SANDIA REPORT

SAND2010-xxxx
Unlimited Release
Printed January 2010

A Linear Algebra Interpretation of Non-Euclidean Scalar Products and Vector Spaces and their impact on Numerical Models and Algorithms

Roscoe A. Bartlett
Optimization and Uncertaintly Quantification

Denis Ridzal
Optimization and Uncertaintly Quantification

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2010-xxxx
Unlimited Release
Printed January 2010

A Linear Algebra Interpretation of Non-Euclidean Scalar Products and Vector Spaces and their impact on Numerical Models and Algorithms

Roscoe A. Bartlett
Optimization and Uncertaintly Quantification

Denis Ridzal
Optimization and Uncertaintly Quantification

Sandia National Laboratories [*], Albuquerque NM 87185 USA,

Abstract

Many numerical algorithms are derived, analyzed and expressed with respect to Euclidean vector spaces. However, many applied mathematicians have shown the utility of expressing and implementing many different types of numerical algorithms with respect to non-Euclidean vector spaces. Coming from a functional analysis background, it natural to express many types of numerical algorithms in non-Euclidean form by introducing the notion of an scalar (or inner) product. The introduction of a non-Euclidean vector space an scalar product fundamentally changes the meaning of linear operators, derivatives, and other constructs commonly used the express numerical algorithms. The purpose of this paper is to provide a foundation for understanding the meaning and implications of expressing numerical algorithms in non-Euclidean form in a way that does not require any knowledge of functional analysis. This discussion is based purely on basic finite-dimensional linear algebra. The goal is to provide the reader with a level of confidence in expressing and implementing numerical models and algorithms in non-Euclidean form. A simple procedure is presented for taking any numerical model and algorithm expressed using Euclidean vector spaces and translating it to non-Euclidean form in the most general way possible. Examples and analysis of the issues involved are demonstrated for different types of numerical algorithms such as Newton methods, quasi-Newton methods, optimization globalization methods, and inequality constraints. The goal of this paper is to not require anyone who writes numerical algorithms to become an expert in functional analysis in order to take advantage of non-Euclidean scalar products provided by the application. Instead, the goal is to empower non-functional-analysis experts with the ability to write numerical software with enough hooks to allow application domain experts with knowledge of the (infinite dimensional) structure of the problem to customize the algorithms in an efficient and practical way.

[*]Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed-Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.

Acknowledgment

The authors would like to thank ...

The format of this report is based on information found in [**?**].

Contents

Appendix

1 Introduction

Many numerical algorithms are written in terms of Euclidean vector spaces where dot products are used for the scalar inner product. For example, the inner loop of a linear conjugate gradient (CG) method $k = 0\ldots$ for solving $Ax = b$, initialized using $r = r_0 = b - Ax_0$, is often written in Euclidean (or dot-product) form as

$$\begin{aligned}
\rho_k &= r^H r, \\
p &= r + \frac{\rho_k}{\rho_{k-1}} p, \\
q &= Ap, \\
\alpha &= \frac{\rho_k}{p^H q}, \\
x &= x + \alpha p, \\
r &= r - \alpha q.
\end{aligned}$$

An experienced mathematician with knowledge of functional analysis will look at the above algorithm an immediately write down the generalized form for non-Euclidean vector spaces by replacing the dot products $r^H r$ and $p^H q$ with the scalar products $\langle r, r \rangle$ and $\langle p, q \rangle$ and restate the inner loop of the above CG algorithm as

$$\begin{aligned}
\rho_k &= \langle r, r \rangle, \\
p &= r + \frac{\rho_k}{\rho_{k-1}} p, \\
q &= Ap, \\
\alpha &= \frac{\rho_k}{\langle p, q \rangle}, \\
x &= x + \alpha p, \\
r &= r - \alpha q.
\end{aligned}$$

Just as with linear CG, many numerical algorithms expressed in Euclidean form with dot products and Euclidean norms $||.||_2$ (such as various optimization algorithms, stability analysis methods, time integration methods, etc.) have straightforward extensions to non-Euclidean vector spaces. What we would like is to have a straightforward process by which we can analyze many different types of existing numerical algorithms expressed in Euclidean form and then write out, if possible, the more general non-Euclidean form of these algorithms. We also want to do this in such a way that we do not have to revisit all of the mathematical assumptions and theorems that went into the development of the algorithm.

Someone not intimately familiar with general scalar products and non-Euclidean vector spaces will naturally ask the following questions. What's the big deal in replacing dot products with scalar products? What is this scalar product $\langle ., . \rangle$ and what does this mean? What is the relationship between vectors p and q for algorithms stated in Euclidean form and in non-Euclidean form? By what justification can one just replace dot products like $p^H q$ with scalar products $\langle p, q \rangle$? What other changes do we need to make due to this subtle change of replacing dot products with scalar products? How are the definition of linear operators, model function derivatives, and other mathematical

objects affected by the introduction of non-Euclidean scalar products? What does all of this buy you?

Here we seek to answer all of these questions in a way that a person without knowledge of functional analysis or other advanced mathematics can understand and appreciate. All that we assume is that the reader has a basic understanding of linear algebra and a familiarity with multi-variable numerical algorithms like Newton's method [???] for nonlinear equations.

Here we present a linear algebra interpretation of finite dimensional non-Euclidean inner product spaces and how they influence numerical algorithms and mathematical models. The goal of this treatment is to present this topic in a way that non-mathematicians can understand and appreciate. The basic approach will be to show the relationship between typical Euclidean-based vectors and vector spaces (i.e. where the dot product is used for the inner product and a linear operator is equivalent to a matrix) and non-Euclidean basis representations, vectors, and vector spaces (i.e. where the inner product is defined by a positive-definite matrix and a linear operator is not necessarily equivalent to a matrix). What we will show is a straightforward way to take many different types of numerical algorithms that are expressed in Euclidean form and then to analyze them for non-Euclidean vectors and spaces and see if they can be transformed for use with general scalar products. What we will show is that the expression of a numerical algorithm in a non-Euclidean space is essentially equivalent to performing a linear (not just diagonal) transformation of variables and model functions except that we do not need to actually explicitly perform the transformation which has many different advantages which include:

- Performing the type of general transformations described here explicitly can be very difficult to implement in the code and can be very expensive to apply. Representing the desired transformations as just inner product operations can be much simpler and more efficient to code up and maintain from the application developers point of view.
- Explicitly performing a linear transformation of variables and functions in general will destroy the sparsity of derivative objects like Jacobians and Hessians. For large scale problems this type of transformation can make the algorithms intractable.
- Keeping the coefficients for the unknowns, and model functions and derivatives in the original form (and units) makes it easier for the user to inspect what a numerical algorithm is doing. Even simple diagonal scaling that is explicitly applied can make it more difficult to make sense of the variables and the functions that a numerical algorithm is working with and makes it more difficult to debug numerical problems when they occur. Keeping the unknown and function coefficients in original form also makes it easier to select convergence tolerances and other algorithmic parameters since the units are more nature to the user and knowledge of the physics or the engineering domain can be applied. Once a general linear transformation is explicitly applied, this becomes very difficult for a general user to do.

2 Introduction to vector spaces, basis representations, scalar products, and natural norms

In this section, we provide a quick overview of the concepts of finite-dimensional vector spaces, vector basis and coefficient representations, scalar products, and norms. The mathematical system described here is that of finite-dimensional Hilbert spaces [???]. Here we show straightforward connections between Euclidean and non-Euclidean representations of vectors and their relationship to linear transformation of variables. In this introductory material, we deal with general vectors in a complex space $\mathbb{C}^n$ with complex scalar elements.

2.1 Basis and coefficient representations of vectors and vector spaces

Consider a complex-valued vector space $\mathcal{S} \subseteq \mathbb{C}^n$ with the basis vectors $e_i \in \mathbb{C}^m$, for $i = 1 \ldots m$, such that any vector $x \in \mathbb{C}^n$ can be represented as the linear combination

$$x = \sum_{i=1}^{m} \tilde{x}_i e_i \tag{1}$$

where $\tilde{x} \in \mathcal{S}$ is known as the *coefficient vector* for x in the space $\mathcal{S}$. In order for the set of vectors $\{e_i\}$ to form a valid basis, they must minimally be linearly independent and $m \leq n$ must be true. (Note: This condition can be violated in some code implementations that actually have an overdetermined basis representation where the basis vectors are linearly dependent.) In a finite dimensional setting, when we say that some vector x is in some space $\mathcal{S}$ what we mean is that x can be composed out of a linear combination of the space's basis vectors as shown in (1).

Another way to represent (1) is in the matrix form

$$x = E\tilde{x} \tag{2}$$

where $E \in \mathbb{C}^{n \times m}$ is called the *Basis Matrix* who's columns are the basis vectors for the space $\mathcal{S}$; in other words

$$E = \begin{bmatrix} e_1 & e_2 & \ldots & e_m \end{bmatrix}. \tag{3}$$

The basis matrix form (2) will allow us to use standard linear algebra notation later in various types of derivations and manipulations.

The choice of which of the two different representations of a vector x or $\tilde{x}$ has a dramatic impact on the specification of the mathematical model's functions and on the interpretation of the operations in a numerical algorithm.

2.2 Standard *vector* operations

A few different types of operations can be performed on just the coefficients for a set of vectors which have the same meaning for the vectors themselves. These are the set of classic *vector* opera-

tions of assignment to zero, vector scaling, and vector addition which are stated as

- $x = 0$:

  $x = E\tilde{x} = 0 \;\Rightarrow\; \tilde{x} = 0$

- $z = \alpha x$:

  $z = E\tilde{z} = \alpha x = \alpha E\tilde{x} = E(\alpha\tilde{x}) \;\Rightarrow\; \tilde{z} = \alpha\tilde{x}$

- $z = x + y$:

  $z = E\tilde{z} = x + y = E\tilde{x} + E\tilde{y} = E(\tilde{x} + \tilde{y}) \;\Rightarrow\; \tilde{z} = \tilde{x} + \tilde{y}.$

Note that other types of element-wise operations on the coefficients such as element-wise products and divisions are not equivalent to the corresponding operations on the vectors themselves and are hence not *vector* operations.

2.3 Square, invertible basis representations

Up to this point, the vector space $\mathcal{S}$ can be a strict subspace of $\mathbb{C}^n$ since $m < n$ may be true. We will now focus on the case where $m = n$ which gives a non-singular basis matrix $E \in \mathbb{C}^{n\times n}$ that can be used to represent any vector $x \in \mathbb{C}^n$. As a result, E^{-1} is well defined and can be used in our expressions and derivations. However, we do not assume that the basis vectors $e_i \in \mathbb{C}^n$ are orthonormal or orthogonal.

2.4 Definition of the scalar (or inner) product for a vector space

Now consider the dot inner product of any two vectors $x, y \in \mathbb{C}^n$ which takes the well known form

$$x^H y = \sum_{i=1}^{n} \text{conjugate}(x_i) y_i. \tag{4}$$

Using the substitution $x = E\tilde{x}$ and $y = E\tilde{y}$, the inner product in (4) can be represented as

$$x^H y = (\tilde{x}^H E^H)(E\tilde{y}) = \tilde{x}^H Q \tilde{y}, \tag{5}$$

where $Q = E^H E$ is a symmetric positive-definite matrix. It is this matrix Q that is said to define the scalar (or inner) product of two coefficient vectors $\tilde{x}, \tilde{y} \in \mathcal{S}$ as

$$x^H y = \tilde{x}^H Q \tilde{y} = \langle \tilde{x}, \tilde{y} \rangle_{\mathcal{S}}\,. \tag{6}$$

2.5 Definition of the natural norm for a vector space

The natural norm $||.||_{\mathcal{S}}$ of a general vector space is defined as

$$||x|| = \sqrt{x^H x} = \sqrt{\langle \tilde{x}, \tilde{x} \rangle_{\mathcal{S}}} = ||\tilde{x}||_{\mathcal{S}} \tag{7}$$

where $\langle \tilde{x}, \tilde{x} \rangle_{\mathcal{S}}$ is defined in terms of the scalar product matrix Q in (6).

2.6 Orthonormal and orthogonal basis representations

Note that all orthonormal sets of basis vectors, i.e.

$$e_i^H e_j = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}$$

result in an *orthogonal matrix*[1] E that gives identity for the scalar product matrix $Q = E^H E = I$. Therefore, all orthonormal sets of basis vectors result in a Euclidean scalar product, even if the basis vectors are not Cartesian (i.e. $e_i^T \neq \begin{bmatrix} 0 & \dots & 1 & \dots & 0 \end{bmatrix}$). Also note that all orthogonal sets of basis vectors give a scalar product matrix $Q = E^H E$ that is diagonal.

When the scalar product matrix Q is diagonal, it is trivial to compute a diagonal scaling matrix $E = Q^{1/2}$ and then use this scaling matrix to scale all vectors and operators before the numerical algorithm even sees them. In these cases, it is questionable whether the more general concept of scalar products is worth the effort in expressing and implementing numerical algorithms, which is our ultimate goal here. Therefore, we are primarily focused on problems that require more than simply diagonal scaling. However, keeping the original variables and not performing the scaling explicitly can help in debugging and in interpreting the output form a numerical algorithm.

2.7 Equivalence of basis representations and the scalar product

One important detail to mention is that given a particular vector space $\mathcal{S}$ with its corresponding scalar product defined using Q in (6), there are infinitely many different selections for the basis E that give the same scalar product matrix $Q = E^T E$. To see this, let $F \in \mathbb{C}^{n \times n}$ be any orthogonal matrix (i.e. $F^H F = I$). We can use any particular choice for F to transform the scalar product as

$$x^H y = \tilde{x}^H Q \tilde{y} = \tilde{x}^H (E^H E) \tilde{y} = \tilde{x}^H E^H (F^H F) E \tilde{y} = \tilde{x}^H (FE)^H (FE) \tilde{y} = (\tilde{x}^H \bar{E}^H)(\bar{E} \tilde{y}) = \bar{x}^H \bar{y} \tag{8}$$

where $\bar{x} = \bar{E}\tilde{x}$, $\bar{x} = \bar{E}\tilde{x}$, and $\bar{E} = FE$. We see that $\bar{E} \in \mathbb{C}^{n \times n}$ actually forms a different vector space $\bar{\mathcal{S}}$, but, for the same coefficient vectors, its scalar product is exactly the same as for $\mathcal{S}$. Therefore,

[1] In most linear algebra text books and literature, the term *orthogonal matrix* is used to denote a matrix who's columns are orthonormal. This means that a matrix with just orthogonal columns (i.e. $e_i^H e_j = \delta \neq 1$ when $i = j$) is not an orthogonal matrix. It would seem to make more sense that a matrix with orthogonal columns should be called an "orthogonal matrix" and a matrix with orthonormal columns should be called an "orthonormal matrix" but this is unfortunately not the standard meaning.

when we define a vector space by its scalar product, we are really defining an infinite collection of vector spaces instead of just one. This is because there are infinitely many different sets of basis vectors that give infinitely many different vector representations for a particular set of coefficients but all have the same scalar product.

2.8 Linear operators

A *linear operator* $A \in \mathbb{C}^m|\mathbb{C}^n$ is an object that maps vectors from the spaces $\mathbb{C}^n$ to $\mathbb{C}^m$ as

$$y = Ax, \tag{9}$$

where $x \in \mathbb{C}^n$ and $y \in \mathbb{C}^m$, and also obeys the linear properties

$$z = A(\alpha u + \beta v) = \alpha Au + \beta Av \tag{10}$$

for all $\alpha, \beta \in \mathbb{C}$ and $u, v \in \mathbb{C}^n$.

The *vector coefficient linear operator A,* denoted $\tilde{A}$, is defined as the operator that directly computes the coefficient vector $\tilde{y}$ given the coefficient vector $\tilde{x}$ as

$$\begin{aligned} y &= Ax \\ E_R\tilde{y} &= E_R\tilde{A}E_D^{-1}(E_D\tilde{x}) \\ &\Rightarrow \qquad (11) \\ \tilde{y} &= \tilde{A}\tilde{x}. \qquad (12) \end{aligned}$$

The coefficient form of a linear operator $\tilde{A}$ is what is actually implemented in a software since it directly deals with the stored coefficient vectors themselves. On the other hand, the general linear operator A is primarily a mathematical tool but it can also be a software abstraction in an abstract interface layer like Thyra [1].

The last type of linear operator to define is the *matrix form of a linear operator* denoted $\hat{A}$ which either stores actual matrix elements or could store them. The linear operator matrix $\hat{A}$ is what actually implements the coefficient form of the linear operator $\tilde{A}$. As described in the upcoming Section 3, the matrix form $\hat{A}$ may just be coefficient linear operator $\tilde{A}$ or it may be augmented with a scalar product operator. When using a Euclidean basis $E = I$, $x = \tilde{x}$ and $\hat{A} = \tilde{A} = A$. More generally, $\hat{A} \neq \tilde{A} \neq A$.

Note that every linear algebra quantity (e.g. $x, \tilde{y}, A, \hat{A}$, etc.) can be represented in simple element-wise coefficient form which we will denote here using square brackets as $[.]$ (e.g. $[x]$, $[\tilde{y}]$, $[A]$, $[\hat{A}]$, etc.). For example, the vector $x \in \mathbb{C}^n$ can be represented using the elements $[x]_{(i)}$ for $i = 1 \ldots n$. Likewise, a linear operator such as $\tilde{A} \in \mathbb{C}^m|\mathbb{C}^n$ can be represented using the matrix elements $[\tilde{A}]_{(i,j)}$ for $i \ldots m,\ j = 1 \ldots n$. Therefore, every linear operator application involving one of these linear algebra objects can always be broken down into element-wise form as

$$y = Ax$$

$$[y]_{(i)} \quad = \quad \overset{\Rightarrow}{\sum_{1}^{m}}[A]_{(i,j)}[x]_{(j)}, \text{ for } i = \ldots n. \tag{13}$$

Therefore, in the sequel when we use this notation $[.]$ and state that $[A] = [B]$, we mean that applying $[A]$ element by element as defined in (13) is equivalent to applying $[B]$ element by element and doing so will give the same output vector coefficients $[y]_{(i)}$.

For every linear operator A it is possible to define another linear operator object associated with it called the *adjoint* linear operator, denoted $A^H \in \mathbb{C}^n|\mathbb{C}^m$, which maps vectors from the spaces $\mathbb{C}^m$ to $\mathbb{C}^n$ as

$$y = A^H x \tag{14}$$

where $x \in \mathbb{C}^m$ and $y \in \mathbb{C}^n$.

For linear operators A in a Euclidean space, it will be shown in Section 3, the adjoint linear operator A^H is equal to the matrix element-wise conjugate transpose, or $A^H = [A]^H$ where we use the notation $[A]^H$ to denote the matrix element-wise conjugate transpose. However, for general vector spaces, this is not true.

Dumb Fact 2.1 *An adjoint linear operator $[\tilde{A}]$ is not the same as the matrix Hermitian transpose $[\tilde{A}]^H$ with respect to the action on the vector coefficients when dealing with non-Euclidean vector spaces.*

In other words, while the adjoint linear operator A^H in Euclidean space is equal to the Hermitian transpose of the forward operator A, this is not generally true for the vector coefficient linear operators $\tilde{A}$ in general vector spaces.

The forward and adjoint vector coefficient linear operators $\tilde{A}$ and $\tilde{A}^H$, respectively, are related to each other with respect to the scalar products through the adjoint relationship

$$v^H(Au) = \left\langle \tilde{v}, \tilde{A}\tilde{u} \right\rangle_{\mathcal{R}} = (A^H v)^H u = \left\langle \tilde{A}^H \tilde{v}, \tilde{u} \right\rangle_{\mathcal{D}} \tag{15}$$

for all $\tilde{u} \in \mathcal{D}$ and $\tilde{v} \in \mathcal{R}$. In (15) we see the relationship between a linear operator, its coefficient linear operator, its adjoint, and the scalar products associated with its range and domain spaces. The identity in (15) will be used in a number of derivations in the following sections.

A linear operator is designated as an *invertible linear operator* if another unique linear operator denoted $A^{-1} \in \mathcal{D}|\mathcal{R}$ exists such that

$$A^{-1}A = AA^{-1} = I.$$

Likewise, the inverse linear operator A^{-1} also has an *adjoint inverse linear operator* denoted $A^{-H} \in \mathcal{R}|\mathcal{D}$ associated with it which satisfies

$$A^{-H}A^H = A^H A^{-H} = I.$$

While the adjoint of a coefficient linear operator is not generally equal to the element-wise Hermitian transpose of the forward linear operator, the inverse is the element-wise matrix inverse in all cases. In other words, the inverse of a coefficient linear operator $\tilde{A}^{-1}$ is in fact equal to the matrix inverse of the forward linear operator $[\tilde{A}]^{-1}$. This will be shown out in Section 3.

Linear operators are used to represent a variety of different types of objects in a numerical algorithm. The origin of many linear operators that appear in a numerical algorithm are the derivatives of the smooth model functions such as Jacobian, gradients, and Hessians. These derivative operators play important roles in a number of numerical algorithms ranging from basic Newton methods all the way up through optimization methods. Even vectors $x \in \mathbb{C}^n$ can be viewed as linear operators $x \in \mathbb{C}^n|\mathbb{C}$ where the domain space for the forward operator is simply $\mathbb{C}$ which gives the forward operator $y = xv$ (where $v \in \mathbb{C}$ and $y \in \mathbb{C}^n$) and the adjoint operator $y = x^H v$ (where $v \in \mathbb{C}^n$ and $y \in \mathbb{C}$). Here we see that a vector x with the corresponding coefficient vector $\tilde{x}$ can take on a dual role. In one role, $\tilde{x}$ is just an array of coefficients with a scalar product attached to it. In another role, x can represent a linear operator where the action of $x^H y$ invokes the scalar product $\langle \tilde{x}, \tilde{y} \rangle$. In turns out that in a abstract interface like Thyra [1], the vector abstraction can be interpreted either way.

2.9 Dealing only with scalar products and vector coefficients in algorithm construction

It is important to recognize that both a vector x and its corresponding coefficient vector $\tilde{x}$ (where $x = E\tilde{x}$) can be represented as arrays of scalars in a computer program. However, our goal is to go about formulating and implementing numerical algorithms and applications so as to only manipulate arrays of the natural coefficient vectors $\tilde{x}$ and never manipulate the coefficients of the Euclidean representation of the vectors $x = (E\tilde{x})$ themselves. The reason that one would only want to deal with the natural coefficients of the vectors in a vector space and the scalar product is that it may be inconvenient and/or very expensive to build a set of basis vectors so that the Euclidean form of the vectors themselves can be formed and manipulated directly. A general linear transformation would also destroy the sparsity of many derivative operator matrix representations which is unacceptable for most large-scale algorithms. This is the case, for example, in many different finite-element discretization methods for PDEs [???].

3 Impact of non-Euclidean scalar products on matrix representations of linear operators

As stated above, for every general linear operator A there is a corresponding *vector coefficient linear operator* form $\tilde{A}$. In addition, every finite-dimensional linear operator has one of several potential *matrix representations* $\hat{A}$. The different representations of $\hat{A}$ depend on how the domain and range space basis matrices E and inner product operators Q relate to the matrix representation $\hat{A}$ and these relationships are explored below.

Here we consider the impact that non-Euclidean vector spaces and scalar products have on Euclidean linear operators $A \in \mathcal{R}|\mathcal{D}$, their non-Euclidean coefficient forms $\tilde{A}$, and their different possible matrix representations $\hat{A}$. We consider two such matrix representations in the following two subsections, the Natural matrix representation $\hat{A}_N$ and the Euclidean matrix representation $\hat{A}_E$.

3.1 The Natural matrix representation of a linear operator ($\hat{A}_N$)

First, lets consider the *Natural matrix representation* of a linear operator denoted $\hat{A}_N$ of a linear operator A in terms of the basis vectors for the spaces $\mathcal{D}$ and $\mathcal{R}$ which takes the form

$$A = E_{\mathcal{R}} \hat{A} E_{\mathcal{D}}^H. \tag{16}$$

Note that if one considers a vector $x = E\tilde{x}$ to be a linear operator, then the "matrix coefficients" for the vector $\tilde{x}$ are always stored in the Natural form where $E_{\mathcal{R}} = E$ and $E_{\mathcal{D}} = \mathbb{C}$. This will have important implications when considering the storage and manipulation of function gradients as vector objects.

Given this matrix form $\hat{A}$, the vector coefficient form of the linear operator $\tilde{A}_N$ is

$$\begin{aligned} y &= E_{\mathcal{R}} \tilde{y} \\ &= Ax \\ &= (E_{\mathcal{R}} \hat{A}_N E_{\mathcal{D}}^H)(E_{\mathcal{D}} \tilde{x}) \\ &= E_{\mathcal{R}} (\hat{A}_N Q_{\mathcal{D}} \tilde{x}) \\ &= E_{\mathcal{R}} (\hat{A}_N Q_{\mathcal{D}}) \tilde{x} \\ &\Rightarrow \\ \tilde{A}_N &= \hat{A}_N Q_{\mathcal{D}} \end{aligned} \tag{17}$$

where $Q_{\mathcal{D}} = E_{\mathcal{D}}^H E_{\mathcal{D}}$ is the scalar product matrix for the space $\mathcal{D}$. Hence, we see that applying the operator A using (16) to transform the vector coefficients $\tilde{x}$ to $\tilde{y}$ involves injecting the scalar product matrix $Q_{\mathcal{D}}$ before multiplying by the Natural coefficient matrix $\hat{A}_N$. Using this notation, we differentiate between the adjoint vector coefficient operator denoted $\tilde{A}_N^H$ and the Hermitian element-wise transpose of the forward vector coefficient operator denoted $[\tilde{A}_N]^H$.

Now lets consider the form of the Natural adjoint vector coefficient linear operator using the Natural matrix shown in (16) which gives

$$\begin{aligned} v &= E_{\mathcal{D}}\tilde{v} \\ &= A^H u \\ &= (E_{\mathcal{D}}\hat{A}_N^H E_{\mathcal{R}}^H)(E_{\mathcal{R}}\tilde{u}) \\ &= E_{\mathcal{D}}(\hat{A}_N^H Q_{\mathcal{R}}\tilde{u}) \\ &= E_{\mathcal{D}}(\hat{A}_N^H Q_{\mathcal{R}})\tilde{u} \\ &\Rightarrow \\ \tilde{A}_N^H &= \hat{A}_N^H Q_{\mathcal{R}} \end{aligned} \tag{18}$$

where $Q_{\mathcal{R}} = E_{\mathcal{R}}^H E_{\mathcal{R}}$ is the scalar product matrix for the space $\mathcal{R}$. This time, the application of the adjoint requires the injection of the scalar product matrix $Q_{\mathcal{R}}$.

Here we see the definition of the adjoint vector coefficient linear operator $\tilde{A}_N^H = \hat{A}_N^H Q_{\mathcal{R}}$ is not equal to the Hermitian transpose of the forward vector coefficient linear operator $[\tilde{A}_N]^H = [Q_{\mathcal{D}}]^H[\hat{A}_N]^H$ (i.e. $[\tilde{A}_N^H] \neq [\tilde{A}_N]^H$). Here we now see the critical difference between a linear operator and a matrix when dealing with linear operators that operate on the vector coefficients of vectors with non-Euclidean basis representations.

Dumb Fact 3.1 *When writing algorithms in vector coefficient form with non-Euclidean scalar products, the adjoint non-Euclidean coefficient linear operator $\tilde{A}^H$ is not the same as the matrix conjugate transpose of the forward non-Euclidean coefficient linear operator $\tilde{A}$. In other words, using our notation, $[\tilde{A}^H] \neq [\tilde{A}]^H$ in general.*

It is easy to show that $\tilde{A}_N$ and $\tilde{A}_N^H$ given in (17) and (18) satisfy the adjoint relationship (15)

$$\begin{aligned} \left\langle \tilde{A}_N\tilde{u}, \tilde{v} \right\rangle_{\mathcal{R}} &= (\hat{A}_N Q_{\mathcal{D}}\tilde{u})^H Q_{\mathcal{R}}(\tilde{v}) \\ &= (\tilde{u}^H Q_{\mathcal{D}}\hat{A}_N^H) Q_{\mathcal{R}}(\tilde{v}) \\ &= (\tilde{u}^H) Q_{\mathcal{D}}(\hat{A}_N^H Q_{\mathcal{R}}\tilde{v}) \\ &= \left\langle \tilde{u}, \tilde{A}_N^H\tilde{v} \right\rangle_{\mathcal{D}} \square \end{aligned} \tag{19}$$

If the linear operator A is invertible such that A^{-1} exists, then the Natural inverse vector coefficient linear operator $\tilde{A}_N^{-1}$ is given by

$$\begin{aligned} y &= E_{\mathcal{D}}\tilde{y} \\ &= A^{-1}x \\ &= (E_{\mathcal{R}}\hat{A}_N E_{\mathcal{D}}^H)^{-1}(E_{\mathcal{R}}\tilde{x}) \\ &= E_{\mathcal{D}}^{-H}\hat{A}_N^{-1}(E_{\mathcal{R}}^{-1}E_{\mathcal{R}})\tilde{x} \\ &= (E_{\mathcal{D}}E_{\mathcal{D}}^{-1})E_{\mathcal{D}}^{-H}\hat{A}_N^{-1}\tilde{x} \\ &= E_{\mathcal{D}}(E_{\mathcal{D}}^{-1}E_{\mathcal{D}}^{-H})\hat{A}_N^{-1}\tilde{x} \end{aligned}$$

$$
\begin{aligned}
&= E_{\mathcal{D}}(Q_{\mathcal{D}}^{-1}\hat{A}_N^{-1})\tilde{x} \\
&\Rightarrow \\
\tilde{A}_N^{-1} &= Q_{\mathcal{D}}^{-1}\hat{A}_N^{-1}. \qquad (20)
\end{aligned}
$$

Therefore, applying the Natural inverse vector coefficient linear operator $\tilde{A}_N^{-1}$ involves applying the inverse of the matrix representation $\hat{A}_N^{-1}$ followed by applying the inverse of the scalar product matrix $Q_{\mathcal{D}}^{-1}$.

The Natural adjoint inverse vector coefficient linear operator $\tilde{A}_N^{-H} \in \mathcal{R}|\mathcal{D}$ is also easy to derive and is given by

$$
\begin{aligned}
y &= E_{\mathcal{R}}\tilde{y} \\
&= A^{-H}x \\
&= (E_{\mathcal{R}}\hat{A}_N E_{\mathcal{D}}^H)^{-H}(E_{\mathcal{D}}\tilde{x}) \\
&= E_{\mathcal{R}}^{-H}\hat{A}_N^{-H}E_{\mathcal{D}}^{-1}E_{\mathcal{D}}\tilde{x} \\
&= (E_{\mathcal{R}}E_{\mathcal{R}}^{-1})E_{\mathcal{R}}^{-H}\hat{A}_N^{-H}\tilde{x} \\
&= E_{\mathcal{R}}(E_{\mathcal{R}}^{-1}E_{\mathcal{R}}^{-H})\hat{A}_N^{-H}\tilde{x} \\
&= E_{\mathcal{R}}(Q_{\mathcal{R}}^{-1}\hat{A}_N^{-H}\tilde{x}) \\
&= E_{\mathcal{R}}(Q_{\mathcal{R}}^{-1}\hat{A}_N^{-H})\tilde{x} \\
&\Rightarrow \\
\tilde{A}_N^{-H} &= Q_{\mathcal{R}}^{-1}\hat{A}_N^{-H}. \qquad (21)
\end{aligned}
$$

Therefore, applying the adjoint inverse of the natural coefficient representation of linear operator to the vector coefficients involves applying the inverse of the scalar product matrix $Q_{\mathcal{R}}^{-1}$.

Here we see that the Natural inverse vector coefficient forward and adjoint linear operators $\tilde{A}_N^{-1}$ and $\tilde{A}_N^{-H}$, respectively, actually are to the simple matrix inverses of the Natural vector coefficient forward and adjoint linear operators $\tilde{A}_N$ and $\tilde{A}_N^H$, respectively.

Dumb Fact 3.2 *The Natural inverse coefficient linear operator $\tilde{A}_N^{-1}$ actually is the same as the matrix inverse of the Natural forward vector coefficient linear operator $\tilde{A}_N$. In other words, using matrix element notation, $[\tilde{A}_N^{-1}] = [\tilde{A}_N]^{-1}$.*

3.2 The Euclidean matrix representation of a linear operator ($\hat{A}_E$)

Now consider another matrix representation of a linear operator, denoted here as the *Euclidean* representation, which defines forward vector coefficient operator $\tilde{A}_E$ as the matrix representation $\hat{A}_E$ as

$$
\begin{aligned}
\tilde{y} &= \tilde{A}_E\tilde{x} = \hat{A}_E\tilde{x} \\
&\Rightarrow \\
\tilde{A}_E &= \hat{A}_E \qquad (22)
\end{aligned}
$$

where $x = E_{\mathcal{D}}\tilde{x}$ and $y = E_{\mathcal{R}}\tilde{y}$. This representation is quite common in many different codes and makes good sense in many cases. This is the assumed form of the forward linear operator by Heinkenschloss & Vicente [2] for instance.

Dumb Fact 3.3 *The Euclidean form of the forward vector coefficient linear operator $\tilde{A}_E$ is (by definition) invariant to the selection of the basis representation.*

Given the matrix representation of the Euclidean forward vector coefficient linear operator $\tilde{A}_E = \hat{A}_E$ in (22) one can derive the Euclidean adjoint vector coefficient operator $\tilde{A}_E^H$ from the adjoint relationship as

$$\begin{aligned}
\left\langle \tilde{A}_E\tilde{u},\tilde{v}\right\rangle_{\mathcal{R}} &= (\hat{A}_E\tilde{u})^H Q_{\mathcal{R}}(\tilde{v}) \\
&= (\tilde{u}^H\hat{A}_E^H)Q_{\mathcal{R}}(\tilde{v}) \\
&= \tilde{u}^H(Q_{\mathcal{D}}Q_{\mathcal{D}}^{-1})\hat{A}_E^H Q_{\mathcal{R}}\tilde{v} \\
&= (\tilde{u}^H)Q_{\mathcal{D}}(Q_{\mathcal{D}}^{-1}\hat{A}_E^H Q_{\mathcal{R}}\tilde{v}) \\
&= \left\langle \tilde{u},\tilde{A}_E^H\tilde{v}\right\rangle_{\mathcal{D}} \\
&\Rightarrow \\
\tilde{A}_E^H &= Q_{\mathcal{D}}^{-1}\hat{A}_E^H Q_{\mathcal{R}} \qquad (23)
\end{aligned}$$

From (23) we can see that applying the adjoint in this case requires applying the inverse of the scalar product matrix $Q_{\mathcal{D}}^{-1}$.

From (22) or (23), one can derive the exact representation of the operator A that is consistent with this matrix representation. First, from (22) we see that

$$\begin{aligned}
y &= E_{\mathcal{R}}\tilde{y} \\
&= E_{\mathcal{R}}(\hat{A}_E\tilde{x}) \\
&= E_{\mathcal{R}}\hat{A}_E(E_{\mathcal{D}}^{-1}E_{\mathcal{D}})\tilde{x} \\
&= (E_{\mathcal{R}}\hat{A}_E E_{\mathcal{D}}^{-1})(E_{\mathcal{D}}\tilde{x}) \\
&= A_E x \\
&\Rightarrow \\
A_E &= E_{\mathcal{R}}\hat{A}_E E_{\mathcal{D}}^{-1}. \qquad (24)
\end{aligned}$$

We can also derive the Euclidean representation of A_E from (23) as

$$\begin{aligned}
y &= E_{\mathcal{D}}\tilde{y} \\
&= E_{\mathcal{D}}(Q_{\mathcal{D}}^{-1}\hat{A}_E^H Q_{\mathcal{R}}\tilde{x}) \\
&= E_{\mathcal{D}}(E_{\mathcal{D}}^H E_{\mathcal{D}})^{-1}\hat{A}_E^H(E_{\mathcal{R}}^H E_{\mathcal{R}})\tilde{x} \\
&= E_{\mathcal{D}}(E_{\mathcal{D}}^{-1}E_{\mathcal{D}}^{-H})\hat{A}_E^H(E_{\mathcal{R}}^H E_{\mathcal{R}})\tilde{x} \\
&= (E_{\mathcal{D}}E_{\mathcal{D}}^{-1})(E_{\mathcal{D}}^{-H}\hat{A}_E^H E_{\mathcal{R}}^H)(E_{\mathcal{R}}\tilde{x})
\end{aligned}$$

$$
\begin{aligned}
&= A_E^H x \\
&\Rightarrow \\
A_E^H &= E_{\mathcal{D}}^{-H} \hat{A}_E^H E_{\mathcal{R}}^H \\
&\Rightarrow \\
A_E &= E_{\mathcal{R}} \hat{A}_E E_{\mathcal{D}}^{-1}.
\end{aligned} \tag{25}
$$

Note that we already know that A_E in (24) and (25) satisfies the adjoint relationship, since (23) was derived from the adjoint relationship.

Given the Euclidean form A in (24), the action of the Euclidean inverse linear operator A^{-1} (should it exist) in the operation $y = A^{-1}x$ is given by

$$
\begin{aligned}
y &= E_{\mathcal{D}} \tilde{y} \\
&= A^{-1} x \\
&= (E_{\mathcal{R}} \hat{A}_E E_{\mathcal{D}}^{-1})^{-1} (E_{\mathcal{R}} \tilde{x}) \\
&= E_{\mathcal{D}} \hat{A}_E^{-1} (E_{\mathcal{R}}^{-1} E_{\mathcal{R}}) \tilde{x} \\
&= E_{\mathcal{D}} (\hat{A}_E^{-1} \tilde{x}) \\
&\Rightarrow \\
\tilde{A}_E^{-1} &= \hat{A}_E^{-1}
\end{aligned} \tag{26}
$$

Likewise, the action of the adjoint inverse linear operator A^{-H} of the form $y = A^{-H}x$ is also easy to derive and is given by

$$
\begin{aligned}
y &= E_{\mathcal{R}} \tilde{y} \\
&= A^{-H} x \\
&= (E_{\mathcal{R}} \hat{A}_E E_{\mathcal{D}}^{-1})^{-H} (E_{\mathcal{D}} \tilde{x}) \\
&= E_{\mathcal{R}}^{-H} \hat{A}_E^{-H} E_{\mathcal{D}}^H E_{\mathcal{D}} \tilde{x} \\
&= (E_{\mathcal{R}} E_{\mathcal{R}}^{-1}) E_{\mathcal{R}}^{-H} \hat{A}_E^{-H} (E_{\mathcal{D}}^H E_{\mathcal{D}}) \tilde{x} \\
&= E_{\mathcal{R}} (Q_{\mathcal{R}}^{-1} \hat{A}_E^{-H} Q_{\mathcal{D}} \tilde{x}) \\
&\Rightarrow \\
\tilde{A}_E^{-H} &= Q_{\mathcal{R}}^{-1} \hat{A}_E^{-H} Q_{\mathcal{D}}
\end{aligned} \tag{27}
$$

Dumb Fact 3.4 *The Euclidean form of the adjoint inverse vector coefficient linear operator $\tilde{A}_E^{-H}$ is <u>not</u> the same as the element-wise Hermitian transpose of the Euclidean form of the inverse coefficient linear operator $[\tilde{A}_E^{-1}]^H$. In other words, $[\tilde{A}_E^{-H}] \neq [\tilde{A}_E^{-1}]^H$.*

3.3 Converting between Natural and Euclidean matrix representations

In this section, we use the results from the prior sections to succinctly define the relationship and the conversions between the Natural and the Euclidean matrix representations of a linear operator. The various vector coefficient linear operators defined in the previous sections are given in Table 1.

Natural forward vector coefficient linear operator:	$\tilde{A}_N$	=	$\hat{A}_N Q_D$
Natural adjoint vector coefficient linear operator:	$\tilde{A}_N^H$	=	$\hat{A}_N^H Q_R$
Natural inverse vector coefficient linear operator:	$\tilde{A}_N^{-1}$	=	$Q_D^{-1}\hat{A}_N^{-1}$
Natural adjoint inverse vector coefficient linear operator:	$\tilde{A}_N^{-H}$	=	$Q_R^{-1}\hat{A}_N^{-H}$
Euclidean forward vector coefficient linear operator:	$\tilde{A}_E$	=	$\hat{A}_E$
Euclidean adjoint vector coefficient linear operator:	$\tilde{A}_E^H$	=	$Q_D^{-1}\hat{A}_E^H Q_R$
Euclidean inverse vector coefficient linear operator:	$\tilde{A}_E^{-1}$	=	$\hat{A}_E^{-1}$
Euclidean adjoint inverse vector coefficient linear operator:	$\tilde{A}_E^{-H}$	=	$Q_R^{-1}\hat{A}_E^{-H} Q_D$

Table 1. Summary of the definitions of vector coefficient linear operators $\tilde{A}$, $\tilde{A}^H$, $\tilde{A}^{-1}$, and $\tilde{A}^{-H}$ for the Natural and the Euclidean matrix representations $\hat{A}_N$ and $\hat{A}_E$.

Comparing the different equivalent vector coefficient forms for the Natural and the Euclidean matrix representations shown in Table 1, it is clear how to convert back and forth between any two matrix representations. Converting between the two matrix representations is given below.

- Converting from the Euclidean to the Natural form: $\hat{A}_N = \tilde{A}_E Q_D^{-1}$
- Converting from the Natural to the Euclidean form: $\hat{A}_E = \tilde{A}_N Q_D$

Therefore, converting between the Natural and the Euclidean matrix representations of a linear operator requires either applying the domain space's inner product operator Q_D or its inverse Q_D^{-1}. This type of transformation will become important later when discussing the impact of non-Euclidean basis representations on the derivatives of model functions in Section ???.

4 Impact of non-Euclidean scalar products on derivative representations

Here we describe how to correctly compute and/or apply the derivative of a multi-variable (vector) function so as to be consistent with the function's domain and range spaces when non-Euclidean basis representations are being used. We will see that these issues are closely related to the discussion of different matrix representations in Section 3.

Here, we will deal with real-valued vector spaces denoted with $\mathbf{R}^n$. The reason we do this is that while derivatives for complex-valued functions are well defined, their use in optimization and other types of numerical algorithms can be a little tricky and therefore we stick with real-valued functions here to avoid trouble.

We now consider multi-variable scalar functions and multi-variable vector functions in the next two subsections.

WARNING: In the derivative discussion below, the usage the space notation $\mathcal{X}$ is incorrect and should be replaced with $\mathbf{R}^n$ in many cases and visa versa.

4.1 Derivatives of multi-variable scalar functions

Consider the scalar-valued function $f(x)$ in Euclidean space

$$x \in \mathbf{R}^n \rightarrow f \in \mathbf{R}.$$

The definition of the first derivative of this function comes from the first-order variation

$$\delta f = \frac{\partial f}{\partial x}\delta x.$$

Therefore, the derivative $\partial f/\partial x$ first and foremost is a linear operator that when applied to some variation in x of δx gives the resulting variation δf in the function f (to first order). For scalar-valued functions, it is common to define the *gradient* of the function which is defined as $\nabla f = \partial f/\partial x^T \in \mathbf{R}^n$ and is usually represented as a vector in the space $\mathbf{R}^n$ and this gives

$$\delta f = \nabla f^T \delta x.$$

Let the coefficients of the gradient vector be denoted as $\tilde{\nabla} f \in \mathcal{X}$ such that $\nabla f = E\tilde{\nabla} f$, where E is the basis for the space $\mathcal{X} \subseteq \mathbf{R}^n$ and we consider the coefficient vector $\tilde{x} \in \mathcal{X}$.

Now consider an implementation of the function $f(x)$ that takes in the coefficients $\tilde{x} \in \mathcal{X}$ and returns f as

$$\tilde{x} \in \mathcal{X} \rightarrow g \in \mathbf{R}.$$

where

$$f(x) = g(\tilde{x}).$$

The function g is what would be directly implemented in a computer code in many cases. Since $\tilde{x} = E^{-1}x$, we see that

$$\frac{\partial f}{\partial x} = \frac{\partial g}{\partial \tilde{x}}\frac{\partial \tilde{x}}{\partial x} = \frac{\partial g}{\partial \tilde{x}}E^{-1}$$

which gives

$$\nabla f = E^{-T}\nabla g. \tag{28}$$

Equating (28) to $\nabla f = E\tilde{\nabla} f$ and performing some manipulation we see that

$$\begin{aligned} \nabla f &= E\tilde{\nabla} f \\ &= E^{-T}\nabla g \\ &\\ &\Rightarrow \\ &\\ \tilde{\nabla} f &= E^{-1}E^{-T}\nabla g \\ &= (E^T E)^{-1}\nabla g \\ &= Q^{-1}\nabla g \end{aligned} \tag{29}$$

where $Q = E^T E$. Therefore, to compute the coefficients $\tilde{\nabla} f$ for the gradient vector ∇f given the gradient ∇g for the function $g(\tilde{x})$, one must apply the inverse of the scalar product matrix Q^{-1} as shown in (29). In some codes, it is actually just as natural to directly compute $\tilde{\nabla} f$ as it is to compute $g(\tilde{x})$ and therefore there is not need to apply Q^{-1} as shown in (29).

Note that this definition of the gradient results in the total variation inner product

$$\delta f = \nabla f^T \delta x = (\tilde{\nabla} f)^T Q(\tilde{\delta x}) = (Q^{-1}\nabla g)^T Q(\tilde{\delta x}) = \nabla g^T (Q^{-1}Q)\tilde{\delta x} = \nabla g^T \tilde{\delta x}$$

which is nothing more than the simple dot product involving arrays of data that are directly stored and manipulated in the computer. This is the first case that we will see of a *scaling invariant* computation where the gradient's scalar product with the unknown variation $\tilde{\delta x}$ is independent of the choice of the basis (scaling invariance is seen by many as a very desirable algorithmic feature [??]). In the case where Q^{-1} must be applied just to have it removed again, it would be more efficient to implement the gradient ∇f^T as a linear operator $\nabla f^T = \partial f/\partial x$ instead of as a vector in order to avoid having to apply the inverse Q^{-1} just to remove its effect later using Q in the scalar product. This scale invariance also means that the total variation δf can be approximated with directional finite differences on the underlying function $g(\tilde{x})$ without any concern for what basis representation or inner product is used.

The vector form of the gradient $\nabla f \in \mathbf{R}^n$ (storing $\tilde{\nabla} f \in \mathcal{S}$) is critical in many types of numerical algorithms since it gets assigned to other vector objects and gets passed to linear operators (i.e. it becomes the right-hand side for a linear system). Therefore implementations have little choice but to implement gradients a vector objects.

Note that the vector representation $\nabla f = E\tilde{\nabla} f$, where $\tilde{\nabla} f = Q^{-1}\nabla g$, is equivalent to the Natural matrix representation of the linear operator $\nabla f \in \mathbf{R}^n|\mathbf{R}$. The transformation of $\tilde{\nabla} f = Q^{-1}\nabla g$ in (29) is nothing more than the transformation form the Euclidean to the Natural matrix representation shown in Section 3.3 where $\hat{A}_N = \tilde{\nabla} f^T$, $\hat{A}_E = \nabla g^T$, and $Q_D = Q$.

While the selection of the basis and inner product has no impact on the total variation $\delta f = \nabla f^T \delta x$, it has a tremendous impact on the inner product $\nabla f^T \nabla f$ shown as

$$\nabla f^T \nabla f = (\nabla \tilde{g}^T Q^{-T} E^T)(E Q^{-1} \nabla \tilde{g}) = \nabla \tilde{g}^T Q^{-1} Q Q^{-1} \nabla \tilde{g} = \nabla \tilde{g}^T Q^{-1} \nabla \tilde{g}. \tag{30}$$

This inner product $\nabla f^T \nabla f$ is used in many different types of algorithms and therefore these algorithms are strongly influenced by the definition of the inner product.

ToDo: Derive and describe the impact of the scalar product on the Hessian matrix for $f(x)$. I do not know what this is exactly but I need to derive this so that I can determine that the Newton step for the minimization algorithm is not effected! In think the Hessian operator is $\nabla^2 f = Q^{-1}\nabla^2 g Q^{-1}$ but we need to verify this for sure.

4.2 Derivatives of multi-variable vector functions

We now consider the extension of the above discussion of scalar-valued functions to vector-valued function $f(x)$ defined in the Euclidean space of the form

$$x \in \mathbf{R}^n \rightarrow f \in \mathbf{R}^m.$$

Again, many different algorithms consider the first-order variation

$$\delta f = \frac{\partial f}{\partial x} \delta x.$$

In this notation, $\partial f / \partial x$ is a linear operator that maps vectors from $\delta x \in \mathbf{R}^n$ to $\delta f \in \mathbf{R}^m$.

The vectors take the form $x = E_x \tilde{x}$, $f = E_f \tilde{f}$, $\delta x = E_x \tilde{\delta x}$ and $\delta f = E_f \tilde{\delta f}$ where $\tilde{x}$, $\tilde{f}$, $\tilde{\delta x}$, and $\tilde{\delta f}$ are the coefficient vectors that would typically be directly stored and manipulated in a computer program.

Now consider the case where function $f(x)$ is implemented in coefficient form through the function

$$\tilde{x} \in \mathcal{X} \rightarrow g \in \mathcal{F}.$$

where

$$f(x) = E_f g(\tilde{x}).$$

The function $g(\tilde{x})$ is what would typically be implemented in a computer code and the matrix $\partial g/\partial \tilde{x}$ could be efficiently and simply computed using automatic differentiation (AD) [???] for example. The full forward linear operator would then be

$$\frac{\partial f}{\partial x} = E_f \frac{\partial g}{\partial \tilde{x}} \frac{\partial \tilde{x}}{\partial x} = E_f \frac{\partial g}{\partial \tilde{x}} E_x^{-1} \tag{31}$$

which takes the same form as the Euclidean representation of the linear operator described in Section 3.2. This operator $A = \partial f/\partial x$ can either be formed and stored using some matrix representation or can be applied implicitly.

There are one of two choices for how to actually implement the operator $A = \partial f/\partial x$ using a matrix representation. The first option is to just explicitly store the matrix $\partial g/\partial \tilde{x}$ that would be directly computed from the function $g(\tilde{x})$ using AD for instance. The forward operation application $y = (\partial f/\partial x)x$ would then be applied in vector coefficient form as

$$\tilde{y} = \frac{\partial g}{\partial \tilde{x}} \tilde{x}.$$

This Euclidean form, however, would then require that the adjoint be implemented as

$$y = \frac{\partial f}{\partial x}^T x \;\Rightarrow\; \tilde{y} = {Q_x}^{-1} \frac{\partial g}{\partial \tilde{x}}^T Q_f \tilde{x} \tag{32}$$

as shown in (23), which requires the application of the inverse of the scalar product matrix ${Q_x}^{-1}$ with each application of the adjoint.

The other option for a matrix representation is to compute and store $\hat{A} = (\partial g/\partial \tilde{x}){Q_x}^{-1}$ and this gives the Natural representation

$$\frac{\partial f}{\partial x} = E_f \frac{\partial g}{\partial \tilde{x}} E_x^{-1} = E_f \frac{\partial g}{\partial \tilde{x}} E_x^{-1} (E_x^{-T} E_x^T) = E_f \frac{\partial g}{\partial \tilde{x}} (E_x^T E_x)^{-1} E_x^T = E_f \hat{A} E_x^T. \tag{33}$$

Note that forming the product $(\partial g/\partial \tilde{x}){Q_x}^{-1}$ may be very expensive to do in practice and can destroy the sparsity of $\partial g/\partial \tilde{x}$. Note that this is equivalent to the vector representation of the gradient ∇f described in Section 4.1.

ToDo: Look at the practical issues of computing and directly storing $(\partial g/\partial \tilde{x}){Q_x}^{-1}$ for a few different discretizations. Is there special structure that makes this easy to compute a sparse matrix? How do people actually do this in practice?

5 Impact of non-Euclidean scalar products on various numerical algorithms

Here we discuss the bread and butter of the impact of scalar products in how they affect numerical algorithms that we develop and implement. The approach taken here is to first start with the algorithms stated in Euclidean form without regard to issues of scalar products. This is fine as long as we recognize that the vectors, x for instance, that we are dealing with will eventually be substituted for there basis and coefficient form $x = E\tilde{x}$ from which we do manipulations. What we will try to do is to see how the expressions in the algorithm change and we will try to perform the manipulations so that we are left with the only the vector coefficients (i.e. $\tilde{x}$), scalar product matrices (i.e. Q_x), and linear operators. We will also try to remove any explicit dependence on the exact form of the basis representation (i.e. the basis E_x should not appear in any final form of the coefficient expressions).

The general approach is summarized as:

1. State the algorithm in Euclidean form using vectors with respect to a Euclidean basis (e.g. x) with simple dot products (e.g. $x^H y$).
2. Substitute the basis representations for all vectors (e.g. $x = E\tilde{x}$) in all expressions.
3. Manipulate the expressions and try to decompose all operations into coefficient form involving only the vector coefficients (e.g. $\tilde{x}$), scalar product matrices (e.g. Q_x), and other model-defined linear operators if needed.
4. Go back and investigate how to implement the remaining linear operators (especially those that are model function derivative operators like Jacobians and Hessians).

To demonstrate the process, consider the Euclidean form of the inner CG iteration

$$\begin{aligned}
\rho_k &= r^H r, \\
p &= r + \frac{\rho_k}{\rho_{k-1}} p, \\
q &= Ap, \\
\alpha &= \frac{\rho_k}{p^H q}, \\
x &= x + \alpha p, \\
r &= r - \alpha q.
\end{aligned}$$

In this algorithm, the linear operator $A \in \mathcal{S}|\mathcal{S}$ is symmetric so we are dealing with just one vector space $\mathcal{S}$ with scalar product Q. Let $E \in \mathbb{C}^{n\times n}$ be any basis representation such that $Q = E^H E$. Substituting $r = E\tilde{r}$, $p = E\tilde{p}$, $q = E\tilde{q}$, and $x = E\tilde{x}$ in the above inner loop expressions yields

$$\begin{aligned}
\rho_k &= \tilde{r}^H E^H E \tilde{r}, \\
E\tilde{p} &= E\tilde{r} + \frac{\rho_k}{\rho_{k-1}} E\tilde{p}, \\
E\tilde{q} &= (E\tilde{A}E^{-1}) E\tilde{p},
\end{aligned}$$

$$
\begin{aligned}
\alpha &= \frac{\rho_k}{\tilde{p}^H E^H E \tilde{q}}, \\
E\tilde{x} &= E\tilde{x} + \alpha E\tilde{p}, \\
E\tilde{r} &= E\tilde{r} - \alpha E\tilde{q},
\end{aligned}
$$

$\Rightarrow$

$$
\begin{aligned}
\rho_k &= \tilde{r}^H Q \tilde{r}, \\
E\tilde{p} &= E(\tilde{r} + \frac{\rho_k}{\rho_{k-1}}\tilde{p}), \\
E\tilde{q} &= E(\tilde{A}\tilde{p}), \\
\alpha &= \frac{\rho_k}{\tilde{p}^H Q \tilde{q}}, \\
E\tilde{x} &= E(\tilde{x} + \alpha\tilde{p}), \\
E\tilde{r} &= E(\tilde{r} - \alpha\tilde{q}),
\end{aligned}
$$

$\Rightarrow$

$$
\begin{aligned}
\rho_k &= \langle \tilde{r}, \tilde{r} \rangle, \\
\tilde{p} &= \tilde{r} + \frac{\rho_k}{\rho_{k-1}}\tilde{p}, \\
\tilde{q} &= \tilde{A}\tilde{p}, \\
\alpha &= \frac{\rho_k}{\langle \tilde{p}, \tilde{q} \rangle}, \\
\tilde{x} &= \tilde{x} + \alpha\tilde{p}, \\
\tilde{r} &= \tilde{r} - \alpha\tilde{q}.
\end{aligned}
$$

As seen in the above example, if after this transformation we can manipulate the expressions such that the coefficient forms do not explicitly involve the basis matrix E but instead only involve the scalar product matrix $Q = E^H E$ and the non-Euclidean coefficient forms of the linear operators, then we have succeeded in deriving a general form of the algorithm that will work for all non-Euclidean vector spaces. The one lingering issue is what is meant by the vector coefficient linear operator $\tilde{A}$ used in $\tilde{q} = \tilde{A}\tilde{p}$? Can we just simply assume the Euclidean form such that the same matrix representation (or operator application code) can be used as $\tilde{A} = \hat{A}$? Who makes this decision?

ToDo: Do a careful analysis of this linear transformation of variables for linear CG. Consider the classic form that uses a symmetric preconditioner as the inner product and make that consistent with a linear transformation of variables. We may have to go back to first principles of CG to do this.

It is critical to note that when the selection of the scalar products affects an algorithm then a good selection for the scalar products can positively impact the performance of the algorithm. The dramatic improvement in the performance of various numerical algorithms that is possible with the proper selection of scalar products is documented in [???] and [???]. Many numerical algorithms applied to applications that are based on discretizations of PDEs can show mesh-independent scaling

when using the proper scalar products for instance [???].

ToDo: We need to dig up these references for ourselves and reproduce some of the finding.

5.1 Newton methods

The first set of methods that we will consider are Newton methods [???]. In their most basic form, a Newton method seeks to solve a set of multi-variable nonlinear equations

$$f(x) = 0$$

where $x \in \mathbf{R}^n$ and

$$x \in \mathbf{R}^n \rightarrow f \in \mathbf{R}^n$$

is a vector function of the form described in Section 4.2 where $f(x) = E_f g(\tilde{x})$ and $g(\tilde{x})$ is what is implemented in the computer. The undampened Newton method seeks to improve the estimate of the solution x_k by solving the linear system

$$\frac{\partial f}{\partial x} d = -f(x_k) \tag{34}$$

and then update the estimate using

$$x_{k+1} = x_k + d. \tag{35}$$

It can be shown than when x_0 is sufficiently close to a solution x^* such that $f(x^*) = 0$, and if $\partial f / \partial x$ is non-singular, then the iterates $x_1, x_2, \ldots, x_k, x_{k+1}$ converge quadratically with

$$||x_{k+1} - x^*|| < C||x_k - x^*||^2$$

for some constant $C \in \mathbf{R}$ [???]. In a real Newton method, some type of modification is generally applied to the step computation in (34) and/or the update in (35) in order to insure convergence from remote starting points x_0.

We now consider the impact that non-Euclidean basis representations and scalar products have on two forms of the Newton step computation: exact and inexact.

5.1.1 Exact Newton methods

In an exact Newton method, the Newton system in (34) is solved to a high precision. Now let's consider the impact that substituting non-Euclidean basis representations have on the Newton method.

The basis representations are $x = E_x\tilde{x}$ and $f = E_f\tilde{f}$ for the spaces $\mathcal{X} \in \mathbf{R}^n$ and $\mathcal{F} \in \mathbf{R}^n$. Now, let us assume the Euclidean representation is used for $\partial f/\partial x$ which gives the coefficient form of (34) as

$$\frac{\partial g}{\partial \tilde{x}}\tilde{d} = -g. \tag{36}$$

We then substitute $\tilde{d}$ into the update in (35) which is

$$\tilde{x}_{k+1} = \tilde{x}_k + \tilde{d}. \tag{37}$$

Comparing (34)–(35) with (36)–(37), it is clear that the choice of the basis vectors for the spaces $\mathcal{X}$ or $\mathcal{F}$ have no impact on the Newton steps that are generated. This *invariance* property of Newton's method is one of its greatest strengths. However, solving the Newton system exactly can be very expensive and taking full steps can cause the algorithm to diverge and therefore modifications to handle these issues are considered later. First, however, the inexact computation of the Newton step is discussed in the next subsection.

5.1.2 Inexact Newton methods

In an inexact Newton method, the linear system in (34) is not solved exactly, but instead is only solved to a tolerance of

$$\frac{||\frac{\partial f}{\partial x}d + f_k||_{\mathcal{F}}}{||f_k||_{\mathcal{F}}} \leq \eta \tag{38}$$

where $\eta \in \mathbf{R}$ is known as the forcing term and typically is selected such that $\eta \propto ||f_k||_{\mathcal{F}}$ to ensure quadratic convergence. The coefficient representation of (38) after squaring takes the form

$$\frac{\left(\frac{\partial g}{\partial \tilde{x}}\tilde{d} + g_k\right)^T Q_{\mathcal{F}} \left(\frac{\partial g}{\partial \tilde{x}}\tilde{d} + g_k\right)}{g_k^T Q_{\mathcal{F}}\, g_k} \leq \eta^2. \tag{39}$$

From (39) we see that the selection of the scalar product matrix $Q_{\mathcal{F}}$ that defines the norm $||.||_{\mathcal{F}}$ (as defined in (7)) can have a large impact on quality the inexact newton step computation. However, assuming the Euclidean form of the forward operator is used as in (36), then the selection of the scalar product for the space $\mathcal{X}$ has no impact on the computed Newton step. Such a invariant computation is said to be *affine invariant* [???].

5.2 Minimization, merit functions and globalization methods

Let's consider the minimization of a multi-variable scalar function

$$\min \qquad f(x) \tag{40}$$

where $f(x) \in \mathbf{R}^n \rightarrow \mathbf{R}$ of the form described in Section 4.1 where $f(x) = g(\tilde{x})$ and $g(\tilde{x}) \in x \rightarrow \mathbf{R}$ is what is actually implemented in a computer program.

As stated in Section 4.1, the vector coefficient for the gradient ∇f, which takes the form $\tilde{\nabla} f = {Q_x}^{-1} \nabla g$, is affected by the definition of the basis E_x but total variation

$$\delta f = \nabla f^T d = ({Q_x}^{-1} \nabla g)^T Q_x(\tilde{d}) = \nabla g^T(\tilde{d}) \tag{41}$$

is not affected, where $d = E_x \tilde{d} \in \mathbf{R}^n$, $\tilde{d} \in x$ is some search direction.

One of the most basic requirements for many minimization algorithms is the descent requirement which can be stated as

$$\nabla f^T d < 0 \tag{42}$$

for $\nabla f \neq 0$.

Consider the steepest-descent direction $d = -\gamma \nabla f$ where $\gamma > 0$ is some constant. With a Euclidean basis, the coefficient vector for this direction takes the form $\tilde{d} = -\gamma \nabla g$. However, when a non-Euclidean basis is used, the coefficient vector for the the steepest-descent direction is

$$\tilde{d} = -\gamma {Q_x}^{-1} \nabla g.$$

Therefore, the choice of the scalar product can have a dramatic impact on the steepest-descent direction. The descent property for the steepest-descent direction then becomes

$$\nabla f^T d = (\nabla g^T {Q_x}^{-1}) Q_x (-\gamma {Q_x}^{-1} \nabla g) = -\gamma \nabla g^T {Q_x}^{-1} \nabla g < 0.$$

for $\nabla g \neq 0$. Therefore, the descent property for the steepest-descent direction is changed even though the scalar product definition itself is not.

Another selection for the step direction takes the form $d = -B^{-1} \nabla f$ where B is some approximation for the Hessian of $f(x)$. Since ∇f changes with a non-Euclidean basis, so will this search direction. The choice of B for variable-metric quasi-Newton methods will be addressed in Section 5.4.

Descent alone is not sufficient to guarantee convergence. Instead, more stringent conditions must be met. One such set of conditions include a sufficient decrease condition

$$f(x_k + \alpha d) \leq f_k + c_1 \alpha (\nabla f_k)^T d \tag{43}$$

(often know as the *Armijo condition*), and a curvature condition

$$(\nabla f(x_k + \alpha d))^T d \leq c_2 (\nabla f_k)^T d \tag{44}$$

where $0 < c_1 < c_2 < 1$. Together, (43)–(44) are known as the *Wolfe conditions* [???].

Now let's consider the vector coefficient form of the conditions in (43)–(44) for non-Euclidean basis' which from (41) become

$$g(\tilde{x}_k + \alpha\tilde{d}) \leq g_k + c_1\alpha(\nabla g_k)^T\tilde{d} \tag{45}$$

and

$$(\nabla g(\tilde{x}_k + \alpha\tilde{d}))^T\tilde{d} \leq c_2(\nabla g_k)^T\tilde{d}. \tag{46}$$

It is clear from (45)–(46) that even through the selection of the scalar product defined by Q_x affects the steepest-descent direction, for instance, it does not actually affect the Wolf conditions for a general direction $\tilde{d}$. The computation of the direction $\tilde{d}$ can, however, be impacted by the choice of the scalar product as described above. What this means is that the Wolfe conditions are invariant to the selection of the basis for the space $\mathcal{X}$. Again, invariance with respect to the selection of the basis is consider a very attractive property for numerical algorithms [???]. It is those parts of a minimization algorithm (like the step computation) that leads to seek out better scalar products.

5.3 Least-squares merit functions

Here we consider the impact that non-Euclidean scalar products have on standard least-square merit functions of the form

$$\phi(x) = f(x)^T f(x) \tag{47}$$

where $f(x)$ is a multi-variable vector-valued function of the form described in Section 4.2 which is implemented in terms of $g(\tilde{x})$ where $f(x) = E_f g(\tilde{x})$. The least-squares function defined in (47) is used in a variety of contexts from globalization methods for nonlinear equations $f(x) = 0$ [???] to data fitting optimization methods [???].

The gradient $\nabla\phi \in \mathbf{R}^n$ of $\phi(x)$ defined in (47) is given by

$$\nabla\phi = \frac{\partial f}{\partial x}^T f. \tag{48}$$

When $\partial f/\partial x$ is represented in Euclidean form as shown in (31), the coefficient form of the adjoint Jacobian-vector product in (48), shown in (32), is given by

$$\tilde{\nabla}\phi = {Q_x}^{-1}\frac{\partial g}{\partial \tilde{x}}^T Q_{\mathcal{F}}\, g. \tag{49}$$

In (49) we see that the gradient direction for the least-squares merit function in (47) is impacted by both the scalar product matrices Q_x and $Q_{\mathcal{F}}$.

5.4 Variable metric quasi-Newton methods

Non-Euclidean scalar products can dramatically improve the performance of optimization methods that use variable-metric quasi-Newton methods [???]. Here we will consider a popular form of variable-metric approximation the BFGS formula [???] which is defined as

$$B_+ = B - \frac{(Bs)(Bs)^T}{s^T Bs} + \frac{yy^T}{y^T s}$$

where B is the current approximation to the Hessian $\nabla^2 f$ and B_+ is the updated approximation.

Generally, the update vectors are defined as $y = \nabla f_k - \nabla f_{k-1}$ and $s = x_k - x_{k-1}$ but the analysis here is independent of the actual choices for these vectors. What will be made clear here is the impact that non-Euclidean scalar products have on the various implementations of this method.

We will consider two forms of the above approximation. First, we consider an explicit implementation that directly stores the coefficients of the matrix in the Natural form (Section 3.1). Second, we consider an implicit implementation that only stores pairs of update vectors and applies the inverse implicitly. The implicit representation then leads naturally to a limited-memory implementation.

5.4.1 Explicit BFGS matrix representation

For the explicit matrix representation we will assume that B and B_+ are being stored in the Natural coefficient forms of $B = E\hat{B}E^T$ and $B_+ = E\hat{B}_+E^T$. Note that the basis matrix E is generally not given explicitly and a unique choice is not known; only the scalar product matrix $Q = E^T E$ is known. By substituting in the coefficient forms of $B = E\hat{B}E^T$, $B_+ = E\hat{B}_+E^T$, $y = E\tilde{y}$, and $s = E\tilde{s}$ into (5.4) and performing some manipulation we obtain

$$\begin{aligned}
E\hat{B}_+E^T &= E\hat{B}E^T - \frac{[(E\hat{B}E^T)(E\tilde{s})][(E\hat{B}E^T)(E\tilde{s})]^T}{(E\tilde{s})^T(E\hat{B}E^T)(E\tilde{s})} + \frac{(E\tilde{y})(E\tilde{y})^T}{(E\tilde{y})^T(E\tilde{s})} \\
&= E\hat{B}E^T - \frac{E(\hat{B}Q\tilde{s})(\hat{B}Q\tilde{s})^T E^T}{\tilde{s}^T Q(\hat{B}Q\tilde{s})} + \frac{E\tilde{y}\tilde{y}^T E^T}{\tilde{y}^T Q\tilde{s}} \\
&= E\left[\hat{B} - \frac{(\hat{B}Q\tilde{s})(\hat{B}Q\tilde{s})^T}{\tilde{s}^T Q(\hat{B}Q\tilde{s})} + \frac{\tilde{y}\tilde{y}^T}{\tilde{y}^T Q\tilde{s}}\right] E^T \\
&\Rightarrow \\
\hat{B}_+ &= \hat{B} - \frac{(\hat{B}Q\tilde{s})(\hat{B}Q\tilde{s})^T}{\tilde{s}^T Q(\hat{B}Q\tilde{s})} + \frac{\tilde{y}\tilde{y}^T}{\tilde{y}^T Q\tilde{s}}.
\end{aligned} \tag{50}$$

What (50) shows is that the Natural matrix representation of B can be updated to B_+ by using the coefficients of the vectors $\tilde{s}$ and $\tilde{y}$, the matrix coefficients $\hat{B}$ themselves, and the action of the scalar product matrix Q. Note that the final expressions for the update do not contain the basis matrix E itself since this matrix is not known in general. Also note that $\tilde{q} = \hat{B}Q\tilde{s}$ is just the coefficient vector from the output of the action of $q = Bs$ and the remaining operations involving Q which are $\tilde{s}^T Q\tilde{q}$ and $\tilde{s}^T Q\tilde{q}$ are simply applications of the scalar products $\langle s, q\rangle$ and $\langle y, y\rangle$ and therefore no direct

access the the Q operator is needed here. However, note that applying the Natural representation of B does require the ability apply Q as a linear operator and not just a scalar product.

What all this means is that code that currently implements an explicit BFGS update assuming a Euclidean basis should only need minor modifications in order to work correctly for non-Euclidean scalar products.

Note that applying the inverse of $B = E\hat{B}E^T$ as $v = B^{-1}u$ is simply a special case of (21) and is given as

$$\begin{aligned} v &= E\tilde{v} \\ &= B^{-1}u \\ &= (E\hat{B}E^T)^{-1}(E\tilde{u}) \\ &= E(Q^{-1}\hat{B}^{-1}\tilde{u}) \\ &\Rightarrow \\ \tilde{v} &= Q^{-1}\hat{B}^{-1}\tilde{u}. \end{aligned} \tag{51}$$

Therefore, applying the inverse of the natural coefficient representation of B involves applying the inverse of the scalar product matrix Q^{-1}.

Note that storing $\hat{B} \in \mathbf{R}^n|\mathbf{R}^n$ as a dense matrix requires $O(n^2)$ data with $O(n^2)$ flops to do the updates (see the update formulas from [???]), this would mean that it might be reasonable to also store $Q \in \mathbf{R}^n|\mathbf{R}^n$ as a dense matrix and then do a Cholesky factorization $Q = LL^T$. Applying the inverse Q^{-1} would then just involve doing back-solves with the Cholesky factor L^{-1} and L^{-T}.

5.4.2 Implicit BFGS matrix representation

For the implicit representation of a BFGS approximation we will consider the approximation of the inverse $H = B^{-1}$ and the update $s = H_+^{-1}y$ using the update vectors s and y which is given by the formula ([???])

$$H_+ = V^T HV + \rho ss^T \tag{52}$$

where

$$\rho = \frac{1}{y^T s}, \tag{53}$$

$$V = I - \rho ys^T. \tag{54}$$

Here we consider a limited-memory implementation (L-BFGS) [???] where m sets of update quantities $\{s_i, y_i, \rho_i\}$ are stored for the iterations $i = k-1, k-2, \ldots, k-m$ which are used to update from the initial matrix inverse approximation $H_0 = B_0^{-1}$ to give H after the m updates (see [???] for details). The implementation of the inverse Hessian-vector product $v = Hu$ is provided by a simple two-loop algorithm involving only simple vector operations like dot products, vector scalings, vector additions, and the application of the linear operator H_0. Therefore, we will go and skip ahead

and write the general non-Euclidean coefficient form of this algorithm. This simple algorithm is called the two-loop recursion [???] which is stated as

L-BFGS two-loop recursion for computing $\tilde{v} = \tilde{H}\tilde{u}$

$\tilde{q} = \tilde{u}$
for $i = k-1, \ldots, k-m$
    $\alpha_i = \rho_i \langle \tilde{s}_i, \tilde{q} \rangle$
    $\tilde{q} = \tilde{q} - \alpha_i \tilde{y}_i$
end
$\tilde{r} = \tilde{H}_0 \tilde{q}$
for $i = k-m, \ldots, k-1$
    $\beta = \rho_i \langle \tilde{y}_i, \tilde{r} \rangle$
    $\tilde{r} = \tilde{r} + (\alpha_i - \beta)\tilde{s}_i$
end
$\tilde{v} = \tilde{r}$

While it is subtle, the insertion of the general scalar products $\langle \tilde{s}_i, \tilde{q} \rangle$ and $\langle \tilde{y}_i, \tilde{r} \rangle$ can result in a dramatic improvement in the performance of minimization methods that use it and it has been shown to have mesh-independent convergence properties (i.e. the number of iterations does not increase as the mesh is refined) for some classes of PDE-constrained optimization problems [???].

5.5 Inequality constraints

Consider a simple set of simple bound inequality constraints of the form

$$a \leq x \tag{55}$$

where $x, a \in \mathcal{S}$ with basis representations $x = E\tilde{x}$ and $a = E\tilde{a}$. Inequality constraints of this form present a difficult problem for numerical algorithms using non-Euclidean basis matrices E since the inequality constraint in (55) is really a set of element-wise constraints

$$a_i \leq x_i, \text{ for } i = 1 \ldots n. \tag{56}$$

The element-wise nature of (56) means that we can not simply substitute the coefficient vector components $\tilde{x}_i$ and $\tilde{a}_i$ in for x_i and a_i. One could, however, simply substitute in the coefficient vector components and have the algorithm enforce

$$\tilde{a}_i \leq \tilde{x}_i, \text{ for } i = 1 \ldots n, \tag{57}$$

but then that may fundamentally change the meaning of these constraints and may destroy the physical utility of these constraints for the application. Although, note that in some categories of applications this type of substitution may be very reasonable. For example, in standard finite-element

discretizations of PDEs, the vector coefficients directly correspond to physical quantities such as temperature, stress, and velocity at the mesh nodes. Therefore, placing inequality constraints directly on these types of coefficients may be very reasonable even through a non-Euclidean scalar product is desirable in order to introduce mesh-dependent scaling into other parts of the algorithm. In other types of discretizations, such as those that use a spectral basis [???], there is no physical meaning for these coefficients so direct inequalities involving these types of coefficients are meaningless.

Note that imposing the inequality constraints in non-Euclidean coefficient form as in (57) is equivalent to imposing the inequalities in Euclidean form as

$$E^{-1}a \leq E^{-1}x \tag{58}$$

which is important when performing the initial transformation from the Euclidean form (i.e. the form using dot products $x^H y$) to the non-Euclidean coefficient form (i.e. using scalar products $\langle \tilde{x}, \tilde{y} \rangle$). Here, we hope that in doing the transformation of the entire algorithm that we can remove any explicit mention of the basis matrix E itself.

In cases where component-wise inequalities on vector coefficients is not meaningful, one has no choice but to form an explicit basis and to pose these constraints as general linear inequality constraints of the form

$$\tilde{b} \leq E\tilde{x}, \tag{59}$$

where $\tilde{b} = E\tilde{a}$ is a new vector directly manipulated in the software. Even if an explicit basis matrix $E \in \mathbf{R}^n | \mathbf{R}^n$ must be formed in order to preserve the meaning of the inequality constraints, there is still utility in expressing an algorithm in general non-Euclidean coefficient form since it avoids having to convert all vectors back and forth using the basis representation or having to invert the basis matrix. Also, it avoid other problems like killing the sparsity structure of the derivative matrices.

In conclusion, if it is reasonable to impose inequality constraints on the elements of the coefficient vectors themselves such as in (57), then ANAs involving inequalities with non-Euclidean scalar products can be very reasonable and straightforward to implement. When replacing the Euclidean inequalities with the vector coefficients is not meaningful, then the an explicit basis representation is required to express the constraints as general inequality constraints as in (59).

TODO: Finish editing from here!

6 Vector Coefficient Forms of Numerical Algorithms

Here we finally come back to reality. Up to this point in the discussion we have been very careful to differentiate the vector $x \in \mathbf{R}^n$ in Euclidean space from the vector coefficients $\tilde{x} \in \mathcal{X}$ in non-Euclidean space related by the equation $x = E\tilde{x}$. We have viewed algorithms in Euclidean form using the vectors x and y and simple Euclidean dot products $x^H y$ and then in non-Euclidean coefficient form using coefficient vectors $\tilde{x}$ and $\tilde{y}$ and scalar products $\langle \tilde{x}, \tilde{y} \rangle$. However, when mathematicians write numerical algorithms in coefficient form they do not typically use math accents like $\tilde{x}$ and $\tilde{A}$ or acknowledge the related Euclidean forms. Instead, they use non-accented identifiers and often the only clue that we are dealing with non-Euclidean vectors, vector spaces, and linear operators expressed in vector coefficient form is that simple dot products like $x^H y$ are replaced with $\langle x, y \rangle$. As we have show above, expressing algorithms in vector coefficient form with non-Euclidean scalar products has a dramatic impact on the definition linear operators, derivative computations, and the meaning of certain types constructs like inequality constraints. For example, we showed in Section ??? that the adjoint non-Euclidean coefficient linear operator $\tilde{A}^H$ is not the same thing as the matrix conjugate transpose of the forward non-Euclidean coefficient linear operator $\tilde{A}$ (i.e. $[\tilde{A}^H] \neq [\tilde{A}]^H$).

Dumb Fact 6.1 *When most mathematicians write a numerical algorithm using the scalar product notation $\langle x, y \rangle$, the vectors x and y are the coefficients of the vectors and all of the linear operators become non-Euclidean coefficient operators which are* ***not*** *equivalent to matrices in general!*

Using the approach outlined above, one can comfortably go between the Euclidean dot product form (i.e. $x^H y$) and the non-Euclidean scalar product form (i.e. $\langle x, y \rangle$) of most algorithms. However, when doing so one has to be careful to keep straight which form is being expressed or things can get confusing very quickly. This is where a software implementation using an abstract interface like Thyra can really help manage the complexity of implementing these types of algorithms.

7 Summary

Here we have presented an approach to looking at non-Euclidean scalar product spaces that deals in very straightforward terms using simple concepts from linear algebra. The idea is to first look at all algorithms assuming Euclidean vector spaces and explicit Euclidean coefficient vectors and then to substitute in the basis representation for non-Euclidean vector spaces. After this substitution, one then tries to manipulate the expressions to come up with the building blocks of scalar products and linear operators and only considers the explicit representation and manipulation of the coefficient vectors and never the Euclidean coefficients of the vectors themselves.

There are numerous advantages to both using well selected transformations of variables and to not applying them directly but instead only performing the transformations indirectly by injecting scalar products. Explicitly applying the needed linear transformation of variable is just not practical for large-scale problems from a number of perspectives including the difficulties in computing the needed transformation matrices, destroying the sparsity structure of large Jacobin and Hessian matrices, and making the algorithm output more difficult to interpret.

References

[1] R. A. Bartlett. Thyra linear operators and vector s: Overview of interfaces and support software for the development and interoperability of abstract numerical algorithms. Technical report SAND2007-5984, Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550, 2007.

[2] M. Heinkenschloss and L. N. Vicente. An interface between optimization and application for the numerical solution of optimal control problems. *ACM Transactions on Mathematical Software*, 25(2):157–190, June 1999.

A ToDo

- To make this type of discussion more helpful, it would be nice to have a concrete application and numerical algorithm example to work through to show the impact of all of this. This could, in fact, make a nice journal paper to show off Thyra if done well.
- ???

Sandia National Laboratories