NC-OPT

Release 0.2.1

Weiwei Kong

Jan 07, 2023

CONTENTS

CHAPTER ONE

COPYRIGHT AND LICENSE

CHAPTER

TWO

INTRODUCTION

NC-OPT is an open source software package for **N**onconvex **C**omposite **Opt**imization in MATLAB. Its main purpose is to: (a) provide a general API for creating models based on a first-order oracle framework; and (b) leverage the fast matrix subroutines in MATLAB to solve these models. The interface of NC-OPT is loosely based on the well-known Gurobi API.

Currently, the following problems classes are supported:

- Unconstrained Composite Optimization
- Linearly Set Constrained Composite Optimization
- Nonconvex-Concave Min-Max Optimization
- Spectral Composite Optimization
- Convex Cone Constrained Composite Optimization

Instances of the above classes include *semidefinite programming*, *convex programming*, *cone programming*, *linear and quadratic programmin*, and *nonconvex programming*.

The components of NC-OPT can be split into the following categories.

`oracles`
: Classes that abstract the idea of a first-order oracle at a point. It contains one or more of the following oracles: *function value*, *function gradient*, and *proximal oracle*.

`solvers`
: A collection of composite optimzation solvers that solve the problem associated with a particular first-order oracle. Some examples include: *the composite gradient method*, *the accelerated composite gradient method*, and *the accelerated inexact proximal point method*.

`frameworks`
: A collection of constrained composite optimization frameworks that use a solver to solve a constrained composite optimization model. Some examples include: *the quadratic penalty framework*, *the augmented Lagrangian framework*, and *the dampened augmented Lagrangian framework*.

`models`
: A class that abstracts the idea of a composite optimization model. It contains properties that describe various aspects of the model, including: *objective function*, *constraints*, *solver*, *framework*, *tolerances*, and *logging*.

CHAPTER THREE

INSTALLATION

Note: You will need a valid MATLAB installation and license in order to use NC-OPT's functions, classes, solvers, and frameworks.

To install NC-OPT, simply clone the latest version from GitHub by running the following command in your terminal:

```
git clone https://github.com/wwkong/nc_opt.git
```

After downloading the files, start up MATLAB and run the `init.m` in the root directory, which will add the paths to all of the functions in the library. You can check if everything is properly configured by running one of the examples in `example` such as:

```
./nc_opt/examples/unconstrained/basic_convex_qp.m
```

CHAPTER

FOUR

OVERVIEW OF THE API

This section presents a high level overview of the NC-OPT MATLAB interface. All sections following this one present a comprehensive description of the key classes, solvers, and frameworks.

4.1 Creating a Model

NC-OPT allows you to model problems of the following form:

$$\begin{array}{ll} \text{minimize} & f_s(x) + f_n(x) \\ \text{subject to} & g(x) \in S. \end{array}$$

Models are generated by using constructors for the `CompModel` or `ConstrCompModel` classes, each containing properties that specify its various components such as solvers, oracles, and tolerances. By default, the empty constructor sets

$$f_n \equiv 0, \quad g \equiv 0, \quad S = \{0\}.$$

While not enforced, it is assumed that the user will configure the model so that the set S and the functions f_s, f_n, and g satisfy the following minimal assumptions:

- The set S is either empty or nonempty and convex.
- The function f_s is continuously differentiable and its gradient is L-Lipschitz continuous.
- The function f_n is proper, closed, and convex.
- The function g is either linear or (cone) convex.
- Either $\inf_u [f_s(u) + f_n(u)] > -\infty$ or the domain of f_n is compact.

After constructing the model, one or more of the following components will then need to be specified, depending on the type of model created.

Initial Point and L

For `CompModel` or `ConstrCompModel` objects, the user needs to provide: (a) an initial point x_0 in the domain of `f_n`; and (b) the Lipschitz constant L of the gradient of f_s. These can be specified by setting the `x0` and `L` properties of the models.

Oracle

For `CompModel` or `ConstrCompModel` objects, the user needs to provide a subset of the functions above or a first-order oracle, represented by an `Oracle` object, to the model. The first-order oracle has a method `eval()` which, when evaluated at point, returns information about the objective function at that point.

Solver

For CompModel or ConstrCompModel objects, the user needs to provide a composite optimization solver to the model. This solver takes an Oracle object and a struct object of hyperparameters as input, and outputs a solution associated the problem underlying the oracle.

Framework

For ConstrCompModel objects, the user needs to provide a constrained optimization framework to the model. This framework takes a solver function, Oracle object, and a struct of hyperparameters as input, and outputs a solution associated the problem underlying the oracle and hyperparameters.

4.2 Solving a Model

When a model has been fully constructed, it can solved by invoking the optimize() method. This method, in particular, applies either the underlying solver or framework to produce an approximate stationary point of the associated optimization problem. For unconstrained problems, this point is a pair (x, v) satisfying

$$\begin{gathered} v \in \nabla f_s(x) + \partial f_n(x), \\ x \in \operatorname{dom} f_n, \quad \|v\| \le \rho, \end{gathered}$$

for a given tolerance $\rho \in \mathbb{R}_{++}$. For constrained problems, this point is a triple (x, v, w) satisfying

$$\begin{gathered} v \in \nabla f_s(x) + \partial f_n(x) + \nabla g(x) y, \\ x \in \operatorname{dom} f_n, \quad g(x) + w \in S, \quad \|v\| \le \rho, \quad \|w\| \le \eta, \end{gathered}$$

for a given tolerance pair $(\rho, \eta) \in \mathbb{R}^2_{++}$. By default, the tolerances are set to $\rho = 10^{-6}$ and $\eta = 10^{-6}$. Messages about the model's configuration and final status will be output to the MATLAB command line.

4.3 A Simple Example

Below is a simple example for solving the unconstrained convex optimization problem

$$\underset{x \in \mathbb{R}}{\text{minimize}} \quad \frac{1}{2}x^2 - x + \frac{1}{2}.$$

```
% Create the Model object and specify the solver.
cvx_qp = CompModel;
cvx_qp.solver = @AIPP;

% Define the function and its gradient.
cvx_qp.f_s = @(x) (1 / 2) * (x - 1) ^ 2;
cvx_qp.grad_f_s = @(x) x - 1;

% Set the Lipschitz constant of the gradient and the starting point x0.
cvx_qp.L = 10;
cvx_qp.x0 = 10;

% Solve the problem.
cvx_qp.optimize;
```

The complete code for this example can be found in ./nc_opt/examples/unconstrained/basic_convex_qp.m.

CHAPTER

FIVE

ORACLES

This section documents the constructors, properties, and methods of the `Oracle` and `SpectralOracle` classes. Hidden and inherited properties are not shown here for simplicity.

Warning: The oracle classes in this section inherit from the copyable `handle` class meaning that all assignments made on these objects will only hold references to the underlying data. **In other words, more than one variable can refer to the same underlying object!** For example, consider the following code:

```
a = Oracle;
a.f_s = @() 1;
b = a;
b.f_s = @() 10;
disp(a.f_s())
```

The above code will return a value of `10` to the terminal, instead of `1` because the underlying data of `a` was modified through `b`.

To make an explicit copy of an oracle object, **without creating a reference to the original object**, one must invoke the `copy()` function before the assignment. For example, consider the following code, which is a modification of the previous example:

```
a = Oracle;
a.f_s = @() 1;
b = copy(a);
b.f_s = @() 10;
disp(a.f_s())
```

The above code, this time, will return a value `1` to the terminal as `a` and `b` are now refering to different objects.

class `src.oracles.`**`Oracle`**(*varargin*)

Bases: `matlab.mixin.Copyable`, `dynamicprops`

An abstact oracle class for unconstrained composite optimization.

`f_s`

A zero argument function that, when evaluated, outputs $f_s(x)$. Defaults to `@() 0`.

Type
function handle

`grad_f_s`

A zero argument function that, when evaluated, outputs $\nabla f_s(x)$. Defaults to

Type
function handle

``@

Type
) zeros(size(x)

f_n

A zero argument function that, when evaluated, outputs $f_n(x)$. Defaults to `@() 0`.

Type
function handle

prox_f_n

A one argument function that, when evaluated at λ, outputs

$$\mathrm{prox}_{\lambda f_n}(x) := \mathrm{argmin}_u \left\{ \lambda f_n(u) + \frac{1}{2}\|u - x\|^2 \right\}.$$

Defaults to `@(lam) x`.

Type
function handle

f_s_at_prox_f_n

A one argument function that, when evaluated at λ, returns the value of f_s at the point given by the function `prox_f_n` at λ. This is generally used to speed up certain algorithms (see src/solvers/ACG.m). Defaults to `None`.

Type
function handle

prod_fn

A two argument function that, when evaluated at $\{a, b\}$, outputs the inner product $\langle a, b \rangle$. Defaults to the Euclidean inner product, i.e. `@(a,b) sum(dot(a, b))`.

Type
function handle

norm_fn

A one function that, when evaluated at a point a, outputs $\|a\|$. Defaults to the Frobenius norm, i.e. `norm(a, 'fro')`.

Type
function handle

Oracle(*varargin*)

The constructor for the Oracle class. It has three ways to initialize:

- `Oracle()` creates an oracle that represents the zero function, i.e. $f_s = f_n \equiv 0$.
- `Oracle(f_s, f_n, grad_f_s, prox_f_n)` creates an Oracle object with the properties `f_s`, `f_n`, `grad_f_s`, and `prox_f_n` filled by the corresponding inputs.
- `Oracle(eval_fn)` creates an Oracle object which, when evaluated at a point x, updates the properties `f_s`, `f_n`, `grad_f_s`, and `prox_f_n` as follows:

```
f_s = eval_fn(x).f_s;
f_n = eval_fn(x).f_n;
grad_f_s = eval_fn(x).grad_f_s;
prox_f_n = eval_fn(x).prox_f_n;
```

Similar updates are made for `f_s_at_prox_f_n`, `f_n_at_prox_f_n`, and `grad_f_s_at_prox_f_n`, if these are generated by `eval_fn()`.

`eval`(*x*)

Evaluates the oracle at a point `x` and updates the properties `f_s`, `f_n`, `grad_f_s`, and `prox_f_n` to be at this point.

`add_smooth_oracle`(*other_obj*)

Modified the suboracles by adding the smooth suboracles of an input oracle to the smooth suboracles of the current oracle. That is, the properties `f_s`, `f_n`, `grad_f_s`, and `prox_f_n` are updated as follows:

```
f_s(x) = f_s(x) + input_oracle.f_s(x);
grad_f_s(x) = grad_f_s(x) + input_oracle.grad_f_s(x);
```

`scale`(*alpha*)

Modified the suboracles by multiplying by a positive constant `alpha`. That is, the properties `f_s`, `f_n`, `grad_f_s`, and `prox_f_n` are updated as follows:

```
f_s(x) = alpha * f_s(x);
f_n(x) = alpha * f_n(x);
grad_f_s(x) = alpha * grad_f_s(x);
prox_f_n(x, lambda) = prox_f_n(x, alpha * lambda);
```

`add_prox`(*x_hat*)

Modified the suboracles by adding a prox term at a point `x_hat`. That is, the properties `f_s`, `f_n`, `grad_f_s`, and `prox_f_n` are updated as follows:

```
f_s(x) = f_s(x) + (1 / 2) * norm_fn(x - x_hat) ^ 2;
grad_f_s(x) = grad_f_s(x) + (x - x_hat);
```

`proxify`(*alpha*, *x_hat*)

Modify the suboracles by multiplying by a positive constant `alpha` and then adding a prox term at a point `x_hat`. That is, the properties `f_s`, `f_n`, `grad_f_s`, and `prox_f_n` are updated as follows:

```
f_s(x) = alpha * f_s(x) + (1 / 2) * norm_fn(x - x_hat) ^ 2;
f_n(x) = alpha * f_n(x);
grad_f_s(x) = alpha * grad_f_s(x) + (x - x_hat);
prox_f_n(x, lambda) = prox_f_n(x, alpha * lambda);
```

`decompose`()

A zero argument method that, when evaluated, returns variants of the properties `f_s`, `f_n`, `grad_f_s`, and `prox_f_n`, where an additional argument `x` is added to the beginning of the of the list of function inputs. For example, the value `f = f_s(x)`, computed from the output function, is equivalent to:

```
my_oracle.eval(x);
f = my_oracle.f_s();
```

where my_oracle refers to the oracle object that is calling `decompose()`.

class `src.oracles.`**`SpectralOracle`**(*varargin*)

Bases: *`src.oracles.Oracle`*

An abstact spectral oracle class for unconstrained spectral composite optimization.

Note: This oracle is specialized for solving nonconvex composite optimization problems in which $f_s = f_{1,s} + f_{2,s}^{\mathcal{V}} \circ \sigma$ and $f_n = f_n^{\mathcal{V}} \circ \sigma$ for absolutely symmetric functions $f_{2,s}^{\mathcal{V}}$ and $f_n^{\mathcal{V}}$. Here, $\sigma()$ denotes operator that maps matrices to some form of type of spectral vector, e.g. a singular value vector.

Unlike the the usual `Oracle` object, this one has an additional method named `spectral_eval()` that takes two arguments, `{X, x}`, and updates (f_s, f_n) with `X` and $(f_{2,s}^{\mathcal{V}}, f_n^{\mathcal{V}})$ with `x` separately. This can be helpful when the spectral vector $\sigma(X)$ has already been computed. If `x=[]`, the expected behavior is that `x` is computed from $\sigma()$.

See also:

Kong, W., & Monteiro, R. D. (2020). Accelerated Inexact Composite Gradient Methods for Nonconvex Spectral Optimization Problems. *arXiv preprint arXiv:2007.11772*.

`spectral_grad_f2_s`

A zero argument function that, when evaluated, outputs $\nabla f_{2,s}^{\mathcal{V}}(x)$. Defaults to `@() zeros(size(x))`.

Type
function handle

`spectral_prox_f_n`

A one argument function that, when evaluated at λ, outputs

$$\text{prox}_{\lambda f_n^{\mathcal{V}}}(x) := \text{argmin}_u \left\{ \lambda f_n^{\mathcal{V}}(u) + \frac{1}{2}\|u - x\|^2 \right\}.$$

Defaults to `@(lam) x`.

Type
function handle

`f1_s`

A zero argument function that, when evaluated, outputs $f_{1,s}(X)$. Defaults to `@() 0`.

Type
function handle

`f2_s`

A zero argument function that, when evaluated, outputs $f_{2,s}(X)$. Defaults to `@() 0`.

Type
function handle

`grad_f1_s`

A zero argument function that, when evaluated, outputs $\nabla f_{1,s}(X)$. Defaults to `@() zeros(size(X))`.

Type
function handle

`grad_f2_s`

A zero argument function that, when evaluated, outputs $\nabla f_{2,s}(X)$. Defaults to `@() zeros(size(X))`.

Type
function handle

`decomp_fn`

A one argument function that, when evaluated at X, outputs three arguments `{P, D, Q}` where `D` is a diagonal matrix, `P` and `Q` are orthogonal matrices, and $X = PDQ^*$. Defaults to `@(X) svd(X, 'econ')`.

Type
function handle

SpectralOracle(*varargin*)

The constructor for the SpectralOracle class. It has three ways to initialize:

- `Oracle()` creates an oracle that represents the zero function, i.e. $f_s = f_n \equiv 0$.
- `Oracle(f_s, f_n, grad_f_s, prox_f_n)` creates an Oracle object with the properties `f_s`, `f_n`, `grad_f_s`, and `prox_f_n` filled by the corresponding inputs.
- `Oracle(spectral_eval_fn)` creates an Oracle object which, when spectrally evaluated at a pair `{X, x}`, updates the properties related to `f_s`, `f_n`, `grad_f_s`, and `prox_f_n` as follows:

```
% Update the suboracles
f_s = spectral_eval_fn(X, x).f_s;
f_n = spectral_eval_fn(X, x).f_n;
grad_f_s = spectral_eval_fn(X, x).grad_f_s;
prox_f_n = spectral_eval_fn(X, x).prox_f_n;
spectral_grad_f2_s = spectral_eval_fn(X, x).spectral_grad_f2_s;
spectral_prox_f_n = spectral_eval_fn(X, x).spectral_grad_f2_s;
```

  Similar updates are made for `f_s_at_prox_f_n`, `f_n_at_prox_f_n`, and `grad_f_s_at_prox_f_n`, if these are generated by the underlying evaluator.

eval(*X*)

An alias for `obj.spectral_eval(X, [])`.

spectral_eval(*X, x*)

Evaluates the spectral oracle at a pair `{X, x}` and updates all of the relevant properties at this pair.

scale(*alpha*)

Modified the suboracles by multiplying by a positive constant `alpha`. That is, the properties `f_s`, `f_n`, `grad_f_s`, and `prox_f_n` are updated as follows:

```
f_s(x) = alpha * f_s(x);
f_n(x) = alpha * f_n(x);
grad_f_s(x) = alpha * grad_f_s(x);
prox_f_n(x, lambda) = prox_f_n(x, alpha * lambda);
```

add_prox(*x_hat*)

Modified the suboracles by adding a prox term at a point `x_hat`. That is, the properties `f_s`, `f_n`, `grad_f_s`, and `prox_f_n` are updated as follows:

```
f_s(x) = f_s(x) + (1 / 2) * norm_fn(x - x_hat) ^ 2;
grad_f_s(x) = grad_f_s(x) + (x - x_hat);
```

vector_linear_proxify(*alpha, x_hat*)

Denoting $\alpha = \text{alpha}$ and $\hat{x} = \text{x\_hat}$, calling this method changes the underlying function of the oracle as follows:

$$\begin{aligned} f_s(x) &\leftarrow \alpha f_{2,s}^{\mathcal{V}}(x) - \langle x, \hat{x} \rangle + \frac{1}{2}\|x\|^2, \\ f_n(x) &\leftarrow \alpha f_n^{\mathcal{V}}(x). \end{aligned}$$

This method is primarily used to convert the oracle from one that acts on a matrix space to one that acts on a vector space. It also makes sure that the `eval()` method is properly adjusted to take in vector inputs rather than matrix inputs.

redistribute_curvature(*alpha*)

Denoting $\alpha = \mathrm{alpha}$, calling this method changes the underlying function of the oracle as follows:

$$\begin{aligned} f_{1,s}(X) &\leftarrow f_{1,s}(X) - \frac{\alpha}{2}\|X\|_F^2, \\ f_{2,s}(X) &\leftarrow f_{2,s}(X) + \frac{\alpha}{2}\|X\|_F^2, \\ f_{2,s}^{\mathcal{V}}(x) &\leftarrow f_{2,s}^{\mathcal{V}}(x) + \frac{\alpha}{2}\|x\|^2. \end{aligned}$$

decompose()

A zero argument method that, when evaluated, returns variants of the properties `f_s`, `f_n`, `grad_f_s`, and `prox_f_n`, where an additional argument `x` is added to the beginning of the of the list of function inputs. For example, the value `f = f_s(x)`, computed from the output function, is equivalent to:

```
my_oracle.eval(x);
f = my_oracle.f_s();
```

where my_oracle refers to the oracle object that is calling `decompose()`.

CHAPTER

SIX

SOLVERS

This section documents the solvers that available for solving unconstrained composite optimization problems. For simplicity, we only list the parameter inputs that affect the behavior of the method for a fixed optimization problem. All other inputs should be passed from the model object that is calling the solver.

Note: All solvers have a parameter `i_logging` that, when set to `true`, will log the function value, (inner) iteration number, and time at each (outer) iteration. These values are then added to the `history` struct that is output by each solver.

6.1 Auxiliary Solvers

These solvers are used as (possible) subroutines for the other solvers of this section.

`src.solvers.`**`ACG`**(*oracle*, *params*)

An accelerated composite gradient (ACG) algorithm for use inside of the accelerated inexact proximal point (AIPP) method.

See also:

src/solvers/AIPP.m

Note: Its iterates are generated according the paper:

Monteiro, R. D., Ortiz, C., & Svaiter, B. F. (2016). An adaptive accelerated first-order method for convex optimization. *Computational Optimization and Applications*, 64(1), 31-73.

Parameters

- **`oracle`** (`Oracle`) – The oracle underlying the optimization problem.
- **`params`** (`struct`) – Contains instructions on how to call the algorithm. This should ideally be customized from by caller of this
- **`algorithm`** –
- **`user.`** (`rather than the`) –

Returns

A pair of structs containing model and history related outputs of the solved problem associated with the oracle and input parameters.

6.2 NCO Solvers

These solvers are used for nonconvex composite optimization problems.

`src.solvers.`**`AC_ACG`**`(`*oracle*, *params*`)`

The average curvature accelerated composite gradient (AC-ACG) method.

Note: Based on the paper:

Liang, J., & Monteiro, R. D. (2019). An average curvature accelerated composite gradient method for nonconvex smooth composite optimization problems. *arXiv preprint arXiv:1909.04248.*

Parameters

- **`oracle`** (`Oracle`) – The oracle underlying the optimization problem.
- **`params.alpha`** (`double`) – Controls the rate at which the upper curvature is updated (see α from the original paper). Defaults to `0.5`.
- **`params.gamma`** (`double`) – Controls the rate at which the upper curvature is updated (see γ from the original paper). Defaults to `0.01`.

Returns

A pair of structs containing model and history related outputs of the solved problem associated with the oracle and input parameters.

`src.solvers.`**`ADAP_FISTA`**`(`*oracle*, *params*`)`

The adaptive nonconvex fast iterative soft theresholding (ADAP-NC-FISTA) method.

See also:

src/solvers/NC_FISTA.m

Note: Based on the paper (see the ADAP-NC-FISTA method):

Liang, J., Monteiro, R. D., & Sim, C. K. (2019). A FISTA-type accelerated gradient algorithm for solving smooth nonconvex composite optimization problems. *arXiv preprint arXiv:1905.07010.*

Parameters

- **`oracle`** (`Oracle`) – The oracle underlying the optimization problem.
- **`params.theta`** (`double`) – Controls how the stepsize is updated (see θ from the original paper). Defaults to `1.25`.

Returns

A pair of structs containing model and history related outputs of the solved problem associated with the oracle and input parameters.

`src.solvers.`**`AG`**`(`*oracle*, *params*`)`

The accelerated gradient (AG) method.

See also:

src/solvers/UPFAG.m

Note: A variant of Algorithm 2 (with the upper curvature M replacing the Lipschitz constant L_f) from the paper:

Ghadimi, S., & Lan, G. (2016). Accelerated gradient methods for nonconvex nonlinear and stochastic programming. *Mathematical Programming, 156*(1-2), 59-99.

Parameters

- **oracle** (`Oracle`) – The oracle underlying the optimization problem.
- **params** (`struct`) – Contains instructions on how to call the algorithm.

Returns

A pair of structs containing model and history related outputs of the solved problem associated with the oracle and input parameters.

`src.solvers.`**`AIPP`**(*oracle*, *params*)

Variants of the accelerated inexact proximal point (AIPP) method

See also:

src/solvers/ACG.m

Note: Based on the papers:

[1] Kong, W., Melo, J. G., & Monteiro, R. D. (2019). Complexity of a quadratic penalty accelerated inexact proximal point method for solving linearly constrained nonconvex composite programs. *SIAM Journal on Optimization, 29*(4), 2566-2593.

[2] Kong, W., Melo, J. G., & Monteiro, R. D. (2020). An efficient adaptive accelerated inexact proximal point method for solving linearly constrained nonconvex composite problems. *Computational Optimization and Applications, 76*(2), 305-346.

[3] Kong, W., & Monteiro, R. D. (2019). An accelerated inexact proximal point method for solving nonconvex-concave min-max problems. *arXiv preprint arXiv:1905.13433*.

In particular, τ is updated using the rule in [3].

Parameters

- **oracle** (`Oracle`) – The oracle underlying the optimization problem.
- **params.aipp_type** (`character vector`) – Specifies which AIPP variant to use. More specifically, ‘aipp’ is the AIPP method from [1], while ‘aipp_c’, ‘aipp_v1’, and ‘aipp_v2’ are the R-AIPPc, R-AIPPv1, and R-AIPPv2 methods from [2]. Defaults to `'aipp_v2'`.
- **params.sigma** (`double`) – Determines the accuracy of the inner ACG call (see σ from [1]). Defaults to `0.3`.
- **params.theta** (`double`) – Determines the accuracy of the inner R-ACG call (see θ from [2]). Defaults to `4`.
- **params.acg_steptype** (`character vector`) – Is either “variable” or “constant”. If it is “variable”, then the ACG call employs a line search subroutine to look for the appropriate upper curvature, with a starting estimate of $L_0 = \lambda(M/100) + 1$. If “constant”, then no subroutine is employed and the upper curvature remains fixed at $L_0 = \lambda M + 1$. Defaults to `'variable'`.

- **params.acg_ls_multiplier** (double) – Determines how quickly the line search subroutine of the ACG call (multiplicatively) increases its estimate. Defaults to `1.25`.
- **params.lambda** (double) – The initial stepsize (see λ from [2]). Defaults to `1 / m`.
- **params.mu_fn** (function handle) – Determines the strong convexity parameter μ given to the ACG call as a function of lambda, the stepsize. Defaults to `@(lambda) 1`.
- **params.tau_mult** (double) – An auxiliary parameter constant used to determine the first value of τ. Defaults to `10`.
- **params.tau** (double) – A parameter constant that determines the accuracy of the inner ACG call (see τ from [2, 3]). Defaults to `tau_mult * (lambda M + 1)` where `lambda` is the initial stepsize and `tau_mult` is the constant in `params.tau_mult`.

Returns
A pair of structs containing model and history related outputs of the solved problem associated with the oracle and input parameters.

`src.solvers.`**`ECG`**(*oracle*, *params*)

The well-known exact composite gradient (ECG) method with constant stepsize.

Note: For reference, see the paper:

Nesterov, Y. (2013). Gradient methods for minimizing composite functions. *Mathematical Programming, 140*(1), 125-161.

Parameters

- **oracle** (Oracle) – The oracle underlying the optimization problem.
- **params** (struct) – Contains instructions on how to call the algorithm.

Returns
A pair of structs containing model and history related outputs of the solved problem associated with the oracle and input parameters.

`src.solvers.`**`NC_FISTA`**(*oracle*, *params*)

The nonconvex fast iterative soft theresholding (ADAP-NC-FISTA) method.

See Also:

src/solvers/ADAP_FISTA.m

Note: Based on the paper (see the NC-FISTA method):

Liang, J., Monteiro, R. D., & Sim, C. K. (2019). A FISTA-type accelerated gradient algorithm for solving smooth nonconvex composite optimization problems. *arXiv preprint arXiv:1905.07010*.

Parameters

- **oracle** (Oracle) – The oracle underlying the optimization problem.
- **params.lambda** (double) – The first stepsize used by the method (see λ from the original paper). Defaults to `0.99 / L`.
- **params.xi** (double) – Determines A_0 from the original paper (see ξ from the original paper). Defaults to `1.05 * m`.

Returns
A pair of structs containing model and history related outputs of the solved problem associated with the oracle and input parameters.

`src.solvers.`**`UPFAG`**(*oracle*, *params*)

The unified problem-parameter free accelerated gradient (UPFAG) method.

See also:

src/solvers/AG.m

Note: A version of the file provided by Ghadimi, Lan, and Zhang, which was heavily edited to conform with the CompModel framework. Based on the paper (see the UPFAG-fullBB method):

Ghadimi, S., Lan, G., & Zhang, H. (2019). Generalized uniformly optimal methods for nonlinear programming. *Journal of Scientific Computing, 79*(3), 1854-1881.

Parameters

- **`oracle`** (`Oracle`) – The oracle underlying the optimization problem.
- **`params`** (`struct`) – Contains instructions on how to call the algorithm.

Returns
A pair of structs containing model and history related outputs of the solved problem associated with the oracle and input parameters.

6.3 Spectral NCO Solvers

These solvers are used for spectral nonconvex composite optimization problems.

`src.solvers.`**`DA_ICG`**(*spectral_oracle*, *params*)

The doubly-accelerated inexact composite gradient (DA-ICG) method.

See also:

src/solvers/IA_ICG.m

Note: Based on the paper:

[1] Kong, W., & Monteiro, R. D. (2020). Accelerated Inexact Composite Gradient Methods for Nonconvex Spectral Optimization Problems. *arXiv preprint arXiv:2007.11772*.

Parameters

- **`spectral_oracle`** (`SpectralOracle`) – The spectral oracle underlying the optimization problem.
- **`params.xi`** (`double`) – Controls the amount of curvature that is initially redistributed from f_1 to f_2. Defaults to `params.M1`.
- **`params.lambda`** (`double`) – The initial stepsize λ. Defaults to `1 / params.M1`.
- **`params.steptype`** (`character vector`) – Either `'adaptive'` or `'constant'`. If it is `adaptive`, then the stepsize is chosen adaptively. If it is constant, then the stepsize is constant. Defaults to `'adaptive'`.

- **params.sigma** (`double`) – Controls the accuracy of the inner subroutine. Defaults to `(1 / 2 - max([params.lambda * (params.M1 - params.xi), 0]))`.
- **params.acg_steptype** (`character vector`) – Is either "variable" or "constant". If it is "variable", then the ACG call employs a line search subroutine to look for the appropriate upper curvature, with a starting estimate of $L_0 = \lambda(M/100) + 1$. If "constant", then no subroutine is employed and the upper curvature remains fixed at $L_0 = \lambda M + 1$. Defaults to `'variable'`.
- **params.Omega_projection** (`function handle`) – A one argument function, which when evaluated at x, computes $\text{Proj}_\Omega(x)$. For details see the definition of Ω in **[1]**. Defaults to `@(X) X`.
- **params.is_monotone** (`bool`) – If `true`, then the sequence of outer iterates forms a monotonically nonincreasing sequence with respect to the objective function. If `false`, then no such property is guaranteed. Defaults to `true`.

Returns

A pair of structs containing model and history related outputs of the solved problem associated with the oracle and input parameters.

`src.solvers.`**`IA_ICG`**(*spectral_oracle*, *params*)

The accelerated inexact composite gradient (AICG) method. The inner-accelerated inexact composite gradient (IA-ICG) method.

See also:

src/solvers/DA_ICG.m

Note: Based on the paper:

Kong, W., & Monteiro, R. D. (2020). Accelerated Inexact Composite Gradient Methods for Nonconvex Spectral Optimization Problems. *arXiv preprint arXiv:2007.11772*.

Parameters

- **spectral_oracle** (`SpectralOracle`) – The spectral oracle underlying the optimization problem.
- **params.xi** (`double`) – Controls the amount of curvature that is initially redistributed from f_1 to f_2. Defaults to `params.M1`.
- **params.lambda** (`double`) – The initial stepsize λ. Defaults to `1 / params.M1`.
- **params.steptype** (`character vector`) – Either `'adaptive'` or `'constant'`. If it is `adaptive`, then the stepsize is chosen adaptively. If it is constant, then the stepsize is constant. Defaults to `'adaptive'`.
- **params.sigma** (`double`) – Controls the accuracy of the inner subroutine. Defaults to `(9 / 10 - max([params.lambda * (params.M1 - params.xi), 0]))`.
- **params.acg_steptype** (`character vector`) – Is either "variable" or "constant". If it is "variable", then the ACG call employs a line search subroutine to look for the appropriate upper curvature, with a starting estimate of $L_0 = \lambda(M/100) + 1$. If "constant", then no subroutine is employed and the upper curvature remains fixed at $L_0 = \lambda M + 1$. Defaults to `'variable'`.

Returns

A pair of structs containing model and history related outputs of the solved problem associated with the oracle and input parameters.

CHAPTER

SEVEN

FRAMEWORKS

This section documents the frameworks that available for solving unconstrained composite optimization problems. For simplicity, we only list the parameter inputs that affect the behavior of the framework for a fixed optimization problem. All other inputs should be passed from the model object that is calling the framework.

Note: All frameworks have a parameter `i_logging` that, when set to `true`, will log the function value, (inner) iteration number, and time at each (outer) iteration. These values are then added to the `history` struct that is output by each framework.

`src.frameworks.`**`AIDAL`**`(~,` *oracle*, *params*`)`

An accelerated inexact dampened augmeneted Lagrangian (AIDAL) framework for solving a nonconvex composite optimization problem with linear constraints

Note: Based on the paper:

TBD

Parameters

- **`oracle`** (`Oracle`) – The oracle underlying the optimization problem.
- **`params`** (`struct`) – Contains instructions on how to call the framework.

Returns

A pair of structs containing model and history related outputs of the solved problem associated with the oracle and input parameters.

`src.frameworks.`**`IAIPAL`**`(~,` *oracle*, *params*`)`

An inner accelerated proximal inexact augmeneted Lagrangian (IAPIAL) framework for solving a nonconvex composite optimization problem with convex (linear or nonliner) cone constraints.

Note: Based on the papers:

[1] Melo, J. G., & Monteiro, R. D. (2020). Iteration-complexity of an inner accelerated inexact proximal augmented Lagrangian method based on the classical Lagrangian function and a full Lagrange multiplier update. *arXiv preprint arXiv:2008.00562*.

[2] Kong, W., Melo, J. G., & Monteiro, R. D. (2020). Iteration-complexity of a proximal augmented Lagrangian method for solving nonconvex composite optimization problems with nonlinear convex constraints. *arXiv preprint arXiv:2008.07080*.

Parameters

- **oracle** (`Oracle`) – The oracle underlying the optimization problem.
- **params** (`struct`) – Contains instructions on how to call the framework.

Returns

A pair of structs containing model and history related outputs of the solved problem associated with the oracle and input parameters.

`src.frameworks.`**`iALM`**`(~, `*oracle*, *params*`)`

An inexact augmented Lagrangian method (iALM) for solving a nonconvex composite optimization model with nonlinear equality constraints, i.e. $g(x) = 0$.

Note: Based on the paper:

Li, Z., Chen, P. Y., Liu, S., Lu, S., & Xu, Y. (2020). Rate-improved inexact augmented Lagrangian method for constrained nonconvex optimization. *arXiv preprint arXiv:2007.01284*.

Parameters

- **oracle** (`Oracle`) – The oracle underlying the optimization problem.
- **params** (`struct`) – Contains instructions on how to call the framework.

Returns

A pair of structs containing model and history related outputs of the solved problem associated with the oracle and input parameters.

`src.frameworks.`**`penalty`**`(`*solver*, *oracle*, *params*`)`

A quadratic penalty-based framework for solving a nonconvex composite optimization model with linear set constraints, i.e. $g(x) = Ax$ where A is a linear operator.

Note: Based on the paper:

Kong, W., Melo, J. G., & Monteiro, R. D. (2020). An efficient adaptive accelerated inexact proximal point method for solving linearly constrained nonconvex composite problems. *Computational Optimization and Applications, 76*(2), 305-346.

Parameters

- **solver** (`function handle`) – A solver for unconstrained composite optimization.
- **oracle** (`Oracle`) – The oracle underlying the optimization problem.
- **params** (`struct`) – Contains instructions on how to call the framework.

Returns

A pair of structs containing model and history related outputs of the solved problem associated with the oracle and input parameters.

`src.frameworks.`**`sProxALM`**`(~, `*oracle*, *params*`)`

A smoothed proximal augmented Lagrangian method (s-Prox-ALM) for solving a nonconvex composite optimization model with linear constraints, where the composite term is the indicator of a polyhedron.

Note: Based on the paper:

Zhang, J., & Luo, Z. (2020). A global dual error bound and its application to the analysis of linearly constrained nonconvex optimization. *arXiv preprint arXiv:2006.16440*.

Parameters

- **oracle** (`Oracle`) – The oracle underlying the optimization problem.
- **params** (`struct`) – Contains instructions on how to call the framework.

Returns

A pair of structs containing model and history related outputs of the solved problem associated with the oracle and input parameters.

CHAPTER

EIGHT

MODELS

This section documents the constructors, properties, and methods of the `CompModel` and `ConstrCompModel` classes. Hidden and inherited properties are not shown here for simplicity.

class `src.models.CompModel`(*varargin*)

Bases: `matlab.mixin.Copyable`

An abstract model class for composite optimization.

Note: The following properties are necessary before the `optimize()` method can be called to solve the model: either {`f_s`, `grad_f_s`} or `oracle`, `L`, `x0`, and `solver`.

`f_s`

A one argument function that, when evaluated at a point x, outputs $f_s(x)$. Defaults to `None`.

Type
function handle

`grad_f_s`

A one argument function that, when evaluated at a point x, outputs $\nabla f_s(x)$. Defaults to `None`.

Type
function handle

`f_n`

A one argument function that, when evaluated at a point x, outputs $f_n(x)$. Defaults to `@(x) zeros(size(x))`.

Type
function handle

`prox_f_n`

A two argument function that, when evaluated at $\{x, \lambda\}$, outputs

$$\text{prox}_{\lambda f_n}(x) := \text{argmin}_u \left\{ \lambda f_n(u) + \frac{1}{2}\|u - x\|^2 \right\}.$$

Defaults to `@(x, lam) x`.

Type
function handle

`L`

A **required** Lipschitz constant of ∇f_s. Defaults to `None`.

Type
double

`M`

An **optional** upper curvature constant of ∇f_s, i.e., a constant satisfying

$$f_s(z) - f_s(u) - \langle \nabla f_s(u), z - u \rangle \leq \frac{M}{2}\|u - z\|^2 \quad \forall u, z \in \operatorname{dom} f_n.$$

Defaults to `None`.

Type
double

`m`

An **optional** lower curvature constant of ∇f_s, i.e. a constant satisfying

$$f_s(z) - f_s(u) - \langle \nabla f_s(u), z - u \rangle \geq -\frac{m}{2}\|u - z\|^2 \quad \forall u, z \in \operatorname{dom} f_n.$$

Defaults to `None`.

Type
double

`x0`

A **required** starting point of the solver. Defaults to `None`.

Type
double vector

`oracle`

An **optional** oracle that may be given to the model, which will be passed to the solver if the flag i_update_oracle is False. Defaults to `None`.

Type
Oracle

`solver`

A **required** function handle to a solver that solves unconstrained composite optimization problems (see src/solvers). Defaults to `None`.

Type
function handle

`model`

The model struct output by the solver. Defaults to `None`.

Type
struct

`history`

The history struct output by the solver. Defaults to `None`.

Type
struct

`solver_hparams`

Contains additional hyperparameters that will be given to the solver when it is called. Defaults to `None`.

Type
struct

`iter_limit`

An upper bound on the total number of gradient/function/proximal evaluations done by the solver. Defaults to `Inf`.

Type
int

`time_limit`

An upper bound on the total time used by the solver. Defaults is `Inf`.

Type
double

`opt_tol`

The tolerance for optimality, i.e. $\rho =$

Type
double

`opt_type`

Is either 'relative' or 'absolute'. If it is 'absolute', then the optimality condition is $\|v\| \leq$

Type
character vector

`prod_fn`

A two argument function that, when evaluated at $\{a, b\}$, outputs the inner product $\langle a, b\rangle$. Defaults to the Euclidean inner product, i.e. `@(a,b) sum(dot(a, b))`.

Type
function handle

`norm_fn`

A one function that, when evaluated at a point a, outputs $\|a\|$. Defaults to the Frobenius norm, i.e. `norm(a, 'fro')`.

Type
function handle

`iter`

The number of gradient/function/proximal evaluations done by the solver. Defaults to `0`. Cannot be set by the user.

Type
int

`runtime`

The total runtime used by the solver. Defaults to `0.0`. Cannot be set by the user.

Type
double

`x`

The stationary point returned by the solver. Defaults to `None`. Cannot be set by the user.

Type
double vector

v

The stationary residual returned by the solver. Defaults to `None`. Cannot be set by the user.

Type
double vector

CONSTRUCTORS

CompModel(*varargin*)

The constructor for the CompModel class. It has three ways to initialize: **(i)** invoking `CompModel` creates a CompModel object with the default properties; **(ii)** invoking `CompModel(in_oracle)` creates a CompModel object with the oracle property set to the input oracle; and **(iii)** invoking `CompModel(f_s, f_n, grad_f_s, prox_f_n)` creates an Oracle object with the properties `f_s`, `f_n`, `grad_f_s`, and `prox_f_n` filled by the corresponding input.

view_flags()

Displays flags that control the behavior of `optimize()`.

view_topology()

Displays functions related to the underlying inner product space.

view_solution()

Displays metrics related to the obtained solution by the solver.

view_curvatures()

Displays the underlying curvatures.

view_history()

Displays the history structure output by the solver.

class src.models.**ConstrCompModel**

Bases: *src.models.CompModel*

An abstract model class for constrained composite optimization.

Note: The following (non-inherited) properties are necessary before the `optimize()` method can be called to solve the model: `framework`, `constr_fn`, `grad_constr_fn`, `set_projector`, and `K_constr`.

framework

A **required** function handle to a framework that solves constrained composite optimization problems (see src/frameworks). Defaults to `None`.

Type
function handle

constr_fn

A **required** one argument function that, when evaluated at a point x, returns the constraint function at that point, i.e. $g(x)$. Defaults to `@(x) 0`.

Type
function handle

grad_constr_fn

A **required** function that represents the gradient of the constraint function. Has two possible prototypes: (i) a one argument function that, when evaluated at a point x, returns $\nabla g(x)$; and (ii) a two argument function that, when evaluated at $\{x, \delta\}$, returns $\nabla g(x)\delta$. Defaults to `@(x) zeros(size(x))`.

Type
function handle

`set_projector`

A one argument function that, when evaluated at a point x, returns the projection of x onto the set S. Defaults to `@(x) zeros(size(x))`.

Type
function handle

`primal_cone_project`

A one argument function that, that, when evaluated at a point x, returns the projection of x onto the cone K. Defaults to `@(x) zeros(size(x))`.

Type
function handle

`dual_cone_projector`

A one argument function that, when evaluated at a point x, returns the projection of x onto the dual cone K^{*}. Defaults to `@(x) x`.

Type
function handle

`B_constr`

A bound on the norm of the constraint function over the domain of f_n. Defaults to 0.0.

`K_constr`

A **required** Lipschitz constant of the constraint function. Defaults to `None`.

`L_constr`

A Lipschitz constant of the gradient of the constraint function. Defaults to 0.0.

`feas_tol`

The tolerance for feasibility, i.e. $\eta =$

Type
double

`feas_type`

Is either 'relative' or 'absolute'. If it is 'absolute', then the optimality condition is $\|w\| \leq$

Type
character vector

`w`

The feasibility residual returned by the solver. Defaults to `None`. Cannot be set by the user.

Type
double vector

CHAPTER NINE

EXAMPLES

This section documents the examples solved by NC-OPT. For brevity, only the optimization model is given. Moreover, to avoid repetition, we will also define

$$\delta_C(x) = \begin{cases} 0, & x \in C, \\ \infty, & x \notin C, \end{cases}$$

for any closed convex set C and any point x.

Note: Additional details about these problems can be found in the papers:

[1] Kong, W., Melo, J. G., & Monteiro, R. D. (2019). Complexity of a quadratic penalty accelerated inexact proximal point method for solving linearly constrained nonconvex composite programs. *SIAM Journal on Optimization, 29*(4), 2566-2593.

[2] Kong, W., Melo, J. G., & Monteiro, R. D. (2020). An efficient adaptive accelerated inexact proximal point method for solving linearly constrained nonconvex composite problems. *Computational Optimization and Applications, 76*(2), 305-346.

[3] Kong, W., & Monteiro, R. D. (2019). An accelerated inexact proximal point method for solving nonconvex-concave min-max problems. *arXiv preprint arXiv:1905.13433*.

[4] Kong, W., & Monteiro, R. D. (2020). Accelerated Inexact Composite Gradient Methods for Nonconvex Spectral Optimization Problems. *arXiv preprint arXiv:2007.11772*.

9.1 Unconstrained Problems

This subsection considers unconstrained composite optimization problems.

`examples/unconstrained/basic_convex_qp.m`

This example solves the convex univariate optimization problem

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \frac{1}{2}x^2 - x + \frac{1}{2} \\ \text{subject to} \quad & x \in \mathbb{R}. \end{aligned}$$

`examples/unconstrained/nonconvex_qp`

This example solves the nonconvex quadratic programming problem

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & -\frac{\xi}{2}\|DBx\|^2 + \frac{\tau}{2}\|Ax - b\|^2 + \delta_{\Delta^n}(x) \\ \text{subject to} \quad & x \in \mathbb{R}^n, \end{aligned}$$

where Δ^n is the unit simplex given by

$$\Delta^n := \left\{ x \in \mathbb{R}^n : \sum_{i=1}^n x_i = 1, 0 \le x \le 1 \right\}.$$

`examples/unconstrained/nonconvex_qsdp.m`

This example solves the nonconvex quadratic semidefinite programming problem

$$\begin{array}{ll} \underset{X}{\text{minimize}} & -\frac{\xi}{2}\|DB(X)\|_F^2 + \frac{\tau}{2}\|A(X)-b\|_F^2 + \delta_{P^n}(X) \\ \text{subject to} & X \in \mathbb{S}_+^n, \end{array}$$

where $\mathbb{S}_+^n$ is the collection of positive semidefinite matrices and P^n is the unit spectrahedron given by

$$P^n := \left\{ X \in \mathbb{S}_+^n : \operatorname{tr} X = 1 \right\}.$$

`examples/unconstrained/nonconvex_svm.m`

This example solves the nonconvex support vector machine problem

$$\begin{array}{ll} \underset{x}{\text{minimize}} & \frac{1}{n}\sum_{i=1}^n \left[1 - \tanh\left(v_i \langle u_i, x\rangle\right)\right] + \frac{1}{2n}\|x\|^2 \\ \text{subject to} & x \in \mathbb{R}^n. \end{array}$$

9.2 Nonconvex-Concave Min-Max Problems

This subsection considers nonconvex-concave min-max composite optimization problems of the form

$$\begin{array}{ll} \underset{x}{\text{minimize}} & \left[\max_y \, \Phi(x,y) + h(x)\right] \\ \text{subject to} & x \in \mathbb{R}^n, \quad y \in \mathbb{R}^m \end{array}$$

under a saddle-point termination criterion based on the one given in **[3]**. More specifically, given a tolerance pair $(\rho_x, \rho_y) \in \mathbb{R}_{++}^2$, the goal is to find a quadruple (x, y, v, w) satisfying the conditions

$$\begin{pmatrix} v \\ w \end{pmatrix} \in \begin{pmatrix} \nabla_x \Phi(x,y) \\ 0 \end{pmatrix} + \begin{pmatrix} \partial h(x) \\ \partial\left[-\Phi(x,\cdot)\right](y) \end{pmatrix}, \quad \|v\| \le \rho_x, \quad \|w\| \le \rho_y.$$

`examples/minmax/nonconvex_minmax_qp.m`

This example solves the nonconvex minmax quadratic programming problem

$$\begin{array}{ll} \underset{x}{\text{minimize}} & \left[\max_{i\in\{1,\ldots,k\}} \; -\frac{\xi_i}{2}\|D_iB_ix\|^2 + \frac{\tau_i}{2}\|A_ix-b\|^2 + \delta_{\Delta^n}(x)\right] \\ \text{subject to} & x \in \mathbb{R}^n, \end{array}$$

where Δ^n is the unit simplex given by

$$\Delta^n := \left\{ x \in \mathbb{R}^n : \sum_{i=1}^n x_i = 1, 0 \le x \le 1 \right\}.$$

`examples/minmax/nonconvex_power_control.m`

This example solves the nonconvex power control problem

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \left[\max_{y} \sum_{k=1}^{K}\sum_{n=1}^{N} f_{k,n}(X,y) + \delta_{B_x}(x) + \delta_{B_y}(y)\right] \\ \text{subject to} \quad & x \in \mathbb{R}^{K\times N}, \quad y \in \mathbb{R}^{N}, \end{aligned}$$

where $f_{k,n}$, B_x, and B_y are given by

$$\begin{aligned} f_{k,n}(X,y) &:= -\log\left(1 + \frac{\mathcal{A}_{k,k,n}X_{k,n}}{\sigma^2 + B_{k,n}y_n + \sum_{j=1,j\neq k}^{K}\mathcal{A}_{j,k,n}X_{j,n}}\right), \\ B_x &:= \left\{X \in \mathbb{R}^{K\times N} : 0 \le X \le R\right\}, \\ B_y &:= \left\{y \in \mathbb{R}^{N} : 0 \le y \le \frac{N}{2}\right\}. \end{aligned}$$

`examples/minmax/nonconvex_robust_regression.m`

This example solves the robust regression problem

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \left[\max_{i\in\{1,\ldots,n\}} \phi_\alpha \circ \ell_i(x)\right] \\ \text{subject to} \quad & x \in \mathbb{R}^{k}, \end{aligned}$$

where ϕ_α and ℓ_j are given by

$$\phi_\alpha(t) := \alpha\log\left(1 + \frac{t}{\alpha}\right), \quad \ell_j := \log\left(1 + e^{-b_j\langle a_j, x\rangle}\right).$$

9.3 Spectral Problems

This subsection considers spectral optimization problems where $f_s = f_{1,s} + f_{2,s}^{\mathcal{V}} \circ \sigma$ and $f_n = f_n^{\mathcal{V}} \circ \sigma$ for absolutely symmetric functions $f_{2,s}^{\mathcal{V}}$ and $f_n^{\mathcal{V}}$.

`examples/spectral/nonconvex_spectral_mc.m`

This example solves the spectral matrix completion problem

$$\begin{aligned} \underset{X}{\text{minimize}} \quad & \frac{1}{2}\|\text{Proj}_\Omega(X - A)\|_F^2 + \mathcal{R}_\mu \circ \sigma(X) + \delta_{B_R}(X) \\ \text{subject to} \quad & X \in \mathbb{R}^{p\times q}, \end{aligned}$$

where $\mathcal{R}_\mu$ is an absolutely symmetric regularization function, $\sigma(\cdot)$ is the function that maps a matrix to its vector of singular values, and B_R is the Euclidean ball of radius R, i.e., $B_R := \{X \in \mathbb{R}^{p\times q} : \|X\|_F \le R\}$.

`examples/spectral/nonconvex_spectral_bmc.m`

This example solves the spectral blockwise matrix completion problem

$$\begin{aligned} \underset{X}{\text{minimize}} \quad & \frac{1}{2}\|\text{Proj}_\Omega(X - A)\|_F^2 + \mathcal{R}_\mu \circ \sigma(X_i) + \delta_{B_R}(X) \\ \text{subject to} \quad & X \in \mathbb{R}^{p\times q}, \end{aligned}$$

where $\{X_i\}_{i=1}^{k}$ are block components of the matrix X, $\mathcal{R}_\mu$ is an absolutely symmetric regularization function, $\sigma(\cdot)$ is the function that maps a matrix to its vector of singular values, and B_R is the Euclidean ball of radius R, i.e., $B_R := \{X \in \mathbb{R}^{p\times q} : \|X\|_F \le R\}$.

9.4 Linearly-Set Constrained Problems

This subsection considers linearly-set constrained composite optimization problems where $g(x) = Ax$ for a linear operator A and S is a closed convex set.

`examples/constrained/lin_constr_nonconvex_qp.m`

This example solves the linearly-constrained nonconvex quadratic programming problem

$$\begin{array}{ll} \underset{x}{\text{minimize}} & -\frac{\xi}{2}\|DBx\|^2 + \frac{\tau}{2}\|Ax-b\|^2 + \delta_{\Delta^n}(x) \\ \text{subject to} & Cx = d, \quad x \in \mathbb{R}^n, \end{array}$$

where Δ^n is the unit simplex given by

$$\Delta^n := \left\{ x \in \mathbb{R}^n : \sum_{i=1}^n x_i = 1, 0 \le x \le 1 \right\}.$$

`examples/constrained/nonconvex_lin_constr_qsdp.m`

This example solves the linearly-constrained nonconvex quadratic semidefinite programming problem

$$\begin{array}{ll} \underset{X}{\text{minimize}} & -\frac{\xi}{2}\|DB(X)\|_F^2 + \frac{\tau}{2}\|A(X)-b\|_F^2 + \delta_{P^n}(X) \\ \text{subject to} & C(X) = d, \quad X \in \mathbb{S}_+^n, \end{array}$$

where $\mathbb{S}_+^n$ is the collection of positive semidefinite matrices and P^n is the unit spectrahedron given by

$$P^n := \left\{ X \in \mathbb{S}_+^n : \operatorname{tr} X = 1 \right\}.$$

`examples/constrained/nonconvex_sparse_pca.m`

This example solves the nonconvex sparse principal component analysis problem

$$\begin{array}{ll} \underset{\Pi,\Phi}{\text{minimize}} & \langle \Sigma, \Pi \rangle + \sum_{i,j=1}^n q_\nu(\Phi_{ij}) + \nu \sum_{i,j=1}^n |\Phi_{ij}| + \delta_{\mathcal{F}^k}(\Pi) \\ \text{subject to} & \Pi - \Phi = 0, \quad (\Pi, \Phi) \in \mathbb{R}^{n\times n} \times \mathbb{R}^{n \times n}, \end{array}$$

where $\mathcal{F}^k$ is the k-Fantope given by

$$\mathcal{F}^k := \left\{ X \in \mathbb{S}_+^n : 0 \preceq X \preceq I, \operatorname{tr} X = k \right\}.$$

`examples/constrained.nonconvex_bounded_mc.m`

This example solves the nonconvex bounded matrix completion problem

$$\begin{array}{ll} \underset{X}{\text{minimize}} & \frac{1}{2}\|\text{Proj}_\Omega(X - A)\|_F^2 + \mathcal{R}_\mu(X) \\ \text{subject to} & X \in B_{[l,u]}, \end{array}$$

where $\mathcal{R}_\mu$ is a nonconvex regularization function and $B_{[l,u]}$ is the box given by

$$B_{[l,u]} := \left\{ X \in \mathbb{R}^{p\times q} : l \le X_{ij} \le u, (i,j) \in \{1,...,p\} \times \{1,...,q\} \right\}.$$

CHAPTER

TEN

CONTACT

Additional questions, feedback, bug reports, and suggestions for improvement can be made on the GitHub repository or by emailing the author at: wwkong92 [at] gmail [dot] com.

MATLAB MODULE INDEX

INDEX