8.3 Nonlinear Eigenvalue Problems with Orthogonality Constraints

contributed by Alan Edelman and Ross Lippert

8.3.1 Introduction

This section increases the scope of the algebraic eigenvalue problem by focusing on the geometrical properties of the eigenspaces. We discuss a template that solves variational problems defined on sets of subspaces. Areas where such problems arise include electronic structures computation, eigenvalue regularization, control theory, signal processing and graphics camera calibration. The underlying geometry also provides the numerical analyst theoretical insight into the common mathematical structure underlying eigenvalue algorithms (see [1, 7, 18]).

Suppose that one wishes to optimize a real valued function $F(Y)$ over Y such that $Y^*Y = I$, where Y^* is either transpose or Hermitian transpose as appropriate. Our template is designed as a means to optimize such an F.

One simple case is optimization over square orthogonal (or unitary) matrices such as the least squares simultaneous diagonalization of symmetric matrices (also known as INDSCAL [6]) problem described later. Another simple case is optimization over the unit sphere, as in symmetric Rayleigh quotient minimization. In between, we have rectangular $n \times p$ $(n \geq p)$ matrices Y with orthonormal columns such as the orthogonal Procrustes problem.

Furthermore, some functions F may have the symmetry property $F(Y) = F(YQ)$ for all orthogonal (unitary) Q with a specified block diagonal structure, which causes some search directions to have no effect on the value of F. For example, if A is a symmetric matrix, $n \geq p$, and $F(Y) = \mathrm{tr}(Y^*AY)$, then $F(Y) = F(YQ)$ for all $p \times p$ orthogonal Q. More generally, some functions F have the property that $F(Y) = F(YQ)$ where Q is orthogonal (unitary) with the **block diagonal** form

$$Q = \begin{bmatrix} Q_1 & 0 & \cdots & 0 \\ 0 & Q_2 & \cdots & 0 \\ .. & .. & .. & .. \\ 0 & 0 & \cdots & Q_p \end{bmatrix},$$

where the Q_i are orthogonal (unitary) matrices. More complicated problems with block diagonal orthogonal (unitary) symmetry can arise in eigenvalue regularization problems (an example of such a problem is the sample nearest Jordan block problem solved in the examples).

This chapter is organized as follows. Section 0.3.2 will discuss the basics of calling the template code. Section 0.3.3 discusses the objective functions and their derivatives for the example problems explored in this chapter. Section 0.3.4 contains sample runs of instances of the examples problems. Section 0.3.5 explains the code structure and the places where a user may wish to make modifications. Section 0.3.6 will cover some of the basic mathematics concerning the geometry of the Stiefel manifold.

8.3.2 Matlab Templates

The templates are ready for immediate use or as a departure point for generalization, e.g. problems over multiple variables with orthogonality constraints, or code optimizations. In the simplest mode, the user needs only to supply a function to minimize $F(Y)$ (and first and second derivatives, optionally) in `F.m` (in `dF.m` and `ddF.m`) and an initial guess `Y0`. The calling sequence is then a single call to `sg_min` (named in honor of Stiefel and Grassmann).

```
[fopt, yopt] =  sg_min(Y0) .
```

For example, if the function `F.m` has the form

```
function f=F(Y)
f=trace( Y' * diag(1:10) * Y * diag(1:3) );
```

we can call `sg_min(rand(10,3))` which specifies a random starting point.

We strongly recommend also providing first derivative information:

```
function df=dF(Y)
df =  2 * diag(1:10) * Y * diag(1:3);
```

The code can do finite differences, but it is very slow and problematic. Second derivative information can also be provided by the user (this is not nearly as crucial for speed as providing first derivative information, but may improve accuracy):

```
function ddf=ddF(Y,H)
ddf =  2 * diag(1:10) * H * diag(1:3);
```

A sample test call where $F(Y)$ is known to have optimal value 10 is

```
>> rand('state',0); % initialize random number generator
>> fmin = sg_min(rand(10,3))
iter    grad            F(Y)              flops         step type
0       1.877773e+01    3.132748e+01         2470       none
  invdgrad: Hessian not positive definite, CG terminating early
1       1.342879e+01    2.011465e+01       163639       Newton step
  invdgrad: Hessian not positive definite, CG terminating early
2       1.130915e+01    1.368044e+01       344198       Newton step
  invdgrad: Hessian not positive definite, CG terminating early
3       5.974819e+00    1.063045e+01       515919       Newton step
  invdgrad: max iterations reached inverting the hessian by CG
4       1.135417e+00    1.006835e+01       789520       Newton step
5       5.526359e-02    1.000009e+01      1049594       Newton step
6       5.072118e-05    1.000000e+01      1337540       Newton step
7       1.276025e-06    1.000000e+01      1762455       Newton step

fmin =
   10.0000
```

The full calling sequence to `sg_min` is

```
[fopt, yopt]=sg_min(Y0,rc,mode,metric,motion,verbose,...
                        gradtol,ftol,partition),
```

argument	description
rc	{'real', 'complex' }
mode	{'newton', 'dog', 'prcg', 'frcg' }
metric	{'flat', 'euclidean', 'canonical' }
motion	{'approximate', 'exact' }
verbose	{'verbose', 'quiet'}
ftol	first convergence tolerance
gradtol	second convergence tolerance
partition	partition of p indicating symmetries of f

TABLE 8.1: **A short list of the optional arguments for sg_min**

where `Y0` is required and the optional arguments are (see Table 1):

`rc` specifies whether the matrices will have complex entries or not. Although most of the code is insensitive to this issue, `rc` is vital for counting the dimension of the problem correctly. When omitted, `sg_min` guesses are based on whether `Y0` has nonzero imaginary part.

`mode` selects the optimization method that will be used. 'newton' selects Newton's method with a conjugate gradient based inversion of the Hessian. 'dog' selects a dog-leg step algorithm which interpolates a steepest descent and a Newton's method step. 'frcg' selects Fletcher-Reeves conjugate gradient. Lastly, 'prcg' selects Polak-Ribiere conjugate gradient which has the advantage that it does not require a very accurate Hessian (and thus, the safest of these methods if one uses a finite difference approximation to implement `ddF.m`). While 'newton' is the default, this is by no means our endorsement of it over the other methods, which might be more accurate, and less expensive for some problems.

`metric` selects the kind of geometry with which to endow the constraint surface. This ultimately affects the definition of the covariant Hessian. 'flat' projects the result of applying the unconstrained Hessian onto the tangent space, while 'euclidean' and 'canonical' add on *connection* terms specific to their geometries. 'euclidean' is the default.

`motion` selects whether line movement along the manifold will be by the analytic solution to the geodesic equations of motions for the metric selected, or by a computationally less expensive approximation to the solution (default). (For a 'flat' metric, there is no geodesic equation, so this argument has no effect in that case.)

`verbose` determines whether the function will display reports on each iteration while the function executes. When this argument is 'verbose' (the default) data will be displayed and also recorded in the global `SGdata`. When this argument is 'quiet' then no convergence data is displayed or recorded.

`gradtol` and `ftol` We declare convergence if both of two conditions are true:
`grad/gradinit < gradtol` (default `1e-7`)
or `(f-fold)/f <  ftol` (default `1e-10`), where `gradinit` is the initial

magnitude of the gradient and `fold` is the value of $F(Y)$ at the last iteration.

`partition` is a cell array whose elements are vectors of indices that represent a disjoint partition of `1:p`. If F has no symmetry, then the partition is `num2cell(1:p)`. If $F(Y) = F(YQ)$ for all orthogonal Q, then the partition is `{1:p}`. The partition is `{1:2,3:5}` if the symmetry in F is $F(Y) = F(YQ)$ for orthogonal Q with sparsity structure

$$\begin{bmatrix} \times & \times & & & \\ \times & \times & & & \\ & & \times & \times & \times \\ & & \times & \times & \times \\ & & \times & \times & \times \end{bmatrix}.$$

The user could equally well specify `{3:5,1:2}` or `{[5 3 4],[2 1]}` for the partition, i.e. a partition is a set of sets. The purpose of the partition is to project away the null-space of the Hessian resulting from any block diagonal orthogonal symmetries of $F(Y)$. If the argument is omitted, our code will pick a partition by determining the symmetries of F (using its `partition.m` function). However, the user should be aware that a wrong partition can destroy convergence.

8.3.3 Sample Problems and Their Differentials

This section serves as a sample guide on the manipulation of objective functions of matrices with orthonormal columns. We have found a few common tricks worth emphasizing.

Once one has a formula for the objective function $F(Y)$, we define the formula for $dF(Y)$ implicitly by $\mathrm{tr}(V^* dF(Y)) = \frac{d}{dt} F(Y(t))|_{t=0}$ where $Y(t) = Y + tV$ (or any curve $Y(t)$ for which $\dot{Y}(0) = V$). The reader may recall that $\mathrm{tr}(A^*B) = \sum_{i,j} A_{ij} B_{ij}$, so it functions just like the real inner product for vectors[1] and the implicit definition of $dF(Y)$ is actually the directional derivative interpretation of the gradient of $F(Y)$ as an unconstrained function in a Euclidean space.

For most of the functions we have used in our examples, the easiest way to obtain the formula for dF is to actually use the implicit definition.

For example, if $F(Y) = \mathrm{tr}(AY^*BY)$ one then has

$$\mathrm{tr}(V^* dF(Y)) = \mathrm{tr}(AV^*BY + AY^*BV).$$

Since the value of the trace is invariant under cyclic permutations of products and transposes, we may rewrite this equation as

$$\mathrm{tr}(V^* dF(Y)) = \mathrm{tr}(V^*BYA + V^*B^*YA^*),$$

and, since V is unrestricted, this implies that $dF(Y) = BYA + B^*Y^*A^*$.

The process we recommend is:

[1] over complex numbers, we always take the real part of $\mathrm{tr}(A^H B)$.

- try to write $F(Y)$ as a trace
- compute $\frac{d}{dt}F(Y(t))|_{t=0}$ where we let $V = \dot{Y}(t)$
- use trace identities to rewrite every term to have a V^* in the front
- strip off the V^* leaving the $dF(Y)$

As a check, we recommend using the finite difference `dF.m` code supplied in the subdirectory `finitediff` to check derivations before proceeding.

The software needs a function called `ddF.m` which returns $\frac{d}{dt}dF(Y(t))|_{t=0}$ for $\dot{Y}(0) = H$. The sort of second derivative information required by the software is easier to derive than the first. If one has an analytic expression for $dF(Y)$, then one need only differentiate.

If, for some reason, the computation for `ddF.m` costs much more than two evaluations of $dF(Y)$ with `dF.m`, the reader may just consider employing the finite difference function for `ddF.m` found in `finitediff` (or simply use `ddF.m` as a check).

It is, however, strongly suggested that one use an analytic expression for computing $dF(Y)$, as the finite difference code for it requires a large number of function evaluations ($2np$).

8.3.3.1 The Procrustes Problem

The Procrustes problem (see [8]) is the minimization of $||AY - YB||_F$ for constant A and B over the manifold $Y^*Y = I$. This minimization determines the nearest matrix $\widehat{A}$ to A for which

$$Q^*\widehat{A}Q = \begin{bmatrix} B & * \\ 0 & * \end{bmatrix},$$

i.e. the columns of B span an invariant subspace of $\widehat{A}$.

The differential of $F(Y) = \frac{1}{2}||AY - YB||_F^2 = \frac{1}{2}\mathrm{tr}(AY - YB)^*(AY - YB)$ is given by

$$dF(Y) = A^*(AY - YB) - (AY - YB)B^*.$$

This can be derived following the process outlined above. Observe that

$$\begin{aligned} \frac{d}{dt}F(Y(t))|_{t=0} &= \frac{1}{2}\mathrm{tr}((AV - VB)^*(AY - YB) + (AY - YB)^*(AV - VB)) \\ &= \mathrm{tr}((AV - VB)^*(AY - YB)) \\ &= \mathrm{tr}(V^*(A^*(AY - YB)) - V^*(AY - YB)B^*). \end{aligned}$$

The second derivative of $F(Y)$ is given by the equation,

$$\frac{d}{dt}dF(Y(t))|_{t=0} = A^*(AH - HB) - (AH - HB)B^*,$$

where $\dot{Y}(0) = H$, which can be obtained by varying the expression for dF.

8.3.3.2 Nearest Jordan Structure

Now suppose that the B block in the Procrustes problem is allowed to vary with Y. Moreover, suppose that $B(Y)$ is in the nearest staircase form to Y^*AY, that is:

$$B(Y) = \begin{bmatrix} \lambda_1 I & * & * \\ 0 & \lambda_2 I & * \\ 0 & 0 & \dots \end{bmatrix}$$

for fixed block sizes, where the *-elements are the corresponding matrix elements of Y^*AY and the $\lambda_i I$ blocks are either fixed or determined by some heuristic, e.g. taking the average trace of the blocks they replace in Y^*AY. Then a minimization of $||AY - YB(Y)||_F$ finds the nearest matrix as a particular Jordan structure, where the structure is determined by the block sizes, and the eigenvalues are λ_i. When the λ_i are fixed, we call this the *orbit* problem, and when the λ_i are selected by the heuristic given we call this the *bundle* problem.

Such a problem can be useful in regularizing the computation of Jordan structures of matrices with ill-conditioned eigenvalues.

The form of the differential of $F(Y)$, surprisingly, is the same as that of $F(Y)$ for the Procrustes problem.

$$dF(Y) = A^*(AY - YB(Y)) - (AY - YB(Y))B(Y)^*.$$

This is because $\mathrm{tr}(-(AY - YB(Y))^* Y\dot{B}) = \mathrm{tr}((B(Y) - Y^*AY)^*\dot{B}) = 0$ for the B selected as above for either orbit or bundle case, where $\dot{B} = \frac{d}{dt}B(Y(t))|_{t=0}$.

In contrast, the form of the second derivatives is a bit more complicated, since the B now depend on Y.

$$\frac{d}{dt}dF(Y(t))|_{t=0} = \frac{d}{dt}dF_{Proc}(Y(t))|_{t=0} - A^*Y\dot{B} - AY\dot{B}^* + YB\dot{B}^* + Y\dot{B}B^*,$$

where $\dot{Y}(0) = H$, dF_{Proc} is just short for the Procrustes (B constant) part of the second derivative, and $\dot{B} = \frac{d}{dt}B(Y(t))|_{t=0}$ which is the staircase part of $Y^*AH + H^*AY$ (with trace averages or zeros on the diagonal depending on whether bundle or orbit).

8.3.3.3 Trace Minimization

In this case we consider a "Rayleigh quotient" style of iteration to find the eigenspaces of the smallest eigenvalues of a symmetric positive definite matrix A. That is, we can minimize $F(Y) = \frac{1}{2}\mathrm{tr}(Y^*AY)$ [2, 10, 16]. The minimizer Y^* will be an $n \times p$ matrix whose columns span the eigenspaces of the lowest p eigenvalues of A.

For this problem

$$dF(Y) = AY,$$

it is easily seen that

$$\frac{d}{dt}dF(Y(t))|_{t=0} = AH,$$

where $\dot{Y}(0) = H$.

8.3.3.4 Trace Minimization with a Nonlinear Term

We consider a related nonlinear eigenvalue problem, to the trace minimization of section 0.3.3.3: minimize $F(Y) = \frac{1}{2}(\mathrm{tr}(Y^*AY) + g(Y))$, where $g(Y)$ is some nonlinear term.

This sort of minimization occurs in electronic structures computations such as Local Density Approximations (LDA) where the columns of Y represent electron wave functions, A is a fixed Hamiltonian, and $g(Y)$ represents the energy of the electron-electron interactions (see, for example, [3, 19]). In this case, the optimal Y represents the state of lowest energy of the system.

The only additions to the differentials of the trace minimization are terms from $g(Y)$ which vary from problem to problem. For our example, we have taken $g(Y) = \frac{K}{4}\sum_i \rho_i^2$ for some coupling constant K, where $\rho_i = \sum_j |Y_{ij}|^2$ (the charge density).

In this case

$$dF(Y) = dF_{\mathrm{tracemin}} + K\mathrm{diag}\ (\rho)Y,$$

and

$$\frac{d}{dt}dF(Y(t))|_{t=0} = \frac{d}{dt}dF_{\mathrm{tracemin}}(Y(t))|_{t=0} + K\mathrm{diag}\ (\rho)H + K\mathrm{diag}\ (\frac{d}{dt}\rho)Y,$$

where $\dot{Y}(0) = H$, and $\frac{d}{dt}\rho_i|_{t=0} = \sum_j Y_{ij}H_{ij}$.

8.3.3.5 Simultaneous Schur Decomposition Problem

Consider two matrices A and B which in the absence of error have the same Schur vectors, i.e. there is a $Y \in O(n)$ such that Y^*AY and Y^*BY are both block upper triangular. Now suppose that A and B are somewhat noisy, from measurement errors, or some other kind of lossy filtering. In that case the Y that upper triangularizes A might not upper triangularize B as well. How does one find the best Y?

This is a problem that was presented to us by W. Schilders [17] who phrased it as a least squares minimization of $F(Y) = \frac{1}{2}(||\mathrm{low}(Y^*AY)||_F^2 + ||\mathrm{low}(Y^*BY)||_F^2)$, where $\mathrm{low}(M)$ is a mask returning the block lower triangular part of M, where M is broken up into 2×2 blocks.

For this problem the differential is a bit tricky and its derivation instructive,

$$\begin{aligned}
\mathrm{tr}(V^*dF(Y)) &= \mathrm{tr}(\mathrm{low}(Y^*AY)^*\mathrm{low}(Y^*AV + V^*AY) + \\
&\quad \mathrm{low}(Y^*BY)^*\mathrm{low}(Y^*BV + V^*BY)) \\
\mathrm{tr}(V^*dF(Y)) &= \mathrm{tr}(\mathrm{low}(Y^*AY)^*(Y^*AV + V^*AY) + \\
&\quad \mathrm{low}(Y^*BY)^*(Y^*BV + V^*BY)) \\
\mathrm{tr}(V^*dF(Y)) &= \mathrm{tr}(V^*(AY\mathrm{low}(Y^*AY)^* + A^*Y\mathrm{low}(Y^*AY) + \\
&= BY\mathrm{low}(Y^*BY)^* + B^*Y\mathrm{low}(Y^*BY))) \\
dF(Y) &= AY\mathrm{low}(Y^*AY)^* + A^*Y\mathrm{low}(Y^*AY) + \\
&\quad BY\mathrm{low}(Y^*BY)^* + B^*Y\mathrm{low}(Y^*BY),
\end{aligned}$$

where the second equation results from observing that $\mathrm{tr}(\mathrm{low}(M)^*\mathrm{low}(N)) = \mathrm{tr}(\mathrm{low}(M)^*N)$, and the third from properties of the trace.

With second derivatives given by,

$$
\begin{aligned}
\frac{d}{dt}dF(Y(t))|_{t=0} \quad = \quad & AH\mathrm{low}(Y^*AY)^* + A^*H\mathrm{low}(Y^*AY) + AY\mathrm{low}(\frac{d(Y^*AY)}{dt})^* \\
& +A^*Y\mathrm{low}(\frac{d(Y^*AY)}{dt}) + BH\mathrm{low}(Y^*BY)^* + B^*H\mathrm{low}(Y^*BY) \\
& +BY\mathrm{low}(\frac{d(Y^*BY)}{dt})^* + B^*Y\mathrm{low}(\frac{d(Y^*BY)}{dt}),
\end{aligned}
$$

where $\dot{Y}(0) = H$, $\frac{d(Y^*AY)}{dt}|_{t=0} = H^*AY + Y^*AH$, and $\frac{d(Y^*BY)}{dt}|_{t=0} = H^*BY + Y^*BH$.

8.3.3.6 Simultaneous Diagonalization

Another problem like simultaneous Schur problem, involving sets of matrices with similar structures, is the problem of finding the best set of eigenvectors for a set of symmetric matrices given that the matrices are known to be simultaneously diagonalizable, but may have significant errors in their entries from noise or measurement error. Instances of this problem arise in the Psychometric literature, where it is called an INDSCAL problem [6].

Phrased in terms of a minimization, one has a set of symmetric matrices A_i and wishes to find $Y \in O(n)$ that minimizes $F(Y) = \frac{1}{2}\sum_i ||[Y, A_i]||_F^2 = \frac{1}{2}\sum_i ||YA_i - A_iY||_F^2$.

We then have

$$dF(Y) = \sum_i [[Y, A_i], A_i^*],$$

and

$$\frac{d}{dt}dF(Y(t))|_{t=0} = \sum_i [[H, A_i], A_i^*],$$

where $\dot{Y}(0) = H$.

8.3.4 Numerical Examples

We have provided implementations for the sample problems described.

We present the examples and their verbose outputs. We have included these outputs so the reader can check that his/her copy of the package is executing properly. The user should be running matlab (version 5 or compatible) in the following directory:

```
>> !ls
Fline.m       dgrad.m      grad.m              move.m       sg_min.m
README        dimension.m  gradline.m          nosym.m      sg_newton.m
clamp.m       dtangent.m   invdgrad_CG.m       partition.m  sg_prcg.m
connection.m  examples     invdgrad_MINRES.m   sg_dog.m     tangent.m
dFline.m      finitediff   ip.m                sg_frcg.m
```

(note, there may also be the additional subdirectories `docs/` or `@cell/`).

Each example problem has a subdirectory in the `examples` subdirectory.

```
>> !ls examples
jordan      ldatoy      procrustes  simdiag     simschur    tracemin
```

Each of these subdirectories contains an implementation for $F(Y)$ ($dF(Y)$ and $ddF(Y,H)$), a `parameters` function to set a global, `FParameters`, of fixed parameters of $F(Y)$ (this function must be called before any other), a `guess` function to generate an initial guess, and possibly some auxiliary functions for computing $F(Y)$, related quantities, or instances of a specific problem.

```
>> !ls examples/ldatoy
F.m          dF.m         guess.m
Kinetic.m    ddF.m        parameters.m
>> !ls examples/jordan
Block.m      dBlock.m     ddF.m        guess.m      post.m
F.m          dF.m         frank.m      parameters.m
```

We executed the examples in the supported optimization modes ('newton', 'dog', 'frcg', and 'prcg') and the 'euclidean' metric. For different instances of the same problem, different modes might perform best, so the reader should not feel that a particular mode will exhibit superior performance in all instances.

8.3.4.1 A Sample Procrustes Problem

In this example, we attempt to find a Y which minimizes $||AY - YB||_F$ for a pair of randomly selected A (12×12) and B (4×4).

```
>> !cp examples/procrustes/*.m .
>> randn('state',0);
>> [A,B] = randprob;
>> parameters(A,B);
>> Y0 = guess;
>> [fn,Yn] = sg_min(Y0,'newton','euclidean');
iter    grad            F(Y)              flops        step type
0       2.334988e+01    3.071299e+01        4751       none
  invdgrad: Hessian not positive definite, CG terminating early
1       1.171339e+01    1.376463e+01      365678       Newton step
  invdgrad: Hessian not positive definite, CG terminating early
2       7.843279e+00    7.616381e+00      677599       Newton step
  invdgrad: Hessian not positive definite, CG terminating early
3       5.131680e+00    4.945824e+00      992823       Newton step
  invdgrad: Hessian not positive definite, CG terminating early
4       5.642834e+00    3.512826e+00     1293761       Newton step
  invdgrad: Hessian not positive definite, CG terminating early
5       5.500553e+00    1.721329e+00     1607969       Newton step
  invdgrad: Hessian not positive definite, CG terminating early
6       4.666307e+00    1.192561e+00     1964675       Newton step
  invdgrad: Hessian not positive definite, CG terminating early
7       3.576069e+00    6.850532e-01     2272355       Newton step
  invdgrad: max iterations reached inverting the hessian by CG
8       1.228119e+00    3.046816e-01     2820625       Newton step
9       4.673779e-02    2.506848e-01     3345266       Newton step
10      6.411668e-04    2.505253e-01     3873582       Newton step
11      1.965463e-06    2.505253e-01     4430245       Newton step
12      1.620267e-06    2.505253e-01     5022629       Newton step
```

Figure 1 shows the convergence curve for this run.

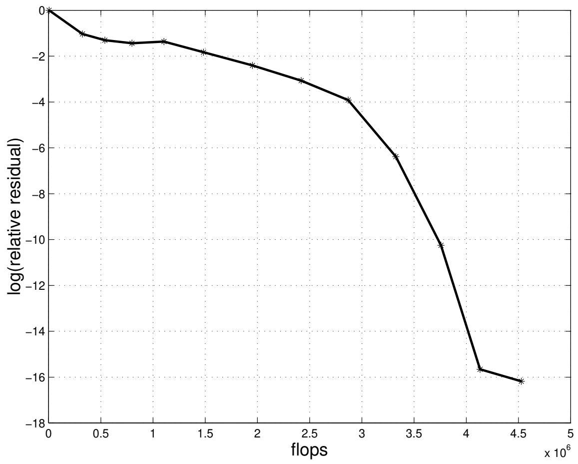

FIGURE 8.1: **Procrustes Problem**

8.3.4.2 A Sample Jordan Block Problem

We now look for a nearby matrix with a Jordan structure $J_4(\alpha) \oplus J_2(\beta) \oplus$ (don't care) to the 12×12 Frank matrix, a matrix whose Jordan structure is known to be very difficult.

```
>> !cp examples/jordan/*.m .
>> A = frank(12);
>> evs = sort(eig(A)); eigvs = [mean(evs(1:4)) mean(evs(5:6))];
>> parameters(A,eigvs,[4; 2],'bundle')
   1 Jordan block  of order 4 of eigenvalue 0.076352
   1 Jordan block  of order 2 of eigenvalue 0.464128
>> Y0 = guess;
>> [fn,Yn] = sg_min(Y0,'dog','euclidean');
iter    grad            F(Y)              flops         step type
0       1.775900e-02    9.122159e-06        16257       none
  invdgrad: max iterations reached inverting the hessian by MINRES
1       1.011261e-03    4.121994e-07       4638811      good dog
  invdgrad: max iterations reached inverting the hessian by MINRES
2       4.250874e-04    3.523843e-07       9274101      good dog
  invdgrad: max iterations reached inverting the hessian by MINRES
3       2.146629e-03    1.696542e-07     13936305       good dog
  invdgrad: max iterations reached inverting the hessian by MINRES
4       5.260491e-04    5.326314e-08     18553494       good dog
  invdgrad: max iterations reached inverting the hessian by MINRES
5       4.861126e-04    7.545415e-09     23231554       good dog
  invdgrad: max iterations reached inverting the hessian by MINRES
6       4.228777e-05    3.014858e-09     27846197       good dog
  invdgrad: max iterations reached inverting the hessian by MINRES
7       2.954536e-06    2.924391e-09     32508118       good dog
8       2.868836e-08    2.924238e-09     37207124       good dog
9       2.093780e-09    2.924238e-09     38384200       good dog
10      2.479121e-09    2.924238e-09     39163025       good dog
11      1.713240e-09    2.924238e-09     39921924       bad dog
```

Figure 2 shows the convergence curve for this run.

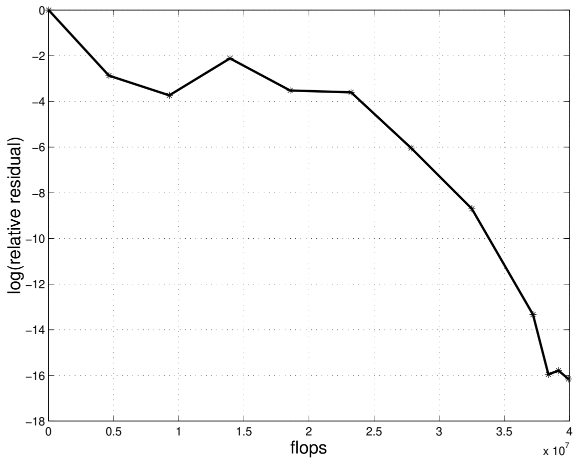

FIGURE 8.2: **Jordan Problem**

8.3.4.3 A Sample Trace Minimization Problem

In this problem we minimize trace(Y^*AY) where A is a 12×12 second difference operator in 1-dimension and Y is 12×4.

Note: we do not recommend that this problem be solved this way, since, for constant A, it can be done faster with conventional eigenvalue solvers.

```
>> !cp examples/tracemin/*.m .
>> randn('state',0);
>> A = Kinetic(12);
>> parameters(A);
>> Y0 = guess(4);
>> [fn,Yn] = sg_min(Y0,'frcg','euclidean');
iter    grad            F(Y)              flops
0       2.249340e+00    4.339273e+00       3807
1       1.635681e+00    1.774835e+00     106770
2       6.506188e-01    1.160615e+00     196319
3       4.400435e-01    1.043566e+00     295232
4       3.460854e-01    9.601858e-01     386244
5       3.418816e-01    9.122465e-01     471214
6       2.591414e-01    8.543284e-01     560763
7       1.225067e-01    8.319959e-01     661414
8       5.660065e-02    8.280403e-01     785192
9       2.346852e-02    8.271969e-01     935587
10      9.258692e-03    8.270659e-01    1105819
11      4.723177e-03    8.270398e-01    1261468
12      3.282461e-03    8.270316e-01    1417625
13      1.984453e-03    8.270281e-01    1576710
14      9.808233e-04    8.270270e-01    1726651
15      4.339561e-04    8.270267e-01    1876592
16      1.990167e-04    8.270267e-01    2022569
17      6.563434e-05    8.270267e-01    2163632
18      3.165945e-05    8.270267e-01    2322727
19      2.015557e-05    8.270267e-01    2468720
20      1.261539e-05    8.270267e-01    2612825
21      7.123104e-06    8.270267e-01    2748702
22      2.552485e-06    8.270267e-01    2935969
23      9.679096e-07    8.270267e-01    3119232
24      3.168742e-07    8.270267e-01    3287067
```

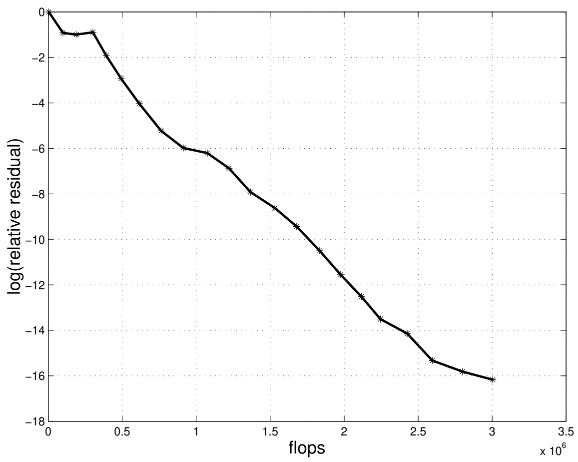

FIGURE 8.3: **Trace Minimization Problem**

```
25      2.062080e-07    8.270267e-01      3472366
```

Figure 3 shows the convergence curve for this run.

8.3.4.4 A Sample LDA Toy Problem (trace minimization with nonlinear term)

We attempt to solve a toy Local Density Approximation problem described in section 0.3.3.4, for four electrons on a 1D grid with twelves points and a coupling constant of 20. In this case, we use the first four eigenvectors of the linear problem as a starting point.

```
>> !cp examples/ldatoy/*.m .
>> K = Kinetic(12);
>> parameters(K,20);
>> Y0 = guess(4);
>> [fn,Yn] = sg_min(Y0,'newton','euclidean');
iter    grad            F(Y)              flops         step type
0       2.432521e+00    7.750104e+00         4554       none
  invdgrad: Hessian not positive definite, CG terminating early
1       4.770038e-01    7.546426e+00       367756       steepest step
2       4.664971e-02    7.531450e+00       789739       Newton step
3       3.363736e-04    7.531232e+00      1284452       Newton step
4       1.809273e-07    7.531232e+00      1976096       Newton step
5       1.843536e-07    7.531232e+00      2778377       Newton step
```

Figure 4 shows the convergence curve for this run.

8.3.4.5 A Sample Simultaneous Schur Decomposition Problem

In this case we use the sample problem given to us by Schilders.

```
>> !cp examples/simschur/*.m .
>> [A,B] = noisy;
>> parameters(A,B);
>> Y0 = guess;
>> [fn,Yn] = sg_min(Y0,'prcg','euclidean');
iter    grad            F(Y)              flops
0       2.205361e-01    1.687421e-02        56493
```

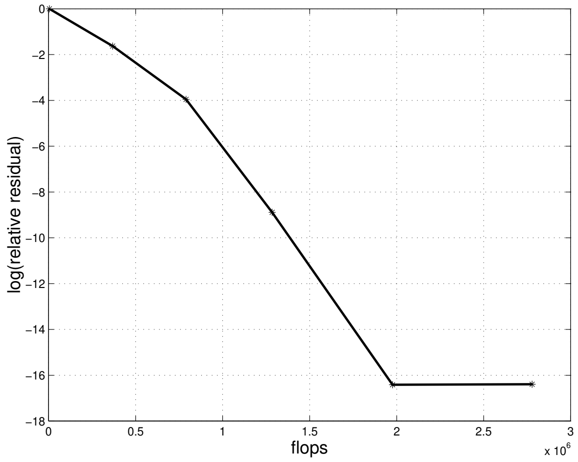

FIGURE 8.4: **LDA Toy Problem**

1	1.087502e-01	8.329681e-03	1537489
2	5.834580e-02	5.947527e-03	3045842
3	3.954742e-02	5.322286e-03	4243901
4	1.920065e-02	5.016544e-03	5700349
5	9.449972e-03	4.938962e-03	7062826
6	4.304992e-03	4.920311e-03	8509245
7	2.649005e-03	4.916373e-03	9982763
8	1.839592e-03	4.914510e-03	11492687
9	1.162756e-03	4.913601e-03	12952876
10	5.450777e-04	4.913326e-03	14401149
11	2.461091e-04	4.913273e-03	15848402
12	1.463300e-04	4.913261e-03	17354270
13	7.851036e-05	4.913257e-03	18762063
14	5.400429e-05	4.913256e-03	20205661
15	2.786051e-05	4.913255e-03	21689844
16	1.570592e-05	4.913255e-03	23166944
17	8.390343e-06	4.913255e-03	24528843
18	4.060847e-06	4.913255e-03	25901423
19	2.047984e-06	4.913255e-03	27271928
20	1.161407e-06	4.913255e-03	28481228
21	7.367792e-07	4.913255e-03	29776935
22	4.640484e-07	4.913255e-03	31533888
23	2.206626e-07	4.913255e-03	33245809
24	1.467168e-07	4.913255e-03	35084169
25	7.946751e-08	4.913255e-03	36975859
26	4.631213e-08	4.913255e-03	38934071
27	2.119895e-08	4.913255e-03	40761932

Figure 5 shows the convergence curve for this run.

8.3.4.6 A Sample Diagonalization Problem

We attempt to simultaneously diagonalize two noisy (random noise) versions of `diag(1:10)` and `diag(1:10)^2`. Our initial point is a small random perturbation of the identity.

```
>> !cp examples/simdiag/*.m .
>> randn('state',0);
>> [A,B] = noisy;
```

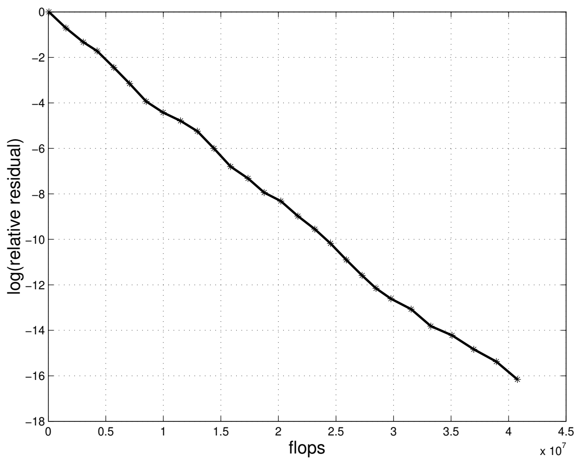

FIGURE 8.5: **Simultaneous Schur Problem**

```
>> parameters(A,B);
>> Y0 = guess;
>> [fn,Yn] = sg_min(Y0,'frcg','euclidean');
iter    grad            F(Y)              flops
0       2.887592e+03    9.671138e+02        30940
1       1.419011e+03    2.411349e+02       343392
2       6.199732e+02    7.601495e+01       718792
3       3.051636e+02    3.300591e+01      1106535
4       2.538995e+02    2.078422e+01      1518380
5       1.440247e+02    1.258174e+01      1923273
6       1.189063e+02    9.500450e+00      2332153
7       9.279907e+01    7.309071e+00      2743897
8       8.483878e+01    6.093323e+00      3161294
9       7.763100e+01    4.117233e+00      3609412
10      7.448803e+01    3.220241e+00      4026633
11      6.404668e+01    2.328540e+00      4439092
      .......(many iterations later).......
169     2.196881e-06    4.010660e-07     67843218
170     2.309773e-06    4.010660e-07     68212219
171     2.435095e-06    4.010660e-07     68558476
172     1.763263e-06    4.010660e-07     68946596
173     2.081518e-06    4.010660e-07     69314166
174     1.251419e-06    4.010660e-07     69639992
175     1.474565e-06    4.010660e-07     70028565
176     8.949991e-07    4.010660e-07     70353929
177     8.506853e-07    4.010660e-07     70742988
178     7.154996e-07    4.010660e-07     71068090
179     8.003997e-07    4.010660e-07     71395085
180     3.164977e-07    4.010660e-07     71721113
181     3.701357e-07    4.010660e-07     72030590
```

Figure 6 shows the convergence curve for this run.

8.3.5 Modifying the Templates

This template is written to function as is, but can be tailored to specific problems, resulting in substantial improvements in performance. There are also functions which could be in-lined and computations which can be reused of

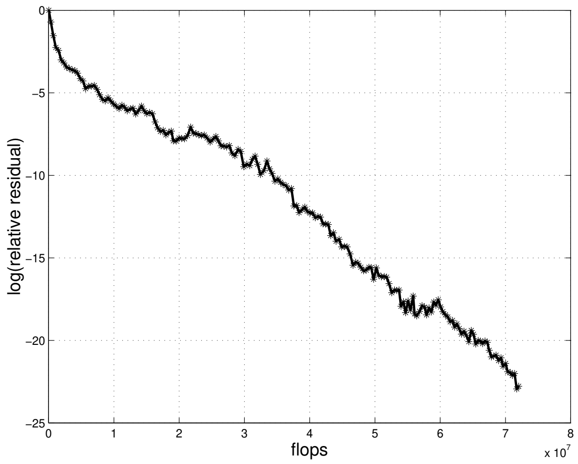

FIGURE 8.6: **Simultaneous Diagonalization Problem**

which we have not taken advantage for the sake of having a more readable code. Many components are the usual routines found in the literature (our line minimization routines are based on matlab's `fmin` and `fzero` for example). Certainly it is possible for improvements to be made in the optimization routines themselves and we welcome suggestions from optimization experts.

To make modifications as painless as possible, we present a section sketching the basic structure of the `sg_min` code. Readers who do not wish to modify any of the functions beyond `F.m`, `dF.m`, and `ddF.m` can skip this section.

8.3.5.1 Subroutine Dependencies

Figure 7 shows the dependencies of the various subroutines on each other. We have grouped together the routines based on four loosely defined roles:

- **Objective Function** These are the routines which implement the objective function and its derivatives. This group is the only group that a user interesting only in the basic functionality need ever modify to adapt the template. Functions in this group are used by functions in the Geometrized Objective Function group and High Level Algorithm group.

- **Geometric Implementation** These routines implement the geometric features of the Stiefel manifold. This includes the projection of unconstrained vectors onto the constraint (i.e. tangent) space (`tangent`), line movement (`move`), and the connection term used for covariant differentiation (`connection`). These routines are independent of the Objective Function group, but are essential to the Geometrized Objective Function group.

- **Geometrized Objective Function** This group uses the routines of the Geometric Implementation to project the differential and second differential of $F(Y)$ onto the constraint (i.e. tangent) surface producing the geometrically covariant gradient (`grad`) and Hessian (`dgrad`). Additionally, two routines for inverting the covariant Hessian (`invdgrad_CG` and `invdgrad_MINRES`) are located here. This group provides the raw tools

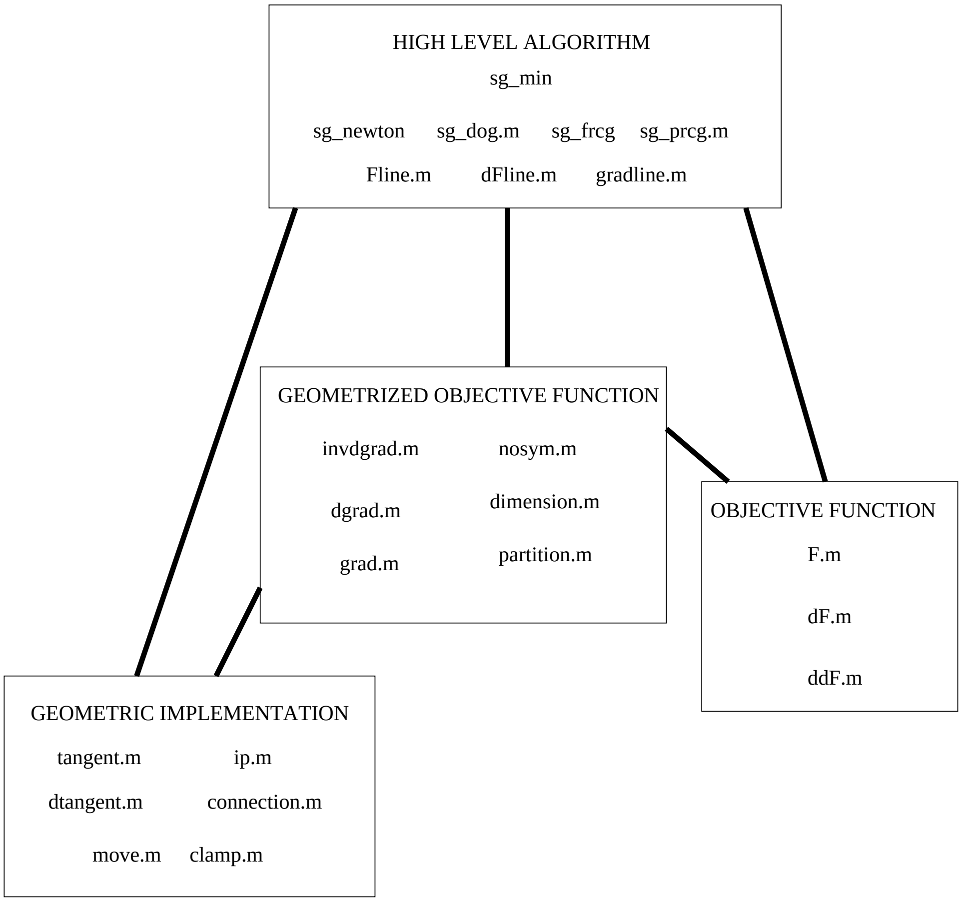

FIGURE 8.7: **Our code tree: dependencies of various modules**

out of which one builds implementations in the High Level Algorithm group. Lastly, functions which detect (`partition`) and remove (`nosym`) block diagonal orthogonal symmetries are found in the group.

- **High Level Algorithm** This group implements the constrained versions of various optimization algorithms. It is here search directions are selected, line minimizations are performed (using `Fline`, `dFline`, and `gradline`), and convergence criteria are defined.

Lastly, every function reads from a global `SGParameters` structure whose fields contain information about the manifold and the computation. In our examples we have used a separate global `FParameters` to store information related to the computation of $F(Y)$. The fields of `SGParameters` are set in `sg_min`.

8.3.5.2 Under the Hood

`sg_min` is built from a couple fairly elementary optimization codes which have been put through a geometrization that allows them to be situated on the Stiefel manifold. The basic elements of a geometrization are the rules for how to take inner products, how to turn unconstrained differentials into constrained gradients, how to differentiate gradient fields covariantly, and how to move about on the manifold.

The `sg_min` routine parses the arguments and sets the defaults. Finally, it calls `sg_newton`, `sg_dog`,`sg_frcg`, or `sg_prcg`.

The pseudocode for `sg_newton` and `sg_frcg` (as examples) are given below.

```
function [fn,Yn]= sg_newton(Y)
    g_initial = grad(Y); g = g_initial;
    f = F(Y); f_old = 2*f;
    while (||g|| > eps) & (||g||/||g_initial|| >gradtol) & (|f_old/f−1 | > ftol)
            sdir = -g;
% steepest descent direction and line minimization on different scales
            fsa = minimize F(move(Y,sdir,sa)) for sa ∈ [−1, 1] · | f / ⟨g,sdir⟩ |;
            fsb = minimize F(move(Y,sdir,sb)) for sb ∈ [−1, 1] · | ⟨g,sdir⟩ / ⟨sdir,dgrad·sdir⟩ |;

            ndir = -dgrad^{−1}· g;
% newton direction and line minimization
            fna = minimize F(move(Y,ndir,na)) for na ∈ [−1, 1] · | f / ⟨g,ndir⟩ |;
            fnb = minimize F(move(Y,ndir,nb)) for nb ∈ [−1, 1] · | ⟨g,ndir⟩ / ⟨ndir,dgrad·ndir⟩ |;

% compare the best newton function value with the best steepest
            if (fsa < fsb) st=sa; fst=fsa; else st=sb; fst=fsb; end
            if (fna < fnb) nt=na; fnt=fna; else nt=nb; fnt=fnb; end
            if (fst <fnt)
                    dir = sdir; t = st; f_new = fst;
            else
                    dir = ndir; t = nt; f_new = fnt;
            end
% move to the new point
            Y = move(Y,dir,t); f_old= f; f = f_new;
            g=grad(Y);
    end
    fn = f;
    Yn = Y;
```

```
function [fn,Yn]= sg_frcg(Y)
    g_initial = grad(Y); g = g_initial;
    f = F(Y); f_old = 2*f;
    while (||g|| > eps) & (||g||/||g_initial|| >gradtol) & (|f_old/f−1 | > ftol)
            dir = -g;
% get a Hessian conjugate direction
            if (not first iteration)
                    dir = dir - (dir·(dgrad·dir_old)) / (dir_old·(dgrad·dir_old)) *dir_old;
            end
% line minimization
            fcga = minimize F(move(Y,dir,cga)) for cga ∈ [−1, 1] · | f / ⟨g,dir⟩ |;
            fcgb = minimize F(move(Y,dir,cgb)) for cgb ∈ [−1, 1] · | ⟨g,dir⟩ / ⟨dir,dgrad·dir⟩ |;

            if (fcga < fcgb) t=cga; f_new=fcga; else t=cgb; f_new=fcgb; end
% move to the new point
            [Y,dir_old] = move(Y,dir,t); f_old= f; f = f_new;
            g=grad(Y);
    end
    fn = f;
    Yn = Y;
```

These are fairly generic routines for optimization. At this level of description, one would not necessarily be able to tell them from unconstrained routines (see [9, 11, 15, 14]). What places them on the Stiefel manifold are the definitions of `grad`, `dgrad`, `ip` (the dot product), and `move`, which have been made in such a way that the constraints of the Stiefel manifold are respected. Likewise, `sg_dog` and `sg_prcg` have been similarly transported.

8.3.5.3 What to Modify

Here is a sketch of specific modifications a user may wish to make on the code.

- **Line Minimization** Currently, line minimizations are being performed by the matlab functions `fmin` and `fzero` functions through the wrappers `Fline`, `dFline`, and `gradline`. However, one may have better ways to line minimize within an optimization routine or one may wish to supplement `fmin` and `fzero` for reasons of economy.

- **High Level Algorithms** The user may wish to use a minimization routine differing from or more sophisticated than the methods currently available, or one may have a different stopping criteria [4] that we did not anticipate. It is possible to adapt other flat optimization algorithms to new `sg_` algorithms without having to know too many details. Most unconstrained minimizations contain line searches (`y` $\rightarrow$ `y+t*v`), Hessian applications (`H*v`) and inner products (`v'w`). To properly geometrize the algorithm, one has to do no more than replace these components with `move(Y,V,t)`, `dgrad(Y,V)`, and `ip(Y,V,W)` respectively.

- **Hessian Inversion** We currently provide codes to invert the covariant Hessian in the functions `invdgrad_CG` (which uses a conjugate gradient method) and `invdgrad_MINRES` (which uses a MINRES algorithm). Any matrix inversion routine which can stably invert black-box linear operators could be used in place of these and no other modifications to the code would be required.

- **Workspaces** To make the code more readable we have not implemented any workspace variables even though there are many computations which could be reused. For example, in the objective function group, for many of the sample problems, products of the form `A*Y` or `Y*B` or complicated matrix functions of `Y` such as `B(Y)` in the Jordan example could be saved in a workspace. Some of the terms in the computation of `grad` and `dgrad` could likewise be saved (in fact, our current implementation essentially recomputes the gradient every time the covariant Hessian is applied).

- **In-lining** Although we have separated the `tangent` and `dtangent` functions from the `grad` and `dgrad` functions, one often finds that when these functions are in-lined, that some terms cancel out and therefore do not need to be computed. One might also consider in-lining the `move` function when performing line minimizations so that one could reuse data to make function evaluations and point updates faster.

- **Sphere or $O(n)$ Geometry** Some problems are always set on the unit sphere or on $O(n)$, both of which have simpler geometries than the Stiefel manifold. Geodesics on both are easier to compute, as are tangent projections. To take advantage of this, one should modify the Geometric Implementation group.

- **Globals** In order to have adaptable and readable code, we chose to use a global structures, `SGParameters` and `FParameters`, to hold data which could be put in the argument lists of all the functions. The user may wish to modify his/her code so that this data is passed explicitly to each function as individual arguments.

- **Preconditioning** The Hessian solve by conjugate gradient routine, `invdgrad`, does not have any preconditioning step. Conjugate gradient can perform very poorly without preconditioning, and the preconditioning of black box linear operators like `dgrad` is a very difficult problem. The user might try to find a good preconditioner for `dgrad`.

8.3.6 Geometric Technicalities

This section is a brief explanation of the geometric concepts used to produce the methods employed by `sg_min` and its subroutines. This information is mostly a recapitulation of material readily available in the differential geometry literature.

8.3.6.1 Manifolds

A manifold, M, is *a collection of points which have a differential structure.* In plain terms, this means that one is able to take derivatives of some reasonable class of functions, the C^∞ functions, defined on M. What this class of differentiable functions is can be somewhat arbitrary, though some technical consistency conditions must be satisfied. We will not be discussing those conditions in this section.

We consider the manifold $\mathrm{Stief}(n,k)$ (for purposes of clearer explanation we restrict our discussion to the real version of the Stiefel manifold) of points which are written as $n \times k$ matrices satisfying the constraint $Y^*Y = I$. We will select for our set of C^∞ functions those real valued functions which are restrictions to $\mathrm{Stief}(n,k)$ of functions of nk variables which are $C^\infty(\mathcal{R}^{n\times k})$ about $\mathrm{Stief}(n,k)$ in the $\mathcal{R}^{n\times k}$ sense. It should not be difficult to convince oneself that the set of such functions must satisfy any technical consistency condition one could hope for.

Additionally, $M = \mathrm{Stief}(n,k)$ is a smooth manifold. This means that about every point of $\mathrm{Stief}(n,k)$ we can construct a local coordinate system which is C^∞. For example, consider the point

$$Y = \begin{bmatrix} I \\ 0 \end{bmatrix}$$

and a point in the neighborhood of Y,

$$\widetilde{Y} = \begin{bmatrix} A \\ B \end{bmatrix}.$$

For small B, we can solve for A in terms of the components of B and the components of an arbitrary small antisymmetric matrix Δ by solving $S^*S = I - B^*B$ for symmetric S and letting $A = e^{\Delta}S$. This can always be done smoothly and in a locally 1-to-1 fashion for small enough B and Δ. Since the components of B are all smooth functions (being restrictions of the coordinate functions of the ambient $\mathcal{R}^{n\times k}$ space), and since the solution for A is $C^\infty(R^{(n-k)\times k} \oplus R^{k\times k})$ for small enough B and Δ, we have shown that any point of Stief(n,k) can be expressed smoothly as a function of $(n-k)k + k(k-1)/2 = nk - k(k+1)/2$ variables. The only difference between $\begin{bmatrix} I \\ 0 \end{bmatrix}$ is a euclidean rigid motion; therefore, this statement holds for all points in Stief(n,k).

Summarizing, a manifold is a set of points, with a set of differentiable functions defined on it as well as a sense of local coordinatization in which the dependent coordinates can be represented as smooth functions of the independent coordinates. Specifically, we see that there are always $nk - k(k+1)/2$ independent coordinates required to coordinatize neighborhoods of points of Stief(n,k). This number is called the *dimension* of the manifold, and it should be the same for every point of a manifold (if not, then the manifold is either disconnected or not actually smooth).

8.3.6.2 The Difference Between a Tangent and a Differential

Given a smooth point x on any manifold and a smooth path $p(s)$ in the manifold such that $p(0) = x$, one can construct a differential operator on the C^∞ functions defined near x by the rule

$$D_p f = \frac{d}{ds} f(p(s))|_{s=0}.$$

These differential operators at x form a finite dimensional vector space, and the dimension of this vector space is equal to the dimension of the manifold. This vector space is called the *tangent space* of M at x, and is often denoted $T_x(M)$.

For the Stiefel manifold, and, generally, all smooth constraint manifolds, the tangent space at any point is easy to characterize. The constraint equation $Y^*Y = I$ can be thought of as $k(k+1)/2$ independent functions on Stief(n,k) which must all have constant value. If H is a tangent vector of Stief(n,k) at Y, it must then be that

$$H^*Y + Y^*H = 0,$$

which is obtained by taking $\frac{d}{dt}Y(t)^*Y(t) = 0|_{t=0}$ (where $\dot{Y}(0) = H$). In a constraint manifold, the tangents at Y can equivalently be identified with those infinitesimal displacements of Y which preserve the constraint equations to first order.

For H to be a tangent, its components must satisfy $k(k+1)/2$ independent equations. These equations are all linear in the components of H, and thus the set of all tangents is a $nk - k(k+1)/2$ dimensional subspace of the vector space $\mathcal{R}^{n\times k}$.

A differential operator D_H may be associated with an $n \times k$ matrix, H, given by

$$D_H f = \frac{d}{dt} f(Y + tH)|_{t=0}.$$

D_H is the directional derivative operator in the H direction. One may observe that for H to be a tangent vector, the tangency condition is equivalent to $D_H(Y'Y) = 0$.

While tangents are $n \times k$ matrices and, therefore, have associated differential operators, *differentials* are something else. Given a C^∞ function f and a point Y, one can consider the equation $D_H f$ for $H \in \mathcal{R}^{n\times k}$ (H is not necessarily tangent). This expression is linear in H and takes on some real value. It is thus possible to represent it as a linear function on the vector space $\mathcal{R}^{n\times k}$,

$$D_H f = \mathrm{tr}(H^* Z),$$

for some appropriate $n \times k$ matrix Z whose values depend on the first order behavior of f near Y. We identify this Z matrix with df, the differential of f at Y. This is the same differential which is computed by the `dF` functions in the sample problems.

For any constraint manifold the differential of a smooth function f can be computed without having to know anything about the manifold itself. One can simply use the differentials as computed in the ambient space (the unconstrained $\mathcal{R}^{n\times k}$ derivatives in our case). If one then restricts one's differential operators to be only those in tangent directions, then one can still use the unconstrained df in $\mathrm{tr}(H^* df)$ to compute $D_H f$ for $H \in T_Y(\mathrm{Stief}(n,k))$. This is why it requires no geometric knowledge to produce the `dF` functions.

8.3.6.3 Inner Products, Gradients, and Differentials

In flat spaces, we often identify the differential of a function with its gradient. However, when dealing with a more general setting, one can run into problems making sense out of such a definition.

For example, the gradient is a vector, and it should be possible to think of vectors as infinitesimal displacements of points. In $\mathrm{Stief}(n,k)$, any infinitesimal displacement δY must satisfy $\delta Y^* Y + Y^* \delta Y = 0$. Thus, df may not always be a vector, since it does not necessarily satisfy this equation. A gradient should be an infinitesimal displacement that points in the direction of the displacement which will give the greatest increase in f.

If the tangent space has an inner product, though, one can find a useful way to identify the df uniquely with a tangent vector. Let $ip(H_1, H_2)$ be a symmetric nondegenerate bilinear form on the tangent space of $\mathrm{Stief}(n,k)$ at Y. Then one can define the gradient, G, implicitly by,

$$ip(G,H) = \mathrm{tr}(H^* df) = D_H f.$$

Since ip is a nondegenerate form, this is sufficient to define G. The function `tangent` carries out this projection from differentials to tangents (shown in Figure 8). This operation is performed by `grad` to produce the gradient of the objective function.

8.3.6.4 Getting Around Stief(n,k)

We've now laid the groundwork for a meaningful definition of the gradient in the setting of a constraint manifold. At this point, one could run off and try to do a steepest descent search to maximize one's objective functions. Trouble will arise, however, when one discovers that there is no sensible way to combine

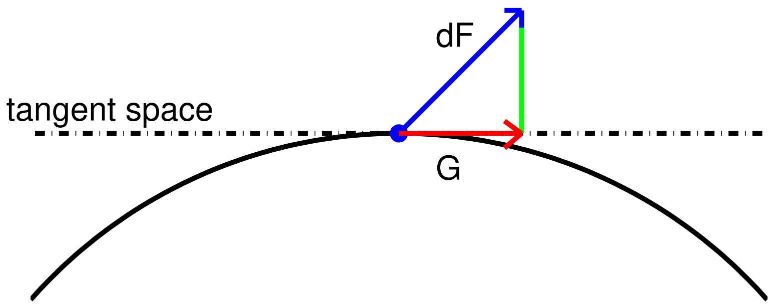

FIGURE 8.8: *The unconstrained differential of $F(Y)$ can be projected to the tangent space to obtain the covariant gradient, G, of F.*

a point and a displacement to produce a new point because, for finite s, $Y+sH$ violates the constraint equations, and thus does not give a point on the manifold.

For any manifold, one updates Y by solving a set of differential *equations of motion* of the form

$$\begin{aligned}\frac{d}{dt}Y &= H\\ \frac{d}{dt}H &= -\Gamma(H,H).\end{aligned}$$

The Γ term is crafted to ensure that H remains a tangent vector for all times t, thus keeping the path on the manifold. It is called the *connection*. (The connection term also depends on Y.)

To see how these equations could be satisfied, we take the infinitesimal constraint equation for H,

$$H^*Y + Y^*H = 0,$$

and differentiate it with respect to t, to get

$$\Gamma(H,H)^*Y + Y^*\Gamma(H,H) = 2H^*H.$$

This can be satisfied generally by $\Gamma(H,H) = Y(H^*H)+T(H,H)$ where $T(H,H)$ is some arbitrary tangent vector.

In the next subsection, we will have reason to consider $\Gamma(H_1,H_2)$ for $H_1 \neq H_2$. For technical reasons, one usually requests that

$$\Gamma(H_1,H_2) = \Gamma(H_2,H_1)$$

(this is called the *torsion free* property), and that

$$ip(\Gamma(H_1,H_3),H_2) + ip(H_1,\Gamma(H_2,H_3)) = 0$$

(this is called the *metric compatibility* property). These two properties uniquely determine the $T(H_1,H_2)$ term, and thereby uniquely specify the connection. On any manifold, the unique connection with these properties is called the *Levi-Civita* connection.

The connection for the Stiefel manifold using two different inner products (the Euclidean and the canonical) can be found in the work by Edelman, Arias,

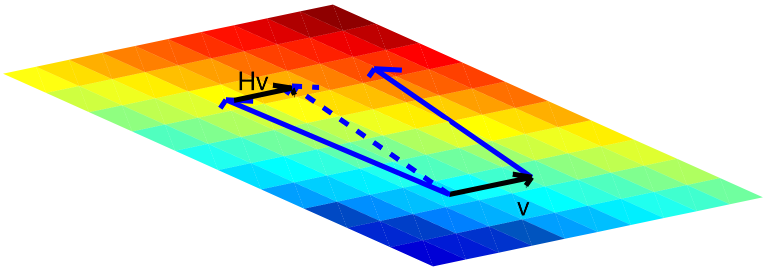

FIGURE 8.9: *In a flat space, comparing vectors at nearby points is not problematic since all vectors lie in the same tangent space.*

and Smith (see [1, 7, 18]). The function `connection` computes $\Gamma(H_1, H_2)$ in the template software.

Usually the solution of the equations of motions on a manifold are very difficult to carry out. For the Stiefel manifold, analytic solutions exist and can be found in the aforementioned literature, though we have found very little performance degradation between moving along paths via the equations of motion and simply performing some orthogonalizing factorization on $Y + sH$, as long as the displacements are small. The `move` function supports multiple methods of geodesic motion, depending on the degree of approximation desired.

8.3.6.5 Covariant Differentiation

With the notion of a gradient and a notion of movement that respects the constraints of the manifold, one might wish to begin with an optimization routine of some sort. A steepest descent could be implemented from these alone. However, in order to carry out sophisticated optimizations, one usually wants some sort of second derivative information about the function.

In particular, one might wish to know by how much one can expect the gradient to change if one moves from Y to $Y + \epsilon H$. This can actually be a difficult question to answer on a manifold. Technically, the gradient at Y is a member of $T_Y(\mathrm{Stief}(n,k))$, while the gradient at $Y + \epsilon H$ is a member of $T_{Y+\epsilon H}(\mathrm{Stief}(n,k))$. While taking their difference would work fine in a flat space (see Figure 9), if this were done on a curved space, it could give a vector which is not a member of the tangent space of either point. (see Figure 10).

A more sophisticated means of taking this difference is to first move the gradient at $Y + \epsilon H$ to Y in some manner which translates it in a parallel fashion from $Y + \epsilon H$ to Y, and then compare the two gradients within the same tangent space. One can check that for $V \in T_{Y+\epsilon H}(\mathrm{Stief}(n,k))$ the rule

$$V \to V + \epsilon\Gamma(V, H),$$

where $\Gamma(V, H)$ is the Levi-Civita connection, takes V to an element of $T_Y(\mathrm{Stief}(n,k))$ and preserves inner product information (to first order in ϵ). This is the standard rule for parallel transport which can found in the usual literature ([5, 20, 13, 12], and others).

Using this rule to compare nearby vectors to each other, one then has the

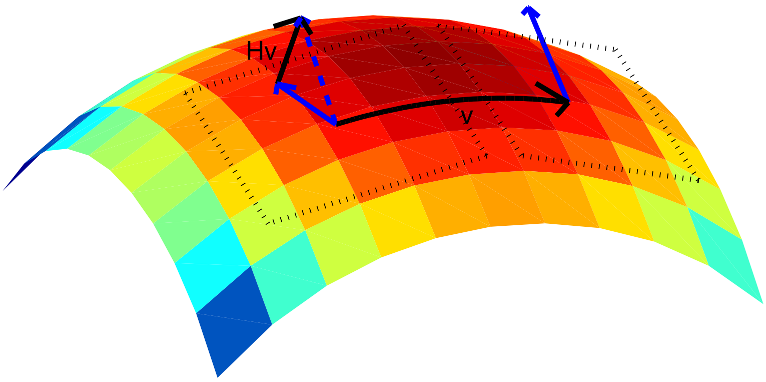

FIGURE 8.10: *In a curved manifold, comparing vectors at nearby points can result in vectors which do not lie in the tangent space.*

following rule for taking derivatives of vector fields:

$$D_H G = \frac{d}{ds} G(Y + sH)|_{s=0} + \Gamma(G, H),$$

where G is any vector field (but we are only interested in derivatives of the gradient field). This is the function implemented by `dgrad` in the software.

In an unconstrained minimization, the second derivative of the gradient $g = \nabla f$ along a vector $\vec{h}$ is the Hessian $[\frac{\partial^2 f}{\partial x_i \partial x_j}]$ times $\vec{h}$. Covariantly, we then have the analogy,

$$[\frac{\partial^2 f}{\partial x_i \partial x_j}]\vec{h} = (\vec{h} \cdot \nabla) g \sim D_H G.$$

8.3.6.6 Inverting the Covariant Hessian (technical considerations)

Since the tangent space of Stief(n, k) is a subspace of $\mathcal{R}^{n \times k}$, the covariant Hessian must be inverted stably on this subspace. This requires any algorithms designed to solve $D_H G = V$ for H to really be pseudoinverters in a least squares or some other sense.

A second consideration is that many useful functions $f(Y)$ have the property that $f(YQ) = f(Y)$ for all block diagonal orthogonal Q (i.e.

$$Q = \begin{bmatrix} Q_1 & 0 & \cdots & 0 \\ 0 & Q_2 & \cdots & 0 \\ .. & .. & .. & .. \\ 0 & 0 & \cdots & Q_p \end{bmatrix},$$

where the Q_i are orthogonal matrices). In this case, all tangent vectors of the form

$$H = Y \begin{bmatrix} H_1 & 0 & \cdots & 0 \\ 0 & H_2 & \cdots & 0 \\ .. & .. & .. & .. \\ 0 & 0 & \cdots & H_p \end{bmatrix},$$

where the H_i are antisymmetric, have the property $D_H f = 0$. These extra symmetry vectors are then null vectors of the linear system $D_H G = V$. Because of these null directions, the effective dimension of the tangent space is reduced by the dimension of this null space (this is the dimension returned by the `dimension` function).

Thus, we see that in order to invert the covariant Hessian, we must take care to use a stable inversion scheme which will project out components of H which do not satisfy the infinitesimal constraint equation and those which are in the direction of the additional symmetries of f. The `invdgrad` function carries out a stable inversion of the covariant Hessian by a conjugate gradient routine, with the `dgrad` function calling the function `nosym` to project out any extra symmetry components.

Acknowledgments (we shall have an Acknowledgment section at the beginning of the book): The authors would like to thank Steve Smith for many invaluable discussions.

Bibliography

[1] T.A. Arias A. Edelman and S.T. Smith. The geometry of algorithms with orthogonality constraints. *SIAM Journal on Matrix Analysis and Applications*, to appear.

[2] I. Andersson. Experiments with the conjugate gradient algorithm for the determination of eigenvalues of symmetric matrices. Technical Report UMINF-4.71, University of Umeå, Sweden, 1971.

[3] T. A. Arias, A. Edelman, and S. T. Smith. Curvature in conjugate gradient eigenvalue computation with applications to materials and chemistry calculations. In J. G. Lewis, editor, *Proceedings of the 1994 SIAM Applied Linear Algebra Conference*, pages 233–238, Philadephia, 1994. SIAM.

[4] M. Arioli, I. S. Duff, and D. Ruiz. Stopping criteria for iterative solvers. Report RAL-91-057, Central Computing Center, Rutherford Appleton Laboratory, 1992.

[5] I. Chavel. *Riemannian Geometry—A Modern Introduction*. The Cambridge University Press, 1993.

[6] J. de Leeuw and W. Heiser. Theory of multidimensional scaling. In P. R. Krishnaiah and L. N. Kanal, editors, *Handbook of Statistics, Vol 2*, pages 285–316. North-Holland Publishing Co., 1982.

[7] A. Edelman and S. T. Smith. On conjugate gradient-like methods for eigen-like problems. *BIT*, to appear. See also *Proc. Linear and Nonlinear Conjugate Gradient-Related Methods*, eds. Loyce Adams and J. L. Nazareth. SIAM, Philadelphia, PA, 1996.

[8] L. Eldén. Algorithms for the regularization of ill-conditioned least-squares problems. *BIT*, 17:134–145, 1977.

[9] R. Fletcher. *Practical Methods of Optimization*. John Wiley & Sons, New York, 2d edition, 1987.

[10] Z. Fu and E. M. Dowling. Conjugate gradient eigenstructure tracking for adaptive spectral estimation. *IEEE Trans. Signal Processing*, 43(5):1151–1160, 1995.

[11] P. E. Gill, W. Murray, and M. H. Wright. *Practical Optimization*. Academic Press, New York, 2d edition, 1981.

[12] S. Helgason. *Differential Geometry, Lie Groups, and Symmetric Spaces.* Academic Press, New York, 1978.

[13] S. Kobayashi and K. Nomizu. *Foundations of Differential Geometry*, volume 1 and 2. Wiley, New York, 1969.

[14] S. G. Nash and Ariela Sofer. *Linear and Nonlinear Programming.* McGraw-Hill, New York, 1995.

[15] E. Polak. *Computational Methods in Optimization.* Academic Press, New York, 1971.

[16] A. H. Sameh and J. A. Wisniewski. A trace minimization algorithm for the generalized eigenvalue problem. *SIAM J. Numerical Analysis*, 19:1243–1259, 1982.

[17] W. H. A. Schilders. Personal communication, September 1997.

[18] S. T. Smith. Optimization techniques on Riemannian manifolds. *Fields Institute Communications*, 3:113–146, 1994.

[19] I. Štich, R. Car, M. Parrinello, and S. Baroni. Conjugate gradient minimization of the energy functional: A new method for electronic structure calculation. *Phys. Rev. B.*, 39:4997–5004, 1989.

[20] Y.-C. Wong. Differential geometry of Grassmann manifolds. *Proc. Natl. Acad. Sci. USA*, 57:589–594, 1967.