tvopt

Release 0.2.7

Nicola Bastianello

Jan 09, 2023

CONTENTS:

CHAPTER ONE

TVOPT PACKAGE

1.1 Submodules

1.2 tvopt.costs module

Cost template definition and examples.

class tvopt.costs.**AbsoluteValue**(*weight=1*)

Bases: *Cost*

Scalar absolute value function.

function(*x*)

An evaluation of the cost. *Implement if needed*.

Parameters

- **x** (*array_like*) – The x where the cost should be evaluated.
- ***args** – The time at which the cost should be evaluated. Not required if the cost is static.
- ****kwargs** – Any other required argument.

gradient(*x*)

An evaluation of the cost's gradient or sub-gradient. *Implement if needed*.

Parameters

- **x** (*array_like*) – The x where the (sub-)gradient should be evaluated.
- ***args** – The time at which the (sub-)gradient should be evaluated. Not required if the cost is static.
- ****kwargs** – Any other required argument.

proximal(*x*, *penalty=1*)

An evaluation of the cost's proximal.

If this method is not overwritten, the default behavior is to recursively compute the proximal via a gradient or Newton backtracking algorithm. See *compute_proximal* for the function that is used for this purpose.

Parameters

- **x** (*array_like*) – The x where the proximal should be evaluated.
- ***args** – The time at which the proximal should be evaluated. Not required if the cost is static.

- **penalty** (*float, optional*) – The penalty parameter ρ for the proximal evaluation. Defaults to 1.
- ****kwargs** – Any other required argument.

class tvopt.costs.**Constant**(*dom*, *c*)

Bases: *Cost*

Constant cost.

This class defines a constant, whose value is stored in the attribute *c*. The *gradient* and *hessian* methods return 0, while the proximal acts as an identity.

dom

The given cost domain, for compatibility with other costs.

Type
sets.Set

c

The constant value.

Type
float

smooth

The smoothness degree, set to 2.

Type
int

function(**args*, ***kwargs*)

An evaluation of the cost.

Returns the costant value.

gradient(**args*, ***kwargs*)

An evaluation of the cost's gradient.

Returns 0.

hessian(**args*, ***kwargs*)

An evaluation of the cost's Hessian.

Returns 0.

proximal(*x*, **args*, ***kwargs*)

An evaluation of the cost's proximal.

Acts as the identity, returning *x*.

class tvopt.costs.**Cost**(*dom*, *time=None*, *prox_solver=None*)

Bases: object

Template for a cost function.

This class defines the template for a cost function

$$f : \mathbb{R}^{n_1 \times n_2 \times \cdots} \times \mathbb{R}_+ \to \mathbb{R} \cup \{+\infty\}$$

which depends on the unknown $\boldsymbol{x} \in \mathbb{R}^{n_1 \times n_2 \times \cdots}$ and, optionally, on the time $t \in \mathbb{R}_+$.

Cost objects support the following operations:

- negation
- sum (by another cost or with a scalar),
- product (by another cost or with a scalar),
- division and power with a scalar.

A *Cost* object should expose, compatibly with the smoothness degree, the methods *function*, *gradient*, *hessian*, *proximal*. The convention for these methods is that the first positional argument is $\boldsymbol{x}$, and only a second positional argument is allowed, for t. Any other argument should be passed as a keyword argument.

If the cost is time-varying, then it should expose the methods *time_derivative* and *sample*, as well; see methods' documentation for the default behavior.

dom

The x domain $\mathbb{R}^{n_1 \times n_2 \times \cdots}$.

Type
sets.Set

time

The time domain $\mathbb{R}_+$. If the cost is static this is None.

Type
sets.T

is_dynamic

Attribute to check if the cost is static or dynamic.

Type
bool

smooth

This attribute stores the smoothness degree of the cost, for example it is 0 if the cost is continuous, 1 if the cost is differentiable, *etc*. By convention it is -1 if the cost is discontinuous.

Type
int

_prox_solver

This attribute specifies the method (gradient or Newton) that should be used to compute the proximal

$$\text{prox}_{\rho f(\cdot;t)}(\boldsymbol{x}) = \text{argmin}_{\boldsymbol{y}} \left\{ f(\boldsymbol{y};t) + \frac{1}{2\rho}\|\boldsymbol{y}-\boldsymbol{x}\|^2 \right\}$$

of the cost, if a closed form is not available. See also the auxiliary function *compute_proximal*.

Type
str or None

Notes

Not all operations preserve convexity.

function(*x*, **args*, ***kwargs*)

An evaluation of the cost. *Implement if needed*.

Parameters

- **x** (*array_like*) – The x where the cost should be evaluated.

- **`*args`** – The time at which the cost should be evaluated. Not required if the cost is static.
- **`**kwargs`** – Any other required argument.

`gradient`(*x*, **args*, ***kwargs*)

An evaluation of the cost's gradient or sub-gradient. *Implement if needed*.

Parameters

- **x** (*array_like*) – The x where the (sub-)gradient should be evaluated.
- **`*args`** – The time at which the (sub-)gradient should be evaluated. Not required if the cost is static.
- **`**kwargs`** – Any other required argument.

`hessian`(*x*, **args*, ***kwargs*)

An evaluation of the cost's Hessian. *Implement if needed*.

Parameters

- **x** (*array_like*) – The x where the Hessian should be evaluated.
- **`*args`** – The time at which the Hessian should be evaluated. Not required if the cost is static.
- **`**kwargs`** – Any other required argument.

`proximal`(*x*, **args*, *penalty=1*, ***kwargs*)

An evaluation of the cost's proximal.

If this method is not overwritten, the default behavior is to recursively compute the proximal via a gradient or Newton backtracking algorithm. See *compute_proximal* for the function that is used for this purpose.

Parameters

- **x** (*array_like*) – The x where the proximal should be evaluated.
- **`*args`** – The time at which the proximal should be evaluated. Not required if the cost is static.
- **penalty** (*float, optional*) – The penalty parameter ρ for the proximal evaluation. Defaults to 1.
- **`**kwargs`** – Any other required argument.

`sample`(*t*)

Sample the cost.

This method returns a *SampledCost* object which exposes the same methods of the cost but fixing the time argument to *t*.

If the cost is static, the cost itself is returned.

Parameters

t (*float*) – The time at which the cost should be sampled.

Returns

The sampled cost or, if static, the cost itself.

Return type

Cost

time_derivative(*x*, *t*, *der='tx'*, ***kwargs*)

A derivative w.r.t. time of the cost.

This method computes derivatives w.r.t. time of the cost, or mixed derivatives w.r.t. both time and x (*e.g.* the derivative in time of the gradient).

If this method is not overwritten, it computes the derivative by default using *backward finite differences*. See *backward_finite_difference* for details.

If the cost is static, 0 is returned.

Parameters

- **x** (*array_like*) – The x where the derivative should be evaluated.
- **t** (*float*) – The time at which the derivative should be evaluated.
- **der** (*str, optional*) – A sequence of "x" and "t" that chooses which derivative should be computed. For example, the default "tx" denotes the derivative w.r.t. time of the cost's (sub-)gradient.
- ****kwargs** – Any other required argument.

Raises

ValueError – If the number of "x" characters in *der* exceeds 2.

Returns

The required derivative or 0.

Return type

array_like

class tvopt.costs.**DiscreteDynamicCost**(*costs*, *t_s=1*)

Bases: *Cost*

Dynamic cost from a sequence of static costs.

This class creates a dynamic cost from a list of static costs. That is, given a sampling time T_s, the cost at time $t_k = kT_\mathrm{s}$ is:

$$f(\boldsymbol{x}; t_k) = f_k(\boldsymbol{x})$$

with f_k the k-th static cost in the list.

function(*x*, *t*, ***kwargs*)

An evaluation of the cost. *Implement if needed.*

Parameters

- **x** (*array_like*) – The x where the cost should be evaluated.
- ***args** – The time at which the cost should be evaluated. Not required if the cost is static.
- ****kwargs** – Any other required argument.

gradient(*x*, *t*, ***kwargs*)

An evaluation of the cost's gradient or sub-gradient. *Implement if needed.*

Parameters

- **x** (*array_like*) – The x where the (sub-)gradient should be evaluated.
- ***args** – The time at which the (sub-)gradient should be evaluated. Not required if the cost is static.

- ****kwargs** – Any other required argument.

hessian(*x*, *t*, ***kwargs*)

An evaluation of the cost's Hessian. *Implement if needed*.

Parameters

- **x** (*array_like*) – The x where the Hessian should be evaluated.
- ***args** – The time at which the Hessian should be evaluated. Not required if the cost is static.
- ****kwargs** – Any other required argument.

proximal(*x*, *t*, ***kwargs*)

An evaluation of the cost's proximal.

If this method is not overwritten, the default behavior is to recursively compute the proximal via a gradient or Newton backtracking algorithm. See *compute_proximal* for the function that is used for this purpose.

Parameters

- **x** (*array_like*) – The x where the proximal should be evaluated.
- ***args** – The time at which the proximal should be evaluated. Not required if the cost is static.
- **penalty** (*float, optional*) – The penalty parameter ρ for the proximal evaluation. Defaults to 1.
- ****kwargs** – Any other required argument.

sample(*t*)

Sample the cost.

The difference with the default *Cost* method is that it returns a cost in the list rather than a *SampledCost*.

Parameters

t (*float*) – The time at which the cost should be sampled.

Returns

The closest cost in the list.

Return type

Cost

class tvopt.costs.**DynamicExample_1D**(*t_s*, *t_max*, *omega=0.06283185307179587*, *kappa=7.5*, *mu=1.75*)

Bases: *Cost*

Scalar benchmark dynamic cost.

The dynamic cost was proppосed in[2] and is defined as:

$$f(x;t) = \frac{1}{2}(x - \cos(\omega t))^2 + \kappa \log(1 + \exp(\mu x))$$

with default parameters $\omega = 0.02\pi$, $\kappa = 7.5$ and $\mu = 1.75$.

approximate_time_derivative(*x*, *t*, *der='tx'*)

[2] A. Simonetto, A. Mokhtari, A. Koppel, G. Leus, and A. Ribeiro, "A Class of Prediction-Correction Methods for Time-Varying Convex Optimization," IEEE Transactions on Signal Processing, vol. 64, no. 17, pp. 4576–4591, Sep. 2016.

function(*x*, *t*)

An evaluation of the cost. *Implement if needed*.

Parameters

- **x** (*array_like*) – The x where the cost should be evaluated.
- ***args** – The time at which the cost should be evaluated. Not required if the cost is static.
- ****kwargs** – Any other required argument.

gradient(*x*, *t*)

An evaluation of the cost's gradient or sub-gradient. *Implement if needed*.

Parameters

- **x** (*array_like*) – The x where the (sub-)gradient should be evaluated.
- ***args** – The time at which the (sub-)gradient should be evaluated. Not required if the cost is static.
- ****kwargs** – Any other required argument.

hessian(*x*, *t=None*)

An evaluation of the cost's Hessian. *Implement if needed*.

Parameters

- **x** (*array_like*) – The x where the Hessian should be evaluated.
- ***args** – The time at which the Hessian should be evaluated. Not required if the cost is static.
- ****kwargs** – Any other required argument.

time_derivative(*x*, *t*, *der='tx'*)

A derivative w.r.t. time of the cost.

This method computes derivatives w.r.t. time of the cost, or mixed derivatives w.r.t. both time and x (*e.g.* the derivative in time of the gradient).

If this method is not overwritten, it computes the derivative by default using *backward finite differences*. See *backward_finite_difference* for details.

If the cost is static, 0 is returned.

Parameters

- **x** (*array_like*) – The x where the derivative should be evaluated.
- **t** (*float*) – The time at which the derivative should be evaluated.
- **der** (*str, optional*) – A sequence of "x" and "t" that chooses which derivative should be computed. For example, the default "tx" denotes the derivative w.r.t. time of the cost's (sub-)gradient.
- ****kwargs** – Any other required argument.

Raises

ValueError – If the number of "x" characters in *der* exceeds 2.

Returns

The required derivative or 0.

Return type

array_like

class tvopt.costs.**DynamicExample_2D**(*t_s*, *t_max*)

Bases: *Cost*

Bi-dimensional benchmark dynamic cost.

The dynamic cost was proposed in[3] and is defined as:

$$f(\boldsymbol{x}; t) = \frac{1}{2}(x_1 - \exp(\cos(t)))^2 + \frac{1}{2}(x_2 - x_1 \tanh(t))^2$$

where we used the notation $\boldsymbol{x} = [x_1, x_2]^\top$.

approximate_time_derivative(*x*, *t*, *der='tx'*)

function(*x*, *t*)

An evaluation of the cost. *Implement if needed*.

Parameters

- **x** (*array_like*) – The x where the cost should be evaluated.
- ***args** – The time at which the cost should be evaluated. Not required if the cost is static.
- ****kwargs** – Any other required argument.

gradient(*x*, *t*)

An evaluation of the cost's gradient or sub-gradient. *Implement if needed*.

Parameters

- **x** (*array_like*) – The x where the (sub-)gradient should be evaluated.
- ***args** – The time at which the (sub-)gradient should be evaluated. Not required if the cost is static.
- ****kwargs** – Any other required argument.

hessian(*x=None*, *t=None*)

An evaluation of the cost's Hessian. *Implement if needed*.

Parameters

- **x** (*array_like*) – The x where the Hessian should be evaluated.
- ***args** – The time at which the Hessian should be evaluated. Not required if the cost is static.
- ****kwargs** – Any other required argument.

time_derivative(*x*, *t*, *der='tx'*)

A derivative w.r.t. time of the cost.

This method computes derivatives w.r.t. time of the cost, or mixed derivatives w.r.t. both time and x (*e.g.* the derivative in time of the gradient).

If this method is not overwritten, it computes the derivative by default using *backward finite differences*. See *backward_finite_difference* for details.

If the cost is static, 0 is returned.

[3] Y. Zhang, Z. Qi, B. Qiu, M. Yang, and M. Xiao, "Zeroing Neural Dynamics and Models for Various Time-Varying Problems Solving with ZLSF Models as Minimization-Type and Euler-Type Special Cases [Research Frontier]," IEEE Computational Intelligence Magazine, vol. 14, no. 3, pp. 52–60, Aug. 2019.

Parameters

- **x** (*array_like*) – The x where the derivative should be evaluated.
- **t** (*float*) – The time at which the derivative should be evaluated.
- **der** (*str, optional*) – A sequence of "x" and "t" that chooses which derivative should be computed. For example, the default "tx" denotes the derivative w.r.t. time of the cost's (sub-)gradient.
- ****kwargs** – Any other required argument.

Raises

`ValueError` – If the number of "x" characters in *der* exceeds 2.

Returns

The required derivative or 0.

Return type

array_like

`class tvopt.costs.Huber(n, threshold)`

Bases: `Cost`

Vector Huber loss.

The cost is defined as

$$f(\boldsymbol{x}) = \begin{cases} \|\boldsymbol{x}\|^2/2 & \text{if } \|\boldsymbol{x}\| \leq \theta \\ \theta(\|\boldsymbol{x}\| - \theta/2) & \text{otherwise} \end{cases}$$

where $\theta > 0$ is a given threshold.

`function(x)`

An evaluation of the cost. *Implement if needed.*

Parameters

- **x** (*array_like*) – The x where the cost should be evaluated.
- ***args** – The time at which the cost should be evaluated. Not required if the cost is static.
- ****kwargs** – Any other required argument.

`gradient(x)`

An evaluation of the cost's gradient or sub-gradient. *Implement if needed.*

Parameters

- **x** (*array_like*) – The x where the (sub-)gradient should be evaluated.
- ***args** – The time at which the (sub-)gradient should be evaluated. Not required if the cost is static.
- ****kwargs** – Any other required argument.

`hessian(x)`

An evaluation of the cost's Hessian. *Implement if needed.*

Parameters

- **x** (*array_like*) – The x where the Hessian should be evaluated.

- ***args** – The time at which the Hessian should be evaluated. Not required if the cost is static.
- ****kwargs** – Any other required argument.

proximal(*x*, *penalty=1*)

An evaluation of the cost's proximal.

If this method is not overwritten, the default behavior is to recursively compute the proximal via a gradient or Newton backtracking algorithm. See *compute_proximal* for the function that is used for this purpose.

Parameters

- **x** (*array_like*) – The x where the proximal should be evaluated.
- ***args** – The time at which the proximal should be evaluated. Not required if the cost is static.
- **penalty** (*float, optional*) – The penalty parameter ρ for the proximal evaluation. Defaults to 1.
- ****kwargs** – Any other required argument.

class tvopt.costs.**Huber_1D**(*threshold*)

Bases: *Cost*

Huber loss.

The cost is defined as

$$f(x) = \begin{cases} x^2/2 & \text{if } |x| \leq \theta \\ \theta(|x| - \theta/2) & \text{otherwise} \end{cases}$$

where $\theta > 0$ is a given threshold.

function(*x*)

An evaluation of the cost. *Implement if needed.*

Parameters

- **x** (*array_like*) – The x where the cost should be evaluated.
- ***args** – The time at which the cost should be evaluated. Not required if the cost is static.
- ****kwargs** – Any other required argument.

gradient(*x*)

An evaluation of the cost's gradient or sub-gradient. *Implement if needed.*

Parameters

- **x** (*array_like*) – The x where the (sub-)gradient should be evaluated.
- ***args** – The time at which the (sub-)gradient should be evaluated. Not required if the cost is static.
- ****kwargs** – Any other required argument.

hessian(*x*)

An evaluation of the cost's Hessian. *Implement if needed.*

Parameters

- **x** (*array_like*) – The x where the Hessian should be evaluated.
- ***args** – The time at which the Hessian should be evaluated. Not required if the cost is static.
- ****kwargs** – Any other required argument.

proximal(*x*, *penalty=1*)

An evaluation of the cost's proximal.

If this method is not overwritten, the default behavior is to recursively compute the proximal via a gradient or Newton backtracking algorithm. See *compute_proximal* for the function that is used for this purpose.

Parameters

- **x** (*array_like*) – The x where the proximal should be evaluated.
- ***args** – The time at which the proximal should be evaluated. Not required if the cost is static.
- **penalty** (*float, optional*) – The penalty parameter ρ for the proximal evaluation. Defaults to 1.
- ****kwargs** – Any other required argument.

class tvopt.costs.**Indicator**(*s*)

Bases: Cost

Indicator function of a given set.

This objects implements the indicator function of a given *Set* object. That is, given the set $\mathbb{S}$ we define:

$$f(\boldsymbol{x}) = \begin{cases} 0 & \text{if } \boldsymbol{x} \in \mathbb{S} \\ +\infty & \text{otherwise.} \end{cases}$$

The proximal operator of the cost is the projection onto the set.

function(*x*)

An evaluation of the cost. *Implement if needed.*

Parameters

- **x** (*array_like*) – The x where the cost should be evaluated.
- ***args** – The time at which the cost should be evaluated. Not required if the cost is static.
- ****kwargs** – Any other required argument.

projection(*x*, ***kwargs*)

proximal(*x*, **args*, *penalty=1*, ***kwargs*)

An evaluation of the cost's proximal.

If this method is not overwritten, the default behavior is to recursively compute the proximal via a gradient or Newton backtracking algorithm. See *compute_proximal* for the function that is used for this purpose.

Parameters

- **x** (*array_like*) – The x where the proximal should be evaluated.
- ***args** – The time at which the proximal should be evaluated. Not required if the cost is static.

- **penalty** (*float, optional*) – The penalty parameter ρ for the proximal evaluation. Defaults to 1.
- ****kwargs** – Any other required argument.

class tvopt.costs.**Linear**(*b*, *c=0*)

Bases: *Cost*

Linear cost.

The function is defined as

$$f(x) = \langle \boldsymbol{x}, \boldsymbol{b} \rangle + c.$$

class tvopt.costs.**LinearRegression**(*A*, *b*)

Bases: *Cost*

Cost for linear regression.

The cost is defined as

$$f(\boldsymbol{x}) = \frac{1}{2} \|\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}\|^2.$$

class tvopt.costs.**Logistic**

Bases: *Cost*

Logistic function.

The function is defined as

$$f(x) = \log\left(1 + \exp(x)\right).$$

function(*x*)

An evaluation of the cost. *Implement if needed.*

Parameters

- **x** (*array_like*) – The x where the cost should be evaluated.
- ***args** – The time at which the cost should be evaluated. Not required if the cost is static.
- ****kwargs** – Any other required argument.

gradient(*x*)

An evaluation of the cost's gradient or sub-gradient. *Implement if needed.*

Parameters

- **x** (*array_like*) – The x where the (sub-)gradient should be evaluated.
- ***args** – The time at which the (sub-)gradient should be evaluated. Not required if the cost is static.
- ****kwargs** – Any other required argument.

hessian(*x*)

An evaluation of the cost's Hessian. *Implement if needed.*

Parameters

- **x** (*array_like*) – The x where the Hessian should be evaluated.
- ***args** – The time at which the Hessian should be evaluated. Not required if the cost is static.
- ****kwargs** – Any other required argument.

proximal(*x*, *penalty=1*, *max_iter=50*, *tol=1e-08*)

An evaluation of the cost's proximal.

If this method is not overwritten, the default behavior is to recursively compute the proximal via a gradient or Newton backtracking algorithm. See *compute_proximal* for the function that is used for this purpose.

Parameters

- **x** (*array_like*) – The x where the proximal should be evaluated.
- ***args** – The time at which the proximal should be evaluated. Not required if the cost is static.
- **penalty** (*float, optional*) – The penalty parameter ρ for the proximal evaluation. Defaults to 1.
- ****kwargs** – Any other required argument.

class tvopt.costs.**LogisticRegression**(*A*, *b*, *weight=0*)

Bases: `Cost`

Cost for logistic regression.

The cost is defined as

$$f(\boldsymbol{x}) = \sum_{i=1}^{m} \log\left(1 + \exp\left(-b_i \langle \boldsymbol{a}_i, \boldsymbol{x} \rangle + x_0\right)\right)$$

where $b_i \in \{-1, 1\}$, $\boldsymbol{a}_i$ are classifier and feature vector, and x_0 is the intercept. An optional ℓ_2 regularization can be added defining its weight *penalty*.

function(*x*)

An evaluation of the cost. *Implement if needed.*

Parameters

- **x** (*array_like*) – The x where the cost should be evaluated.
- ***args** – The time at which the cost should be evaluated. Not required if the cost is static.
- ****kwargs** – Any other required argument.

gradient(*x*)

An evaluation of the cost's gradient or sub-gradient. *Implement if needed.*

Parameters

- **x** (*array_like*) – The x where the (sub-)gradient should be evaluated.
- ***args** – The time at which the (sub-)gradient should be evaluated. Not required if the cost is static.

- ****kwargs** – Any other required argument.

hessian(*x*)

An evaluation of the cost's Hessian. *Implement if needed.*

Parameters

- **x** (*array_like*) – The x where the Hessian should be evaluated.
- ***args** – The time at which the Hessian should be evaluated. Not required if the cost is static.
- ****kwargs** – Any other required argument.

proximal(*x*, *penalty=1*, *tol=1e-05*, *max_iter=100*)

An evaluation of the cost's proximal.

If this method is not overwritten, the default behavior is to recursively compute the proximal via a gradient or Newton backtracking algorithm. See *compute_proximal* for the function that is used for this purpose.

Parameters

- **x** (*array_like*) – The x where the proximal should be evaluated.
- ***args** – The time at which the proximal should be evaluated. Not required if the cost is static.
- **penalty** (*float, optional*) – The penalty parameter ρ for the proximal evaluation. Defaults to 1.
- ****kwargs** – Any other required argument.

class tvopt.costs.**Norm_1**(*n=1*, *weight=1*)

Bases: *Cost*

Class for ℓ_1 norm.

The function is defined as

$$f(\boldsymbol{x}) = w\|\boldsymbol{x}\|_1$$

for $\boldsymbol{x} \in \mathbb{R}^n$ and $w > 0$.

function(*x*)

An evaluation of the cost. *Implement if needed.*

Parameters

- **x** (*array_like*) – The x where the cost should be evaluated.
- ***args** – The time at which the cost should be evaluated. Not required if the cost is static.
- ****kwargs** – Any other required argument.

gradient(*x*)

An evaluation of the cost's gradient or sub-gradient. *Implement if needed.*

Parameters

- **x** (*array_like*) – The x where the (sub-)gradient should be evaluated.
- ***args** – The time at which the (sub-)gradient should be evaluated. Not required if the cost is static.

- ****kwargs** – Any other required argument.

proximal(*x*, *penalty=1*)

Proximal evaluation of ℓ_1 norm, a.k.a. soft-thresholding.

See also:

`utils.soft_thresholding`

class tvopt.costs.**Norm_2**(*n=1*, *weight=1*)

Bases: *Cost*

Square 2-norm.

class tvopt.costs.**Norm_inf**(*n=1*, *weight=1*)

Bases: *Cost*

Class for ℓ_∞ norm.

function(*x*)

An evaluation of the cost. *Implement if needed*.

Parameters

- **x** (*array_like*) – The x where the cost should be evaluated.
- ***args** – The time at which the cost should be evaluated. Not required if the cost is static.
- ****kwargs** – Any other required argument.

proximal(*x*, *penalty=1*, *tol=1e-05*)

Proximal evaluation of ℓ_∞ norm.

See[4] for the formula.

References

class tvopt.costs.**PowerCost**(*cost*, *p*)

Bases: *Cost*

Power cost.

This class defines a cost as the given power of a cost. It is used for implementing the * operation.

function(**args*, ***kwargs*)

An evaluation of the power cost.

gradient(**args*, ***kwargs*)

An evaluation of the power cost (sub-)gradient.

hessian(**args*, ***kwargs*)

An evaluation of the power cost Hessian.

class tvopt.costs.**ProductCost**(*c_1*, *c_2*)

Bases: *Cost*

Product of two costs.

This class defines a cost from the product of two given costs. Derivatives are computed using the chain rule.

[4] A. Beck, First-Order Methods in Optimization. Philadelphia, PA: Society for Industrial and Applied Mathematics, 2017.

function(*x*, **args*, ***kwargs*)

An evaluation of the product cost.

gradient(*x*, **args*, ***kwargs*)

An evaluation of the product cost (sub-)gradient.

hessian(*x*, **args*, ***kwargs*)

An evaluation of the product cost Hessian.

class tvopt.costs.**Quadratic**(*A*, *b*, *c=0*)

Bases: *Cost*

Quadratic cost.

The function is defined as

$$f(x) = \frac{1}{2}\boldsymbol{x}^\top \boldsymbol{A}\boldsymbol{x} + \langle \boldsymbol{x}, \boldsymbol{b} \rangle + c$$

with the given matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ and vector $\boldsymbol{b} \in \mathbb{R}^n$.

function(*x*)

An evaluation of the cost. *Implement if needed.*

Parameters

- **x** (*array_like*) – The x where the cost should be evaluated.
- ***args** – The time at which the cost should be evaluated. Not required if the cost is static.
- ****kwargs** – Any other required argument.

gradient(*x*)

An evaluation of the cost's gradient or sub-gradient. *Implement if needed.*

Parameters

- **x** (*array_like*) – The x where the (sub-)gradient should be evaluated.
- ***args** – The time at which the (sub-)gradient should be evaluated. Not required if the cost is static.
- ****kwargs** – Any other required argument.

hessian(*x=None*)

An evaluation of the cost's Hessian. *Implement if needed.*

Parameters

- **x** (*array_like*) – The x where the Hessian should be evaluated.
- ***args** – The time at which the Hessian should be evaluated. Not required if the cost is static.
- ****kwargs** – Any other required argument.

proximal(*x*, *penalty=1*)

An evaluation of the cost's proximal.

If this method is not overwritten, the default behavior is to recursively compute the proximal via a gradient or Newton backtracking algorithm. See *compute_proximal* for the function that is used for this purpose.

Parameters

- **x** (*array_like*) – The x where the proximal should be evaluated.
- ***args** – The time at which the proximal should be evaluated. Not required if the cost is static.
- **penalty** (*float, optional*) – The penalty parameter ρ for the proximal evaluation. Defaults to 1.
- ****kwargs** – Any other required argument.

class tvopt.costs.**Quadratic_1D**(*a*, *b*, *c=0*)

Bases: *Cost*

Scalar quadratic cost.

The cost is defined as

$$f(x) = ax^2/2 + bx + c.$$

function(*x*)

An evaluation of the cost. *Implement if needed.*

Parameters

- **x** (*array_like*) – The x where the cost should be evaluated.
- ***args** – The time at which the cost should be evaluated. Not required if the cost is static.
- ****kwargs** – Any other required argument.

gradient(*x*)

An evaluation of the cost's gradient or sub-gradient. *Implement if needed.*

Parameters

- **x** (*array_like*) – The x where the (sub-)gradient should be evaluated.
- ***args** – The time at which the (sub-)gradient should be evaluated. Not required if the cost is static.
- ****kwargs** – Any other required argument.

hessian(*x=None*)

An evaluation of the cost's Hessian. *Implement if needed.*

Parameters

- **x** (*array_like*) – The x where the Hessian should be evaluated.
- ***args** – The time at which the Hessian should be evaluated. Not required if the cost is static.
- ****kwargs** – Any other required argument.

proximal(*x*, *penalty=1*)

An evaluation of the cost's proximal.

If this method is not overwritten, the default behavior is to recursively compute the proximal via a gradient or Newton backtracking algorithm. See *compute_proximal* for the function that is used for this purpose.

Parameters

- **x** (*array_like*) – The x where the proximal should be evaluated.

- ***args** – The time at which the proximal should be evaluated. Not required if the cost is static.
- **penalty** (*float, optional*) – The penalty parameter ρ for the proximal evaluation. Defaults to 1.
- ****kwargs** – Any other required argument.

class tvopt.costs.**RobustLinearRegression**(*A*, *b*, *threshold*)

Bases: `Cost`

Cost for robust linear regression.

Let $h : \mathbb{R} \to \mathbb{R}$ be the Huber loss, then the cost is defined as:

$$f(\boldsymbol{x}) = \sum_{i=1}^{m} h(a_i \boldsymbol{x} - b_i)$$

where $a_i \in \mathbb{R}^{1 \times n}$ are the rows of the data matrix $\boldsymbol{A} \in \mathbb{R}^{m \times n}$, and b_i the elements of the data vector $\boldsymbol{b}$.

function(*x*)

An evaluation of the cost. *Implement if needed.*

Parameters

- **x** (*array_like*) – The x where the cost should be evaluated.
- ***args** – The time at which the cost should be evaluated. Not required if the cost is static.
- ****kwargs** – Any other required argument.

gradient(*x*)

An evaluation of the cost's gradient or sub-gradient. *Implement if needed.*

Parameters

- **x** (*array_like*) – The x where the (sub-)gradient should be evaluated.
- ***args** – The time at which the (sub-)gradient should be evaluated. Not required if the cost is static.
- ****kwargs** – Any other required argument.

hessian(*x*)

An evaluation of the cost's Hessian. *Implement if needed.*

Parameters

- **x** (*array_like*) – The x where the Hessian should be evaluated.
- ***args** – The time at which the Hessian should be evaluated. Not required if the cost is static.
- ****kwargs** – Any other required argument.

class tvopt.costs.**SampledCost**(*cost*, *t*)

Bases: `Cost`

Sampled cost.

This class defines a *static* cost by sampling a *dynamic* cost at a given time.

function(*x*, ***kwargs*)

An evaluation of the cost. *Implement if needed.*

Parameters

- **x** (*array_like*) – The x where the cost should be evaluated.
- ***args** – The time at which the cost should be evaluated. Not required if the cost is static.
- ****kwargs** – Any other required argument.

gradient(*x*, ***kwargs*)

An evaluation of the cost's gradient or sub-gradient. *Implement if needed.*

Parameters

- **x** (*array_like*) – The x where the (sub-)gradient should be evaluated.
- ***args** – The time at which the (sub-)gradient should be evaluated. Not required if the cost is static.
- ****kwargs** – Any other required argument.

hessian(*x*, ***kwargs*)

An evaluation of the cost's Hessian. *Implement if needed.*

Parameters

- **x** (*array_like*) – The x where the Hessian should be evaluated.
- ***args** – The time at which the Hessian should be evaluated. Not required if the cost is static.
- ****kwargs** – Any other required argument.

proximal(*x*, ***kwargs*)

An evaluation of the cost's proximal.

If this method is not overwritten, the default behavior is to recursively compute the proximal via a gradient or Newton backtracking algorithm. See *compute_proximal* for the function that is used for this purpose.

Parameters

- **x** (*array_like*) – The x where the proximal should be evaluated.
- ***args** – The time at which the proximal should be evaluated. Not required if the cost is static.
- **penalty** (*float, optional*) – The penalty parameter ρ for the proximal evaluation. Defaults to 1.
- ****kwargs** – Any other required argument.

class tvopt.costs.**ScaledCost**(*cost*, *s*)

Bases: *Cost*

Scaled cost.

This class defines a cost scaled by a constant. That is, given the cost $f : \mathbb{R}^n \times \mathbb{R}_+ \to \mathbb{R} \cup \{+\infty\}$ and $c \in \mathbb{R}$ it defines:

$$g(\boldsymbol{x}; t) = c f(\boldsymbol{x}; t).$$

The class is used for the product and division by a constant.

function(**args*, ***kwargs*)

An evaluation of the cost. *Implement if needed*.

Parameters

- **x** (*array_like*) – The x where the cost should be evaluated.
- ***args** – The time at which the cost should be evaluated. Not required if the cost is static.
- ****kwargs** – Any other required argument.

gradient(**args*, ***kwargs*)

An evaluation of the cost's gradient or sub-gradient. *Implement if needed*.

Parameters

- **x** (*array_like*) – The x where the (sub-)gradient should be evaluated.
- ***args** – The time at which the (sub-)gradient should be evaluated. Not required if the cost is static.
- ****kwargs** – Any other required argument.

hessian(**args*, ***kwargs*)

An evaluation of the cost's Hessian. *Implement if needed*.

Parameters

- **x** (*array_like*) – The x where the Hessian should be evaluated.
- ***args** – The time at which the Hessian should be evaluated. Not required if the cost is static.
- ****kwargs** – Any other required argument.

proximal(**args*, ***kwargs*)

An evaluation of the cost's proximal.

If this method is not overwritten, the default behavior is to recursively compute the proximal via a gradient or Newton backtracking algorithm. See *compute_proximal* for the function that is used for this purpose.

Parameters

- **x** (*array_like*) – The x where the proximal should be evaluated.
- ***args** – The time at which the proximal should be evaluated. Not required if the cost is static.
- **penalty** (*float, optional*) – The penalty parameter ρ for the proximal evaluation. Defaults to 1.
- ****kwargs** – Any other required argument.

class tvopt.costs.**SeparableCost**(*costs*)

Bases: *Cost*

Separable cost function.

This class defines a separable cost, that is

$$f(\boldsymbol{x}; t) = \sum_{i=1}^{N} f_i(x_i; t)$$

where $x_i \in \mathbb{R}^{n_1 \times n_2 \times \cdots}$ for each $i = 1, \ldots, N$. Each of the component functions f_i can be either static or dynamic. This is useful for defining distributed optimization problems.

The overall dimension of the domain is $n_1 \times n_2 \times \ldots \times N$, meaning that the last dimension indexes the components.

The class exposes the same methods as any *Cost*, with the difference that the keyword argument *i* can be used to evaluate only a single component. If all components are evaluated, an ndarray is returned with the last dimension indexing the components.

The class has the *Cost* attributes, with the following additions or differences.

costs

The component costs.

Type
list

N

The number of components.

Type
int

is_dynamic

True if at least one component is dynamic.

Type
bool

smooth

This is the minimum of the smoothness degrees of all components.

Type
int

function(*x*, **args*, *i=None*, ***kwargs*)

An evaluation of the cost. *Implement if needed.*

Parameters

- **x** (*array_like*) – The x where the cost should be evaluated.
- ***args** – The time at which the cost should be evaluated. Not required if the cost is static.
- ****kwargs** – Any other required argument.

gradient(*x*, **args*, *i=None*, ***kwargs*)

An evaluation of the cost's gradient or sub-gradient. *Implement if needed.*

Parameters

- **x** (*array_like*) – The x where the (sub-)gradient should be evaluated.
- ***args** – The time at which the (sub-)gradient should be evaluated. Not required if the cost is static.
- ****kwargs** – Any other required argument.

hessian(*x*, **args*, *i=None*, ***kwargs*)

An evaluation of the cost's Hessian. *Implement if needed.*

Parameters

- **x** (*array_like*) – The x where the Hessian should be evaluated.

- ***args** – The time at which the Hessian should be evaluated. Not required if the cost is static.
- ****kwargs** – Any other required argument.

proximal(*x*, **args*, *penalty=1*, *i=None*, ***kwargs*)

An evaluation of the cost(s) proximal(s).

This is the same as calling _evaluate with "proximal", with the difference that is customized to handle the penalty parameter. In particular, the penalty can either be a scalar, in which case the same penalty is used for all components, or a list of component-wise penalties.

class tvopt.costs.**SumCost**(**costs*)

Bases: `Cost`

Sum of costs.

This class defines a cost as the sum of an arbitrary number of costs. That is, given the costs $f_i : \mathbb{R}^n \times \mathbb{R}_+ \to \mathbb{R} \cup \{+\infty\}$ with $i = 1, \dots, N$, the class defines:

$$f(\boldsymbol{x}; t) = \sum_{i=1}^{N} f_i(\boldsymbol{x}; t)$$

The *function*, *gradient* and *hessian* are defined from the components' methods using the sum rule, while the proximal by default is computed recursively.

function(*x*, **args*, ***kwargs*)

An evaluation of the cost. *Implement if needed.*

Parameters

- **x** (*array_like*) – The x where the cost should be evaluated.
- ***args** – The time at which the cost should be evaluated. Not required if the cost is static.
- ****kwargs** – Any other required argument.

gradient(*x*, **args*, ***kwargs*)

An evaluation of the cost's gradient or sub-gradient. *Implement if needed.*

Parameters

- **x** (*array_like*) – The x where the (sub-)gradient should be evaluated.
- ***args** – The time at which the (sub-)gradient should be evaluated. Not required if the cost is static.
- ****kwargs** – Any other required argument.

hessian(*x*, **args*, ***kwargs*)

An evaluation of the cost's Hessian. *Implement if needed.*

Parameters

- **x** (*array_like*) – The x where the Hessian should be evaluated.
- ***args** – The time at which the Hessian should be evaluated. Not required if the cost is static.
- ****kwargs** – Any other required argument.

`tvopt.costs.`**`backward_finite_difference`**(*signal*, *t*, *order=1*, *step=1*)

Compute the backward finite difference of a signal.

This function computes an approximate derivative of a given signal using backward finite differences. Given the signal $s(t)$, it computes:

$$s^o(t) = \sum_{i=0}^{o} (-1)^i \binom{o}{i} s(t - iT_s)/T_s^o$$

where $o \in \mathbb{N}$ is the derivative order and T_s is the sampling time, see[5] for more details.

Notice that if samples before $t = 0$ are required, they are set to zero.

Parameters

- **signal** – A function of a single scalar argument that represents the signal.
- **t** (*float*) – The time where the derivative should be evaluated.
- **order** (*int, optional*) – The derivative order, defaults to 1.
- **step** (*float, optional*) – The sampling time, defaults to 1.

Raises

ValueError – For invalid *order* or *step* arguments.

Returns

The approximate derivative.

Return type

ndarray

References

`tvopt.costs.`**`compute_proximal`**(*f*, *x*, *penalty*, *solver=None*, ***kwargs*)

Compute the proximal of a cost.

This function (approximately) computes the proximal of a given cost if there is no closed form solution. The function uses either a Newton method or a gradient method, both with backtracking line search.

Parameters

- **f** (Cost) – The static cost whose proximal is required.
- **x** (*array_like*) – Where the proximal has to be evaluated.
- **penalty** (*float*) – The penalty of the proximal.
- **solver** (*str, optional*) – The method to use for computing the proximal, Newton or gradient. If not specified, Newton is used for twice differentiable function, gradient otherwise.
- ****kwargs** (*dict*) – Parameters for the Newton or gradient method.

Returns

y – The proximal.

Return type

ndarray

[5] A. Quarteroni, R. Sacco, and F. Saleri, Numerical mathematics, 2nd ed. Berlin; New York: Springer, 2007.

See also:

`solvers.backtracking_gradient`, `solvers.newton`

1.3 tvopt.distributed_solvers module

Distributed solvers.

`tvopt.distributed_solvers.`**`admm`**(*problem*, *penalty*, *rel*, *w_0=0*, *num_iter=100*)

Distributed relaxed alternating direction method of multipliers (ADMM).

This function implements the distributed ADMM, see[6] and references therein. The algorithm is characterized by the following updates

$$x_i^\ell = \text{prox}_{f_i/(\rho d_i)}([\boldsymbol{A}^\top z^\ell]_i/(\rho d_i))$$

$$z_{ij}^{\ell+1} = (1-\alpha)z_{ij}^\ell - \alpha z_{ji}^\ell + 2\alpha\rho x_j^\ell$$

for $\ell = 0, 1, \ldots$, where d_i is node i's degree, ρ and α are the penalty and relaxation parameters, and $\boldsymbol{A}$ is the arc incidence matrix. The algorithm is guaranteed to converge to the optimal solution.

Parameters

- **`problem`** (*`dict`*) – A dictionary containing the network describing the multi-agent system and the cost describing the problem.
- **`penalty`** (*`float`*) – The penalty parameter ρ of the algorithm (convergence is guaranteed for any positive value).
- **`rel`** (*`float`*) – The relaxation parameter α of the algorithm (convergence is guaranteed for values in $(0, 1)$).
- **`w_0`** (*`ndarray, optional`*) – The initial value of the dual nodes' states. This can be either an ndarray of suitable size with the last dimension indexing the nodes, or a scalar. If it is a scalar then the same initial value is used for all components.
- **`num_iter`** (*`int, optional`*) – The number of iterations to be performed.

Returns

- **x** (*ndarray*) – The nodes' states after *num_iter* iterations.
- **w** (*ndarray*) – The dual variables of the nodes after *num_iter* iterations.

References

`tvopt.distributed_solvers.`**`aug_dgm`**(*problem*, *step*, *x_0=0*, *num_iter=100*)

Augmented distributed gradient method (Aug-DGM).

This function implements the Aug-DGM algorithm (see[7]). The algorithm is characterized by the following updates

$$\boldsymbol{y}^\ell = \boldsymbol{W}\left(\boldsymbol{y}^{\ell-1} + \nabla f(\boldsymbol{x}^\ell) - \nabla f(\boldsymbol{x}^{\ell-1})\right) \quad (1.1)$$

$$\boldsymbol{x}^{\ell+1} = \boldsymbol{W}\left(\boldsymbol{x}^\ell - \boldsymbol{A}\boldsymbol{y}^\ell\right) \quad (1.2)$$

[6] N. Bastianello, R. Carli, L. Schenato, and M. Todescato, "Asynchronous Distributed Optimization over Lossy Networks via Relaxed ADMM: Stability and Linear Convergence," IEEE Transactions on Automatic Control.

[7] J. Xu, S. Zhu, Y. C. Soh, and L. Xie, "Augmented distributed gradient methods for multi-agent optimization under uncoordinated constant stepsizes," in 2015 54th IEEE Conference on Decision and Control (CDC), Osaka, Japan, Dec. 2015, pp. 2055–2060.

for $\ell = 0, 1, \dots$ where $\boldsymbol{A}$ is a diagonal matrix of uncoordinated step-sizes. The algorithm is guaranteed to converge to the optimal solution.

Parameters

- **problem** (*dict*) – A dictionary containing the network describing the multi-agent system and the cost describing the problem.
- **step** (*float*) – A common step-size or a list of local step-sizes, one for each node.
- **x_0** (*ndarray, optional*) – The initial states of the nodes. This can be either an ndarray of suitable size with the last dimension indexing the nodes, or a scalar. If it is a scalar then the same initial value is used for all components of the states.
- **num_iter** (*int, optional*) – The number of iterations to be performed.

Returns
x – The nodes' states after *num_iter* iterations.

Return type
ndarray

References

`tvopt.distributed_solvers.`**`average_consensus`**(*net*, *x_0*, *num_iter=100*)

Average consensus.

Compute the average consensus over the network *net* with initial states *x_0*.

Parameters

- **net** (`networks.Network`) – The network describing the multi-agent system.
- **x_0** (*ndarray*) – The initial states in a ndarray, with the last dimension indexing the nodes.
- **num_iter** (*int, optional*) – The number of iterations to be performed.

Returns
x – The nodes' states after *num_iter* iterations.

Return type
ndarray

`tvopt.distributed_solvers.`**`dpgm`**(*problem*, *step*, *x_0=0*, *num_iter=100*)

Distributed proximal gradient method (DPGM).

This function implements the DPGM algorithm proposed in[8] (see also[9] for the gradient-only version). The algorithm is characterized by the following updates

$$\begin{aligned} \boldsymbol{y}^\ell &= \boldsymbol{W}\boldsymbol{x}^\ell - \alpha \nabla f(\boldsymbol{x}^\ell) \\ \boldsymbol{x}^{\ell+1} &= \mathrm{prox}_{\alpha g}(\boldsymbol{y}^\ell) \end{aligned} \tag{1.3}$$

for $\ell = 0, 1, \dots$. The algorithm is guaranteed to converge to a neighborhood of the optimal solution.

Parameters

- **problem** (*dict*) – A dictionary containing the network describing the multi-agent system and the costs describing the (possibly composite) problem. The dictionary should contain f and the network, and optionally g.

[8] Bastianello, N., Ajalloeian, A., & Dall'Anese, E. (2020). Distributed and Inexact Proximal Gradient Method for Online Convex Optimization. arXiv preprint arXiv:2001.00870.

[9] Yuan, K., Ling, Q., & Yin, W. (2016). On the convergence of decentralized gradient descent. SIAM Journal on Optimization, 26(3), 1835-1854.

- **step** (*float*) – The step-size.
- **x_0** (*ndarray, optional*) – The initial states of the nodes. This can be either an ndarray of suitable size with the last dimension indexing the nodes, or a scalar. If it is a scalar then the same initial value is used for all components of the states.
- **num_iter** (*int, optional*) – The number of iterations to be performed.

Returns
x – The nodes' states after *num_iter* iterations.

Return type
ndarray

References

`tvopt.distributed_solvers.`**`dual_ascent`**(*problem*, *step*, *w_0=0*, *num_iter=100*)

Distributed dual ascent a.k.a. dual decomposition (DD).

This function implements the DD algorithm[10]. The algorithm is characterized by the following updates

$$\boldsymbol{x}^\ell = \arg\min_{\boldsymbol{x}} \left\{ f(\boldsymbol{x}) - \langle (\boldsymbol{I} - \boldsymbol{W})\boldsymbol{x}, \boldsymbol{w}^\ell \rangle \right\}$$

$$\boldsymbol{w}^{\ell+1} = \boldsymbol{w}^\ell - \alpha(\boldsymbol{I} - \boldsymbol{W})\boldsymbol{x}^\ell$$

for $\ell = 0, 1, \ldots$, where $\boldsymbol{w}$ is the vector of Lagrange multipliers. The algorithm is guaranteed to converge to the optimal solution.

Parameters

- **system** (*A dictionary containing the network describing the multi-agent*) – and the cost describing the problem.
- **step** (*float*) – The step-size.
- **w_0** (*ndarray, optional*) – The initial value of the dual nodes' states. This can be either an ndarray of suitable size with the last dimension indexing the nodes, or a scalar. If it is a scalar then the same initial value is used for all components.
- **num_iter** (*int, optional*) – The number of iterations to be performed.

Returns

- **x** (*ndarray*) – The nodes' states after *num_iter* iterations.
- **w** (*ndarray*) – The dual variables of the nodes after *num_iter* iterations.

References

`tvopt.distributed_solvers.`**`gossip_consensus`**(*net*, *x_0*, *num_iter=100*, *q=0.5*)

Average consensus.

Compute the average consensus over the network *net* with initial states *x_0* using the symmetric gossip protocol.

Parameters

- **net** (`networks.Network`) – The network describing the multi-agent system.
- **x_0** (*ndarray*) – The initial states in a ndarray, with the last dimension indexing the nodes.

[10] Simonetto, A. (2018). Dual Prediction–Correction Methods for Linearly Constrained Time-Varying Convex Programs. IEEE Transactions on Automatic Control, 64(8), 3355-3361.

- **num_iter** (*int, optional*) – The number of iterations to be performed.
- **q** (*float, optional*) – The weight used in the convex combination of the nodes that communicate at each iteration.

Returns
x – The nodes' states after *num_iter* iterations.

Return type
ndarray

tvopt.distributed_solvers.**max_consensus**(*net*, *x_0*, *num_iter=100*)

Max consensus.

Compute the maximum of the nodes' states *x_0*.

Parameters

- **net** (networks.Network) – The network describing the multi-agent system.
- **x_0** (*ndarray*) – The initial states in a ndarray, with the last dimension indexing the nodes.
- **num_iter** (*int, optional*) – The number of iterations to be performed.

Returns
x – The nodes' states after *num_iter* iterations.

Return type
ndarray

tvopt.distributed_solvers.**nids**(*problem*, *step*, *x_0=0*, *num_iter=100*)

Network InDependent Step-size algorithm (NIDS).

This function implements the NIDS algorithm proposed in[11]. The algorithm is characterized by the following updates

$$\begin{aligned} \boldsymbol{y}^\ell &= \boldsymbol{y}^{\ell-1} - \boldsymbol{x}^\ell - \tilde{\boldsymbol{W}}(2\boldsymbol{x}^\ell - \boldsymbol{x}^{\ell-1} - \mathrm{diag}(\boldsymbol{\alpha})(\nabla f(\boldsymbol{x}^\ell) - \nabla f(\boldsymbol{x}^{\ell-1}))) \\ \boldsymbol{x}^{\ell+1} &= \mathrm{prox}_{\boldsymbol{\alpha} g}(\boldsymbol{y}^\ell) \end{aligned} \tag{1.5}$$

for $\ell = 0, 1, \ldots$, where $\boldsymbol{\alpha}$ is a column vector containing the independent step-sizes of the nodes, and

$$\tilde{\boldsymbol{W}} = \boldsymbol{I} + c\,\mathrm{diag}(\boldsymbol{\alpha})(\boldsymbol{W} - \boldsymbol{I})$$

with $c = 0.5/\max_i\{\alpha_i\}$. The algorithm is guaranteed to converge to the optimal solution.

Parameters

- **problem** (*dict*) – A dictionary containing the network describing the multi-agent system and the costs describing the (possibly composite) problem. The dictionary should contain f and the network, and optionally g.
- **step** (*float or list*) – A common step-size or a list of local step-sizes, one for each node.
- **x_0** (*ndarray, optional*) – The initial states of the nodes. This can be either an ndarray of suitable size with the last dimension indexing the nodes, or a scalar. If it is a scalar then the same initial value is used for all components of the states.
- **num_iter** (*int, optional*) – The number of iterations to be performed.

[11] Li, Z., Shi, W., & Yan, M. (2019). A decentralized proximal-gradient method with network independent step-sizes and separated convergence rates. IEEE Transactions on Signal Processing, 67(17), 4494-4506.

Returns
x – The nodes' states after *num_iter* iterations.

Return type
ndarray

References

`tvopt.distributed_solvers.`**`pg_extra`**(*problem*, *step*, *x_0=0*, *num_iter=100*)

Proximal gradient exact first-order algorithm (PG-EXTRA).

This function implements the PG-EXTRA algorithm proposed in[12] (see also[13] for the gradient-only version, EXTRA). The algorithm is characterized by the following updates

$$\begin{aligned} \boldsymbol{y}^\ell &= \boldsymbol{y}^{\ell-1} + \boldsymbol{W}\boldsymbol{x}^\ell - \tilde{\boldsymbol{W}}\boldsymbol{x}^{\ell-1} - \alpha(\nabla f(\boldsymbol{x}^\ell) - \nabla f(\boldsymbol{x}^{\ell-1})) \\ \boldsymbol{x}^{\ell+1} &= \mathrm{prox}_{\alpha g}(\boldsymbol{y}^\ell) \end{aligned} \tag{1.7}$$

for $\ell = 0, 1, \ldots$, where $\tilde{\boldsymbol{W}} = (\boldsymbol{I} + \boldsymbol{W})/2$. The algorithm is guaranteed to converge to the optimal solution.

Parameters

- **`problem`** (*`dict`*) – A dictionary containing the network describing the multi-agent system and the costs describing the (possibly composite) problem. The dictionary should contain f and the network, and optionally g.
- **`step`** (*`float`*) – The step-size.
- **`x_0`** (*`ndarray, optional`*) – The initial states of the nodes. This can be either an ndarray of suitable size with the last dimension indexing the nodes, or a scalar. If it is a scalar then the same initial value is used for all components of the states.
- **`num_iter`** (*`int, optional`*) – The number of iterations to be performed.

Returns
x – The nodes' states after *num_iter* iterations.

Return type
ndarray

References

`tvopt.distributed_solvers.`**`prox_aac`**(*problem*, *step*, *x_0=0*, *num_iter=100*, *consensus_steps=[True, True, True]*)

Proximal adapt-and-combine (Prox-AAC).

This function implements the Prox-AAC algorithm (see[1] for the gradient only version). The algorithm is characterized by the following updates

$$\boldsymbol{z}^\ell = \boldsymbol{W}_1\boldsymbol{x}^\ell$$

$$\boldsymbol{y}^\ell = \boldsymbol{z}^\ell - \alpha\nabla f(\boldsymbol{z}^\ell)$$

[12] Shi, W., Ling, Q., Wu, G., & Yin, W. (2015). A proximal gradient algorithm for decentralized composite optimization. IEEE Transactions on Signal Processing, 63(22), 6013-6023.
[13] Shi, W., Ling, Q., Wu, G., & Yin, W. (2015). Extra: An exact first-order algorithm for decentralized consensus optimization. SIAM Journal on Optimization, 25(2), 944-966.
[1] Chen, J., & Sayed, A. H. (2013). Distributed Pareto optimization via diffusion strategies. IEEE Journal of Selected Topics in Signal Processing, 7(2), 205-220.

$$\boldsymbol{x}^{\ell+1} = \boldsymbol{W}_3 \operatorname{prox}_{\alpha g}(\boldsymbol{W}_2 \boldsymbol{y}^{\ell})$$

for $\ell = 0, 1, \ldots$, where $\boldsymbol{W}_1$, $\boldsymbol{W}_2$ and $\boldsymbol{W}_3$ are doubly stochastic matrices (or the identity).

Parameters

- **problem** (*dict*) – A dictionary containing the network describing the multi-agent system and the costs describing the (possibly composite) problem. The dictionary should contain f and the network, and optionally g.
- **step** (*float or list*) – A common step-size or a list of local step-sizes, one for each node.
- **x_0** (*ndarray, optional*) – The initial states of the nodes. This can be either an ndarray of suitable size with the last dimension indexing the nodes, or a scalar. If it is a scalar then the same initial value is used for all components of the states.
- **num_iter** (*int, optional*) – The number of iterations to be performed.
- **consensus_steps** (*list*) – A list specifying which consensus steps to perform; the list must have three elements that can be interpreted as bools.

Returns

x – The nodes' states after *num_iter* iterations.

Return type

ndarray

References

`tvopt.distributed_solvers.prox_ed`(*problem*, *step*, *x_0=0*, *num_iter=100*)

Proximal exact diffusion (Prox-ED).

This function implements the Prox-ED algorithm[14]. The algorithm is characterized by the following updates

$$\boldsymbol{y}^{\ell} = \boldsymbol{x}^{\ell} - \alpha \nabla f(\boldsymbol{x}^{\ell}) \tag{1.9}$$

$$\boldsymbol{u}^{\ell} = \boldsymbol{z}^{\ell-1} + \boldsymbol{y}^{\ell} - \boldsymbol{y}^{\ell-1} \tag{1.10}$$

$$\boldsymbol{z}^{\ell} = \tilde{\boldsymbol{W}} \boldsymbol{u}^{\ell} \tag{1.11}$$

$$\boldsymbol{x}^{\ell+1} = \operatorname{prox}_{\alpha g}(\boldsymbol{z}^{\ell}) \tag{1.12}$$

for $\ell = 0, 1, \ldots$, where $\tilde{\boldsymbol{W}} = (\boldsymbol{I} + \boldsymbol{W})/2$. The algorithm is guaranteed to converge to the optimal solution.

Parameters

- **problem** (*dict*) – A dictionary containing the network describing the multi-agent system and the costs describing the (possibly composite) problem. The dictionary should contain f and the network, and optionally g.
- **step** (*float*) – The step-size.
- **x_0** (*ndarray, optional*) – The initial states of the nodes. This can be either an ndarray of suitable size with the last dimension indexing the nodes, or a scalar. If it is a scalar then the same initial value is used for all components of the states.
- **num_iter** (*int, optional*) – The number of iterations to be performed.

Returns

x – The nodes' states after *num_iter* iterations.

[14] S. A. Alghunaim, E. Ryu, K. Yuan, and A. H. Sayed, "Decentralized Proximal Gradient Algorithms with Linear Convergence Rates," IEEE Transactions on Automatic Control, 2020.

Return type
ndarray

References

`tvopt.distributed_solvers.ratio_consensus`(*net*, *x_0*, *num_iter=100*)

Ratio consensus.

Compute the average consensus over the network *net* with initial states *x_0* using the ratio consensus protocol.

Parameters

- **net** (`networks.Network`) – The network describing the multi-agent system.
- **x_0** (*ndarray*) – The initial states in a ndarray, with the last dimension indexing the nodes.
- **num_iter** (*int, optional*) – The number of iterations to be performed.

Returns
x – The nodes' states after *num_iter* iterations.

Return type
ndarray

1.4 tvopt.networks module

Network tools.

`class tvopt.networks.DynamicNetwork`(*nets*, *t_s=1*)

Bases: *Network*

Time-varying network.

This class creates a time-varying network from a list of network objects, and possibly a sampling time that specifies how often the network changes.

`broadcast`(*t*, **args*, ***kwargs*)

Broadcast transmission.

This method implements a broadcast transmission in which a node sends the same packet to all its neighbors. The packet is also transmitted to the node itself. The method is implemented using the *send* method.

Parameters

- **sender** (*int*) – The index of the transmitting node.
- **packet** (*array_like*) – The packet to ne communicated.

`consensus`(*t*, **args*, ***kwargs*)

Consensus mixing.

This method implements a consensus step over the network, mixing the given nodes' states using the weight matrix of the network or a different one.

Parameters

- **x** (*array_like*) – The nodes' local states in an array with the last dimension indexing the nodes.
- **weights** (*ndarray, optional*) – The consensus weight matrix to be used instead of the one created at initialization.

Returns
y – The local states after a consensus step.

Return type
ndarray

max_consensus(*t*, **args*, ***kwargs*)

Max-consensus.

This method implements a step of max-consensus, where each node selects the (element-wise) maximum between the packets received from its neighbors and its own state. See[15] for a reference on max-consensus.

Parameters
x (*array_like*) – The nodes' local states in an array with the last dimension indexing the nodes.

Returns
x – The local states after a max-consensus step.

Return type
ndarray

References

sample(*t*)

Sample the dynamic network.

This method returns the network object that is active at time *t*.

Parameters
t (*float*) – The time when the network should be sampled.

Returns
The sampled network.

Return type
Network

send(*t*, **args*, ***kwargs*)

Node-to-node transmission (sender phase).

This method simulates a node-to-node transmission by storing the packet to be communicated in the *buffer*. In particular, if i is the sender and j the receiver, then the packet is introduced in *buffer* with keyword (j, i).

Note that older information (if any) in the *buffer* is overwritten whenever *send* is called.

Parameters

- **sender** (*int*) – The index of the transmitting node.
- **receiver** (*int*) – The index of the recipient.
- **packet** (*array_like*) – The packet to ne communicated.

class tvopt.networks.LossyNetwork(*adj_mat*, *loss_prob*, *weights=None*)

Bases: *Network*

Network with random communication failures.

[15] F. Iutzeler, P. Ciblat, and J. Jakubowicz, "Analysis of Max-Consensus Algorithms in Wireless Channels," IEEE Transactions on Signal Processing, vol. 60, no. 11, pp. 6103–6107, Nov. 2012.

Representation of a connected, undirected network, whose communication protocol is subject to packet losses. Packet sent from a node to another may be lost with a certain probability.

send(*sender*, *receiver*, *packet*)

Node-to-node transmission (sender phase).

This method simulates a node-to-node transmission by storing the packet to be communicated in the *buffer*. In particular, if i is the sender and j the receiver, then the packet is introduced in *buffer* with keyword (j, i).

Note that older information (if any) in the *buffer* is overwritten whenever *send* is called.

Parameters

- **sender** (*int*) – The index of the transmitting node.
- **receiver** (*int*) – The index of the recipient.
- **packet** (*array_like*) – The packet to ne communicated.

class tvopt.networks.**Network**(*adj_mat*, *weights=None*)

Bases: `object`

Representation of an undirected network.

The class implements an undirected network defined from the adjacency matrix. The class provides methods for different communication protocols, such as node-to-node and broadcast.

Transmissions are implemented via the *buffer* attribute of the network: the sender stores the packet to be transmitted in the *buffer* dictionary, specifying the recipient, which can then access the packet.

By convention, the nodes in the network are indexed from 0 to $N - 1$, where N is the total number of nodes.

adj_mat

The adjacency matrix of the network.

Type
ndarray

N

The number of nodes in the network.

Type
ndarray

weights

The consensus weight matrix, if not specified in the constructor this is the Metropolis-Hastings weight matrix.

Type
ndarray

neighbors

A list whose i-th element is a list of node i's neighors.

Type
list

degrees

The number of neighbors of each node.

Type
list

buffer

The dictionary used for node-to-node transmissions.

Type
dict

broadcast(*sender*, *packet*)

Broadcast transmission.

This method implements a broadcast transmission in which a node sends the same packet to all its neighbors. The packet is also transmitted to the node itself. The method is implemented using the *send* method.

Parameters

- **sender** (*int*) – The index of the transmitting node.
- **packet** (*array_like*) – The packet to ne communicated.

consensus(*x*, *weights=None*)

Consensus mixing.

This method implements a consensus step over the network, mixing the given nodes' states using the weight matrix of the network or a different one.

Parameters

- **x** (*array_like*) – The nodes' local states in an array with the last dimension indexing the nodes.
- **weights** (*ndarray, optional*) – The consensus weight matrix to be used instead of the one created at initialization.

Returns
y – The local states after a consensus step.

Return type
ndarray

max_consensus(*x*)

Max-consensus.

This method implements a step of max-consensus, where each node selects the (element-wise) maximum between the packets received from its neighbors and its own state. See[16] for a reference on max-consensus.

Parameters
x (*array_like*) – The nodes' local states in an array with the last dimension indexing the nodes.

Returns
x – The local states after a max-consensus step.

Return type
ndarray

[16] F. Iutzeler, P. Ciblat, and J. Jakubowicz, "Analysis of Max-Consensus Algorithms in Wireless Channels," IEEE Transactions on Signal Processing, vol. 60, no. 11, pp. 6103–6107, Nov. 2012.

References

receive(*receiver*, *sender*, *default=0*, *destructive=True*)

Node-to-node transmission (receiver phase).

This method simulates the reception of a packet previously transmitted using the *send* method. In patricular, the method accesses the packet in the *buffer* dictionary. If the packet is not present, a default value is returned.

Reads from the *buffer* can be destructive, meaning that the packet is read and removed, which is the default, or not.

Parameters

- **receiver** (*int*) – The index of the recipient.
- **sender** (*int*) – The index of the transmitting node.
- **default** (*array_like, optional*) – The value returned when a packet from *sender* to *receiver* is not found in the *buffer*.
- **destructive** (*bool, optional*) – Specifies if the packet should be removed from the *buffer* after being read (which is the default) or not.

Returns

The packet or a default value.

Return type

array_like

send(*sender*, *receiver*, *packet*)

Node-to-node transmission (sender phase).

This method simulates a node-to-node transmission by storing the packet to be communicated in the *buffer*. In particular, if i is the sender and j the receiver, then the packet is introduced in *buffer* with keyword (j, i).

Note that older information (if any) in the *buffer* is overwritten whenever *send* is called.

Parameters

- **sender** (*int*) – The index of the transmitting node.
- **receiver** (*int*) – The index of the recipient.
- **packet** (*array_like*) – The packet to ne communicated.

class tvopt.networks.**NoisyNetwork**(*adj_mat*, *noise_var*, *weights=None*)

Bases: *Network*

Network with Gaussian communication noise.

Representation of a connected, undirected network, whose communication protocol is subject to additive white Gaussian noise. The network's transmission methods add normal noise to all packets (unless they are sent from a node to itself).

send(*sender*, *receiver*, *packet*)

Node-to-node transmission (sender phase).

This method simulates a node-to-node transmission by storing the packet to be communicated in the *buffer*. In particular, if i is the sender and j the receiver, then the packet is introduced in *buffer* with keyword (j, i).

Note that older information (if any) in the *buffer* is overwritten whenever *send* is called.

Parameters

- **sender** (*int*) – The index of the transmitting node.
- **receiver** (*int*) – The index of the recipient.
- **packet** (*array_like*) – The packet to ne communicated.

class tvopt.networks.**QuantizedNetwork**(*adj_mat*, *step*, *thresholds=None*, *weights=None*)

Bases: *Network*

Network with quantized communications.

Representation of a connected, undirected network, whose communications are quantized. The network's transmission methods quantize all packets (unless they are sent from a node to itself).

send(*sender*, *receiver*, *packet*)

Node-to-node transmission (sender phase).

This method simulates a node-to-node transmission by storing the packet to be communicated in the *buffer*. In particular, if i is the sender and j the receiver, then the packet is introduced in *buffer* with keyword (j, i).

Note that older information (if any) in the *buffer* is overwritten whenever *send* is called.

Parameters

- **sender** (*int*) – The index of the transmitting node.
- **receiver** (*int*) – The index of the recipient.
- **packet** (*array_like*) – The packet to ne communicated.

tvopt.networks.**circle_graph**(*N*)

Generate a circle graph.

Parameters
N (*int*) – Number of nodes in the graph.

Returns
adj_mat – Adjacency matrix of the generated graph.

Return type
ndarray

See also:

circulant_graph
Circulant graph generator

tvopt.networks.**circulant_graph**(*N*, *num_conn*)

Generate a circulant graph.

Parameters

- **N** (*int*) – Number of nodes in the graph.
- **num_conn** (*int*) – Number of neighbors on each side of a node.

Returns
adj_mat – Adjacency matrix of the generated graph.

Return type
ndarray

Notes

If *num_conn* is larger than *N* / 2 a complete graph is returned.

`tvopt.networks.complete_graph(N)`

Generate a complete graph.

Parameters
N (*int*) – Number of nodes in the graph.

Returns
adj_mat – Adjacency matrix of the generated graph.

Return type
ndarray

See also:

circulant_graph
Circulant graph generator

`tvopt.networks.erdos_renyi(N, prob)`

Generate a random Erdos-Renyi graph.

Parameters

- **N** (*int*) – Number of nodes in the graph.
- **prob** (*float*) – The probability of adding an edge between any two nodes.

Returns
adj_mat – Adjacency matrix of the generated graph.

Return type
ndarray

Raises
ValueError. –

`tvopt.networks.incidence_matrix(adj_mat, n=1)`

Build the incidence matrix.

The edges $e = (i, j)$ are ordered with $i \leq j$, so that in the e-th column the i-th element is 1 and the j-th is -1 (the remaining are of course 0).

Parameters

- **adj_mat** (*ndarray*) – Adjacency matrix describing the graph.
- **n** (*int, optional*) – Size of the local states.

Returns
incid_mat – The incidence matrix.

Return type
ndarray

`tvopt.networks.is_connected(adj_mat)`

Verify if a graph is connected.

Parameters
adj_mat (*ndarray*) – Adjacency matrix describing the graph.

Returns
True if the graph is connected, False otherwise.

Return type
bool

Notes

The connectedness of the graph is checked by verifying whether the N-th power of the adjacency matrix plus the identity is a full matrix (no zero elements), with N the number of nodes.

`tvopt.networks.`**`metropolis_hastings`**`(`*adj_mat*`)`

Compute a consensus matrix based on the Metropolis-Hastings rule.

The Metropolis-Hastings rule generates a matrix W with off-diagonal elements equal to:

$$w_{ij} = \frac{1}{1 + \max\{d_i, d_j\}}$$

where i is a node index and $j \neq i$ the index of one of its neighbors, and d_i, d_j are their respective degrees. The diagonal elements are assigned as:

$$w_{ii} = 1 - \sum_{j \in \mathcal{N}_i} w_{ij}$$

to guarantee double stochasticity.

Parameters
`adj_mat` (*`ndarray`*) – Adjacency matrix describing the graph.

Returns
mh_mat – Metropolois-Hastings consensus matrix.

Return type
ndarray

`tvopt.networks.`**`random_graph`**`(`*N*, *radius*`)`

Generate a random geometric graph.

Parameters

- **`N`** (*`int`*) – Number of nodes in the graph.
- **`radius`** (*`float`*) – Radius of each node's neighborhood, must be in $[0, 1)$.

Returns
adj_mat – Adjacency matrix of the generated graph.

Return type
ndarray

Raises
`ValueError.` –

Notes

The function recursively generates random positions for the nodes on the $[0, 1] \times [0, 1]$ square, and then builds the graph by setting as neighbors each pair of nodes within a distance no larger than *radius*. The process is repeated until the result is a connected graph. For this reason, combinations of small N and *radius can yield exceedingly long computation times*. If the computation does not succeed after *2500* iterations, an error is raised.

`tvopt.networks.star_graph(N)`

Generate a star graph.

Parameters
N (*int*) – Number of nodes in the graph.

Returns
adj_mat – Adjacency matrix of the generated graph.

Return type
ndarray

1.5 tvopt.prediction module

Cost prediction tools.

`class tvopt.prediction.ExtrapolationPrediction(cost, order=2)`

Bases: *Prediction*

Extrapolation-based prediction.

This prediction strategy, proposed in[17], predicts the cost at time t_{k+1} as:

$$\hat{f}(\boldsymbol{x}; t_{k+1}) = \sum_{i=1}^{I} \ell_i f(\boldsymbol{x}; t_{k-i+1})$$

where $I \in \mathbb{N}$ denotes the order, that is, the number of past functions to use, and with coefficients:

$$\ell_i = \prod_{1 \leq j \leq I,\ j \neq i} \frac{j}{j - i}.$$

`update(t)`

Update the current prediction.

This method updates the current prediction by building a new predicted cost using the samples observed up to time *t*. By default this method samples the dynamic cost, and should be overwritten when implementing a custom prediction strategy.

Parameters
t (*float*) – The time of the last sampled cost.

[17] N. Bastianello, A. Simonetto, and R. Carli, "Primal and Dual Prediction-Correction Methods for Time-Varying Convex Optimization," arXiv:2004.11709 [cs, math], Oct. 2020. Available: http://arxiv.org/abs/2004.11709.

class tvopt.prediction.**Prediction**(*cost*)

Bases: `Cost`

Prediction of a dynamic cost.

This class creates a cost object that predicts a given dynamic function. The object stores a dynamic cost and a predicted cost, which can be modified using new information through the method *update*.

function(*x*, ***kwargs*)

An evaluation of the cost. *Implement if needed.*

Parameters

- **x** (*array_like*) – The x where the cost should be evaluated.
- ***args** – The time at which the cost should be evaluated. Not required if the cost is static.
- ****kwargs** – Any other required argument.

gradient(*x*, ***kwargs*)

An evaluation of the cost's gradient or sub-gradient. *Implement if needed.*

Parameters

- **x** (*array_like*) – The x where the (sub-)gradient should be evaluated.
- ***args** – The time at which the (sub-)gradient should be evaluated. Not required if the cost is static.
- ****kwargs** – Any other required argument.

hessian(*x*, ***kwargs*)

An evaluation of the cost's Hessian. *Implement if needed.*

Parameters

- **x** (*array_like*) – The x where the Hessian should be evaluated.
- ***args** – The time at which the Hessian should be evaluated. Not required if the cost is static.
- ****kwargs** – Any other required argument.

proximal(*x*, *penalty=1*, ***kwargs*)

An evaluation of the cost's proximal.

If this method is not overwritten, the default behavior is to recursively compute the proximal via a gradient or Newton backtracking algorithm. See *compute_proximal* for the function that is used for this purpose.

Parameters

- **x** (*array_like*) – The x where the proximal should be evaluated.
- ***args** – The time at which the proximal should be evaluated. Not required if the cost is static.
- **penalty** (*float, optional*) – The penalty parameter ρ for the proximal evaluation. Defaults to 1.
- ****kwargs** – Any other required argument.

update(*t*, **args*, ***kwargs*)

Update the current prediction.

This method updates the current prediction by building a new predicted cost using the samples observed up to time *t*. By default this method samples the dynamic cost, and should be overwritten when implementing a custom prediction strategy.

Parameters
t (*float*) – The time of the last sampled cost.

class tvopt.prediction.**TaylorPrediction**(*cost*)

Bases: *Prediction*

Taylor expansion-based prediction.

This prediction strategy, proposed in[18] and see also[19], predicts the cost at time t_{k+1} using its Taylor expansion around t_k and a given $\boldsymbol{x}_k$:

$$\hat{f}(\boldsymbol{x}; t_{k+1}) = f(\boldsymbol{x}_k; t_k) + \langle \nabla_x f(\boldsymbol{x}_k; t_k), \boldsymbol{x} - \boldsymbol{x}_k \rangle + T_s \nabla_t f(\boldsymbol{x}_k; t_k) + (T_s^2/2) \nabla_{tt} f(\boldsymbol{x}_k; t_k) + \\ + T_s \langle \nabla_{tx} f(\boldsymbol{x}_k; t_k), \boldsymbol{x} - \boldsymbol{x}_k \rangle + \frac{1}{2} (\boldsymbol{x} - \boldsymbol{x}_k)^\top \nabla_{xx} f(\boldsymbol{x}_k; t_k)(\boldsymbol{x} - \boldsymbol{x}_k)$$

where T_s is the sampling time.

References

update(*t*, *x*, *gradient_only=True*, ***kwargs*)

Update the current prediction.

This method updates the current prediction by building a new predicted cost using the samples observed up to time *t*. By default this method samples the dynamic cost, and should be overwritten when implementing a custom prediction strategy.

Parameters
t (*float*) – The time of the last sampled cost.

1.6 tvopt.sets module

Set template and examples.

class tvopt.sets.**AffineSet**(*A*, *b*)

Bases: *Set*

Affine set.

This class implements:

$$\{x \in \mathbb{R}^n \mid Ax = b\}$$

[18] A. Simonetto, A. Mokhtari, A. Koppel, G. Leus, and A. Ribeiro, “A Class of Prediction-Correction Methods for Time-Varying Convex Optimization,” IEEE Transactions on Signal Processing, vol. 64, no. 17, pp. 4576–4591, Sep. 2016.

[19] N. Bastianello, A. Simonetto, and R. Carli, “Primal and Dual Prediction-Correction Methods for Time-Varying Convex Optimization,” arXiv:2004.11709 [cs, math], Oct. 2020. Available: http://arxiv.org/abs/2004.11709.

for given matrix $A \in \mathbb{R}^{m \times n}$ and vector $b \in \mathbb{R}^m$.

contains(*x*)
Check if the input belongs to the set.

projection(*x*)
Project the input onto the set.

class tvopt.sets.**Ball**(*center*, *radius*)
Bases: *Set*

Ball set.

This class implements:

$$\{x \in \mathbb{R}^n \mid \|x - c\| \leq r\}$$

for a center c and radius $r > 0$.

contains(*x*)
Check if the input belongs to the set.

projection(*x*)
Project the input onto the set.

class tvopt.sets.**Ball_l1**(*center*, *radius*)
Bases: *Set*

ℓ_1-ball set.

This class implements:

$$\{x \in \mathbb{R}^n \mid \|x - c\|_1 \leq r\}$$

for a center c and radius $r > 0$.

contains(*x*)
Check if the input belongs to the set.

projection(*x*, *tol=1e-05*)
Project the input onto the set.

class tvopt.sets.**Box**(*l*, *u*, *n=1*)
Bases: *Set*

Box set.

This class implements:

$$\{x \in \mathbb{R}^n \mid l \leq x \leq u\}$$

with bounds l, u either scalar (applied element-wise) or vectors.

contains(x)
Check if the input belongs to the set.

projection(x)
Project the input onto the set.

class tvopt.sets.**Halfspace**(a, b)
Bases: *Set*

Halfspace.

This class implements:

$$\{x \in \mathbb{R}^n \mid \langle a, x \rangle \leq b\}$$

for given vetor $a \in \mathbb{R}^n$ and scalar $b \in \mathbb{R}$.

contains(x)
Check if the input belongs to the set.

projection(x)
Project the input onto the set.

class tvopt.sets.**IntersectionSet**(**sets*)
Bases: *Set*

Intersection of sets.

Given the sets $\mathbb{S}_i$, $i = 1, \ldots, N$ this class implements

$$\bigcap_{i=1}^{N} \mathbb{S}_i.$$

contains(x)
Check if the input belongs to the set.

projection(x, **args*, ***kwargs*)
Projection onto the intersection.

This method returns an approximate projection onto the intersection of sets, computed using the method of alternating projections.

See also:

alternating_projections
method of alternating projection

class tvopt.sets.**NonnegativeOrthant**(n)
Bases: *Set*

Non-negative orthant.

This class implements:

$$\{x \in \mathbb{R}^n \mid x \geq 0\}$$

where $x \geq 0$ if x is component-wise non-negative.

contains(*x*)

Check if the input belongs to the set.

projection(*x*)

Project the input onto the set.

class tvopt.sets.**R**(**dims*)

Bases: *Set*

The underlying space.

This class implements the underlying space $\mathbb{R}^{n_1 \times n_2 \times \cdots}$.

contains(*x*)

Check if the input belongs to the set.

projection(*x*)

Project the input onto the set.

class tvopt.sets.**ScaledSet**(*s*, *c*)

Bases: *Set*

Scaled set.

Given a set $\mathbb{S}$ and a scalar c, this class defines

$$\{cx \; \forall x \in \mathbb{S}\}.$$

contains(*x*)

Check if the input belongs to the set.

projection(*x*, **args*, ***kwargs*)

Project the input onto the set.

class tvopt.sets.**Set**(**dims*)

Bases: object

Template for a set.

This class defines a non-empty, closed, convex set in $\mathbb{R}^{n_1 \times n_2 \times \cdots}$. These objects are defined by a *contains* method (to check if an input belongs to the set) and a *projection* method.

Sets can be translated and scaled (via the respective methods). The *contains* method can also be accessed via the built-in *in* operator. Using + it is possible to intersect sets.

shape

The dimensions of the underlying space.

Type

tuple

ndim

The number of dimensions of the underlying space.

Type
int

size

The product of each dimension's size.

Type
int

check_input(*x*)

Check dimension of input.

This method verifies if the argument *x* belong to the space underlying the set, possibly reshaping it. If it is not compatible or cannot be reshaped (using numpy's broadcasting rules), and exception is raised.

Parameters
x (*array_like*) – The input to be checked.

Returns
The (possibly reshaped) input if it is compatible with the space.

Return type
ndarray

contains(*x*)

Check if the input belongs to the set.

projection(*x*, **args*, ***kwargs*)

Project the input onto the set.

scale(*c*)

Scale the set.

translate(*x*)

Translate the set.

class tvopt.sets.**T**(*t_s*, *t_min=0*, *t_max=inf*)

Bases: *Set*

Set of sampling times.

This class implements the set of sampling times:

$$\{t_k \geq 0,\ k \in \mathbb{N}\}$$

with $t_{k+1} - t_k = T_\mathrm{s}$ for a sampling time T_s.

check_input(*t*)

Check dimension of input.

This method verifies if the argument *x* belong to the space underlying the set, possibly reshaping it. If it is not compatible or cannot be reshaped (using numpy's broadcasting rules), and exception is raised.

Parameters
x (*array_like*) – The input to be checked.

Returns
The (possibly reshaped) input if it is compatible with the space.

Return type
ndarray

contains(*t*)
Check if the input belongs to the set.

projection(*t*)
Project the input onto the set.

scale(*c*)
Scale the set.

translate(*t*)
Translate the set.

class tvopt.sets.**TranslatedSet**(*s*, *t*)
Bases: *Set*

Translated set.

Given a set $\mathbb{S}$ and a vector t, this class defines

$$\{x + t \, \forall x \in \mathbb{S}\}.$$

contains(*x*)
Check if the input belongs to the set.

projection(*x*, **args*, ***kwargs*)
Project the input onto the set.

tvopt.sets.**alternating_projections**(*sets*, *x*, *tol=1e-10*, *num_iter=10*)
Method of alternating projections.

This function returns a point in the intersection of the given convex sets, computed using the method of alternating projections (MAP)[20].

Parameters

- **sets** (*list*) – The list of sets.
- **x** (*array_like*) – The starting point.
- **tol** (*float, optional*) – The stopping condition. If the difference between consecutive iterates is smaller than or equal to *tol*, then the function returns. Defaults to 10^{-10}.
- **num_iter** (*int, optional*) – The maximum number of iterations of the projection algorithm. Defaults to 1000. This stopping condition is enacted if the algorithm does not reach *tol*.

Returns
x – A point in the intersection.

Return type
ndarray

[20] H. Bauschke and V. Koch, “Projection Methods: Swiss Army Knives for Solving Feasibility and Best Approximation Problems with Halfspaces,” in Contemporary Mathematics, vol. 636, S. Reich and A. Zaslavski, Eds. Providence, Rhode Island: American Mathematical Society, 2015, pp. 1–40.

References

1.7 tvopt.solvers module

Solvers.

`tvopt.solvers.admm`(*problem*, *penalty*, *rel=1*, *w_0=0*, *num_iter=100*, *tol=None*)

Alternating direction method of multipliers (ADMM).

This function implements the ADMM to solve the constrained problem

$$\min_{\boldsymbol{x},\boldsymbol{y}} \{f(\boldsymbol{x}) + g(\boldsymbol{y})\} \tag{1.13}$$

$$\text{s.t. } \boldsymbol{A}\boldsymbol{x} + \boldsymbol{B}\boldsymbol{y} = \boldsymbol{c} \tag{1.14}$$

The algorithm is characterized by the updates:

$$\boldsymbol{x}^\ell = \arg\min_{\boldsymbol{x}} \left\{ f(\boldsymbol{x}) - \langle \boldsymbol{z}^\ell, \boldsymbol{A}\boldsymbol{x} \rangle + \frac{\rho}{2} \|\boldsymbol{A}\boldsymbol{x} - \boldsymbol{c}\|^2 \right\} \tag{1.15}$$

$$\boldsymbol{w}^\ell = \boldsymbol{z}^\ell - \rho(\boldsymbol{A}\boldsymbol{x}^\ell - \boldsymbol{c}) \tag{1.16}$$

$$\boldsymbol{y}^\ell = \arg\min_{\boldsymbol{y}} \left\{ g(\boldsymbol{y}) - \langle 2\boldsymbol{w}^\ell - \boldsymbol{z}^\ell, \boldsymbol{B}\boldsymbol{y} \rangle + \frac{\rho}{2} \|\boldsymbol{B}\boldsymbol{y}\|^2 \right\} \tag{1.17}$$

$$\boldsymbol{u}^\ell = 2\boldsymbol{w}^\ell - \boldsymbol{z}^\ell - \rho\boldsymbol{B}\boldsymbol{y}^\ell \tag{1.18}$$

$$\boldsymbol{z}^{\ell+1} = \boldsymbol{z}^\ell + 2\alpha(\boldsymbol{u}^\ell - \boldsymbol{w}^\ell) \tag{1.19}$$

for a given penalty $\rho > 0$ and $\alpha \in (0, 1]$ is the relaxation constant.

Parameters

- **problem** (*dict*) – Problem dictionary defining the costs f and g, and the constraints A, B and c.
- **penalty** (*float*) – The algorithm's penalty.
- **rel** (*float, optional*) – The relaxation constant.
- **w_0** (*array_like, optional*) – The dual initial condition. This can be either an ndarray of suitable size, or a scalar. If it is a scalar then the same initial value is used for all components of $\boldsymbol{w}$.
- **num_iter** (*int, optional*) – The number of iterations to be performed.
- **tol** (*float, optional*) – If given, this argument specifies the tolerance t in the dual stopping condition $\|\boldsymbol{w}^{\ell+1} - \boldsymbol{w}^\ell\| \leq t$.

Returns

- **x** (*ndarray*) – The approximate primal solution $\boldsymbol{x}$ after *num_iter* iterations.
- **y** (*ndarray*) – The approximate primal solution $\boldsymbol{y}$ after *num_iter* iterations.
- **w** (*ndarray*) – The approximate dual solution after *num_iter* iterations.

`tvopt.solvers.backtracking_gradient`(*problem*, *r=0.2*, *c=0.5*, *x_0=0*, *num_iter=100*, *tol=None*)

Gradient method with backtracking line search.

This function implements the gradient method

$$\boldsymbol{x}^{\ell+1} = \boldsymbol{x}^\ell - \alpha^\ell \nabla f(\boldsymbol{x}^\ell)$$

where α^ℓ is chosen via a backtracking line search. In particular, at each iteration we start with $\alpha^\ell = 1$ and, while

$$f(\boldsymbol{x}^\ell - \alpha^\ell \nabla f(\boldsymbol{x}^\ell)) > f(\boldsymbol{x}^\ell) - c\alpha^\ell \|\nabla f(\boldsymbol{x}^\ell)\|^2$$

we set $\alpha^\ell = r\alpha^\ell$ until a suitable step is found.

Note that the backtracking line search does not stop until a suitable step-size si found; this means that large *r* parameters may result in big computation times.

Parameters

- **problem** (*dict*) – Problem dictionary defining the cost f.
- **r** (*float, optional*) – The value by which a candidate step-size is multiplied if it does not satisfy the descent condition. *r* should be in $(0, 1)$.
- **c** (*float, optional*) – The parameter defining the descent condition that a candidate step must satisfy.
- **x_0** (*array_like, optional*) – The initial condition. This can be either an ndarray of suitable size, or a scalar. If it is a scalar then the same initial value is used for all components of $\boldsymbol{x}$.
- **num_iter** (*int, optional*) – The number of iterations to be performed.
- **tol** (*float, optional*) – If given, this argument specifies the tolerance t in the stopping condition $\|\boldsymbol{x}^{\ell+1} - \boldsymbol{x}^\ell\| \leq t$.

Returns

x – The approximate solution after *num_iter* iterations.

Return type

ndarray

`tvopt.solvers.dual_ascent`(*problem*, *penalty*, *w_0=0*, *num_iter=100*, *tol=None*)

Dual ascent.

This function implements the dual ascent to solve the constrained problem

$$\min_{\boldsymbol{x}} f(\boldsymbol{x}) \text{ s.t. } \boldsymbol{A}\boldsymbol{x} = \boldsymbol{c}.$$

The algorithm is characterized by the updates:

$$\boldsymbol{x}^\ell = \arg\min_{\boldsymbol{x}} \left\{ f(\boldsymbol{x}) - \langle \boldsymbol{w}^\ell, \boldsymbol{A}\boldsymbol{x} \rangle \right\} \quad (1.20)$$

$$\boldsymbol{w}^{\ell+1} = \boldsymbol{w}^\ell - \rho(\boldsymbol{A}\boldsymbol{x}^\ell - \boldsymbol{c}) \quad (1.21)$$

for a given penalty $\rho > 0$.

Parameters

- **problem** (*dict*) – Problem dictionary defining the cost f, and the constraints A and c.

- **penalty** (*float*) – The algorithm's penalty.
- **w_0** (*array_like, optional*) – The dual initial condition. This can be either an ndarray of suitable size, or a scalar. If it is a scalar then the same initial value is used for all components of $\boldsymbol{w}$.
- **num_iter** (*int, optional*) – The number of iterations to be performed.
- **tol** (*float, optional*) – If given, this argument specifies the tolerance t in the dual stopping condition $\|\boldsymbol{w}^{\ell+1} - \boldsymbol{w}^\ell\| \leq t$.

Returns

- **x** (*ndarray*) – The approximate primal solution after *num_iter* iterations.
- **w** (*ndarray*) – The approximate dual solution after *num_iter* iterations.

tvopt.solvers.**dual_fbs**(*problem*, *penalty*, *rel=1*, *w_0=0*, *num_iter=100*, *tol=None*)

Dual forward-backward splitting.

This function implements the dual FBS to solve the constrained problem

$$\min_{\boldsymbol{x},\boldsymbol{y}} \{f(\boldsymbol{x}) + g(\boldsymbol{y})\} \tag{1.22}$$

$$\text{s.t. } \boldsymbol{A}\boldsymbol{x} + \boldsymbol{B}\boldsymbol{y} = \boldsymbol{c} \tag{1.23}$$

The algorithm is characterized by the updates:

$$\boldsymbol{x}^\ell = \arg\min_{\boldsymbol{x}} \{f(\boldsymbol{x}) - \langle \boldsymbol{w}, \boldsymbol{A}\boldsymbol{x} \rangle\} \tag{1.24}$$

$$\boldsymbol{u}^\ell = \boldsymbol{w}^\ell - \rho(\boldsymbol{A}\boldsymbol{x}^\ell - \boldsymbol{c}) \tag{1.25}$$

$$\boldsymbol{y}^\ell = \arg\min_{\boldsymbol{y}} \left\{g(\boldsymbol{y}) - \langle \boldsymbol{u}^\ell, \boldsymbol{B}\boldsymbol{y} \rangle + \frac{\rho}{2}\|\boldsymbol{B}\boldsymbol{y}\|^2\right\} \tag{1.26}$$

$$\boldsymbol{w}^{\ell+1} = (1-\alpha)\boldsymbol{w}^\ell + \alpha(\boldsymbol{u}^\ell - \rho\boldsymbol{B}\boldsymbol{y}^\ell) \tag{1.27}$$

for a given penalty $\rho > 0$ and $\alpha \in (0, 1]$ is the relaxation constant.

Parameters

- **problem** (*dict*) – Problem dictionary defining the costs f and g, and the constraints A, B and c.
- **penalty** (*float*) – The algorithm's penalty.
- **rel** (*float, optional*) – The relaxation constant.
- **w_0** (*array_like, optional*) – The dual initial condition. This can be either an ndarray of suitable size, or a scalar. If it is a scalar then the same initial value is used for all components of $\boldsymbol{w}$.
- **num_iter** (*int, optional*) – The number of iterations to be performed.
- **tol** (*float, optional*) – If given, this argument specifies the tolerance t in the dual stopping condition $\|\boldsymbol{w}^{\ell+1} - \boldsymbol{w}^\ell\| \leq t$.

Returns

- **x** (*ndarray*) – The approximate primal solution $\boldsymbol{x}$ after *num_iter* iterations.
- **y** (*ndarray*) – The approximate primal solution $\boldsymbol{y}$ after *num_iter* iterations.
- **w** (*ndarray*) – The approximate dual solution after *num_iter* iterations.

`tvopt.solvers.fbs(problem, step, rel=1, x_0=0, num_iter=100, tol=None)`

Forward-backward splitting (FBS).

This function implements the forward-backward splitting (a.k.a. proximal gradient method) to solve the composite problem

$$\min_{\boldsymbol{x}} \{f(\boldsymbol{x}) + g(\boldsymbol{x})\}.$$

The algorithm is characterized by the update:

$$\boldsymbol{x}^{\ell+1} = (1-\alpha)\boldsymbol{x}^{\ell} + \alpha \operatorname{prox}_{\rho g}(\boldsymbol{x}^{\ell} - \rho \nabla f(\boldsymbol{x}^{\ell}))$$

where $\rho > 0$ is the step-size and $\alpha \in (0, 1]$ is the relaxation constant.

Parameters

- **problem** (*dict*) – Problem dictionary defining the costs f and g.
- **step** (*float*) – The algorithm's step-size.
- **rel** (*float, optional*) – The relaxation constant.
- **x_0** (*array_like, optional*) – The initial condition. This can be either an ndarray of suitable size, or a scalar. If it is a scalar then the same initial value is used for all components of $\boldsymbol{x}$.
- **num_iter** (*int, optional*) – The number of iterations to be performed.
- **tol** (*float, optional*) – If given, this argument specifies the tolerance t in the stopping condition $\|\boldsymbol{x}^{\ell+1} - \boldsymbol{x}^{\ell}\| \leq t$.

Returns

x – The approximate solution after *num_iter* iterations.

Return type

ndarray

`tvopt.solvers.gradient(problem, step, x_0=0, num_iter=100, tol=None)`

Gradient method.

This function implements the gradient method

$$\boldsymbol{x}^{\ell+1} = \boldsymbol{x}^{\ell} - \alpha \nabla f(\boldsymbol{x}^{\ell})$$

for a given step-size $\alpha > 0$.

Parameters

- **problem** (*dict*) – Problem dictionary defining the cost f.
- **step** (*float*) – The algorithm's step-size.
- **x_0** (*array_like, optional*) – The initial condition. This can be either an ndarray of suitable size, or a scalar. If it is a scalar then the same initial value is used for all components of $\boldsymbol{x}$.
- **num_iter** (*int, optional*) – The number of iterations to be performed.

- **tol** (*float, optional*) – If given, this argument specifies the tolerance t in the stopping condition $\|\boldsymbol{x}^{\ell+1} - \boldsymbol{x}^{\ell}\| \leq t$.

Returns
x – The approximate solution after *num_iter* iterations.

Return type
ndarray

`tvopt.solvers.mm`(*problem*, *penalty*, *w_0=0*, *num_iter=100*, *tol=None*)

Method of multipliers (MM).

This function implements the method of multipliers to solve the constrained problem

$$\min_{\boldsymbol{x}} f(\boldsymbol{x}) \text{ s.t. } \boldsymbol{A}\boldsymbol{x} = \boldsymbol{c}.$$

The algorithm is characterized by the updates:

$$\boldsymbol{x}^{\ell} = \arg\min_{\boldsymbol{x}} \left\{ f(\boldsymbol{x}) - \langle \boldsymbol{w}^{\ell}, \boldsymbol{A}\boldsymbol{x} \rangle + \frac{\rho}{2}\|\boldsymbol{A}\boldsymbol{x} - \boldsymbol{c}\|^2 \right\} \tag{1.28}$$

$$\boldsymbol{w}^{\ell+1} = \boldsymbol{w}^{\ell} - \rho(\boldsymbol{A}\boldsymbol{x}^{\ell} - \boldsymbol{c}) \tag{1.29}$$

for a given penalty $\rho > 0$.

Parameters

- **problem** (*dict*) – Problem dictionary defining the cost f, and the constraints A and c.
- **penalty** (*float*) – The algorithm's penalty.
- **w_0** (*array_like, optional*) – The dual initial condition. This can be either an ndarray of suitable size, or a scalar. If it is a scalar then the same initial value is used for all components of $\boldsymbol{w}$.
- **num_iter** (*int, optional*) – The number of iterations to be performed.
- **tol** (*float, optional*) – If given, this argument specifies the tolerance t in the dual stopping condition $\|\boldsymbol{w}^{\ell+1} - \boldsymbol{w}^{\ell}\| \leq t$.

Returns

- **x** (*ndarray*) – The approximate primal solution after *num_iter* iterations.
- **w** (*ndarray*) – The approximate dual solution after *num_iter* iterations.

`tvopt.solvers.newton`(*problem*, *r=0.2*, *c=0.5*, *x_0=0*, *num_iter=100*, *tol=None*)

Newton method with backtracking line search.

This function implements the Newton method

$$\boldsymbol{x}^{\ell+1} = \boldsymbol{x}^{\ell} - \alpha^{\ell}\nabla^2 f(\boldsymbol{x}^{\ell})^{-1}\nabla f(\boldsymbol{x}^{\ell})$$

where α^{ℓ} is chosen via a backtracking line search. In particular, at each iteration we start with $\alpha^{\ell} = 1$ and, while

$$f(\boldsymbol{x}^{\ell} - \alpha^{\ell}\nabla^2 f(\boldsymbol{x}^{\ell})^{-1}\nabla f(\boldsymbol{x}^{\ell})) > f(\boldsymbol{x}^{\ell}) - c\alpha^{\ell}\|\nabla f(\boldsymbol{x}^{\ell})\|^2$$

we set $\alpha^\ell = r\alpha^\ell$ until a suitable step is found.

Note that the backtracking line search does not stop until a suitable step-size si found; this means that large r parameters may result in big computation times.

Parameters

- **problem** (*dict*) – Problem dictionary defining the cost f.
- **r** (*float, optional*) – The value by which a candidate step-size is multiplied if it does not satisfy the descent condition. r should be in $(0, 1)$.
- **c** (*float, optional*) – The parameter defining the descent condition that a candidate step must satisfy.
- **x_0** (*array_like, optional*) – The initial condition. This can be either an ndarray of suitable size, or a scalar. If it is a scalar then the same initial value is used for all components of $\boldsymbol{x}$.
- **num_iter** (*int, optional*) – The number of iterations to be performed.
- **tol** (*float, optional*) – If given, this argument specifies the tolerance t in the stopping condition $\|\boldsymbol{x}^{\ell+1} - \boldsymbol{x}^\ell\| \leq t$.

Returns

x – The approximate solution after *num_iter* iterations.

Return type

ndarray

`tvopt.solvers.ppa`(*problem*, *penalty*, *x_0=0*, *num_iter=100*, *tol=None*)

Proximal point algorithm (PPA).

This function implements the proximal point algorithm

$$\boldsymbol{x}^{\ell+1} = \mathrm{prox}_{\rho f}(\boldsymbol{x}^\ell)$$

where $\rho > 0$ is the penalty parameter and we recall that

$$\mathrm{prox}_{\rho f}(\boldsymbol{x}) = \arg\min_{\boldsymbol{y}} \left\{ f(\boldsymbol{y}) + \frac{1}{2\rho}\|\boldsymbol{y} - \boldsymbol{x}\|^2 \right\}.$$

Parameters

- **problem** (*dict*) – Problem dictionary defining the cost f.
- **penalty** (*float*) – The penalty parameter for the proximal evaluation.
- **x_0** (*array_like, optional*) – The initial condition. This can be either an ndarray of suitable size, or a scalar. If it is a scalar then the same initial value is used for all components of $\boldsymbol{x}$.
- **num_iter** (*int, optional*) – The number of iterations to be performed.
- **tol** (*float, optional*) – If given, this argument specifies the tolerance t in the stopping condition $\|\boldsymbol{x}^{\ell+1} - \boldsymbol{x}^\ell\| \leq t$.

Returns

x – The approximate solution after *num_iter* iterations.

Return type
ndarray

`tvopt.solvers.prs(problem, penalty, rel=1, x_0=0, num_iter=100, tol=None)`

Peaceman-Rachford splitting (PRS).

This function implements the Peaceman-Rachford splitting to solve the composite problem

$$\min_{\boldsymbol{x}} \{ f(\boldsymbol{x}) + g(\boldsymbol{x}) \}.$$

The algorithm is characterized by the updates:

$$\boldsymbol{x}^\ell = \operatorname{prox}_{\rho f}(\boldsymbol{z}^\ell) \tag{1.30}$$
$$\boldsymbol{y}^\ell = \operatorname{prox}_{\rho g}(2\boldsymbol{x}^\ell - \boldsymbol{z}^\ell) \tag{1.31}$$
$$\boldsymbol{z}^{\ell+1} = \boldsymbol{z}^\ell + 2\alpha(\boldsymbol{y}^\ell - \boldsymbol{x}^\ell) \tag{1.32}$$

where $\rho > 0$ is the penalty and $\alpha \in (0, 1]$ is the relaxation constant.

Parameters

- **problem** (*dict*) – Problem dictionary defining the costs f and g.
- **penalty** (*float*) – The algorithm's penalty parameter.
- **rel** (*float, optional*) – The relaxation constant.
- **x_0** (*array_like, optional*) – The initial condition. This can be either an ndarray of suitable size, or a scalar. If it is a scalar then the same initial value is used for all components of $\boldsymbol{x}$.
- **num_iter** (*int, optional*) – The number of iterations to be performed.
- **tol** (*float, optional*) – If given, this argument specifies the tolerance t in the stopping condition $\|\boldsymbol{x}^{\ell+1} - \boldsymbol{x}^\ell\| \leq t$.

Returns
x – The approximate solution after *num_iter* iterations.

Return type
ndarray

`tvopt.solvers.stop(x, x_old, tol=None)`

Stopping condition.

This function checks the stopping condition

$$\|\boldsymbol{x}^{\ell+1} - \boldsymbol{x}^\ell\| \leq t$$

if *t* is specified.

Parameters

- **x** (*ndarray*) – The current iterate.
- **x_old** (*ndarray*) – The previous iterate.

- **tol** (*float, optional*) – The tolerance in the stopping condition.

Returns
True if *tol* is given and the stopping condition is verified, False otherwise.

Return type
bool

`tvopt.solvers.`**`subgradient`**(*problem*, *x_0=0*, *num_iter=100*, *tol=None*)

Sub-gradient method.

This function implements the sub-gradient method

$$\boldsymbol{x}^{\ell+1} = \boldsymbol{x}^{\ell} - \alpha^{\ell} \tilde{\nabla} f(\boldsymbol{x}^{\ell})$$

where $\tilde{\nabla} f(\boldsymbol{x}^{\ell}) \in \partial f(\boldsymbol{x}^{\ell})$ is a sub-differential and $\alpha^{\ell} = 1/(\ell + 1)$.

Parameters

- **problem** (*dict*) – Problem dictionary defining the cost f.
- **x_0** (*array_like, optional*) – The initial condition. This can be either an ndarray of suitable size, or a scalar. If it is a scalar then the same initial value is used for all components of $\boldsymbol{x}$.
- **num_iter** (*int, optional*) – The number of iterations to be performed.
- **tol** (*float, optional*) – If given, this argument specifies the tolerance t in the stopping condition $\|\boldsymbol{x}^{\ell+1} - \boldsymbol{x}^{\ell}\| \leq t$.

Returns
x – The approximate solution after *num_iter* iterations.

Return type
ndarray

1.8 tvopt.utils module

Utility tools.

`tvopt.utils.`**`bisection_method`**(*f*, *a*, *b*, *tol=1e-05*)

Minimize using the bisection method.

This function minimizes a function *f* using the bisection method, stopping when $a - b \leq t$ for some threshold t.

Parameters

- **f** – The scalar function to be minimized.
- **a** (*float*) – The lower bound of the initial interval.
- **b** (*float*) – the upper bound of the initial interval.
- **tol** (*float, optional*) – The stopping condition, defaults to 1e-5.

Returns
x – The approximate minimizer.

Return type
float

`tvopt.utils.dist(s, r, ord=2)`

Distance of a signal from a reference.

This function computes the distance of a signal *s* from a reference *r*. The reference can be either constant or a signal itself. Different norm orders can be used, that can be specified using the *numpy.linalg.norm* argument *ord*.

Parameters

- **s** (*array_like*) – The signal, with the last dimension indexing time.
- **r** (*array_like*) – The reference, either a single array or a signal with the last dimension indexing time.
- **ord** (*optional*) – Norm order, see *numpy.linalg.norm*.

Raises

ValueError – For incompatible dimensions of signal and reference.

Returns

The distance of the signal from the reference as an array with length equal to the last dimension of *s*.

Return type

ndarray

`tvopt.utils.fpr(s, ord=2)`

Fixed point residual.

This function computes the fixed point residual of a signal *s*, that is

$$\{\|s^\ell - s^{\ell-1}\|_i\}_{\ell \in \mathbb{N}}.$$

Different norm orders can be used, that can be specified using the *numpy.linalg.norm* argument *ord*.

Parameters

- **s** (*array_like*) – The signal, with the last dimension indexing time.
- **ord** (*optional*) – Norm order, see *numpy.linalg.norm*.

Returns

The fixed point residual.

Return type

ndarray

`tvopt.utils.initialize_trajectory(x_0, shape, num_iter)`

`tvopt.utils.is_scalar(c)`

Check if scalar.

`tvopt.utils.is_square(mat)`

Check if the matrix is 2-D and square.

Parameters

mat (*ndarray*) – The given matrix.

Returns

True if the matrix is 2-D and square, False otherwise.

Return type

bool

`tvopt.utils.`**`is_stochastic`**`(`*mat*, *row=True*, *col=True*`)`

Verify if a given matrix is row, column or doubly stochastic.

Parameters

- **`mat`** (*`ndarray`*) – The given matrix.
- **`row`** (*`bool, optional`*) – Check for row stochasticity, default True.
- **`col`** (*`bool, optional`*) – Check for column stochasticity, default True.

Returns

True if the matrix is stochastic (row, column or doubly, as specified by the arguments).

Return type

bool

Raises

`ValueError` – If neither *row* nor *col* are True.

`tvopt.utils.`**`norm`**`(`*x*`)`

Compute the norm of the given vector.

Parameters

`x` (*`array_like`*) – The vector array.

Returns

The square norm.

Return type

ndarray

See also:

`square_norm`

Square norm

Notes

The function reshapes *x* to a column vector, so it does not correctly handle n-dimensional arrays. For n-dim arrays use *numpy.linalg.norm*.

`tvopt.utils.`**`normalize`**`(`*x*`)`

Normalize a vector to unit vector.

Parameters

`x` (*`array_like`*) – The vector array.

Returns

The normalized vector.

Return type

ndarray

Notes

The function reshapes x to a column vector, so it does not correctly handle n-dimensional arrays. For n-dim arrays use *numpy.linalg.norm*.

`tvopt.utils.orthonormal_matrix(dim)`

Generate a random orthonormal matrix.

This function generates uniformly distributed random orthonormal matrices using Householder reflections (see Section 7 of this paper).

Parameters
dim (*int*) – Size of the matrix.

Returns
orth_mat – The random orthonormal matrix.

Return type
ndarray

Raises
ValueError – For invalid *dim*.

`tvopt.utils.positive_semidefinite_matrix(dim, max_eig=None, min_eig=None)`

Generate a random positive semi-definite matrix.

The matrix is generated as

$$M = O \mathrm{diag}\{\lambda_i\} O^\top$$

where O is a random orthonormal matrix and λ_i are random eigenvalues uniformly drawn between *min_eig* and *max_eig*. If *dim* is larger than or equal to two, *min_eig* and *max_eig* are included in the eigenvalues list.

Parameters

- **dim** (*int*) – Size of the matrix.
- **eigs** (*array-like, optional*) – The list of eigenvalues for the matrix; if None, the eigenvalues are uniformly drawn from $[10^{-2}, 10^2]$.

Returns
The random positive semi-definite matrix.

Return type
ndarray

Raises
ValueError. –

See also:

random_matrix
Random matrix generator.

`tvopt.utils.print_progress(i, num_iter, bar_length=80, decimals=2)`

Print the progresso to command line.

Parameters

- **i** (*int*) – Current iteration.

- **num_iter** (*int*) – Total number of iterations.
- **bar_length** (*int, optional*) – Length of progress bar.
- **decimals** (*int, optional*) – Decimal places of the progress percent.

Notes

Adapted from here.

tvopt.utils.**random_matrix**(*eigs*)

Generate a random matrix.

The matrix is generated as

$$M = O\text{diag}\{\lambda_i\}O^\top$$

where O is a random orthonormal matrix and λ_i are the specified eigenvalues.

Parameters
eigs (*array-like*) – The list of eigenvalues for the matrix.

Returns
The random positive semi-definite matrix.

Return type
ndarray

See also:

orthonormal_matrix
Orthonormal matrix generator.

tvopt.utils.**regret**(*f*, *s*, *r=None*)

Cost over time or regret.

This function computes the cost evaluated using f incurred by an approximate minimizer s

$$\{\frac{1}{\ell}\sum_{j=1}^{\ell} f(s^j)\}_{\ell\in\mathbb{N}}$$

or, if a reference r is specified, then the function computes the regret

$$\{\frac{1}{\ell}\sum_{j=1}^{\ell} f(s^j) - f(r^j)\}_{\ell\in\mathbb{N}}$$

where r is either a constant array or a signal.

Parameters

- **f** (costs.Cost) – The cost to evaluate in the signal.
- **s** (*array_like*) – The sequence of approximate minimizers.

- **r** (*array_like, optional*) – The reference, either a single array or a signal with the last dimension indexing time.

Returns
The sequence of cost evaluations or regret.

Return type
ndarray

`tvopt.utils.`**`soft_thresholding`**(*x*, *penalty*)

Soft-thresholding.

The function computes the element-wise soft-trhesholding defined as

$$\text{sign}(x) \max\{|x| - \rho, 0\}$$

where ρ is a positive penalty parameter.

Parameters

- **x** (*array_like*) – Where to evaluate the soft-thresholding.
- **penalty** (*float*) – The positive penalty parameter ρ.

Returns
The soft-thresolding of *x*.

Return type
ndarray

`tvopt.utils.`**`solve`**(*a*, *b*)

`tvopt.utils.`**`square_norm`**(*x*)

Compute the square norm of the given vector.

Parameters
x (*array_like*) – The vector array.

Returns
The square norm.

Return type
ndarray

Notes

The function reshapes *x* to a column vector, so it does not correctly handle n-dimensional arrays. For n-dim arrays use *numpy.linalg.norm*.

`tvopt.utils.`**`uniform_quantizer`**(*x*, *step*, *thresholds=None*)

Function to perform uniform quantization.

The function applies the uniform quantization

$$q(x) = \Delta \, \text{floor} \left(\frac{x}{\Delta} + \frac{1}{2} \right)$$

where Δ is the given step. Moreover, a saturation to upper and lower thresholds is peformed if given as argument.

Parameters

- **x** (*ndarray*) – The array to be quantized.
- **step** (*float*) – The step of the quantizer.
- **thresholds** (*list, optional*) – The upper and lower saturation thresholds.

Returns
The quantized array.

Return type
ndarray

1.9 Module contents

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

t

INDEX

G

H

I

L

M

N

O

P

Q

R

S

T

U

W