NewNoteOfRIPM

Riemannian Interior Point Methods (RIPM)

Consider the nonlinear optimization problems on Riemannian manifolds, which aims to minimize the cost function in the given problem structure with (in)equality constraints:

$$\begin{array}{ll} \min_{x\in\mathcal{M}} & f(x) \\ \text{s.t.} & h(x)=0, \text{ and } g(x)\leq 0. \end{array}$$

where

- $\mathcal{M}$ is a Riemannian submanifold of $\mathcal{E}$; for now, we only consider its embedded geometry inherent from $\mathcal{E}$
- $f:\mathcal{M}\to\mathbb{R}$ is smooth
- $h:\mathcal{M}\to\mathcal{E}_{eq}$ is smooth
- $g:\mathcal{M}\to\mathcal{E}_{ineq}$ is smooth

overview:

- This is a **Riemannian Interior Point Methods (RIPM) solver**. The implementation is based on the toolbox 'Manopt'. Before running the codes, you must install the solver 'Manopt'.
- For a full description of the algorithm and theorems about RIPM, see the our paper: Z. Lai and A. Yoshise. *Riemannian Interior Point Methods for Constrained Optimization on Manifolds.*
- This solver **requires access to the gradients and the Hessians** of the cost function and all the constraints functions.
- As a interior point method, the **inequality constraints are necessary**. If your problem is not constrained by inequalities, try other methods such as a Riemannian augmented Lagrangian method. See more.
- **Equality constraints are optional. But there should never be redundant.** In fact we want the Riemannian gradients of all equality constraints to be linearly independent, otherwise Newton's equations may be singular.
- We prepared a additional document Implement Note of Global RIPM with detailed instructions for the implementation.

Usage of Solver `RIPM.m`

```
% function [xfinal, costfinal, residual, info, options] = RIPM(problem)
% function [xfinal, costfinal, residual, info, options] = RIPM(problem, x0)
% function [xfinal, costfinal, residual, info, options] = RIPM(problem, x0, options)
% function [xfinal, costfinal, residual, info, options] = RIPM(problem, [], options)
```

Input of `RIPM.m`

`problem` is a structure containing all descriptions about problem. See next secti

`x0`: The initial iterate is `x0` if it is provided. Otherwise, a random point on the manifold `problem.M` is picked. We strongly recommend using a strictly feasible point,i.e, $g(x)<0$, if any. To specify options whilst not specifying an initial iterate, give `x0` as `[]` (the empty matrix).

`option` is a structure, which is used to overwrite the default values. All options have a default value and are hence optional. To force an option value, pass an options structure with a field `options."..."`, where `"..."` is one of the following:

Field Name	Default Value	Description
Stopping criteria	-	-
`maxiter`	500	The algorithm terminates if maxiter iterations have been executed.
`maxtime`	3600	The algorithm terminates if maxtime seconds elapsed.
`tolKKTres`	1e-6	The algorithm terminates if the KKT residual drops below this.
Solve Newton equation	-	-
`KrylovIterMethod`	false	How to solve the condensed Newton equation. 0: Use representing matrix method. 1: Use Krylov subspace iterative method, specifically, the Conjugate Residual (CR) Method. See `TangentSpaceConjResMethod.m`
`KrylovTolrelres`	1e-9	CR method terminates when relative residual drops below this.
`KrylovMaxiter`	1000	CR method terminates if maxiter iterations have been executed.
`checkNTequation`	false	Whether to check the Newton equation. If true, more information will be contained in the output `info`.
Line search	-	-
`gamma`	0.9	Initial parameter for central functions. The value must be greater than 0.5 and less than 1.
`ls_execute_fun2`	false	Whether to execute the backtracking for the second central function.
`ls_beta`	1e-4	Constant for the backtracking line search (ls_). The value must belong to the interval (0, 1).
`ls_theta`	0.5	Reduction rate of the backtracking to find an appropriate step length. The value must belong to the interval (0, 1).
`ls_max_steps`	50	The algorithm breaks the backtracking if the ls_max_steps trial have been executed.
Initialization	-	-
`y`	zeros	Initial Lagrange multiplier vector for equality constraints.
`z`	random	Initial Lagrange multiplier vector for inequality constraints.
`s`	random	Initial slack vector.
Other parameters	-	-
`heuristic_z_s`	false	Whether to use a heuristic initial z and s. 0: Random initial z and s. 1: A heuristic way to construct z and s.
`desired_tau_1`	0.5	Construct the particular initial z,s such that tau1 equals `desired_tau_1`. The value must belong to the interval (0, 1). `desired_tau_1` is used only when `heuristic_z_s` is true.
`important`	1e0	Important controls tau_2 and initial rho. Important is used only when heuristic_z_s is true.
Display	-	-
`verbosity`	3	Integer number used to tune the amount and the frequency of output the algorithm generates during execution (mostly as text in the command window). The higher, the more output. 0 means silent.
Only when using fixed-rank manifolds	-	-
`rankviopena`	1e+8	If considering optimization on fixed-rank manifolds, the value is used as the penalty when violating the fixed-rank constraints.

Output of `RIPM.m`

`xfinal`, `costfinal`, `residual` are the last reached point on the manifold along with its cost value, and its KKT residual.

`info` is a struct-array which contains the following information at the each iteration. For example, `[info.KKT_residual]` returns a vector of the successive the KKT residual reached at each iteration.

Field Name	Type	Description
`iter`	integer	The iteration number. The initial value is 0.
`xcurrent`	depend on `problem.M`	The current point x. The initial value is 0.
`xCurCost`	double	The corresponding cost value.
`xCurPhi`	double	The corresponding merit value Phi.
`KKT_residual`	double	The corresponding residual of the KKT conditions (KKT residual).
`sigma`	double	The current barrier parameter σ for the perturbed Newton equation.
`rho`	double	The current barrier parameter μ for the perturbed Newton equation.
`time`	double	The total elapsed time in seconds (s) to reach the corresponding cost.
`stepsize`	double	The step size determined by the backtracking line search. 0<= `stepsize` <=1.
`ls_iters`	integer	The number of loops executed by backtracking line search.
`ls_max_steps_break`	Boolean	Whether the line search ends due to the excess of the maximal number of the backtracking, i.e., `ls_max_steps`: 0: the backtracking ends normally. 1: the backtracking ends due to the excess of the number.
`dist`	double	The Riemannian distance between the previous and the current points. If unavailable in Manopt, compute it with the Euclidean distance of its ambient space.

`info` will contains more fields if `options.checkNTequation` is true.

Field Name	Type	Description
`NTdir_error1`	double	The residual of the non-condensed Newton equation. It should be 0.
`NTdir_error2`	double	If the computed Newton direction is correct, then `NTdir_error2` should be 0.
`NTdir_norm`	double	The norm of the Newton direction.
`NTdir_angle`	double	The angle between `grad phi` and the Newton direction.

Describing the `Problem` Structure

`problem` is a structure containing all descriptions about our problem. `problem`'s field names are described in table below.

Some notes:

- the fields `M`, `cost`, `egrad`, `ehess` are identical to Manopt's usage for solving the unconstrained problem $\min_{x\in\mathcal{M}} f(x)$. They contain the all information about objective function of our problem. They are necessary.
- `Euc_ineq, ineq_con, barGx, ehess_barGx, barGxaj` are specificized fields for our RIPM solver. They contain the all information about inequality constraints. The inequality constraints are necessary, and so are they.
- `Euc_eq, eq_con, barHx, ehess_barHx, barHxaj` are also specificized fields for our RIPM solver. They contain the all information about equality constraints. However, the equality constraints are optional, and so are they.
- As a generalized solver, we require that only the Euclidean gradients and hessians information about the problem be passed to the solver. All of the above are automatically transformed into Riemannian counterparts, since we consider embedded Riemannian submanifold $\mathcal{M}$.
- There are two types of fields: 1. @ the MATLAB function handle. 2. manifold structure from Manopt's manifold factory, see more.
- For guidance on how to calculate the gradient (and Hessian) of a function step-by-step, please read Example 5.17 and Section 3.1, Section 4.7 of Boumal's book.

Field Name	Type	Description and Calculation	Input	Output	Note
`M`	manifold structure	$\mathcal{M}$ is a Riemannian submanifold of $\mathcal{E}$ for some ambient space $\mathcal{E}$, which often is a vector or matrix space.	-	-	Thus, $$x \in \mathcal{M} \subset \mathcal{E}; \Delta x \in T_x\mathcal{M} \subset \mathcal{E}$$; size of x and Δx are same.
`cost`	@(x)	$f(x)$ is the objective/cost function we want to minimize. f is a real-valued smooth function and at least is well-defined on $\mathcal{M}$, but always on whole $\mathcal{E}$ as well.	$x \in \mathcal{M}$	$\mathbb{R}$	For some manifold factory in Manopt, the point x is not stored in the full matrix representation. In particular, see `fixedrankembeddedfactory` at website. Also, there is a example `Fixedrank_projection_nonnega.m` where we used some tools (e.g., `M.triplet2matrix`) to write the correct cost function. This warning is appliable for all computations below.
`egrad`	@(x)	Euclidean gradient of f at point x, $\operatorname{egrad} f(x)$. You can (1) empirically compute it (2) use Matrix Calculus to compute it (3) compute it manually by the formal definition of gradient: $\mathrm{D} f(x)[v] = \langle \operatorname{egrad} f(x), v\rangle, \forall v \in \mathcal{E}$.	$x \in \mathcal{M}$	same size as x	We get the expression of $\operatorname{egrad} f$.
`ehess`	@(x, dx)	Euclidean Hessian operator of f at point x with its argument Δx (the tangent vector at x), $\operatorname{ehess} f(x)[\Delta x]$. You had better compute it manually by the formal definition of Hessian: $\operatorname{ehess} f(x)[\Delta x] = \mathrm{D}\left(\operatorname{egrad} f\right)(x)[\Delta x]$.	$x \in \mathcal{M}; \Delta x \in T_x\mathcal{M}$	same size as Δx	We need a Hessian operator, not the usual Hessian matrix! The premise of the calculation: $\operatorname{egrad} f$.
`Euc_ineq`	manifold structure	Euclidean space $\mathcal{E}_{ineq}$ is the codomain of function g, i.e., $g(x) \in \mathcal{E}_{ineq}$. Then, the multiplier $z \in \mathcal{E}_{ineq}$ and $\langle z, g(x)\rangle$ is computed using the Euclidean inner product in $\mathcal{E}_{ineq}$.	-	-	**The inequality constraints are necessary and redundant constraints are allowed.** The choice of $\mathcal{E}_{ineq}$ is free and it does not affect Newton's equations at all. This is in contrast to the space $\mathcal{E}_{eq}$ below.
`ineq_con`	@(x)	$g: \mathcal{M} \to \mathcal{E}_{ineq}$ mean inequality constraints, and we consider $g(x) \leq 0$. (0 is zero element of $\mathcal{E}_{ineq}$)	$x \in \mathcal{M}$	$\mathcal{E}_{ineq}$	
`barGx`	@(x, z)	Fix z, define $\tilde{g}_z(x) := \langle z, g(x)\rangle$, which is a real-valued function on $\mathcal{M}$. Then, compute the Euclidean gradient of $\tilde{g}_z$ at point x, $\operatorname{egrad} \tilde{g}_z(x)$. You can (1) empirically compute it (2) use Matrix Calculus to compute it (3) compute it manually by the formal definition of gradient: $\mathrm{D}\tilde{g}_z(x)[v] = \langle \operatorname{egrad} \tilde{g}_z(x), v\rangle, \forall v \in \mathcal{E}$.	$x \in \mathcal{M}; z \in \mathcal{E}_{ineq}$	same size as x	We get the expression of $\operatorname{egrad} \tilde{g}_z$.
`ehess_barGx`	@(x, z, dx)	Again, fix z and consider $\tilde{g}_z$. Then, compute the Euclidean Hessian operator of $\tilde{g}_z$ at point x with its argument Δx, $\operatorname{ehess} \tilde{g}_z(x)[\Delta x]$. You had better compute it manually by the formal definition of Hessian: $\operatorname{ehess} \tilde{g}_z(x)[\Delta x] = \mathrm{D}\left(\operatorname{egrad} \tilde{g}_z\right)(x)[\Delta x]$.	$x \in \mathcal{M}; z \in \mathcal{E}_{ineq}; \Delta x \in T_x\mathcal{M}$	same size as Δx	We need a Hessian operator, not the usual Hessian matrix! The premise of the calculation: $\operatorname{egrad} \tilde{g}_z$.
`barGxaj`	@(x, dx)	Fix x, define a function $\bar{G}_x: \mathcal{E}_{ineq} \to \mathcal{E}$ by $\bar{G}_x(z) = \operatorname{egrad} \tilde{g}_z(x)$. Then, compute the adjoint operator $\bar{G}_x^*: \mathcal{E} \to \mathcal{E}_{ineq}$. There seems to be no better way than manual computation according to the definition of adjoint: $\langle \bar{G}_x(z), \Delta x\rangle = \langle z, \bar{G}_x^*(\Delta x)\rangle, \forall z \in \mathcal{E}_{ineq}, \Delta x \in \mathcal{E}$.	$x \in \mathcal{M}; \Delta x \in T_x\mathcal{M}$	same size as z	If we restrict domain of $\bar{G}_x^* \$ to subspace \$ T_x\mathcal{M}$ of $\mathcal{E}$, then $\bar{G}_x^* = G_x^*$, thus, we write argument of $\bar{G}_x^*$ as Δx. The premise of the calculation: $\operatorname{egrad} \tilde{g}_z$.

Field Name	Type	Description and Calculation	Input	Output	Note
`Euc_eq`	manifold structure	Euclidean space $\mathcal{E}_{eq}$ is the codomain of function h, i.e., $h(x) \in \mathcal{E}_{eq}$. Then, the multiplier $y \in \mathcal{E}_{eq}$ and $\langle y, h(x)\rangle$ is computed using the Euclidean inner product in $\mathcal{E}_{eq}$.	-	-	**Equality constraints are optional. But there should never be redundant.** In fact we want the Riemannian gradients of all equality constraints to be linearly independent, otherwise Newton's equations may be singular. Be cautious: Equality constraints in matrix form often contain duplicate expressions, e.g., $h(x) = X^T X - I$. When $h(x) = 0$, only the diagonal and above component functions are independent and valid constraints, while the rest is redundant. To avoid this problem, it is sufficient to set `problem.Euc_eq` to be `symmetricfactory(n)`. Here, `problem. Euc_eq` is the codomain of h, its dimension is the actual number of equality constraints. See examples of `Euc_linear_or_projection_nonnega_orthogonalEq.m` and `Euc_projection_nonnega_symmetricEq.m`
`eq_con`	@(x)	$h : \mathcal{M} \to \mathcal{E}_{eq}$ mean equality constraints, and we consider $h(x) = 0$. (0 is zero element of $\mathcal{E}_{eq}$)	$x \in \mathcal{M}$	$\mathcal{E}_{eq}$	
`barHx`	@(x, y)	Fix y, define $\tilde{h}_y(x) := \langle y, h(x)\rangle$, which is a real-valued function on $\mathcal{M}$. Then, compute the Euclidean gradient of $\tilde{h}_y$ at point x, $\operatorname{egrad} \tilde{h}_y(x)$. You can (1) empirically compute it (2) use Matrix Calculus to compute it (3) compute it manually by the formal definition of gradient: $\mathrm{D}\tilde{h}_y(x)[v] = \langle \operatorname{egrad} \tilde{h}_y(x), v\rangle, \forall v \in \mathcal{E}$.	$x \in \mathcal{M}; y \in \mathcal{E}_{eq}$	same size as x	We get the expression of $\operatorname{egrad} \tilde{h}_y$.
`ehess_barHx`	@(x, y, dx)	Again, fix y and consider $\tilde{h}_y$. Then, compute the Euclidean Hessian operator of $\tilde{h}_y$ at point x with its argument Δx, $\operatorname{ehess} \tilde{h}_y(x)[\Delta x]$. You had better compute it manually by the formal definition of Hessian: $\operatorname{ehess} \tilde{h}_y(x)[\Delta x] = \mathrm{D}\left(\operatorname{egrad} \tilde{h}_y\right)(x)[\Delta x]$.	$x \in \mathcal{M}; y \in \mathcal{E}_{eq}; \Delta x \in T_x\mathcal{M}$	same size as Δx	We need a Hessian operator, not the usual Hessian matrix! The premise of the calculation: $\operatorname{egrad} \tilde{h}_y$.
`barHxaj`	@(x, dx)	Fix x, define a function $\bar{H}_x : \mathcal{E}_{eq} \to \mathcal{E}$ by $\bar{H}_x(y) := \operatorname{egrad} \tilde{h}_y(x)$. Then, compute the adjoint operator $\bar{H}_x^* : \mathcal{E} \to \mathcal{E}_{eq}$. There seems to be no better way than manual computation according to the definition of adjoint: $\langle \bar{H}_x(y), \Delta x\rangle = \langle y, \bar{H}_x^*(\Delta x)\rangle, \forall y \in \mathcal{E}_{eq}, \Delta x \in \mathcal{E}$.	$x \in \mathcal{M}; \Delta x \in T_x\mathcal{M}$	same size as y	If we restrict domain of $\bar{H}_x^*$ to subspace $T_x\mathcal{M}$ of $\mathcal{E}$, then $\bar{H}_x^* = H_x^*$, thus, we write argument of $\bar{H}_x^*$ as Δx. The premise of the calculation: $\operatorname{egrad} \tilde{h}_y$.

Appendix: Calculus

Let U, V be open sets in two linear spaces $\mathcal{E}, \mathcal{F}$. If $F : U \to V$ is smooth at x, the differential of F at x is the linear map $\mathrm{D}F(x) : \mathcal{E} \to \mathcal{F}$ defined by

$$\mathrm{D}F(x)[u] = \lim_{t \to 0} \frac{F(x + tu) - F(x)}{t}.$$

For a smooth function $f : \mathcal{E} \to \mathbb{R}$ defined on a Euclidean space $\mathcal{E}$, the (Euclidean) gradient of f is the map $\operatorname{grad} f : \mathcal{E} \to \mathcal{E}$ defined by the following property:

$$\mathrm{D}f(x)[v] = \lim_{t \to 0} \frac{f(x + tv) - f(x)}{t} = \langle \operatorname{grad} f(x), v\rangle, \quad \forall x, v \in \mathcal{E}.$$

The (Euclidean) Hessian of f at x is the linear map $\operatorname{Hess} f(x) : \mathcal{E} \to \mathcal{E}$ defined by

$$\operatorname{Hess} f(x)[v] = \mathrm{D}(\operatorname{grad} f)(x)[v] = \lim_{t \to 0} \frac{\operatorname{grad} f(x + tv) - \operatorname{grad} f(x)}{t}.$$