) = \min\{f(x^{(1)}), \ldots, f(x^{(k)})\},$$

i.e., $f(x^{(k)}_{\text{best}})$ is the best objective value found in $k$ iterations. Since $f(x^{(k)}_{\text{best}})$ is decreasing, it has a limit (which can be $-\infty$)

For subgradient descent method, the key difference from gradient descent is that the former algorithm's step sizes are pre-specified instead of adaptively computed. There are several different types of step size. For example, constant step size $\alpha_k = h, k = 1, 2, 3, \ldots$. Another choice of step size is called diminishing step

size $_k$, which meets conditions:

$$\sum_{k=1}^{\infty} \alpha_k^2 < \infty, \sum_{k=1}^{\infty} \alpha_k = \infty,$$

i.e., square summable but not summable.

The convergence of subgradient method is given by the following theorem. Assume that $f$ is convex with domain equals $\mathbb{R}^n$, and also that $f$ is Lipschitz continuous with constant $G > 0$, i.e.,

$$|f(x) - f(y)| \leq G \parallel x - y \parallel_2, \text{for all } x, y$$

**Theorem 1** *[DD18] For a fixed step size $h$, subgradient method satisfies*

$$\lim_{x \to \infty} f(x_{best}^{(k)}) \leq f(x^*) + G^2 \frac{h}{2},$$

*where $x^*$ is the optimum.*

**Theorem 2** *[Loi+21] For diminishing step sizes, subgradient method satisfies*

$$\lim_{x \to \infty} f(x_{best}^{(k)}) = f(x^*),$$

*where $x^*$ is the optimum.*

Subgradient method has convergence rate $\mathcal{O}(\frac{1}{\epsilon^2})$, which is shower than $\mathcal{O}(\frac{1}{\epsilon})$ rate of gradient descent.

# 3 Nesterov's accelerated gradient method

Plain gradient descent algorithm can be improved by stochastic gradient descent in terms of complexity and generalization. Unfortunately, there is some condition that even stochastic gradient descent can become very slow, for example when the gradient is consistently small. This is due to the update rule of he algorithm that only depends on the gradients at each iteration only. However, noisy gradients can be a problem too since stochastic gradient descent will frequently follow the wrong gradient. Momentum method can be used to mitigate these problems and accelerate convergence process compared to plain stochastic gradient descent. Developed in 1964 by Polyak[Pol64], Momentum method is a technique that can accelerate gradient descent by taking accounts of previous gradients in the update rule at each iteration. This can be observed in the update rule in each iteration, which is

$$y_{k+1} = \mu(x_k - x_{k-1}) - \eta \nabla f(x_k) = \mu y_k - \eta \nabla f(x_k)$$
$$x_{k+1} = x_k + y_{k+1}$$

where $y$ is the velocity term representing the direction and speed at which the parameter should be tweaked and $\mu$ is the decaying hyper-parameter, which determines how quickly accumulated previous gradients will decay. If $\mu$ is much greater than $\eta$, the accumulated previous gradients will be dominant in the update rule so the gradient at the iteration will not change the current direction sharply. This is beneficial when the gradient is noisy because the gradient will stay in the true direction for good. On the other hand, if $\mu$ is much smaller than $\eta$, the accumulated previous gradients can act as a smoothing factor for the gradient.

Nesterov's accelerated gradient descent (NAG) is closely related to momentum method. The difference between momentum method and Nesterov accelerated gradient descent lies in the gradient computation

phase. In momentum method, the gradient was computed using current parameter $x_k$, whereas in Nesterov's accelerated gradient descent, the gradient was computed using interim parameter which is obtained by applying the velocity to current parameter. Specifically, the update rule in each iteration of NAG is

$$y_{k+1} = \mu(x_k - x_{k-1}) - \eta\nabla f\big(x_k + \mu(x_k - x_{k-1})\big) = \mu y_k - \eta\nabla f\big(x_k + \mu(x_k - x_{k-1})\big)$$
$$x_{k+1} = x_k + y_{k+1}$$

We can view Nesterov's accelerated gradient descent as the correction factor for momentum method. Considering the case when the velocity added to the parameters resulting in immediate unwanted high loss, e.g., exploding gradient case. In this case, the momentum method can be very slow since the optimization path taken exhibits large oscillations. In Nesterov's accelerated gradient descent case, we can view it as looking before leaping, i.e., peeking through the interim parameters where the added velocity will lead the parameters. If the velocity update leads to bad loss, then the gradients will direct the update back towards $x_k$. Looking ahead helps NAG in correcting its course quicker than momentum method. Hence, the oscillations are smaller and so are the chances of escaping minima valley. The intuition behind this algorithm is that Nesterov's accelerated gradient descent performs a simple step of gradient descent to go from $x_s$ to $y_{s+1}$, and then it 'slides' a little bit further than $y_{s+1}$ in the direction given by the previous point $y_s$.

The convergence of Nesterov's accelerated gradient descent is guaranteed by the following theorem.

**Theorem 3** *Let $f$ be a convex and $\beta-$smooth function, then Nesterov's accelerated gradient descent satisfies* $f(x_t) - f(x^*) \leq \frac{2\beta\|x_1 - x^*\|^2}{t^2}$

Nesterov's accelerated gradient descent achieves the optimal convergence rate of $\mathcal{O}(\frac{1}{t^2})$.

# 4 Implementation Examples

As Nesterov's accelerated gradient descent is mostly used in Neural Network model, we are hard to find an implementataion directly based on that. So, our implementation will use Subgradient descent and based on a real-world problem. [G C06]

This implementation examples will focus on the construction of an Index fund. An index fund is a portfolio designed to track the movement of the market as a whole or some selected broad market segment. With the development of the economic theories and the empirical performance in the past decades, people has realized that Index Fund is a good way to do the investment.

The most traditional strategies for forming index funds involves choosing a broad market index as a proxy for an entire market, e.g. the Standard and Poor list of 500 stocks (S&P500). A pure indexing approach consists in purchasing all the issues in the index, with the same exact weights as in the index. In most instances, this approach is impractical (many small positions) and expensive (rebalancing costs may be incurred frequently).

An index fund with $q$ stocks, where $q$ is substantially smaller than the size $n$ of the target population seems desirable. We propose a large-scale deterministic model for aggregating a broad market index of stocks into a smaller and more manageable index fund. This approach will produce a portfolio that closely replicates the underlying market population.

We present a model that clusters the assets into groups of similar assets and selects one representative asset from each group to be included in the index fund portfolio.

## 4.1 Index Fund Construction

### 4.1.1 Model Building

We will build this index fund based on the correlation of the stock's daily return, defined as $\rho_{ij}$. Then the original optimization model is:

$$
\begin{aligned}
Z = \max \quad & \sum_{i=1}^{n}\sum_{j=1}^{n} \rho_{ij} x_{ij} \\
\text{subject to} \quad & \sum_{j=1}^{n} y_j = q \\
& \sum_{j=1}^{n} x_{ij} = 1 \quad \text{for } i = 1, \ldots, n \\
& x_{ij} \le y_j \quad \text{for } i = 1, \ldots, n; j = 1, \ldots, n \\
& x_{ij}, y_j = 0 \text{ or } 1 \quad \text{for } i = 1, \ldots, n; j = 1, \ldots, n.
\end{aligned}
$$

Here, variables $y_j$ describe which stocks $j$ are in the index fund ($y_j = 1$ if $j$ is selected in the fund, 0 otherwise). For each stock $i = 1, \ldots, n$, the variable $x_{ij}$ indicates which stock $j$ in the index fund is most similar to $i$ ($x_{ij} = 1$ if $j$ is the most similar stock in the index fund, 0 otherwise).

And for the constrains, we have that the first constraint selects $q$ stocks in the fund. The second constraint imposes that each stock $i$ has exactly one representative stock $j$ in the fund. The third constraint guarantees that stock $i$ can be represented by stock $j$ only if $j$ is in the fund. The objective of the model maximizes the similarity between the $n$ stocks and their representatives in the fund.

After we got the selected $q$ stocks, each selected stock will represent some stocks in the original index. The next thing we do is to assign the weight of the represented stocks in the original index to our corresponding selected stocks. In this way, we got the weight to build our simulated index.

Branch-and-bound is a natural candidate for solving our original model. However, the formulation can be very large. (e.g., for the S&P 500, there are 250,000 variables $x_{ij}$ and 250,000 constraints $x_{ij} \le y_j$. So the linear programming relaxation needed to get upper bounds in the branch-and-bound) algorithm is a very large linear program to solve. Instead, we can use Lagrangian relaxation and Subgradient descent method to solve it.

### 4.1.2 Optimization

After using the Lagrangian relaxation, the original model becomes for any vector $u = (u_1, \ldots, u_n)$ The objective function $L(u)$ may be equivalently stated as

$$
L(u) = \max \sum_{i=1}^{n}\sum_{j=1}^{n} (\rho_{ij} - u_i)\, x_{ij} + \sum_{i=1}^{n} u_i
$$

Let

$$
(\rho_{ij} - u_i)^+ = \begin{cases} (\rho_{ij} - u_i) & \text{if } \rho_{ij} - u_i > 0 \\ 0 & \text{otherwise} \end{cases}
$$

and

$$
C_j = \sum_{i=1}^{n} (\rho_{ij} - u_i)^+
$$

Then our model under Lagrangian relaxation is

$$L(u) = \max \sum_{j=1}^{n} C_j y_j + \sum_{i=1}^{n} u_i$$

$$\text{subject to } \sum_{j=1}^{n} y_j = q$$

$$y_j = 0 \text{ or } 1 \text{ for } j = 1, \ldots, n.$$

The set of $q$ stocks corresponding to the $q$ largest values of $C_j$ is a heuristic solution for original model. Specifically, construct an index fund containing these $q$ stocks and assign each stock $i = 1, \ldots, n$ to the most similar stock in this fund. This solution is feasible to original model, although not necessarily optimal. This heuristic solution provides a lower bound on the optimum value $Z$ of original model. As $L(u)$ provides an upper bound on $Z$. So for any vector $u$, we can compute quickly both a lower bound and an upper bound on the optimum value of $(M)$. To improve the upper bound $L(u)$, we would like to solve the nonlinear problem

$$\min L(u).$$

Since $L(u)$ is non-differentiable and convex, we use the Subgradient method. At each iteration, a revised set of Lagrange multipliers $u$ and an accompanying lower bound and upper bound to the original model are computed. The algorithm terminates when the two bounds are close enough or when a maximum number of iterations is reached.

Our Subgradient descent to do the optimization can be described as following.

**Algorithm 1** Framework of Sub-gradient Descent for Index Fund

**Input:** initial guess $u^0$; step size $\alpha$, stopping criteria $\epsilon$;
**Output:** Minimized Value of $L(u)$;

1: Set $k \leftarrow 0$, and then iterate until a stopping criteria is met;
2: Determine a Sub-gradient $s^k$ to $L$ at $u^k$, in our case, finding a subgradient is easy $s_i^k = 1 - \sum_{j=1}^{n} x_{ij}$;
3: For a given step size $\alpha^k$, set $u^{k+1} \leftarrow u^k + \alpha^k s^k$;
4: Calculate Lower Bound $Z^{k+1}$ and Upper Bound $L(u^{k+1})$;
5: If differnect between $L(u^{k+1})$ and $Z^{k+1}$ greater than $\epsilon$, $k = k + 1$, repeat steps 2 to 4, else go to step 6;
6: **return** $L(u^*)$ and $u^*$, here $^*$ denotes the optimal;

Next in our example, we chose to use the Dow Jones Industrial Average Index(DJI) as our market index, because it only contains 30 stocks, this amount of data is suitable for our computation limit. And we choose to use 10 stocks of them to construct our simulated Index for DJI. We use data of the component stocks of DJI from 2012-01-01 to 2021-05-30 to calculate the correlation, choose the stocks and pair the weights. After that, we use the data of DJI and chosen stocks from 2021-06-01 to 2021-9-30 to do the test of our simulated Index to compare our simulated Index's trend and DJI's real trend.

In Figure 1, we can see the error of our optimal problem converges to 0 with the iteration increases.

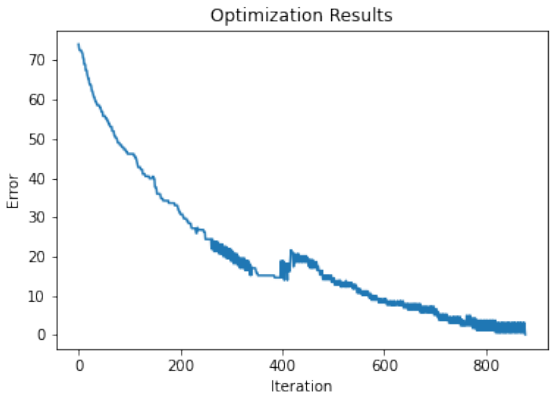


**Figure 1:** Optimization Result of Sub-gradient Descent

And in Figure 2, we compared our simulated Index's trend with the real DJI's trend in the test period. We can see that our simulated Index basically follows the trend of real DJI. This shows that our simulated Index built based on this algorithm can perform similarly as its corresponding real market index. This shows that our algorithm can obtain the expected performance during the out-of-sample test and has high effectiveness. In practice, we can update the training set data over time and periodically update our selected stocks and their weights. In this way, we can use this algorithm to construct index funds at a lower cost than usual.

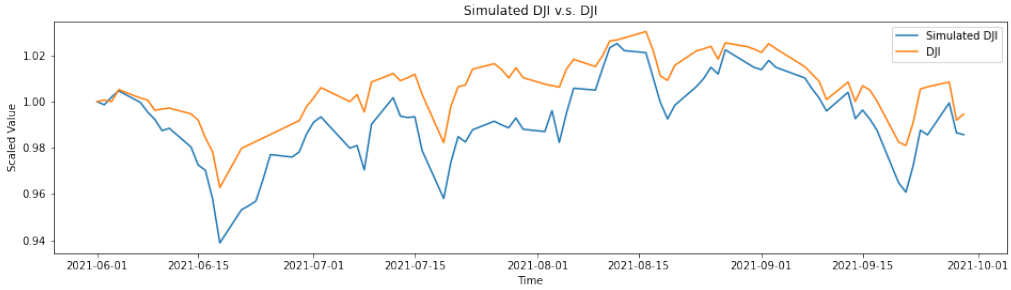


**Figure 2:** Simulated DJI Trend and DJI Trend Comparison

# 5 Conclusion

In this project, we successfully built a python package "ProNfGD" that can implement Subgradient descent and Nesterov's fast gradient method. The package will make it easier for people to utilize the two algorithms in optimization problems for variable tasks.

An example of implementing subgradient on a practical problem demonstrate the usage of the developed package. In this example, we constructed an Index fund by using only part of its corresponding market Index's components. The result of the optimization with the optimal model on the out-of-sample period shows the high effectiveness of the algorithm. Our constructed Index follows the trend of corresponding Market Index pretty well.

# 6 Appendix

# A Python Packages

The python packages is built in the way that people can use it by following the usage routine of the scipy.optimize [Sci]. Here is a brief conceived example:

```
import sys
sys.path.append("C:\\Users\\15715\\Documents\\UMICH T4\\STATS 606\\PythonPackageCode")
# Load Packages
import numpy as np
import math
import ProNFGD
#  Nesterovs  accelerated gradient method
def func(x):
    return x**2 + 2*x

ProNFGD.nesterov_descent(func, 5, 3, 0.5)
>>>-1.0003941327521175

# Subgradient Descent
def fun(x):
    return np.abs(x)

def subgradient_abs_stoc(x_single):
    if x_single > 0:
        return 1
    if x_single < 0:
        return -1

ProNFGD.run_subgradient_descent(1, fun, subgradient_abs_stoc, 0.001, 1)
>>>-8.205242041370298e-16
```

We can see that from above example, our package successfully got the minimum points of the fucntions.

# References


[al] Stephen Boyd et al. *Subgradient Methods*. URL: https://web.stanford.edu/class/ee392o/subgrad_method.pdf.

[DD18] Damek Davis and Dmitriy Drusvyatskiy. "Stochastic subgradient method converges at the rate O(k 1/4 ) on weakly convex functions". In: (2018).

[G C06] R Tütüncü G Cornuéjols J F Peña. *Optimization Method in Finance*. Cambridge University Press, 2006.

[Loi+21] Nicolas Loizou et al. "Stochastic Polyak Step-size for SGD: An Adaptive Learning Rate for Fast Convergence". In: 130 (2021).

[Pol64] B. T. Polyak. "Some methods of speeding up the convergence of iteration methods". In: *USSR Computational Mathematics and Mathematical Physics* 4 (5 1964), pp. 1–17. ISSN: 00415553. DOI: 10.1016/0041-5553(64)90137-5.

[Rud16] Sebastian Ruder. “An overview of gradient descent optimization algorithms”. In: (Sept. 2016). URL: http://arxiv.org/abs/1609.04747.

[Sci] Scipy. *Scipy Document 1.8.0*. https://docs.scipy.org/doc/scipy/reference/optimize.html.