Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Mathematical Sciences

Even Stephansen Kjemsås

The Riemannian Frank–Wolfe Algorithm

Master's thesis in Applied Physics and Mathematics
Supervisor: Ronny Bergmann
July 2022

NTNU
Norwegian University of
Science and Technology

Even Stephansen Kjemsås

The Riemannian Frank–Wolfe Algorithm

Master's thesis in Applied Physics and Mathematics
Supervisor: Ronny Bergmann
July 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Mathematical Sciences

Abstract

The Frank–Wolfe method is an iterative optimization algorithm commonly used to solve constrained convex optimization problems. Based on simplifying the problem to a linear subproblem, which is solved in each iteration, the algorithm is especially useful in cases where this subproblem can be solved in closed form.

While the Frank–Wolfe algorithm was originally formulated for solving problems in Euclidean space, it can often be beneficial to work with problems on manifolds. Sometimes the constraints of a Euclidean problem actually constrain the solution to lie on a manifold, and in other cases the data or the domain of the objective function may already be defined on the manifold.

In this thesis we study convex problems on manifold domains where, in addition to the manifold constraint, we also have further constraints on the solution. We work with a generalization of the classical Frank–Wolfe algorithm to Riemannian space, and examine how it differs from the Euclidean version – especially with regard to how the subproblem is no longer linear, and thus requires more sophisticated techniques to solve.

Our main contribution lies in implementing the algorithm as a general solver for problems on arbitrary Riemannian manifolds, using the interface provided by Manopt.jl, a Julia package for manifold optimization. We test the implementation numerically on the manifold of symmetric positive-definite matrices, but we note that the performance of the algorithm is not as good as we expected.

Sammendrag

Frank–Wolfe-algoritmen er en iterativ optimeringsalgoritme for å løse betingede optimeringsproblemer. Metoden er basert på å forenkle problemet til et lineært delproblem som løses i hver iterasjon, og fungerer spesielt bra når delproblemet har en løsning på lukket form.

Algoritmen ble opprinnelig utviklet for å løse problemer i euklidske rom, men det kan ofte være nyttig å jobbe med problemer på mangfoldigheter. I noen tilfeller er løsningen til et euklidisk problem begrenset til å ligge på en mangfoldighet, og i andre tilfeller er dataene eller domenet til objektivfunksjonen allerede definert på mangfoldigheten.

I denne oppgaven studerer vi konvekse problemer på mangfoldigheter, hvor vi i tillegg til mangfoldighetsbetingelsen også har ytterligere betingelser på løsningen. Vi ser på en generalisering av den klassiske Frank–Wolfe-algoritmen til riemannske rom, og undersøker hvordan den skiller seg fra den euklidiske versjonen – spesielt med tanke på hvordan delproblemet ikke lenger er lineært, og dermed må løses med mer avanserte teknikker.

Vårt hovedbidrag ligger i å implementere algoritmen som en generell løser for problemer på vilkårlige riemannske mangfoldigheter, ved å bruke grensesnittet til Manopt.jl, et Julia-bibliotek for optimering på mangfoldigheter. Vi tester implementasjonen vår på den symmetriske positivt bestemte matrisemangfoldigheten, men vi merker oss at ytelsen til algoritmen ikke er så god som forventet.

Preface

This thesis concludes my six years as a student in the applied physics and mathematics programme at NTNU. While there have been both ups and downs, it has been an incredible experience, and I am grateful for all the people who have contributed to that.

Nothing quite prepares you for the work that goes into writing a thesis – for the countless hours spent debugging code, and the late nights trying to figure out why that one proof doesn't check out. Nor is there anything that will prepare you for how fast the months fly by; at some point you suddenly go from having nearly all the time in the world on your hands, to not seeing how on earth you could possibly finish the thing in time. And yet, in the end, everything usually works out after all.

I would like to thank my supervisor, Ronny Bergmann, for our many insightful discussions and his helpful feedback, as well as for his near-endless patience in teaching me about differential geometry. It has been challenging at times, navigating through this unfamiliar branch of mathematics, but in the end it has been a fascinating journey.

Even Stephansen Kjemsås
July 2022
Trondheim

Contents

Chapter 1

Introduction

Starting with the development of linear programming in the middle of the 20th century – just as electronic computers were becoming capable of applying these techniques in practice – mathematical optimization has seen an increasing number of useful applications and developments in the past few decades [1]. Although this work was initially limited to problems on Euclidean domains, there has since been an increased interest in optimization in non-Euclidean space. Recently, the efforts towards generalizing Euclidean optimization techniques to Riemannian manifolds have intensified – in part due to the wide variety of successful applications that have emerged, within fields such as machine learning, robotics and signal processing [2]. In addition to the theoretical developments, a large effort has also gone into the computational aspects of adapting the familiar and well-tried algorithms to work on manifolds: particularly with the *Manopt* projects – aiming to implement general solvers for Riemannian optimization problems in Matlab[3], Python [4] and Julia [5].

We consider mathematical optimization problems on some Riemannian manifold $\mathcal{M}$, where the goal is to find the extremal values of a given *cost function* $f\colon \mathcal{M} \to \mathbb{R}$, possibly subject to some *constraint* on the solution. The basic optimization problem on manifolds is formulated as

$$\arg\min_{p\in\mathcal{M}} f(p).$$

Note that by choosing $\mathcal{M} = \mathbb{R}^d$ as our manifold, we are back in the Euclidean setting, and thus this can be viewed as a generalization of the classical case. In many cases, we will also want to constrain the solution to being within a given region of the manifold. In the constrained case, we additionally restrict our search for a solution to some subset $\mathcal{C} \subseteq \mathcal{M}$ – called the *feasible set* – and look for the point in $\mathcal{C}$ which is closest to a solution of the unconstrained problem:

$$\arg\min_{p\in\mathcal{C}} f(p).$$

Although many optimization problems on manifolds can technically be solved by embedding the manifold in $\mathbb{R}^n$ and using techniques for constrained Euclidean optimization, there can be a lot to gain from working natively on the manifold. Working directly on the manifold instead of embedding it into a higher-dimensional space – if such an embedding is even available – allows us to exploit the underlying geometric structure of the manifold, possibly resulting in better performance [6]. Further, it

reduces the dimensionality of the problem, which can also affect the ease of finding a solution. Working with manifolds can also have considerable benefits when working with constrained Euclidean optimization problems. In fact, many common constraints in Euclidean optimization – such a requiring the solution to be a point on the n-dimensional sphere surface, or a symmetric positive-definite matrix – can be viewed as unconstrained problems on the appropriate manifold.

While this does take care of certain classes of constraints for Euclidean problems, only working with unconstrained problems on manifolds severely limits the scope of the applications we can handle. We often want to solve problems that are not only defined on a manifold, but where we have additional restrictions on the space of feasible points.Just like in the Euclidean setting, constrained optimization on manifolds is – in general – more difficult than in the unconstrained case, and thus requires more sophisticated techniques to handle.

A property shared by several methods that have been developed for constrained manifold optimization, like projected gradient descent [7] or proximal gradient methods [8], is that they rely on projecting the iterates onto the feasible set at each iteration. In some cases, these projections can be quite costly operations, making such *projection-based* algorithms less suitable for these types of problems [9].

1.1 The Frank–Wolfe algorithm

The *Frank–Wolfe algorithm* – also known as the *conditional gradient method* – for constrained optimization in Euclidean space was developed in 1956 by Marguerite Frank and Philip Wolfe [10]. Originally formulated for solving quadratic problems on polyhedral domains, it can be adapted to solve arbitrary constrained, convex optimization problems. The algorithm marks a historical tipping point in the field of optimization – being one of the first of its kind, able to tackle general constrained convex optimization [11].

The Frank–Wolfe algorithm is an iterative algorithm based on simplifying the problem to a *linear subproblem* on the form

$$\arg\min_{z \in \mathcal{C}} \langle \nabla f(x), z \rangle,$$

constrained to the same feasible set as the original problem. In each iteration of the algorithm, the main task is to compute this subproblem, which gives the next point to move towards. The next step is then taken along a straight line between the current iterate and the solution to the subproblem. Since we don't want to do any projection back to the feasible set, an important assumption for using the algorithm is therefore that the feasible set is convex.

The method assumes access to a *linear oracle* that is able to output the solution to the subproblem at a given iteration. For many problems, there exist closed-form solutions to the subproblem, which means that the oracle can be implemented in a very efficient manner. Even when this is not the case, the oracle can still be implemented by solving the subproblem with e.g. the simplex method.

Since the subproblem is solved with the same constraint as the main problem, each new iterate – chosen along a line between the current iterate and the subproblem solution – will always stay within the feasible set. This is the main advantage of Frank–Wolfe: Since we can guarantee that each iterate will be within the feasible set, there is no need to project back to the feasible set at any point. Thus, the algorithm is able to efficiently

solve problems even when these projections are expensive to do, as long as we have access to an oracle. The drawback is that the algorithm converges at a quite slow rate, $\mathcal{O}(1/k)$ in the number of iterations k, which makes it less useful in cases where we can easily project onto the feasible set, and hence use a faster algorithm. In the Riemannian setting there are, however, many problems where projecting onto the constraint set is expensive [9] – making algorithms that don't require projections very useful.

In 2017 Melanie Weber and Suvrit Sra [12] published a paper (with a new and considerably expanded revision from 2021 [9]) in which they expanded the algorithm to work in the Riemannian setting. In addition to demonstrating the algorithm on various example problems, they also proved that the Riemannian Frank–Wolfe algorithm retained all the desirable properties of the Euclidean version – including the convergence (albeit still slow) of the algorithm. Just like in the Euclidean setting, the Riemannian algorithm requires access to an oracle that can solve the subproblem, and a large part of their work concerns examples of practical settings where such an oracle can be implemented efficiently.

1.2 Our contribution

Our work expands on the work by Weber and Sra by implementing the Frank–Wolfe algorithm as a general-purpose solver for constrained Riemannian optimization problems. The algorithm is implemented in *Julia* using the framework provided by *Manopt.jl* [5]. Whereas Weber and Sra have implemented the algorithm for the specific problems they have considered, our implementation can be used with arbitrary manifolds and cost functions, as long as the user has access to the appropriate oracle. We also extend the algorithm to working with iterative solvers for the subproblem, for cases where a closed-form oracle either isn't available or can't be implemented in an efficient manner.

The remainder of the thesis is structured as follows:

Chapter 2 contains a brief presentation of some background matter necessary for working with optimization on manifolds. This lets us properly define all the concepts we work with in the rest of the thesis, and also serves the purpose of fixing and clarifying our notation. This includes an introduction to manifolds and related concepts, and a short summary of optimization in Euclidean space and on manifolds. Additionally, we briefly present some useful notions from the Riemannian geometry of the manifold of symmetric positive-definite matrices, $\mathcal{P}_n$, and the special orthogonal group, $\mathrm{SO}(n)$.

The main body of the thesis begins in chapter 3, where we introduce the Euclidean Frank–Wolfe algorithm and its properties. This lets us introduce the algorithm in a familiar setting, before moving it into the realm of manifolds in the next chapter. We also provide a summary of the well-established convergence properties of the algorithm.

The Riemannian Frank–Wolfe algorithm is introduced in chapter 4, and we provide some motivation for how it is derived, as a generalization of the Euclidean algorithm. Further, we discuss the differences when using the algorithm on Riemannian manifolds, in particular with regard to solving the subproblem. We show how the subproblem can be solved iteratively with gradient-based optimization methods, and we also provide closed-form solutions for example problems in $\mathcal{P}_n$ and $\mathrm{SO}(n)$.

Chapter 5 contains proofs of the convergence of the algorithm, verifying that the most important convergence properties of the Euclidean algorithm carry over to the Riemannian setting. We provide proofs for the global convergence of Riemannian Frank–Wolfe, both when the subproblem is solved exactly and when it is only solved approximately. Further, we show that the algorithm converges linearly in the special case where

the solution is located strictly in the interior of the constraint set.

In chapter 6, we discuss in detail how we have implemented the algorithm. We also provide a numerical demonstration on the manifold of symmetric positive-definite matrices, comparing the performance of different variations of the algorithm.

Finally, in chapter 7, we give a summary of our work, and provide a brief outlook on further work on the topic.

Chapter 2

Background

2.1 A brief introduction to manifolds

This preliminary section is meant to give the reader a brief introduction to manifolds, by introducing the necessary concepts required to work with optimization in the Riemannian setting. In addition to providing a short summary of Riemannian optimization, it also lets us fix the notation we are going to use in the rest of the thesis. We define several important notions that we will use in the rest of the thesis, starting with what a manifold is, and continuing with curves, tangents, differentials, inner products, geodesics and the exponential and logarithmic maps.

For a more thorough overview of Riemannian geometry and manifold optimization, we refer to one of the many textbooks on the subject – for instance the books by Absil, Mahony and Sepulchre [13], do Carmo [14] or Boumal [2]. The contents of this first section has, for the most part, been adapted from these books. Note that I also wrote a very similar preliminary section for my *Industrial Mathematics Specialization Project (TMA4500)*, titled *Convolutions on Manifolds for Deep Learning*. Since this project isn't published, we do not formally cite it here, but we note that this section will necessarily have some overlap with that work.

Manifolds

A *manifold* $\mathcal{M}$ is a topological space of dimension d that is second-countable, Hausdorff and locally homeomorphic to $\mathbb{R}^d$. That is, all pairs of distinct points on $\mathcal{M}$ have disjoint neighbourhoods, its topology has a countable basis, and for each point p on the manifold, there exists a neighbourhood that is homeomorphic to an open subset of $\mathbb{R}^d$. Informally, we can say that manifolds *locally resemble* $\mathbb{R}^d$. We note especially that $\mathbb{R}^n$ is a manifold, and thus everything defined in this section also applies to this space. We will use the convention of $x, y, \dots$ to denote points in euclidean space, and $p, q, \dots$ to denote manifold points.

Charts and atlases

A *chart* of a manifold $\mathcal{M}$ is a bijective map ϕ from a subset $U \subset \mathcal{M}$ to an open subset of $\mathbb{R}^d$, denoted (U, ϕ). An *atlas* of $\mathcal{M}$ is a disjoint union of charts (U_α, ϕ_α) that covers $\mathcal{M}$, that is, $\bigcup_\alpha U_\alpha = \mathcal{M}$. Thus, the manifold can be described locally by a single chart, and globally by the atlas of charts.

Additionally, in order for the manifold to be *differentiable*, we require that the transition between charts in the atlas is smooth in the Euclidean sense, i.e. C^∞ differentiable. Thus, for any two overlapping charts, $U_\alpha \cap U_\beta \neq \emptyset$, the sets $\phi_\alpha(U_\alpha \cap U_\beta) := R_\alpha$ and $\phi_\beta(U_\alpha \cap U_\beta) := R_\beta$ must be open sets in $\mathbb{R}^d$, and the change of coordinates

$$\phi_\beta \circ \phi_\alpha^{-1} = \phi_\beta\left(\phi_\alpha^{-1}(x)\right) : R_\alpha \to R_\beta$$

should be smooth. In the following, we will work only with differentiable manifolds.

A mapping between two manifolds $f\colon \mathcal{M} \to \mathcal{N}$ is called *smooth* if, for all charts (U, ϕ), with $\phi\colon U \to R \subseteq \mathbb{R}^d$, the function

$$f \circ \phi^{-1}\colon R \to V \subseteq \mathcal{N}$$

is smooth in the Euclidean sense. We let $C^\infty(\mathcal{M})$ denote the set of real-valued smooth functions on $\mathcal{M}$.

Curves and tangent vectors

A *curve* on a manifold $\mathcal{M}$ is a smooth mapping $\gamma\colon [0, T] \to \mathcal{M}$.The choice of T is arbitrary – leading only to a different scaling – but we will mostly use $T = 1$, so that the curve maps the interval $[0, 1]$ to the manifold.

At each point p on the manifold, we define a set of *tangent vectors*. Consider any curve γ starting at p, i.e. $\gamma(0) = p$. A tangent vector X at p is the linear operator $\dot\gamma\colon C^\infty(\mathcal{M}) \to \mathbb{R}$ such that, for any function $f \in C^\infty(\mathcal{M})$,

$$\dot\gamma(0)f = \left(\frac{\mathrm{d}}{\mathrm{d}t}(f \circ \gamma)\right)(0).$$

As we can construct a curve passing through a point in any direction (since the manifold is locally homeomorphic to $\mathbb{R}^d$), and we can give the curve arbitrary velocity (by choosing an appropriate value for T), there exist tangent vectors with arbitrary orientation and magnitude. We denote tangent vectors by $X, Y, \ldots$.

Further, we can define the *tangent space* $\mathcal{T}_p\mathcal{M}$ at $p \in \mathcal{M}$. This is a vector space of the same dimension as $\mathcal{M}$, containing its tangent vectors at p. The disjoint union of all tangent vectors at every point of $\mathcal{M}$ is also a manifold, called the *tangent bundle*, $T\mathcal{M} = \bigcup_p \mathcal{T}_p\mathcal{M}$.

A *vector field* is a smooth function $\xi\colon \mathcal{M} \to T\mathcal{M}$ that assigns a tangent vector $X \in \mathcal{T}_p\mathcal{M}$ to each point $p \in \mathcal{M}$. We describe the action of a vector field ξ on a (smooth) function f as $(\xi f)(p) = Df[\xi(p)](p)$, so that $\xi f\colon \mathcal{M} \to \mathbb{R}$.

Differentials

Let $f\colon \mathcal{M} \to \mathcal{N}$ be a smooth mapping between two manifolds M and $\mathcal{N}$, and let $X \in \mathcal{T}_p\mathcal{M}$ be a tangent vector at the point $p \in \mathcal{M}$, defined by a curve γ such that $\gamma(0) = p$. The *differential*, or *directional derivative*, of f at p along X is the linear map $D_pf[X]\colon \mathcal{T}_p\mathcal{M} \to \mathcal{T}_{f(p)}\mathcal{N}$ defined by

$$D_pf(p)[X] = \left.\frac{\mathrm{d}}{\mathrm{d}t} f\big(\gamma(t)\big)\right|_{t=0} = (f \circ \gamma)'(0).$$

Next, we define the *adjoint differential* operator, $(D_pf)^*\colon \mathcal{T}_{f(p)}\mathcal{M} \to \mathcal{T}_p\mathcal{M}$, as the operator fulfilling [15]

$$\langle D_pf(p)[X], Y\rangle_{f(p)} = \langle X, (D_pf)^*(p)[Y]\rangle_{f(p)}, \quad X, Y \in \mathcal{T}_{f(p)}\mathcal{M}.$$

The familiar *chain rule* for differentiation in Euclidean space has an analouge on manifolds [16]. Let $g\colon \mathcal{M} \to \mathcal{M}$ and $h\colon \mathcal{M} \to \mathcal{M}$ be two functions on a manifold $\mathcal{M}$, and let $f\colon \mathcal{M} \to \mathcal{M}$ be their composition,

$$f(p) = (g \circ h)(p) = g\big(h(p)\big).$$

Further, let $p \in \mathcal{M}$ and $X \in \mathcal{T}_p\mathcal{M}$. Then the differential $D_p f(p)[X]$ is given by

$$D_p f(p)[X] = D_{h(p)} g(q)\big[D_p h(p)[X]\big],$$

where the outer differential on the right-hand side is evaluated at $q = h(p)$.

The Riemannian metric and distances

We can also introduce a notion of distance on the tangent space: the *Riemannian metric* on $\mathcal{T}_p\mathcal{M}$. This is an inner product, i.e. a bilinear, symmetric positive-definite form, $\langle \cdot, \cdot \rangle_p\colon \mathcal{T}_p\mathcal{M} \times \mathcal{T}_p\mathcal{M} \to \mathbb{R}$, varying smoothly with p. That is, for any two smooth vector fields ξ, η on $\mathcal{M}$, the function $g(p) = \langle \xi(p), \eta(p) \rangle_p$ is smooth. The Riemannian metric induces a norm on the tangent space,

$$||X||_p = \sqrt{\langle X, X \rangle_p}.$$

We sometimes omit the subscript if it is clear from context which tangent space we are considering. If every tangent space $\mathcal{T}_p\mathcal{M}$ can be endowed with a Riemannian metric, we say that $\mathcal{M}$ is a *Riemannian manifold*.

We can define a notion of distance on Riemannian manifolds, $\mathrm{dist}\colon \mathcal{M} \times \mathcal{M} \to \mathbb{R}$, as the length of the shortest curve joining two points p and q:

$$\mathrm{dist}(p, q) = \inf L(\gamma),$$

where $L(\gamma)$ is the length of the curve $\gamma\colon [0, T] \to \mathcal{M}$, defined as

$$L(\gamma) = \int_0^T ||\dot{\gamma}(t)||_p \, \mathrm{d}t.$$

We define an *(open) ball* of radius ρ around $p \in \mathcal{M}$ as the subset $\mathcal{B}_\rho(p) \subseteq \mathcal{M}$ such that

$$\mathcal{B}_\rho(p) = \{q \in \mathcal{M} \mid \mathrm{dist}(p, q) < \rho\}.$$

Affine connections

A note on the notation used in this definition: The nabla symbol ∇ is often used in the literature to represent affine connections, while we will usually prefer to use it to represent the Euclidean gradient on a vector space. In keeping with Absil et al. and do Carmo, we have used the symbol to represent affine connections in this section. However, as we don't directly use affine connections in the rest of the thesis, there is little risk of ambiguity, and so in later sections ∇ will mean the Eucldiean gradient unless otherwise noted.

Denoting by $\mathfrak{X}(\mathcal{M})$ the set of all smooth vector fields on $\mathcal{M}$, an *affine connection* on $\mathcal{M}$ is a bilinear mapping $\nabla\colon \mathfrak{X}(\mathcal{M}) \times \mathfrak{X}(\mathcal{M}) \to \mathfrak{X}(\mathcal{M})$, denoted $(\eta, \xi) \xrightarrow{\nabla} \nabla_\eta \xi$, satisfying the following:

1. Linearity in η (over real-valued functions): $\quad \nabla_{f\eta + g\chi}\xi = f\nabla_\eta \xi + g\nabla_\chi \xi$

2. Linearity in ξ (over scalars): $\nabla_\eta(a \cdot \xi + b \cdot \zeta) = a\nabla_\eta\xi + b\nabla_\eta\zeta$

3. Product rule (Leibniz' law): $\nabla_\eta(f\xi) = (\eta f)\xi + f\nabla_\eta\xi$,

where η, χ, ξ, ζ are vector fields on $\mathcal{M}$, f, g are smooth, real-valued functions on $\mathcal{M}$ and $a, b \in \mathbb{R}$. For a given affine connection ∇, the vector field $\nabla_\eta\xi$ is called the *covariant derivative* of ξ with respect to η.

Given two vector fields η, ξ on $\mathcal{M}$, their *Lie bracket* is defined as

$$[\eta, \xi]f = \eta(\xi f) - \xi(\eta f).$$

On Riemannian manifolds there exists a (unique) preferred affine connection, called the *Levi-Civita connection*, which satisfies two important properties: First, it is *symmetric*, meaning that it satisfies

$$\nabla_\eta\xi - \nabla_\xi\eta = [\eta, \xi] \quad \forall \eta, \xi \in \mathfrak{X}(\mathcal{M}).$$

Second, it preserves the Riemannian metric, so that

$$\chi\langle\eta, \xi\rangle_p = \langle\nabla_\chi\eta, \xi\rangle_p + \langle\eta, \nabla_\chi\xi\rangle_p.$$

Let $\gamma\colon [0, 1] \to \mathcal{M}$ be a smooth curve on $\mathcal{M}$. We define the *induced covariant derivative* (iinduced by ∇) as the (unique) operator $\frac{\mathrm{D}}{\mathrm{d}t}\colon \mathfrak{X}(\gamma) \to \mathfrak{X}(\gamma)$ satisfying

1. $\frac{\mathrm{D}}{\mathrm{d}t}(a \cdot \xi + b \cdot \zeta) = a\frac{\mathrm{D}}{\mathrm{d}t}\xi + b\frac{\mathrm{D}}{\mathrm{d}t}\zeta$
2. $\frac{\mathrm{D}}{\mathrm{d}t}(f\xi) = f'\xi + f\frac{\mathrm{D}}{\mathrm{d}t}\xi$
3. $\frac{\mathrm{D}}{\mathrm{d}t}(\eta \circ \gamma)(t) = \nabla_{\dot\gamma(t)}\eta$,

and we define the *acceleration vector field* on γ as

$$\frac{\mathrm{D}^2}{\mathrm{d}t^2}\gamma = \frac{\mathrm{D}}{\mathrm{d}t}\left(\frac{\mathrm{d}\gamma}{\mathrm{d}t}\right) = \frac{\mathrm{D}}{\mathrm{d}t}\dot\gamma.$$

Parallel transport

Given an affine connection ∇ on $\mathcal{M}$, a vector field ξ along a curve γ is called *parallel* if it satisfies $\frac{\mathrm{D}}{\mathrm{d}t}\xi = 0$ for any t. Given some real value t_0 such that $\gamma(t_0) = p$ and tangent vector $X \in \mathcal{T}_p\mathcal{M}$, there exists a unique parallel vector field ξ along γ satisfying

$$\xi(\gamma(t_0)) = \xi(p) = X.$$

As the vector field is parallel, the vectors $\xi(\gamma(t))$ remain unchanged along the curve. The operation $P^\gamma_{t\leftarrow t_0}$ sending the vector field from $\xi(\gamma(t_0))$ to $\xi(\gamma(t))$ along γ is called *parallel transport*. We will sometimes leave out the curve in the notation, denoting it just $P_{t\leftarrow t_0}$.

Whereas in Euclidean space, moving a vector in $\mathcal{T}_p\mathcal{M}$ from p to q along a given path γ_a will always have the same result as moving it along another path γ_b, this is not necessarily the case on a manifold. The vector may be rotated along the path, and thus parallel transport is *path-dependent*.

Geodesics and the exponential and logarithmic maps

A *geodesic* is a generalization of the notion of a straight line between two points p and q. Let $\mathcal{M}$ be a manifold endowed with an affine connection ∇. For two points $p, q \in \mathcal{M}$, define a curve $\gamma : [0, T] \to \mathcal{M}$ such that $\gamma(0) = p$ and $\gamma(T) = q$. γ is a geodesic if it has zero acceleration for all $t \in [0, T]$, that is,

$$\frac{\mathrm{D}}{\mathrm{d}t}\left(\frac{\mathrm{d}\gamma}{\mathrm{d}t}\right) = \frac{\mathrm{D}}{\mathrm{d}t}\dot{\gamma} = 0$$

Further, a *shortest geodesic* between p and q is the geodesic connecting the two points that minimizes the distance between them. Note that the shortest geodesic is not necessarily unique – there may be more than one shortest path (take e.g. the north and south poles of the sphere $\mathcal{S}^2$). However, for points sufficiently close, a unique shortest geodesic exists.

When defining a (shortest) geodesic by the two points it connects, we will use the convention that $t \in [0, 1]$, i.e. $T = 1$. Thus, given two points $p, q \in \mathcal{M}$, we denote a geodesic γ connecting the two points, with $\gamma(0) = p$ and $\gamma(1) = q$, by $\gamma_{p \to q}$. Then the point located at "distance" $t \in (0, 1)$ from p to q is denoted $\gamma_{p \to q}(t)$.

We now define the *exponential map* on the manifold, $\mathrm{Exp}_p \colon T_p\mathcal{M} \to \mathcal{M}$, mapping a vector X in the tangent space at p to a new point on the manifold.

Given a tangent vector $X \in \mathcal{T}_p\mathcal{M}$, there exists a unique geodesic $\gamma \colon [0, T] \to \mathcal{M}$ (for some T), such that $\gamma(0) = p$ and $\dot{\gamma}(0) = X$. Then the exponential map is defined as

$$\mathrm{Exp}_p(X) = \gamma(1).$$

In other words, $\mathrm{Exp}_p X$ follows the geodesic from p in the direction of X for a "distance" of unit length. When, for every point $p \in \mathcal{M}$, the exponential map is defined on the entire tangent space $\mathcal{T}_p\mathcal{M}$, we say that $\mathcal{M}$ is *geodesically complete*. That is, for any geodesic γ starting at p, $\gamma(t)$ is defined for any value of $t \in \mathbb{R}$. In fact, geodesic completeness coincides with the notion of completeness for $\mathcal{M}$ (with the Riemannian distance) as a metric space [13].

We can also define the *logarithmic map*, which is the inverse of the exponential:

$$\mathrm{Log}_p\, q = X \quad \iff \quad \exp_p X = q,$$

i.e. the logarithmic map at p takes a manifold point q and returns the tangent vector $X \in \mathcal{T}_p\mathcal{M}$ such that $\mathrm{Exp}_p\, X = q$. Note that the logarithmic map is not always uniquely defined, but there exists a neighbourhood of p where X has an inverse – i.e. for any q in the neighbourhood, there exists a unique tangent vector X such that $\mathrm{Log}_p\, q = X$. This neighbourhood is called the *injectivity radius* of p. The injectivity radius of the manifold itself is the infimum of the injectivity radii at every point $p \in \mathcal{M}$.

Hadamard manifolds

We use the *sectional curvature* of a Riemannian manifold with demension $d > 1$ to describe the infinitesimal geometry of the manifold.

For any three points $p, q, r \in \mathcal{M}$, we define a *geodesic triangle* as a triangle constining of three geodesics

$$\gamma_{p \to q}, \quad \gamma_{p \to r}, \quad \gamma_{r \to q}$$

connecting the three points. A *comparison triangle* is a corresponding triangle in $\mathbb{R}^2$ with vertices $\bar{p}, \bar{q}, \bar{r}$ such that its sides have the same length as these geodesics, that is,

$$\text{dist}(p,q) = ||\bar{q} - \bar{p}||, \quad \text{dist}(p,r) = ||\bar{r} - \bar{p}||, \quad \text{dist}(q,r) = ||\bar{r} - \bar{q}||.$$

Let $u = \gamma_{p\to q}(t_u)$ and $v = \gamma_{p\to r}(t_v)$, $t_u, t_v \in [0,1]$, be points on two of the geodesics. We say that $\mathcal{M}$ has nonpostitive sectional curvature if, for every such geodesic triangle and points u, v, with corresponding points $\bar{u}, \bar{v}$ on its comparison triangle,

$$\text{dist}(u,v) \leq ||\bar{u} - \bar{v}||.$$

A manifold with nonpositive sectional curvature is called a *CAT(0) manifold*, meaning that its sectional curvature is upper bounded at 0. A Riemannian manifold that is geodescially complete, simply connected and has nonpositive sectional curvature is called a *Hadamard manifold* [17].

2.2 Optimization basics

We now introduce a few basic concepts from optimization on $\mathbb{R}^n$. For a more comprehensive introduction to this topic, we refer to e.g. the textbooks by Boyd and Vandenberghe[18] or Nocedal and Wright[19]. Although we describe these concepts in an Euclidean setting, much of it is also applicable (or very similar) in the Riemannian setting. Anything for which this is not the case will be further detailed in section 2.3.

Problems, solutions and constraints

An optimization problem is partially defined by its *objective function* $f\colon \mathbb{R}^n \to \mathbb{R}$, or *cost function*, which is the function we want to find the extrema of. The convention is to solve optimization problems as minimization problems; if we instead wish to maximize the function f, we can simply minimize its negative, $-f$. A general (constrained) optimization problem in $\mathbb{R}^n$ is formulated as

$$\arg\min_{x\in\mathcal{C}} f(x). \tag{2.1}$$

For any given problem, we denote the *optimal solution* (or just *solution*) by x^*. If the problem has more than one optimal solution, we usually don't care which one we find. In such cases, by a slight abuse of notation we will refer to the set of optimal solutions as *the* solution x^*.

For a given cost function f, we say that an iterate x_k is an *ϵ-approximate solution* of problem (2.1) if

$$f(x_k) - f(x^*) \leq \epsilon.$$

For constrained optimization problems, we define the *feasible set*, or *constraint set*, $\mathcal{C} \subseteq \mathbb{R}^n$ in which we want to look for a solution. If the solution is not contained in the feasible set, we seek the point in $\mathcal{C}$ which gives the best possible function value. We can use the same formulation when considering unconstrained problems: we just replace $\mathcal{C}$ by $\mathbb{R}^n$.

The constraints are often given on the form

$$\begin{aligned} g_i(p) &\leq 0 \\ h_j(p) &= 0, \end{aligned} \tag{2.2}$$

for a given set of indices $i = 1, \dots, N$ and $j = 1, \dots, M$. Thus, $\mathcal{C}$ is defined to be the set of points satisfying the (in)equalities in (2.2) for all these indices.

In Euclidean space, a constraint set $\mathcal{C} \in \mathbb{R}^n$ compact if and only if it is closed and bounded. This is important to ensure that the solution is actually reachable within a finite number of steps.

Convexity

The set $\mathcal{S} \subseteq \mathbb{R}^n$ is said to be *convex* if, for all $x, y \in \mathcal{S}$, then any point $z \in \mathbb{R}^n$ satisfying

$$z = \eta x + (1 - \eta)y, \quad \forall \eta \in [0, 1],$$

is also in $\mathcal{S}$. In other words, any point z on a line segment between two points $x, y \in \mathcal{S}$ is also in $\mathcal{S}$. The term $\eta x + (1 - \eta)y$ is known as a *convex combination* of x and y

Similarly, a function $f \colon \mathbb{R}^n \to \mathbb{R}$ is said to be *convex* if, for any $x, y \in \mathbb{R}^n$,

$$f\left(\eta x + (1 - \eta)y\right) \leq \eta f(x) + (1 - \eta) f(y), \quad \forall \eta \in [0, 1]. \tag{2.3}$$

That is, the line segment (in $\mathbb{R}^{n+1}$) going from $\big(x, f(x)\big)$ to $\big(y, f(y)\big)$ lies above the graph of f [18].

Optimality and search directions

Consider the problem (2.1). If the constraint set $\mathcal{C}$ and objective function f are convex and x^* is a local optimum, then x^* is also a global optimum [19]. Additionally,

$$\langle \nabla f(x^*), y - x^* \rangle \geq 0 \quad \forall y \in \mathcal{C}. \tag{2.4}$$

In other words, if x is the current iterate of the algorithm, and there exists some $y \in \mathcal{C}$ such that

$$\langle \nabla f(x), y - x \rangle < 0,$$

then x is not an optimum of the function. This means that there must be some direction $d = y - x$ we can move towards in which the value of the objective function is lower. If, additionally, we can move in this direction without leaving the feasible set, we call it a *feasible descent direction*. Further, if $\mathcal{C}$ is convex, all points on the line from x to y will be contained in $\mathcal{C}$, so there must be a feasible descent direction.

2.3 Manifold optimization

This section provides an introduction to some important concepts relating to Riemannian optimization. For a more extensive treatment of this subject we refer to the books by Absil et al. [13] and Boumal [2]. For convenience, we repeat the general formulation of a constrained Riemannian optimization problem:

$$\arg\min_{p \in \mathcal{C}} f(p). \tag{2.5}$$

We again denote the cost function by f, but now it maps from $\mathcal{M}$ to $\mathbb{R}$. Riemannian constraints work in the same way as they do in Euclidean space, and again we can return to the unconstrained case by replacing $\mathcal{C}$ by $\mathcal{M}$. Note that we denote manifold points by $p, q, \dots \in \mathcal{M}$, as opposed to $x, y, \dots \in \mathbb{R}^n$. Unlike in $\mathbb{R}^n$, we now have to make a distinction between points and tangent vectors, so these will be denoted $X, Y, \dots \in \mathcal{T}_p\mathcal{M}$.

The Riemannian gradient

While the familiar Euclidean gradient is defined for functions on manifolds (embedded in Euclidean space), the Euclidean gradient at point p will, in general, not be in the tangent space $\mathcal{T}_p\mathcal{M}$. For a function $f\colon \mathcal{M} \to \mathbb{R}$, we can instead define the *Riemannian gradient* $\operatorname{grad} f(p)$ as the tangent vector at p that satisfies [16]

$$\langle \operatorname{grad} f(p), X \rangle_p = D_p f(p)[X] \quad \forall X \in \mathcal{T}_p\mathcal{M}.$$

If $\mathcal{M}$ is embedded in some Euclidean space $\mathbb{R}^n$, the Riemannian gradient at p is the orthogonal projection of the Euclidean gradient onto $\mathcal{T}_p\mathcal{M}$ [2]. If there may be some ambiguity regarding which point p we are taking the gradient with respect to, we will sometimes use a subscript grad_p to clarify this, but in most cases this should be clear from the context.

Lipschitz smoothness

In later sections, we will sometimes need the cost function to satisfy a certain notion of smoothness [9]. A function $f\colon \mathcal{M} \to \mathbb{R}$ is said to have *L-Lipschitz* gradients if there exists a constant L such that for any $p, q \in \mathcal{C}$ we have

$$|| \operatorname{grad} f(q) - P_{q \leftarrow p} \operatorname{grad} f(p)|| \leq L \cdot d(p, q), \quad \forall p, q \in \mathcal{M}. \tag{2.6}$$

For a given constant L, we say that a function satisfying (2.6) is *L-smooth*. An equivalent definition of Lipschitz continuity that we will also use is

$$f(q) - f(p) \leq \left\langle \operatorname{grad} f(p), \operatorname{Log}_p(q) \right\rangle_p + \frac{L}{2} d(p, q)^2 \quad \forall p, q \in \mathcal{M}. \tag{2.7}$$

(Geodesic) convexity

In the Riemannian setting, we can define a similar notion of convexity as for Euclidean domains [2]. A subset $\mathcal{S} \subseteq \mathcal{M}$ is said to be *geodesically convex* if, for every set of points $p, q \in \mathcal{S}$, there exists a geodesic segment $\gamma\colon [0, 1] \to \mathcal{M}$ such that $\gamma(0) = p$, $\gamma(1) = q$ and $\gamma(t) \in \mathcal{S}$ for all $t \in [0, 1]$. That is, for any geodesic connecting two points in $\mathcal{S}$, every point on the geodesic segment between the two points is also within $\mathcal{S}$. When working in the Riemannian setting, we will often refer to geodesic convexity as just convexity when there is no risk of ambiguity.

We can now define geodesic convexity for functions. Consider a geodesically convex subset, $\mathcal{S} \subseteq \mathcal{M}$, and a function $f\colon \mathcal{S} \to \mathbb{R}$. Given two points $p, q \in \mathcal{S}$ with a geodesic $\gamma_{p \to q}$ connecting them, define $g\colon [0, 1] \to \mathbb{R}$ such that

$$g(t) = (f \circ \gamma_{p \to q})(t) = f(\gamma_{p \to q}(t)).$$

The function f is said to be geodesically convex on $\mathcal{S}$ if, for any choice of points $p, q \in \mathcal{S}$, g satisfies the Euclidean definition of convexity (2.3).

An equivalent definition of a geodesically convex function on $\mathcal{S} \subseteq \mathcal{M}$, which we will more often make use of, is [9]

$$f(q) - f(p) \geq \langle X, Log_p(q) \rangle_p \quad \forall p, q \in \mathcal{S},\ X \in \mathcal{T}_p\mathcal{M}. \tag{2.8}$$

Further, f is *μ-strongly geodesically convex* if it satisfies

$$f(q) - f(p) \geq \langle X, Log_p(q) \rangle_p + \frac{\mu}{2} d(p, q)^2 \quad \forall p, q \in \mathcal{S},\ X \in \mathcal{T}_p\mathcal{M}. \tag{2.9}$$

For Hadamard manifolds, the notions of geodesic convexity and μ-strong geodesic convexity are equivalent [2].

Optimality

Consider the problem (2.5). Assume the constraint set $\mathcal{C}$ and the objective function f are both geodesically convex. If p^* is a local optimum for the problem, then p^* is also a global optimum. Further, the following inequality holds [9]:

$$\left\langle \operatorname{grad} f(p^*), \operatorname{Log}_{p^*}(q) \right\rangle_{p^*} \geq 0 \quad \forall q \in \mathcal{C}. \tag{2.10}$$

Just like in the Euclidean case, if p is the current iterate of the algorithm, and there exists some $q \in \mathcal{C}$ such that

$$\left\langle \operatorname{grad} f(p), \operatorname{Log}_{p}(q) \right\rangle_{p^*} < 0,$$

then p is not an optimum of the function. Due to the convexity of $\mathcal{C}$, this again means that there must be a feasible descent direction.

Finally, we also introduce the notion of compactness in the general (non-Euclidean) setting. A collection of subsets $\mathcal{A}$ of a set $\mathcal{S} \subseteq \mathcal{M}$ is said to *cover* $\mathcal{S}$ if the union of all its elements is equal to $\mathcal{S}$,

$$\mathcal{S} = \bigcup_{\alpha \in \mathcal{A}} .$$

The covering is called *open* if all its elements are open subsets of $\mathcal{S}$. A set $\mathcal{S} \subseteq \mathcal{M}$ is *compact* if every open covering of $\mathcal{S}$ contains a finite subcollection $\mathcal{F}$ that also covers $\mathcal{S}$:

$$\mathcal{S} = \bigcup_{\phi \in \mathcal{F}} .$$

In Euclidean space, this is in fact equivalent to the subset $\mathcal{S} \subseteq \mathbb{R}^n$ being closed and bounded, and – just like in the Euclidean case – compactness is important to ensure that the solution is reachable within a finite number of steps [2].

2.4 The Riemannian geometry of the SPD manifold

In this section, which is adapted from Weber and Sra [9] and Persch [15], we briefly provide some background on the Riemannian geometry of the manifold of $n \times n$ symmetric positive-definite (SPD) matrices, which we will use in our numerical demonstrations. We denote this manifold by $\mathcal{P}_n$. Formally, we have

$$\mathcal{P}_n = \{p \in \mathbb{R}^{n \times n} \mid p = p^\mathsf{T},\ A^\mathsf{T} p A > 0 \quad \forall A \in \mathbb{R}^n\}.$$

Although we will restrict our attention to SPD matrices, we note that these results also generalize to Hermitian positive-definite (HPD) matrices.

The tangent space at a point $p \in \mathcal{P}_n$ is identified with the (Euclidean) space of symmetric matrices of size $n \times n$, denoted $\operatorname{Sym}(n)$:

$$\mathcal{T}_p\mathcal{P}_n = \{p^{1/2} A p^{1/2} \mid A \in \operatorname{Sym}(n)\}.$$

Note that when $p = I$ (the identity), the tangent space is equal to $\operatorname{Sym}(n)$. Since the tangent space vectors are also matrices, of the same size as the manifold points, multiplying manifold points and tangent vectors (through regular matrix multiplication) is a well-defined operation.

We define a partial ordering on $\mathcal{P}_n$. Given two symmetric positive-definite matrices p and q, we say that $p \preceq q$ (or $p \prec q$) if $q-p$ is positive semi-definite (respectively positive-definite). A *positive-definite interval* from p to q is then defined as $\{r \in \mathcal{P}_n \mid p \preceq r \preceq q\}$.

As our Riemannian metric on $\mathcal{P}_n$ we use the affine invariant metric, for which the inner product on $\mathcal{P}_n$ is given by

$$\langle X, Y \rangle_p = \operatorname{tr}\left(p^{-1} X p^{-1} Y\right).$$

Denoting by $||\cdot||_F$ the Frobenius norm, the corresponding Riemannian distance is then

$$\operatorname{dist}(p, q) = ||\log\left(p^{-1/2} q p^{-1/2}\right)||_F.$$

A geodesic $\gamma_{p\to q}$ on $\mathcal{P}_n$ between p and q is given by

$$\gamma(t) = p^{1/2}\left(p^{-1/2} q p^{-1/2}\right)^t p^{1/2}.$$

Further, the exponential and logarithmic maps are given by

$$\operatorname{Exp}_p X = p^{1/2} \exp\left(p^{-1/2} X p^{-1/2}\right) p^{1/2},$$
$$\operatorname{Log}_p q = p^{1/2} \log\left(p^{-1/2} q p^{-1/2}\right) p^{1/2},$$

where exp and log denote the standard matrix exponential and logarithm.

Finally, the Riemannian gradient of a function $f\colon \mathcal{P}_n \to \mathbb{R}$ is defined as

$$\operatorname{grad} f(p) = p\left(\nabla f(p) + \nabla f(p)^{\mathsf{T}}\right) p,$$

where ∇ denotes the Euclidean gradient.

We will also need the following result, provided by Bhatia [20]:

Lemma 2.1. *Let $p, q \in \mathcal{P}_n$. Let $\lambda^{\downarrow}(p), \lambda^{\uparrow}(p)$ denote the vectors of the eigenvalues of p in decreasing (respectively increasing) order, and let $\cdot$ denote the elementwise product. Then*

$$\lambda^{\downarrow}(p) \cdot \lambda^{\uparrow}(q) \leq \lambda(AB) \leq \lambda^{\downarrow}(p) \cdot \lambda^{\downarrow}(q.$$

Further, if p, q are symmetric, we have

$$\left\langle \lambda^{\downarrow}(p), \lambda^{\uparrow}(q) \right\rangle \leq \operatorname{tr}(pq) \leq \left\langle \lambda^{\downarrow}(p), \lambda^{\downarrow}(q) \right\rangle,$$

with equality if the matrix product pq is symmetric.

2.5 The Riemannian geometry of SO(n)

This section, which we have adapted from Persch [15], provides a brief introduction to the Riemannian geometry of the manifold known as the *special orthogonal group*, $\mathcal{M} = \mathrm{SO}(n)$, consisting of $n \times n$ rotation matrices – that is, orthogonal matrices with determinant 1:

$$\mathrm{SO}(n) = \{p \in \mathbb{R}^{n\times n} \mid p^{\mathsf{T}} p = p p^{\mathsf{T}} = I, \det p = 1\}.$$

The tangent space at p is given by the space of $n \times n$ *skew-symmetric* matrices:

$$\mathcal{T}_p \mathrm{SO}(n) = \{pX \mid p \in \mathrm{SO}(n), X \in \mathbb{R}^{n\times n}, X = -X^{\mathsf{T}}\}.$$

Just like on $\mathcal{P}_n$, manifold points and tangent vectors are both $n \times n$ matrices, and thus multiplying them is a well-defined operation.

The Riemannian metric we use on this manifold is given by

$$\langle X, Y \rangle_p = \operatorname{tr}\left(p^{\mathsf{T}} q\right).$$

Further, the exponential and logarithmic maps are given by

$$\operatorname{Exp}_p X = p \exp(X)$$
$$\operatorname{Log}_p q = p \log(q)$$

where exp and log denote the standard matrix exponential and logarithm.

Chapter 3

The Frank–Wolfe algorithm on $\mathbb{R}^n$

3.1 Euclidean Frank–Wolfe

The Frank–Wolfe algorithm, first developed by Marguerite Frank and Philip Wolfe in 1956 [10], is an optimization algorithm for solving constrained problems with convex constraints. Although we are mainly interested in working with the Riemannian version of the algorithm – which we will introduce later – it is convenient to introduce it first in the more familiar Euclidean setting. This lets us explain the basic workings of the algorithm and the motivation behind it, before moving on to the specifics of adapting it to and using it on manifolds later.

The algorithm takes as input an objective function $f\colon \mathbb{R}^n \to \mathbb{R}$, a convex set of constraints $\mathcal{C}$ and a starting point $x_0 \in \mathcal{C}$. We also require that $\mathcal{C}$ is compact, to ensure that the problem has a solution. Additionally, one needs to be able to compute the gradient ∇f of the objective function. The algorithm is displayed in Algorithm 1.

Algorithm 1 Euclidean Frank–Wolfe on R^n

Input: Objective function $f\colon \mathbb{R}^n \to \mathbb{R}$, constraint set $\mathcal{C} \subseteq \mathbb{R}^n$, initial point $x_0 \in \mathcal{C}$
for $k = 0, 1, \cdots$ **do**
  $z_k \leftarrow \arg\min_{z \in \mathcal{C}} \langle \nabla f(x_k), z \rangle$
  $\alpha_k \leftarrow \frac{2}{k+2}$
  $x_{k+1} \leftarrow (1 - \alpha_k) x_k + \alpha_k z_k$
end for

The idea behind the algorithm is that of reducing the problem to solving another, simpler *subproblem* on the same constraint set. This is done by taking a linear approximation of the objective function, minimizing this in order to find a search direction, and then taking a short step in this direction. Consider the linearization of the function $f\colon \mathbb{R}^n \to \mathbb{R}$ around a given point $x_k \in \mathbb{R}^n$, given by

$$f(z) \approx f(x_k) + \langle \nabla f(x), z - x_k \rangle. \tag{3.1}$$

By minimizing this approximation of the original cost function, the problem is reduced to a linear, convex problem. Technically, the linear subproblem we now want to solve is

$$\arg\min_{z \in \mathcal{C}} f(x_k) + \langle \nabla f(x_k), z - x_k \rangle.$$

However, the inner product is linear, so

$$f(x_k) + \langle \nabla f(x_k), z - x_k \rangle = f(x_k) + \langle \nabla f(x_k), z \rangle - \langle \nabla f(x_k), x_k \rangle.$$

Since both the terms $f(x_k)$ and $-\langle \nabla f(x_k), x_k \rangle$ are constant for a given x_k, they do not affect the minimum, and can thus be dropped from the computation. The remaining term $\langle \nabla f(x_k), z \rangle$ now matches that in line 3 of Algorithm 1.

On its own, this procedure gives us a sequence of iterates converging to the solution of the linear approximation of the original problem – it doesn't actually solve the original problem (unless the solutions happen to coincide). The key to the Frank–Wolfe algorithm lies in the next two steps of the algorithm: taking ever-decreasing steps between the current iterate and the solution to the subproblem. Instead of jumping straight to the solution of the subproblem at each iteration of the algorithm loop, we solve the subproblem in order to determine a search direction for our next step.

As long as the optimality condition (2.4) isn't satisfied, i.e.

$$\langle \nabla f(x_k), z - x_k \rangle < 0,$$

there must be a feasible descent direction from the current iterate x_k. Thus, unless we have already found the solution, the minimal value of the subproblem must be negative. This means that by minimizing the subproblem, we are finding the search direction $d = z - x_k$ that is the most negatively correlated with the gradient – i.e. the direction from x_k where the objective function decreases most quickly. Finally, in line 5 of the algorithm we take a step of size α_k in this direction, following a straight line from the current iterate.

Note that it is not actually necessary to use the stepsize $\alpha_k = 2/(k+2)$ given in Algorithm 1. Other ways to determine the stepsize can be used, such as a line search, or even an exact minimization, although this requires a slightly more expensive computation. The given stepsize has been found to give good results with minimal computational overhead, as well as providing a guarantee for the convergence of the algorithm [9].

A critical aspect of this algorithm is the fact that the constraint set is convex. Since we are only taking steps along straight lines between points in the set, the iterates will always stay within the constraints. This avoids the need to project back to the feasible set with each iteration, replacing it instead with a linear minimization. As linear minimizations are generally simpler than projections, this can have a significant effect on the cost of computation [21].

For many problems, the subproblem can even be solved in closed form. This means that the optimization problem in line 3 is reduced to consulting a *linear minimization oracle*, which returns the minimizer of $\langle \nabla f(x_k), z - x_k \rangle$ over $\mathcal{C}$. It is assumed that the constraints only need to be accessed indirectly, through the oracle. The Frank–Wolfe algorithm is particularly well suited to problems where this is the case: When the oracle can be computed efficiently, each iteration of the algorithm can be computed in just three simple, closed-form steps [22]. Note that since the subproblem is linear, it is never necessary to solve it using iterative methods. Even if there isn't a closed-form solution, linear problems can always be solved very efficiently using e.g. the simplex method or interior-point methods [18].

3.2 Convergence of Euclidean Frank–Wolfe

The great disadvantage of the Frank–Wolfe method is that it only converges at a sublinear rate: Frank and Wolfe established that there exists a constant $\beta > 0$ and an integer K such that for all $k > K$,

$$f(x_k) - f(x^*) \leq \frac{\beta}{k}.$$

Thus, the worst-case convergence rate is $\mathcal{O}(1/k)$ in the number of iterations k [10, 23].

Given some additional assumptions on the problem – which usually hold in practical problems – there is also a tight lower bound on how fast the algorithm can converge, as shown by Canon and Cullum [24]. Assuming that the optimum x^* of the problem lies on the boundary of $\mathcal{C}$ and x_k stays on the interior of $\mathcal{C}$ for infinitely many iterations k, if x_k is an ϵ-approximate solution, it holds that for any $\epsilon > 0$ and stepsize $\alpha_k > 0$,

$$f(x_k) - f(x^*) \geq \frac{\alpha_k}{k^{1+\epsilon}}$$

for an infinite number of k.

The worst-case convergence rate $\mathcal{O}(1/k)$ cannot, in general, be improved upon. This is due to something described in the literature as the *zig-zagging phenomenon*[22, 24]. When the solution is located on the boundary of $\mathcal{C}$, the search directions computed at each iteration can start to alternate back and forth between different vertices of $\mathcal{C}$, causing the slow convergence rate [25].

However, for certain types of problems, faster rates can be attained. In particular, when the solution lies in the interior of the constraint set, a linear convergence rate is attained [26] – i.e. for some positive constant $C < 1$ we have

$$f(x_{k+1}) - f(x^*) \leq C\big(f(x_k) - f(x^*)\big).$$

Thus, the main practical consideration when using Frank–Wolfe is whether computing a linear minimization is significantly cheaper than computing a projection onto the feasible set. That is, for a given constraint set $\mathcal{C}$, is it cheaper to compute

$$\arg\min_{x \in \mathcal{C}} \langle x, y \rangle \quad \text{or} \quad \arg\min_{x \in \mathcal{C}} ||x - y|| ?$$

If computing the subproblem is more costly than projecting onto the feasible set, the Frank–Wolfe algorithm's slow rate of convergence puts it at a disadvantage against faster, projection-based algorithms. Thus, for less complicated constraints it is usually preferable to use an algorithm with a faster rate of convergence, such as projected gradient descent [19]. Especially in the unconstrained case, where we don't have to do any projections at all, using a simpler algorithm is better.

The Frank–Wolfe algorithm is most useful when projecting onto the feasible set incurs a significant computational cost. However, there are many problems where projecting onto the constraint set can be computationally expensive: Combettes and Pokutta [21] provide several examples of such sets, which commonly appear as constraints in optimization problems.

Chapter 4

The Frank–Wolfe algorithm on $\mathcal{M}$

4.1 Riemannian Frank–Wolfe

We are now ready to introduce the Riemannian Frank–Wolfe algorithm, which generalizes the classical Frank–Wolfe algorithm to work on Riemannian manifolds. The algorithm is displayed in Algorithm 2. This version of the algorithm was first developed by Weber and Sra in 2017 [12]. The idea behind it is that the general, high-level concept of the Euclidean algorithm is equally valid in the Riemannian setting, and thus the algorithm can be adapted to working on manifolds simply by replacing the vector space operations with their more general manifold counterparts.

The algorithm takes the same input as the Euclidean version: a cost function f, a constraint set $\mathcal{C}$ and a starting point $p_0 \in \mathcal{C}$. The main difference is that the cost function is now defined on the manifold, $f\colon \mathcal{M} \to \mathbb{R}$, and the constraint set is a subset of the manifold, $\mathcal{C} \subseteq \mathcal{M}$. We require that the manifold $\mathcal{M}$ is geodesically complete, to ensure that the exponential map is defined on the entire manifold. Further, we assume that both the constraint set $\mathcal{C}$ and the objective function f are geodesically convex. The first assumption ensures that the iterates stay within the constraints, and the second ensures that any local solution is also the global solution – and thus that the algorithm converges to the solution.

Algorithm 2 Riemannian Frank–Wolfe on $\mathcal{M}$

1: **Input:** Objective function $f\colon \mathcal{M} \to \mathbb{R}$, constraint set $\mathcal{C} \subseteq \mathcal{M}$, initial point $p_0 \in \mathcal{C}$
2: **for** $k = 0, 1, \cdots$ **do**
3: $\quad q_k \leftarrow \arg\min_{q \in \mathcal{C}} \langle \operatorname{grad} f(p_k), \operatorname{Log}_{p_k} q \rangle_{p_k}$
4: $\quad \alpha_k \leftarrow \frac{2}{k+2}$
5: $\quad p_{k+1} \leftarrow \gamma_{p_k \to q_k}(\alpha_k)$
6: **end for**

The main loop of Riemannian algorithm is very similar to the Euclidean one, and we refer to section 3.1 for details about its general operation. In summary, at each iteration we solve the subproblem to find a search direction, and as long as the optimality condition (2.10) isn't satisfied, i.e.

$$\langle \nabla f(p_k), \operatorname{Log}_{p_k} q \rangle_{p_k} < 0,$$

there is a feasible descent direction for the problem. Thus, due to the convexity of $\mathcal{C}$, we can move in this direction to find a better objective value.

Since working with arbitrary manifolds requires a slightly more general approach than in the Euclidean setting, we have to make a few adjustments to the algorithm. The first change is in the *subproblem* in line 3,

$$\arg\min_{q\in\mathcal{C}} \langle \operatorname{grad} f(p_k), \operatorname{Log}_{p_k} q\rangle_{p_k}. \tag{4.1}$$

Since we are now working on a manifold, the inner product $\langle\cdot,\cdot\rangle_{p_k}$ is defined on the tangent space $\mathcal{T}_{p_k}\mathcal{M}$ – thus we need to ensure that its arguments are in the tangent space. In general, the Euclidean gradient $\nabla f(p_k)$ in the first argument is not in the tangent space at p_k, so we need to replace it by the Riemannian gradient.

In Euclidean space we do not have to make a distinction between points and tangent vectors in the inner product. This allowed us to take the difference between the two points $(z-x_k)$, and use the linearity of the inner product to get rid of the term containing x_k. This is not possible in general Riemannian spaces, which means that we need to keep the entire subtraction term $\langle \nabla f(x_k), z - x_k\rangle$ from equation (3.1) in mind when adapting this step to work on manifolds. Thus, in order to get a proper tangent vector for the second argument, we replace the subtraction by the corresponding manifold operation: the logarithmic map with base point p_k, applied to q.

The second important change is in line 5:

$$p_{k+1} \leftarrow \gamma_{p_k\to q_k}(\alpha_k).$$

We have to ensure that we stay on the manifold, so instead of taking a step along a straight line from p_k to q – as we could in $\mathbb{R}^n$ – we have to take the step along a geodesic connecting the two points. Thus, we must now compute the geodesic $\gamma_{p_k\to q_k}$, with $\gamma_{p_k\to q_k}(0) = p_k$ and $\gamma_{p_k\to q_k}(1) = q$, and take a step of size α_k along this geodesic.

Since both the logarithm and the geodesic are generalizations of their corresponding Euclidean operations, we see that choosing $\mathcal{M} = \mathbb{R}^n$ returns the original Euclidean algorithm. Note that while the changes are straightforward in theory, this is not necessarily the case in practice. Whereas the simple additions, subtractions and scalar multiplications used in the Euclidean algorithm have relatively little computational overhead, computing the logarithmic map or geodesic may require more demanding computations, depending on the manifold.

4.2 The Riemannian oracle

A significant difference from the Euclidean setting is that the subproblem is no longer linear, due to the use of the nonlinear logarithmic map in the inner product. In fact, we can't even guarantee that the Riemannian subproblem is convex [9]. This means that the subproblem can be much more difficult to solve than in the Euclidean case. However, many problems do still have a closed-form solution for the subproblem, and the idea of a solution oracle is still valid in the Riemannian setting. These oracles generally involve more complicated computations than in the Euclidean case, and thus the question of what is more difficult – calling the oracle or projecting onto the feasible set – is even more important in the Riemannian setting.

Even in cases where we do not have a closed-form solution to the subproblem, the Frank–Wolfe algorithm can still be used, but the subproblem now has to be solved

iteratively, using a different optimization algorithm – which we will refer to as the *subsolver*.

However, we have now essentially replaced one constrained problem with another, trading the original problem (solved by Frank–Wolfe) for the subproblem (solved by the subsolver) – which even has to be solved every iteration. Thus an important question is whether using this method to solve the subproblem is any more efficient than just using it to solve the original problem directly. While it's difficult to say anything about this in general, as it depends on the form of the problem, one can hope that the linearization simplifies the problem enough that it can be solved more efficiently by the subsolver.

One problem is the fact that we are still dealing with the same constraint set, which means that if it was difficult to use projection-based methods on the original problem, we will have the same issue with the subproblem. Thus, we will ideally want to use another projection-free method to solve the subproblem, for instance the augmented Lagrangian method or the exact penalty method, as discussed by Liu and Boumal [27].

4.2.1 Gradient-based subsolver methods

Although we could use any constrained Riemannian optimization method to solve the subproblem, many common optimization methods are – like Frank–Wolfe – gradient-based. Thus, in order to apply these to the subproblem we need a closed form expression for the gradient of the subproblem cost,

$$\operatorname{grad}_q g(q) = \operatorname{grad}_q \langle \operatorname{grad}_{p_k} f(p_k), \operatorname{Log}_{p_k}(q) \rangle_{p_k},$$

where we note that $\operatorname{grad}_{p_k} f(p_k)$ denotes the Riemannian gradient of the cost f with respect to its argument, (pre)evaluated at the point p_k. To derive this gradient we follow the approach used by Pfaue [28], who solves a similar problem when deriving KKT conditions for smooth manifolds.

We consider a composition of two functions,

$$\begin{aligned} h\colon \mathcal{M} \to \mathcal{T}_{p_k}\mathcal{M} \qquad & h(q) = \operatorname{Log}_{p_k}(q) \qquad && D_q h\colon \mathcal{T}_q\mathcal{M} \to \mathcal{T}_{h(q)}(\mathcal{T}_{p_k}\mathcal{M}) \cong \mathcal{T}_{p_k}\mathcal{M}, \\ j\colon \mathcal{T}_{p_k}\mathcal{M} \to \mathbb{R} \qquad & j(X) = \left\langle \operatorname{grad}_{p_k} f(p_k), X \right\rangle_{p_k} && D_X j\colon \mathcal{T}_X(\mathcal{T}_{p_k}\mathcal{M}) \cong \mathcal{T}_{p_k}\mathcal{M} \to \mathbb{R}, \end{aligned}$$

so that

$$g(q) = \big(j \circ h\big)(q) = j\big(h(q)\big) = \langle \operatorname{grad} f(p_k), \operatorname{Log}_{p_k}(q) \rangle_{p_k}.$$

We can now use the chain rule to derive an expression for the gradient of g. By the definition of the gradient, for any $Y \in \mathcal{T}q\mathcal{M}$,

$$\begin{aligned} \langle \operatorname{grad}_q g(q), Y \rangle_q &= \langle \operatorname{grad}_q (j \circ h)(q), Y \rangle_q \\ &= D_q(j \circ h)(q)[Y] \\ &= D_{h(q)} j(q)\big[D_q h(q)[Y]\big] \\ &= \langle \operatorname{grad}_q j(X), D_q h(q)[Y] \rangle_{\mathcal{T}_{p_k}\mathcal{M}}. \end{aligned}$$

We can now use the adjoint differential operator to isolate Y in the second argument of the inner product:

$$\begin{aligned} \langle \operatorname{grad}_q g(q), Y \rangle_q &= \langle \operatorname{grad}_q j(X), D_q h(q)[Y] \rangle_{\mathcal{T}_{p_k}\mathcal{M}}. \\ &= \left\langle (D_q h)^*(h(q))\big[\operatorname{grad}_q j(X)\big], Y \right\rangle_q. \end{aligned}$$

This, finally, means that

$$\operatorname{grad}_q g(q) = (D_q h)^*(q)\big[\operatorname{grad}_Y j(X)\big]$$

The first argument of the inner product $j(X)$ is the gradient of the main cost function, $\operatorname{grad} f(p)$, evaluated at the current iterate p_k. This term is independent of the variable X we want to optimize over – it is essentially a generic vector in the tangent space $\mathcal{T}_{p_k}\mathcal{M}$. Thus, the gradient of $j(X)$ is simply

$$\operatorname{grad}_X j(X) = \operatorname{grad}_X \left\langle \operatorname{grad}_{p_k} f(p_k), X \right\rangle_{p_k} = X,$$

and the gradient of g is

$$\begin{aligned}\operatorname{grad} g(q) &= (D_q h)^*(h(q))\big[\operatorname{grad}_Y j(X)\big] \\ &= (D_q h)^*(Z)[X] \\ &= (D_q \operatorname{Log}_{p_k})^*(Z)[X],\end{aligned}$$

where $Z = h(q) = \operatorname{Log}_{p_k}(q) \in \mathcal{T}_{p_k}\mathcal{M}$ is the tangent vector the adjoint differential is evaluated at.

We will not need to go any further than this for our implementation, as the adjoint of the differential of the logarithm is already implemented natively in *Manopt.jl*. However, for the sake of completeness we include a brief description of the closed-form expression for the adjoint. For a complete treatment of this, see Persch [15].

From the definition of the adjoint differential, we have

$$(D_q h)^*(q)[X] = \sum_{l=1}^{d} \langle X, \Xi_l(T)\rangle_{\mathcal{T}_q\mathcal{M}}\, \alpha_l \Xi_l(0),$$

where $\{\Xi_1(t), \cdots, \Xi_d(t)\}$ is an orthonormal basis of $\mathcal{T}_{h(q)}(\mathcal{T}_q\mathcal{M}) \cong \mathcal{T}_q\mathcal{M}$, parallel transported along the geodesic $\gamma_{q\to p_k}$ – i.e. $\Xi_l(t)$ is the l-th basic vector transported from $\gamma_{q\to p_k}(0) = q$ to $\gamma_{q\to p_k}(t)$.

The coefficients α_l correspond to the eigenvalues λ_l of the basis. For the function $h(q) = \log_{p_k}(q)$, the parameter $T = 1$ and the following coefficients α_l are provided by Persch [15] (Lemma 2.3):

$$\alpha_l = \begin{cases} \frac{\sqrt{-\lambda_l}}{\sinh(\sqrt{-\lambda_l})}, & \lambda_l < 0, \\ 1, & \lambda_l = 0, \\ \frac{\sqrt{\lambda_l}}{\sinh(\sqrt{\lambda_l})}, & \lambda_l > 0. \end{cases}$$

4.2.2 Derivation of the SPD Riemannian oracle

Weber and Sra [9] derive a Riemannian oracle for the manifold of SPD matrices $\mathcal{P}_n$. We provide a slightly adapted version of this derivation here, showing that the subproblem in this case admits a closed-form solution. For a brief overview of the Riemannian geometry of $\mathcal{P}_n$, we refer back to section 2.4.

We consider an objective function $f\colon \mathcal{P}_n \to \mathbb{R}$, and a constraint set consisting of a positive-definite interval:

$$\mathcal{C} = \{p \mid L \preceq p \preceq U\}.$$

Then the subproblem (4.1) has the form

$$\operatorname*{arg\,min}_{L \preceq q \preceq U} \left\langle \operatorname{grad} f(p_k), \operatorname{Log}_p q \right\rangle . \tag{4.2}$$

The inner product in equation (4.2) can be rewritten as

$$\begin{aligned}
\left\langle \operatorname{grad} f(p), \operatorname{Log}_p q \right\rangle &= \left\langle \operatorname{grad} f(p), p^{1/2} \log\left(p^{-1/2} q p^{-1/2}\right) p^{1/2} \right\rangle \\
&= \left\langle p^{-1/2} \operatorname{grad} f(p) p^{-1/2}, \log\left(p^{-1/2} q p^{-1/2}\right) \right\rangle \\
&= \operatorname{tr}\left(p^{-1} \left(p^{-1/2} \operatorname{grad} f(p) p^{-1/2}\right) p^{-1} \log\left(p^{-1/2} q p^{-1/2}\right)\right) \\
&= \operatorname{tr}\left(p^{-3/2} \operatorname{grad} f(p) p^{-3/2} \log\left(p^{-1/2} q p^{-1/2}\right)\right) \\
&= \operatorname{tr}\left(G \log(r)\right),
\end{aligned}$$

where $G = p^{-3/2} \operatorname{grad} f(p) p^{-3/2}$ and $r = p^{-1/2} q p^{-1/2}$. We now have a minimization problem over r, with the following transformed constraints:

$$\begin{aligned}
L' \preceq r \preceq U', \\
L' = p^{-1/2} L p^{-1/2}, \\
U' = p^{-1/2} U p^{-1/2}.
\end{aligned}$$

By diagonalizing G as $G = QDQ^{\mathsf{T}}$ and letting $s = Q^{\mathsf{T}} r Q$, we have

$$\begin{aligned}
\operatorname{tr}\left(G \log(r)\right) &= \operatorname{tr}\left(QDQ^{\mathsf{T}} \log(r)\right) = \operatorname{tr}\left(DQ^{\mathsf{T}} \log(r) Q\right) \\
&= \operatorname{tr}\left(D \log(s)\right).
\end{aligned}$$

The result is, again, a new minimization problem, this time over s. It follows that the constraints are transformed in the same way as r:

$$\begin{aligned}
L'' \preceq s \preceq U'', \\
L'' = Q^{\mathsf{T}} L' Q \left(= Q^{\mathsf{T}} p^{-1/2} L p^{-1/2} Q, \right) \\
U'' = Q^{\mathsf{T}} U' Q \left(= Q^{\mathsf{T}} p^{-1/2} U p^{-1/2} Q. \right)
\end{aligned}$$

By performing the Cholesky decomposition $U'' - L'' = P^{\mathsf{T}} P$ and letting $R = (P^{\mathsf{T}})^{-1}(s - L'')P^{-1}$ we can again rewrite the constraint as

$$\begin{aligned}
& 0 \preceq s - L'' \preceq U'' - L'' \\
\Longrightarrow\ & 0 \preceq (P^{\mathsf{T}})^{-1}(s - L'')P^{-1} \preceq (P^{\mathsf{T}})^{-1}(P^{\mathsf{T}} P)P^{-1} \\
\Longrightarrow\ & 0 \preceq R \preceq I.
\end{aligned}$$

From the definition of R, we now get

$$s = P^{\mathsf{T}} R P + L'',$$

and thus the subproblem (4.2) finally turns into

$$\operatorname*{arg\,min}_{0 \preceq R \preceq I} \operatorname{tr}\left(D \log(P^{\mathsf{T}} R P + L'')\right).$$

Since the trace is equal to the sum of eigenvalues, minimizing the trace is equivalent to minimizing the sum of eigenvalues. From Lemma 2.1 we have

$$\operatorname{tr}\left(D\log(P^{\mathsf{T}}RP+L'')\right)=\left\langle\lambda^{\downarrow}(D),\lambda^{\downarrow}\left(\log(P^{\mathsf{T}}RP+L'')\right)\right\rangle,$$

where the inner product here denotes the regular Euclidean dot product. Since D is diagonal, its elements are equal to its eigenvalues:

$$\lambda^{\downarrow}(D)_j=D_{ii},$$

where the j's are some permutation of $1,\cdots,n$. Thus, in order to minimize the inner product of eigenvalues, we need to minimize the eigenvalues of $\log(P^{\mathsf{T}}RP+L'')$ corresponding to positive elements of D, and maximize those corresponding to negative elements of D.

Now, without loss of generality, we assume that R is diagonal. Since $0\preceq R\preceq I$, we must then have $0\leq R_{ii}\leq 1$. Due to the fact that both the matrix logarithm and the map $R\mapsto PP^{\mathsf{T}}RP+L''$ are operator monotone, the minimum is hence achieved by setting

$$R_{ii}=\begin{cases}0 & \text{if } D_{ii}\geq 0\\ 1 & \text{if } D_{ii}<0\end{cases}.$$

Thus, the optimal point for the subproblem is given by

$$\begin{aligned}q&=p^{1/2}rp^{1/2}\\&=p^{1/2}Q\big(P^{\mathsf{T}}RP+L''\big)Q^{\mathsf{T}}p^{1/2}\\&=p^{1/2}Q\big(P^{\mathsf{T}}[-\operatorname{sgn}(D)]_+P+L''\big)Q^{\mathsf{T}}p^{1/2}.\end{aligned}$$

4.2.3 Derivation of the SO(n) Riemannian oracle

To illustrate another setting where a Riemannian oracle is available, we now derive a Riemannian oracle for *unconstrained* problems on $\mathrm{SO}(n)$. The contents of this section are again adapted from Weber and Sra [9]. For a brief overview of the Riemannian geometry of $\mathrm{SO}(n)$, we refer back to section 2.5.

We consider the objective function $f\colon \mathrm{SO}(n)\to\mathbb{R}$, so that the subproblem (4.1) has the form

$$\begin{aligned}\operatorname*{arg\,min}_{q\in\mathrm{SO}(n)}\langle\operatorname{grad}f(p),\operatorname{Log}_p q\rangle&=\operatorname*{arg\,min}_{p\in\mathrm{SO}(n)}\langle\operatorname{grad}f(p),p\log(p^{\mathsf{T}}q)\rangle\\&=\operatorname*{arg\,min}_{q\in\mathrm{SO}(n)}\operatorname{tr}\left(\operatorname{grad}f(p)^{\mathsf{T}}p\log(p^{\mathsf{T}}q)\right)\\&=\operatorname*{arg\,min}_{q\in\mathrm{SO}(n)}\operatorname{tr}\left(G\log(p^{\mathsf{T}}q)\right),\end{aligned}\tag{4.3}$$

where we denote $G=\operatorname{grad}f(p)^{\mathsf{T}}p$. Taking the singular value decomposition $G=UDV^*$, such that U and V are unitary, we have

$$\begin{aligned}\operatorname{tr}\left(G\log(p^{\mathsf{T}}q)\right)&=\operatorname{tr}\left(UDV^*\log(p^{\mathsf{T}}q)\right)\\&=\operatorname{tr}\left(DV^*\log(p^{\mathsf{T}}q)U\right)\\&=\operatorname{tr}\left(D\log(V^*p^{\mathsf{T}}qU)\right)\\&=\operatorname{tr}\left(D\log(W)\right),\end{aligned}$$

where $W = V^* p^\mathsf{T} q U$. Note that the inner product in (4.3) is negative as long as there is a feasible descent direction, which means that

$$\begin{aligned}\operatorname{tr}(D\log(W)) &= \langle \operatorname{grad} f(p), \operatorname{Log}_p q\rangle \\ &= -|\langle \operatorname{grad} f(p), \operatorname{Log}_p q\rangle| \\ &= -|\operatorname{tr}(D\log(W))|\end{aligned}$$

Hence, noting that D is diagonal, it follows from the Cauchy–Schwarz inequality and the definition of the inner product that

$$\begin{aligned}\operatorname{tr}(D\log(W)) &= -|\operatorname{tr}(D\log(W))| \\ &= -|\langle D^\mathsf{T}, \log(W)\rangle_p| \\ &\geq -\langle D, D\rangle_p \langle \log(W), \log(W)\rangle_p \\ &= -\operatorname{tr}\left(D^\mathsf{T} D\right) \operatorname{tr}\left(\log(W)^\mathsf{T} \log(W)\right) \\ &= -\operatorname{tr}\left(D^2\right) \left(\sum_i \sigma_i(\log(W))\right)^{1/2}.\end{aligned}$$

where $\sigma_i(\log(W))$ are the singular values of $\log(W)$. Now, D depends only on the gradient, which is constant for given p. Thus, in order to minimize $\operatorname{tr}(D\log(W))$, we need to maximize the sum of singular values of W. Further, W is an orthogonal matrix, meaning it has all singular values $\sigma_i(W) = 1$. Thus, the right hand side can be minimized by setting $W = I$, and we get

$$I = W = V^* p^\mathsf{T} q U.$$

Thus, the linear oracle is given by

$$q = pVU^*.$$

Chapter 5

Convergence of Riemannian Frank–Wolfe

5.1 Global convergence

In this section, we will prove that the Riemannian version of the Frank–Wolfe algorithm converges at the same rate as the Euclidean version. In general, it is not possible to guarantee anything better than sublinear convergence, although with a certain assumption on the optimum – namely that it is located strictly in the interior of the constraint set – the algorithm achieves linear convergence. The proofs in this chapter are mostly adapted from the work by Weber and Sra [9], although we expand on some of the techniques they have used, going into greater detail.

In order to prove the convergence of the algorithm, we have to make a few assumptions about the objective function f. First, we require that it is sufficiently smooth – that is, it has a locally Lipschitz continuous gradient on the constraint set $\mathcal{C}$, as given by (2.6):

$$||\nabla f(x) - P_{q \leftarrow p} \nabla f(q)|| \leq L \cdot d(p, q), \quad \forall p, q \in \mathcal{M}.$$

Further, we assume that f is geodesically convex on $\mathcal{C}$, satisfying (2.8):

$$f(q) - f(p) \geq \langle X, Log_p(q) \rangle_p \quad \forall p, q \in \mathcal{C},\ \forall X \in \mathcal{T}_p\mathcal{M}.$$

With these assumptions we will prove the following theorem, showing the convergence rate of the Riemannian Frank–Wolfe algorithm:

Theorem 5.1. *Let $\mathcal{M}$ be a Riemannian manifold, $\mathcal{C} \subseteq \mathcal{M}$ a geodesically convex subset, and let $f \colon \mathcal{M} \to \mathbb{R}$ be geodesically convex and have a locally Lipschitz continuous gradient on $\mathcal{C}$. Then, for some constant C_f, the sequence of iterates produced by Algorithm 2 satisfies*

$$f(p_k) - f(p^*) \leq \frac{2C_f}{k+2}.$$

The curvature constant C_f

We begin by introducing the *curvature constant*, C_f, introduced in the Riemannian setting by Weber and Sra [9]. This Riemannian version is adapted from a similar constant that has been used in convergence proofs for the Euclidean Frank–Wolfe algorithm [22, 29].

Definition 5.2. Let $\mathcal{M}$ be a Riemannian manifold, $\mathcal{C} \subseteq \mathcal{M}$, and let $f\colon \mathcal{M} \to \mathbb{R}$ have locally Lipschitz continuous gradients. Let $p, q, r \in \mathcal{C}$ and $\eta \in [0,1]$, where $r = \gamma_{p\to q}(\eta)$ – i.e. r lies on a geodesic from p to q. We define the *curvature constant* C_f of f on $\mathcal{C}$ as

$$C_f := \sup_{p,q,r\in\mathcal{C}} \frac{2}{\eta^2}\left[f(r) - f(p) - \langle \operatorname{grad} f(p), Log_p(r)\rangle_p\right].$$

To motivate the definition of this constant, consider the linear approximation of the objective function f around p, evaluated at the point $r = \gamma_{p\to q}(\eta)$:

$$f(r) \approx f(p) + \left\langle \operatorname{grad} f(p), \operatorname{Log}_p(r)\right\rangle_p.$$

If the curvature constant is bounded, then the deviation of $f(r)$ from the linearization of f at p,

$$f(r) - f(p) - \langle \operatorname{grad} f(p), Log_p(r)\rangle_p$$

is bounded. Thus, in a sense, C_f gives us a bound on the "nonlinearity" of f along any geodesic $\gamma_{p\to q}$ on $\mathcal{M}$. For instance, we see that for a linear function f, $C_f = 0$ [29].

The weighting factor $2/\eta^2$ is motivated by the following, which relates the curvature constant to the Lipschitz bound:

Consider the *diameter* of the constraint set, given by $\operatorname{diam}(\mathcal{C}) = \sup_{p,q\in\mathcal{C}} d(p,q)$. If $f\colon \mathcal{M} \to \mathbb{R}$ is L-smooth on $\mathcal{C}$, then the curvature constant C_f satisfies the bound

$$C_f \leq L \operatorname{diam}(\mathcal{C})^2.$$

To show this, let $p, q, r \in \mathcal{C}$, $\eta \in (0,1)$ and again $r = \gamma_{p\to q}(\eta)$. This means that

$$d(p,r) = \eta \cdot d(p,q) \implies \frac{1}{\eta^2} d(p,r)^2 = d(p,q)^2.$$

Using the Lipschitz smoothness of f from (2.7) – which is equivalent to our assumption of (2.6) – we have

$$f(r) - f(p) - \left\langle \operatorname{grad} f(p), \operatorname{Log}_p(r)\right\rangle_p \leq \frac{L}{2} d(p,r)^2.$$

This, in turn, means that

$$\begin{aligned} C_f &= \sup_{p,q,r\in\mathcal{C}} \frac{2}{\eta^2}\left(f(r) - f(p) - \langle \operatorname{grad} f(p), Log_p(r)\rangle_p\right) \\ &\leq \sup_{p,q,r\in\mathcal{C}} L \cdot \frac{1}{\eta^2} d(p,r)^2 \\ &= L \cdot \sup_{p,q\in\mathcal{C}} d(p,q)^2 \\ &= L \operatorname{diam}(\mathcal{C})^2. \end{aligned}$$

Lipschitz bound

We can now redefine the Lipschitz inequality (2.7) using the curvature constant:

Lemma 5.1. *Let $\mathcal{M}$ be a Riemannian manifold, $\mathcal{C} \subseteq \mathcal{M}$, and let $f\colon \mathcal{M} \to \mathbb{R}$ have locally Lipschitz continuous gradients. For any two points $p, q \in \mathcal{C}$, let $\eta \in [0,1]$, and let $r = \gamma_{p\to q}(\eta)$. Finally, let the curvature constant C_f be defined as in Definition 5.2. Then*

$$f(r) \leq f(p) + \eta \left\langle \operatorname{grad} f(p), \operatorname{Log}_p(q)\right\rangle + \frac{1}{2} C_f \eta^2. \tag{5.1}$$

To show this, note that the definition of C_f implies that, for any choice of two points $p, q \in \mathcal{M}$,

$$C_f \geq \frac{2}{\eta^2}\left(f(r) - f(p) - \langle \operatorname{grad} f(p), Log_p(r)\rangle_p\right),$$

and thus we have

$$f(r) \leq f(p) + \left\langle \operatorname{grad} f(p), \operatorname{Log}_p(r)\right\rangle + \frac{1}{2}C_f\eta^2. \tag{5.2}$$

Further, the fact that $r = \gamma_{p\to q}(\eta)$ means that

$$\operatorname{Log}_p(r) = \eta \operatorname{Log}_p(q),$$

and we have

$$\begin{aligned}\left\langle \operatorname{grad} f(p), \operatorname{Log}_p(r)\right\rangle &= \left\langle \operatorname{grad} f(p), \eta \operatorname{Log}_p(q)\right\rangle \\ &= \eta\left\langle \operatorname{grad} f(p), \operatorname{Log}_p(q)\right\rangle.\end{aligned}$$

Finally, inserting this into inequality (5.2) yields

$$f(r) \leq f(p) + \eta\left\langle \operatorname{grad} f(p), \operatorname{Log}_p(q)\right\rangle + \frac{1}{2}C_f\eta^2.$$

Stepsize

The following result shows that our choice of stepsize produces a decreasing sequence. Let (A_k) be a nonnegative sequence satisfying

$$A_{k+1} \leq (1-\alpha_k)A_k + \frac{1}{2}\alpha_k^2 C_f, \tag{5.3}$$

where $0 < \alpha_k \leq 1$ is a parameter (the *stepsize*) depending on k. Then, if we let $\alpha_k = \frac{2}{k+2}$, the sequence satisfies

$$A_k \leq \frac{2C_f}{k+2}. \tag{5.4}$$

We show this by induction. For the base case, we have $k = 0$ and $\alpha_k = 1$, and thus

$$A_1 \leq 0 \cdot A_0 + \frac{1}{2}C_f \leq \frac{2}{3}C_f = \frac{2C_f}{1+2}.$$

Now assume the induction hypothesis holds, i.e. $A_k \leq \frac{2C_f}{k+2}$. Then

$$\begin{aligned}A_{k+1} &\leq (1-\alpha_k)A_k + \frac{1}{2}\alpha_k^2 C_f \\ &\leq \frac{k}{k+2}\frac{2C_f}{k+2} + \frac{1}{2}\frac{4}{(k+2)^2}C_f \\ &= 2C_f\left(\frac{k}{(k+2)^2} + \frac{1}{(k+2)^2}\right) \\ &= \frac{2C_f}{k+3}\left(\frac{(k+1)(k+3)}{(k+2)^2}\right) \\ &= \frac{2C_f}{k+3}\left(\frac{(k^2+3k)+(k+3)}{k^2+4k+4}\right) \\ &= \frac{2C_f}{k+3}\left(\frac{k^2+4k+3}{k^2+4k+4}\right) \\ &< \frac{2C_f}{(k+1)+2}.\end{aligned}$$

Global convergence rate

Finally, we can show the global convergence rate of the algorithm. Let the stepsize $\alpha_k = \frac{2}{k+2}$, and let p^* be a minimum of the cost function f. Then the sequence of iterates generated by Algorithm 2 satisfies

$$f(p_k) - f(p^*) = \mathcal{O}(1/k).$$

To show this, note first that q_k is—by definition—the minimizer of the subproblem in line 3 of the algorithm, so we have

$$\left\langle \operatorname{grad} f(p_k), \operatorname{Log}_{p_k}(q_k) \right\rangle \leq \left\langle \operatorname{grad} f(p_k), \operatorname{Log}_{p_k}(p^*) \right\rangle.$$

From the Lipschitz bound (5.1), with $p = p_k$, $q = q_k$ and $r = p_{k+1} = \gamma_{p_k \to q_k}(\alpha_k)$, as well as the step size α_k serving the role of η, we therefore have

$$\begin{aligned} f(p_{k+1}) - f(p^*) &\leq \left(f(p_k) + \alpha_k \left\langle \operatorname{grad} f(p_k), \operatorname{Log}_{p_k}(q_k) \right\rangle + \frac{1}{2} C_f \alpha_k^2 \right) - f(p^*) \\ &\leq \big(f(p_k) - f(p^*) \big) + \alpha_k \left\langle \operatorname{grad} f(p_k), \operatorname{Log}_{p_k}(p^*) \right\rangle + \frac{1}{2} C_f \alpha_k^2. \end{aligned} \tag{5.5}$$

Now, using the geodesic convexity defined in equation (2.8), we have

$$\langle \operatorname{grad} f(p_k), Log_p(p^*) \rangle_p \leq f(p^*) - f(p_k),$$

which means that

$$\begin{aligned} \big(f(p_k) - f(p^*) \big) + \alpha_k \left\langle \operatorname{grad} f(p_k), \operatorname{Log}_{p_k}(p^*) \right\rangle &\leq \big(f(p_k) - f(p^*) \big) + \alpha_k \big(f(p^*) - f(p_k) \big) \\ &= \big(f(p_k) - f(p^*) \big) - \alpha_k \big(f(p_k) - f(p^*) \big) \\ &= (1 - \alpha_k) \big(f(p_k) - f(p^*) \big). \end{aligned}$$

Inserted into equation (5.5) this yields

$$f(p_{k+1}) - f(p^*) \leq (1 - \alpha_k) \big(f(p_k) - f(p^*) \big) + \frac{1}{2} C_f \alpha_k^2.$$

We now see that setting $A_{k+1} = f(p_{k+1}) - f(p^*)$ in inequality (5.4) satisfies the requirement (5.3), and we get the desired $\mathcal{O}(1/k)$ convergence rate, proving Theorem 5.1:

$$f(p_k) - f(p^*) \leq \frac{2C_f}{k+2}.$$

5.2 Optimum in the interior of $\mathcal{C}$: Linear convergence

When the solution is located on the boundary of the feasible set, the Riemannian Frank–Wolfe algorithm suffers from the same *zig-zagging phenomenon* as the Euclidean algorithm [9]. However, just like in the Euclidean case, the Riemannian Frank–Wolfe algorithm converges linearly when the optimal solution lies in the interior of the constraint set. The only caveat is that we need to make the additional assumption that, for some μ, the cost function f is μ-strongly convex, satisfying (2.9):

$$f(q) - f(p) \geq \langle X, Log_p(q) \rangle_p + \frac{\mu}{2} d(p,q)^2 \quad \forall p, q \in \mathcal{S},\ X \in \mathcal{T}_p\mathcal{M}.$$

Theorem 5.3. *Let $\mathcal{M}$ be a Riemannian manifold, $\mathcal{C} \subseteq \mathcal{M}$ a geodesically convex subset, and let $f\colon \mathcal{M} \to \mathbb{R}$ be μ-strongly geodesically convex and have a locally Lipschitz continuous gradient on $\mathcal{C}$. Assume that there exists some ρ such that the minimizer p^* of f lies within a ball of radius ρ, $\mathcal{B}_\rho(p^*)$, strictly inside $\mathcal{C}$. Then it is possible to choose a stepsize α_k so that the sequence of iterates produced by Algorithm 2 satisfies*

$$f(p_{k+1}) - f(p^*) \leq \left(1 - \frac{3}{4}\frac{\rho^2\mu}{C_f}\right)\big(f(p_k) - f(p^*)\big).$$

We now prove theorem 5.3. We follow a strategy based on the one used by Garber and Hazan [30] for the Euclidean case and by Weber and Sra [9] for their Riemannian proof. At each iteration of algorithm 2, we compute the point q_k in the constraint set $\mathcal{C}$ satisfying

$$q_k = \arg\min_{q\in\mathcal{C}} \langle \operatorname{grad} f(p_k), \operatorname{Log}_{p_k}(q)\rangle$$

that is, the point lying in the direction of steepest descent in the tangent space $\mathcal{T}_{p_k}\mathcal{M}$. Let

$$W_k = \arg\max_{Z\in\mathcal{T}_{p_k}\mathcal{M},\ ||Z||\leq 1} \langle \operatorname{grad} f(p_k), Z\rangle,$$

be a vector in the direction of steepest ascent, limited to be of magnitude at most 1 – it thus points in the opposite direction of the minimizer of the subproblem, q_k. Note that if we have found the direction of steepest ascent, we can always increase the value of the inner product by choosing a longer vector, which means that in practice W_k will always have magnitude exactly equal to 1. Hence this vector is in fact just the gradient (which, by definition, is a vector of steepest ascent), normalized, i.e.

$$W_k = \frac{\operatorname{grad} f(p_k)}{||\operatorname{grad} f(p_k)||}.$$

If we choose p_0 in the interior of $\mathcal{C}$, every subsequent point produced by the algorithm will also be in the interior, due to the following: Since $\mathcal{C}$ is convex, any point on the geodesic between p_k and q_k is in $\mathcal{C}$. Thus, as long as the stepsize $\alpha_k < 1$, then $p_{k+1} = \gamma_{p_k\to q_k}(\alpha_k)$ can not be on the boundary—even if q_k is. This means that there exists some minimum radius ρ such that for every point p_k produced by the algorithm (including, in the limit as $k\to\infty$, p^*, as established by Theorem 5.1),

$$\mathcal{B}_\rho(p_k) \subset \mathcal{C}.$$

If we now take a point

$$p_\rho = \operatorname{Exp}_{p_k}(\rho W_k),$$

then $p_\rho \in \mathcal{B}_\rho(p_k) \subset \mathcal{C}$, since $||W_k|| \leq 1$. It then holds that

$$\begin{aligned}
\langle \operatorname{grad} f(p_k), -\operatorname{Log}_{p_k}(p_\rho)\rangle &= -\langle \operatorname{grad} f(p_k), \operatorname{Log}_{p_k}\big(\operatorname{Exp}_{p_k}(\rho W_k)\big)\rangle \\
&= -\langle \operatorname{grad} f(p_k), \rho W_k\rangle \\
&= -\left\langle \operatorname{grad} f(p_k), \rho\frac{\operatorname{grad} f(p_k)}{||\operatorname{grad} f(p_k)||}\right\rangle \\
&= -\rho\frac{\langle \operatorname{grad} f(p_k), \operatorname{grad} f(p_k)\rangle}{||\operatorname{grad} f(p_k)||} \\
&= -\rho\cdot||\operatorname{grad} f(p_k)||.
\end{aligned} \tag{5.6}$$

For ease of notation, we define the error in the solution at step k as $\Delta_k = f(p_k) - f(p^*)$. From the Lipschitz bound (5.1) we have

$$f(p_{k+1}) - f(p_k) \leq \alpha_k \langle \operatorname{grad} f(p_k), \operatorname{Log}_{p_k}(q_k) \rangle + \frac{C_f}{2}\alpha_k^2.$$

It follows that

$$\begin{aligned} \Delta_{k+1} &= f(p_{k+1}) - f(p^*) \\ &= f(p_k) - f(p^*) + \big(f(p_{k+1}) - f(p_k)\big) \\ &= \Delta_k + \big(f(p_{k+1}) - f(p_k)\big) \\ &\leq \Delta_k + \alpha_k \langle \operatorname{grad} f(p_k), \operatorname{Log}_{p_k}(q_k) \rangle + \frac{C_f}{2}\alpha_k^2. \end{aligned} \tag{5.7}$$

We now consider the inner product in (5.7). Note first that $-\operatorname{Log}_{p_k}(p_\rho) = \operatorname{Log}_p(r_\rho)$, for some point r_ρ in the opposite direction of p_ρ from p_k. Now, q_k is – by definition – the point that minimizes $\langle \operatorname{grad} f(p_k), \operatorname{Log}_{p_k}(q_k) \rangle$, so we have

$$\begin{aligned} \langle \operatorname{grad} f(p_k), \operatorname{Log}_{p_k}(q_k) \rangle &\leq \langle \operatorname{grad} f(p_k), \operatorname{Log}_{p_k}(r_\rho) \rangle \\ &= \langle \operatorname{grad} f(p_k), -\operatorname{Log}_{p_k}(p_\rho) \rangle. \\ &= -\rho \cdot || \operatorname{grad} f(p_k)||, \end{aligned}$$

where the last equality follows from (5.6). Inserted into equation (5.7), this yields

$$\Delta_{k+1} \leq \Delta_k - \alpha_k \cdot \rho \cdot || \operatorname{grad} f(p_k)|| + \frac{C_f}{2}\alpha_k^2. \tag{5.8}$$

Denoting the optimal value of the objective function f by f^*, μ-strong convexity implies the following Riemannian extension of the Polyak-Łojasiewicz (PL) inequality [9]:

$$\frac{1}{2}|| \operatorname{grad} f(p)||^2 \geq \mu \cdot \big(f(p) - f^*\big) \quad \forall p \in \mathcal{C},$$

which in our case implies

$$-|| \operatorname{grad} f(p_k)|| \leq -\sqrt{2\mu}\sqrt{f(p_k) - f(p^*)} = -\sqrt{2\mu}\sqrt{\Delta_k}.$$

Inserting this into equation (5.8), we get

$$\Delta_{k+1} \leq \Delta_k - \alpha_k \cdot \rho \cdot \sqrt{2\mu}\sqrt{\Delta_k} + \frac{C_f}{2}\alpha_k^2. \tag{5.9}$$

Finally, by setting the algorithm step size to

$$\alpha_k = \frac{\rho\sqrt{\mu\Delta_k}}{\sqrt{2}C_f},$$

we get the desired bound:

$$\Delta_{k+1} \leq \left(1 - \frac{3}{4}\frac{\rho^2\mu}{C_f}\right)\Delta_k.$$

Note that this choice of stepsize α_k requires knowing the current error Δ_k, and hence also the optimal value f^* – which is generally not available. However, this bound provides a theoretical guarantee that the algorithm converges, and we can still get the same rate of convergence by using a slightly worse stepsize [9].

We remark that our bound contains a slightly different constant than the one obtained by Weber and Sra. There seems to be a slight mistake in their proof, when inserting the stepsize into (5.9). The bound obtained by Weber and Sra is essentially the same as the one obtained in the Euclidean case by Garber and Hazan, but they use slightly different forms of the PL inequality in their proofs. The error seems to come from the fact that whereas Weber and Sra use a factor $\sqrt{2\mu}$ in (5.9), Garber and Hazan instead have a factor $\sqrt{\mu/2}$, resulting in a different constant factor in the final bound. We note that this is a very minor error, and that it does not affect the main result, namely that the algorithm converges linearly. Further, one could still obtain the same constant factor as them, by setting the stepsize to

$$\alpha_k = \left(2 + \sqrt{3}\right) \frac{\rho\sqrt{\mu\Delta_k}}{\sqrt{2}C_f}.$$

5.3 Convergence when solving the subproblem approximately

As we have noted in section 4.2, a Riemannian oracle for the subproblem is not always available. In that case, we can't guarantee that the subproblem will be solved exactly, as we will usually have to solve it by e.g. an iterative method. While this is likely to result in worse performance, we can show that the algorithm still converges. In fact, it converges at the same (asymptotic) rate as when a closed-form oracle is used. Weber and Sra [9] provide a proof of the convergence of the Riemannian Frank–Wolfe algorithm when the subproblem is solved only approximately, which we have adapted here.

We first introduce a notion of computational accuracy: For $\delta \geq 0$ and a stepsize $0 < \eta \leq 1$ we say that q is a *δ-approximate linear minimizer* of the subproblem 4.1 if

$$\left\langle \operatorname{grad} f(p_k), \operatorname{Log}_{p_k}(q) \right\rangle \leq \min_{q' \in \mathcal{C}} \left\langle \operatorname{grad} f(p_k), \operatorname{Log}_{p_k}(q') \right\rangle + \delta \frac{1}{2} C_f \eta.$$

The following theorem then provides the convergence guarantee of the Riemannian Frank–Wolfe algorithm when the subproblem is only solved approximately:

Theorem 5.4. *Let $\mathcal{M}$ be a Riemannian manifold, $\mathcal{C} \subseteq \mathcal{M}$ a geodesically convex subset, and let $f\colon \mathcal{M} \to \mathbb{R}$ be geodesically convex and have a locally Lipschitz continuous gradient on $\mathcal{C}$. Let the Riemannian oracle in the subproblem of line 3 of Algorithm 2 be solved to an accuracy of $\delta \geq 0$. Then the sequence of iterates produced by the algorithm satisfies*

$$f(p_k) - f(p^*) \leq (1 + \delta) \frac{2C_f}{k + 2}.$$

We begin by proving an intermediate result, namely that

$$f(p_{k+1}) \leq f(p_k) - \alpha_k \big(f(p_k) - f(p^*)\big)(1 + \delta) \frac{1}{2} C_f \alpha_k^2$$

Once again, we let $p, q, r \in \mathcal{C}$ and $\eta \in [0, 1]$, where $r = \gamma_{p \to q}(\eta)$, so r lies on a geodesic between p and q. Also, let q be a δ-approximate linear minimizer of the subproblem

and p^* be the optimal solution of the main problem 2.5. By definition, we then have

$$\begin{aligned}\left\langle \operatorname{grad} f(p), \operatorname{Log}_p(q)\right\rangle &\leq \min_{q'\in\mathcal{C}} \left\langle \operatorname{grad} f(p), \operatorname{Log}_p(q')\right\rangle + \delta\frac{1}{2}C_f\eta \\ &\leq \left\langle \operatorname{grad} f(p), \operatorname{Log}_p(p^*)\right\rangle + \delta\frac{1}{2}C_f\eta \\ &\leq f(p^*) - f(p) + \delta\frac{1}{2}C_f\eta \\ &= -\left(f(p) - f(p^*)\right) + \delta\frac{1}{2}C_f\eta,\end{aligned}$$

where the inequality in the second-to-last line follows from the convexity (2.8) of f.

Replacing η by α_k, and letting $p = p_k$, $q = q_k$ and $r = p_{k+1}$ – where q_k is now a δ-approximate solution of the subproblem – the Lipschitz bound (5.1) then gives

$$\begin{aligned}f(p_{k+1}) &\leq f(p_k) + \alpha_k \left\langle \operatorname{grad} f(p_k), \operatorname{Log}_{p_k}(q_k)\right\rangle + \frac{1}{2}C_f\alpha_k^2 \\ &\leq f(p_k) + \alpha_k\left(\big(f(p_k) - f(p^*)\big) + \delta\frac{1}{2}C_f\alpha_k\right) + \frac{1}{2}C_f\alpha_k^2 \\ &\leq f(p_k) - \alpha_k\big(f(p_k) - f(p^*)\big) + \delta\frac{1}{2}C_f\alpha_k^2 + \frac{1}{2}C_f\alpha_k^2 \\ &= f(p_k) - \alpha_k\big(f(p_k) - f(p^*)\big) + (1+\delta)\frac{1}{2}C_f\alpha_k^2,\end{aligned}$$

Again using the notation $\Delta_k = f(p_k) - f(p^*)$, we thus have

$$\begin{aligned}\Delta_{k+1} &= f(p_{k+1}) - f(p^*) \\ &\leq \left(f(p_k) - \alpha_k\big(f(p_k) - f(p^*)\big) + (1+\delta)\frac{1}{2}C_f\alpha_k^2\right) - f(p^*) \\ &= \big(f(p_k) - f(p^*)\big) - \alpha_k\big(f(p_k) - f(p^*)\big) + (1+\delta)\frac{1}{2}C_f\alpha_k^2 \\ &= (1-\alpha_k)\Delta_k + (1+\delta)\frac{1}{2}C_f\alpha_k\end{aligned}$$

Convergence of the sequence Δ_k

In a similar fashion as in the proof of global convergence in section 5.1, we now have a sequence (Δ_k), with

$$\Delta_{k+1} \leq (1-\alpha_k)\Delta_k + (1+\delta)\frac{1}{2}\alpha_k^2 C_f.$$

We will now show that by letting $\alpha_k = \frac{2}{k+2}$, this sequence satisfies

$$\Delta_k \leq (1+\delta)\frac{2C_f}{k+2}.$$

The proof is again by induction: For the base case, we have $k = 0$ and $\alpha_k = 1$, and thus

$$\Delta_1 \leq 0\cdot\Delta_0 + (1+\delta)\frac{1}{2}C_f \leq \frac{2}{3}(1+\delta)C_f = (1+\delta)\frac{2C_f}{1+2}.$$

Now assume the induction hypothesis holds, i.e. $\Delta_k \leq (1+\delta)\frac{2C_f}{k+2}$. Then

$$\begin{aligned}
\Delta_{k+1} &\leq (1-\alpha_k)\Delta_k + (1+\delta)\frac{1}{2}\alpha_k^2 C_f \\
&\leq \frac{k}{k+2}(1+\delta)\frac{2C_f}{k+2} + (1+\delta)\frac{1}{2}\frac{4}{(k+2)^2}C_f \\
&= (1+\delta)2C_f\left(\frac{k}{(k+2)^2} + \frac{1}{(k+2)^2}\right) \\
&= (1+\delta)\frac{2C_f}{k+3}\left(\frac{(k+1)(k+3)}{(k+2)^2}\right) \\
&= (1+\delta)\frac{2C_f}{k+3}\left(\frac{(k^2+3k)+(k+3)}{k^2+4k+4}\right) \\
&= (1+\delta)\frac{2C_f}{k+3}\left(\frac{k^2+4k+3}{k^2+4k+4}\right) \\
&< (1+\delta)\frac{2C_f}{(k+1)+2}.
\end{aligned}$$

Thus the algorithm converges at a rate of $\mathcal{O}(1/k)$ even when an approximate subproblem solver is used. Note especially that if $\delta = 0$, we get the same result as in section 5.1.

Chapter 6

Numerical results

In this chapter, we cover the practical aspects of the Riemannian Frank–Wolfe algorithm. The predominant effort of our work has gone into implementing the Riemannian Frank–Wolfe algorithm, and we discuss in detail how we have implemented two different versions of the solver: one using a Riemannian oracle, and one using a subsolver. We then perform some numerical experiments, demonstrating how the solvers perform in practice and how they compare to each other.

6.1 Implementation details

All our code is implemented in the Julia programming language [31]. The Riemannian Frank–Wolfe algorithm is implemented in the style of the Manopt.jl [5] framework, in order to work with the existing features of the framework and be as general-purpose as possible. Manopt.jl is built upon a general interface for working programmatically with manifolds, provided by Manifolds.jl and ManifoldsBase.jl [32]. To ensure reproducibility of our results, we provide here the version numbers for the software packages that have been used in the development of the code, as well as in the experiments described in this section:

- *Julia*: Version 1.6.2
- *Manopt.jl*: v0.3.26
- *Manifolds.jl*: v0.8.3
- *ManifoldsBase.jl*: v0.13.8

The Manifolds.jl interface

Briefly, the Manifolds.jl and ManifoldsBase.jl packages provide an interface for working with general manifolds in the Julia language. The ManifoldsBase.jl interface facilitates implementing generic methods that work independently of specific manifolds, by providing access to common manifold properties such as tangent spaces, geodesics and the exponential map. Various classes of manifolds are already implemented in Manifolds.jl, and since they all use the same interface it is easy to write code that works on arbitrary manifolds.

The Manopt.jl package is based on the interface provided by ManifoldsBase.jl, which allows for writing generic optimization algorithms that work on arbitrary manifolds and objective functions. The high-level interface simply requires a function call of the desired optimization method, giving the manifold, objective function and other required parameters as arguments. Other parameters, such as step sizes and stopping criteria, can also be specified as optional arguments – otherwise they use the default values of the given method. Manopt.jl provides general implementations of parameters like step sizes and stopping criteria, which are meant to work for every optimization method implemented into the framework. Further, it also allows easily recording various values, such as iterates, cost function values or stepsizes, during runtime. As an example, a very basic function call to the gradient descent algorithm could look like

```
gradient_descent(M, F, gradF).
```

Implementing the Frank–Wolfe algorithm in Manopt.jl

On a lower level, a Manopt.jl solver takes its input arguments and sets up two objects: a *problem* and an *options* object. The *problem* object stores the constant parameters of the solver, such as the objective function, while the *options* store transitory variables like the current iterate and stepsize. These objects are specialized for the given method, so in the case of the Frank–Wolfe algorithm we have implemented the `AbstractFrankWolfeProblem` and `AbstractFrankWolfeOptions` abstract types. These are subtypes of the `AbstractGradientProblem` and `AbstractGradientOptions` types, allowing us to use their previously implemented functionality for working with gradients. The Frank–Wolfe types are further subdivided into specific structs for the oracle and subsolver cases, as we have implemented the high-level interface for the two variants in separate functions.

When the problem and options objects are set up, the solver runs the function `solve`, which runs the main loop of the algorithm until the given stopping criteria is reached (or the default, if none are provided). The actions of `solve` is split into two main subroutines: `initialize_solver!`, which performs the required initialization steps for the algorithm, and `step_solver!`, which computes each iteration of the loop. (Note the exclamation marks, which in the Julia language signify functions that change one or more of their input arguments.)

Our implementation of the Frank–Wolfe method is built upon this foundation. As the method does not require any particular initialization steps, `initialize_solver!` just returns immediately. It is `step_solver!` that does all the heavy lifting of the algorithm, implementing lines 3 through 5 of Algorithm 2. Using Julia's multiple dispatch capabilities, the implementation of `step_solver` is split into separate functions for the oracle and subsolver versions of the algorithm, depending on the type of its arguments – that is, the *problem* and *options*. This allows us to share most of the code used by the solver between the two implementations.

Both versions of the algorithm share a common subroutine for the geodesic step in line 5, and the stepsize in line 4 is handled (by default) by the built-in `DecreasingStepsize`. When running the algorithm, it is also possible to choose any of the other stepsizes implemented into Manopt.jl, such as `ArmijoLinesearch`. Thus, the main difference in the implementation of the two variants of the algorithm lies in how they handle the subproblem.

Riemannian Frank–Wolfe with oracle

As it is the *black-box* Riemannian oracle that carries the main computational burden of this form of the Frank–Wolfe algorithm, the implementation of the oracle-based algorithm is fairly simple. The main high-level call to the Frank–Wolfe solver takes in as a (required) argument a function providing the oracle, which is then passed on to the `FrankWolfeOracleProblem` object. Since we assume that the oracle only returns points that satisfy the constraints, this version of the solver doesn't need to handle the constraints directly.

When `step_solver!` is called at each iteration, it first computes the gradient of the cost function at the current iterate p_k. This is then passed on to the oracle function, which computes its (closed-form) solution to the subproblem. Finally, this is input to the geodesic step subroutine, which computes the next iterate.

Riemannian Frank–Wolfe with subsolver

As the Frank–Wolfe algorithm is among the first algorithms in the Manopt framework to use a subsolver in the implementation, we had to find a good solution for how to achieve this, while still staying compatible with the rest of the interface. In the subsolver implementation, the high-level Frank–Wolfe call takes as input a *problem* and an *options* object, corresponding to the analogous objects that would be created internally when calling the high-level function for the solver method to be used on the subproblem.

As these objects are not meant to be part of the high-level interface of Manopt.jl, we have also created a helper function that sets up these objects for the user. We remark that this solution is not perfect, as users of the high-level interface should, ideally, not be required to deal with these low-level features at all. Thus, while our implementation works well enough as a proof of concept, this is something that should be addressed in further development.

The function also takes the constraint functions, in the form of vector-valued functions representing inequality and equality constraints, as well as their gradients. These are then passed on to the subsolver. Once the subsolver has been setup, the rest of the work is done by `step_solver!`. Again, it starts by computing the gradient of the cost function at the current iterate p_k, which it then uses to generate the cost function for the subproblem at the current step:

$$g(q) = \langle \operatorname{grad} f(p_k), \operatorname{Log}_{p_k}(q) \rangle_{p_k}.$$

Next, the gradient of g is computed as in section 4.2. This is done with a call to the built-in Manopt.jl function `adjoint_differential_log_argument`. The sub-problem cost function and its gradient is passed on to the subsolver *problem*, along with the current iterate. Then the `solve` function is called on the subsolver *problem* and *options* in order to retrieve the minimizer of the subsolver. Finally, the geodesic step subroutine is called in order to compute the next iterate.

The fake oracle

Instead of implementing the subsolver-based algorithm as a separate solver, we could also just use the oracle version, and provide it with a so-called *fake oracle* – a function that serves the same purpose as a Riemannian oracle, but instead of providing a closed-form solution, it uses an iterative solver to solve the subproblem. However, one would imagine that a native implementation of the subsolver could be more efficient, as it works

on the subproblem directly – hence avoiding the computational overhead of calling the high-level interface of the internal solver.

In order to test whether having two different variants of the Frank–Wolfe algorithm is actually necessary, we also test this hybrid approach where we use a fake oracle. If the internal solver of the fake oracle is given the exact same parameters as the subsolver, the iterates produced should be the same. Note that this is indeed the case in our implementation as well – the iterates are identical to those of the subsolver. The question is then whether the native subsolver implementation is able to compute the iterates more efficiently, generating the same sequence of iterates in a shorter amount of time.

6.2 Computing the Riemannian centroid of SPD matrices

In our numerical demonstration we consider a problem on the manifold of symmetric positive-definite (SPD) matrices of size $n \times n$, denoted $\mathcal{P}_n$. This problem is one of the examples studied by Weber and Sra [9], and we use the same approach as them in our demonstration. The class of problems we will consider is

$$\arg\min_{p \in \mathcal{C}} f(p), \tag{6.1}$$

where the constraint set is on the form

$$\mathcal{C} = \{p \in \mathcal{P}_n \mid L \preceq p \preceq U\}.$$

This constraint set, a *positive-definite interval* (see section 2.4), is geodesically convex, but does not admit easy projections [9]. We will further assume that $L \prec U$, since otherwise we would already have the solution $p = L = U$.

The specific example problem we consider is that of finding the (weighted) geometric matrix mean, also known as the Riemannian centroid. While this is a simple problem for real numbers, due to the noncommutativity of matrices there does not exist a closed-form solution in the matrix case. However, we can instead view this as the optimization problem

$$\arg\min_{p \in \mathcal{P}_n} \sum_{i=1}^{m} w_i \operatorname{dist}(p, s_i)^2, \tag{6.2}$$

where $\{w_i\}$ is a set of m weights, and $\{s_i\}$ is a set of m *data matrices*. We assume that not all the data matrices are equal, as otherwise the problem is not particularly interesting. This problem, when the matrices considered are symmetric positive-definite, is in fact geodesically convex [33].

The objective function $f(p) = \sum_{i=1}^{m} w_i \operatorname{dist}(p, s_i)^2$ is thus a weighted sum of squared distances. Bhatia [34] shows that if $g(p) = \operatorname{dist}(p, q)^2$, then $\operatorname{grad} g(p) = p^{-1} \log(pq^{-1})$. It follows that

$$\operatorname{grad} f(p) = \sum_{i=1}^{m} w_i p^{-1} \log(p s_i^{-1}).$$

One can also show that the following inequality, well-known from the real numbers, also holds in the matrix case [34]. Let H, G, A denote the harmonic, geometric and arithmetic means, respectively, with H and A given by

$$H = \left(\sum_{i=1}^{m} w_i s_i^{-1} \right)^{-1}, \quad A = \sum_{i=1}^{m} w_i s_i.$$

Then it holds that

$$H \preceq G \preceq A.$$

Thus, it makes sense to impose this as our constraints, i.e. $\mathcal{C} = \{p \in \mathcal{P}_n \mid H \preceq p \preceq A\}$. Note that this is, in some sense, not a *true* constraint – we have merely found a set that we know to contain the true solution, and hence limit our search to this set. Thus, this could just as well be solved as an unconstrained problem, as the constraints do not place any additional restrictions on the solution.

This means that the problem (6.1) does not really demonstrate the constrained optimization capabilities of the Riemannian Frank–Wolfe algorithm; in fact, we could just as well use an algorithm for unconstrained Riemannian optimization, like gradient descent. It does, however, provide a nice demonstration of how the Frank–Wolfe algorithm can be used to solve optimization problems by using a Riemannian oracle.

Experimental setup

We will now compare the performance of the different Frank–Wolfe variants on the problem (6.1). We solve the problem with the method using a Riemannian oracle and with the subsolver variant, and we also compare against using the oracle-based solver with an iterative *fake oracle* subsolver. As a performance baseline, we compare these methods against the gradient descent solver in Manopt.jl.

We generate random data on the SPD manifold by first generating a random point $p \in \mathcal{P}_n$. We then generate a set of m tangent vectors $X_i \in \mathcal{T}_p\mathcal{P}_n$, using a Gaussian distribution with standard deviation σ. For each $i = 1, \cdots, m$, we then have $s_i = \operatorname{Exp}_p(X_i)$. Thus, given that m is large enough (and σ is not too large relative to p), we would expect that the (unweighted) geometric mean of the points coincides with the base point p. We compute this *true mean* using the built-in `mean` function in manopt. We solve the minimization problem (6.2), with equal weights $w_i = 1/m$, so the problem simplifies to

$$\operatorname*{arg\,min}_{H \preceq p \preceq A} f(p) = \frac{1}{m} \sum_{i=1}^{m} \operatorname{dist}(p, s_i)^2. \tag{6.3}$$

In most experiments, and unless otherwise noted, we initialize the solvers with a random point on $\mathcal{P}_n$.

In the oracle solver, we use the Riemannian oracle derived in section 4.2.2. To avoid errors caused by numerical inaccuracies, we symmetrize the oracle output at each function call, $q \leftarrow \frac{1}{2}(q + q^\mathsf{T})$.

In the subsolver implementation, we use a gradient descent solver to solve the subproblem. Internally, this solver uses a decreasing stepsize, starting at $\beta_0 = 0.5$, with $\beta_{k+1} = 0.9\beta_k$, and stops when either the norm of the subproblem gradient reaches below a certain threshold, $||(D_q \operatorname{Log}_{p_k})^*(Z)[X]||_{p_k} < 10^{-9}$, or when a maximum of 10 iterations have been reached. We also use the same setup for the internal solver of the fake oracle.

For the gradient descent solver, we have used a constant stepsize $\alpha = 1$. We also tested it with a line search, but in our experiments this seems to give worse results for this problem.

We use the same stopping criterion for all the methods. The algorithm terminates once the error in the cost function, evaluated at the current iterate, reaches below a certain threshold τ:

$$\Delta_k = f(p_k) - f(p^*) < \tau.$$

Results and analysis

Table 6.1: Convergence times for the various solvers when used to minimize problem (6.3), with $\sigma = 0.1$.

m	50			500		
n	6	12	18	6	12	18
Frank Wolfe (oracle)	1.48 s	3.50 s	6.05 s	29.48 s	59.34 s	143.40 s
Frank Wolfe (subsolver)	2.88 s	19.56 s	95.76 s	5.49 s	27.56 s	106.27 s
Frank Wolfe (fake oracle)	2.22 s	17.37 s	75.83 s	4.57 s	24.80 s	107.64 s
Gradient descent	0.93 s	1.55 s	2.19 s	4.93 s	9.06 s	20.67 s

Table 6.1 shows the time for the algorithms to converge for $\sigma = 0.1$, $n = 6, 12, 18$, and for two different sizes of datasets, $m = 50, 500$, initialized to a random point in $\mathcal{P}_n$. Figure 6.1 shows the convergence of the iterates when running the solvers with $n = 6, 12, 18$, $m = 500$ and $\sigma = 0.1, 0.5$, to a convergence threshold of $\tau = 10^{-6}$. The convergence plots for $m = 50$ are almost identical, so we show them only for $m = 500$. Note that since the iterates produced by the fake oracle are the same as those produced by the subsolver, we do not include it in the iteration plots.

The first thing we note from table 6.1 is that the runtime difference between the subsolver and the fake oracle seems to be negligible, with differences usually on the order of at most a second. Only for larger n does the difference seem to become more prominent, however, and the fake oracle actually performs significantly better than the subsolver in some cases. This is somewhat surprising, as we expected that the subsolver implementation would perhaps be slightly more efficient than running the same computation, indirectly, through the oracle implementation.

We note also that the oracle solver usually doesn't reach the desired convergence threshold within the 50 iterations shown in the plots, whereas the other solvers usually converge quite quickly. However, the iterates don't tell the whole story: Compared to gradient descent, and especially to the other Frank–Wolfe solvers, the Riemannian oracle version of Frank–Wolfe actually computes its iterations significantly faster – taking roughly half the time to perform the same amount of iterations as gradient descent. This is perhaps not so surprising: The Riemannian oracle performs a quite efficient matrix computation once per iteration, whereas the subsolvers perform a gradient descent internally each time; one would expect that the iterations would take longer, but that the algorithm then converges in fewer iterations.

On the smaller datasets ($m = 50$), the oracle outperforms the subsolver algorithms; especially for larger problem dimensions n, the oracle is much faster. However, when the dataset gets larger, this difference seems to even out. In some cases, particularly for small n and larger m and σ, the fake oracle even outperforms gradient descent.

Table 6.2 shows the number of iterations and the time required for each of the solvers to converge, for $\sigma = 0.4, 0.8, 1.2$, with $n = 12$ and $m = 250$. We notice that while the other solvers seem largely unaffected by the change in variance of the dataset, the oracle solver performs increasingly worse as σ increases. This can also be seen from figure 6.1, where the behaviour of the oracle iterates changes much more prominently than that of the other solvers. Seeing as using the same oracle-based solver implementation with the fake oracle performs (at least) on par with the subsolver-based version in every case, we conclude that these differences must be because of the Riemannian oracle.

Table 6.2: Time and number of iterations required for the solvers to converge, with $n = 12$ and $m = 250$.

	Time			Iterations		
σ	0.4	0.8	1.2	0.4	0.8	1.2
Frank Wolfe (oracle)	11.88 s	13.69 s	25.74 s	68	70	127
Frank Wolfe (subsolver)	3.96 s	2.30 s	4.76 s	4	2	3
Frank Wolfe (fake oracle)	3.74 s	2.31 s	3.08 s	4	2	3
Gradient descent	1.62 s	2.05 s	3.19 s	5	6	8

The only way the oracle depends (directly) on the data is through the upper and lower *boundary matrices*, H and A. When σ is small, the entire dataset is clustered close together, which means that the positive-definite interval bounded by H and A is small. This effect is further compounded if the dataset is large. Thus it appears that the performance of the oracle is highly dependent on how far apart these bounding matrices are.

In most of our experiments, with the exception of those where we use very low values of σ, the oracle-based solver exhibits a noticeable zig-zagging behaviour. While this would be expected in the general, constrained case, it is somewhat surprising for this problem, which is essentially unconstrained – and for which the solution is on the interior of $\mathcal{C}$, anyway. Notably, with a similar setup, Weber and Sra [9] achieve good performance with the Riemannian oracle. We are not sure what causes this, but we assume the problem lies in the oracle itself.

Finally, we remark that he initialization of the solvers seems to make a difference in the performance. When initialized to a random point, both gradient descent and the subsolvers (which, we note, uses gradient descent internally) usually outperform the oracle-based Frank–Wolfe solver. However, when we initialize the algorithms to the harmonic-arithmetic mean $\frac{1}{2}(A + H)$ – which Weber and Sra [9] note is a good approximation for the geometric mean, thus providing a initial point fairly close to the solution – the oracle-based solver converges in around the same time as with a random point, while the other methods perform considerably worse. This could be caused by our choice of parameters for the gradient descent method (including the ones used in the subsolvers), which may not be appropriate for that initialization. However, while the oracle solver manages to retain its performance with this choice of starting point, it still does not perform on the same level as gradient descent initialized with a random point.

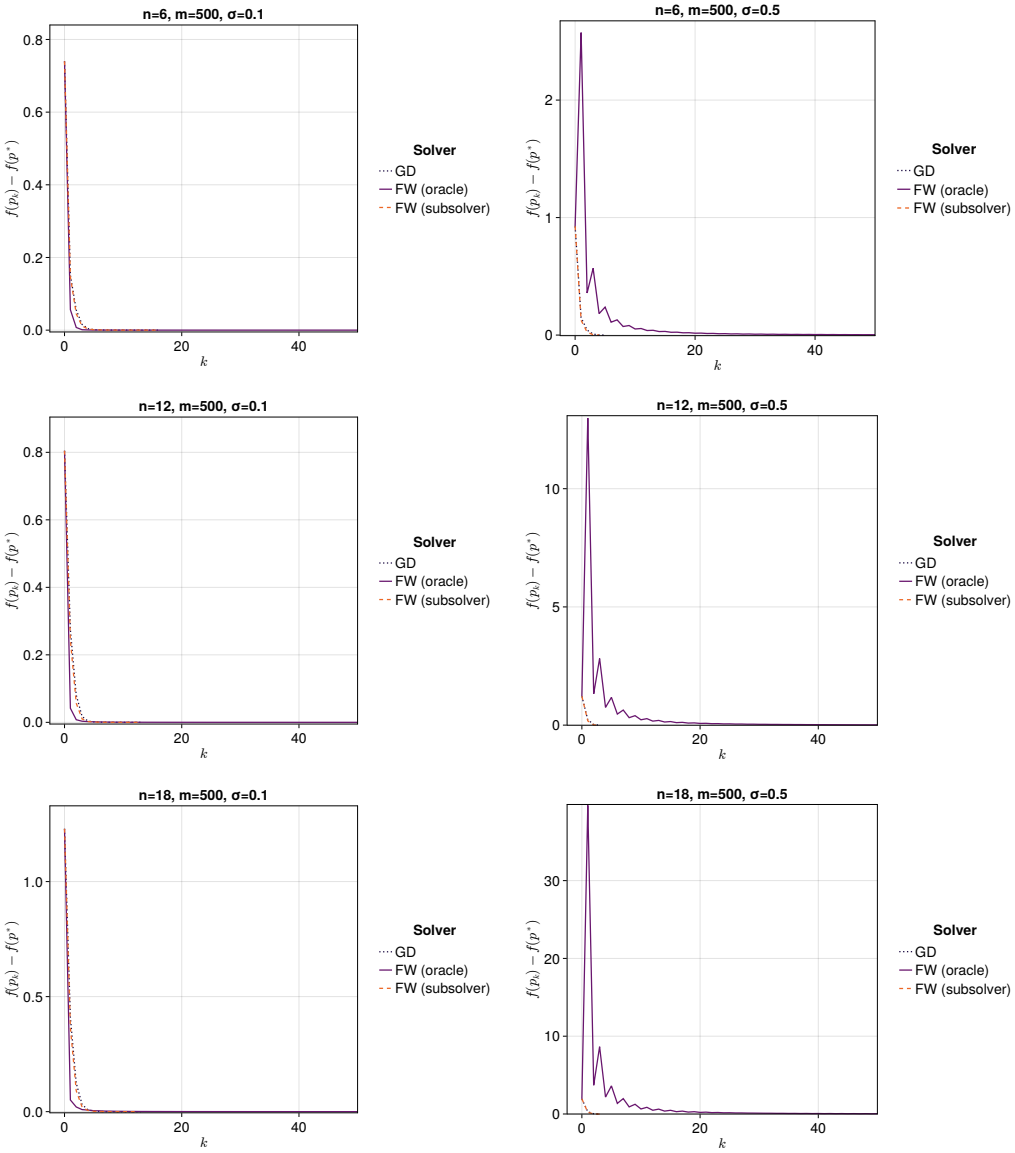


Figure 6.1: Plots showing the convergence of the iterates when the solvers are applied to problem (6.3). We use the parameters $n = 6, 12, 18$, $m = 500$ and $\sigma = 0.1, 0.5$.

Chapter 7

Closing remarks

As we have demonstrated in chapter 6, the Riemannian Frank–Wolfe provides adequate performance, although the results for the oracle-based solver on the Riemannian centroid problem on $\mathcal{P}_n$ were not as good as expected. This seems to be caused by the Riemannian oracle we used; in particular, the oracle for this problem seems to be highly dependent on the variance in the data . For this solver to be of practical use, we need access to an efficient Riemannian oracle, but the implementation of the solver itself seems to work as it should. In the future, more work could be done to explore which classes of problems admit efficient closed-form solutions to the Riemannian subproblem, for use with the oracle-based solver, and if these can perform more competitively against established methods.

We also discovered, somewhat surprisingly, that running the oracle solver with a subsolver-based *fake oracle* seems to outperform the native subsolver algorithm, at least in the specific setting we looked at. While the difference between the performance of the two implementations isn't too large, it could be worth looking more into this to see if both the Riemannian oracle case and the subsolver case could actually be handled by the oracle implementation.

We note that this essentially unconstrained problem of finding matrix centroids is not a setting where we expect Frank–Wolfe to outperform simpler methods like gradient descent. Unfortunately, we were unable to test the subsolver algorithm on a true constrained problem, as we did not have a constrained Riemannian solver that would work as a subsolver in our implementation. However, based on the results from our experiments on $\mathcal{P}_n$, we imagine that the subsolver algorithm could also perform satisfactorily on certain constrained problems.

The next step will naturally be to get these methods fully integrated into the Manopt.jl framework and make them available for users of the package. While we have developed working implementations of both the subsolver- and oracle-based algorithms, there are still some improvements that can be made, in particular with regard to making the subsolver more user-friendly.

Since the Euclidean Frank–Wolfe algorithm was first introduced in 1956, many variations have been developed in order to improve the performance of the algorithm. A logical next step would be to try adapting these methods to work on manifolds as well, and to investigate whether these improvements carry into the Riemannian setting. Among the most common variants are the *away-step* [26], *pairwise* [35] and *fully-corrective* [36] Frank–Wolfe methods, but other variations have also been proposed [22,

25, 29, 37]. Some of these variants even achieve global linear convergence rates, under certain assumptions. A fairly straightforward way to improve upon the standard Riemannian Frank–Wolfe algorithm would be to use more sophisticated techniques for choosing the stepsize. Although we have not gone beyond the simple decreasing stepsize recommended by Weber and Sra, many adaptive stepsize strategies are mentioned in the literature [22, 29, 38].

While the Frank–Wolfe algorithm cannot solve nonconvex problems directly, it can be used to find their stationary points. Although we have chosen to limit our scope to convex problems in this thesis, we note that Weber and Sra [9] also discuss applying the algorithm to nonconvex problems. This has been more thoroughly studied by Lacoste-Julien [39] in the Euclidean setting. Weber and Sra have also developed a stochastic version of the Riemannian algorithm [40], which works on nonconvex problems.

The work in this thesis represents some early steps in constrained convex optimization on manifolds, building primarily upon the initial work by Weber and Sra [9]. By implementing the Riemannian Frank–Wolfe algorithm into the general-purpose Riemannian optimization framework provided by Manopt.jl, and hence making it accessible (eventually) in a publicly available software package, we hope that our work may be of some use to others working in the field of constrained Riemannian optimization.

Bibliography

[1] Ding-Zhu Du, Panos M. Pardalos, and Weili Wu. ≪History of optimization≫. In: *Encyclopedia of Optimization*. Ed. by Christodoulos A. Floudas and Panos M. Pardalos. Boston, MA: Springer US, 2009, pp. 1538–1542. ISBN: 978-0-387-74759-0. DOI: 10.1007/978-0-387-74759-0_268.

[2] Nicolas Boumal. *An introduction to optimization on smooth manifolds*. To appear with Cambridge University Press. Apr. 2022. URL: http://www.nicolasboumal.net/book.

[3] N. Boumal, B. Mishra, P.-A. Absil, and R. Sepulchre. ≪Manopt, a Matlab Toolbox for Optimization on Manifolds≫. In: *Journal of Machine Learning Research* 15.42 (2014), pp. 1455–1459. URL: https://www.manopt.org.

[4] James Townsend, Niklas Koep, and Sebastian Weichwald. ≪Pymanopt: A Python Toolbox for Optimization on Manifolds using Automatic Differentiation≫. In: *Journal of Machine Learning Research* 17.137 (2016), pp. 1–5. URL: http://jmlr.org/papers/v17/16-177.html.

[5] Ronny Bergmann. ≪Manopt.jl: Optimization on Manifolds in Julia≫. In: *Journal of Open Source Software* 7.70 (2022), p. 3866. DOI: 10.21105/joss.03866.

[6] Wolfgang Ring and Benedikt Wirth. ≪Optimization Methods on Riemannian Manifolds and Their Application to Shape Space≫. In: *SIAM Journal on Optimization* 22.2 (2012), pp. 596–627. DOI: 10.1137/11082885X. URL: https://doi.org/10.1137/11082885X.

[7] Adrian Hauswirth, Saverio Bolognani, Gabriela Hug, and Florian Dörfler. ≪Projected gradient descent on Riemannian manifolds with applications to online power system optimization≫. In: *2016 54th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. 2016, pp. 225–232. DOI: 10.1109/ALLERTON.2016.7852234.

[8] Shixiang Chen, Shiqian Ma, Anthony Man-Cho So, and Tong Zhang. ≪Proximal Gradient Method for Nonsmooth Optimization over the Stiefel Manifold≫. In: (Nov. 2018). arXiv: 1811.00980 [math.OC].

[9] Melanie Weber and Suvrit Sra. *Riemannian Optimization via Frank-Wolfe Methods*. Nov. 2021. arXiv: 1710.10770 [math.OC].

[10] Marguerite Frank and Philip Wolfe. ≪An algorithm for quadratic programming≫. In: *Naval Research Logistics Quarterly* 3.1-2 (1956), pp. 95–110. DOI: https://doi.org/10.1002/nav.3800030109. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/nav.3800030109.

[11] J J O'Connor and E F Robertson. *MacTutor History of Mathematics: Marguerite Josephine Straus Frank*. Nov. 2020. URL: https://mathshistory.st-andrews.ac.uk/Biographies/Frank_Marguerite/.

[12] Melanie Weber and Suvrit Sra. ≪Frank-Wolfe methods for geodesically convex optimization with application to the matrix geometric mean≫. In: (Oct. 2017). arXiv: 1710.10770 [math.OC].

[13] P.A. Absil, R. Mahony, and R. Sepulchre. *Optimization Algorithms on Matrix Manifolds*. Princeton University Press, 2009. ISBN: 9781400830244.

[14] Manfredo do Carmo. *Riemannian Geometry*. Mathematics: Theory and Applications. Birkhäuser Boston, 1992. ISBN: 9780817634902.

[15] Johannes Persch. ≪Optimization Methods for Manifold-Valued Image Processing≫. PhD thesis. Technische Universität Kaiserslautern, Feb. 2018.

[16] Ronny Bergmann and Pierre-Yves Gousenbourger. ≪A variational model for data fitting on manifolds by minimizing the acceleration of a Bézier curve≫. In: (July 2018). DOI: 10.3389/fams.2018.00059. arXiv: 1807.10090 [math.NA].

[17] Miroslav Bačák. *Convex Analysis and Optimization in Hadamard Spaces*. De Gruyter Series in Nonlinear Analysis and Applications. De Gruyter, 2014. ISBN: 9783110391084.

[18] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004. DOI: 10.1017/CBO9780511804441.

[19] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. 2006. ISBN: 978-0387-30303-1.

[20] Rajendra Bhatia. *Matrix Analysis*. 1997.

[21] Cyrille W. Combettes and Sebastian Pokutta. *Complexity of Linear Minimization and Projection on Some Sets*. 2021. DOI: 10.48550/ARXIV.2101.10040. URL: https://arxiv.org/abs/2101.10040.

[22] Simon Lacoste-Julien and Martin Jaggi. ≪On the Global Linear Convergence of Frank-Wolfe Optimization Variants≫. In: (Nov. 2015). arXiv: 1511.05932 [math.OC].

[23] Evgeny S. Levitin and Boris T. Polyak. ≪Constrained minimization methods≫. In: *Ussr Computational Mathematics and Mathematical Physics* 6 (1966), pp. 1–50.

[24] M. D. Canon and C. D. Cullum. ≪A Tight Upper Bound on the Rate of Convergence of the Frank-Wolfe Algorithm≫. In: *SIAM Journal on Control* 6.4 (1968), pp. 509–516. DOI: 10.1137/0306032.

[25] Zhaoyue Chen, Mokhwa Lee, and Yifan Sun. ≪Continuous Time Frank-Wolfe Does Not Zig-Zag, But Multistep Methods Do Not Accelerate≫. In: (June 2021). arXiv: 2106.05753 [math.OC].

[26] Jacques Guélat and Patrice Marcotte. ≪Some comments on Wolfe's 'away step'≫. In: *Mathematical Programming* 35 (1986), pp. 110–119.

[27] Changshuo Liu and Nicolas Boumal. *Simple algorithms for optimization on Riemannian manifolds with constraints*. 2019. DOI: 10.48550/ARXIV.1901.10000. URL: https://arxiv.org/abs/1901.10000.

[28] Jan-Philipp Pfaue. ≪Constrained optimization on Riemannian manifolds using geodesic polygonal sets≫. Bachelor's Thesis. Technische Universität Chemnitz, Jan. 2022.

[29] Martin Jaggi. ≪Revisiting Frank-Wolfe: Projection-Free Sparse Convex Optimization≫. In: *Proceedings of the 30th International Conference on Machine Learning*. Ed. by Sanjoy Dasgupta and David McAllester. Vol. 28. Proceedings of Machine Learning Research 1. Atlanta, Georgia, USA: PMLR, June 2013, pp. 427–435. URL: https://proceedings.mlr.press/v28/jaggi13.html.

[30] Dan Garber and Elad Hazan. ≪Faster Rates for the Frank-Wolfe Method over Strongly-Convex Sets≫. In: (June 2014). arXiv: 1406.1305 [math.OC].

[31] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. ≪Julia: A fresh approach to numerical computing≫. In: *SIAM Review* 59.1 (2017), pp. 65–98. DOI: 10.1137/141000671. URL: https://epubs.siam.org/doi/10.1137/141000671.

[32] Seth D. Axen, Mateusz Baran, Ronny Bergmann, and Krzysztof Rzecki. *Manifolds.jl: An Extensible Julia Framework for Data Analysis on Manifolds*. 2021. arXiv: 2106.08777.

[33] Miklós Pálfia. ≪Operator means of probability measures and generalized Karcher equations≫. In: *Advances in Mathematics* 289 (2016), pp. 951–1007. ISSN: 0001-8708. DOI: https://doi.org/10.1016/j.aim.2015.11.019. URL: https://www.sciencedirect.com/science/article/pii/S000187081500479X.

[34] R. Bhatia. *Positive Definite Matrices*. Princeton Series in Applied Mathematics. Princeton University Press, 2007. ISBN: 9781400827787.

[35] B. F. Mitchell, V. F. Dem'yanov, and V. N. Malozemov. ≪Finding the Point of a Polyhedron Closest to the Origin≫. In: *Siam Journal on Control* 12 (1974), pp. 19–26.

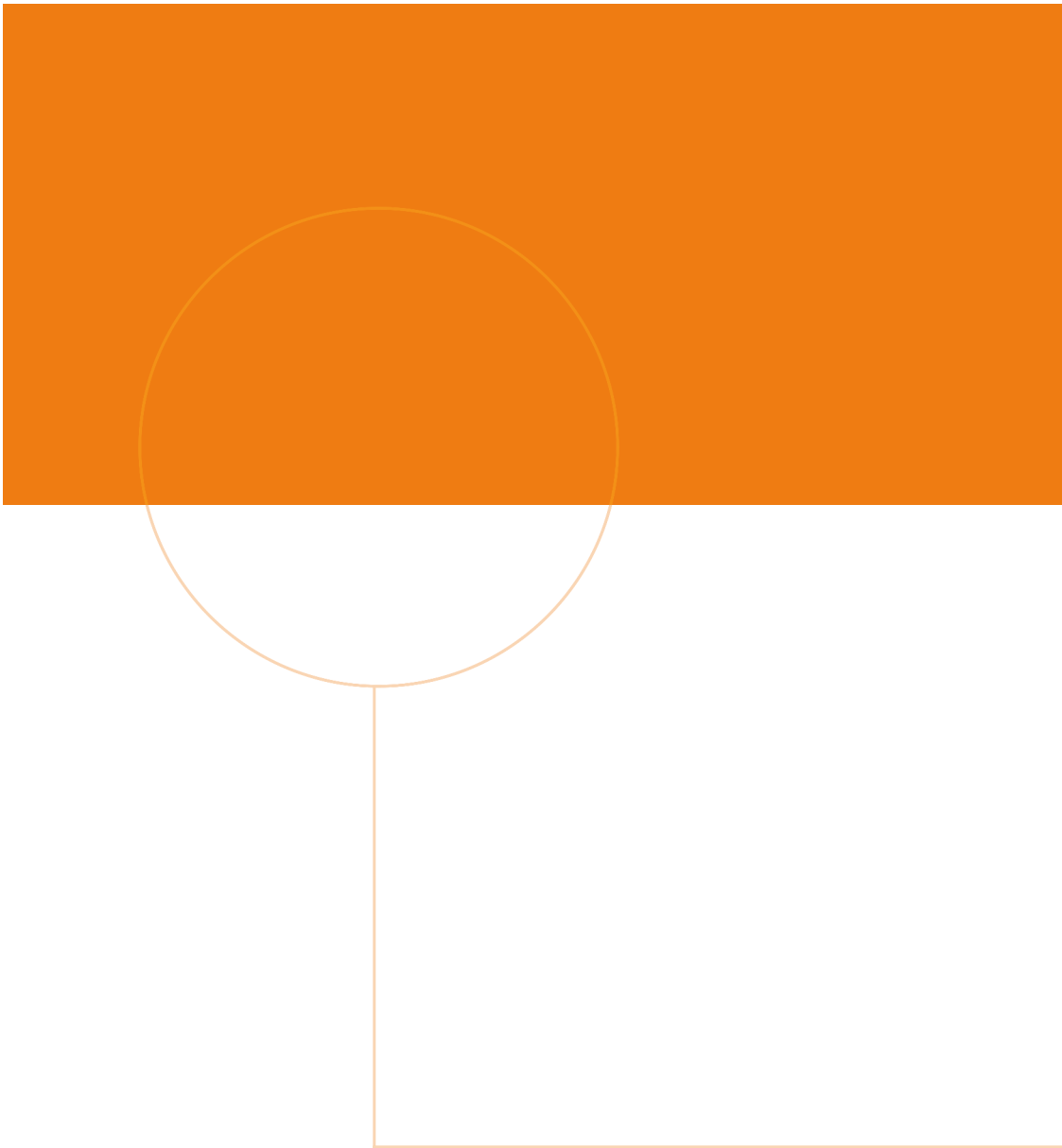
[36] Charles A. Holloway. ≪An extension of the frank and Wolfe method of feasible directions≫. In: *Mathematical Programming* 6 (1974), pp. 14–27.

[37] J.C Dunn and S Harshbarger. ≪Conditional gradient algorithms with open loop step size rules≫. In: *Journal of Mathematical Analysis and Applications* 62.2 (1978), pp. 432–444. ISSN: 0022-247X. DOI: https://doi.org/10.1016/0022-247X(78)90137-3. URL: https://www.sciencedirect.com/science/article/pii/0022247X78901373.

[38] Fabian Pedregosa, Geoffrey Negiar, Armin Askari, and Martin Jaggi. ≪Linearly Convergent Frank-Wolfe with Backtracking Line-Search≫. In: *Proceedings of the 23rd International Conference on Artificial Intelligence and Statistics (AISTATS) 2020* (June 2018). arXiv: 1806.05123 [math.OC].

[39] Simon Lacoste-Julien. ≪Convergence Rate of Frank-Wolfe for Non-Convex Objectives≫. In: (July 2016). arXiv: 1607.00345 [math.OC].

[40] Melanie Weber and Suvrit Sra. ≪Projection-free nonconvex stochastic optimization on Riemannian manifolds≫. In: (Oct. 2019). arXiv: 1910.04194 [math.OC].



NTNU

Norwegian University of
Science and Technology