An Exploration of Optimization on Smooth Manifolds

Brandon D. Finley
Applied Mathematics
University of Colorado, Boulder
Email: brfi3983@colorado.edu

David Lujan
Aerospace Engineering Sciences
University of Colorado, Boulder
Email: dalu9892@colorado.edu

April 26, 2021

1 Introduction

This project delves into the topic of optimization on smooth manifolds. For problems without any constraints the problem is said to be an unconstrained problem where the optimization variable x is allowed to move freely around $\mathbb{R}^n$. Adding constraints changes the structure of the problem and changes the available methods to solve it. For constrained optimization x is not allowed to move freely around $\mathbb{R}^n$ and must remain on the surface defined by the constraints. Under certain conditions, to be explained later, the constraints may satisfy the definition of a smooth manifold. Viewing the constraints as a smooth manifold allows for the use of calculus on manifolds that comes from differential geometry, and expands the set of tools available to solve the problem. Many problems in convex optimization can be recast into an optimization problem on a smooth manifold.

This project report will explain the basic theory needed to understand the Riemannian Gradient Descent and Riemannian Newton Method algorithms. Both of these algorithms are developed, implemented, and tested on example problems and compared to solutions of built-in optimization packages. The first example is the minimization of the Rayleigh quotient on the $(n-1)$-dimensional sphere defined by the l_2 norm. The second example is a semi-definite program with one constraint. Following the examples is a discussion of the drawbacks and difficulties associated with manifold optimization. Then a discussion on other algorithms and uses of manifold optimization will follow. All code can be found on GitHub.

2 Background

2.1 Smooth Manifold Optimization Problem Statement

The problem statement for smooth manifold optimization is stated as

$$\min_{x\in\mathcal{M}} f(x) \tag{1}$$

where f is a smooth function of x, $\mathcal{M} = \{x|h(x) = 0\}$, and rank $\left(\frac{\mathrm{d}h}{\mathrm{d}x}\right) = k \; \forall x \in \mathcal{M}$. The value of k is the dimension of the manifold and is less than or equal to the dimension n of x. This problem statement is a subset of the general optimization problem

$$\begin{aligned} &\min_x f(x) \\ &\quad f_i(x) \leq b_i, \quad i = 1, 2, ..., m \\ &\quad h_i(x) = 0, \quad i = 1, 2, ..., p \end{aligned} \tag{2}$$

such that $m = 0$.

A convex problem is defined by Problem (2) where each function is convex. Notice that this can be recast as a smooth manifold problem when there are no inequality constraints and each function is sufficiently differentiable. For Riemannian Gradient Descent each function h_i only need be in C^1 while for Riemannian Newton's Method each function h_i needs to be in C^2. Not all convex problems can be recast into a smooth manifold problem. A simple example of such a convex problem is one that includes the l_1 norm in either the cost function or in h_i.

It is Problem (1) that the optimization algorithms aim to solve. Before presenting the algorithms it is necessary to go through the theory of differential geometry and present the definitions required to develop the algorithms.

2.2 Differential Geometry

Most people that have taken calculus are familiar with derivatives and gradients, so when differentiating a function on a manifold one might naively compute the derivative as they have learned. What isn't taught in calculus is that those derivatives are derivatives in Euclidean spaces, where x is allowed to take on any value. For a manifold, however, x is restricted to being on the manifold, so the classical definition (Equation (3)) of a derivative does not hold. This is because $f(x+h)$ is likely not defined as $x+h$ may not be on the manifold.

$$f'(x) = \lim_{h\to\infty} \frac{f(x+h)-f(x)}{h} \tag{3}$$

To compute a gradient we first need to define[1] a derivative on a manifold, but before that we need the notion of a tangent space of a manifold $\mathcal{M}$.

Definition 2.1. Let $\mathcal{M}$ be a subset of a Euclidean space E. For all $x \in \mathcal{M}$ define

$$T_x\mathcal{M} = \{c'(0) | c : I \to \mathcal{M} \text{ is smooth around 0 and } c(0) = x\}. \tag{4}$$

That is, v is in $T_x\mathcal{M}$ if and only if there exists a smooth curve on $\mathcal{M}$ passing through x with velocity v.

With this we define the differential of a function on a manifold.

Definition 2.2. The differential of $f : \mathcal{M} \to \mathcal{M}'$ at x is a linear operator $Df(x) : T_x\mathcal{M} \to T_{f(x)}\mathcal{M}'$ defined by

$$Df(x)[v] = \frac{\mathrm{d}}{\mathrm{d}t} f(c(t))|_{t=0}, \tag{5}$$

where c is a smooth curve on $\mathcal{M}$ passing through x at $t=0$ with velocity v.

Before we can define the gradient on a manifold we need to define what a Riemannian metric is for the manifold. This is because the choice of metric will change the notion of the gradient. For Euclidean space the metric is always the Euclidean metric, so this typically isn't specified.

Definition 2.3. A metric $\langle .,. \rangle_x$ on $\mathcal{M}$ is a Riemannian metric if it varies smoothly with x, in the sense that if V, W are two smooth vector fields on $\mathcal{M}$ then the function $x \mapsto \langle V(x), W(x)\rangle_x$ is smooth from $\mathcal{M}$ to $\mathbb{R}$.

Any manifold $\mathcal{M}$ equipped with a Riemannian metric $\langle .,. \rangle_x$ is defined as a Riemannian manifold. With this in mind we can now define the Riemannian gradient on a manifold.

Definition 2.4. The Riemannian gradient of f is the vector field $grad\ f$ on $\mathcal{M}$ *uniquely* defined by the following

$$\forall (x,v) \in T\mathcal{M} \quad Df(x)[v] = \langle v, grad\ f(x)\rangle_x \tag{6}$$

For the special case of Riemannian submanifolds - when the manifold is a linear subspace of a Euclidean space equipped with the Euclidean inner product - the Riemannian gradient reduces to the orthogonal projection of the Euclidean gradient onto the tangent space of $\mathcal{M}$ defined by

$$\begin{gathered} gradf(x) = \mathrm{Proj}_x(grad\bar{f}(x)), \\ \mathrm{Proj}_x : E \mapsto T_x\mathcal{M} \subseteq E \end{gathered} \tag{7}$$

where $\bar{f}$ is any smooth extension of f to a neighborhood of $\mathcal{M}$ in E. A smooth extension of f means that $\bar{f}(x) = f(x)$ $\forall x \in \mathcal{M} \cap U$, where $U \subseteq E$. This is an important observation as it provides a concrete and methodical way of computing Riemannian gradients when the manifold is a Riemannian submanifold. The sphere is a Riemannian submanifold and is one reason it is a nice test problem to use.

Now that the gradient on a manifold has been defined there is another important function to enable algorithms to operate on manifolds. This function is called a retraction.

[1]All definitions come from Boumal[1]

Definition 2.5. A retraction on $\mathcal{M}$ is a smooth map $R : T\mathcal{M} \to \mathcal{M}$ with the following properties. For each $x \in \mathcal{M}$ let $R_x : T_x\mathcal{M} \to \mathcal{M}$ be the restriction of R at x so that $R_x(v) = R(x, v)$. Then,

1. $R_x(0) = x$
2. $DR_x(0) : T_x\mathcal{M} \to T_x\mathcal{M}$ is the identity map: $DR_x(0)[v] = v$.

The retraction is essentially a function that can take a point from the tangent space $T_x\mathcal{M}$ and map it to the manifold $\mathcal{M}$.

With these definitions in place we are ready to go through the Riemannian Gradient Descent algorithm, however to understand the Riemannian Newton's Method algorithm we also need to understand what a Hessian is on a manifold.

2.3 Importance of Positive-Definite H_x's

For second order algorithms, we actively use the Hessian of the cost function, denoted as $H_x := Hess\ f(x)$ or a similar linear operator in the optimization algorithms. Because of this, many numerical issues arise due to H_x not being sufficiently positive-definite. Thus, we will take a departure to talk about why a positive-definite H_x or sufficiently positive-definite H_x is important. In order to understand this, recall the definition of a Riemannian gradient

$$Df(x)[v] = \langle v, grad\ f(x)\rangle_x \tag{8}$$

and since we are trying to solve for the Newton step, we have the Newton equations,

$$H_x[s] = -grad\ f(x) \tag{9}$$

$$s = -(H_x)^{-1} grad\ f(x) \tag{10}$$

so using the Riemannian gradient and plugging in s, we have

$$Df(x)[s] = \langle gradf(x), s\rangle_x \tag{11}$$

$$= -\langle gradf(x), (H_x)^{-1} grad\ f(x)\rangle_x \tag{12}$$

which, by the definition of a positive-definite matrix, is only required to be negative if H_x is positive-definite. Requiring this to be true causes the Newton step to follow along a descent direction that favors local minima rather than saddle points or local maxima.

Alternatively, if one does not have a positive-definite operator H_x, one could modify this approach and require that H_x acts sufficiently close to the Hessian by adding a linear term E_k, then we say we have a *quasi-Newton*[3] method where we compute the modified Newton step,

$$(Hess\ (f(x_k) + E_k)[s] = -gradf(x_k) \tag{13}$$

Although this guarantees (or increases the inclination for quasi methods) a descent direction, Newton's method still fundamentally acts as a local convergence method. It is only after the combination of the superlinear convergence properties of Newton's method with the global convergence properties of a Trust-region, that an algorithm is obtained with both superlinear local convergence and global convergence. This method will be presented in section 3.4.

3 Algorithms

In this project we explore two main algorithms for smooth manifold optimization. The first is the Riemannian Gradeint Descent (RGD), a first-order algorithm utilizing the Riemannian gradient, which is the Euclidean equivalent of Gradient Descent for unconstrained optimization. The second is the Riemannian Newton's Method (RNM), a second-order algorithm utilizing both the Riemannian gradient and Hessian. Both algorithms perform "unconstrained" optimization on smooth manifolds in the sense that the manifold is the only space that x sees. For RGD a backtracking line search is implemented, and for RNM a Trust-region method is implemented. These tools are meant to increase the speed of convergence for each algorithm.

3.1 Riemannian Gradient Descent

The basic idea for RGD is simple. Starting with a point $x_0 \in \mathcal{M}$ the gradient is computed to get a direction to take a step in. The step, which resides in the tangent space $T_{x_0}\mathcal{M}$, is then retracted down onto the manifold to get the updated point x_1. This is similar to Gradient Descent with the added step of the retraction to ensure the updated point

Algorithm 1: Riemannian Gradient Descent

```
Initialize: $x_0 \in \mathcal{M}$, $t_0$;
while $error \leq \epsilon$ do
    Choose $t_k$
    $s_k = -t_k gradf(x_k)$
    $x_{k+1} = R_{x_k}(s_k)$
    $error = \frac{||x_{k+1} - x_k||}{||x_k||}$
end
```

stays on the manifold. The main reason the Riemannian gradient is needed is because the Euclidean gradient of the cost function $f(x)$ may not lie in $T_x\mathcal{M}$ and hence won't be the direction of steepest descent. This may not be a big deal in some cases, as will be seen, but it is a good idea to step in a direction along the true gradient to ensure the method is a descent method.

Algorithm 1 is written here to be amenable to either a fixed step size or a step size determined from a line search. The backtracking line search can be found in Algorithm 2. This algorithm may look very similar to the Projected Gradient Descent (PGD) algorithm as they are similar. The retraction function plays the same roll as the projection function as it takes a linearized update and moves it back to the surface defined by the constraints. The main difference, then, is the update to the gradient and how the linearized update is moved back to the constraint surface. In PGD the gradient is taken to be the Euclidean gradient of the cost function, and may not be the best linear update given the constraints, while in RGD the gradient is the Riemannian gradient and is guaranteed to be in the direction of steepest descent. The projection function in PGD is defined to be the point x_{k+1} that is closest to the linearized update and lies on the constraint surface, whereas the retraction function in RGD is a direct mapping from the tangent space to the manifold. Considering this information RGD should be a more accurate and faster algorithm than PGD given the same set of constraints.

3.2 Riemannian Newton Method

To implement Riemannian Newton Method (RNM), we consider a model $m_{x_k} : T_{x_k}\mathcal{M} \mapsto \mathbb{R}$ that approximates $\hat{f}_x(s)$. Here, we will also define a linear operator $H_x = Hess\ f(x)$. Recall that from Taylor expansions and second-order retractions, it holds that

$$\hat{f}_x(s) \approx m_x(s) \triangleq f(x) + \langle grad\ f(x)\ , s\rangle_x + \frac{1}{2}\langle H_x[s], s\rangle_x \tag{14}$$

Here, we have a quadratic approximation of our cost function. And so the goal of this method is to solve for an s_k such that our approximate model is minimized.

Now note that a minimizer of m_x, if it exists, must be a critical point of m_x. So, if we take the gradient of (14) and set it equal to 0, we get the following

$$grad\ m_x(s) = grad\ f(x) + H_x[s] = 0 \tag{15}$$

$$\Rightarrow H_x[s] = -grad\ f(x) \tag{16}$$

Next, since s is a critical point of m_x, then if H_x is invertible, we obtain a unique solution to the linear equations above (Newton equations), called the Newton step. Solving for s is the key to Newton's method as it will minimize our approximate value, and move along its direction.

Note: Recall that in section 2.3, we illustrated the importance of positive definite H_x's due to potentially favoring local maximas and saddle points. Refer to that section for more detail.

3.3 Line Search

A backtracking line search is implemented that satisfies the following Armijo-Goldstein condition

$$f(x_k) - f(R_{x_k}(-t_k gradf(x_k))) \geq rt_k||gradf(x_k)||^2 \tag{17}$$

where $r \in (0, 1)$[1]. The line search is used at each iteration k to find the best step size to achieve faster convergence. The algorithm for the line search is given in Algorithm 2.

Algorithm 2: Line Search

```
Initialize: $t_k > 0$, $\alpha$;
while $f(x_k) - f(R_{x_k}(-t_k gradf(x_k))) \geq rt_k ||gradf(x_k)||^2$ do
    $t_k \leftarrow \alpha t_k$
end
```

3.4 Riemannian Trust-region

As noted above, a Riemannian Trust-region is motivated by attempting to find a balance between the local, superlinear convergence of Newton's method and the global convergence properties of a line-search model. To delve into how this is done, we consider a quadratic approximation, $\hat{m}_{x_k}(\eta)$, at our iterate x_k,

$$\hat{m}_{x_k}(\eta) = \left[f(x_k) + \langle grad\ f(x_k), \eta \rangle + \frac{1}{2} \langle H_k[\eta], \eta \rangle \right] \tag{18}$$

Given this, we wish to minimize (or more practically, approximately minimize) $\hat{m}_{x_k}(\eta)$, with the constraints that our value is within the trust-region radius and lies within the tangent space,

$$\eta_k = \underset{\substack{\eta \in T_{x_k}\mathcal{M} \\ \langle \eta, \eta \rangle \leq \Delta_k^2}}{\arg\min} \quad \hat{m}_{x_k}(\eta) \tag{19}$$

Given the result of this sub-problem, we then are given a new candidate for the iterate after we apply the retraction, $x_{k+1}^+ = R_{x_k}(\eta_k)$. Now, to determine the quality of this proposed candidate, we compare the actual difference in our cost function to the estimated difference given by $\hat{m}_{x_k}(\eta)$,

$$\rho_k = \frac{f(x_k) - f(x_{k+1}^+)}{\hat{m}_{x_k}(0) - \hat{m}_{x_k}(\eta_k)} \tag{20}$$

called the *quality quotient*.

Finally, we use this quotient to determine if we should accept or reject the new iterate. In short, given a threshold $\rho^* \in (0, 0.25]$, typically set to be 0.1, if $\rho_k > \rho^*$, then the candidate is accepted and otherwise reject the candidate and keep our old iterate.

In addition to accepting or rejecting the iterate, we also consider the trust region and update if necessary. That is, we define the updated radius of the trust region, Δ_k as the following, where $\bar{\Delta}$ is the maximum radius,

$$\Delta_{k+1} = \begin{cases} \frac{1}{4}\Delta_k & \text{if } \rho_k < \frac{1}{4}, \\ \min(2\Delta_k, \bar{\Delta}) & \text{if } \rho_k > \frac{3}{4} \text{ and } \|s_k\|_{x_k} = \Delta_k, \\ \Delta_k & \text{otherwise.} \end{cases} \tag{21}$$

which decreases the radius if the model produced a poor approximation, decreases it if it lies on the boundary of the radius but is a good approximation, and keeps the previous radius if it is doing well. The reason why the radius decreases even if it lies on the boundary is because the iterates will grow in magnitude, so if they lie on or outside the boundary there is no way they can come back inside the trust region.

To combat numerical issues, such as running into high differences in cost and lose in model approximations, where the error is amplified, consider the following forms of the denominator,

$$m_k(0) - m_k(s_k) = \left(f(x) + \langle grad\ f(x), 0 \rangle_x + \frac{1}{2} \langle H_x(0), 0 \rangle_x \right) \tag{22}$$

$$- \left(f(x) + \langle grad\ f(x), s_k \rangle_x + \frac{1}{2} \langle H_x(s_k), s_k \rangle_x \right) \tag{23}$$

$$= -\langle s_k, grad\ f(x) \rangle_x - \frac{1}{2} \langle s_k, H_x(s_k) \rangle_x \tag{24}$$

Using this form, we can normalize this quotient by adding a small value δ_k, where $\delta_k = \max(1, |f(x_k)|)\epsilon\rho_{reg}$. In practice, ρ_{reg} is usually set to be 10^3. Thus, my ρ_k became

$$\rho_k = \frac{f(x) - f(s_k) + \delta_k}{-\langle s_k, grad\ f(x) \rangle_x - \frac{1}{2} \langle s_k, H_x(s_k) \rangle_x + \delta_k} \tag{25}$$

Another thing to note is that when the current iterate v_n lies on or outside the trust region, increased error can be deterred by traveling back along the H-conjugate direction p_{n-1}, the linear amount we move t, is given by solving the following quadratic and obtaining the positive root[1],

$$\|v_{n-1} + tp_{n-1}\|_x^2 - \Delta^2 \tag{26}$$

We can solve this by expanding it and using the quadratic formula,

$$\|v_{n-1} + tp_{n-1}\|_x^2 - \Delta^2 = \underbrace{\|p_{n-1}\|_x^2}_{a} t^2 + \underbrace{2\langle v_{n-1}, p_{n-1}\rangle_x}_{b} t + \underbrace{\|v_{n-1}\|_x^2 - \Delta^2}_{c} \tag{27}$$

$$= at^2 + bt + c \tag{28}$$

More information and background regarding the H-conjugate directions are referenced in section 3.4.1.

3.4.1 Approximating the sub-problem with Truncated-CG

To solve the sub-problem in the trust region method, we opt for either finding an exact solution to minimize our quadratic model such as a line search or using conjugated gradients for an approximate solution. In essence, if the model $m_k(s)$ is to be minimized, then an iterative sequence of $v_0, v_1, \cdots, v_n \in T_x\mathcal{M}$ can be found such that v_n increasingly approaches s.

However, since we are given an operator H, and we use conjugate gradients, then the approximation norm is the following

$$\|u\|_H = \sqrt{\langle u, Hu\rangle_x} \tag{29}$$

and so, by expanding the squared norm with $u = v - s$ to find the approximation error, we define

$$g(v) = \frac{1}{2}\langle v, Hv\rangle_x - \langle v, b\rangle_x \tag{30}$$

such that

$$\|v - s\|_H^2 = 2g(v) + \langle s, Hs\rangle_x \tag{31}$$

where $Hs = -grad\ f(x) = b$ and $H = H^*$ due to H being self-adjoint on the Riemannian metric. Thus, to get a close approximation to s, the minimum of $g(v)$ needs to be found, meaning the value of s need not be known.

Now, using the fact that $g(v)$ can be minimized over the space spanned by nonzero tangent vectors, $p_0, \cdots, p_{n-1}$, and the constraint that the operator H is positive-definite, then any vector v can be defined as a linear combination of coefficients $y_0, \cdots, y_{n-1}$ and a basis of $p_0, \cdots, p_{n-1}$, that is

$$v = y_0 p_0 + \cdots + y_{n-1} p_{n-1} \tag{32}$$

Thus, to solve for the minimum of $g(v)$, we redefine v in terms of $y_i p_i$ and find the coefficients $y_0, \cdots, y_{n-1}$ such that $\frac{\partial g}{\partial y_i} = 0$, giving

$$y_i = \frac{\langle p_i, b\rangle}{\langle p_i, Hp_i\rangle} \tag{33}$$

Given a new conjugate direction, we can iterate to a updated approximation v_{n+1},

$$v_{n+1} = v_n + \frac{\langle p_n, b\rangle}{\langle p_n, Hp_n\rangle} p_n \tag{34}$$

This is important because with each iteration v_n, the dimension of the spanned subspace increases and therefore have an approximation at least as good as the previous iterate. This allows us to create a maximum of $dim\mathcal{M}$ iterations before the global minimizer of the quadratic model is found.

Thus, we can use this idea of conjugate gradients, consider a stopping criteria, and modify it by including the following

1. Looking out for an H that is not positive definite
2. Checking to see if we are in the trust region
3. Terminating early if the above two don't happen

to generate the full truncated-CG (tCG) method[1] defined in Algorithm 3.

Algorithm 3: Truncated Conjugate Gradient

```
Set: $\kappa = 0.1, \theta = 1$;
Initialize: $\mathrm{v}_0 = 0, r_0 = b, p_0 = r_0$;
for $n$ = 1, 2, $\cdots$ do
    $w$ = $H\, p_{n-1}$
    $y$ = $\langle p_{n-1}, w \rangle_x$
    $\alpha_n = \frac{\|r_{n-1}\|_x^2}{y}$
    $v_{n-1}^{+} = v_{n-1} + \alpha_n p_{n-1}$
    if $y \leq 0$ or $\|v_{n-1}^{+}\|_x \geq \Delta$ then
        $v_{n-1}^{+} = v_{n-1} + t p_{n-1}$
        output $s = v_n$ and $Hs = b - r_{n-1} + tw$
    else
        $v_n = v_{n-1}^{+}$
    end
    $r_n = r_{n-1} - \alpha_n w$
    if $\|r_n\|_x \leq \|r_0\|_x \min(\|r_0\|_x^{\theta}, \kappa)$ then
        output $s = v_n$ and $Hs = b - r_n$
    else
    end
    $\beta_n = \frac{\|r_n\|_x^2}{\|r_{n-1}\|_x^2}$
    $p_n = r_n + \beta_n p_{n-1}$
end
```

4 Results

Two convex test problems are chosen to show that smooth manifold optimization is a valid alternative to convex optimization. The first example is the minimization of the Rayleigh quotient on the sphere S^{n-1}. We use both RGD and RNM to solve the problem and compare the results to the built in manifold optimization package *PyManOpt*. A comparison is also made to using the Euclidean gradient inside RGD instead of the Riemannian gradient, as well as a comparison of fixed step lengths to a line search. The next example is the minimization of a semi-definite program. The problem is solved with the convex optimization package *CONVEX.jl* and attempted to be solved with RGD.

4.1 Sphere

Consider the manifold $\mathcal{M}$ to be the unit sphere in $\mathbb{R}^n$,

$$S^{n-1} = \{x \in \mathbb{R} : \|x\| = 1\} \tag{35}$$

and the cost function to be

$$f(x) = \frac{1}{2} x^T A x \tag{36}$$

where A is an $n \times n$ symmetric matrix. Then the optimization problem becomes

$$\min_{x \in \mathcal{M}} f(x) \tag{37}$$

In order to implement any Riemannian optimization algorithm, we must at least consider the Riemannian gradient and retraction for the manifold. We are able to locally approximate movement along the sphere with a tangent space, defined as,

$$T_x S^{n-1} = \{v \in \mathbb{R}^n : v^T x + x^T v = 0\} \tag{38}$$

$$= \{v \in \mathbb{R}^n : x^T v = 0\} \tag{39}$$

Using the idea of a retraction and the projected component onto the tangent space, we can define our Riemannian metric and therefore have a gradient. This provides the following identities

$$grad\, f(x) = Proj_x(grad\, \bar{f}(x)) = 2\left(Ax - (x^T A x)x\right) \tag{40a}$$

$$R_x(v) = \frac{x + v}{\|x + v\|} \tag{40b}$$

4.1.1 First Order Methods

In this section, we document our results from first order methods, namely Riemannian gradient descent with and without a line search. Additionally, we supplied RGD with the Euclidean gradient rather than the Riemannian gradient to see how necessary the Riemannian gradient is. Using the framed Problem (37), we obtain the following graph as a sample comparison between algorithms. The left plot contains the relative error between iterations while the right plot contains the objective function value at each iterate.

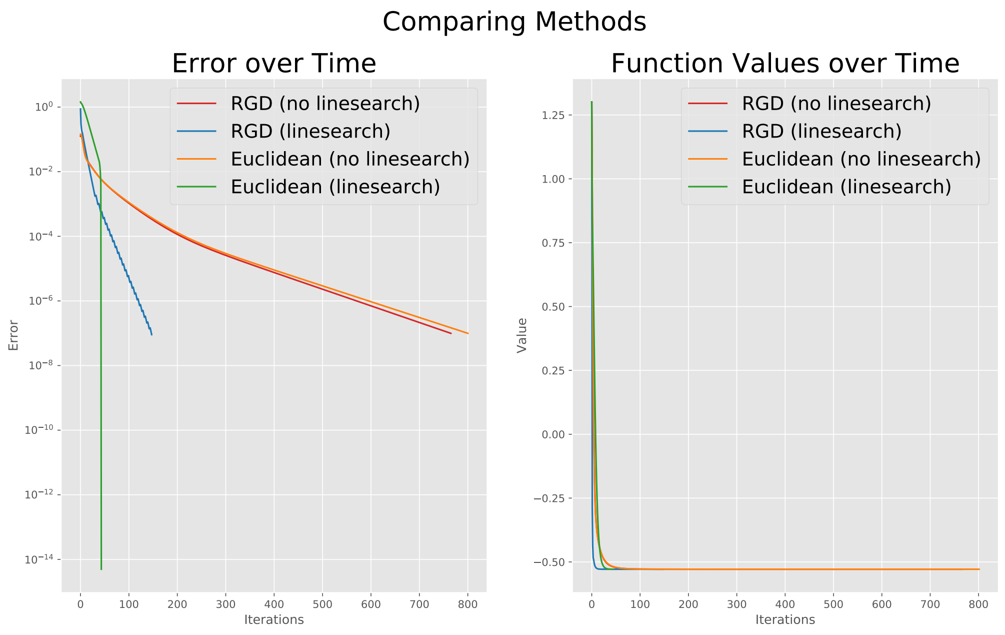

Figure 1: Comparing Methods

The models that included a line search converged much faster than those without one. It is, however, interesting to note that the “Euclidean” descent algorithm used above converged comparable to RGD and actually fastest with a line search. Looking into the results it was found that the method did not converge to the true solution. It was very close, but had a higher absolute error than the other three methods. This shows that although we optimized over a sphere, using just the Euclidean gradient rather than the Riemannian gradient proved to be feasible. We assume that this is due to the simplicity of the sphere where we possibly could have been take large steps around the sphere and being projected back onto, effectively traveling quickly around it without much effort in considering the step size.

Examining the right plot it can be seen that each method behaves as a descent method, that is $f(x_{k+1}) < f(x_k)$, regardless of using a line search or not.

Using these methods a comparison can be made with *PyManOpt*. Table 1 shows a comparison between the four methods and *PyManOpt* with the assumption that solution produced from *PyManOpt* is the true optimal value for x. Each method was timed and the error at the end was computed for 100 trials of a randomly populated symmetric matrix A. This allowed for a statistical evaluation of the methods to determine which is fastest and most accurate.

Algorithm	Mean			Standard Deviation		
	Error	Iterations	Time(s)	Error	Iterations	Time(s)
RGD	0.000010	1149.58	0.112739	0.000010	1025.697004	0.098169
RGD (linesearch)	**0.000000**	315.16	0.056891	0.000001	307.356038	0.055480
Euclidean	0.000010	1195.68	0.067975	0.000011	1061.842181	0.073473
Euclidean (linesearch)	0.031465	**89.59**	**0.021435**	0.048156	77.417194	0.029355
PyManOpt	0.000000	143.41	0.083971	0.000000	109.739974	0.061593

Table 1: First-Order Results

As you can see, RGD (with a line search) was the most accurate with an l_2 norm error of approximately 0. This was followed by RGD without a linesearch and then Euclidean gradient descent without a linesearch. However, we also note that Euclidean gradient descent had the least iterations and execution time, on average. We assume this is because of the choppy, but quick maneuvering around the unit sphere and suspect this would not work nearly this well on a more complicated manifold. Lastly, comparing our results to *PyManOpt* it can be noticed that while our method maintains similar levels of accuracy (to the sixth decimal before deviation) the execution times are faster.

We should also make note of the standard deviations captured as well. It appears that many of the values are significantly high in value. Because we randomly generated the matrix A it seems feasible that such deviations exist. This implies that some of time the methods converged in very few iterations, while an equal amount of time the methods converged in a large number of iterations. In retrospect, we should have kept a fixed value for A and randomly generated an initial starting point, x_0, to test the algorithms' performance. There are many ways to go about testing its computational prowess; however, the important note is that we see a pattern among the different models regardless.

4.1.2 Second Order Methods

For second order methods, we considered a Riemannian Newton method as well as Riemannian Trust-region. We also used *PyManOpt*'s second order Trust-region solving for comparison. A sample graph of a single instance of using the algorithms is found in Figure 2.

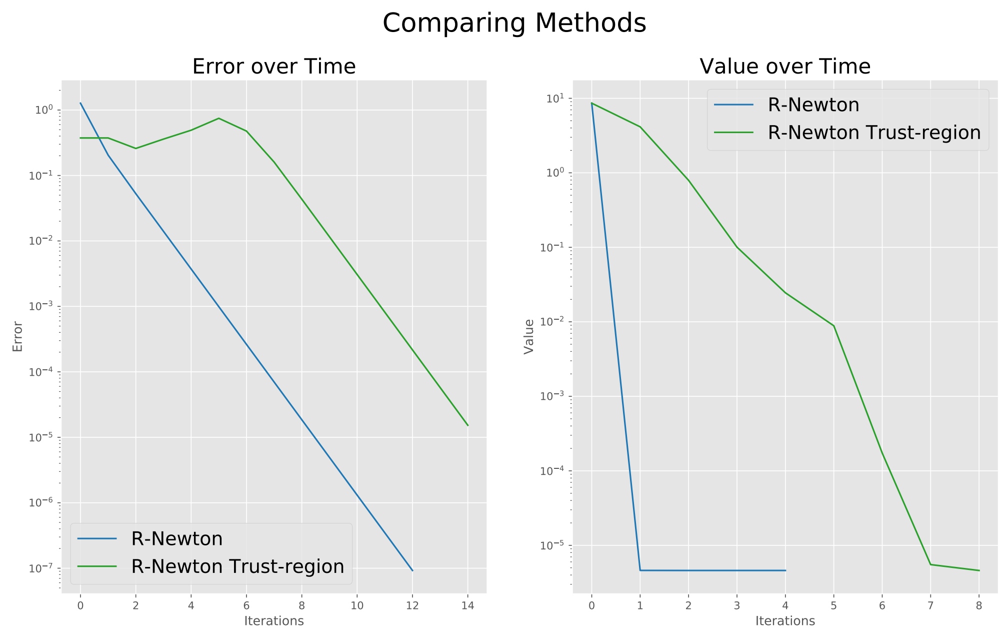

Figure 2: Second Order Methods

It appears that the Trust-region model initially had constant error, possibly because its radius was being optimized and so our quality quotient value was low. Only when the iterate updated to a new value did the error decrease. Once it started decreasing, it appeared to decrease at a similar rate than that of Newton's. A similar observation can be made for the function valuation, where it decreased overall slower but progressively got faster.

Again, using cross-validation for 100 iterations, with *PyManOpt*'s optimal value being the true value for determining error, we can see how the algorithms performed. This is illustrated in Table 2.

Algorithm	Mean			Standard Deviation		
	Error	Iterations	Time(s)	Error	Iterations	Time(s)
Riemannian Newton	**0.000000**	12.31	**0.004345**	0.000001	10.402591	0.003014
Riemannian Trust-region	0.000003	14.06	0.055266	0.000005	7.645679	0.037213
PyManOpt	0.000000	**7.53**	0.067466	0.000000	1.252637	0.013629

Table 2: Second-Order Results

Observing the results for the second order methods, it becomes apparent that Newton's method is the fastest among the three. It is also interesting to note that it has a lower error than that of the Trust-region model. We suspect this is because we redefined the stopping criteria that does not consider the norm, but rather the gradients of the iterates. Nonetheless, it is still acceptable given how quickly it converges.

It is also important to note that while Newton's method converges fastest, it does not have global convergence properties like the other two Trust-region models. Thus, on more topologically interesting manifolds, you may be able to find local minima with Newton's but fail to converge to a global minimum. We would also like to point out that our method was, on average, faster in execution time than *PyManOpt*. This may be due to the reading and writing to a log file and overhead time.

4.2 Semi-Definite Program

This section showcases another example of a convex problem that can be converted to a manifold problem. Consider the convex semi-definite program

$$\begin{aligned} &\min_{X\in S^n} \langle C, X\rangle \\ &\langle A_i, X\rangle = b_i, \quad i = 1,2,...,m \\ &X \succeq 0 \end{aligned} \tag{41}$$

where $\langle U, V\rangle = \text{Tr}(U^T V)$. By setting $m = 1$, $A_1 = I_n$, and $b_1 = 1$ then the spectrahedral manifold is formed[5]. This is the set of symmetric, positive semi-definite matrices with fixed rank and unit trace. The reason for doing this is that this is a well-known manifold so the gradient and retraction on this manifold is known. To solve this problem let $X = YY^T$. The resulting problem can be stated as

$$\min_{Y\in\mathbb{R}^{n\times n}} \text{Tr}(Y^T C^T Y) \tag{42}$$
$$\text{Tr}(YY^T) = 1$$

The condition $\text{Tr}(YY^T) = 1$ is satisfied when $||Y||_F = 1$. This provides a convenient way to get back to the manifold in a similar way to the sphere.

The gradient on this manifold is given by

$$gradf(Y) = grad\bar{f}(Y) - \langle grad\bar{f}(Y), Y\rangle Y, \tag{43}$$

recalling that $\bar{f}(x)$ is the smooth extension of $f(x)$ to a neighborhood of $\mathcal{M}$ in E.

The retraction is given by

$$R_Y(U) = \frac{Y+U}{||Y+U||_F} \tag{44}$$

4.2.1 Riemannian Gradient Descent

The semi-definite program defined by Problem (42) is attempted to be solved with RGD. The dimension n of the problem is chosen to be 10 and the C matrix is chosen from a uniformly random distribution. Both a constant step size and a line search are used. The results from this are in Figure 3.

From the figure it appears that using the line search results in very fast convergence while using a fixed step size results in a non-optimal solution. What's interesting about the figure is that when using a fixed step size we do not get the behavior of a descent method. This seems to imply something is not right about the implementation of the

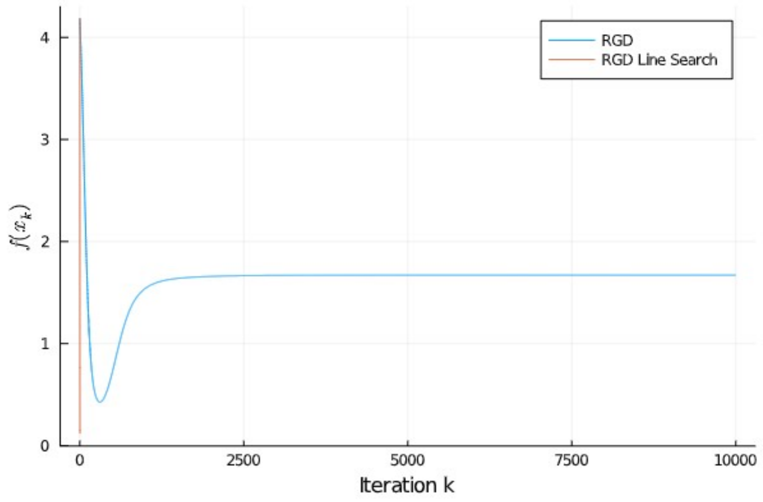

Figure 3: Riemannian Gradient Descent applied to the semi-definite program defined by Problem (42).

gradient or the retraction, however no issues were found at the time of this paper. The solution at the end of each solver satisfied $||Y||_F = 1$, implying the retraction function seems to be working properly, so the issue must lie in the gradient. Unfortunately the gradient checker that was constructed at the beginning of the semester will not work on Riemannian gradients, so we do not have the ability to check the gradient.

Evaluating the gradient at the last iteration of the line search shows that the gradient is not 0, so the solution is not a local minimum. The reason the algorithm terminates so soon is because the updated solution is approximately equal to the current solution. This is caused by the value of the step size returned from the line search being near the underflow value.

4.2.2 *CONVEX.jl*

Since the semi-definite program defined by Problem (41) is convex then Julia's convex optimization package *CONVEX.jl* can be used to solve the problem. The package solved the problem in 4.22 seconds with an optimal value of -1.16579115216234 and the optimal solution does indeed satisfy the constraints. Comparing the optimal value to Figure 3 further confirms that RGD with the line search did not find the optimal solution.

5 Drawbacks of Manifold Optimization

Optimization on smooth manifolds allows for a methodical way to handle a variety of equality constraints by using calculus on Riemannian manifolds. The idea for RGD is simple: compute the normal gradient of the objective function, project it onto the tangent space of the manifold, and then retract from the tangent space onto the manifold to get the updated solution. The issue is that computing the Riemannian gradient and retraction is not in general trivial. While there exist numerical methods to compute the Euclidean gradient, numerically computing the Riemannian gradient is not so straight forward, and changes when the manifold changes.

In the case that the manifold is a Riemannian submanifold, the Riemannian gradient is the orthogonal projection of the Euclidean gradient onto the tangent space of the manifold. The retraction is a mapping from the tangent space of the manifold onto the manifold and can take multiple forms. One of these forms is the exponential map, however this is difficult to find in general. This is due to the exponential map being derived from the idea of geodesic, a second-order ODE.

From the literature covered, there does not appear to be general methods to get the Riemannian gradient and a retraction map, so smooth manifold optimization seems to be limited to optimization on the well-known manifolds.

This framework is limited by the condition that the cost function and all constraint functions be sufficiently smooth. This eliminates all problems that include the l_1 norm, and hence most of the homework problems that were encountered during the semester. There are algorithms for handling non-smooth cost functions, however. Another limitation of this framework is that it cannot handle inequality constraints. A naive way to handle the inequality constraints would be to convert them to equality constraints and then wrap the problem into another optimization problem that takes the form

$$\min_{a_i \in \mathbb{R}} \left[\min_{x \in \mathcal{M}} f(x), \quad f_i(x) = a_i, \quad i = 1, 2, ..., m \right]$$
$$a_i \leq b_i, \quad i = 1, 2, ..., m$$

under the condition that each f_i is in C^1. Another, more efficient, method to account for inequality constraints is by using active set methods[4]. Doing this can greatly expand the number of problems that can be handled by smooth manifold optimization.

6 Other Algorithms

This project only scratches the surface in terms of the variety of algorithms that have been developed for manifold optimization. In a dissertation by Huang[2], a large amount of algorithms are presented for manifold optimization. He presents a secant condition on Riemannian manifolds, a Broyden family of quasi-Newton methods, and a method for optimizing partly smooth cost functions. Additionally, he applies these algorithms to a variety of example problems.

The manifold optimization packages that are available to Matlab, Python, and Julia contain numerous other algorithms as well and include Nesterov acceleration methods for RGD which were not considered in our work. If we were given more time, we would have explored the accelerated methods to see how they compare in iterations, time, and l_2 error.

7 Connections to Class

For this project, our aim was to apply techniques and concepts learned in class but on manifolds rather than Euclidean space. This allowed us to see the bigger picture while also practicing useful techniques, as we believe manifold optimization is getting more popular. To keep it short, the connections between our project and Advanced Convex Optimization that we noticed were the following

- Gradient Descent algorithms with linesearch
- Conjugate Gradient methods with Trust-Regions
- Comparing our methods to a semi-definite program
- Using linear and adjoint operators to reduce evaluations

Of course, most of this report is oriented around descent algorithms, so apart from the theory needed to implement these on non-Euclidean geometries, a lot of connections were made from our optimization algorithms, directly implementing what we learned in class but to a different domain.

8 Conclusion

Optimization on smooth manifolds is a relatively new topic that allows convenient ways to optimize over constraints that define a manifold by utilizing calculus on Riemannian manifolds. Additionally, it allows for alternative ways to think about and solve more traditional problems. We implemented the Riemannian Gradient Descent and Riemannian Newton's Method algorithms to a couple test problems with success on the sphere and had issues on the semi-definite program.

The effort to implement RGD is no more difficult than to implement a Projected Gradient Descent or Proximal Gradient Descent algorithms. The main difference is that the gradient of the cost function needs to be altered so that the gradient is defined on the manifold. In fact, there's a lot of similarity between these algorithms. However, from our tests on the sphere, using the Euclidean gradient in place of the Riemannian gradient does not have a significant affect on the performance of RGD.

In short, if you are given a problem that can be modeled without the use of retractions or Riemannian gradients, it is worthwhile to do so. However, if your data is constrained to move on the manifold, then you should opt for Newton's method if you believe the topology of the manifold is not so complicated and you need speed. However, if you need to utilize global performance properties, then using Riemannian Trust-region with either an approximated qudratic with truncated-CG or an exact minimum with a line search is needed. In either the Newton case or the Trust-region case, we urge you to treat the Hessian operator, H, with utmost importance, possibly adding a linear operator to cause it to be sufficiently positive-definite. Although simple, another method would be to just use the Riemannian gradient descent algorithm.

In the future, we would have like to apply our algorithms to new manifolds to test their rigidity. I also would have liked to experiement with solving the subproblem and their approximations, testing to see how much is enough and on what type of complexity of a manifold. Similarly, I would have wanted to test using a non-positive-definite operator, H, to see how much error the algorithms would incur. Nonetheless, we had fun doing this project and were glad we could apply the techniques learned in class in more of a non-traditional setting.

References

[1] Nicolas Boumal. *An Introduction to Optimization on Smooth Manifolds*. 2020.

[2] Wen Huang. Optiization algorithms on riemannian manifolds with applications, 2013.

[3] Melvin Leok. Optimization on manifolds, 2021.

[4] David G. Luenberger and Yinyu Ye. *Linear and Nonlinear Programming*. Springer, 2008.

[5] P.A. Absil M. Journée, F. Bach and R. Sepulchre. Low-rank optimization on the cone of positive semidefinite matrices, 2010.