MATH 470: Numerical Optimization and Convex Optimization with Sparsity

Zijian Pei

December 9, 2021

Contents

1 Abstraction and Description

This paper reviews what I accomplished for the course MATH470. The content of this course is Numerical Optimization and Convex Optimization, which is practical and useful for machine learning. The whole paper is divided into four major sections. The first part is Preparation, in which I included part of my learning notes based on both online sources and the book Numerical Optimization [4] given by Prof. Sahir Bhatnagar. This section includes basic knowledge I summarized that is important to understand the overall content of this paper. The second part is Numerical Optimization, which is also largely based on the book Numerical Optimization [4]. This section includes some classical optimization algorithms and techniques, such as ideas of controlling step length and using loops to update parameters, that serve as prerequisites for the next section Convex Optimization with Sparsity. In the second section, I also run simulations, and compared the outputs of different classical algorithms. Without understanding of these fundamentals, it will be extremely hard to get the idea of Convex Optimization. In the Convex Optimization with Sparsity section, a lot of content incorporates the idea from Optimization with Sparsity-Inducing Penalties [1] and Carnegie Mellon University's online lectures. [6] Indeed, my notes summarized a dramatic number of formulas and equations about different kinds of sparsity-inducing penalty functions, including typical lasso, group lasso, and overlapping-group penalty functions. However, I decided to put only a small subset of them to make the whole report nice and clear, as most of them are not considered as indispensable for my whole research project. For the last section, I originally planned to design my own optimization algorithm, but later inspired by Prof. Sahir Bhatnagar, I switched to running simulations for different convex optimization algorithms I learned. The last section is correlated with Optimization with Sparsity, and procedures of implementation of the algorithms used in the last section is presented in the second last section.

2 Preparation

2.1 Some fundamental concepts

All the techniques and methods used in this report are restricted to convex optimization, in which a convex function is examined. During the optimization process, we introduce the parameter β as an estimator for the minimizer of the convex function. Besides that, some background of the following terms is required to understand different optimization algorithms implemented in this report.[2]

- Convex Sets: A set $S \in \mathbb{R}^n$ is convex if for $x \in S$ and $y \in S$, we have $tx + (1-t)y \in S$ for all $\alpha \in [0,1]$.
- Convex functions: A function f is convex if $f(tx + (1-t)y) \le tf(x) + (1-t)f(y)$, and $dom(f)$ convex.
- Optimization problem: Minimize $f(x)$, where $f(x)$ convex, subject to some restrictions.
- Hessian: Second-order partial derivative of some $f : \mathbb{R}^n \longrightarrow \mathbb{R}$, usually in a matrix form.
- Soft-thresholding: Simplified lasso with $X = I$, $S_\lambda(x) = \underset{\beta}{min}\, \frac{1}{2}\|x-\beta\|_2^2 + \lambda\|\beta\|_1$

2.2 Some techniques used in algorithms

Some algorithms listed in this report incorporate the principal of gradient descent, which is the very basic optimization algorithm. In addition to that, one useful technique called backtracking line-search is also essential to some algorithms in which the step length needs to be updated. [6]

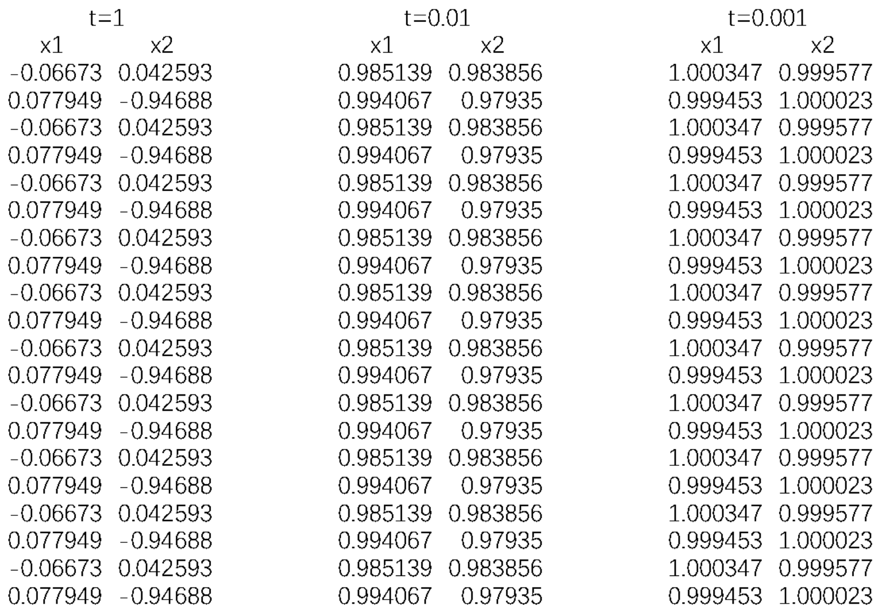
- Gradient method
  Consider unconstrained smooth convex optimization problem: $\underset{x}{min}\, f(x)$
  Choose initial point $x^{(0)} \in \mathbb{R}^n$, repeat: $x^{(k)} = x^{(k-1)} - t_k \bigtriangledown f(x^{(k-1)})$
  Second-order Taylor expansion:$f(y) = f(x) + \bigtriangledown f(x)^T(y-x) + \frac{1}{2}(y-x) \bigtriangledown^2 f(x)(y-x)$.
  Replace hessian by $\frac{1}{t}I$: $f(y) = f(x) + \bigtriangledown f(x)^T(y-x) + \frac{1}{2t}\|y-x\|_2^2$. This is called quadratic approximation.
- Backtracking line-search (Useful for updating step length)
  First fix parameter $0 < \beta < 1$, and $0 < \alpha \le 1/2$. Then at each iteration, start with $t = t_{init}$, and while $f(x - t \bigtriangledown f(x)) > f(x) - \alpha t\|\bigtriangledown f(x)\|_2^2$, shrink $t = \beta t$, else perform gradient descent update.

3 Numerical Optimization

The first half of this research project is based largely on the book *Numerical Optimization* by Jorge Nocedal and Stephen J.Wright, in which the fundamentals of numerical optimization is presented. A large number of regression methods are included in this book, and I managed to implement some of them and compared their performance. Meanwhile, some background knowledge from this book is also essential to understand some details in this report, such as matrix calculus and basic mathematical analysis.

3.1 Steepest descent

One may have noticed that steepest descent is exactly the same as simple gradient method, but with a different name. In steepest descent, the direction is guaranteed to be descent, because it computes the direction at each iteration by just taking the negative value of the gradient at the estimated parameter-value updated from the last iteration. If this is not intuitive, one can test the inner product of the direction and the gradient. Since there are only two variables need to be computed in one iteration, the only concern left should be the step length. Therefore, it is reasonable to assume that the overall performance of steepest descent is highly affected by the value of the step length. In general, it is believed that if a step-length-update method is not induced for steepest descent, the function value would diverge after some certain iterations when it reaches a point where such great step length is too big to meet the descent criteria. To verify this, I wrote a simple program to compute the optimal parameter-value by assigning different values to the initial step length and having them fixed. To visualize the assumption, I use parameter values got from the last 20 out of 20000 iterations under the setting of Rosenbrock function. [4]
Rosenbrock function: $f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$ with optimal value at (1.0, 1.0), and initial point at (1.2, 1.2)



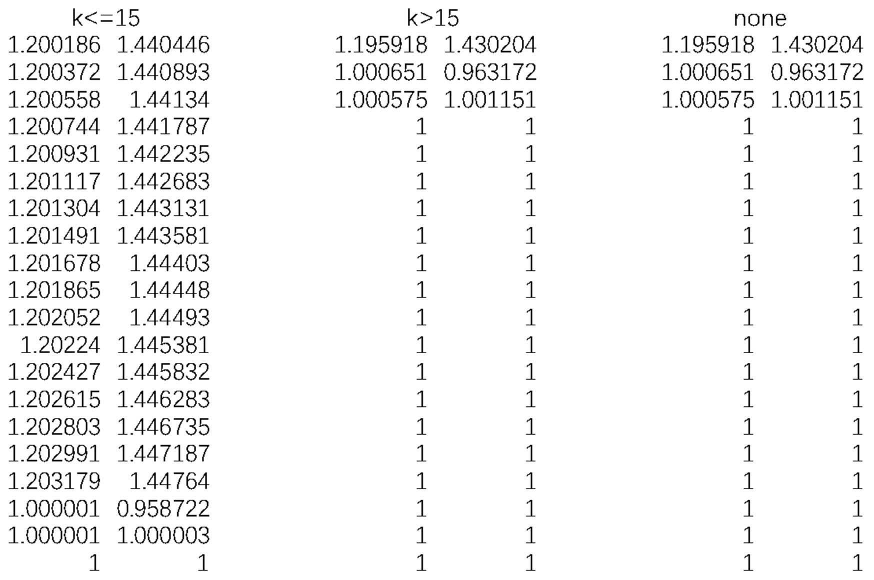
t=1		t=0.01		t=0.001	
x1	x2	x1	x2	x1	x2
-0.06673	0.042593	0.985139	0.983856	1.000347	0.999577
0.077949	-0.94688	0.994067	0.97935	0.999453	1.000023
-0.06673	0.042593	0.985139	0.983856	1.000347	0.999577
0.077949	-0.94688	0.994067	0.97935	0.999453	1.000023
-0.06673	0.042593	0.985139	0.983856	1.000347	0.999577
0.077949	-0.94688	0.994067	0.97935	0.999453	1.000023
-0.06673	0.042593	0.985139	0.983856	1.000347	0.999577
0.077949	-0.94688	0.994067	0.97935	0.999453	1.000023
-0.06673	0.042593	0.985139	0.983856	1.000347	0.999577
0.077949	-0.94688	0.994067	0.97935	0.999453	1.000023
-0.06673	0.042593	0.985139	0.983856	1.000347	0.999577
0.077949	-0.94688	0.994067	0.97935	0.999453	1.000023
-0.06673	0.042593	0.985139	0.983856	1.000347	0.999577
0.077949	-0.94688	0.994067	0.97935	0.999453	1.000023
-0.06673	0.042593	0.985139	0.983856	1.000347	0.999577
0.077949	-0.94688	0.994067	0.97935	0.999453	1.000023
-0.06673	0.042593	0.985139	0.983856	1.000347	0.999577
0.077949	-0.94688	0.994067	0.97935	0.999453	1.000023
-0.06673	0.042593	0.985139	0.983856	1.000347	0.999577
0.077949	-0.94688	0.994067	0.97935	0.999453	1.000023

Fig. 1. Parameter values for different step length

From the figure, we can tell that even for step length with value as small as 0.01, the parameter values after 20000 iterations are still not decent. In other words, 0.01 is not a suitable value after certain number of iterations. Therefore, it's important to find a way to bound the step length when steepest descent algorithm is implemented. Why bounding, instead of shrinking? Since while continuously shrinking the step length, if its value is too small, the speed of the descent process would be extremely low. This can be testified by using the same data set above. (1.0319695655907413,1.0662342714642972) is the parameter value of the case when the step length is set to 0.001 after 2000 iterations. This value is not even better than what we get after 200 iterations with step length 0.01. In addition to that, one may have realized that the step length for steepest descent can not be fixed, and it should be updated accordingly as the direction changes.

3.2 Newton's method

In Newton's method, different from steepest descent, the direction may not always be descent. This is because Newton's method uses the negative value of the product of hessian and the gradient as the direction. To secure a descent direction, we must have a positive-definite hessian, but that is not guaranteed in the computation. In my simulation, non-positive-definite hessian value did not occur in any iteration. To demonstrate the exact outcome of inducing a flawed hessian value in computation, I manually changed some components of the hessian in some iterations when implementing Newton's method. It turns out when the hessian value is not positive-definite, the function value will diverge. In addition to that, one can even directly identify that the distance between the updated parameters and the optimal value is becoming farther after each iteration. However, if some components of the hessian are changed, as long as the hessian after modification is positive-definite, Newton's method still works. I verify this by initially setting hessian to a non-positive-definite value, and later setting it to an arbitrary positive-definite value.



k<=15		k>15		none	
1.200186	1.440446	1.195918	1.430204	1.195918	1.430204
1.200372	1.440893	1.000651	0.963172	1.000651	0.963172
1.200558	1.44134	1.000575	1.001151	1.000575	1.001151
1.200744	1.441787	1	1	1	1
1.200931	1.442235	1	1	1	1
1.201117	1.442683	1	1	1	1
1.201304	1.443131	1	1	1	1
1.201491	1.443581	1	1	1	1
1.201678	1.44403	1	1	1	1
1.201865	1.44448	1	1	1	1
1.202052	1.44493	1	1	1	1
1.20224	1.445381	1	1	1	1
1.202427	1.445832	1	1	1	1
1.202615	1.446283	1	1	1	1
1.202803	1.446735	1	1	1	1
1.202991	1.447187	1	1	1	1
1.203179	1.44764	1	1	1	1
1.000001	0.958722	1	1	1	1
1.000001	1.000003	1	1	1	1
1	1	1	1	1	1

Fig. 2. Parameter values under modified B_k with different iterations

This should be an interesting question, is hessian not important in Newton's method? Can any square matrix that shares the same size with hessian but positive definite serve as a perfect substitute of the hessian?
By modifying hessian, denoted B_k, with all the entries of B_k set to some large positive value, a substitute of hessian is created. I manually create different functions, including functions with two, three, and four parameters, and have them implement Newton's method with a positive-definite matrix as a substitute for the actual hessian. All of these functions show convergent features under the condition of small step length and large number of iterations. The conclusion seems to be very solid: any positive definite matrix that shares the same size with hessian can act like hessian in Newton's method. This can also be verified in a theoretical form: we may assume B_k is positive definite, and by taking the inner product of p_k and the gradient, we can get a negative value. Therefore the angle between these two vector should be in $(\frac{\pi}{2}, \pi)$, which means p_k must be a descent direction.
Another surprising finding from the comparison between the two figures is that Newton's method in general converges to the optimal value much faster than gradient descent with the same step length. My guess is that because in gradient descent, we only address the optimization problem in a linear form. However, in Newton's method, we are using quadratic approximation to examine the convex function, which I believe should be a more concise auxiliary function that mimics the true shape of the complex function. To further verify the performance of Newton's method, the same step lengths as in steepest descent are chosen. To maintain consistency, I collect the data from the last 20 out of 20000 iterations and with several different positive-definite matrix as substitutes for hessian. In fact, in all three different values of step lengths, the parameter converges to the optimal value much earlier than gradient descent.
However, Newton's method does not always behave so well as my simulation reflects, it is still sensitive to

the size of the step length. We still need to bound the step length such that it remains in a range that should guarantee decent and prevent the descent speed being too slow.

3.3 Quasi-Newton method

Unlike Newton's method, quasi-Newton method makes it unnecessary to compute hessian during each iteration, and the direction is defined as $-B_k^{-1} \nabla f_k$ as always, but in this case, B_k is not the B_k defined in Newton's method, which in Newton's method is nothing but hessian. In quasi-Newton, B_k is an approximation of hessian, and we need to manually set a positive value for B_0 first, and use SRI or BFGS formula to update B_k for the following iterations. The SRI and BFGS formulas are listed here: [4]
SRI:

$$B_{k+1} = B_k + \frac{(y_k - B_k s_k)(y_k - B_k s_k)^T}{(y_k - B_k s_k)^T s_k} \tag{1}$$

BFGS:

$$B_{k+1} = B_k - \frac{B_k s_k s_k^T B_k}{s_k^T B_k s_k} + \frac{y_k y_k^T}{y_k^T s_k} \tag{2}$$

where:

$$s_k = x_{k+1} - x_k \quad y_k = \nabla f_{k+1} - \nabla f_k \tag{3}$$

Note that, the hessian approximation computed using BFGS method is guaranteed to be positive-definite if the initial B_0 is positive definite. The proof is quite messy and complex, and its detail is stated in book $Numerical\ Optimization$ written by Jorge Nocedal and Stephen J.Wright. When we get a positive-definite B_k in BFGS, we can directly use it to compute the direction. However, in SRI, simply using $-B_k^{-1} \nabla f_k$ to get the direction is not allowed, as the hessian approximation may not always be positive-definite.
From what indicated above, it's reasonable to assume that for problems in which the second-order derivative computation is complicated or non-descent directions occur during the iterative process, the expected performance of quasi-Newton can be quite nice. Again, due to the fact that it incorporates some features of Newton's method, the convergence speed will outstrip the speed of gradient descent. Although quasi-Newton seems to be a quite nice algorithm to implement, it still faces the same problem as the above two methods——step length.

3.4 Newton's method with hessian modification

Personally, I would avoid using this method. In Newton's method with hessian modification, too much effort is put on hessian, and trying to make B_k positive-definite. Normally, it's required to check if the hessian is positive-definite before performing the parameter update. If not, then B_k is computed by adding up hessian and a matrix with positive value along the diagonal. However, based on this algorithm, it seems that B_k can not be arbitrarily decided, a complicated method call modified Cholesky Factorization is implemented to make a weird step: keep hessian unchanged if hessian is already positive definite, otherwise, add a positive-diagonal matrix to ensure B_k is sufficiently positive definite. But the definition of "sufficient" is not clearly stated either. From my perspective, this method in theory is nothing but trying to create a positive-definite B_k, and such can be easily achieved by randomly choosing a feasible matrix.

3.5 Control the step length

- Wolfe conditions [4]

  $$f(x_k + \alpha_k p_k) \le f(x_k) + c_1 \alpha_k \nabla f_k^T p_k \tag{4}$$

  $$\nabla f(x_k + \alpha_k p_k)^T p_k \ge c_2 \nabla f_k^T p_k \tag{5}$$

  with $0 < c_1 < c_2 < 1$
  The first condition is used to make step length small enough to achieve descent, and the second condition is used to prevent the step length being too small to get enough decrease.

- Strong Wolfe conditions
  With identical first condition, but second condition in an absolute value form.

- Goldstein conditions [4]

$$f(x_k) + (1-c)\alpha_k \nabla f_k^T p_k \leq f(x_k + \alpha_k p_k) \tag{6}$$

$$f(x_k + \alpha_k p_k) \leq f(x_k) + c\alpha_k \nabla f_k^T p_k \tag{7}$$

  with $0 < c < 1/2$

The above three conditions only describe the inequality that should be satisfied in the value range of the step length. The key procedure to bound the step length is missing. Personally, I tend to use backtracking line-search to find the feasible step length.

- Backtracking line-search (Description presented in Preparation)
  In backtracking line-search, the step length is used primarily to guarantee the function value of the next iteration is less than the function value of the current iteration. Note that the result does not guarantee enough decrease. Therefore, choosing a suitable value for β should be tricky. Typically, $\frac{1}{2}$ is a good choice. This is not a rigorous choice, but it works in all my simulations.

4 Convex Optimization with Sparsity

The second half of this research project is based largely on the book Optimization with Sparsity-Inducing Penalties by Francis Bash, Rodolphe Jenatton, Julien Mairal, and Guillaume Obozinski, the book The Elements of Statistical Learning by Trevor Hastie, Robert Tibshirani, and Jerome Friedman, and cmu's convex optimization online lectures, from which I learned the fundamentals of convex optimization and the idea of a inducing a penalty function, usually in a Euclidean norm form. A penalty function is added up to the normal convex function to control the behavior of the parameter. For example, lasso regression with penalty function as a coefficient with l_1 norm, usually results in zero value in some unimportant parameters. While for ridge regression, the unimportant components are never equal to zero but reduced to very small values.

4.1 Proximal gradient method

[3] This algorithm is particularly designed to optimize problems with one smooth term, and one non-smooth term. A smooth term combined with a non-smooth term, such as l_1 norm, makes the problem impossible to be solved by just taking the gradient and iteratively updating the value of x, since the whole function we need to examine is not always differentiable. In such problems, it is a good option to separately consider the smooth term and the non-smooth term. For the smooth term, we incorporate the idea of gradient descent; for the combination part, we introduce a concept called proximal operator, which is equivalent to finding the minimizer of the sum of the smooth and the non-smooth term. The vector we get after taking the proximal operator can be used as the parameter for the next iteration.

Steps of proximal method [1]:

- Step 1: Quadratically approximate the smooth term g, and use the method of gradient descent to update x, and we may call the updated value x^+
- Step 2: The above x^+ serves as an intermediate. We need to further compute the proximal operator of x^+.

Steps written in the form of formulas, with h as the penalty function:

$$\begin{aligned}
x^{++} &= \underset{z}{argmin}\, \overline{g}(z) + h(z) \\
&= \underset{z}{argmin}\, g(x) + \nabla g(x)^T (z-x) + \frac{1}{2t}||z-x||_2^2 + h(z) \\
&= \underset{z}{argmin}\, \frac{1}{2t}||z-(x-t\nabla g(x))||_2^2 + h(z) \\
&== prox_t(x^+)
\end{aligned}$$

4.2 Subgradient method

Subgradient can be easily understood as the slope of the tangent line that intersects the convex function at the current parameter point. Usually for continuous and differentiable function, it has only one subgradient at one certain point. But for function differentiable and not continuous, then at the sharp corner, any value between two sided limits can be the subgradient. And subdifferential is the collection of all subgradients at some certain point. Subgradient function updates parameters exactly like simple gradient method, but its direction is some element in the subdifferential of the convex function. Subgradient method is one of the most unique convex optimization methods I learned. It does not require any update of the step length, but with only an initially fixed step length. Although it looks quite simple, the step to find an appropriate subgradient is not always easy. The simplest example I came up with is l_1 norm with only one parameter. The shape of this function should be like "V". However, if with initial point at origin, then any gradient between $[-1, 1]$ can be the subgradient of this function. Meanwhile, with a non-zero step length, the function will diverge. Therefore, it's reasonable to assume that the condition to implement subgradient method is very strict. Everything works fine if the function is convex and smooth. But for function that is convex and non-differentiable at some points, such as lasso regression, this algorithm would sometimes produce increasing value in the loss function. As a result of that, designing a method to address non-differentiable points might be a good choice. That is something I originally tried to do, but did not due to limited amount of time.

4.3 Coordinate descent

In coordinate descent, an initial $x^{(0)}$ is chosen, and iteratively update $x^{(k)}$ for k=1,2,3... Coordinate descent is one of the easiest algorithm to be implemented, but it takes more computation in each step. Theoretically, the time complexity for coordinate descent is very high, as for the same number of iterations, coordinate descent needs to update each parameter once before completing the general concept of "a whole iteration". Therefore, a formula for computing the specific minimizer is always required to lower down the computation complexity of this method. [6]

$$
\begin{aligned}
&x_1^{(k)} \in argmin_{x_1} f(x_1, x_2^{(k-1)}, x_3^{(k-1)}, ..., x_n^{(k-1)})\\
&x_2^{(k)} \in argmin_{x_2} f(x_1^{(k)}, x_2, x_3^{(k-1)}, ..., x_n^{(k-1)})\\
&x_3^{(k)} \in argmin_{x_3} f(x_1^{(k)}, x_2^{(k)}, x_3, ..., x_n^{(k-1)})\\
&.........\\
&x_n^{(k)} \in argmin_{x_n} f(x_1^{(k)}, x_2^{(k)}, x_3^{(k)}, ..., x_n)
\end{aligned}
$$

5 Efficiency Measurement for Different Convex Optimization Algorithms

5.1 Description

Suggested by Prof. Sahir Bhatnagar, I decided to run a simulation that reflects the efficiency and the performance of different convex optimization algorithms. The initial goal was to compare three algorithms, including Nestrov's method, Proximal method, and Coordinate descent. Due to time limits, I decided to cut off Nestrov's method in the final simulation, and replaced it with Proximal method without step length update to show how the step length update functions will affect the computation complexity of the original method. The simulation up to now compares Proximal method, Proximal method without step length update, and Coordinate descent. Initial parameters and approximation are generated with R function provided by Ryan Tibshirani on GitHub. [5] The generator function is called sim.xy. It should have seven input variables, and we only concern some of them. The number of training points, the number of parameters, number of validation observations, number of non-zero parameters, parameter types, and signal-to-noise ratio. This function generates a list of information, and we need only some of them: The initial parameter β, predictor matrix x, and response vector y. β is generated with a repeated sequence with the number of non-zero parameters, but with 1 first, and 0 the others when the type is set to 1. I used type=1 for all the simulations. For example, if the number of non-zero parameters is 5, and the number of parameters is 10. Then the initial β should be: {1,0,0,0,0,1,0,0,0,0}. The predictor matrix is generated using rnorm() function, which is a matrix generator function that follows a random

normal distribution, followed by an auto-correlation. The number of rows is equal to the number of test data points, and the number of columns is equal to the number of parameters. The response vector y is the product of approximation matrix and the initial parameter plus a random noise. More details can be found here. [5]
For the assumption part, I suppose that due to the large number of valid parameters, the shape for the true convex function would show some weird behavior. In other words, it's hard to keep a balance between the step length and the direction. Therefore, what I originally thought was the behavior of Proximal method without update of step length should be unpredictable. Meanwhile, the performance of Proximal method with backtracking line-search should be steady. As for coordinate descent, I suppose the square loss will converge to some value, since the continuous update of each component of the parameters will eventually approaches its limit.
The simulation was conducted with lasso regression, which is easy to implement and ideal to show the features of different algorithms.

5.2 Computations and outputs

Before writing the body of different algorithms, I decided to simplify the computation step as much as I can to lower down the computation complexity and reduce the time cost. To achieve that, I pre-computed the gradient of the smooth-term of lasso regression and adopt the formula for soft-thresholding, which are two essential components for both Proximal method and Coordinate descent.
After the preparation step, with the training data generated, I wrote two functions for Proximal method with step length update. The step length update function takes two inputs, one for the current step length, and one for the coefficient update. Unless the descent criteria is met for the quadratic approximation function, the function will continue updating the step length with backtracking line-search algorithm. To simplify the computation, I transform the complex procedure that includes proximal operator to a more concise form such that it looks similar to the normal gradient method: $x^{k+1} = x^k + G(x)$, with $G(x)$ also included in step length verification part, to further reduce the computation complexity. Then before doing each iteration, an step length verification procedure is conducted to check if the current step length would yield a decreasing output for the quadratic approximation of the convex function. If not, then the step length update function is implemented. After getting the feasible step length, we can directly compute the next parameter of the quadratic approximation function, and use this result as input for the soft-thresholding function to get the parameters for the next iteration. Similarly, in coordinate descent, I computed the formula for the intermediate factor, which later used as the input for soft-thresholding function to lower down the computation complexity, specifically to reduce the computation complexity within each iteration. This is not considered as a preparation part because I realized after I successfully implemented Proximal method.
The performance of each algorithm is measured by a log-subtraction function, which has the square loss of each iteration subtracted with the square loss got in the last iteration. We may denote this value for each iteration as "log Square Loss Subtraction". This function is useful to check if an algorithm would diverge at some point. However, it's not perfect, as it can not distinguish between divergence and convergence. For example, if the square loss is steady from the 80^{th} iteration, then subtracting this value by the very end square loss would equal to zero. Take log with respect to zero would get a NA, which is the same as taking log of a negative value. Therefore, the following plots must be analysed together with a manual check for square loss to see if the algorithm actually converges. After doing that, we can distinguish divergent and convergent parts of each curve. Combined with close check on x-axis, we can tell which algorithm converges faster. Still, to measure the performance of each algorithm on reducing the whole convex function, we again have to manually check the square loss for each iteration.

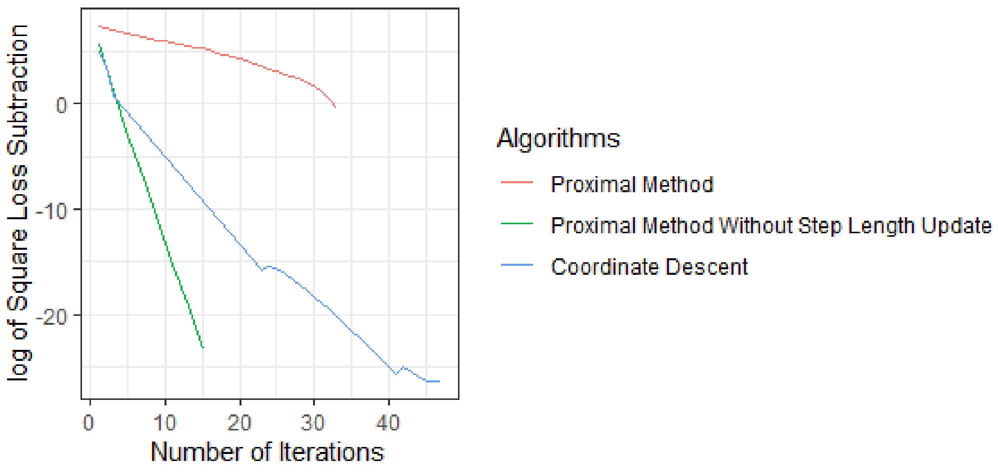


Fig. 3. 20 parameters and 80 training points

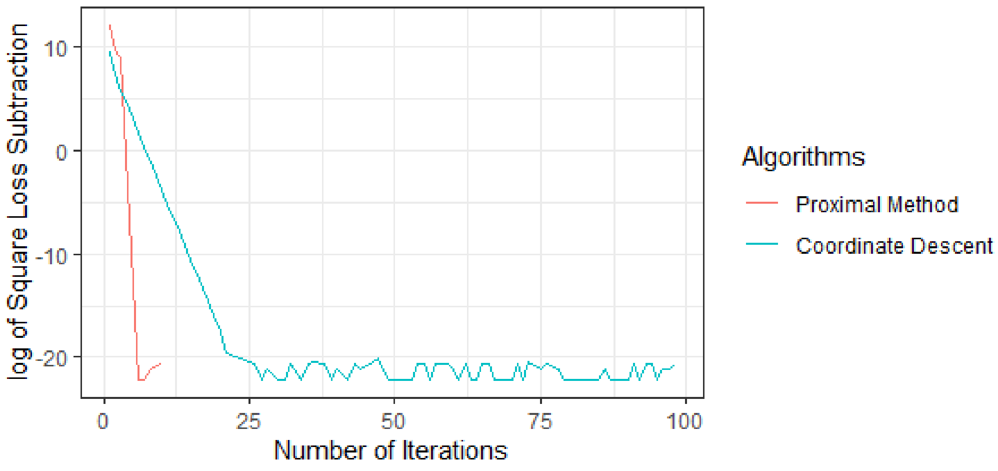


Fig. 4. 100 parameters and 1000 training points

5.3 Conclusion

For the first simulation, all three algorithms converge, Proximal method without step length update converge much faster than the other two algorithms. And the normal Proximal method converges slowest compared to the other two. As for the square loss, Proximal method converges to 7758.528, Proximal method without update of step length converges to 9823.093, and Coordinate descent converges to 7803.017. We can tell that Proximal method and Coordinate descent have similar performance on reducing the value of the convex functions, in other words, finding the minimizer of the function.
For the second simulation, only two algorithms converge. The curve for Proximal method without step length update is missing. It's understandable that the missing one is Proximal method without update of step length, since the direction is not ensured to be descent. Here's some data set for the first 5 iterations, and last 5 iterations for this flawed algorithm: 1.526073e+06 1.358600e+06 1.566260e+06 1.437315e+06 1.789794e+06 and 2.241477e+45 6.301690e+45 1.771673e+46 4.980975e+46 1.400383e+47. In fact, the square loss increases all the time, and this algorithm seems like doing negative work on optimization. As for Proximal method and Coordinate descent, one converges to 1343200, and the other 1343255, still very close. In this case, Proximal method converges much faster than Coordinate descent.
Based on the simulation, without consideration of gradient descent without update of step length, it's clear that in large number optimization problems, Proximal method does much better work than Coordinate descent, vise versa for small number's problems.
For the accuracy part, lambda values of two simulations are chosen using cross validation with R function cv.glmnet(). For the first simulation, I set 5 non-zero parameters first. And for all three algorithms, two out

of three algorithms have no non-zero elements, which is very strange. Only Proximal Method has one zero in the final output. Therefore I decided to change the number of non-zero coefficients in the process of data generation. With 0, 5, 10, 15 as the number of non-zero coefficients the final output should have, I got only one zero in 5, one zero in 15 for Proximal Method; and one zero in 10 for Proximal Method without step length update. I also tested the output for the second simulation, and only Proximal Method can sometimes have 1 or 2 non-zero parameters. I manually changed the value of lambda to see if this outcome can be attributed to my "bad" lambda choice. In fact, after my change to lambda, the final square loss value shows a increasing trend. This indicates that for all three algorithms, accuracy can not be guaranteed. However, there are indeed some very small values (<0.01) occurring in the first simulation for all three methods, and for two algorithms except for Proximal Method without step length update in the second simulation. This shows that Lasso regression is also useful to reduce the value of some parameters while the accuracy is affected by different implementations of algorithms.
For the computation time, I select the last 10 entries for each method with 1000 test data and 100 parameters.

- Proximal method:

1.04386117	1.05681505	1.06886185	1.08333000	1.09583403
1.10817613	1.12033850	1.13222137	1.14464737	1.15694680

- Proximal method without step length update:

0.096545935	0.096545935	0.096545935	0.096545935	0.096545935
0.096545935	0.096545935	0.096545935	0.096545935	0.096545935

- Coordinate descent:

7.96554017	8.05207205	8.11604404	8.20905113	8.30013514
8.38402915	8.46812010	8.54983211	8.61703515	8.69992208

The above data set shows that Proximal method generally performs much better than Coordinate descent in large scale optimization problems, even in the case when each component that needs to be updated in Coordinate descent is generalized into a simple formula.

References

[1] Francis Bach et al. “Optimization with sparsity-inducing penalties”. In: *arXiv preprint arXiv:1108.0775* (2011).

[2] Jerome Friedman, Trevor Hastie, Robert Tibshirani, et al. *The elements of statistical learning*. Vol. 1. 10. Springer series in statistics New York, 2001.

[3] Trevor Hastie, Robert Tibshirani, and Martin Wainwright. *Statistical learning with sparsity: the lasso and generalizations*. Chapman and Hall/CRC, 2016.

[4] Jorge Nocedal and Stephen Wright. *Numerical optimization*. Springer Science & Business Media, 2006.

[5] Ryan Tibshirani. *best-subset/bestsubset/R/sim.R*. Website. https://github.com/ryantibs/best-subset/blob/master/bestsubset/R/sim.R. 2021.

[6] Ryan Tibshirani. *Convex Optimization: Fall 2019*. Website. https://www.stat.cmu.edu/~ryantibs/convexopt/. 2019.