Large scale unconstrained optimization

Introduction

This project aims at comparing the performance of several numerical methods in large scale, unconstrained convex optimization problems, ‘large scale’ typically refers to solving optimization problems with thousands or even more variables. In such setting, conventional methods such as Newton’s method might either take too long, or require too much memory to converge. More efficient methods is imperative to give reasonable performance in solving large scale problems.

Problem formulation

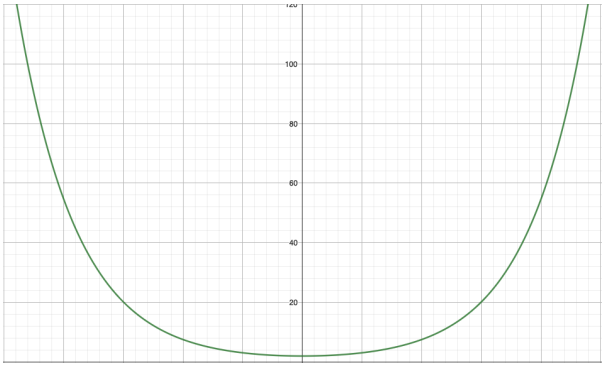
The objective of this project is to minimize the function:

$$f(\mathbf{x}) = \sum_{i=1}^{n} e^{x_i/\alpha_i} + e^{-x_i/\alpha_i} \qquad \mathbf{x} \in \mathbb{R}^n$$

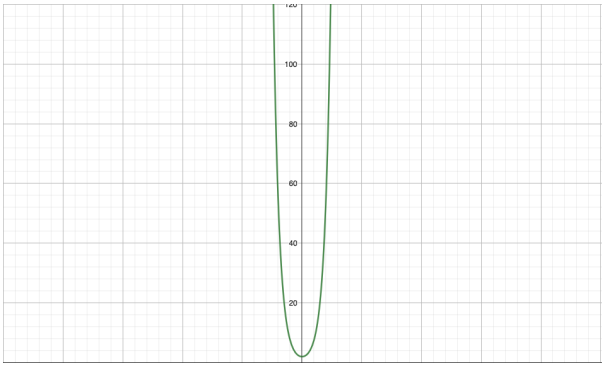
$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{1}{\alpha_1}(e^{x_1/\alpha_1} - e^{-x_1/\alpha_1}) \\ \vdots \\ \frac{1}{\alpha_n}(e^{x_n/\alpha_n} - e^{-x_n/\alpha_n}) \end{bmatrix}, \qquad \nabla^2 f(\mathbf{x}) = \mathbf{diag}(\frac{1}{\alpha_i^2}(e^{x_i/\alpha_i} + e^{-x_i/\alpha_i}))$$

where α is the parameter that controls the convexity. We shall investigate the performance of methods under ideal conditions (condition number=1), ill-conditioned, and flat curvature.

e.g. flat curvature



e.g. ill-conditioned



The optimal solution would be $\mathbf{x} = 0$, with optimal cost $= 2n \qquad (x_i = 0 \quad \forall i = 1, ..., n)$

Notations

The methods presented below are largely referenced from Nocedal and Wright (2006), to prevent confusion, we follow the notations used in the book. Some common ones are:

- x_k: k^{th} iterate for a particular method
- f_k: the function evaluated at the k^{th} iterate, i.e. $f_k := f(x_k)$ to simplify notations. Likewise, $\nabla f_k := \nabla f(x_k)$ and $\nabla^2 f_k := \nabla^2 f(x_k)$

Other specific notations shall be explicitly defined by the time it is introduced.

In addition, merely copying and pasting proof or algorithms does not add value to the discussion, therefore the following sections focus on higher level intuition, pros and cons, or motivations for the method. Nevertheless, exact page or section from the book for algorithms, proofs, or theorems would be marked in red for reference, such as (Algorithm 4.2), or (p. 123-124).

Background

The method of interest of this project are:

1. Steepest descent
2. Newton's method
3. BFGS method
4. L-BFGS method
5. Conjugate gradient (CG) methods (nonlinear)
6. Line search newton CG method
7. Partially separable functions

1. **Steepest descent**

   One of the oldest and simplest method. This method simply follows the negative gradient at each iteration, namely:

   $$x_{k+1} = x_k - \alpha_k \nabla f_k$$

   where α_k is the line search parameter. Despite its simplicity, under certain assumptions, namely α_k being the exact line search parameter, it is guaranteed to globally converge which is one of its strengths.

   However, steepest descent has the notorious 'zig-zag' path that hinders its convergence rate. To see why, assuming that α_k being the exact line search parameter, i.e.

   $$\alpha_k = \arg\min_{\alpha} f(x_k - \alpha \nabla f_k)$$

   we take the derivative of $f(x_k - \alpha \nabla f_k)$ with respect to α and obtain:

$$\begin{aligned}\frac{d}{d\alpha} f(x_k - \alpha \nabla f_k)|_{\alpha=\alpha_k} &= \nabla f(x_k - \alpha_k \nabla f_k)^T \nabla f_k = 0 \\ &= \nabla f_{k+1}^T \nabla f_k = 0\end{aligned}$$

In other words, the successive descent directions are orthogonal. As a result, it can only achieve linear convergence rate (Theorem 3.3).

2. **Newton's method**

   In addition to the 'zig-zag' path of steepest descent, being a first-order method inherently undermines the convergence rate of steepest descent. In contrast, Newton's method utilizes second order information to obtain significantly superior convergence performance, i.e. quadratically convergent (Theorem 3.5).

   Instead of attempting to optimize the actual objective function, Newton's method first approximate the local curvature with a model m_k, by second order taylor series:

   $$m_k(p) = f(x_k + p) \approx f_k + \nabla f_k^T p + \frac{1}{2} p^T \nabla^2 f_k p$$

   In addition to the gradient information we have in first order methods, we now also posses second order information, which can be interpreted as the curvature at current iterate. This way, the update can be thought as being 'far-sighted' rather than the myopic, greedy steepest descent. Depending on the function, steepest descent does not necessarily point towards the global optimal.

   It then computes the direction p_k that optimizes the model, i.e.

   $$\nabla m_k(p_k) = \nabla f_k + \nabla^2 f_k p_k = 0$$

   we can finally obtain the analytic solution of the descent direction

   $$p_k = -\nabla^2 f_k^{-1} \nabla f_k$$

   Since Newton's method is basically a second order approximation of actual objective function, it is identical to the objective function when it happens to also be a convex quadratic function, in fact it converges to the optimal solution with one step regardless of initial position.

   As powerful as it may seem, Newton's method do suffer from some major drawbacks. Generally speaking, second order approximations are not necessarily good approximations to the local curvature of the objective function, in a sense that the paraboloid might not represent the genuine curvature information at all. Consider a 'golf-course' like contour (Figure 4.1), particular iterations would lead to only little reduction in the cost and as a consequence, more steps are required for convergence. Fortunately, this can be easily mitigated by say, imposing trust region restrictions. Another obvious drawback is that the hessian matrix has to be inverted to compute the descent direction at each iteration. Although, practically speaking, one can instead solve the systems of equations $\nabla^2 f_k p_k = -\nabla f_k$, this operation is still unpleasant, especially as the number of variables grows.

3. **BFGS method (Chapter 6.1)**

   Quasi-Newton methods is a class of methods that mimic Newton's method, but without using the actual hessian matrix. By relaxing the requirement of computing the hessian at every iteration, it allows wide range of variants to approximate the hessian in a computationally cheaper fashion. After all, most problems do not converge in one single step, so exactness is not necessary in every iteration as long as the approximation is reasonable.

   The BFGS method is one of the popular methods in the Quasi-Newton method class. Just like Newton's method, we start off by approximating the local curvature with second order taylor series:

$$m_k(p) = f(x_k + p) \approx f_k + \nabla f_k^T p + \frac{1}{2} p^T B_k p$$

   but this time, we use an approximated hessian B_k instead of the exact $\nabla^2 f_k$.

   The heart of BFGS is how a 'reasonable' approximation of hessian is defined. We require the gradient of the model should match the actual gradient, i.e.:

$$\nabla m_{k+1}(-\alpha_k p_k) = \nabla m_k(0) = \nabla f(x_k)$$

   Simply put, when the solution is updated to the new iterate, if we backtrack to the last iterate we could still recover the same gradient. Intuitively speaking, the model is no longer as accurate as taylor approximation, but the model could still represent the local first order information. Following the above equality, we get:

$$\begin{aligned} \nabla m_{k+1}(-\alpha_k p_k) = \nabla f_{k+1} - \alpha_k B_{k+1} p_k &= \nabla f_k \\ \alpha_k B_{k+1} p_k &= \nabla f_{k+1} - \nabla f_k \\ B_{k+1} s_k &= y_k \end{aligned}$$

$$\text{where} \qquad s_k = \alpha_k p_k = x_{k+1} - x_k \qquad y_k = \nabla f_{k+1} - \nabla f_k$$

   the last equality is also known as the secant equation.

   Now that we know the approximated hessian B_{k+1} should satisfy the secant equation, but solving for the $n \times n$ variables does not neccesarily guarantee uniqueness of B_{k+1}. Fortunately, we can premultiply the secant equation with s_k and get $s_k^T B_{k+1} s_k = s_k^T y_k > 0$, where the final inequality is obtained if the step size used (α_k) satisfies the curvature condition of line search (p.33-34). This implies B_{k+1} is a positive definite matrix, and we can impose additional constraints to solve for the 'best', as in the closest B_{k+1} to B_k by semidefinite programming:

$$\begin{gathered} \min ||B - B_K|| \\ \text{s.t. } B = B^T, B s_k = y_k, B \in S_+^n \end{gathered}$$

We now only have to solve for $n(n+1)/2$ variables and this problem always produces a unique solution. In fact, the analytic solution can be computed by equation 6.13.

However, B_k is only the approximated hessian, recall that our ultimate goal is to solve for the update direction by $p_k = -B_k^{-1}\nabla f_k$. We could speed things up by directly approximate the inverse hessian instead of the hessian, by:

$$\begin{gathered}\min ||H - H_K|| \\ \text{s.t. } H = H^T, s_k = Hy_k\end{gathered}$$

where H being the approximated inverse hessian. We can also obtain an analytic solution to H by:

$$\begin{gathered}H_{k+1} = (I - \rho_k s_k y_k^T)H_k(I - \rho_k y_k s_k^T) + \rho_k s_k s_k^T \\ \text{where} \qquad \rho_k = 1/y_k^T s_k\end{gathered}$$

With all ingredients in place, the BFGS algorithm is straightforward. We update the iterates by:

$$x_{k+1} = x_k - \alpha_k H_k \nabla f_k$$

We follow the usual logic in Newton's method, but this time the computational cost is significantly reduced: instead of computing the exact inverse hessian, we obtain the approximated inverse hessian iteratively, which is simply matrix operations.

4. **L-BFGS (Chapter 7.2)**

   To push it even further, the limited memory BFGS method (L-BFGS) shrinks the memory usage of vanilla BFGS method. As discussed above, computing H_k by BFGS is cheaper than $\nabla^2 f_k^{-1}$ by Newton's method, but both methods still have to store the $n \times n$ matrix, which could be problematic for large number of variables.

   L-BFGS exploits the fact that the term $H_k \nabla f_k$ can be recursively computed by $\{y_i, s_i\}, i < k$ (equation 7.19). In fact, we don't have to use all $\{y_i, s_i\}, i < k$ and find the exact $H_k \nabla f_k$, one could only use the m most recent (y_i, s_i) pairs and still obtain reasonably accurate approximations. Therefore throughout the algorithm, we can replace the complete H_k matrix by the set of vectors $\{y_i, s_i\}, i = k-1, ..., k-m$, and compute $H_k \nabla f_k$ by two-loop recursion (Algorithm 7.4) and proceed as usual.

5. **Conjugate gradient methods (Chapter 5)**

   We now shift gears to discuss a type of methods with slightly different approach. Just like steepest descent and unlike second order methods, conjugate gradient (CG) methods do not utilize second order information, it descend in a particular direction solely based on gradients. Recall that in steepest descent, the convergence rate is being held back by the 'zig-zag' pattern due to its simple / myopic strategy, whereas for conjugate gradient methods, as we shall see, are able to obtain a 'big picture' of the global curvature, and therefore descend more swiftly and directly than steepest descent.

Linear CG (Chapter 5.1)

Minimizing the objective function $f(x) = (1/2)x^T Ax - b^T x + c$ is equivalent to solving the linear system $\nabla f = Ax - b$, where A is a symmetric positive definite matrix. Define residual $r_k = Ax_k - b$, and error $e_k = x_k - x^*$ where x^* is the global optimum, i.e. $Ax^* = b$. We can therefore relate residual and error by $r_k = Ax_k - b = A(x_k - x^*) = Ae_k$. Intuitively, this can be interpreted as residual being the error in A space.

Loosely speaking, the main idea of CG methods is that the search direction taken throughout the algorithm are mutually orthogonal. One might wonder how does CG methods differ from steepest descent when both methods also consists of orthogonal search directions, one subtle difference is that in steepest descent, only immediately successive search directions have to be orthogonal, whereas CG methods require all search directions to be mutually orthogonal. This property prevents CG methods from revisiting previous directions over and over again, which is the culprit of slow convergence of steepest descent.

We now formally discuss the idea of conjugacy. A set of nonzero vectors $p = \{p_0, ..., p_{n-1}\}$ is said to be conjugate with respect to a positive definite matrix A if:

$$p_i^T A p_j = 0 \qquad \forall i \neq j$$

which can be thought of p_i, p_j being orthogonal in the A space. It can also be easily proven that vectors of p are linearly independent, in other words, the set p consists of basis vectors and spans $\mathbb{R}^n$. This implies that with appropriate step size, the optimum $x^* \in \mathbb{R}^n$ can always be attained by a unique linear combination of p_i and step size σ_i:

$$x^* = x_0 + \sigma_0 p_0 + ... + \sigma_{n-1} p_{n-1}$$

Coincidentally and interestingly, σ_i happens to be the exact line search parameter α_i of the corresponding search direction(Theorem 5.1). Put it another way, CG methods minimizes the objective function along the search direction at each iteration, which is in fact the optimal solution of subspace spanned by all previous directions, i.e. $x_k = \min_{x \in C} f(x), \quad C = \{x \,|\, x_0 + \mathbf{span}\{p_0, ..., p_{k-1}\}\}$ (Theorem 5.2), so we can conclude that the CG method converges in n steps, as x_n minimizes the space spanned by all basis vectors. One caveat for this nice property to hold is that the basis p should align with the major axes of the ellipse (Figure 5.1, 5.2).

One might wonder how the set of conjugate vectors can be obtained, unfortunately there is no simple rule in deciding so. For instance, the set of eigenvectors is a suitable candidate as they are mutually orthogonal, but it is not economical to compute eigenvectors in large scale applications.

Nevertheless, it is not necessary to compute all search directions beforehand, we could compute search directions iteratively:

$$p_k = -r_k + \beta_k p_{k-1}$$

where β_k is a scalar coefficient. To ensure p_k satisfies conjugacy, we premultiply the equation by $p_{k-1}^T A$ and obtain the equality:

$$p_{k-1}^T A p_k = -p_{k-1}^T A r_k + \beta_k p_{k-1}^T A p_{k-1} = 0$$
$$\beta_k = \frac{p_{k-1}^T A r_k}{p_{k-1}^T A p_{k-1}}$$

Turns out this iterative method not only restricts p_{k-1}, p_k to be A-orthogonal, but also satisfies conjugacy: $p_k^T A p_i = 0, \quad i = k-1, ..., 0$ (Theorem 5.3). The caveat for the proof to hold is that p_0 has to be initialized as $-\nabla f(x_0)$. This method requires minimal storage and computation, yet still provides satisfactory convergence characteristics under certain (ideal) assumptions.

Nonlinear CG (p. 121, 122)

Nonlinear CG methods deal with general objective functions that are not convex quadratic, i.e. ∇f is no longer $r = Ax - b$. Still, the linear CG method can be easily extended to handle such situations. One of the methods, namely the Fletcher-Reeves method requires two modifications of linear CG to become a nonlinear version:

1. step size α_k should minimize the nonlinear function f_k along p_k by linear search (instead of the analytic solution for α_k in linear CG)
2. residual should be defined as the gradient of the nonlinear function (instead of simply setting $r = Ax - b$)

The modifications require parameters to be 'tailor-made' for the nonlinear functions instead of the original parameters in linear CG. Otherwise, the algorithm is more or less identical to linear CG. Once again, only gradients and objective function are computed, little to no computation of hessian or matrix operations which makes CG methods in general highly appealing for large scale applications.

6. **Line search newton CG method (p.168-170)**

Recall that in Newton's method, we obtain the descent direction by solving:

$$\nabla^2 f_k p_k = -\nabla f_k$$

, from Quasi-Newton methods we saw that the cost of solving this equation can be reduced by approximating the hessian. Alternatively, we could also approximate the descent direction p_k, or take a different strategy in determining p_k based on second order methods. Bottom line is that we do not exactly determine p_k by the above linear system, this class of methods is referred to Inexact Newton methods.

We focus on one of the methods: Line search newton CG method in this section. It is a variant of Newton's method, with only slight modifications. Instead of computing the exact p_k, it applies CG methods to solve the 'inner problem' with second order approximation:

$$m_k(p) = f(x_k + p) \approx f_k + \nabla f_k^T p + \frac{1}{2} p^T B_k p$$

, that is to compute the minimum of this model, which then becomes the resultant direction of a particular iteration. This method is inexact since second order approximations are not necessarily well conditioned, and there are no guarantees in convergence characteristics of CG methods in this case. Yet, CG methods could provide reasonably well approximations to the exact solution. Following this inner problem, we update the iterate by an additional line search and then proceed as if Newton's method were used.

Just like the argument in the BFGS method, exactness is not necessary for every iteration, and the cheaper computation allows this method to be appealing for large scale problems. Additionally, in order to solve the inner problem by CG methods, it suffices to only use the product $\nabla f_k^2 d$, where d is directions taken in the inner problem, instead of explicit knowledge of ∇f_k^2, and we could in fact obtain an approximation of $\nabla f_k^2 d$ by finite difference:

$$\nabla^2 f_k d \approx \frac{\nabla f(x_k + hd) - \nabla f(x_k)}{h}$$

which further cuts down both storage and computation cost.

Nevertheless, the additional step of line search (compared to Newton's method) is this method's vulnerability. Consider a situation that the current iterate has a near-singular hessian (flat curvature), line search potentially takes a significant number of steps just to result in slight reduction of cost.

7. **Algorithms for partially separable function**

   Separable objective functions are ones that can be decomposed into 'sub-functions', and it is generally easier to optimize multiple m-dimensional functions than a single n-dimensional function, with $m < n$.

   $$f(x) = f_1(x_i) + f_2(x_j, x_k) + f_3(x_l, x_m, x_n)$$

   Often times objective functions are not completely separable, we could still, to some extent decompose the function into smaller ones, namely element functions. If element functions remain unaffected as we move along linearly independent directions, the objective function is said to be partially separable.

   Algorithms for partially separable function per se is not an optimization algorithm, but a mathematical trick to 'compactify' the derivatives of the objective function. With k element functions, the derivatives of partially separable functions can be written as:

   $$\nabla f = \sum_{i=1}^{k} \nabla f_i, \qquad \nabla^2 f = \sum_{i=1}^{k} \nabla^2 f_i$$

   which are inherently sparse. For example, consider a simple objective function $f(\mathbf{x}) = x_1^2 + x_2^2$, the derivatives would be:

   $$\nabla f = 2 \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \qquad \nabla^2 f = 2 \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Imagine $\mathbf{x} \in \mathbb{R}^n$, where n is significant in large scale applications, the hessian consists of $n^2 - n$ zeros, which definitely is not ideal for second order methods.

Take our problem as an example, the objective is to minimize the function:

$$f(\mathbf{x}) = \sum_{i=1}^{n} e^{x_i/\alpha_i} + e^{-x_i/\alpha_i} \qquad \mathbf{x} \in \mathbb{R}^n$$

, in this case the objective function is completely separable, which is an extreme and uncommon situation. A more realistic use case is illustrated in the book (p. 187-188).

Without loss of generality, let $n = 3$.

$$f(\mathbf{x}) = \sum_{i=1}^{3} f_i(\mathbf{x}), \qquad f_i(\mathbf{x}) = e^{x_i/\alpha_i} + e^{-x_i/\alpha_i}$$

Let $U_i \in \mathbb{R}^3$ be an indicator matrix, such that $U_i\mathbf{x} = x_i$, that is:

$$U_1 = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}, \quad U_2 = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}, \quad U_3 = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$$

, and let $\phi_i(U_i\mathbf{x}) = f_i(\mathbf{x})$. Altogether we have:

$$\begin{aligned} f_i(\mathbf{x}) &= \phi_i(U_i\mathbf{x}) \\ \nabla f_i(\mathbf{x}) &= U_i^T \nabla \phi_i(U_i\mathbf{x}) \\ \nabla^2 f_i(\mathbf{x}) &= U_i^T \nabla^2 \phi_i(U_i\mathbf{x}) U_i \end{aligned}$$

Take $i = 1$ as an example, we have:

$$\nabla^2 f_1(\mathbf{x}) = \frac{1}{\alpha_1^2} \begin{bmatrix} e^{x_1/\alpha_1} + e^{-x_1/\alpha_1} & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad \nabla^2 \phi_1(U_1\mathbf{x}) = \frac{1}{\alpha_1^2}(e^{x_1/\alpha_1} + e^{-x_1/\alpha_1})$$

By convex optimization theory, we know that if a convex function $f(x, y) = g(x) + h(y)$, then $\min_{x,y} f(x, y) = \min g(x) + \min h(y)$. Likewise, in our problem, we could optimize the smaller element functions independently. The above transformation results in a 1-dimensional second derivative. We could therefore optimize 1-dimensional element functions, for n times.

If, however, the function is only partially separable, such as:

$$f(x_1, x_2, x_3) = f_1(x_1, x_2) + f(x_2, x_3)$$

the above discussion is still a powerful tool in reducing computational complexity. For example, we could apply Quasi-newton methods such that $B_i \approx \nabla^2 \phi_i$ and recover the full hessian approximation by:

$$B = \sum_{i=1}^{k} U_i^T B_i U_i$$

, instead of computing the sparse $n \times n$ matrix every time, which would take a considerable amount of space and time for large n.

References

Nocedal, J., & Wright, S. J. (2006). *Numerical optimization*. Springer.

Shewchuk, J. R. (1994). *An introduction to the conjugate gradient method without the agonizing pain*. Carnegie-Mellon University. Department of Computer Science.