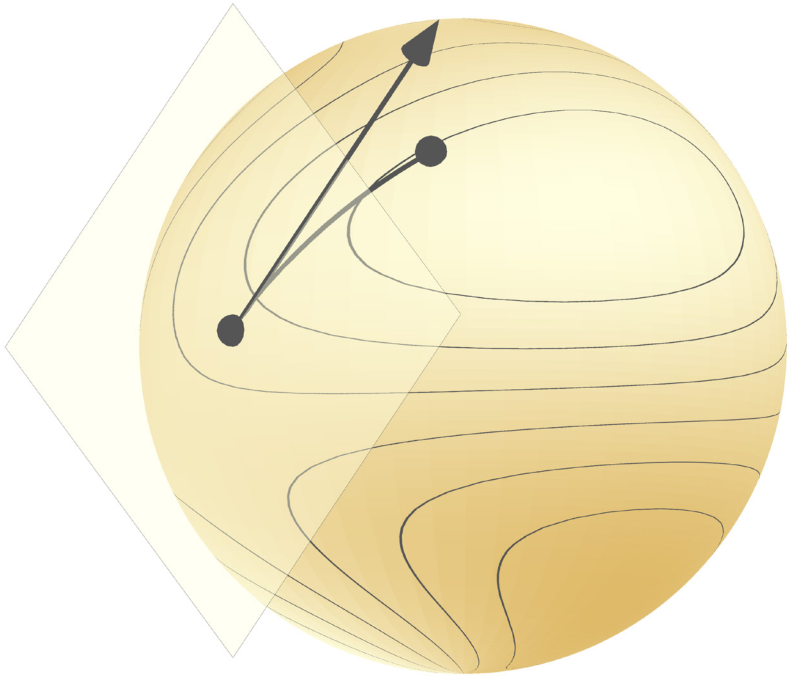
Hands on with Manopt

A toolbox for optimization on manifolds

Nicolas Boumal – OPTIM

Chair of Continuous Optimization

Institute of Mathematics, EPFL



Manopt is a toolbox to solve:

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

where $f : \mathcal{M} \to \mathbf{R}$ is defined on a smooth manifold $\mathcal{M}$.

This presentation is about the Matlab version.

Latest code on GitHub: github.com/NicolasBoumal/manopt.

Website with numbered releases, tutorial, forum, contributors list: manopt.org.

In particular, see the tutorial: manopt.org/tutorial.html.

There are also Python and Julia versions led by other teams. Find them at manopt.org.

If you want to play along:

Get Manopt:
https://www.manopt.org/downloads.html (unzip somewhere) or
https://github.com/NicolasBoumal/manopt (clone somewhere).

In Matlab, go to the code folder and run `importmanopt`.
You can `savepath` so you don't need to import in future Matlab sessions.

Go to `/checkinstall` (may be in `/auxiliaries`) and run `basicexample.m`.
Let me know if errors.

What kind of cost functions?

The toolbox handles differentiable cost functions best.

For nonsmooth f, smooth approximations are often a good option:

$$|x| \approx \sqrt{x^2 + \epsilon^2}, \max(x, y) \approx \epsilon \log\left(e^{x/\epsilon} + e^{y/\epsilon}\right), \max(0, x) \approx \epsilon \log\left(1 + e^{x/\epsilon}\right)$$

The toolbox *can* accommodate nonsmooth solvers, but we do not have a good general purpose nonsmooth solver yet. Contributors welcome.

What kinds of manifolds?

Smooth manifolds endowed with a Riemannian structure.

Needs to be explicit enough that we know how to:

- Numerically represent points on the manifold
- Numerically represent tangent vectors on the manifold
- Move around the manifold following tangent vectors (retraction)
- Use the Riemannian structure to compute gradients, Hessians, ...

This is all encoded in a manifold structure created by a factory.

Examples to follow; handle products of manifolds with:
`productmanifold` and `powermanifold`.

General principle of the toolbox

To specify an optimization problem "$\min_{x \in \mathcal{M}} f(x)$", you need to

1. Pick a manifold $\mathcal{M}$ (with a Riemannian structure) using a factory,
2. Describe the cost f and (ideally) its derivatives using function handles.

This description of the problem is stored in a problem structure.

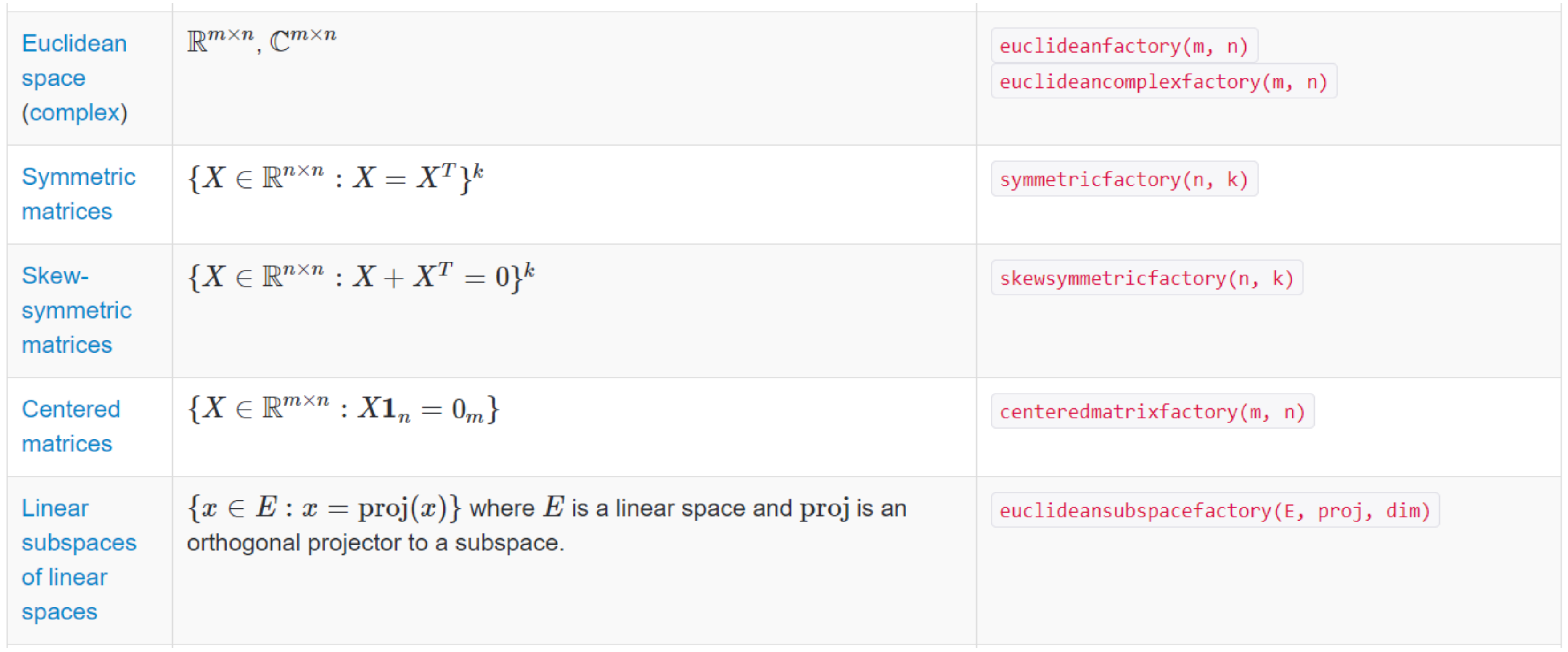
That structure is fed to a solver (an optimization algorithm).

E.g.: `steepestdescent`, `conjugategradient`, `trustregions`, `arc`, `barzilaiborwein`, `rlbfgs`, `stochasticgradient`, `pso`, `neldermead`.

What kinds of manifolds?

Smooth manifolds endowed with a Riemannian structure. E.g.:

Euclidean spaces (matrix spaces, both real and complex)

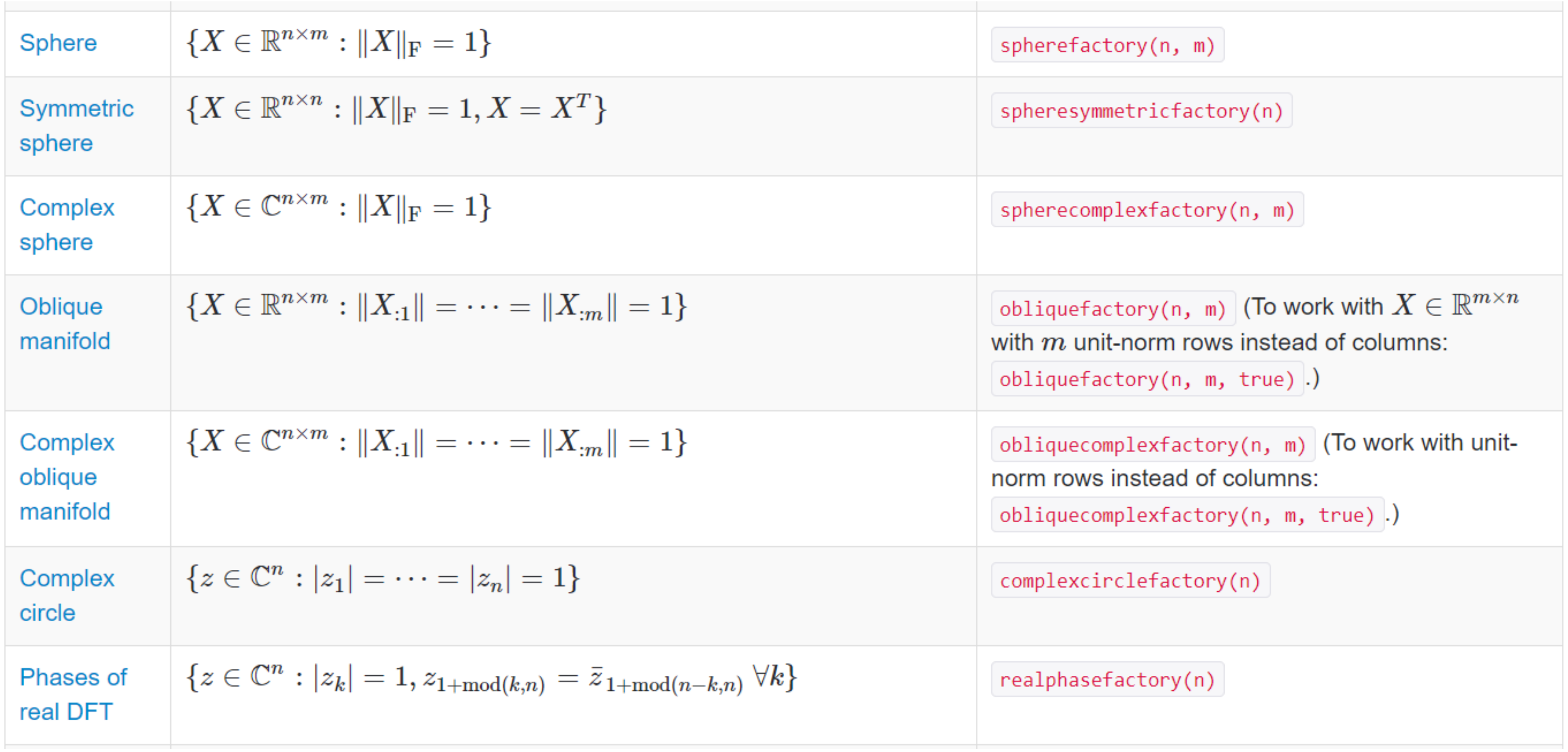


Euclidean space (complex)	$\mathbb{R}^{m\times n}$, $\mathbb{C}^{m\times n}$	`euclideanfactory(m, n)` `euclideancomplexfactory(m, n)`
Symmetric matrices	$\{X \in \mathbb{R}^{n\times n} : X = X^T\}^k$	`symmetricfactory(n, k)`
Skew-symmetric matrices	$\{X \in \mathbb{R}^{n\times n} : X + X^T = 0\}^k$	`skewsymmetricfactory(n, k)`
Centered matrices	$\{X \in \mathbb{R}^{m\times n} : X\mathbf{1}_n = 0_m\}$	`centeredmatrixfactory(m, n)`
Linear subspaces of linear spaces	$\{x \in E : x = \mathrm{proj}(x)\}$ where E is a linear space and proj is an orthogonal projector to a subspace.	`euclideansubspacefactory(E, proj, dim)`

What kinds of manifolds?

Smooth manifolds endowed with a Riemannian structure. E.g.:

Spheres, phases (unit-norm constraints over vectors, matrices)

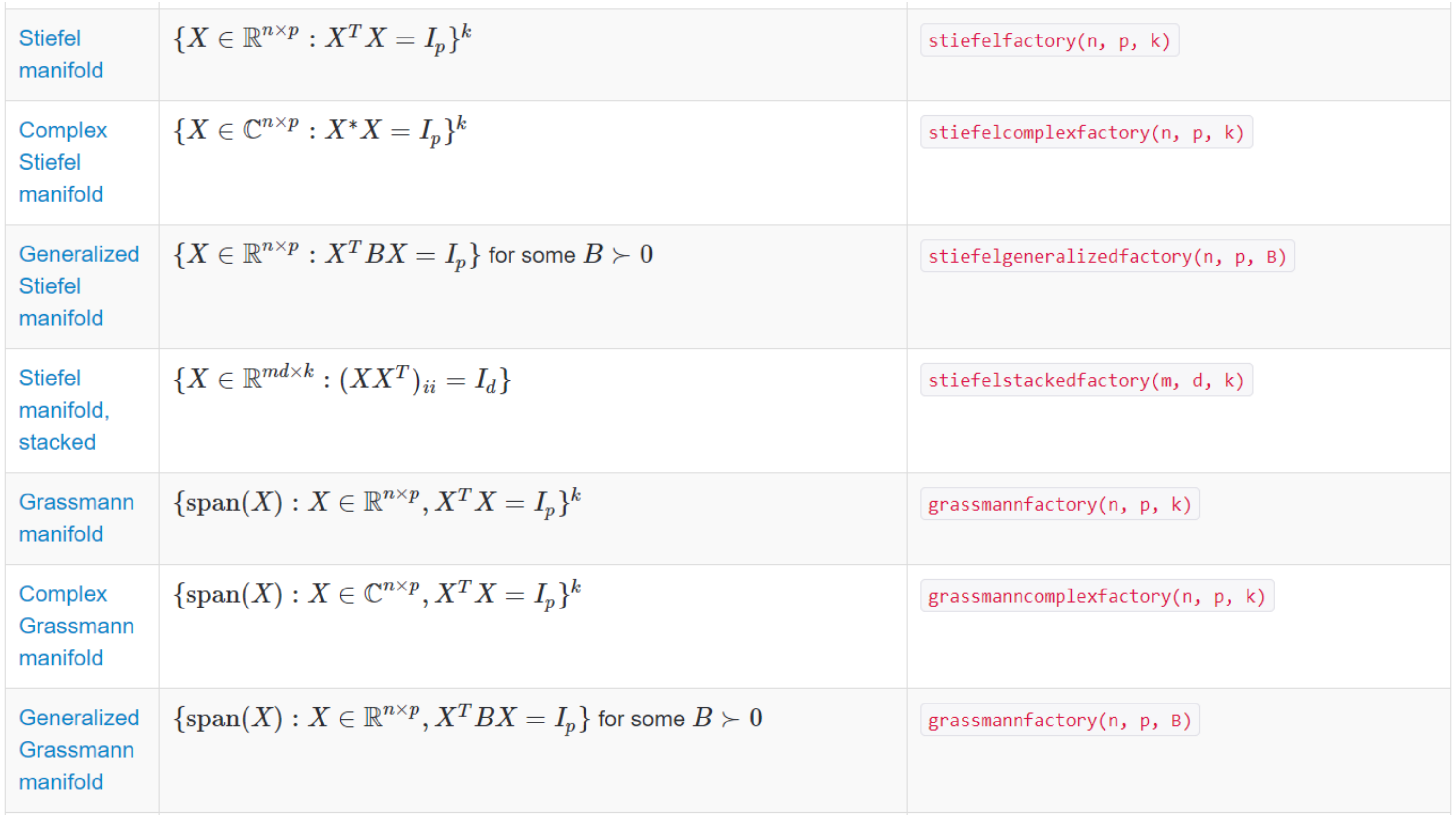


Sphere	$\{X \in \mathbb{R}^{n\times m} : \|X\|_\mathrm{F} = 1\}$	`spherefactory(n, m)`
Symmetric sphere	$\{X \in \mathbb{R}^{n\times n} : \|X\|_\mathrm{F} = 1, X = X^T\}$	`spheresymmetricfactory(n)`
Complex sphere	$\{X \in \mathbb{C}^{n\times m} : \|X\|_\mathrm{F} = 1\}$	`spherecomplexfactory(n, m)`
Oblique manifold	$\{X \in \mathbb{R}^{n\times m} : \|X_{:1}\| = \cdots = \|X_{:m}\| = 1\}$	`obliquefactory(n, m)` (To work with $X \in \mathbb{R}^{m\times n}$ with m unit-norm rows instead of columns: `obliquefactory(n, m, true)`.)
Complex oblique manifold	$\{X \in \mathbb{C}^{n\times m} : \|X_{:1}\| = \cdots = \|X_{:m}\| = 1\}$	`obliquecomplexfactory(n, m)` (To work with unit-norm rows instead of columns: `obliquecomplexfactory(n, m, true)`.)
Complex circle	$\{z \in \mathbb{C}^{n} : \|z_1\| = \cdots = \|z_n\| = 1\}$	`complexcirclefactory(n)`
Phases of real DFT	$\{z \in \mathbb{C}^{n} : \|z_k\| = 1, z_{1+\mathrm{mod}(k,n)} = \bar{z}_{1+\mathrm{mod}(n-k,n)}\ \forall k\}$	`realphasefactory(n)`

What kinds of manifolds?

Smooth manifolds endowed with a Riemannian structure. E.g.:

Stiefel, Grassmann (orthonormal bases, linear subspaces)

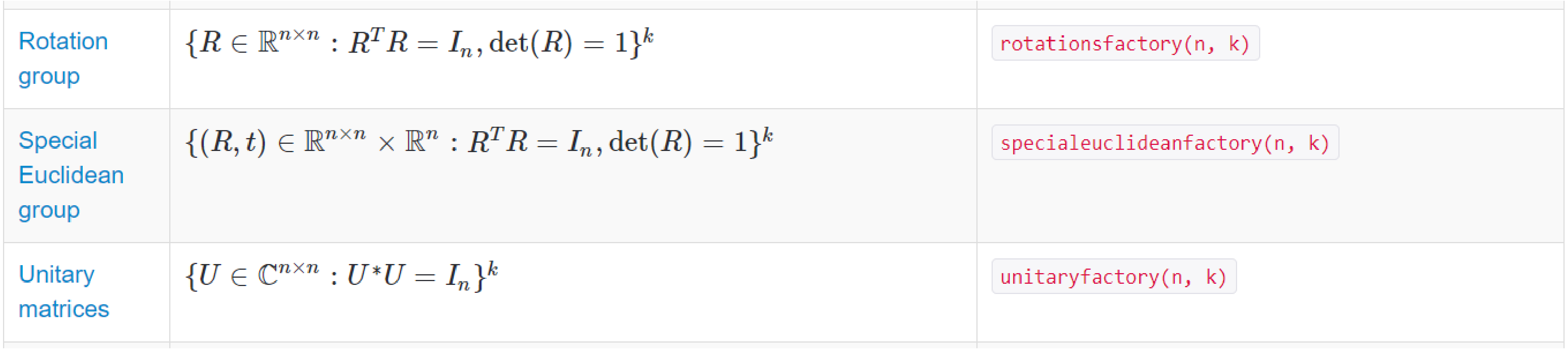


Stiefel manifold	$\{X \in \mathbb{R}^{n\times p} : X^TX = I_p\}^k$	`stiefelfactory(n, p, k)`
Complex Stiefel manifold	$\{X \in \mathbb{C}^{n\times p} : X^*X = I_p\}^k$	`stiefelcomplexfactory(n, p, k)`
Generalized Stiefel manifold	$\{X \in \mathbb{R}^{n\times p} : X^TBX = I_p\}$ for some $B \succ 0$	`stiefelgeneralizedfactory(n, p, B)`
Stiefel manifold, stacked	$\{X \in \mathbb{R}^{md\times k} : (XX^T)_{ii} = I_d\}$	`stiefelstackedfactory(m, d, k)`
Grassmann manifold	$\{\mathrm{span}(X) : X \in \mathbb{R}^{n\times p}, X^TX = I_p\}^k$	`grassmannfactory(n, p, k)`
Complex Grassmann manifold	$\{\mathrm{span}(X) : X \in \mathbb{C}^{n\times p}, X^TX = I_p\}^k$	`grassmanncomplexfactory(n, p, k)`
Generalized Grassmann manifold	$\{\mathrm{span}(X) : X \in \mathbb{R}^{n\times p}, X^TBX = I_p\}$ for some $B \succ 0$	`grassmannfactory(n, p, B)`

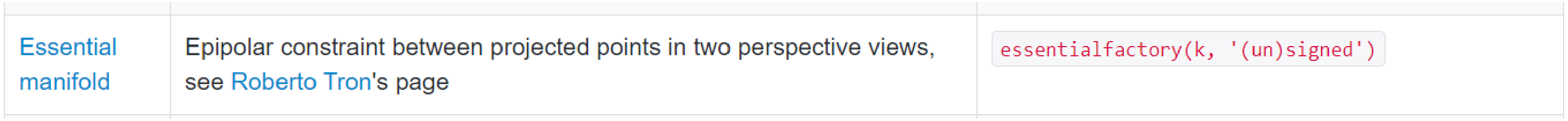
What kinds of manifolds?

Smooth manifolds endowed with a Riemannian structure. E.g.:

Orthogonal group, rotations, rigid motions (and other CV manifolds)



Rotation group	$\{R \in \mathbb{R}^{n\times n} : R^T R = I_n, \det(R) = 1\}^k$	`rotationsfactory(n, k)`
Special Euclidean group	$\{(R,t) \in \mathbb{R}^{n\times n} \times \mathbb{R}^n : R^T R = I_n, \det(R) = 1\}^k$	`specialeuclideanfactory(n, k)`
Unitary matrices	$\{U \in \mathbb{C}^{n\times n} : U^* U = I_n\}^k$	`unitaryfactory(n, k)`

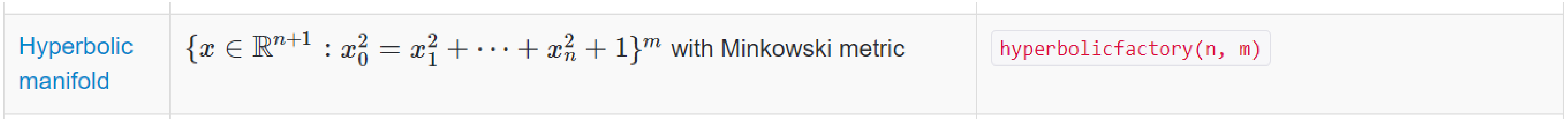


Essential manifold	Epipolar constraint between projected points in two perspective views, see Roberto Tron's page	`essentialfactory(k, '(un)signed')`

What kinds of manifolds?

Smooth manifolds endowed with a Riemannian structure. E.g.:

Hyperbolic space

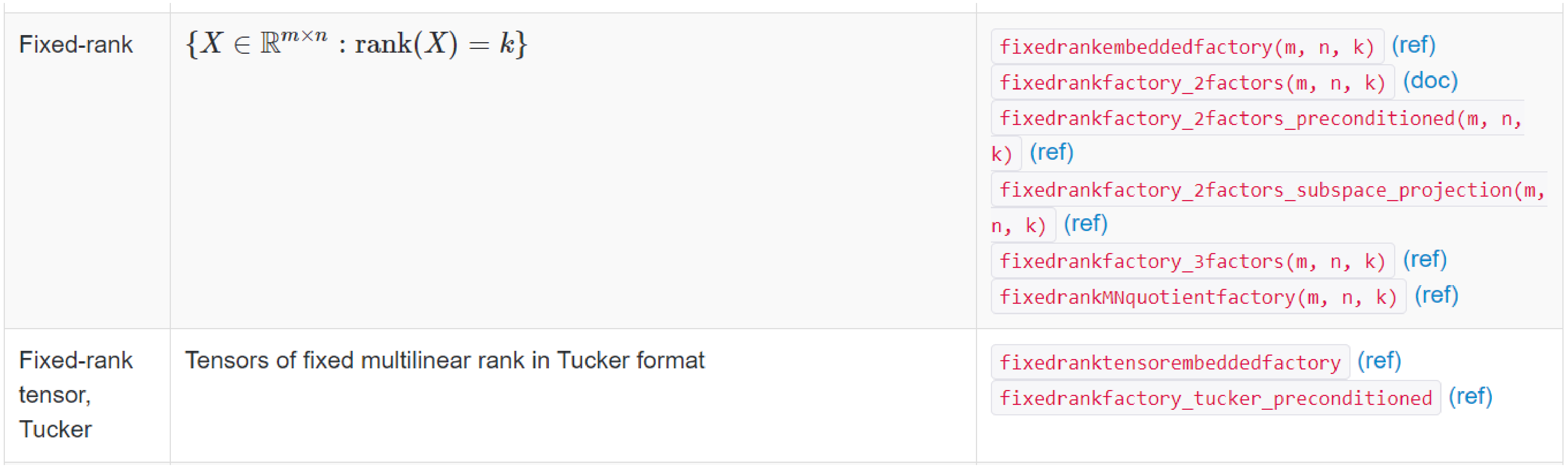


Hyperbolic manifold	$\{x \in \mathbb{R}^{n+1} : x_0^2 = x_1^2 + \cdots + x_n^2 + 1\}^m$ with Minkowski metric	`hyperbolicfactory(n, m)`

What kinds of manifolds?

Smooth manifolds endowed with a Riemannian structure. E.g.:

Fixed-rank matrices and tensors

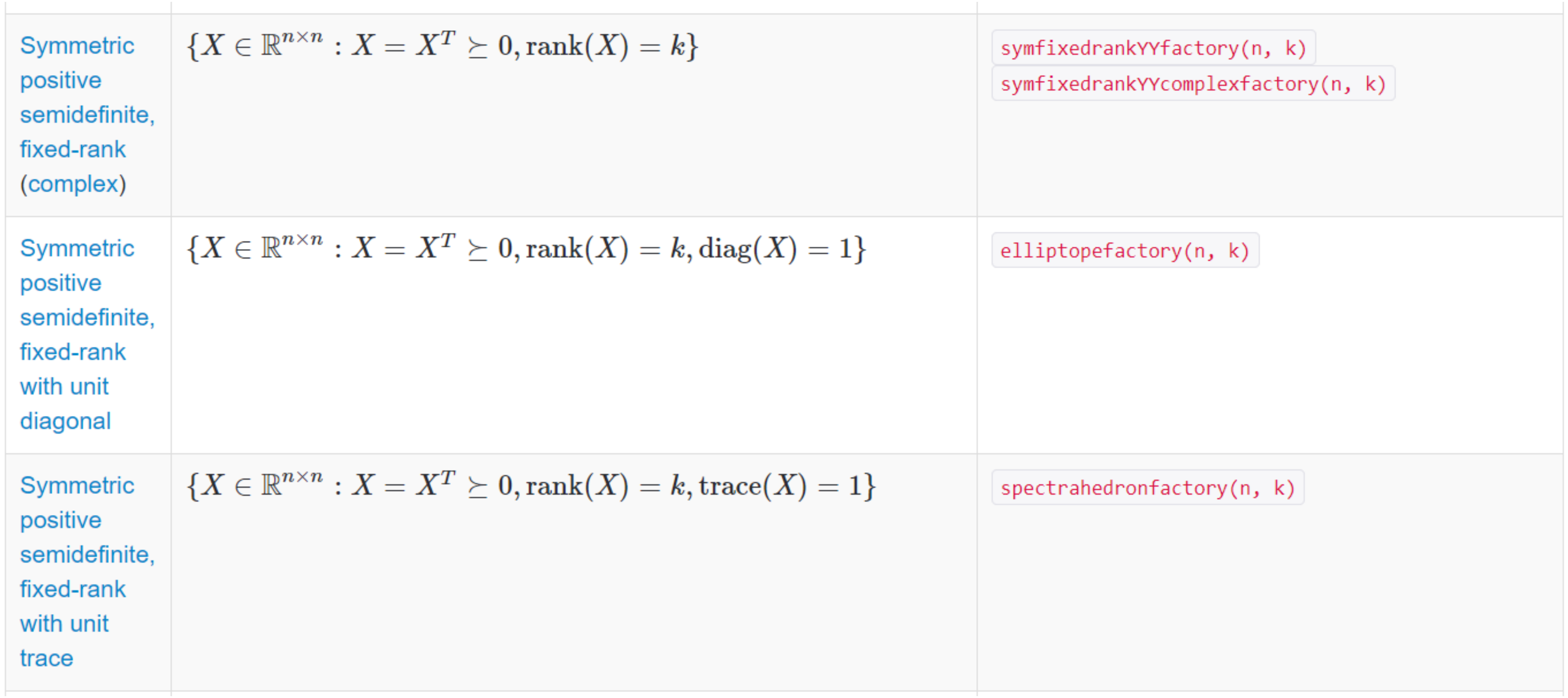


Fixed-rank	$\{X \in \mathbb{R}^{m \times n} : \operatorname{rank}(X) = k\}$	`fixedrankembeddedfactory(m, n, k)` (ref) `fixedrankfactory_2factors(m, n, k)` (doc) `fixedrankfactory_2factors_preconditioned(m, n, k)` (ref) `fixedrankfactory_2factors_subspace_projection(m, n, k)` (ref) `fixedrankfactory_3factors(m, n, k)` (ref) `fixedrankMNquotientfactory(m, n, k)` (ref)
Fixed-rank tensor, Tucker	Tensors of fixed multilinear rank in Tucker format	`fixedranktensorembeddedfactory` (ref) `fixedrankfactory_tucker_preconditioned` (ref)

What kinds of manifolds?

Smooth manifolds endowed with a Riemannian structure. E.g.:

Fixed-rank matrices, positive semidefinite

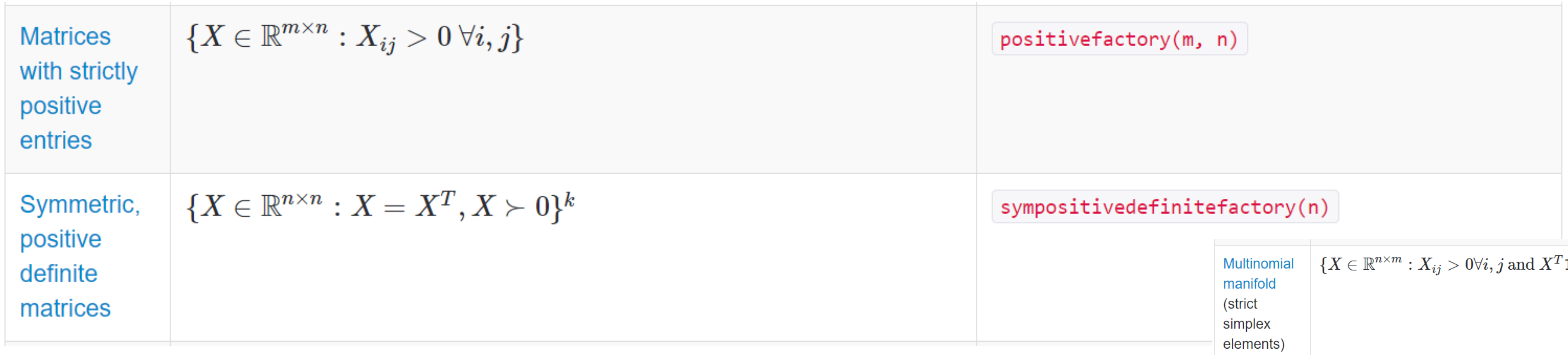


Symmetric positive semidefinite, fixed-rank (complex)	$\{X \in \mathbb{R}^{n\times n} : X = X^T \succeq 0, \mathrm{rank}(X) = k\}$	`symfixedrankYYfactory(n, k)` `symfixedrankYYcomplexfactory(n, k)`
Symmetric positive semidefinite, fixed-rank with unit diagonal	$\{X \in \mathbb{R}^{n\times n} : X = X^T \succeq 0, \mathrm{rank}(X) = k, \mathrm{diag}(X) = 1\}$	`elliptopefactory(n, k)`
Symmetric positive semidefinite, fixed-rank with unit trace	$\{X \in \mathbb{R}^{n\times n} : X = X^T \succeq 0, \mathrm{rank}(X) = k, \mathrm{trace}(X) = 1\}$	`spectrahedronfactory(n, k)`

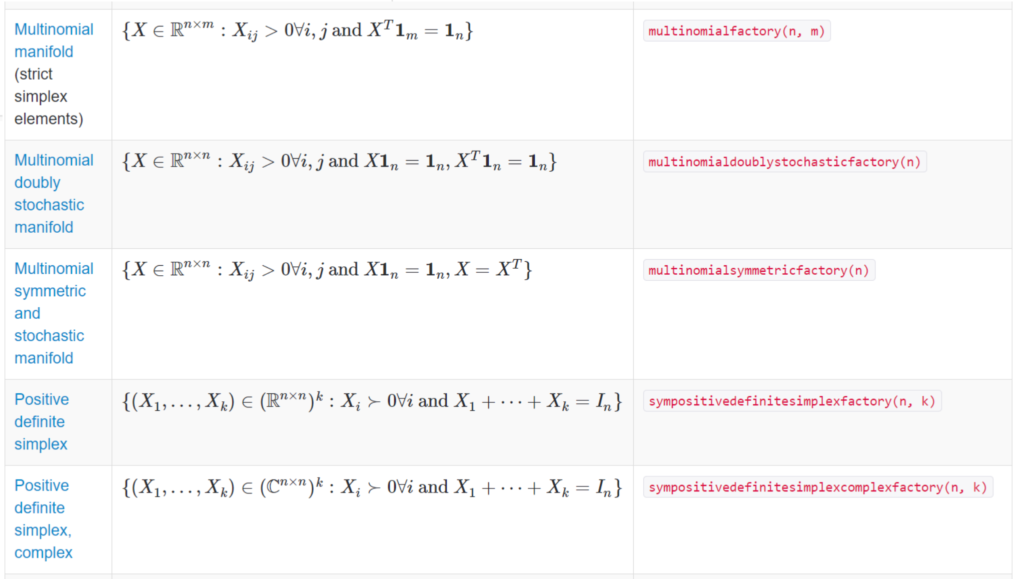
What kinds of manifolds?

Smooth manifolds endowed with a Riemannian structure. E.g.:

Positive definite matrices, matrices with positive entries



Matrices with strictly positive entries	$\{X \in \mathbb{R}^{m\times n} : X_{ij} > 0 \; \forall i, j\}$	`positivefactory(m, n)`
Symmetric, positive definite matrices	$\{X \in \mathbb{R}^{n\times n} : X = X^T, X \succ 0\}^k$	`sympositivedefinitefactory(n)`



Multinomial manifold (strict simplex elements)	$\{X \in \mathbb{R}^{n\times m} : X_{ij} > 0 \forall i, j \text{ and } X^T\mathbf{1}_m = \mathbf{1}_n\}$	`multinomialfactory(n, m)`
Multinomial doubly stochastic manifold	$\{X \in \mathbb{R}^{n\times n} : X_{ij} > 0 \forall i, j \text{ and } X\mathbf{1}_n = \mathbf{1}_n, X^T\mathbf{1}_n = \mathbf{1}_n\}$	`multinomialdoublystochasticfactory(n)`
Multinomial symmetric and stochastic manifold	$\{X \in \mathbb{R}^{n\times n} : X_{ij} > 0 \forall i, j \text{ and } X\mathbf{1}_n = \mathbf{1}_n, X = X^T\}$	`multinomialsymmetricfactory(n)`
Positive definite simplex	$\{(X_1, \ldots, X_k) \in (\mathbb{R}^{n\times n})^k : X_i \succ 0 \forall i \text{ and } X_1 + \cdots + X_k = I_n\}$	`sympositivedefinitesimplexfactory(n, k)`
Positive definite simplex, complex	$\{(X_1, \ldots, X_k) \in (\mathbb{C}^{n\times n})^k : X_i \succ 0 \forall i \text{ and } X_1 + \cdots + X_k = I_n\}$	`sympositivedefinitesimplexcomplexfactory(n, k)`

What’s in a factory-produced manifold?

Example: stripped down and simplified spherefactory

```matlab
function M = spherefactory(n)
    M.name = @() sprintf('Sphere S^%d', n-1);
    M.dim = @() n-1;
    M.inner = @(x, u, v) u'*v;
    M.norm = @(x, u) norm(u);
    M.dist = @(x, y) real(2*asin(.5*norm(x - y)));
    M.typicaldist = @() pi;
    M.proj = @(x, u) u - x*(x'*u);
    M.tangent = M.proj;
    M.tangent2ambient_is_identity = true;
    M.tangent2ambient = @(x, u) u;
    M.egrad2rgrad = M.proj;
    M.ehess2rhess = @(x, egrad, ehess, u) ...
                        M.proj(x, ehess - (x'*egrad)*u);
    M.exp = @exponential;
    M.retr = @(x, u) (x+u)/norm(x+u);
    M.invretr = @inverse_retraction;
    M.log = @logarithm;
    M.hash = @(x) ['z' hashmd5(x)];
    M.rand = @() ...;
    M.randvec = @(x) ...;
    M.zerovec = @(x) zeros(n, 1);
    M.lincomb = @matrixlincomb;
    M.transp = @(x, y, u) M.proj(y, u);
    M.isotransp = @(x, y, u) ...;
    M.pairmean = @pairmean;
    M.vec = @(x, u_mat) u_mat;
    M.mat = @(x, u_vec) reshape(u_vec, [n, 1]);
    M.vecmatareisometries = @() true;
end
```

First example: let’s dive right in

Given a symmetric $A \in \mathbf{R}^{n \times n}$, we want:

$$\max_{x \in \mathbf{R}^n} x^\top A x \;\text{ subject to } \|x\| = 1$$

The manifold is the sphere:

$$\mathcal{M} = S^{n-1} = \{x \in \mathbf{R}^n : \|x\| = 1\}$$

The cost function *to minimize* is:

$$f(x) = -x^\top A x$$

```
n = 20;
A = randn(n);
A = A+A';

problem.M = spherefactory(n);

problem.cost = @(x) -x'*A*x;

x = steepestdescent(problem);

x'*A*x, max(eig(A)) % compare
```

Run steepest descent

```
n = 20;
A = randn(n); A = A+A';
problem.M = spherefactory(n);
problem.cost = @(x) -x'*A*x;

x = steepestdescent(problem);
x'*A*x, max(eig(A)) % compare
```

```
Warning: No gradient provided. Using an FD approximation (slow).
It may be necessary to increase options.tolgradnorm.
To disable this warning: warning('off','manopt:getGradient:approx')

iter                    cost val         grad. norm
    0  -6.8964471239672420e-01    1.11661942e+01
    1  -7.2433166754551808e+00    6.93446878e+00
    2  -8.8747349682088839e+00    1.07491407e+01
...................
   63  -1.3416668060076006e+01    7.17861797e-06
   64  -1.3416668060076006e+01    7.17861797e-06
Last stepsize smaller than minimum allowed; options.minstepsize = 1e-10.
Total time is 0.206028 [s] (excludes statsfun)

ans = 13.4167 % x'*A*x
ans = 13.4167 % max(eig(A))
```

Provide Euclidean gradient

```
n = 20;
A = randn(n); A = A+A';
problem.M = spherefactory(n);
problem.cost = @(x) -x'*A*x;
problem.egrad = @(x) -2*A*x;
x = steepestdescent(problem);
x'*A*x, max(eig(A)) % compare
```

```
 iter                   cost val            grad. norm
    0  +3.6273059865027202e-01      1.25950958e+01
    1  -4.5684590336508029e+00      8.28195678e+00
    2  -4.5851972109302022e+00      1.51110110e+01
..............................
   64  -1.2223151484178318e+01      4.47433851e-06
   65  -1.2223151484178524e+01      9.38151399e-07

Gradient norm tolerance reached; options.tolgradnorm = 1e-06.
Total time is 0.043975 [s] (excludes statsfun)

ans = 12.2232 % x'*A*x
ans = 12.2232 % max(eig(A))
```

Run trust-regions

```
n = 20;
A = randn(n); A = A+A';
problem.M = spherefactory(n);
problem.cost = @(x) -x'*A*x;
problem.egrad = @(x) -2*A*x;

x = trustregions(problem);
x'*A*x, max(eig(A)) % compare
```

```
Warning: No Hessian provided. Using an FD approximation instead.
To disable this warning: warning('off', 'manopt:getHessian:approx')
> In trustregions (line 325)
In Tutorial01_rayleigh (line 14)
                                                f: -1.359943e+00   |grad|: 1.344318e+01
acc TR+   k:     1     num_inner:     1     f: -5.988531e+00   |grad|: 1.079847e+01   negative curvature
acc       k:     2     num_inner:     1     f: -9.152545e+00   |grad|: 6.683919e+00   exceeded trust region
acc       k:     3     num_inner:     3     f: -1.037445e+01   |grad|: 2.611111e+00   exceeded trust region
acc       k:     4     num_inner:     4     f: -1.063347e+01   |grad|: 2.183730e-01   reached target residual-kappa (linear)
acc       k:     5     num_inner:     5     f: -1.063569e+01   |grad|: 1.377073e-02   reached target residual-kappa (linear)
acc       k:     6     num_inner:     7     f: -1.063570e+01   |grad|: 1.112603e-04   reached target residual-theta (superlinear)
acc       k:     7     num_inner:    12     f: -1.063570e+01   |grad|: 9.671030e-09   reached target residual-theta (superlinear)
Gradient norm tolerance reached; options.tolgradnorm = 1e-06.
Total time is 0.042530 [s] (excludes statsfun)

ans = 10.6357 % x'*A*x
ans = 10.6357 % max(eig(A))
```

Run trust-regions

```
n = 20;
A = randn(n); A = A+A';
problem.M = spherefactory(n);
problem.cost = @(x) -x'*A*x;
problem.egrad = @(x) -2*A*x;

x = trustregions(problem);
x'*A*x, max(eig(A)) % compare
```

```
Warning: No Hessian provided. Using an FD approximation instead.
To disable this warning: warning('off', 'manopt:getHessian:approx')
> In trustregions (line 325)
In Tutorial01_rayleigh (line 14)
                                                f: -1.359943e+00   |grad|: 1.344318e+01
acc TR+   k:     1    num_inner:     1     f: -5.988531e+00   |grad|: 1.079847e+01
acc       k:     2    num_inner:     1     f: -9.152545e+00   |grad|: 6.683919e+00
acc       k:     3    num_inner:     3     f: -1.037445e+01   |grad|: 2.611111e+00
acc       k:     4    num_inner:     4     f: -1.063347e+01   |grad|: 2.183730e-01
acc       k:     5    num_inner:     5     f: -1.063569e+01   |grad|: 1.377073e-02
acc       k:     6    num_inner:     7     f: -1.063570e+01   |grad|: 1.112603e-04
acc       k:     7    num_inner:    12     f: -1.063570e+01   |grad|: 9.671030e-09
Gradient norm tolerance reached; options.tolgradnorm = 1e-06.
Total time is 0.042530 [s] (excludes statsfun)

ans = 10.6357 % x'*A*x
ans = 10.6357 % max(eig(A))
```

Provide Euclidean Hessian

```
n = 20;
A = randn(n); A = A+A';
problem.M = spherefactory(n);
problem.cost = @(x) -x'*A*x;
problem.egrad = @(x) -2*A*x;
problem.ehess = @(x, u) -2*A*u;
x = trustregions(problem);
x'*A*x, max(eig(A)) % compare
```

```
                                               f: +1.085417e+00   |grad|: 9.168786e+00
acc TR+   k:     1     num_inner:     1     f: -2.028789e+00   |grad|: 8.556623e+00
acc       k:     2     num_inner:     1     f: -6.919949e+00   |grad|: 7.271589e+00
acc       k:     3     num_inner:     1     f: -9.440813e+00   |grad|: 6.269407e+00
acc       k:     4     num_inner:     2     f: -1.052834e+01   |grad|: 3.971600e+00
acc       k:     5     num_inner:     8     f: -1.104792e+01   |grad|: 8.459219e-01
acc       k:     6     num_inner:     7     f: -1.114649e+01   |grad|: 6.471174e-02
acc       k:     7     num_inner:     7     f: -1.114702e+01   |grad|: 3.446551e-03
acc       k:     8     num_inner:    10     f: -1.114702e+01   |grad|: 1.036685e-05
acc       k:     9     num_inner:    15     f: -1.114702e+01   |grad|: 4.357143e-11
Gradient norm tolerance reached; options.tolgradnorm = 1e-06.
Total time is 0.015170 [s] (excludes statsfun)
```

Want solvers to be quieter? Set `options.verbosity = 0;` (or 1, 2, 3, …)

```
n = 20;
A = randn(n); A = A+A';
problem.M = spherefactory(n);
problem.cost = @(x) -x'*A*x;
problem.egrad = @(x) -2*A*x;
problem.ehess = @(x, u) -2*A*u;
```

Check the gradient

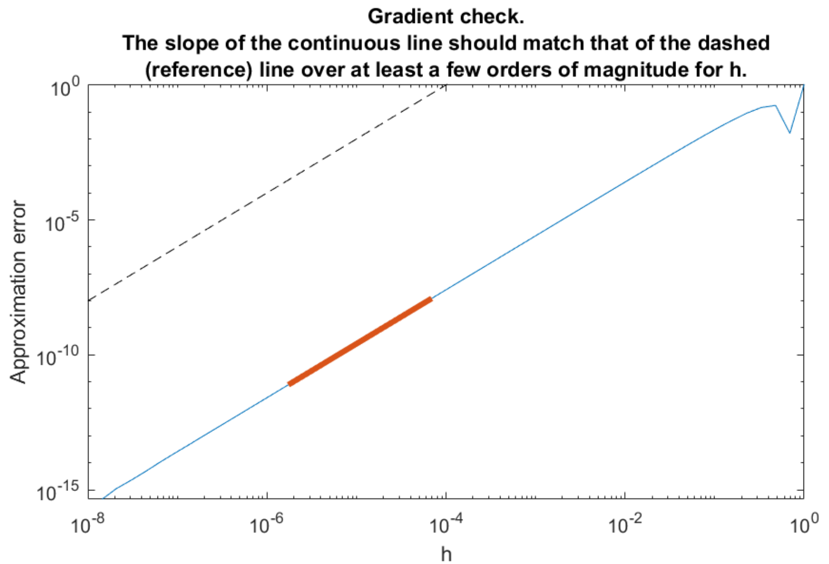
```
>> checkgradient(problem)

The slope should be 2. It appears to be: 2.00003.
If it is far from 2, then directional derivatives might be erroneous.
The residual should be 0, or very close. Residual: 2.80318e-16.
If it is far from 0, then the gradient is not in the tangent space.
In certain cases (e.g., hyperbolicfactory), the test is inconclusive.
```

How? Manopt generates a random $x \in \mathcal{M}$ and $u \in \mathrm{T}_x\mathcal{M}$, then checks that the error term in

$$f\big(\mathrm{R}_x(tu)\big) = f(x) + t\langle \mathrm{grad} f(x), u\rangle_x + O(t^2)$$

is indeed $O(t^2)$.



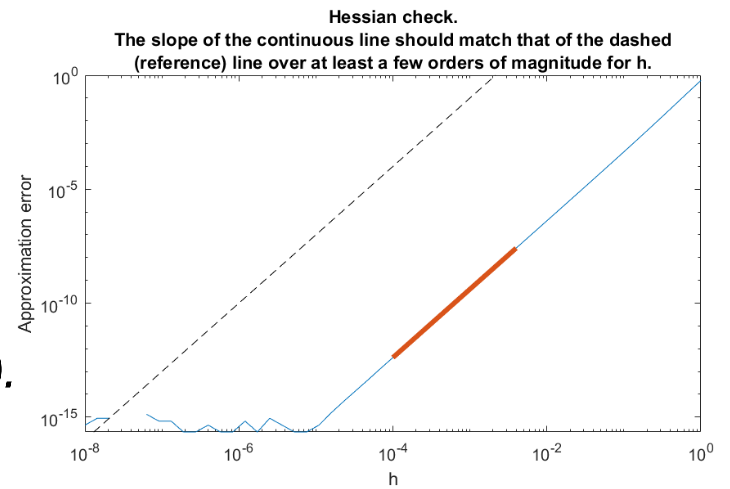
Check the Hessian

```
n = 20;
A = randn(n); A = A+A';
problem.M = spherefactory(n);
problem.cost = @(x) -x'*A*x;
problem.egrad = @(x) -2*A*x;
problem.ehess = @(x, u) -2*A*u;
```

```
>> checkhessian(problem)
```

```
The slope should be 3. It appears to be: 3.00074.
If it is far from 3, then directional derivatives,
the gradient or the Hessian might be erroneous.
Tangency residual should be zero, or very close; residual: 9.1425e-16.
If it is far from 0, then the Hessian is not in the tangent space.
||a*H[d1] + b*H[d2] - H[a*d1+b*d2]|| should be zero, or very close.
      Value: 2.20443e-15 (norm of H[a*d1+b*d2]: 8.96316)
If it is far from 0, then the Hessian is not linear.
<d1, H[d2]> - <H[d1], d2> should be zero, or very close.
      Value: 0.172729 - 0.172729 = -4.44089e-16.
If it is far from 0, then the Hessian is not symmetric.
```

How? Same principle, checking for $O(t^3)$ error term.
This only works if the retraction is second-order and/or if $\mathrm{grad} f(x) = 0$.



Calling solvers: general pattern

```
[x, cost, info, options] = myfavoritesolver(problem, x0, options)
```

Inputs:

- A problem structure
- Optional: initial point (by default: `M.rand()` to generate random x_0)
- Optional: a structure to specify options (stopping criteria, verbosity, ...)

Outputs:

- Best computed point
- Cost function value at returned point
- Information struct-array (time/function value/gradient norm/... at each iter.)
- Options structure (so you know which default values were used)

Things you find in the info struct-array

```
[x, cost, info, options] = myfavoritesolver(problem, x0, options)
```

It varies from solver to solver: see `help mysolver`.

Typical stuff: `iter, cost, time, gradnorm`. Try:

```
    plot([info.iter], [info.cost], '.-’);
    semilogy([info.time], [info.gradnorm] , '.-');
```

You can also record fancy original things using `options.statsfun`:

```
options.statsfun = @mystatsfun;
function stats = mystatsfun(problem, x, stats)
    stats.current_point = x;
end
% Results are stored in info(k).current_point for k = 0, 1, ...
```

```
problem.cost = @(x) -x'*A*x;
problem.egrad = @(x) -2*A*x;
```

Ways to define the cost and gradient

`f = problem.cost(x)`	$f(x)$
`g = problem.grad(x)`	grad $f(x)$ (Riemannian gradient)
`[f, g] = problem.costgrad(x)`	both together

Why define cost and gradient together? Reduce redundant computation:

```
problem.costgrad = @(x) mycostgrad(A, x);
function [f, g] = mycostgrad(A, x)
    Ax = A*x;             % Compute A*x once: O(n²) operations.
    f = -x'*Ax;
    g = -2*Ax;            % Compute Euclidean gradient, re-using Ax.
    g = g - (x'*g)*x;     % Convert to Riemannian gradient; see M.egrad2rgrad.
end
```

```
problem.cost = @(x) -x'*A*x;
problem.egrad = @(x) -2*A*x;
```

Ways to define the cost and gradient (2)

`f = problem.cost(x)`	$f(x)$
`g = problem.grad(x)`	grad $f(x)$ (Riemannian gradient)
`[f, g] = problem.costgrad(x)`	both together

If $\mathcal{M}$ is embedded in a Euclidean space, f can be smoothly extended to $\bar{f}$ around $\mathcal{M}$.
The toolbox can convert Euclidean gradients to Riemannian gradients automatically:

`eg = problem.egrad(x)`	grad $\bar{f}(x)$ (Euclidean gradient)

Under the hood, Manopt calls `g = M.egrad2rgrad(x, eg)` to convert.

There is no "`problem.costegrad`". Just use `cost+egrad`, or call `M.egrad2rgrad` manually, or use manual caching which is recommended anyway if you're going to use the Hessian.

```
problem.cost = @(x) -x'*A*x;
problem.egrad = @(x) -2*A*x;
problem.ehess = @(x, u) -2*A*u;
```

Ways to define the Hessian

`h = problem.hess(x, u)` $\mathrm{Hess} f(x)[u]$ (Riemannian Hessian as operator)

If $\mathcal{M}$ is embedded in a Euclidean space, f can be smoothly extended to $\bar{f}$ around $\mathcal{M}$.
The toolbox can convert Euclidean gradient+Hessian to Riemannian Hessian:

`eh = problem.ehess(x, u)` $\mathrm{Hess} \bar{f}(x)[u]$ (Euclidean Hessian)

Under the hood, Manopt calls `h = M.ehess2rhess(x, eg, eh, u)` to convert.

Note: `u` and `h` are tangent vectors whereas `eg` and `eh` are vectors in the ambient space.

Manual caching: store structures

Manopt does a fair amount of caching under the hood, but it can’t be as efficient as you telling it which intermediate results to store and how to use them.

Each new point an algorithm visits gets *its own* store structure.

```
function store = prepare(x, store)
    if ~isfield(store, 'Ax')
        store.Ax = A*x;
    end
end
```

```
function [f, store] = cost(x, store)
    store = prepare(x, store);
    Ax = store.Ax;
    f = -x'*Ax;
end
```

```
function [g, store] = egrad(x, store)
    store = prepare(x, store);
    Ax = store.Ax;
    g = -2*Ax;
end
```

```
function [h, store] = ehess(x, u, store)
    % store = prepare(x, store);
    % Not needed in this simple example.
    % Would be useful too in general.
    h = -2*A*u;
end
```

Moreover, `store.shared` contains memory shared by all *x*.

Counters: fancy ways to keep track

See `/examples/using_counters.m`

https://github.com/NicolasBoumal/manopt/blob/master/examples/using_counters.m

Counters allow you to keep track of events inside your code for f, grad$f$, Hess$f$.

Convenient to compare algorithms; can also serve in stopping criterion.

Code extracts:

```
stats = statscounters({'costcalls', 'gradcalls', 'hesscalls', ...
                       'Aproducts', 'functiontime'});
options.statsfun = statsfunhelper(stats);
store = incrementcounter(store, 'Aproducts');  % In cost, grad etc.
store = incrementcounter(store, 'functiontime', toc(t));
```

What happens if I omit the gradient?

Manopt automatically uses finite difference approximations and issues a warning.

You can silence the warning.

FD approximations of the gradient are *expensive* and reduce your final accuracy.

It's only good for quick prototyping. (PS: check automatic differentiation, manoptAD.)

How? `getGradientFD` generates a random orthonormal basis $u_1, \ldots, u_d$ of $\mathrm{T}_x\mathcal{M}$ with `tangentorthobasis` (not free!) and uses the formula $\mathrm{grad} f(x) = \sum_{i=1}^{d} \alpha_i u_i$ with

$$\alpha_i = \langle \mathrm{grad} f(x), u_i \rangle_x = \mathrm{D}f(x)[u_i] \approx \frac{f\left(\mathrm{R}_x(tu_i)\right) - f(x)}{t}$$

using a small value of t, specifically, 2^{-23}. Requires d calls to f and the retraction!

What happens if I omit the Hessian?

Manopt automatically uses finite difference approximations and issues a warning.

You can silence the warning.

FD approximations of the Hessian are *mostly harmless*.

They're often the same price as getting the true Hessian, and hardly hurt convergence.

How? `getHessianFD` uses vector transport $\mathrm{T}_{x\leftarrow y}$ from $\mathrm{T}_y\mathcal{M}$ to $\mathrm{T}_x\mathcal{M}$ with $y = \mathrm{R}_x(tu)$ (for example, Proj_x is fine for Riemannian submanifolds of Euclidean space) and

$$\mathrm{Hess}f(x)[u] \approx \frac{\mathrm{T}_{x\leftarrow y}\big(\mathrm{grad}f(y)\big) - \mathrm{grad}f(x)}{t}$$

setting $t = 2^{-14}/\|u\|_x$. Requires one call to $\mathrm{grad}f$, one retraction and one transport.

When do solvers terminate?

There are a few standard stopping criteria controlled by `options`:

`maxiter, maxtime, tolcost, tolgradnorm`

You can also define your own (evaluated after the standard ones). E.g.,

```
options.stopfun = @mystopfun;
function stopnow = mystopfun(problem, x, info, last)
    stopnow = (last >= 3 && info(last-2).cost - info(last).cost < 1e-3);
end
```

Interactive stopping criteria (these allow you to force the solver to terminate from outside the code):

```
options.stopfun = @stopifclosedfigure;
options.stopfun = stopifdeletedfile();
```

More things we didn't discuss

You can use a preconditioner with `problem.precon, problem.sqrtprecon`.

You can define "partial gradients" (for SGD) and "subgradients" (for nonsmooth f).

There are many line-search algorithms (select with `options.linesearch`); you can define your own; you can provide hints to line-search algorithms (through `problem.linesearch`).

There are many tools that can make your life easier: see the tutorial.

Hessian stuff: `hessianspectrum, hessianextreme, hessianmatrix`

More diagnostics tools: `checkretraction(M), checkmanifold(M)`

Plotting: `plotprofile, surfprofile`

Also: `criticalpointfinder, tangentspherefactory, tangentspacefactory, orthogonalize, tangentorthobasis, smallestinconvexhull, operator2matrix, ...`

Contributors welcome!

You can write your own solvers, manifold factories, examples, tools.

Questions / discussions welcome on the forum:
https://groups.google.com/g/manopttoolbox

Can also post bug reports / pull requests / raise issues on GitHub:
https://github.com/NicolasBoumal/manopt