## CONTENTS

1 Installation 3

2 Quick start 5
   2.1 Example with the consensus algorithm ................................. 5
   2.2 Example with distributed optimization ............................... 5

3 Tutorial 7
   3.1 Distributed optimization set-ups ..................................... 7
   3.2 Objective functions and constraints ................................... 9
   3.3 Local data of optimization problems ................................ 11
   3.4 Agents in the network .................................................. 13
   3.5 Algorithms ............................................................ 14

4 Examples 17
   4.1 Basic examples ....................................................... 17
   4.2 Complex examples ................................................... 55

5 API Documentation 71
   5.1 Agents ............................................................... 71
   5.2 Communicators ....................................................... 73
   5.3 Algorithms .......................................................... 75
   5.4 Functions ........................................................... 100
   5.5 Constraints ........................................................ 110
   5.6 Problem Classes .................................................... 116
   5.7 Utilities ............................................................ 121

6 Advanced features 125
   6.1 Optimization Problems ............................................... 125
   6.2 Implementing custom functions ....................................... 126

7 Acknowledgements 127

Bibliography 129

Index 131
**disropt** is a Python package developed within the excellence research program ERC in the project OPT4SMART. The aim of this package is to provide an easy way to run distributed optimization algorithms that can be executed by a network of peer computing systems.

A comprehensive guide to **disropt** can be found in the Tutorial. Many examples are provided in the Examples section, while the API Documentation can be checked for more details. The package is equipped with some commonly used objective functions and constraints which can be directly used.

**disropt** currently supports MPI in order to emulate peer-to-peer communication. However, custom communication protocols can be also implemented.

For example, the following Python code generates an unconstrained, quadratic optimization problem with a 5-dimensional decision variable, which is solved using the so-called Gradient Tracking.

**Listing 1: basic_example.py**

```python
import numpy as np
from disropt.agents import Agent
from disropt.algorithms.gradient_tracking import GradientTracking
from disropt.functions import QuadraticForm, Variable
from disropt.utils.graph_constructor import MPIgraph
from disropt.problems import Problem

# generate communication graph (everyone uses the same seed)
comm_graph = MPIgraph('random_binomial', 'metropolis')
agent_id, in_nbrs, out_nbrs, in_weights, _ = comm_graph.get_local_info()

# size of optimization variable
n = 5

# generate quadratic cost function
np.random.seed()
Q = np.random.randn(n, n)
Q = Q.transpose() @ Q
x = Variable(n)
func = QuadraticForm(x - np.random.randn(n, 1), Q)

# create Problem and Agent
agent = Agent(in_nbrs, out_nbrs, in_weights=in_weights)
agent.set_problem(Problem(func))

# run the algorithm
x0 = np.random.rand(n, 1)
algorithm = GradientTracking(agent, x0)
algorithim.run(iterations=1000, stepsize=0.01)

print("Agent {0} - solution estimate: {1}".format(agent_id, algorithm.get_result().flatten()))
```

This code can be executed over a network with 8 agents by issuing the following command:

```bash
mpirun -np 8 python example.py
```
The `disropt` package supports Python 3 and can be installed through pip by running in your terminal:

```
pip install disropt
```

An MPI implementation needs to be installed on your platform in order to use `disropt`. 
For the installation of the package, refer to the *Installation* section.

To run an algorithm, it suffices to create an instance of the corresponding class and then call the method `run()`. The class constructor requires an instance of the *Agent* class, which must contain the local information available to the agent to run the algorithm.

### 2.1 Example with the consensus algorithm

For example, to run the *Consensus* algorithm, first create an instance of the *Agent* class with the graph information:

```python
agent = Agent(in_neighbors, out_neighbors, in_weights)
```

where the variables `in_neighbors`, `out_neighbors` and `in_weights` are previously initialized lists. Then, create an instance of the *Consensus* class with the agent’s initial condition and call the method `run()`:

```python
algorithm = Consensus(agent=agent, initial_condition=x0)
algorithm.run(iterations=100)
```

The method `get_result()` can be called to get the output of the algorithm:

```python
print("Output of agent \{\}: \{\}.".format(agent.id, algorithm.get_result()))
```

All the code showed so far is python code and must be enclosed in a script file. To actually run the code with MPI (which is the default *Communicator*), run on a terminal:

```bash
mpirun -np 8 python script.py
```

where in this case the script file `script.py` is executed over 8 processors.

### 2.2 Example with distributed optimization

For distributed optimization algorithms, the workflow is almost the same, except that the *Agent* class must be equipped with the problem data that is locally available to the agent. The problem data should be passed as an instance of the *Problem* class (or one of its children) before creating the instance of the algorithm class.

For example, to run the *Distributed subgradient* algorithm, the cost function must be passed to the instance of the *Agent* class after its initialization:

```python
problem = Problem(objective_function)
agent.set_problem(problem)
```
where the variable \texttt{objective\_function} is the agent’s objective function in the cost-coupled problem.

Then, the algorithm can be run just like in the Consensus case:

```python
algorithm = SubgradientMethod(agent=agent, initial_condition=x0)
algorithm.run(iterations=100)
print("Output of agent {}:
{:.}").format(agent.id, algorithm.get_result())
```

and on the terminal:

```
mpirun -np 8 python script.py
```
disropt is a Python package for distributed optimization over peer-to-peer networks of computing units called agents. The main idea of distributed optimization is to solve an optimization problem (enjoying a given structure) over a (possibly unstructured) network of processors. Each agent can perform local computation and can exchange information with only its neighbors in the network. A distributed algorithm consists of an iterative procedure in which each agent maintains a local estimate of the problem solution which is properly updated in order to converge towards the solution.

Formally, an optimization problem is a mathematical problem which consists in finding a minimum of a function while satisfying a given set of constraints. In symbols,

$$\min_x f(x)$$

subject to \( x \in X \),

where \( x \in \mathbb{R}^d \) is called optimization variable, \( f : \mathbb{R}^d \rightarrow \mathbb{R} \) is called cost function and \( X \subseteq \mathbb{R}^d \) describes the problem constraints. The optimization problem is assumed to be feasible and has finite optimal cost. Thus, it admits at least an optimal solution that is usually denoted as \( x^* \). The optimal solution is a vector that satisfies all the constraints and attains the optimal cost.

### 3.1 Distributed optimization set-ups

Distributed optimization problems usually arising in applications usually enjoy a proper structure in their mathematical formulation. In disropt, three different optimization set-ups are available. As for their solution, see the Algorithms page for a list of all the implemented distributed algorithms (classified by optimization set-up to which they apply).

#### 3.1.1 Cost-coupled set-up

In this optimization set-up, the cost function is expressed as the sum of cost functions \( f_i \) and all of them depend on a common optimization variable \( x \). Formally, the set-up is

$$\min_x \sum_{i=1}^N f_i(x)$$

subject to \( x \in X \),

where \( x \in \mathbb{R}^d \) and \( X \subseteq \mathbb{R}^d \). The global constraint set \( X \) is common to all agents, while \( f_i : \mathbb{R}^d \rightarrow \mathbb{R} \) is assumed to be known by agent \( i \) only, for all \( i \in \{1, \ldots, N\} \).
In some applications, the constraint set $X$ can be expressed as the intersection of local constraint sets, i.e.,

$$X = \bigcap_{i=1}^{N} X_i$$

where each $X_i \subseteq \mathbb{R}^d$ is meant to be known by agent $i$ only, for all $i \in \{1, \ldots, N\}$.

The goal for distributed algorithms for the cost-coupled set-up is that all agent estimates are eventually consensual to an optimal solution $x^*$ of the problem.

### 3.1.2 Common cost set-up

In this optimization set-up, there is a unique cost function $f$ that depends on a common optimization variable $x$, and the optimization variable must further satisfy local constraints. Formally, the set-up is

$$\min_x f(x) \quad \text{subject to } x \in \bigcap_{i=1}^{N} X_i,$$

where $x \in \mathbb{R}^d$ and each $X_i \subseteq \mathbb{R}^d$. The cost function $f$ is assumed to be known by all the agents, while each set $X_i$ is assumed to be known by agent $i$ only, for all $i \in \{1, \ldots, N\}$.

The goal for distributed algorithms for the common-cost set-up is that all agent estimates are eventually consensual to an optimal solution $x^*$ of the problem.

### 3.1.3 Constraint-coupled set-up

In this optimization set-up, the cost function is expressed as the sum of local cost functions $f_i$ that depend on a local optimization variable $x_i$. The variables must satisfy local constraints (involving only each optimization variable $x_i$) and global coupling constraints (involving all the optimization variables). Formally, the set-up is

$$\min_{x_1, \ldots, x_N} \sum_{i=1}^{N} f_i(x_i)$$

$$\text{subject to } x_i \in X_i, \quad i \in \{1, \ldots, N\}$$

$$\sum_{i=1}^{N} g_i(x_i) \leq 0,$$

where each $x_i \in \mathbb{R}^{d_i}$, $X_i \subseteq \mathbb{R}^{d_i}$, $f_i : \mathbb{R}^{d_i} \to \mathbb{R}$ and $g_i : \mathbb{R}^{d_i} \to \mathbb{R}^S$ for all $i \in \{1, \ldots, N\}$. Here the symbol $\leq$ is also used to denote component-wise inequality for vectors. Therefore, the optimization variable consists of the stack of all $x_i$, namely the vector $(x_1, \ldots, x_N)$. All the quantities with the index $i$ are assumed to be known by agent $i$ only, for all $i \in \{1, \ldots, N\}$. The function $g_i$, with values in $\mathbb{R}^S$, is used to express the $i$-th contribution to $S$ coupling constraints among all the variables.

The goal for distributed algorithms for the constraint-coupled set-up is that each agent estimate is asymptotically equal to its portion $x_i^* \in X_i$ of an optimal solution $(x_1^*, \ldots, x_N^*)$ of the problem.
3.2 Objective functions and constraints

3.2.1 Functions

disropt comes with many already implemented mathematical functions. Functions are defined in terms of optimization variables (Variable) or other functions. Let us start by defining a Variable object as:

```python
from disropt.functions import Variable
n = 2  # dimension of the variable
x = Variable(n)
print(x.input_shape)  # -> (2, 1)
print(x.output_shape)  # -> (2, 1)
```

Now, suppose you want to define an affine function \( f(x) = A^T x - b \) with \( A \in \mathbb{R}^{2 \times 2} \) and \( b \in \mathbb{R}^2 \):

```python
import numpy as np
a = 1
A = np.array([[1, 2], [2, 4]])
b = np.array([[1], [1]])
f = A @ x - b
# or, alternatively
from disropt.functions import AffineForm
f = AffineForm(x, A, b)
```

The composition of functions is fully supported. Suppose you want to define a function \( g(x) = f(x)^T Q f(x) \), then:

```python
from disropt.functions import QuadraticForm
Q = np.random.rand(2, 2)
g = QuadraticForm(f, Q)  # or: g = f @ (Q.transpose() @ f)
print(g.input_shape)  # -> (2, 1)
print(g.output_shape)  # -> (1, 1)
```

Currently supported operations with functions are sum (+), difference (-), product (*) and matrix product (@). Combination with numpy arrays is supported as well.

Function properties and methods

Each function has three properties that can be checked: differentiability, being affine and quadratic:

```python
g.is_differentiable  # -> True
g.is_affine  # -> False
g.is_quadratic  # -> True
f.is_affine  # -> True
```

and their input and output shapes can be obtained as

```python
g.output_shape  # -> (1,1)
g.input_shape  # -> (2,1)
```

Moreover, it is possible to evaluate functions at desired points and to obtain the corresponding (sub)gradient/jacobian/hessian as:

```python
pt = np.random.rand(2, 1)
# the value of g computed at pt is obtained as
```

(continues on next page)
For affine and quadratic functions, a method called `get_parameters` is implemented, which returns the matrices and vectors that define those functions. The generic form for an affine function is $A^T x + b$ while the one for a quadratic form is $x^T P x + q^T x + r$:

```py
f = A @ x + b
f.get_parameters() # -> A, b
```

### 3.2.2 Defining constraints from functions

Constraints are represented in the canonical forms $f(x) = 0$ and $f(x) \leq 0$.

They are directly obtained from functions:

```py
constraint = g == 0 # g(x) = 0
constraint = g >= 0 # g(x) >= 0
constraint = g <= 0 # g(x) <= 0
```

On the right side of (in)equalities, numpy arrays and functions (with appropriate shapes) are also allowed:

```py
c = np.random.rand(2,1)
constr = f <= c
```

which is automatically translated in the corresponding canonical form.

Constraints can be evaluated at any point by using the `eval` method which returns a boolean value if the constraint is satisfied. Moreover, the function defining a constraints can be retrieved with the `function` method:

```py
pt = np.random.rand(2,1)
constr.eval(pt) # -> True if f(pt) <= c
constr.function.eval(pt) # -> value of f - c at pt
```

### Affine and quadratic constraints

Parameters defining affine and quadratic constraints can be easily obtained. They can be accessed by calling the `get_parameters` method:

```py
f = A @ x + b
constraint = f == 0 # affine equality constraint
# f has the form A^T x + b
constraint.get_parameters() # returns A and b

q = f @ f
constraint = q == 0 # quadratic equality constraint
```
Projection onto a constraint set

The projection of a point onto the set defined by a constraint can be computed via the `projection` method:

```python
projected_point = f.projection(pt)
```

Constraint sets

Some particular constraint sets (for which projection of points is easy to compute) are also available through specialized classes, which are extensions of the class `Constraint`. For instance, suppose you want all the components of $x$ to be in $[-1, 1]$. Then you can define a `Box` constraint as:

```python
from disropt.constraints import Box
bound = np.ones((2,1))
constr = Box(-bound, bound)
```

Two methods are available: `projection` and `intersection`. The first one returns the projection of a given point on the set, while the second one intersects the set with another one. This feature is particularly useful in set-membership estimation algorithms.

Constraint sets can be converted into a list of constraints through the method `to_constraints`.

### 3.3 Local data of optimization problems

The class `Problem` allows one to define and solve optimization problems of various types. It is discussed in detail in a dedicated section.

In the distributed framework of `disropt`, the `Problem` class is also meant to specify local data (available to the agent) of global optimization problems. The class should be used in different ways, depending on the distributed optimization set-up (refer to the general forms), and must be provided to the agent (see also Quick start).

#### 3.3.1 Cost-coupled set-up

For the cost-coupled set-up, the two objects that can be specified are

- the local contribution to the cost function, i.e., the function $f_i(x)$
- the local constraints (if any), i.e., the set $X_i$ (which must be described through a list of constraints).

To this end, create an instance of the class `Problem` and set the objective function to $f_i(x)$ and the constraints to $X_i$.

For instance, suppose $x \in \mathbb{R}^2$ and assume the agent knows

$$f_i(x) = \|x\|^2, \quad X_i = \{x \mid -1 \leq x \leq 1\}.$$
from disropt.functions import SquaredNorm, Variable
from disropt.problems import Problem

x = Variable(2)
objective_function = SquaredNorm(x)
constraints = [x >= -1, x <= 1]
problem = Problem(objective_function, constraints)

If there are no local constraints (in the example $X_i \equiv \mathbb{R}^2$), then no constraints should be passed to Problem:

x = Variable(2)
objective_function = SquaredNorm(x)
problem = Problem(objective_function)  # no constraints

### 3.3.2 Common-cost set-up

For the common-cost set-up, the objective function $f(x)$ is assumed to be known by all the agents. Therefore, the two objects that must be specified are

- the global cost function, i.e., the function $f(x)$
- the local constraints, i.e., the set $X_i$ (which must be described through a list of constraints).

To this end, create an instance of the class Problem and set the objective function to $f(x)$ and the constraints to $X_i$.

For instance, suppose $x \in \mathbb{R}^2$ and assume the agent knows

$$f(x) = \|x\|^2, \quad X_i = \{x \mid -1 \leq x \leq 1\}.$$  

The corresponding Python code is:

from disropt.functions import SquaredNorm, Variable
from disropt.problems import Problem
x = Variable(2)
objective_function = SquaredNorm(x)
constraints = [x >= -1, x <= 1]
problem = Problem(objective_function, constraints)

### 3.3.3 Constraint-coupled set-up

For the constraint-coupled set-up, the three objects that can be specified are

- the local contribution to the cost function, i.e., the function $f_i(x_i)$
- the local contribution to the coupling constraints, i.e., the function $g_i(x_i)$
- the local constraints (if any), i.e., the set $X_i$ (which must be described through a list of constraints).

To this end, create an instance of the class ConstraintCoupledProblem and set the objective function to $f_i(x_i)$, the coupling function to $g_i(x_i)$ and the constraints to $X_i$.

For instance, suppose $x_i \in \mathbb{R}^2$ and assume the agent knows

$$f_i(x_i) = \|x_i\|^2, \quad g_i(x_i) = x_i, \quad X_i = \{x \mid -1 \leq x \leq 1\}.$$  

The corresponding Python code is:  

from disropt.functions import SquaredNorm, Variable
from disropt.problems import ConstraintCoupledProblem
x = Variable(2)
objective_function = SquaredNorm(x)
coupling_function = x
coupons = [x >= -1, x <= 1]
problem = ConstraintCoupledProblem(objective_function, coupons, coupling_function)

If there are no local constraints (in the example $X_i \equiv \mathbb{R}^2$), then no constraints should be passed to ConstraintCoupledProblem:

```python
x = Variable(2)
objective_function = SquaredNorm(x)
coupling_function = x
problem = ConstraintCoupledProblem(
    objective_function=objective_function,
    coupling_function=coupling_function)  # no local constraints
```

### 3.4 Agents in the network

The `Agent` class is meant to represent the local computing units that collaborate in the network in order to solve some specific problem.

Agents are instantiated by defining their in/out-neighbors and the weights they assign their neighbors. For example, consider the following network

![Network Diagram]

Then, agent 0 is defined as:

```python
from disropt.agents import Agent
agent = Agent(in_neighbors=[1,2],
              out_neighbors=[2],
              weights=[0.3, 0.2])
```
3.4.1 Local data of an optimization problem

Assigning a local optimization problem to an agent is done via the `set_problem` method, which modifies the `problem` attribute of the agent.

Assume that the variable `problem` contains the local problem data, according to the procedure described in the previous page. Then, the variable is assigned to the agent by:

```python
agent.set_problems(problem)
```

Local objective functions, constraints and all the operations related to the problem can be accessed through the attribute `problem`. For example:

```python
agent.problem.objective_function.eval(pt)  # evaluate the objective function at pt
agent.problem.constraints                   # -> return the list of local constraints
```

3.5 Algorithms

In disropt, there are many implemented distributed optimization algorithms. Each algorithm is tailored for a specific distributed optimization set-up (see Tutorial).

3.5.1 Basic

- *Consensus* (standard and block wise, synchronous and asynchronous)

3.5.2 Cost-coupled set-up

- *Distributed Subgradient* (standard and block wise)
- *Gradient Tracking*
- *Distributed Dual Decomposition*
- *Distributed ADMM*
- *ASYMM*

3.5.3 Common cost set-up

- *Constraints Consensus*
- *Set membership* (synchronous and asynchronous)
3.5.4 Constraint-coupled set-up

- Distributed Dual Subgradient
- Distributed Primal Decomposition

3.5.5 Miscellaneous

- Logic AND (synchronous and asynchronous)
Here we report some examples that show how to use **disropt**. We divide the examples into two groups:

- **Basic examples**: to show how each algorithm should be used
- **Complex examples**: realistic application scenarios, e.g., to make comparative studies on different algorithms

### 4.1 Basic examples

We provide an example for each implemented distributed algorithm (see also the *list of implemented algorithms*).

#### 4.1.1 Consensus algorithms

**Classical consensus**

The classical consensus algorithm is implemented through the `Consensus` class.

From the perspective of agent $i$ the classical consensus algorithm works as follows. For $k = 0, 1, \ldots$

$$x_i^{k+1} = \sum_{j=1}^{N} w_{ij} x_j^k$$

where $x_i \in \mathbb{R}^n$ and $w_{ij}$ is the weight assigned by agent $i$ to agent $j$. Usually, average consensus (i.e., the convergence of the local estimate sequences to the initial average) is guaranteed only if the weights $w_{ij}$ form a doubly-stochastic matrix. Otherwise, agreement is still reached but at some other point.

In order to simulate a consensus algorithm over a static undirected graph (with a doubly-stochastic weight matrix), create a file containing the following code and call it `launcher.py`

```python
import numpy as np
from mpi4py import MPI
from disropt.agents import Agent
from disropt.algorithms import Consensus
from disropt.utils.graph_constructor import binomial_random_graph, metropolis_hastings

# get MPI info
comm = MPI.COMM_WORLD
nproc = comm.Get_size()
local_rank = comm.Get_rank()
```

Listing 1: launcher.py

(continues on next page)
# Generate a common graph (everyone use the same seed)
Adj = binomial_random_graph(nproc, p=0.3, seed=1)
W = metropolis_hastings(Adj)

# reset local seed
np.random.seed()

# create local agent
agent = Agent(in_neighbors=np.nonzero(Adj[local_rank, :])[0].tolist(),
               out_neighbors=np.nonzero(Adj[:, local_rank])[0].tolist(),
               in_weights=W[local_rank, :].tolist())

# instantiate the consensus algorithm
n = 4  # decision variable dimension (n, 1)
x0 = np.random.rand(n, 1)
algorithm = Consensus(agent=agent,
                       initial_condition=x0,
                       enable_log=True)  # enable storing of the generated sequences

# run the algorithm
sequence = algorithm.run(iterations=100)

# print solution
print("Agent {0}: {1}".format(agent.id, algorithm.get_result()))

# save data
np.save("agents.npy", nproc)
np.save("agent_{0}_sequence.npy".format(agent.id), sequence)

And then execute it with the desired number of agents:

mpirun -np 12 --oversubscribe python launcher.py

where the flag --oversubscribe is necessary only if the requested number of agents (12 in this case) is higher than the available number of cores (or computing units).

**Plot the generated sequences**

In order to plot the local sequences generated by the algorithm, we create the file *results.py*.

Listing 2: results.py

```python
import numpy as np
import matplotlib.pyplot as plt

# Load the number of agents
N = np.load("agents.npy")

# Load the locally generated sequences
sequence = {}
for i in range(N):
    filename = "agent_{0}_sequence.npy".format(i)
    sequence[i] = np.load(filename)

# Plot the evolution of the local estimates
# generate N colors
colors = {}
```
for i in range(N):
    colors[i] = np.random.rand(3, 1).flatten()

# generate figure
plt.figure()
for i in range(N):
    dims = sequence[i].shape
    iterations = dims[0]
    for j in range(dims[1]):
        if j == 0:  # to avoid generating multiple labels
            plt.plot(np.arange(iterations), sequence[i][:, j, 0], color=colors[i],
                      label='agent ({}).format(i+1))
        else:
            plt.plot(np.arange(iterations), sequence[i][:, j, 0], color=colors[i])
plt.legend(loc='upper right')
plt.title("Evolution of the local estimates")
plt.xlabel(r"iterations ($k$)"")
plt.ylabel(r"$x_i^k$"")
plt.savefig('results_fig.png')
plt.show()

We execute results.py through:

python results.py
Time-varying graphs

Convergence of consensus algorithms is still achieved over time-varying (possibly directed) graphs, provided that they are jointly strongly connected.

In this case, one can use the `set_neighbors` and `set_weights` methods of the `Agent` class in order to modify the communication network when necessary. This can be done in various ways, for example by overriding the `run` method of the algorithm of by calling it multiple times over different graphs. For example, suppose that the graph changes every 100 iterations and that you want to perform 1000 iterations, then:

```python
for g in range(10):
    # generate a new common graph (everyone use the same seed)
    Adj = construct_graph(nproc, p=0.3, seed=g)
    W = metropolis_hastings(Adj)

    # set new neighbors and weights
    agent.set_neighbors(in_neighbors=np.nonzero(Adj[local_rank, :])[0].tolist(),
                        out_neighbors=np.nonzero(Adj[:, local_rank])[0].tolist())
    agent.set_weights(weights=W[local_rank, :].tolist())

algorithm.run(iterations=100)
```

Block-wise consensus

Consensus can be also performed block-wise with respect to the decision variable by using the `BlockConsensus` class.

From the perspective of agent \( i \) the algorithm works as follows. At iteration \( k \), if the agent is awake, it selects a random block \( \ell^k_i \) of its local solution and updates

\[
x_{i,\ell}^{k+1} = \begin{cases} 
\sum_{j \in N_i} w_{ij} x_j^k & \text{if } \ell = \ell^k_i \\
x_i^k & \text{otherwise}
\end{cases}
\]

where \( N_i \) is the current set of in-neighbors and \( x_j \), \( j \in N_i \) is the local copy of \( x_j \) available at node \( i \) and \( x_{i,\ell} \) denotes the \( \ell \)-th block of \( x_i \). Otherwise \( x_{i,\ell}^{k+1} = x_i^k \).

Moreover, at each iteration, each agent can update its local estimate or not at each iteration according to a certain probability (awakening_probability), thus modeling some asynchrony.

The algorithm can be instantiated by providing a list of blocks of the decision variable and the probabilities of drawing each block:

```python
algorithm = BlockConsensus(agent=agent,
                           initial_condition=x0,
                           enable_log=True,
                           blocks_list=[(0, 1), (2, 3)],
                           probabilities=[0.3, 0.7],
                           awakening_probability=0.5)
```
Asynchronous consensus

Asynchronous consensus can be seen as a sort of synchronous consensus over time-varying graphs with delays (which may model non negligible computation times and unreliable links) and it is implemented in the `AsynchronousConsensus` class.

When running this algorithm, you can control the computation and sleep times of each agent and the communication channels failure probabilities. Moreover, when running asynchronous algorithms, you have to set the total duration of the execution (and not the number of iterations). We provide the following example.

```python
import numpy as np
from mpi4py import MPI
from disropt.agents import Agent
from disropt.algorithms import AsynchronousConsensus
from disropt.utils.graph_constructor import binomial_random_graph, metropolis_hastings

# get MPI info
comm = MPI.COMM_WORLD
nproc = comm.Get_size()
local_rank = comm.Get_rank()

# Generate a common graph (everyone use the same seed)
Adj = binomial_random_graph(nproc, p=0.3, seed=1)
W = metropolis_hastings(Adj)

# reset local seed
np.random.seed()

# create local agent
agent = Agent(
in_neighbors=np.nonzero(Adj[local_rank, :])[0].tolist(),
out_neighbors=np.nonzero(Adj[:, local_rank])[0].tolist(),
in_weights=W[local_rank, :].tolist())

# instantiate the asynchronous consensus algorithm
x0 = np.random.rand(2, 1)
algorithm = AsynchronousConsensus(agent=agent,
    initial_condition=x0,
    enable_log=True,
    force_sleep=True,
    maximum_sleep=0.1,
    sleep_type="random",
    force_computation_time=True,
    maximum_computation_time=0.1,
    computation_time_type="random",
    force_unreliable_links=True,
    link_failure_probability=0.1)

# run the algorithm
timestamp_sequence_awake, timestamp_sequence_sleep, sequence = algorithm.run(running_time=4)

# print solution
print("Agent {}: ".format(agent.id, algorithm.get_result()))

# save data
np.save("agents.npy", nproc)
```

(continues on next page)

4.1. Basic examples
Plot the generated sequences

```python
import numpy as np
import matplotlib.pyplot as plt

# number of agents
N = np.load("agents.npy")

# retrieve local sequences
sequence = {}
timestamp_sequence_awake = {}
timestamp_sequence_sleep = {}
colors = {}
t_init = None
for i in range(N):
    colors[i] = np.random.rand(3, 1).flatten()
    filename = "agent_{}/sequence.npy".format(i)
    sequence[i] = np.load(filename, allow_pickle=True)

    filename = "agent_{}/timestamp_sequence_awake.npy".format(i)
    timestamp_sequence_awake[i] = np.load(filename, allow_pickle=True)

    filename = "agent_{}/timestamp_sequence_sleep.npy".format(i)
    timestamp_sequence_sleep[i] = np.load(filename, allow_pickle=True)

    if t_init is not None:
        m = min(timestamp_sequence_awake[i])
        t_init = min(t_init, m)
    else:
        t_init = min(timestamp_sequence_awake[i])

for i in range(N):
    timestamp_sequence_awake[i] = timestamp_sequence_awake[i] - t_init
    timestamp_sequence_sleep[i] = timestamp_sequence_sleep[i] - t_init

# plot
plt.figure()
for i in range(N):
    colors[i] = np.random.rand(3, 1).flatten()
    dims = sequence[i].shape
    for j in range(dims[1]):
        if j == 0:
            plt.plot(timestamp_sequence_sleep[i], sequence[i][j, :], color=colors[i],
                    label="Agent {}: awakenings={}".format(i+1, dims[0]))
        else:
            plt.plot(timestamp_sequence_sleep[i], sequence[i][j, :], color=colors[i])
```

(continues on next page)
plt.xlabel("time (s)")
plt.ylabel("x_i")
plt.title("Local estimates sequences")
plt.legend()

plt.savefig('sequences.png')

S = {}
for i in range(N):
    aw = np.array(timestamp_sequence_awake[i])
    aw = np.vstack([aw, np.zeros(aw.shape)])
    sl = np.array(timestamp_sequence_sleep[i])
    sl = np.vstack([sl, np.ones(sl.shape)])
    aux = np.hstack([aw, sl]).transpose()
    signal = aux[aux[:, 0].argsort()]
    inverse_signal = np.zeros(signal.shape)
    inverse_signal += signal
    inverse_signal[:, 1] = abs(inverse_signal[:, 1] - 1)
    ww = np.empty([signal.shape[0]*2, 2])
    ww[::2] = signal
    ww[1::2] = inverse_signal
    S[i] = ww

fig, axes = plt.subplots(int(N),1, sharex=True)
for i in range(N):
    axes[i].plot(S[i][:, 0], S[i][:, 1], color=colors[i])
    axes[i].set_ylabel("{}").format(i)

plt.xlabel("time (s)")
plt.savefig('sleep_awake.png')
plt.show()
4.1.2 Distributed Logic AND

This is an example on how to use the LogicAnd class.

Listing 3: examples/setups/logic-and/launcher.py

```python
import numpy as np
import networkx as nx
from mpi4py import MPI
from disropt.agents import Agent
from disropt.algorithms.misc import LogicAnd
from disropt.utils.graph_constructor import binomial_random_graph, metropolis_hastings

# get MPI info
comm = MPI.COMM_WORLD
nproc = comm.Get_size()
local_rank = comm.Get_rank()

# Generate a common graph (everyone use the same seed)
Adj = binomial_random_graph(nproc, p=0.1, seed=1)
W = metropolis_hastings(Adj)
graph = nx.DiGraph(Adj)
graph_diameter = nx.diameter(graph)
```

(continues on next page)
# create local agent
agent = Agent(in_neighbors=np.nonzero(Adj[local_rank, :])[0].tolist(),
              out_neighbors=np.nonzero(Adj[:, local_rank])[0].tolist(),
              in_weights=W[local_rank, :].tolist())

# instantiate the logic-and algorithm
flag = True
algorithm = LogicAnd(agent, graph_diameter, flag=flag)

algorithm.run(maximum_iterations=100)
print(algorithm.S)

The file can be executed by issuing the following command in the example folder:

```
> mpirun -np 30 --oversubscribe python launcher.py
```

### 4.1.3 Distributed Max Consensus

This is an example on how to use the `MaxConsensus` class.

Listing 4: examples/setups/max-consensus/launcher.py

```python
import numpy as np
from mpi4py import MPI
from disropt.agents import Agent
from disropt.algorithms.misc import MaxConsensus
from disropt.utils.graph_constructor import ring_graph

# get MPI info
comm = MPI.COMM_WORLD
nproc = comm.Get_size()
local_rank = comm.Get_rank()

# Generate a common graph (everyone use the same seed)
Adj = ring_graph(nproc)
graph_diam = nproc-1
n_vars = 3

# create local agent
agent = Agent(in_neighbors=np.nonzero(Adj[local_rank, :])[0].tolist(),
              out_neighbors=np.nonzero(Adj[:, local_rank])[0].tolist())

# instantiate the max-consensus algorithm
np.random.seed(local_rank*5)
x0 = np.random.rand(n_vars)*10
algorithm = MaxConsensus(agent, x0, graph_diam, True)

print('Initial value of agent {:}:\n'.format(local_rank, x0))

# execute algorithm
x_sequence = algorithm.run(iterations=100)

# get result
```

(continues on next page)
The file can be executed by issuing the following command in the example folder:

```
> mpirun -np 30 --oversubscribe python launcher.py
```

### 4.1.4 Distributed Set Membership

This is an example on how to use the `SetMembership` class. See also the reference [FaGa18].

Listing 5: examples/setups/set_membership/launcher.py

```python
import numpy as np
from mpi4py import MPI
from disropt.agents import Agent
from disropt.algorithms import SetMembership
from disropt.constraints.projection_sets import CircularSector
from disropt.utils.graph_constructor import binomial_random_graph, metropolis_hastings

# get MPI info
comm = MPI.COMM_WORLD
nproc = comm.Get_size()
local_rank = comm.Get_rank()

# Generate a common graph (everyone use the same seed)
Adj = binomial_random_graph(nproc, p=0.3, seed=1)
W = metropolis_hastings(Adj)

# reset local seed
np.random.seed(10*local_rank)

# create local agent
agent = Agent(
    in_neighbors=np.nonzero(Adj[local_rank, :])[0].tolist(),
    out_neighbors=np.nonzero(Adj[:, local_rank])[0].tolist(),
    in_weights=W[local_rank, :].tolist(),
)

# dimension of the variable
n = 2

def rotate(p, c, theta):
    # rotate p around c by theta (rad)
    xr = np.cos(theta)*(p[0]-c[0])-np.sin(theta)*(p[1]-c[1]) + c[0]
    yr = np.sin(theta)*(p[0]-c[0])+np.cos(theta)*(p[1]-c[1]) + c[1]
    return np.array([xr, yr]).reshape(p.shape)

def measure_generator():
    # define a measure generator
    np.random.seed(1)
    common_point = np.random.randn(n, 1)
```

(continues on next page)
np.random.seed(10*local_rank)

position = np.random.randn(n, 1)
eps_ang = 2*1e-1
eps_dist = 1e-1

np.random.seed()
rd = np.linalg.norm(common_point-position, 2)
rd2 = rd + eps_dist * np.random.rand()
measure = position + rd2*(common_point-position)/rd
rotation = eps_ang * (np.random.rand()-0.5)
measure = rotate(measure, position, rotation)

vect = measure - position
angle = np.arctan2(vect[1], vect[0])
radius = np.linalg.norm(vect) + eps_dist*np.random.rand()
return CircularSector(vertex=position,
                        angle=angle,
                        radius=radius,
                        width=2*eps_ang)

# Run algorithm
algorithm = SetMembership(agent, np.random.rand(n, 1), enable_log=True)
# assign measure generator to the agent
algorithm.set_measure_generator(measure_generator)
sequence = algorithm.run(iterations=1000)

# print solution
print("Agent {}: ".format(agent.id, algorithm.get_result()))

# save data
np.save("agents.npy", nproc)
np.save("agent_{}/sequence.npy".format(agent.id), sequence)
import numpy as np
import matplotlib.pyplot as plt

N = np.load("agents.npy")
sequence = {}
for i in range(N):
    filename = "agent_//sequence.npy".format(i)
    sequence[i] = np.load(filename)

plt.figure()
for i in range(N):
    dims = sequence[i].shape
    iterations = dims[0]
    for j in range(dims[1]):
        plt.plot(np.arange(iterations), sequence[i][:, j, 0])
plt.show()

The two files can be executed by issuing the following commands in the example folder:

> mpirun -np 30 --oversubscribe python launcher.py
> python results.py

### 4.1.5 Distributed Subgradient

This is an example on how to use the `SubgradientMethod` class. See also the reference [NeOz09].

```
import dill as pickle
import numpy as np
from mpi4py import MPI
from disropt.agents import Agent
from disropt.algorithms.subgradient import SubgradientMethod
from disropt.functions import QuadraticForm, Variable
from disropt.utils.graph_constructor import binomial_random_graph, metropolis_hastings
from disropt.problems import Problem

# get MPI info
comm = MPI.COMM_WORLD
nproc = comm.Get_size()
local_rank = comm.Get_rank()

# Generate a common graph (everyone use the same seed)
Adj = binomial_random_graph(nproc, p=0.3, seed=1)
W = metropolis_hastings(Adj)

# reset local seed
np.random.seed(10*local_rank)

agent = Agent(
    in_neighbors=np.nonzero(Adj[local_rank, :])[0].tolist(),
    out_neighbors=np.nonzero(Adj[:, local_rank])[0].tolist(),
```

(continues on next page)
in_weights=W[local_rank, :].tolist()

# variable dimension
n = 2

# declare a variable
x = Variable(n)

# define the local objective function
P = np.random.rand(n, n)
P = P.transpose() @ P
bias = np.random.rand(n, 1)
fn = QuadraticForm(x - bias, P)

# define a (common) constraint set
constr = [x<= 1, x >= -1]

# local problem
pb = Problem(fn, constr)
agent.set_problem(pb)

# instantiate the algorithms
initial_condition = 10*np.random.rand(n, 1)
algorithm = SubgradientMethod(agent=agent,
                               initial_condition=initial_condition,
                               enable_log=True)

def step_gen(k):  # define a stepsize generator
    return 1/((k+1)**0.51)

# run the algorithm
sequence = algorithm.run(iterations=1000, stepsize=step_gen)
print("Agent {}").format(agent.id, algorithm.get_result().flatten())

np.save("agents.npy", nproc)

# save agent and sequence
np.save("agent_{id}_sequence.npy".format(agent.id), sequence)
with open('agent_{id}_function.pkl'.format(agent.id), 'wb') as output:
    pickle.dump(fn, output, pickle.HIGHEST_PROTOCOL)

Listing 8: examples/setups/distributed_subgradient/results.py

import numpy as np
import matplotlib.pyplot as plt
import pickle

N = np.load("agents.npy")
n = 2

sequence = {}
local_function = {}
for i in range(N):
    filename = "agent_{}/sequence.npy".format(i)
    sequence[i] = np.load(filename)
    with open('agent_{}/function.pkl'.format(i), 'rb') as inp:
        local_function[i] = pickle.load(inp)

plt.figure()
colors = {}
for i in range(N):
    colors[i] = np.random.rand(3, 1).flatten()
    dims = sequence[i].shape
    print(dims)
    iterations = dims[0]
    for j in range(dims[1]):
        plt.plot(np.arange(iterations), sequence[i][:, j, 0], color=colors[i])

function = np.zeros([iterations, 1])
for k in range(iterations):
    avg = np.zeros([n, 1])
    for i in range(N):
        avg += sequence[i][k, :, 0].reshape(n, 1)
    avg = avg/N
    for i in range(N):
        function[k] += local_function[i].eval(avg).flatten()

plt.figure()
plt.semilogy(function)
plt.show()

The two files can be executed by issuing the following commands in the example folder:

> mpirun -np 30 --oversubscribe python launcher.py
> python results.py

4.1.6 Gradient Tracking

This is an example on how to use the GradientTracking class.

Listing 9: examples/setupgradient_tracking/launcher.py

import dill as pickle
import numpy as np
from mpi4py import MPI
from disropt.agents import Agent
from disropt.algorithms.gradient_tracking import GradientTracking
from disropt.functions import QuadraticForm, Variable
from disropt.utils.utilities import is_pos_def
from disropt.utils.graph_constructor import binomial_random_graph, metropolis_hastings
from disropt.problems.problem import Problem

# get MPI info
comm = MPI.COMM_WORLD
nproc = comm.Get_size()
local_rank = comm.Get_rank()
# Generate a common graph (everyone use the same seed)
Adj = binomial_random_graph(nproc, p=0.3, seed=1)
W = metropolis_hastings(Adj)

# reset local seed
np.random.seed()

agent = Agent(
in_neighbors=np.nonzero(Adj[local_rank, :])[0].tolist(),
out_neighbors=np.nonzero(Adj[:, local_rank])[0].tolist(),
in_weights=W[local_rank, :].tolist())

# variable dimension
d = 4

# generate a positive definite matrix
P = np.random.randn(d, d)
while not is_pos_def(P):
    P = np.random.randn(d, d)
bias = np.random.randn(d, 1)
# declare a variable
x = Variable(d)

# define the local objective function
fun = QuadraticForm(x - bias, P)

# local problem
pb = Problem(fun)
agent.set_problem(pb)

# instantiate the algorithms
initial_condition = np.random.rand(d, 1)
algorithm = GradientTracking(agent=agent,
    initial_condition=initial_condition,
    enable_log=True)

# run the algorithm
sequence = algorithm.run(iterations=1000, stepsize=0.001)
print("Agent {}:")

np.save("agents.npy", nproc)

# save agent and sequence
with open('agent_{}/function.pkl'.format(agent.id), 'wb') as output:
    pickle.dump(agent.problem.objective_function, output, pickle.HIGHEST_PROTOCOL)
np.save("agent_{}/sequence.npy".format(agent.id), sequence)

Listing 10: examples/setups/gradient_tracking/results.py

import numpy as np
import matplotlib.pyplot as plt
import pickle
N = np.load("agents.npy")
d = 4

sequence = {} local_function = {}
for i in range(N):
    filename = "agent_{}/sequence.npy".format(i)
    sequence[i] = np.load(filename)
    with open('agent_{}/function.pkl'.format(i), 'rb') as input:
        local_function[i] = pickle.load(input)

plt.figure()
colors = {}
for i in range(N):
    colors[i] = np.random.rand(3, 1).flatten()
dims = sequence[i].shape
print(dims)
iterations = dims[0]
for j in range(dims[1]):
    plt.plot(np.arange(iterations), sequence[i][:, j, 0], color=colors[i])

function = np.zeros([iterations, 1])
for k in range(iterations):
    avg = np.zeros([d, 1])
    for i in range(N):
        avg += sequence[i][k, :, 0].reshape(d, 1)
    avg = avg/N
    for i in range(N):
        function[k] += local_function[i].eval(avg).flatten()

plt.figure()
plt.plot(function)
plt.show()

The two files can be executed by issuing the following commands in the example folder:

> mpirun -np 30 --oversubscribe python launcher.py
> python results.py

### 4.1.7 Distributed Dual Decomposition

This is an example on how to use the `DualDecomposition` class.

Listing 11: examples/setups/distributed_dual_decomposition/launcher.py

```python
import dill as pickle
import numpy as np
from mpi4py import MPI
from disropt.agents import Agent
from disropt.algorithms.dual_decomp import DualDecomposition
from disropt.functions import QuadraticForm, Variable
from disropt.utils.graph_constructor import binomial_random_graph
from disropt.problems import Problem
```

4.1. Basic examples
# get MPI info
comm = MPI.COMM_WORLD
nproc = comm.Get_size()
local_rank = comm.Get_rank()

# Generate a common graph (everyone use the same seed)
Adj = binomial_random_graph(nproc, p=0.3, seed=1)

# reset local seed
np.random.seed(10*local_rank)

agent = Agent(
    in_neighbors=np.nonzero(Adj[local_rank, :])[0].tolist(),
    out_neighbors=np.nonzero(Adj[:, local_rank])[0].tolist())

# variable dimension
n = 2

# generate a positive definite matrix
P = np.random.randn(n, n)
P = P.transpose() @ P
bias = 3*np.random.randn(n, 1)

# declare a variable
x = Variable(n)

# define the local objective function
fn = QuadraticForm(x - bias, P)

# define a (common) constraint set
constr = [np.eye(n) @ x <= 1, np.eye(n) @ x >= -1]

# local problem
pb = Problem(fn, constr)
agent.set_problem(pb)

# instantiate the algorithms
initial_condition = {}

for j in agent.in_neighbors:
    initial_condition[j] = 10*np.random.rand(n, 1)

algorithm = DualDecomposition(agent=agent,
                               initial_condition=initial_condition,
                               enable_log=True)

def step_gen(k):
    # define a stepsize generator
    return 1/((k+1)**0.51)

# run the algorithm
x_sequence, lambda_sequence = algorithm.run(iterations=100, stepsize=step_gen)
x_t, lambda_t = algorithm.get_result()
print("Agent {}: primal {} dual {}".format(agent.id, x_t.flatten(), lambda_t))

np.save("agents.npy", nproc)
with open('agent_{}/function.pkl'.format(agent.id), 'wb') as output:
    pickle.dump(agent.problem.objective_function, output, pickle.HIGHEST_PROTOCOL)
with open('agent_{}/dual_sequence.pkl'.format(agent.id), 'wb') as output:
    pickle.dump(lambda_sequence, output, pickle.HIGHEST_PROTOCOL)
np.save("agent_{}/primal_sequence.npy".format(agent.id), x_sequence)

Listing 12: examples/setups/distributed_dual_decomposition/results.py

import dill as pickle
import numpy as np
import matplotlib.pyplot as plt

N = np.load("agents.npy")
n = 2

lambda_sequence = {}
x_sequence = {}
local_obj_function = {}
for i in range(N):
    with open('agent_{}/dual_sequence.pkl'.format(i), 'rb') as input:
        lambda_sequence[i] = pickle.load(input)
    x_sequence[i] = np.load("agent_{}/primal_sequence.npy".format(i))
    with open('agent_{}/function.pkl'.format(i), 'rb') as input:
        local_obj_function[i] = pickle.load(input)

    # plot sequence of x
    plt.figure()
    plt.title("Primal sequences")
    colors = {}
    for i in range(N):
        colors[i] = np.random.rand(3, 1).flatten()
        dims = x_sequence[i].shape
        print(dims)
        iterations = dims[0]
        for j in range(dims[1]):
            plt.plot(np.arange(iterations), x_sequence[i][j, 0], color=colors[i])

    # plot primal cost
    plt.figure()
    plt.title("Primal cost")
    function = np.zeros([iterations, 1])
    for k in range(iterations):
        avg = np.zeros([n, 1])
        for i in range(N):
            avg += x_sequence[i][k, :].reshape(n, 1)
        avg = avg/N
        for i in range(N):
            function[k] += local_obj_function[i].eval(avg).flatten()
    plt.plot(np.arange(iterations), function)
plt.show()
The two files can be executed by issuing the following commands in the example folder:

```bash
> mpirun -np 30 --oversubscribe python launcher.py
> python results.py
```

### 4.1.8 Distributed ADMM

This is an example on how to use the ADMM class.

Listing 13: examples/setups/distributed_ADMM/launcher.py

```python
import dill as pickle
import numpy as np
from mpi4py import MPI
from disropt.agents import Agent
from disropt.algorithms.admm import ADMM
from disropt.functions import QuadraticForm, Variable
from disropt.utils.graph_constructor import binomial_random_graph
from disropt.problems import Problem

# get MPI info
comm = MPI.COMM_WORLD
nproc = comm.Get_size()
local_rank = comm.Get_rank()

# Generate a common graph (everyone use the same seed)
Adj = binomial_random_graph(nproc, p=0.3, seed=1)

# reset local seed
np.random.seed(10*local_rank)
agent = Agent(
    in_neighbors=np.nonzero(Adj[local_rank, :])[0].tolist(),
    out_neighbors=np.nonzero(Adj[:, local_rank])[0].tolist())

# variable dimension
n = 2

# generate a positive definite matrix
P = np.random.randn(n, n)
P = P.transpose() @ P
bias = 3*np.random.randn(n, 1)

# declare a variable
x = Variable(n)

# define the local objective function
fn = QuadraticForm(x - bias, P)

# define a (common) constraint set
constr = [x <= 1, x >= -1]

# local problem
pb = Problem(fn, constr)
agent.set_problem(pb)
```

(continues on next page)
# instantiate the algorithms
initial_z = np.ones((n, 1))
initial_lambda = {local_rank: 10*np.random.rand(n, 1)}
initial_lambda = {local_rank: np.ones((n, 1))}

for j in agent.in_neighbors:
    initial_lambda[j] = 10*np.random.rand(n, 1)
initial_lambda[j] = np.ones((n, 1))

algorithm = ADMM(agent=agent,
                 initial_lambda=initial_lambda,
                 initial_z=initial_z,
                 enable_log=True)

# run the algorithm
x_sequence, lambda_sequence, z_sequence = algorithm.run(iterations=100, penalty=0.1,
                 verbose=True)
x_t, lambda_t, z_t = algorithm.get_result()

print("Agent {}:
primal {} dual {}
auxiliary primal {}").format(agent.id, x_t.flatten(), lambda_t, z_t.flatten())

np.save("agents.npy", nproc)

# save agent and sequence
if local_rank == 0:
    with open('constraints.pkl', 'wb') as output:
        pickle.dump(constr, output, pickle.HIGHEST_PROTOCOL)
    with open('agent_{0}_function.pkl'.format(agent.id), 'wb') as output:
        pickle.dump(agent.problem.objective_function, output, pickle.HIGHEST_PROTOCOL)
    with open('agent_{0}_dual_sequence.pkl'.format(agent.id), 'wb') as output:
        pickle.dump(lambda_sequence, output, pickle.HIGHEST_PROTOCOL)
    np.save("agent_{0}_primal_sequence.npy".format(agent.id), x_sequence)
    np.save("agent_{0}_auxiliary_primal_sequence.npy".format(agent.id), z_sequence)

Listing 14: examples/setups/distributed_ADMM/results.py

import dill as pickle
import numpy as np
import matplotlib.pyplot as plt
from disropt.problems import Problem

N = np.load("agents.npy")
n = 2

lambda_sequence = {}
x_sequence = {}
z_sequence = {}
local_obj_function = {}

for i in range(N):
    with open('agent_{0}_dual_sequence.pkl'.format(i), 'rb') as input:
        lambda_sequence[i] = pickle.load(input)
x_sequence[i] = np.load("agent_{0}_primal_sequence.npy".format(i))
z_sequence[i] = np.load("agent_{0}_auxiliary_primal_sequence.npy".format(i))

    with open('agent_{0}_function.pkl'.format(i), 'rb') as input:
        local_obj_function[i] = pickle.load(input)

    with open('constraints.pkl', 'rb') as input:

(continues on next page)
constr = pickle.load(input)
iters = x_sequence[0].shape[0]

# solve centralized problem
global_obj_func = 0
for i in range(N):
    global_obj_func += local_obj_function[i]

global_pb = Problem(global_obj_func, constr)
x_centr = global_pb.solve()

# compute cost errors
cost_err = np.zeros((N, iters)) - cost_centr
for i in range(N):
    for t in range(iters):
        cost_err[i, t] += local_obj_function[i].eval(x_sequence[i][t, :])

# plot maximum cost error
plt.figure()
plt.title('Maximum cost error (among agents)')
plt.xlabel(r'iteration $k$')
plt.ylabel(r'$\max_i \left| \sum_{j=1}^N f_j(x_i^k) - f^\star \right|$')
plt.semilogy(np.arange(iters), np.amax(np.abs(cost_err), axis=0))

# plot maximum solution error
sol_err = np.zeros((N, iters))
for i in range(N):
    sol_err[i] = np.linalg.norm(np.squeeze(x_sequence[i]) - x_centr[None, :], axis=1)

plt.figure()
plt.title('Maximum solution error (among agents)')
plt.xlabel(r'iteration $k$')
plt.ylabel(r'$\max_i \|x_i^k - x^\star\|$')
plt.semilogy(np.arange(iters), np.amax(sol_err, axis=0))
plt.show()
### 4.1.9 ASYM\textsuperscript{M}

This is an example on how to use the ASYM class. See also the reference [FaGa19b].

Listing 15: examples/setups/asymm/launcher.py

```python
import numpy as np
import networkx as nx
import dill as pickle
from mpi4py import MPI
from disropt.agents import Agent
from disropt.algorithms.asymm import ASYM
from disropt.utils.graph_constructor import binomial_random_graph, metropolis_hastings
from disropt.functions import SquaredNorm, Norm, Variable
from disropt.problems import Problem

# get MPI info
comm = MPI.COMM_WORLD
nproc = comm.Get_size()
local_rank = comm.Get_rank()

# Generate a common graph (everyone use the same seed)
Adj = binomial_random_graph(nproc, p=1, seed=1)
W = metropolis_hastings(Adj)
graph = nx.DiGraph(Adj)
graph_diameter = nx.diameter(graph)

if local_rank == 0:
    print(Adj)

# create local agent
agent = Agent(in_neighbors=np.nonzero(Adj[local_rank, :])[0].tolist(),
              out_neighbors=np.nonzero(Adj[:, local_rank])[0].tolist(),
              in_weights=W[local_rank, :].tolist())

# problem set-up
n = 2

# target point
x_true = np.random.randn(n, 1)

if local_rank == 0:
    print(x_true)

# reset local seed
np.random.seed(10 * local_rank)

# local position
c = np.random.randn(n, 1)

# true distance
distance = np.linalg.norm(x_true-c, ord=2)

# declare a variable
x = Variable(n)

# define the local objective function
objective = (x - c) @ (x - c)
```

(continues on next page)
# local constraint
upper_bound = (x-c) @ (x-c) <= (distance**2 + 0.00001*np.random.rand())
lower_bound = (x-c) @ (x-c) >= (distance**2 - 0.00001*np.random.rand())

constraints = [upper_bound, lower_bound]

# define local problem
pb = Problem(objective_function=objective, constraints=constraints)
agent.set_problem(pb)

###
x0 = np.random.randn(n, 1)
algorith = ASYMM(agent=agent,
                graph_diameter=graph_diameter,
                initial_condition=x0,
                enable_log=True)

timestamp_sequence_awake, timestamp_sequence_sleep, sequence = algorithm.run(running_time=30.0)

# print solution
print("Agent {}
      value: {}").format(agent.id, algorithm.get_result()))

# save data
with open('agent_{}.constraints.pkl'.format(agent.id), 'wb') as output:
    pickle.dump(agent.problem.constraints, output, pickle.HIGHEST_PROTOCOL)
np.save("agent_{}.sequence.npy".format(agent.id), sequence)
np.save("agent_{}.timestamp_sequence_awake.npy".format(agent.id), timestamp_sequence_awake)
np.save("agent_{}.timestamp_sequence_sleep.npy".format(agent.id), timestamp_sequence_sleep)

Listing 16: examples/setups/asymm/results.py

```python
import numpy as np
import dill as pickle
import matplotlib.pyplot as plt

# number of agents
N = np.load("agents.npy")

# retrieve local sequences
sequence = {}
constraints = {}
timestamp_sequence_awake = {}
timestamp_sequence_sleep = {}
colors = {}
t_init = None
for i in range(N):
    colors[i] = np.random.rand(3, 1).flatten()
    filename = "agent_{}.sequence.npy".format(i)
    sequence[i] = np.load(filename, allow_pickle=True)
    filename = "agent_{}.timestamp_sequence_awake.npy".format(i)
    timestamp_sequence_awake[i] = np.load(filename)
    filename = "agent_{}.timestamp_sequence_sleep.npy".format(i)
    timestamp_sequence_sleep[i] = np.load(filename)
```

(continues on next page)
timestamp_sequence_awake[i] = np.load(filename, allow_pickle=True)
filename = "agent_/{}/timestamp_sequence_sleep.npy".format(i)
timestamp_sequence_sleep[i] = np.load(filename, allow_pickle=True)

if t_init is not None:
    m = min(timestamp_sequence_awake[i])
    t_init = min(t_init, m)
else:
    t_init = min(timestamp_sequence_awake[i])

with open('agent_/{}/constraints.pkl'.format(i), 'rb') as input:
    constraints[i] = pickle.load(input)

for i in range(N):
    timestamp_sequence_awake[i] = timestamp_sequence_awake[i] - t_init
    timestamp_sequence_sleep[i] = timestamp_sequence_sleep[i] - t_init

# plot
plt.figure()
for i in range(N):
    dims = sequence[i].shape
    for j in range(dims[1]):
        for m in range(dims[2]):
            plt.plot(timestamp_sequence_sleep[i], sequence[i][j, m])
plt.ylim([-5, 5])

# # plt
plt.figure()
for i in range(N):
    dims = sequence[i].shape
    iterations = dims[0]
    print("Agent {}, iterations {}".format(i, iterations))
    feasibility = np.zeros([iterations, 2])
    for k in range(iterations):
        for idx, constr in enumerate(constraints[i]):
            flag = constr.eval((sequence[i][k, :, :]).reshape(2,1))
            if flag == False:
                feasibility[k, idx] += abs(constr.function.eval((sequence[i][k, :, :]).reshape(2,1))).flatten()
        plt.semilogy(timestamp_sequence_sleep[i], feasibility)
    # plt.xlim([-1, 15])

plt.show()

The two files can be executed by issuing the following commands in the example folder:

> mpirun -np 30 --oversubscribe python launcher.py
> python results.py
4.1.10 Constraints Consensus

This is an example on how to use the ConstraintsConsensus class. See also the reference [NoBu11].

Listing 17: examples/setups/constraints_consensus/launcher.py

```python
import dill as pickle
import numpy as np
from mpi4py import MPI
from disropt.agents import Agent
from disropt.algorithms import ConstraintsConsensus
from disropt.functions import Variable, QuadraticForm
from disropt.problems import Problem

# get MPI info
NN = MPI.COMM_WORLD.Get_size()
agent_id = MPI.COMM_WORLD.Get_rank()

# Generate a common graph (everyone uses the same seed)
Adj = binomial_random_graph(NN, p=0.03, seed=1)

# Problem parameters
np.random.seed(10*agent_id)

# linear objective function
dim = 2
z = Variable(dim)
c = np.ones([dim,1])
obj_func = c @ z

# constraints are circles of the form (z-p)^\top (z-p) <= 1
# equivalently z^\top z - 2(A p)^\top z + p^\top A p <= 1
I = np.eye(dim)
p = np.random.rand(dim,1)
r = 1 # unitary radius
constr = []
ff = QuadraticForm(z,I,- 2*(I @ p),(p.transpose() @ I @ p) - r**2)
constr.append(ff<= 0)

# local agent and problem
agent = Agent(
    in_neighbors=np.nonzero(Adj[agent_id, :])[0].tolist(),
    out_neighbors=np.nonzero(Adj[:, agent_id])[0].tolist())
pb = Problem(obj_func, constr)
agent.set_problem(pb)

# instantiate the algorithm
algorithm = ConstraintsConsensus(agent=agent,
    enable_log=True)

(continues on next page)
```
# run the algorithm
n_iter = NN*3
x_sequence = algorithm.run(iterations=n_iter, verbose=True)

# print results
x_final = algorithm.get_result()
print("Agent {}: {}".format(agent_id, x_final.flatten()))

# save results to file
if agent_id == 0:
    with open('info.pkl', 'wb') as output:
        pickle.dump({'N': NN, 'size': dim, 'iterations': n_iter}, output, pickle.HIGHEST_PROTOCOL)
    with open('objective_function.pkl', 'wb') as output:
        pickle.dump(obj_func, output, pickle.HIGHEST_PROTOCOL)
    with open('agent_{}/constr.pkl'.format(agent_id), 'wb') as output:
        pickle.dump(constr, output, pickle.HIGHEST_PROTOCOL)
    np.save("agent_{}/seq.npy".format(agent_id), x_sequence)

import numpy as np
import matplotlib.pyplot as plt
from disropt.problems import Problem
import pickle
import tikzplotlib

# initialize
with open('info.pkl', 'rb') as inp:
    info = pickle.load(inp)
NN = info['N']
iters = info['iterations']
size = info['size']
# load agent data
sequence = np.zeros((NN, iters, size))
local_constr = {}
for i in range(NN):
    sequence[i, :, :] = np.load("agent_{}/seq.npy".format(i), allow_pickle=True).reshape((iters, size))
    with open('agent_{}/constr.pkl'.format(i), 'rb') as inp:
        local_constr[i] = pickle.load(inp)
with open('objective_function.pkl', 'rb') as inp:
    obj_func = pickle.load(inp)

# solve centralized problem
global_constr = []
for i in range(NN):
    global_constr.extend(local_constr[i])
global_pb = Problem(obj_func, global_constr)
x_centr = global_pb.solve()
cost_centr = obj_func.eval(x_centr)

# compute cost errors
cost_err = np.zeros((NN, iters))
for i in range(NN):
    for t in range(iters):
        cost_err[i, t] = abs(obj_func.eval(sequence[i, t, :].reshape((size, 1))) - cost_centr)

# compute max violation
vio_err = np.zeros((NN, iters))
for i in range(NN):
    for t in range(iters):
        xt = sequence[i, t, :].reshape((size, 1))
        max_err = np.zeros((len(global_constr), 1))
        for c in range(len(global_constr)):
            max_err[c] = global_constr[c].function.eval(xt)
        vio_err[i, t] = np.max(max_err)

# Plot the evolution of the local estimates
# generate N colors
colors = {}
for i in range(NN):
    colors[i] = np.random.rand(3, 1).flatten()

# plot cost error
plt.figure()
plt.title('Evolution of cost error')
plt.xlabel(r"iteration $k$")
plt.ylabel(r"$|f(x_i^k) - f^\star|$")
for i in range(NN):
    plt.plot(np.arange(iters), cost_err[i, :], color=colors[i])

# plot violation error
plt.figure()
plt.title('Maximum constraint violation')
plt.xlabel(r"iteration $k$")
plt.ylabel(r"$\max_j g(x_i^k)$")
for i in range(NN):
    plt.plot(np.arange(iters), vio_err[i, :], color=colors[i])
plt.show()

The two files can be executed by issuing the following commands in the example folder:

> mpirun -np 30 --oversubscribe python launcher.py
> python results.py
4.1.11 Distributed Simplex

This is an example on how to use the DistributedSimplex class. See also the reference [BuNo11].

Listing 19: examples/setups/distributed_simplex/launcher.py

```python
import dill as pickle
import numpy as np
from mpi4py import MPI
from disropt.agents import Agent
from disropt.algorithms.constraintexchange import DistributedSimplex
from disropt.functions import Variable
from disropt.utils.graph_constructor import ring_graph
from disropt.utils.LP_utils import generate_LP
from disropt.problems import LinearProblem

# get MPI info
comm = MPI.COMM_WORLD
nproc = comm.Get_size()
local_rank = comm.Get_rank()

# Generate a ring graph (for which the diameter is nproc-1)
Adj = ring_graph(nproc)
graph_diam = nproc-1

# reset local seed
np.random.seed(1)

# number of constraints
n_constr = 3

# number of columns for each processor
k = 2

# generate a feasible optimization problem of size k * nproc
c_glob, A_glob, b_glob, x_glob = generate_LP(k * nproc, n_constr, 50, constr_form='eq →')

# extract the columns assigned to this agent
local_indices = list(np.arange(k*local_rank, k*(local_rank+1)))
c_loc = c_glob[local_indices, :]
A_loc = A_glob[:, local_indices]
b_loc = b_glob

# define the local problem data
x = Variable(k)
obj = c_loc @ x
constr = A_loc.transpose() @ x == b_loc
problem = LinearProblem(objective_function=obj, constraints=constr)

# create agent
agent = Agent(in_neighbors=np.nonzero(Adj[local_rank, :])[0].tolist(),
              out_neighbors=np.nonzero(Adj[:, local_rank])[0].tolist())
agent.problem = problem

# instantiate the algorithm
algorithm = DistributedSimplex(agent, enable_log=True, problem_size=nproc*k,
```

(continues on next page)
# run the algorithm
x_sequence, J_sequence = algorithm.run(iterations=100, verbose=True)

# print results
_, _, _, J_final = algorithm.get_result()
print("Agent {} - {} iterations - final cost {}".format(agent.id, len(J_sequence), J_final))

# save results to file
if agent.id == 0:
    with open('info.pkl', 'wb') as output:
        pickle.dump({'N': nproc, 'n_vars': k * nproc, 'n_constr': n_constr, 'c': c_glob, 'A': A_glob, 'b': b_glob, 'opt_sol': x_glob}, output, pickle.HIGHEST_PROTOCOL)

np.save("agent_{}/x_seq.npy".format(agent.id), x_sequence)
np.save("agent_{}/J_seq.npy".format(agent.id), J_sequence)

Listing 20: examples/setups/distributed_simplex/results.py

import numpy as np
import matplotlib.pyplot as plt
import dill as pickle
from disropt.functions import Variable
from disropt.problems import LinearProblem

# initialize
with open('info.pkl', 'rb') as inp:
    info = pickle.load(inp)
NN = info['N']
c = info['c']
opt_sol = info['opt_sol']

# load agent data
sequence_J = {}
for i in range(NN):
    sequence_J[i] = np.load("agent_{}/J_seq.npy".format(i))

# compute optimal cost
opt_cost = (c.transpose() @ opt_sol).flatten()

# plot cost evolution
plt.figure()
plt.title("Cost evolution")
colors = np.random.rand(NN+1, 3)
max_iters = 0
for i in range(NN):
    seq_J_i = sequence_J[i]
    n_iters_i = len(seq_J_i)
    max_iters = max(max_iters, n_iters_i)
    plt.plot(np.arange(n_iters_i), seq_J_i, color=colors[i])

(continues on next page)
# plot optimal cost
plt.plot(np.arange(max_iters), np.ones(max_iters)*opt_cost, '--', color=colors[NN])
plt.show()

The two files can be executed by issuing the following commands in the example folder:

```bash
> mpirun -np 10 --oversubscribe python launcher.py
> python results.py
```

## 4.1.12 Distributed Simplex (dual problem)

This is an example on how to use the `DualDistributedSimplex` class, which forms the dual problem of the given optimization problem and solves it with the Distributed Simplex algorithm.

Listing 21: examples/setups/distributed_simplex/launcher_dual.py

```python
import dill as pickle
import numpy as np
from mpi4py import MPI
from disropt.agents import Agent
from disropt.algorithms.constraintexchange import DualDistributedSimplex
from disropt.functions import Variable
from disropt.utils.graph_constructor import ring_graph
from disropt.utils.LP_utils import generate_LP
from disropt.problems import LinearProblem

# get MPI info
comm = MPI.COMM_WORLD
nproc = comm.Get_size()
local_rank = comm.Get_rank()

# Generate a ring graph (for which the diameter is nproc-1)
Adj = ring_graph(nproc)
graph_diam = nproc-1

# reset local seed
np.random.seed(1)

# number of variables
n_vars = 3

# number of constraints for each processor
k = 2

# generate a feasible optimization problem with k * nproc constraints
c_glob, A_glob, b_glob, x_glob = generate_LP(n_vars, k * nproc, 50, direction='max')

# extract the constraints assigned to this agent
local_indices = list(np.arange(k*local_rank, k*(local_rank+1)))

c_loc = c_glob
A_loc = A_glob[local_indices, :]
b_loc = b_glob[local_indices, :]

# define the local problem data
```

(continues on next page)
import numpy as np
import matplotlib.pyplot as plt
import dill as pickle
from disropt.functions import Variable
from disropt.problems import LinearProblem

# initialize
with open('info.pkl', 'rb') as inp:
    info = pickle.load(inp)
    NN = info['N']
    c = info['c']
    opt_sol = info['opt_sol']

# load agent data
sequence_J = {}
for i in range(NN):
    sequence_J[i] = np.load("agent_{}/J_seq.npy".format(i))

# compute optimal cost
opt_cost = (c.transpose() @ opt_sol).flatten()

# plot cost evolution
```python
plt.figure()
plt.title("Cost evolution")
colors = np.random.rand(NN+1, 3)
max_iters = 0

for i in range(NN):
    seq_J_i = sequence_J[i]
    n_iters_i = len(seq_J_i)
    max_iters = max(max_iters, n_iters_i)
    plt.plot(np.arange(n_iters_i), seq_J_i, color=colors[i])

# plot optimal cost
plt.plot(np.arange(max_iters), np.ones(max_iters)*opt_cost, '--', color=colors[NN])
plt.show()
```

The two files can be executed by issuing the following commands in the example folder:

```bash
> mpirun -np 10 --oversubscribe python launcher_dual.py
> python results_dual.py
```

### 4.1.13 Distributed Dual Subgradient

**Warning:** This example is currently under development

This is an example on how to use the `DualSubgradientMethod` class. See also the reference [FaMa17].

Listing 23: examples/setups/distributed_dual_subgradient/launcher.py

```python
# WARNING: this file is currently under development

import dill as pickle
import numpy as np
from mpi4py import MPI
from disropt.agents import Agent
from disropt.algorithms.dual_subgradient import DualSubgradientMethod
from disropt.functions import QuadraticForm, Variable, AffineForm
from disropt.utils.utilities import is_pos_def
from disropt.constraints.projection_sets import Box
from disropt.problems.constraint_coupled_problem import binomial_random_graph, metropolis_hastings
from disropt.problems.constraint_coupled_problem import ConstraintCoupledProblem

# get MPI info
comm = MPI.COMM_WORLD
nproc = comm.Get_size()
local_rank = comm.Get_rank()

# Generate a common graph (everyone use the same seed)
Adj = binomial_random_graph(nproc, p=0.3, seed=1)
W = metropolis_hastings(Adj)

# reset local seed
np.random.seed()
```

(continues on next page)
agent = Agent(
    
in_neighbors=np.nonzero(Adj[local_rank, :])[0].tolist(),
    out_neighbors=np.nonzero(Adj[:, local_rank])[0].tolist(),
    in_weights=W[local_rank, :].tolist())

# local variable dimension - random in [2,5]
n_i = np.random.randint(2, 6)

# number of coupling constraints
S = 3

# generate a positive definite matrix
P = np.random.randn(n_i, n_i)
while not is_pos_def(P):
    P = np.random.randn(n_i, n_i)
bias = np.random.randn(n_i, 1)

# declare a variable
x = Variable(n_i)

# define the local objective function
fn = QuadraticForm(x - bias, P)

# define the local constraint set
constr = [x>=-2, x<=2]

# define the local contribution to the coupling constraints
A = np.random.randn(S, n_i)
coupling_fn = A.transpose() @ x

# create local problem and assign to agent
pb = ConstraintCoupledProblem(objective_function=fn,
                               constraints=constr,
                               coupling_function=coupling_fn)
agent.set_problem(pb)

# initialize the dual variable
lambda0 = np.random.rand(S, 1)
xhat0 = np.zeros((n_i, 1))

algorithm = DualSubgradientMethod(agent=agent,
                                   initial_condition=lambda0,
                                   initial_runavg=xhat0,
                                   enable_log=True)

def step_gen(k): # define a stepsize generator
    return 0.1/np.sqrt(k+1)

# run the algorithm
lambda_sequence, xhat_sequence = algorithm.run(iterations=1000, stepsize=step_gen)
lambda_t, xhat_t = algorithm.get_result()
print("Agent {}: dual {} primal {}".format(agent.id, lambda_t.flatten(), xhat_t.flatten()))
np.save("agents.npy", nproc)

(continues from previous page)
# save agent and sequence

```python
with open('agent_{}/obj_function.pkl'.format(agent.id), 'wb') as output:
    pickle.dump(agent.problem.objective_function, output, pickle.HIGHEST_PROTOCOL)
with open('agent_{}/coup_function.pkl'.format(agent.id), 'wb') as output:
    pickle.dump(agent.problem.coupling_function, output, pickle.HIGHEST_PROTOCOL)
np.save("agent_{}/dual_sequence.npy".format(agent.id), lambda_sequence)
np.save("agent_{}/runavg_sequence.npy".format(agent.id), xhat_sequence)
```

---

Listing 24: examples/setups/distributed_dual_subgradient/results.py

```python
import numpy as np
import matplotlib.pyplot as plt
import pickle
N = np.load("agents.npy")
S = 3
lambda_sequence = {}
xhat_sequence = {}
local_obj_function = {}
local_coup_function = {}
for i in range(N):
    lambda_sequence[i] = np.load("agent_{}/dual_sequence.npy".format(i))
xhat_sequence[i] = np.load("agent_{}/runavg_sequence.npy".format(i))
    with open('agent_{}/obj_function.pkl'.format(i), 'rb') as input:
        local_obj_function[i] = pickle.load(input)
    with open('agent_{}/coup_function.pkl'.format(i), 'rb') as input:
        local_coup_function[i] = pickle.load(input)

# plot dual solutions
plt.figure()
plt.title("Dual solutions")
colors = {}
for i in range(N):
    colors[i] = np.random.rand(3, 1).flatten()
    dims = lambda_sequence[i].shape
    iterations = dims[0]
    for j in range(dims[1]):
        plt.plot(np.arange(iterations), lambda_sequence[i][..., j, 0], color=colors[i])

# plot cost of running average
plt.figure()
plt.title("Primal cost (running average)")
obj_function = np.zeros([iterations, 1])
for k in range(iterations):
    for i in range(N):
        obj_function[k] += local_obj_function[i].eval(xhat_sequence[i][k, :, 0].
        reshape(-1,1)).flatten()
plt.plot(obj_function)

# plot coupling constraint utilization
plt.figure()
plt.title("Coupling constraint utilization (running average)")
```

---

4.1. Basic examples
coup_function = np.zeros([iterations, S])
for k in range(iterations):
    for i in range(N):
        coup_function[k] += local_coup_function[i].eval(xhat_sequence[i][k, :, 0].reshape(-1,1)).flatten()

plt.plot(coup_function)
plt.show()

The two files can be executed by issuing the following commands in the example folder:

```
> mpirun -np 30 --oversubscribe python launcher.py
> python results.py
```

### 4.1.14 Distributed Primal Decomposition

This is an example on how to use the `PrimalDecomposition` class.

Listing 25: examples/setups/distributed_primal_decomposition/launcher.py

```python
import dill as pickle
import numpy as np
from mpi4py import MPI
from disropt.agents import Agent
from disropt.algorithms.primal_decomp import PrimalDecomposition
from disropt.functions import QuadraticForm, Variable
from disropt.utils.utilities import is_pos_def
from disropt.constraints.projection_sets import Box
from disropt.utils.graph_constructor import binomial_random_graph
from disropt.problems.constraint_coupled_problem import ConstraintCoupledProblem

# get MPI info
comm = MPI.COMM_WORLD
nproc = comm.Get_size()
local_rank = comm.Get_rank()

# Generate a common graph (everyone use the same seed)
Adj = binomial_random_graph(nproc, p=0.3, seed=1)

# reset local seed
np.random.seed()

agent = Agent(
in_neighbors=np.nonzero(Adj[local_rank, :])[0].tolist(),
out_neighbors=np.nonzero(Adj[:, local_rank])[0].tolist())

# local variable dimension - random in [2,5]
n_i = np.random.randint(2, 5)

# number of coupling constraints
S = 3

# generate a positive definite matrix
P = np.random.randn(n_i, n_i)
```

(continues on next page)
P = P.transpose() @ P
bias = np.random.randn(n_i, 1)

# declare a variable
x = Variable(n_i)

# define the local objective function
fn = QuadraticForm(x - bias, P)

# define the local constraint set
low = -2*np.ones((n_i, 1))
up = 2*np.ones((n_i, 1))
constr = Box(low, up)

# define the local contribution to the coupling constraints
A = np.random.randn(S, n_i)
coupling_fn = A.transpose() @ x

# create local problem and assign to agent
pb = ConstraintCoupledProblem(objective_function=fn,
                               constraints=constr,
                               coupling_function=coupling_fn)
agent.set_problem(pb)

# initialize allocation
y0 = np.zeros((S, 1))
algorithm = PrimalDecomposition(agent=agent,
                                 initial_condition=y0,
                                 enable_log=True)

def step_gen(k):
    # define a stepsize generator
    return 0.1/np.sqrt(k+1)

# run the algorithm
x_sequence, y_sequence, J_sequence = algorithm.run(iterations=1000, stepsize=step_gen,
                                                        M=100.0, verbose=True)
x_t, y_t, J_t = algorithm.get_result()
print("Agent {}: primal {} allocation ".format(agent.id, x_t.flatten(), y_t.flatten()))
np.save("agents.npy", nproc)

# save agent and sequence
with open('agent_{}/obj_function.pkl'.format(agent.id), 'wb') as output:
    pickle.dump(agent.problem.objective_function, output, pickle.HIGHEST_PROTOCOL)
with open('agent_{}/coup_function.pkl'.format(agent.id), 'wb') as output:
    pickle.dump(agent.problem.coupling_function, output, pickle.HIGHEST_PROTOCOL)
np.save("agent_{}/allocation_sequence.npy".format(agent.id), y_sequence)
np.save("agent_{}/primal_sequence.npy".format(agent.id), x_sequence)

Listing 26: examples/setups/distributed_primal_decomposition/results.py
import pickle

N = np.load("agents.npy")
S = 3

y_sequence = {}
x_sequence = {}
local_obj_function = {}
local_coup_function = {}
for i in range(N):
y.sequence[i] = np.load("agent_{}/allocation_sequence.npy".format(i))
x.sequence[i] = np.load("agent_{}/primal_sequence.npy".format(i))
    with open('agent_{}/obj_function.pkl'.format(i), 'rb') as input:
        local_obj_function[i] = pickle.load(input)
    with open('agent_{}/coup_function.pkl'.format(i), 'rb') as input:
        local_coup_function[i] = pickle.load(input)

# plot cost of primal sequence
plt.figure()
plt.title("Primal cost")
iterations = x.sequence[i].shape[0]
obj_function = np.zeros([iterations, 1])
for k in range(iterations):
    for i in range(N):
        obj_function[k] += local_obj_function[i].eval(x.sequence[i][k, :, 0].reshape(-1,1)).flatten()

plt.semilogy(obj_function)

# plot coupling constraint utilization
plt.figure()
plt.title("Coupling constraint utilization")
coup_function = np.zeros([iterations, S])
for k in range(iterations):
    for i in range(N):
        coup_function[k] += local_coup_function[i].eval(x.sequence[i][k, :, 0].reshape(-1,1)).flatten()

plt.plot(coup_function)
plt.show()

The two files can be executed by issuing the following commands in the example folder:

```bash
> mpirun -n 30 --oversubscribe python launcher.py
> python results.py
```
4.2 Complex examples

We provide three complex examples on realistic applications, one for each optimization set-up (see also the tutorial introduction).

4.2.1 Cost-coupled: classification via Logistic Regression

For the cost-coupled set-up, we consider a classification scenario [NedOz09]. In this example, a linear model is trained by minimizing the so-called logistic loss functions. The complete code of this example is given at the end of this page.

Problem formulation

Suppose there are \( N \) agents, where each agent \( i \) is equipped with \( m_i \) points \( p_{i,1}, \ldots, p_{i,m_i} \in \mathbb{R}^d \) (which represent training samples in a \( d \)-dimensional feature space). Moreover, suppose the points are associated to binary labels, that is, each point \( p_{i,j} \) is labeled with \( \ell_{i,j} \in \{-1, 1\} \), for all \( j \in \{1, \ldots, m_i\} \) and \( i \in \{1, \ldots, N\} \). The problem consists of building a linear classification model from the training samples by maximizing the a-posteriori probability of each class. In particular, we look for a separating hyperplane of the form \( \{ z \in \mathbb{R}^d \mid w^\top z + b = 0 \} \), whose parameters \( w \) and \( b \) can be determined by solving the convex optimization problem

\[
\min_{w,b} N \sum_{i=1}^{N} m_i \sum_{j=1}^{m_i} \log \left[ 1 + e^{-\left( w^\top p_{i,j} + b \right) \ell_{i,j}} \right] + \frac{C}{2} \| w \|^2,
\]

where \( C > 0 \) is a parameter affecting regularization. Notice that the problem is cost coupled (refer to the general formulation), with each local cost function \( f_i \) given by

\[
f_i(w, b) = m_i \sum_{j=1}^{m_i} \log \left[ 1 + e^{-\left( w^\top p_{i,j} + b \right) \ell_{i,j}} \right] + \frac{C}{2N} \| w \|^2, \quad i \in \{1, \ldots, N\}.
\]

The goal is to make agents agree on a common solution \( (w^*, b^*) \), so that all of them can compute the separating hyperplane as \( \{ z \in \mathbb{R}^d \mid (w^*)^\top z + b^* = 0 \} \).

Data generation model

We consider a bidimensional sample space \( (d = 2) \). Agents generate a certain number of samples (between 2 and 5) for both labels. For each label, the samples are drawn according to a multivariate gaussian distribution, with covariance matrix equal to the identity and mean equal to \((0,0)\) (for the label 1) and \((3,2)\) (for the label -1). The regularization parameter is set to \( C = 10 \).

Complete code

Listing 27: examples/setups/logistic_regression/launcher.py

```
# COST-COUPLED Example
# Logistic Regression for Classification
#
# Each agent has a certain number of randomly generated points, labeled 1 or -1.
# The points are generated by agents according to a multivariate normal distribution,
# with different mean and covariance for the two labels.

###############################################################
# COST-COUPLED Example
# Logistic Regression for Classification
#
# Each agent has a certain number of randomly generated points, labeled 1 or -1.
# The points are generated by agents according to a multivariate normal distribution,
# with different mean and covariance for the two labels.

(continues on next page)
```
# Compared Algorithms:
#
# - Distributed Subgradient
# - Gradient Tracking

# get MPI info
NN = MPI.COMM_WORLD.Get_size()
agent_id = MPI.COMM_WORLD.Get_rank()

# Generate a common graph (everyone uses the same seed)
Adj = binomial_random_graph(NN, p=0.3, seed=1)
W = metropolis_hastings(Adj)

np.random.seed(10*agent_id)

# Problem parameters
# parameters of gaussians
mu = (np.array([0, 0]).transpose(), np.array([3, 2]).transpose())
sigma = (np.eye(2), np.eye(2))

# number of samples (min 2 max 5 for each label)
nsamp = (np.random.randint(2, 6), np.random.randint(2, 6))

# regularization parameter
C = 10

# Generate problem data

# points
points = np.zeros((dim, nsamp[0]+nsamp[1]))
points[:, 0:nsamp[0]] = np.random.multivariate_normal(mu[0], sigma[0], nsamp[0]).
  -transpose()
points[:, nsamp[0]:] = np.random.multivariate_normal(mu[1], sigma[1], nsamp[1]).
  -transpose()

# labels
labels = np.ones((sum(nsamp), 1))
labels[nsamp[0]:] = -labels[nsamp[0]:]

# cost function
z = Variable(dim+1)
A = np.ones((dim+1, 1))
A[-1] = 0
obj_func = (C / (2 * NN)) * SquaredNorm(A @ z)

for j in range(sum(nsamp)):
    e_j = np.zeros((sum(nsamp), 1))
    e_j[j] = 1
    A_j = np.vstack((points @ e_j, 1))
    obj_func += Logistic(- labels[j] * A_j @ z)

# Distributed algorithms

# local agent and problem
agent = Agent(
    in_neighbors=np.nonzero(Adj[agent_id, :])[0].tolist(),
    out_neighbors=np.nonzero(Adj[:, agent_id])[0].tolist(),
    in_weights=W[agent_id, :].tolist())
pb = Problem(obj_func)
agent.set_problem(pb)

# instantiate the algorithms
x0 = 5*np.random.rand(dim+1, 1)
subgr = SubgradientMethod(agent=agent,
                           initial_condition=x0,
                           enable_log=True)
gradtr = GradientTracking(agent=agent,
                           initial_condition=x0,
                           enable_log=True)

def step_gen(k):
    return 1/((k+1)**0.6)
constant_stepsize = 0.001
num_iterations = 20000

# run the algorithms
subgr_seq = subgr.run(iterations=num_iterations, stepsize=step_gen)
gradtr_seq = gradtr.run(iterations=num_iterations, stepsize=constant_stepsize)

# print results
print("Subgradient method: agent {}:").format(agent_id, subgr.get_result().flatten())
print("Gradient tracking: agent {}:").format(agent_id, gradtr.get_result().flatten())

# save information
if agent_id == 0:
    with open('info.pkl', 'wb') as output:
        pickle.dump({'N': NN, 'size': dim+1, 'iterations': num_iterations}, output,
                    pickle.HIGHEST_PROTOCOL)

    with open('agent_{}/_func.pkl'.format(agent_id), 'wb') as output:
pickle.dump(obj_func, output, pickle.HIGHEST_PROTOCOL)

np.save("agent_/\seq_subgr.npy", format(agent_id), np.squeeze(subgr_seq))
np.save("agent_/\seq_gradtr.npy", format(agent_id), np.squeeze(gradtr_seq))

Listing 28: examples/setups/logistic_regression/results.py

import numpy as np
import matplotlib.pyplot as plt
from disropt.problems import Problem
import pickle

# initialize
with open('info.pkl', 'rb') as inp:
    info = pickle.load(inp)
NN = info['N']
iters = info['iterations']
size = info['size']

# load agent data
seq_subgr = np.zeros((NN, iters, size))
seq_gradtr = np.zeros((NN, iters, size))
local_function = {}
for i in range(NN):
    seq_subgr[i,:,:] = np.load("agent_/\seq_subgr.npy".format(i))
    seq_gradtr[i,:,:] = np.load("agent_/\seq_gradtr.npy".format(i))
with open('agent_/\func.pkl'.format(i), 'rb') as inp:
    local_function[i] = pickle.load(inp)

# solve centralized problem
global_obj_func = 0
for i in range(NN):
    global_obj_func += local_function[i]

global_pb = Problem(global_obj_func)
x_centr = global_pb.solve()
cost_centr = global_obj_func.eval(x_centr)
x_centr = x_centr.flatten()

# compute cost errors
cost_err_subgr = np.zeros((NN, iters))
cost_err_gradtr = np.zeros((NN, iters))

for i in range(NN):
    for t in range(iters):
        # first compute global function value at local point
        cost_i_tt_subgr = 0
        cost_i_tt_gradtr = 0
        for j in range(NN):
            cost_i_tt_subgr += local_function[j].eval(seq_subgr[i, t, :])[:, None]
            cost_i_tt_gradtr += local_function[j].eval(seq_gradtr[i, t, :])[:, None]

        # then compute errors
        cost_err_subgr[i, t] = abs(cost_i_tt_subgr - cost_centr)
cost_err_gradtr[i, t] = abs(cost_i_tt_gradtr - cost_centr)
# plot maximum cost error
plt.figure()
plt.title('Maximum cost error (among agents)')
plt.xlabel(r"iteration $k$")
plt.ylabel(r"$\max_{i} \left| \left( \sum_{j=1}^{N} f_j(x_i^k) - f^\star \right)/f^\star \right|$")
plt.semilogy(np.arange(iters), np.amax(cost_err_subgr/cost_centr, axis=0), label='Distributed Subgradient')
plt.semilogy(np.arange(iters), np.amax(cost_err_gradtr/cost_centr, axis=0), label='Gradient Tracking')
plt.legend()
plt.show()

# plot maximum solution error
sol_err_subgr = np.linalg.norm(seq_subgr - x_centr[None, None, :], axis=2)
sol_err_gradtr = np.linalg.norm(seq_gradtr - x_centr[None, None, :], axis=2)

plt.figure()
plt.title('Maximum solution error (among agents)')
plt.xlabel(r"iteration $k$")
plt.ylabel(r"$\max_{i} \|x_i^k - x^\star\|$")
plt.semilogy(np.arange(iters), np.amax(sol_err_subgr, axis=0), label='Distributed Subgradient')
plt.semilogy(np.arange(iters), np.amax(sol_err_gradtr, axis=0), label='Gradient Tracking')
plt.legend()
plt.show()

The two files can be executed by issuing the following commands in the example folder:

```bash
> mpirun -np 20 --oversubscribe python launcher.py
> python results.py
```

References

4.2.2 Common-cost: classification via Support Vector Machine

For the common-cost set-up, we consider a classification scenario [NoBü11]. In this example, a linear model is trained by maximizing the distance of the separating hyperplane from the training points. The complete code of this example is given at the end of this page.

Problem formulation

Suppose there are $N$ agents, where each agent $i$ is equipped with $m_i$ points $p_{i,1}, \ldots, p_{i,m_i} \in \mathbb{R}^d$ (which represent training samples in a $d$-dimensional feature space). Moreover, suppose the points are associated to binary labels, that is, each point $p_{i,j}$ is labeled with $\ell_{i,j} \in \{-1, 1\}$, for all $j \in \{1, \ldots, m_i\}$ and $i \in \{1, \ldots, N\}$. The problem consists of building a classification model from the training samples. In particular, we look for a separating hyperplane of the form $\{z \in \mathbb{R}^d \mid w^Tz + b = 0\}$ such that it separates all the points with $\ell_i = -1$ from all the points with $\ell_i = 1$.

In order to maximize the distance of the separating hyperplane from the training points, one can solve the following...
(convex) quadratic program:

\[
\min_{w, b, \xi} \frac{1}{2} \|w\|^2 + C \sum_{i=1}^{N} \sum_{j=1}^{m_i} \xi_{i,j} \\
\text{subject to } \ell_{i,j}(w^T p_{i,j} + b) \geq 1 - \xi_{i,j}, \quad \forall j, i \\
\xi \geq 0,
\]

where \( C > 0 \) is a parameter affecting regularization. The optimization problem above is called soft-margin SVM since it allows for the presence of outliers by activating the variables \( \xi_{i,j} \) in case a separating hyperplane does not exist. Notice that the problem can be viewed either as a common-cost problem or as a cost-coupled problem (refer to the general formulations). Here we consider the problem as a common cost, with the common objective function equal to

\[
f(w, b, \xi) = \frac{1}{2} \|w\|^2 + C \sum_{i=1}^{N} \sum_{j=1}^{m_i} \xi_{i,j}
\]

and each local constraint set \( X_i \) given by

\[
X_i = \{(w, b, \xi) \mid \xi \geq 0, \ell_{i,j}(w^T p_{i,j} + b) \geq 1 - \xi_{i,j}, \text{ for all } j \in \{1, \ldots, m_i\}\}.
\]

The goal is to make agents agree on a common solution \((w^*, b^*, \xi^*)\), so that all of them can compute the soft-margin separating hyperplane as \(\{z \in \mathbb{R}^d \mid (w^*)^T z + b^* = 0\}\).

**Data generation model**

We consider a bidimensional sample space \((d = 2)\). Agents generate a certain number of samples (between 2 and 5) for both labels. For each label, the samples are drawn according to a multivariate gaussian distribution, with covariance matrix equal to the identity and mean equal to \((0, 0)\) (for the label 1) and \((3, 2)\) (for the label -1). The regularization parameter is set to \(C = 10\).

**Complete code**

Listing 29: examples/setups/svm/launcher.py

```python
import dill as pickle
import numpy as np
from mpi4py import MPI
from disropt.agents import Agent
from disropt.algorithms import ConstraintsConsensus
from disropt.functions import Variable, SquaredNorm
from disropt.utils.graph_constructor import binomial_random_graph
from disropt.problems import Problem
```

(continues on next page)
```python
# get MPI info
NN = MPI.COMM_WORLD.Get_size()
agent_id = MPI.COMM_WORLD.Get_rank()

# Generate a common graph (everyone uses the same seed)
Adj = binomial_random_graph(NN, p=0.03, seed=1)

np.random.seed(10*agent_id)

# Problem parameters

# parameters of gaussians
mu = (np.array([0, 0]).transpose(), np.array([3, 2]).transpose())
sigma = (np.eye(2), np.eye(2))

dim = mu[0].shape[0] # dimension of sample space

# number of samples (min 2 max 5 for each label)
nsamp = (np.random.randint(2, 6), np.random.randint(2, 6))

# regularization parameter
C = 10

# Generate problem data

# points
points = np.zeros((dim, nsamp[0]+nsamp[1]))
points[:, 0:nsamp[0]] = np.random.multivariate_normal(mu[0], sigma[0], nsamp[0]).transpose()
points[:, nsamp[0]:] = np.random.multivariate_normal(mu[1], sigma[1], nsamp[1]).transpose()

# labels
labels = np.ones((sum(nsamp), 1))
labels[nsamp[0]:] = -labels[nsamp[0]:]

# cost function
z = Variable(dim+1+NN)
A = np.zeros((dim+1+NN, dim))
A[0:dim:, :] = np.eye(dim) # w = A @ z
B = np.zeros((dim+1+NN, 1))
B[dim+1:dim+NN+1] = np.ones((NN, 1)) # x_1 + ... + x_N = B @ z
D = np.zeros((dim+1+NN, 1))
D[dim] = 1 # b = D @ z
E = np.zeros((dim+1+NN, 1))
E[dim+1+agent_id] = 1 # x_i = E @ z

obj_func = (1/2) * (A @ z) @ (A @ z) + C * (B @ z)

# constraints
F = np.zeros((dim+1+NN, NN))
F[dim+1:dim+NN+1:, :] = np.eye(NN)

(continues on next page)
```

4.2. Complex examples
constr = []
for idx in np.arange(F.shape[1]):
    constr.append(F[:, idx] @ z >= 0)

for j in range(sum(nsamp)):
    constr.append(float(labels[j]) * (points[:, j].reshape(2, 1) @ (A @ z) + D @ z) >= 1 - E @ z)  # j-th point

#########################
# Distributed algorithms
#########################

# local agent and problem
agent = Agent(
in_neighbors=np.nonzero(Adj[agent_id, :])[0].tolist(),
out_neighbors=np.nonzero(Adj[:, agent_id])[0].tolist())
pb = Problem(obj_func, constr)
agent.set_problem(pb)
# instantiate the algorithm
constrcons = ConstraintsConsensus(agent=agent, enable_log=True)

n_iter = NN*3

# run the algorithm
constrcons_seq = constrcons.run(iterations=n_iter, verbose=True)

# print results
constrcons_x = constrcons.get_result()
print("Agent {}\: {}").format(agent_id, constrcons_x.flatten()))  # save information

if agent_id == 0:
    with open('info.pkl', 'wb') as output:
        pickle.dump({'N': NN, 'size': NN+dim+1, 'iterations': n_iter}, output, pickle.HIGHEST_PROTOCOL)

    with open('objective_function.pkl', 'wb') as output:
        pickle.dump(obj_func, output, pickle.HIGHEST_PROTOCOL)

    with open('agent_{}/constr.pkl'.format(agent_id), 'wb') as output:
        pickle.dump(constr, output, pickle.HIGHEST_PROTOCOL)

    np.save("agent_{}/seq.npy".format(agent_id), constrcons_seq)

Listing 30: examples/setup/svm/results.py

import numpy as np
import matplotlib.pyplot as plt
from disropt.problems import Problem
import pickle
import tikzplotlib

# initialize
with open('info.pkl', 'rb') as inp:
    info = pickle.load(inp)
NN = info['N']
iters = info['iterations']
size = info['size']

# load agent data
sequence = np.zeros((NN, iters, size))
local_constr = []
for i in range(NN):
    sequence[i, :, :] = np.load("agent_{}_seq.npy".format(i), allow_pickle=True).
    reshape((iters, size))
    with open('agent_{}_constr.pkl'.format(i), 'rb') as inp:
        local_constr[i] = pickle.load(inp)
with open('objective_function.pkl', 'rb') as inp:
    obj_func = pickle.load(inp)

# solve centralized problem
global_constr = []
for i in range(NN):
    global_constr.extend(local_constr[i])
global_pb = Problem(obj_func, global_constr)
x_centr = global_pb.solve()
cost_centr = obj_func.eval(x_centr)

# compute cost errors
cost_err = np.zeros((NN, iters))
for i in range(NN):
    for t in range(iters):
        cost_err[i, t] = abs(obj_func.eval(sequence[i, t, :].reshape((size, 1))) - cost_centr)

# compute max violation
vio_err = np.zeros((NN, iters))
for i in range(NN):
    for t in range(iters):
        xt = sequence[i, t, :].reshape((size, 1))
        max_err = np.zeros((len(global_constr), 1))
        for c in range(len(global_constr)):
            max_err[c] = global_constr[c].function.eval(xt)
        vio_err[i, t] = np.max(max_err)

# Plot the evolution of the local estimates
# generate N colors
colors = {} 
for i in range(NN):
    colors[i] = np.random.rand(3, 1).flatten()

# plot cost error
plt.figure()
plt.title('Evolution of cost error')
plt.xlabel(r"iteration $k$")
plt.ylabel(r"$|f(x_i^k) - f^\star|$")
for i in range(NN):
    plt.plot(np.arange(iters), cost_err[i, :], color=colors[i])

# plot violation error
plt.figure()
plt.title('Maximum constraint violation')
The two files can be executed by issuing the following commands in the example folder:

```bash
> mpirun -np 30 --oversubscribe python launcher.py
> python results.py
```

### References

#### 4.2.3 Constraint-coupled: charging of Plug-in Electric Vehicles (PEVs)

For the constraint-coupled set-up, we consider the problem of determining an optimal overnight charging schedule for a fleet of Plug-in Electric Vehicles (PEVs) [FalMa17]. The model described in this page is inspired by the model in the paper [VuEs16] (we consider the “charge-only” case without the integer constraints on the input variables). The complete code of this example is given at the end of this page.

### Problem formulation

Suppose there is a fleet of $N$ PEVs (agents) that must be charged by drawing power from the same electricity distribution network. Assuming the vehicles are connected to the grid at a certain time (e.g., at midnight), the goal is to determine an optimal overnight schedule to charge the vehicles, since the electricity price varies during the charging period.

Formally, we divide the entire charging period into a total of $T = 24$ time slots, each one of duration $\Delta T = 20$ minutes. For each PEV $i \in \{1, \ldots, N\}$, the charging power at time step $k$ is equal to $P_i u_i(k)$, where $u_i(k) \in [0, 1]$ is the input to the system and $P_i$ is the maximum charging power that can be fed to the $i$-th vehicle.

### System model

The state of charge of the $i$-th battery is denoted by $e_i(k)$, its initial state of charge is $E_i^{\text{init}}$, which by the end of the charging period has to attain at least $E_i^{\text{ref}}$. The charging conversion efficiency is denoted by $\zeta_i^u \triangleq 1 - \zeta_i$, where $\zeta_i > 0$ encodes the conversion losses. The battery’s capacity limits are denoted by $E_i^{\text{min}}, E_i^{\text{max}} \geq 0$. The system’s dynamics are therefore given by

\[
e_i(0) = E_i^{\text{init}}
\]

\[
e_i(k + 1) = e_i(k) + P_i \Delta T \zeta_i^u u_i(k), \quad k \in \{0, \ldots, T - 1\}
\]

\[
e_i(T) \geq E_i^{\text{ref}}
\]

\[
e_i^{\text{min}} \leq e_i(k) \leq E_i^{\text{max}}, \quad k \in \{1, \ldots, T\}
\]

\[
u_i(k) \in [0, 1], \quad k \in \{0, \ldots, T - 1\}.
\]

To model congestion avoidance of the power grid, we further consider the following (linear) coupling constraints among all the variables

\[
\sum_{i=1}^{N} P_i u_i(k) \leq P^{\text{max}}, \quad k \in \{0, \ldots, T - 1\},
\]
where $P_{\text{max}}$ is the maximum power that the be drawn from the grid.

**Optimization problem**

We assume that, at each time slot $k$, electricity has unit cost equal to $C^u(k)$. Since the goal is to minimize the overall consumed energy price, the global optimization problem can be posed as

$$\min_{u,e} \sum_{i=1}^{N} \sum_{k=0}^{T-1} C^u(k) P_i u_i(k)$$

subject to

$$\sum_{i=1}^{N} P_i u_i(k) \leq P_{\text{max}}, \quad k \in \{0, \ldots, T - 1\}$$

$$(u_i, e_i) \in X_i, \quad i \in \{1, \ldots, N\}.$$ 

The problem is recognized to be a *constraint-coupled* problem, with local variables $x_i$ equal to the stack of $e_i(k), u_i(k)$ for $k \in \{0, \ldots, T - 1\}$, plus $e_i(T)$. The local objective function is equal to

$$f_i(x_i) = \sum_{k=0}^{T-1} P_i u_i(k) C^u(k),$$

the local constraint set is equal to

$$X_i = \{(e_i, u_i) \in \mathbb{R}^{T+1} \times \mathbb{R}^T \text{ such that local dynamics is satisfied}\}$$

and the local coupling constraint function $g_i : \mathbb{R}^{2T+1} \to \mathbb{R}^T$ has components

$$g_{i,k}(x_i) = P_i u_i(k) - \frac{P_{\text{max}}}{N}, \quad k \in \{0, \ldots, T - 1\}.$$ 

The goal is to make each agent compute its portion $x^*_i = (e^*_i, u^*_i)$ of an optimal solution $(x^*_1, \ldots, x^*_N)$ of the optimization problem, so that all of them can know their own assignment of the optimal charging schedule, given by $(u^*_i(0), \ldots, u^*_i(T - 1))$.

**Data generation model**

The data are generated according to table in [VuEs16] (see Appendix).

**Complete code**

```python
# CONSTRAINT-COUPLED Example
# Charging of Plug-in Electric Vehicles (PEVs)
#
# The problem consists of finding an optimal overnight schedule to
# charge electric vehicles. See [Vu16] for the problem model
# and generation parameters.
#
# Note: in this example we consider the "charging-only" case

# Compared Algorithms:
```

(continues on next page)
```python
# - Distributed Dual Subgradient
# - Distributed Primal Decomposition

import dill as pickle
import numpy as np
from numpy.random import uniform as rnd
from mpi4py import MPI
from disropt.agents import Agent
from disropt.algorithms import DualSubgradientMethod, PrimalDecomposition
from disropt.functions import Variable
from disropt.utils.graph_constructor import binomial_random_graph, metropolis_hastings, binomial_random_graph_sequence
from disropt.problems import ConstraintCoupledProblem

# get MPI info
NN = MPI.COMM_WORLD.Get_size()
agent_id = MPI.COMM_WORLD.Get_rank()

# Generate a common graph (everyone uses the same seed)
Adj = binomial_random_graph(NN, p=0.2, seed=1)
W = metropolis_hastings(Adj)

# generate edge probabilities
edge_prob = np.random.uniform(0.3, 0.9, (NN, NN))
edge_prob[Adj == 0] = 0
i_lower = np.tril_indices(NN)
edge_prob[i_lower] = edge_prob.T[i_lower]  # symmetrize

# Generate problem parameters

# Problem parameters are generated according to the table in [Vu16]

#### Common parameters
TT = 24  # number of time windows
DeltaT = 20  # minutes
# PP_max = 3 * NN # kWh
PP_max = 0.5 * NN  # kWh
CC_u = rnd(19,35, (TT, 1))  # EUR/MWh - TT entries

#### Individual parameters
np.random.seed(10*agent_id)
PP = rnd(3,5)  # kW
EE_min = 1  # kWh
EE_max = rnd(8,16)  # kWh
EE_init = rnd(0.2,0.5) * EE_max  # kWh
EE_ref = rnd(0.55,0.8) * EE_max  # kWh
zeta_u = 1 - rnd(0.015, 0.075)  # pure number

# Generate problem object
```

(continues on next page)
### normalize unit measures
DeltaT = DeltaT/60 # minutes -> hours
CC_u = CC_u/1e3 # Euro/MWh -> Euro/KWh

### optimization variables
z = Variable(2*TT + 1) # stack of e (state of charge) and u (input charging power)
e = np.vstack((np.eye(TT+1), np.zeros((TT, TT+1)))) @ z # T+1 components (from 0 to T)
u = np.vstack((np.zeros((TT+1, TT)), np.eye(TT))) @ z # T components (from 0 to T-1)

### objective function
obj_func = PP * (CC_u @ u)

### coupling function
coupling_func = PP*u - (PP_max/NN)

### local constraints
e_0 = np.zeros((TT+1, 1))
e_T = np.zeros((TT+1, 1))
e_0[0] = 1
e_T[TT] = 1
cstr = [e_0 @ e == EE_init, e_T @ e >= EE_ref] # feedback and reference constraints

for kk in range(0, TT):
    e_cur = np.zeros((TT+1, 1))
    u_cur = np.zeros((TT, 1))
    e_new = np.zeros((TT+1, 1))
    e_cur[kk] = 1
    u_cur[kk] = 1
    e_new[kk+1] = 1
    cstr.append(e_new @ e == e_cur @ e + PP*DeltaT*zeta_u*u_cur @ u) # dynamics
    cstr.extend([u_cur @ u <= 1, u_cur @ u >= 0]) # input constraints
    cstr.extend([e_new @ e <= EE_max, e_new @ e >= EE_min]) # state constraints

### Distributed algorithms

#### local agent and problem
agent = Agent(
in_neighbors=np.nonzero(Adj[agent_id, :])[0].tolist(),
out_neighbors=np.nonzero(Adj[:, agent_id])[0].tolist(),
in_weights=W[agent_id, :].tolist())

pb = ConstraintCoupledProblem(obj_func, cstr, coupling_func)
agent.set_problem(pb)

# instantiate the algorithms
# y0 = np.zeros((TT, 1))
# mu0 = np.zeros((TT, 1))
y0 = 10*np.random.rand(TT, 1)
mu0 = 10*np.random.rand(TT, 1)

theothers = [i for i in range(NN) if i != agent_id]
y_others = agent.communicator.neighbors_exchange(y0, theothers, theothers, False)
y_others[agent_id] = y0

(continues on next page)
\[ y_{\text{mean}} = \text{sum}\{x \text{ for } _, x \text{ in } y_{\text{others}}.\text{items()})/\text{NN} \]
\[ y_0 -= y_{\text{mean}} \]

\[ \text{dds = DualSubgradientMethod(agent=agent,} \]
\[ \text{initial_condition=mu0,} \]
\[ \text{enable_log=True)} \]
\[ \text{dpd = PrimalDecomposition (agent=agent,} \]
\[ \text{initial_condition=y0,} \]
\[ \text{enable_log=True)} \]

\[ \text{num_iterations = 1000} \]

\[ \text{# generate sequence of adjacency matrices} \]
\[ \text{Adj_seq = binomial_random_graph_sequence(Adj, num_iterations, edge_prob, NN, 1)} \]

\[ \text{def step_gen(k):} \# \text{define a stepsize generator} \]
\[ \text{\quad return 1/((k+1)**0.6)} \]

\[ \text{def update_graph(k):} \# \text{define a stepsize generator} \]
\[ \text{\quad return 0.01} \]

\[ \text{def step_gen(k): # define a stepsize generator} \]
\[ \text{\quad return 1/((k+1)**0.6)} \]

\[ \text{# run the algorithms} \]
\[ \text{if agent_id == 0:} \]
\[ \text{\quad print("Distributed dual subgradient")} \]
\[ \text{\quad , dds_seq = dds.run(iterations=num_iterations, stepsize=step_gen, verbose=True)} \]
\[ \text{if agent_id == 0:} \]
\[ \text{\quad print("Distributed primal decomposition")} \]
\[ \text{\quad dpd_seq, _, _ = dpd.run(iterations=num_iterations, stepsize=step_gen, M=30.0,verbose=True)} \]

\[ \text{# save information} \]
\[ \text{if agent_id == 0:} \]
\[ \text{\quad with open('info.pkl', 'wb') as output:} \]
\[ \text{\quad \quad pickle.dump({'N': NN, 'iterations': num_iterations, 'n_coupling': TT}, output, pickle.HIGHEST_PROTOCOL)} \]
\[ \text{\quad with open('agent_#{agent_id}_objective_func.pkl'.format(agent_id), 'wb') as output:} \]
\[ \text{\quad \quad pickle.dump(obj_func, output, pickle.HIGHEST_PROTOCOL)} \]
\[ \text{\quad with open('agent_#{agent_id}_coupling_func.pkl'.format(agent_id), 'wb') as output:} \]
\[ \text{\quad \quad pickle.dump(coupling_func, output, pickle.HIGHEST_PROTOCOL)} \]
\[ \text{\quad with open('agent_#{agent_id}_local_constr.pkl'.format(agent_id), 'wb') as output:} \]
\[ \text{\quad \quad pickle.dump(constr, output, pickle.HIGHEST_PROTOCOL)} \]
\[ \text{\quad np.save("agent_#{agent_id}_seq_dds.npy", format(agent_id), dds_seq)} \]
\[ \text{\quad np.save("agent_#{agent_id}_seq_dpd.npy", format(agent_id), dpd_seq)} \]
Listing 32: examples/setups/pev/results.py

```python
import numpy as np
import matplotlib.pyplot as plt
from disropt.problems import Problem
from disropt.functions import ExtendedFunction
from disropt.constraints import ExtendedConstraint
import dill as pickle

# initialize
with open('info.pkl', 'rb') as inp:
    info = pickle.load(inp)
NN = info['N']
iters = info['iterations']
SS = info['n_coupling']

# load agent data
seq_dds = {}
seq_dpd = {}
local_obj_func = {}
local_coup_func = {}
local_constr = {}
for i in range(NN):
    seq_dds[i] = np.load("agent_{i}_seq_dds.npy".format(i))
    seq_dpd[i] = np.load("agent_{i}_seq_dpd.npy".format(i))
    with open('agent_{i}_objective_func.pkl'.format(i), 'rb') as inp:
        local_obj_func[i] = pickle.load(inp)
    with open('agent_{i}_coupling_func.pkl'.format(i), 'rb') as inp:
        local_coup_func[i] = pickle.load(inp)
    with open('agent_{i}_local_constr.pkl'.format(i), 'rb') as inp:
        local_constr[i] = pickle.load(inp)

# solve centralized problem
dim = sum([f.input_shape[0] for f in local_obj_func.items()])  # size of overall variable
centr_obj_func = 0
centr_constr = []
centr_coupling_func = 0
pos = 0
for i in range(NN):
    n_i = local_obj_func[i].input_shape[0]
    centr_obj_func += ExtendedFunction(local_obj_func[i], n_var = dim-n_i, pos=pos)
    centr_constr.extend(ExtendedConstraint(local_constr[i], n_var = dim-n_i, pos=pos))
    centr_coupling_func += ExtendedFunction(local_coup_func[i], n_var = dim-n_i,
                                                 --pos=pos)
    pos += n_i
centr_constr.append(centr_coupling_func <= 0)
global_pb = Problem(centr_obj_func, centr_constr)
x_centr = global_pb.solve()
cost_centr = centr_obj_func.eval(x_centr).flatten()
x_centr = x_centr.flatten()

# compute costs and coupling values
cost_seq_dds = np.zeros(iters) - cost_centr
```

(continues on next page)
cost_seq_dpd = np.zeros(iters) - cost_centr
coup_seq_dds = np.zeros((iters, SS))
coup_seq_dpd = np.zeros((iters, SS))

for t in range(iters):
    for i in range(NN):
        cost_seq_dds[t] += local_obj_func[i].eval(seq_dds[i][t, :])
        cost_seq_dpd[t] += local_obj_func[i].eval(seq_dpd[i][t, :])
        coup_seq_dds[t] += np.squeeze(local_coup_func[i].eval(seq_dds[i][t, :]))
        coup_seq_dpd[t] += np.squeeze(local_coup_func[i].eval(seq_dpd[i][t, :]))

# plot cost
plt.figure()
plt.title('Evolution of cost error')
plt.xlabel(r"iteration $k$")
plt.ylabel(r"$| \sum_{i=1}^N f_i(x_i^k) - f^\star|/f^\star$"
plt.semilogy(np.arange(iters), np.abs(cost_seq_dds/cost_centr), label='Distributed Dual Subgradient')
plt.semilogy(np.arange(iters), np.abs(cost_seq_dpd/cost_centr), label='Distributed Primal Decomposition')
plt.legend()

# plot maximum coupling constraint value
plt.figure()
plt.title('Evolution of maximum coupling constraint value')
plt.xlabel(r"iteration $k$")
plt.ylabel(r"$\max_s \sum_{i=1}^N g_{is}(x_i^k)$"
plt.plot(np.arange(iters), np.amax(coup_seq_dds, axis=1), label='Distributed Dual Subgradient')
plt.plot(np.arange(iters), np.amax(coup_seq_dpd, axis=1), label='Distributed Primal Decomposition')
plt.legend()
plt.show()

The two files can be executed by issuing the following commands in the example folder:

> mpirun -np 50 --oversubscribe python launcher.py
> python results.py

References
disropt has been designed to make the execution and implementation of distributed algorithms as easy as possible. Here you find the full documentation.

**5.1 Agents**

**5.1.1 Agent**

```python
class disruptagents.Agent (in_neighbors=None, out_neighbors=None, communicator=None, in_weights=None, out_weights=None, auto_local=True)
```

Bases: object

The Agent object represents an agent in a network with communication capabilities

**Parameters**

- **in_neighbors** *(list)* – list of agents from which communication is received
- **out_neighbors** *(list)* – list of agents to which information is sent
- **communicator** *(Communicator, optional)* – a Communicator object used to perform communications (if none is provided, it is automatically set to MPICommunicator). Defaults to None.
- **in_weights** *(list or dict, optional)* – list or dict containing weights to assign to information coming from each in-neighbor. If a list is provided, it must have length equal to the number of agents in the network. If a dict is provided, it must have a key for each in-neighbor and, associated to it, the corresponding weight. Defaults to None, implies equal in_weights to in-neighbors.
- **out_weights** *(list or dict, optional)* – list or dict containing weights to assign to out-neighbor. If a list is provided, it must have length equal to the number of agents in the network. If a dict is provided, it must have a key for each out-neighbor and, associated to it, the corresponding weight. Defaults to None, implies equal in_weights to out-neighbors.
- **auto_local** *(bool, optional)* – If False the (in-)weight for the local agent must be provided. Otherwise it is set automatically, provided that the in_weights have sum in [0,1]. Defaults to True.

**id**

id of the Agent

**Type** int

**in_neighbors**

list of in-neighbors
Type list

**out_neighbors**

list of out-neighbors

Type list

**in_weights**

a dict containing weights to assign to information coming from each in-neighbor.

Type dict

**out_weights**

a dict containing weights to assign to out-neighbors.

Type dict

**communicator**

Communicator object used to perform communications.

Type *Communicator*

**problem**

Local optimization problem.

Type *Problem*

**neighbors_exchange**(obj, dict_neigh=False, stop_event=None)

Exchange data with neighbors (synchronously). Send obj to the out-neighbors and receive received_obj from in-neighbors

**Parameters**

- **obj** (*Any*) – object to send
- **dict_neigh** – True if obj contains a dictionary with different objects for each neighbor. Defaults to False.

**Returns** a dictionary containing an object for each in-neighbor

**Return type** dict

**neighbors_receive_asynchronous**()

Receive data from in-neighbors (if any have been sent)

**Returns** a dictionary containing an object for each in-neighbor that has sent one

**Return type** dict

**neighbors_send**(obj)

Send data to out-neighbors

**Parameters** **obj** (*Any*) – object to send

**set_neighbors**(in_neighbors, out_neighbors)

Set in and out neighbors

**Parameters**

- **in_neighbors** (*list*) – list of agents from which communication is received
- **out_neighbors** (*list*) – list of agents to which information is send

**set_problem**(problem)

set the local optimization problem

**Parameters** **problem** (*Problem*) – Problem object
Raises **TypeError** – Input must be a Problem object

**set_weights** *(in_weights=None, out_weights=None, auto_local=True)*
Set in_weights to assign to in-neighbors and the one for agent itself.

**Parameters**

- **in_weights** *(Union[list, dict, None]) – list or dict containing in_weights to assign to information coming from each in-neighbor. If a list is provided, it must have length equal to the number of agents in the network. If a dict is provided, it must have a key for each in-neighbor and, associated to it, the corresponding weight. Defaults to None, implies equal in_weights to in-neighbors.*

- **out_weights** *(Union[list, dict, None]) – list or dict containing in_weights to assign to out-neighbors. If a list is provided, it must have length equal to the number of agents in the network. If a dict is provided, it must have a key for each out-neighbor and, associated to it, the corresponding weight. Defaults to None, implies equal in_weights to out neighbors.*

- **auto_local** *(bool) – If False the weight for the local agent must be provided. Otherwise it is set automatically, provided that the in_weights have sum in [0,1]. Defaults to True.*

**Raises**

- **ValueError** – If a dict is provided as argument, it must contain a key for each in-neighbor.

- **ValueError** – Input must be list or dict

- **ValueError** – If auto_local is not False, the provided in_weights must have sum in [0,1]

## 5.2. Communicators

### 5.2.1 Communicator

**class** `disropt.communicators.Communicator`

Communicator abstract class

- **neighbors_exchange** *(send_obj, in_neighbors, out_neighbors, dict_neigh, stop_event)*
  
  *Return type* `Dict[int, Any]`

- **neighbors_receive** *(neighbors, stop_event)*
  
  *Return type* `Dict[int, Any]`

- **neighbors_receive_asynchronous** *(neighbors)*
  
  *Return type* `Dict[int, Any]`

- **neighbors_send** *(obj, neighbors)*
5.2.2 MPICommunicator

class disropt.communicators.MPICommunicator
   Bases: disropt.communicators.communicators.Communicator

   Communicator class that performs communications through MPI. Requires mpi4py.

   comm
      communication world

   size
      size of the network.
      Type int

   rank
      rank of the processor
      Type int

   neighbors_exchange
      (send_obj, in_neighbors, out_neighbors, dict_neigh=False, stop_event=None)
      Exchange information (synchronously) with neighbors.

      Parameters
      • send_obj (Any) – object to send
      • in_neighbors (List[int]) – list of in-neighbors
      • out_neighbors (List[int]) – list of out-neighbors
      • dict_neigh (bool) – True if send_obj contains a dictionary with different objects for each neighbor. Defaults to False
      • stop_event (Optional[Event]) – an Event object that is monitored during the execution. If the event is set, the function returns immediately. Defaults to None (does not wait upon any event)

      Return type Dict[int, Any]
      Returns data received by in-neighbors

   neighbors_receive
      (neighbors, stop_event=None)
      Receive data from neighbors (waits until data are received from all neighbors).

      Parameters
      • neighbors (List[int]) – list of in-neighbors
      • stop_event (Optional[Event]) – an Event object that is monitored during the execution. If the event is set, the function returns immediately. Defaults to None (does not wait upon any event)

      Return type Dict[int, Any]
      Returns data received by in-neighbors

   neighbors_receive_asynchronous
      (neighbors)
      Receive data (if any) from neighbors.

      Parameters
      • neighbors (List[int]) – list of in-neighbors

      Return type Dict[int, Any]
      Returns data received by in-neighbors (if any)
neighbors_send(obj, neighbors)
    Send data to neighbors.

Parameters
    • obj (Any) – object to send
    • neighbors (List[int]) – list of out-neighbors

5.3 Algorithms

5.3.1 Consensus Algorithms

Consensus

class disropt.algorithms.consensus.Consensus(agent, initial_condition, enable_log=False)
    Bases: disropt.algorithms.algorithm.Algorithm
    Consensus Algorithm [OlSa07]
    From the perspective of agent $i$ the algorithm works as follows. For $k = 0, 1, \ldots$
    \[
    x_{i}^{k+1} = \sum_{j=1}^{N} w_{ij} x_{j}^{k}
    \]
    where $x_{i} \in \mathbb{R}^{n}$. The weight matrix $W = [w_{ij}]$ should be doubly-stochastic in order to have convergence to the average of the local initial conditions. If $W$ is row-stochastic convergence is still attained but at a different point. Other type of matrices can be used, but convergence is not guaranteed. Also time-varying graphs can be adopted.

Parameters
    • agent (Agent) – agent to execute the algorithm
    • initial_condition (numpy.ndarray) – initial condition
    • enable_log (bool) – True for enabling log

agent
    agent to execute the algorithm
    Type Agent

x0
    initial condition
    Type numpy.ndarray

x
    current value of the local solution
    Type numpy.ndarray

shape
    shape of the variable
    Type tuple

x_neigh
    dictionary containing the local solution of the (in-)neighbors
**Type**  
dict

**enable_log**  
True for enabling log

**Type**  
bool

**get_result()**  
Return the actual value of x

**Returns**  
value of x

**Return type**  
numpy.ndarray

**iterate_run(**kwargs)**  
Run a single iterate of the algorithm

**run**(iterations=100, verbose=False, **kwargs)**  
Run the algorithm for a given number of iterations

**Parameters**
- **iterations** *(int)* – Number of iterations. Defaults to 100.
- **verbose** *(bool)* – If True print some information during the evolution of the algorithm. Defaults to False.

**AsynchronousConsensus**

class disrupt.algorithms.consensus.AsynchronousConsensus(agent, initial_condition, enable_log=False, force_sleep=False, maximum_sleep=0.01, sleep_type='random', force_computation_time=False, maximum_computation_time=0.01, computation_time_type='random', force_unreliable_links=False, link_failure_probability=0)

**Bases:** disrupt.algorithms.algorithm.Algorithm

Asynchronous Consensus Algorithm

From the perspective of agent $i$ the algorithm works as follows. When agent $i$ gets awake it updates its local solution as

$$x_i \leftarrow \sum_{j \in \mathcal{N}_i} w_{ij} x_{ji}$$

where $\mathcal{N}_i$ is the current set of in-neighbors and $x_{ji}, j \in \mathcal{N}_i$ is the local copy of $x_j$ available at node $i$ (which can be outdated, due to asynchrony, computation time and link failures).

**Parameters**
- **agent** *(Agent)* – agent to execute the algorithm
- **initial_condition** *(ndarray)* – initial condition
- **enable_log** *(bool)* – True for enabling log. Defaults to False.
• **force_sleep** (bool) – True if one wants to force sleep after the computation phase. Defaults to False.

• **maximum_sleep** (float) – Maximum allowed sleep. Defaults to 0.01.

• **sleep_type** (str) – Type of sleep time (“constant”, “random”). Defaults to “random”.

• **force_computation_time** (bool) – True if one wants to force length computation phase. Defaults to False.

• **maximum_computation_time** (float) – Maximum allowed computation time. Defaults to 0.01.

• **computation_time_type** (str) – Type of computation time (“constant”, “random”). Defaults to “random”.

• **force_unreliable_links** (bool) – True if one wants to force unreliable links. Defaults to False.

• **link_failure_probability** (float) – Probability of incoming links failure. Defaults to 0.

agent
agent to execute the algorithm

    Type Agent

x0
initial condition

    Type numpy.ndarray

x
current value of the local solution

    Type numpy.ndarray

shape
shape of the variable

    Type tuple

x_neigh
dictionary containing the local solution of the (in-)neighbors

    Type dict

enable_log
True for enabling log

    Type bool

timestamp_sequence_awake
list of timestamps at which node get awake

    Type list

timestamp_sequence_sleep
list of timestamps at which node go to sleep

    Type list

force_sleep
True if one wants to force sleep after the computation phase. Defaults to False.
maximum_sleep
    Maximum allowed sleep. Defaults to 0.01.

sleep_type
    Type of sleep time("constant", “random”). Defaults to “random”.

force_computation_time
    True if one want sto force length computation phase. Defaults to False.

maximum_computation_time
    Maximum allowed computation time. Defaults to 0.01.

computation_time_type
    Type of computation time (“constant”, “random”). Defaults to “random”.

force_unreliable_links
    True if one wants to force unreliable links. Defaults to False.

link_failure_probability
    Probability of incoming links failure. Defaults to 0.

get_result()
    Return the actual value of x

    Returns  value of x
    Return type  numpy.ndarray

iterate_run(**kwargs)
    Run a single iterate

run(running_time=5.0)
    Run the asynchronous consensus algorithm for a certain amount of time

    Parameters running_time(float) – Total run time. Defaults to 5.0.
    Returns  timestamp_sequence_awake, timestamp_sequence_sleep, sequence
    Return type  tuple

BlockConsensus

class disrupt.algorithms.consensus.BlockConsensus(agent, initial_condition, enable_log=False, blocks_list=None, probabilities=None, awakening_probability=1.0)

Bases: disrupt.algorithms.algorithm.Algorithm

Block-wise consensus [FaNo19]

At each iteration, the agent can update its local estimate or not at each iteration according to a certain probability (awakening_probability). From the perspective of agent $i$ the algorithm works as follows. At iteration $k$ if the agent is awake, it selects a random block $\ell_i^k$ of its local solution and updates

$$x_{i,\ell}^{k+1} = \begin{cases} \sum_{j \in \mathcal{N}_i} w_{ij} x_j^k & \text{if } \ell_i^k = \ell_i^k \\ x_{i,\ell}^k & \text{otherwise} \end{cases}$$

where $\mathcal{N}_i$ is the current set of in-neighbors and $x_{j|i}, j \in \mathcal{N}_i$ is the local copy of $x_j$ available at node $i$ and $x_{i,\ell}$ denotes the $\ell$-th block of $x_i$. Otherwise $x_{i,\ell}^{k+1} = x_{i,\ell}^k$.

Parameters
• **agent** (*Agent*) – agent to execute the algorithm
• **initial_condition** (*ndarray*) – initial condition
• **enable_log** (*bool*) – True for enabling log
• **blocks_list** (*Optional[List[Tuple]]*) – the list of blocks (list of tuples)
• **probabilities** (*Optional[List[float]]*) – list of probabilities of drawing each block
• **awakening_probability** (*float*) – probability of getting awake at each iteration

### get_result()
Return the actual value of x

**Returns**
value of x

**Return type**
numpy.ndarray

### iterate_run(**kwargs)
Run a single iterate of the algorithm

### run(iterations=100, verbose=False)
Run the algorithm for a given number of iterations

**Parameters**

• **iterations** (*int*) – Number of iterations. Defaults to 100.
• **verbose** (*bool*) – If True print some information during the evolution of the algorithm. Defaults to False.

### PushSumConsensus
class disropt.algorithms.consensus.PushSumConsensus(*agent, initial_condition, enable_log=False*)

Bases: disropt.algorithms.algorithm.Algorithm

Push-Sum Consensus Algorithm

From the perspective of agent $i$ the algorithm works as follows. For $k = 0, 1, \ldots$

\[
x_i^{k+1} = \sum_{j=1}^{N} w_{ij} x_j^k
\]

\[
y_i^{k+1} = \sum_{j=1}^{N} w_{ij} y_j^k
\]

\[
z_i^{k+1} = \frac{x_i^{k+1}}{y_i^{k+1}}
\]

where $x_i \in \mathbb{R}^n$. The weight matrix $W = [w_{ij}]$ should be column-stochastic in order to let $z_i^k$ converge to the average of the local initial conditions. Also time-varying graphs can be adopted.

**Parameters**

• **agent** (*Agent*) – agent to execute the algorithm
• **initial_condition** (*numpy.ndarray*) – initial condition
• **enable_log** (*bool*) – True for enabling log
agent
tagent to execute the algorithm
  Type Agent

z0
initial condition
  Type numpy.ndarray

z
  current value of the local solution
  Type numpy.ndarray

shape
  shape of the variable
  Type tuple

x_neigh
dictionary containing the x values of the (in-)neighbors
  Type dict

y_neigh
dictionary containing the y values of the (in-)neighbors
  Type dict

enable_log
  True for enabling log
  Type bool

get_result()
  Return the actual value of x
  Returns value of x
  Return type numpy.ndarray

iterate_run(**kwargs)
  Run a single iterate of the algorithm

run(iterations=100, verbose=False, **kwargs)
  Run the algorithm for a given number of iterations
  Parameters
    • iterations (int) – Number of iterations. Defaults to 100.
    • verbose (bool) – If True print some information during the evolution of the algorithm.
      Defaults to False.
5.3.2 (Sub)gradient based Algorithms

Distributed projected (Sub)gradient Method

class disropt.algorithms.subgradient.SubgradientMethod(agent, initial_condition, enable_log=False)

Bases: disropt.algorithms.consensus.Consensus

Distributed projected (sub)gradient method.

From the perspective of agent $i$ the algorithm works as follows. For $k = 0, 1, \ldots$

$$y_i^k = \sum_{j=1}^{N} w_{ij} x_j^k$$

$$x_i^{k+1} = \Pi_X [y_i^k - \alpha^k \nabla f_i(y_i^k)]$$

where $x_i, y_i \in \mathbb{R}^n$, $X \subseteq \mathbb{R}^n$, $\alpha^k$ is a positive stepsize, $w_{ij}$ denotes the weight assigned by agent $i$ to agent $j$, $\Pi_X []$ denotes the projection operator over the set $X$ and $\nabla f_i(y_i^k) \in \partial f_i(y_i^k)$ a (sub)gradient of $f_i$ computed at $y_i^k$. The weight matrix $W = [w_{ij}]_{i,j=1}^N$ should be doubly stochastic.

The algorithm, as written above, was originally presented in [NeOz09]. Many other variants and extension has been proposed, allowing for stochastic objective functions, time-varying graphs, local stepsize sequences. All these variant can be implemented through the `SubgradientMethod` class.

run(iterations=1000, stepsize=0.001, verbose=False)

Run the algorithm for a given number of iterations

Parameters

- **iterations** (int) – Number of iterations. Defaults to 1000.
- **stepsize** (Union[float, Callable]) – If a float is given as input, the stepsize is constant. If a function is given, it must take an iteration $k$ as input and output the corresponding stepsize. Defaults to 0.1.
- **verbose** (bool) – If True print some information during the evolution of the algorithm. Defaults to False.

Raises

- **TypeError** – The number of iterations must be an int
- **TypeError** – The stepsize must be a float or a callable
- **ValueError** – Only sets (children of AbstractSet) with explicit projections are currently supported
- **ValueError** – Only one constraint per time is currently supported

Return type: ndarray

Returns: return the sequence of estimates if enable_log=True.
Randomized Block (Sub)gradient Method

class disropt.algorithms.subgradient.BlockSubgradientMethod(agent, initial_condition, **kwargs)

Bases: disropt.algorithms.consensus.BlockConsensus

Distributed block subgradient method. This is a special instance of the Block Proximal Method in [FaNo19].

At each iteration, the agent can update its local estimate or not at each iteration according to a certain probability (awakening_probability). From the perspective of agent $i$ the algorithm works as follows. At iteration $k$ if the agent is awake, it selects a random block $\ell^k_i$ of its local solution and updates

$$y^k_i = \sum_{j \in N_i} w_{ij} x^k_{ji}$$  \hspace{1cm} (5.1)

$$x^{k+1}_{i,\ell} = \begin{cases} 
\Pi_{X_\ell} \left[ y^k_i - \alpha^k_i \nabla f_i(y^k_i) \right] & \text{if } \ell = \ell^k_i \\
 x^k_{i,\ell} & \text{otherwise}
\end{cases}$$  \hspace{1cm} (5.2)

then it broadcasts $x^{k+1}_{i,\ell}$ to its out-neighbors. Otherwise (if the agent is not awake) $x^{k+1}_{i,\ell} = x^k_{i,\ell}$. Here $N_i$ is the current set of in-neighbors and $x^k_{ji}, j \in N_i$ is the local copy of $x_j$ available at node $i$ and $x_{i,\ell}$ denotes the $\ell$-th block of $x_i$. The weight matrix $W = [w_{ij}]_{N_i \times N_j}$ should be doubly stochastic.

Notice that if there is only one block and $\text{awakening\_probability}=1$ the $\text{BlockSubgradientMethod}$ reduces to the $\text{SubgradientMethod}$.

run (iterations=1000, stepsize=0.1, verbose=False)

Run the algorithm for a given number of iterations

Parameters

- **iterations** (int) – Number of iterations. Defaults to 1000.
- **stepsize** (Union[Callable[[int], float]]) – If a float is given as input, the stepsize is constant. If a function is given, it must take an iteration $k$ as input and output the corresponding stepsize. Defaults to 0.1.
- **verbose** (bool) – If True print some information during the evolution of the algorithm. Defaults to False.

Raises

- **TypeError** – The number of iterations must be an int
- **TypeError** – The stepsize must be a float or callable
- **ValueError** – Only sets (children of AbstractSet) with explicit projections are currently supported
- **ValueError** – Only one constraint per time is currently supported

Return type

Returns

return the sequence of estimates if enable_log=True.
Distributed Gradient Tracking

class disropt.algorithms.gradient_tracking.GradientTracking(agent, \( \text{initial\_condition} \), enable_log=False)

Bases: disropt.algorithms.algorithm.Algorithm

Gradient Tracking Algorithm [...]

From the perspective of agent \( i \) the algorithm works as follows. For \( k = 0, 1, \ldots \)

\[
x_i^{k+1} = \sum_{j=1}^{N} w_{ij} x_j^k - \alpha d_i^k
\]

\[
d_i^{k+1} = \sum_{j=1}^{N} w_{ij} d_j^k - [\nabla f_i(x_i^{k+1}) - \nabla f_i(x_i^k)]
\]

where \( x_i \in \mathbb{R}^n \) and \( d_i \in \mathbb{R}^n \). The weight matrix \( W = [w_{ij}] \) must be doubly-stochastic. Extensions to other class of weight matrices \( W \) are not currently supported.

Parameters

- **agent** (*Agent*) – agent to execute the algorithm
- **initial\_condition** (*numpy.ndarray*) – initial condition for \( x_i \)
- **enable\_log** (*bool*) – True for enabling log

**agent**

agent to execute the algorithm

Type *Agent*

**x0**

initial condition

Type *numpy.ndarray*

**x**

current value of the local solution

Type *numpy.ndarray*

**d**

current value of the local tracker

Type *numpy.ndarray*

**shape**

shape of the variable

Type *tuple*

**x\_neigh**

dictionary containing the local solution of the (in-)neighbors

Type *dict*

**d\_neigh**

dictionary containing the local tracker of the (in-)neighbors

Type *dict*

**enable\_log**

True for enabling log
**get_result()**

Return the actual value of x

**Returns** value of x

**Return type** numpy.ndarray

**iterate_run**(stepsize, **kwargs)

Run a single iterate of the gradient tracking algorithm

**run**(iterations=1000, stepsize=0.1, verbose=False)

Run the gradient tracking algorithm for a given number of iterations

**Parameters**

- **iterations**(int) – Number of iterations. Defaults to 1000.
- **stepsize**(Union[float, Callable]) – If a float is given as input, the stepsize is constant. Default is 0.01.
- **verbose**(bool) – If True print some information during the evolution of the algorithm. Defaults to False.

**Raises**

- **TypeError** – The number of iterations must be an int
- **TypeError** – The stepsize must be a float

**Return type** ndarray

**Returns** return the sequence of estimates if enable_log=True.

---

**Distributed Gradient Tracking (over directed, unbalanced graphs)**

class disropt.algorithms.gradient_tracking.DirectedGradientTracking(agent, initial_condition, enable_log=False)

**Bases:** disropt.algorithms.consensus.PushSumConsensus

Gradient Tracking Algorithm [XiKh18]

From the perspective of agent $i$ the algorithm works as follows. For $k = 0, 1, \ldots$

\[ x_i^{k+1} = \sum_{j=1}^{N} a_{ij} x_j^k - \alpha y_i^k \]

\[ y_i^{k+1} = \sum_{j=1}^{N} b_{ij} (y_j^k - [\nabla f_j(x_j^{k+1}) - \nabla f_j(x_j^k)]) \]

where $x_i \in \mathbb{R}^n$ and $y_i \in \mathbb{R}^n$. The weight matrix $A = [a_{ij}]$ must be row-stochastic, while $B = [b_{ij}]$ must be column-stochastic. The underlying graph can be directed (and unbalanced).

**get_result()**

Return the actual value of x

**Returns** value of x

**Return type** numpy.ndarray
iterate_run(**kwargs)
    Run a single iterate of the algorithm

References

5.3.3 Dual and Primal/Dual algorithms

Distributed dual decomposition

class disropt.algorithms.dual_decomp.DualDecomposition(agent, initial_condition, enable_log=False)

    Bases: disropt.algorithms.algorithm.Algorithm
    Distributed dual decomposition.

    From the perspective of agent $i$ the algorithm works as follows.

    Initialization: $\lambda_{ij}^0$ for all $j \in \mathcal{N}_i$

    For $k = 0, 1, \ldots$
    • Gather $\lambda_{ji}^k$ from neighbors $j \in \mathcal{N}_i$
    • Compute $x^k_i$ as an optimal solution of
      \[
      \min_{x_i \in X_i} f_i(x_i) + x_i^T \sum_{j \in \mathcal{N}_i} (\lambda_{ij}^k - \lambda_{ji}^k)
      \]
    • Gather $x^k_j$ from neighbors $j \in \mathcal{N}_i$
    • Update for all $j \in \mathcal{N}_i$
      \[
      \lambda_{ij}^{k+1} = \lambda_{ij}^k + \alpha^k (x^k_i - x^k_j)
      \]
    where $x_i \in \mathbb{R}^n$, $\lambda_{ij} \in \mathbb{R}^n$ for all $j \in \mathcal{N}_i$, $X_i \subseteq \mathbb{R}^n$ for all $i$ and $\alpha^k$ is a positive stepsize.

    The algorithm has been presented in ???.

    get_result()
        Return the current value primal solution and dual variable

        Returns value of primal solution (np.ndarray), dual variables (dictionary of np.ndarray)

        Return type tuple (primal, dual)

    iterate_run(stepsizes, **kwargs)
        Run a single iterate of the algorithm

    run(iterations=1000, stepsize=0.1, verbose=False, **kwargs)
        Run the algorithm for a given number of iterations

    Parameters

        • iterations (int) – Number of iterations. Defaults to 1000.
        • stepsize (Union[float, Callable]) – If a float is given as input, the stepsize is constant. If a function is given, it must take an iteration k as input and output the corresponding stepsize. Defaults to 0.1.
        • verbose (bool) – If True print some information during the evolution of the algorithm. Defaults to False.

    Raises
Distributed ADMM

class disropt.algorithms.admm.ADMM(agent, initial_lambda, initial_z, enable_log=False)

Bases: disropt.algorithms.algorithm.Algorithm

Distributed ADMM.

From the perspective of agent $i$, the algorithm works as follows.

Initialization: $\lambda_{i,j}^0$ for all $j \in \mathcal{N}_i$, $\lambda_{i,i}^0$ and $z_i^0$

For $k = 0, 1, \ldots$,

• Compute $x_i^k$ as the optimal solution of
  \[
  \min_{x_i \in X_i} f_i(x_i) + x_i^T \left( \sum_{j \in \mathcal{N}_i \cup \{i\}} \lambda_{i,j}^k \right) + \frac{\rho}{2} \sum_{j \in \mathcal{N}_i \cup \{i\}} \| x_i - z_j^k \|^2
  \]

• Gather $x_j^k$ and $\lambda_{j,i}^k$ from neighbors $j \in \mathcal{N}_i$

• Update $z_i^{k+1}$
  \[
  z_i^{k+1} = \frac{\sum_{j \in \mathcal{N}_i \cup \{i\}} x_j^k}{|\mathcal{N}_i| + 1} + \frac{\sum_{j \in \mathcal{N}_i \cup \{i\}} \lambda_{j,i}^k}{\rho(|\mathcal{N}_i| + 1)}
  \]

• Gather $z_j^{k+1}$ from neighbors $j \in \mathcal{N}_i$

• Update for all $j \in \mathcal{N}_i$
  \[
  \lambda_{i,j}^{k+1} = \lambda_{i,j}^k + \rho (x_i^k - z_j^{k+1})
  \]

where $x_i, z_i \in \mathbb{R}^n$, $\lambda_{i,j} \in \mathbb{R}^n$ for all $j \in \mathcal{N}_i \cup \{i\}$, $X_i \subseteq \mathbb{R}^n$ for all $i$ and $\rho$ is a positive penalty parameter.

The algorithm has been presented in ???.

get_result()

Return the current value primal solution, dual variable and auxiliary primal variable

Returns value of primal solution (np.ndarray), dual variables (dictionary of np.ndarray), auxiliary primal variable (np.ndarray)

Return type tuple (primal, dual, auxiliary)

initialize_algorithm()

Initializes the algorithm

iterate_run(rho, **kwargs)

Run a single iterate of the algorithm

run(iterations=1000, penalty=0.1, verbose=False, **kwargs)

Run the algorithm for a given number of iterations

Parameters

• iterations (int) – Number of iterations. Defaults to 1000.
• **penalty** (float) – ADMM penalty parameter. Defaults to 0.1.
• **verbose** (bool) – If True print some information during the evolution of the algorithm. Defaults to False.

Raises **TypeError** – The number of iterations must be an int

Return type: ndarray

Returns: return a tuple (x, lambda, z) with the sequence of primal solutions, dual variables and auxiliary primal variables if enable_log=True.

**Distributed dual subgradient method**

```python
class disropt.algorithms.dual_subgradient.DualSubgradientMethod(agent, initial_condition, initial_runavg=None, enable_log=False)
```

Bases: `disropt.algorithms.consensus.Consensus`

Distributed dual subgradient method.

From the perspective of agent \(i\) the algorithm works as follows. For \(k = 0, 1, \ldots\)

\[
y_i^k = \sum_{j=1}^{N} w_{ij} \lambda_j^k
\]

\(x_i^{k+1} \in \text{arg} \min_{x_i \in X_i} f_i(x_i) + g_i(x_i)\top y_i^k\)

\(\lambda_i^{k+1} = \Pi_{\lambda \geq 0}[y_i^k + \alpha^k g_i(x_i^{k+1})]\)

\(\hat{x}_i^{k+1} = \hat{x}_i^k + \frac{\alpha^k}{\sum_{r=0}^{k} \alpha^r} (x_i^{k+1} - \hat{x}_i^k)\)

where \(x_i, \hat{x}_i \in \mathbb{R}^{n_i}, \lambda_i, y_i \in \mathbb{R}^S, X_i \subset \mathbb{R}^{n_i}\) for all \(i\), \(\alpha^k\) is a positive stepsize and \(\Pi_{\lambda \geq 0}[]\) denotes the projection operator over the nonnegative orthant.

The algorithm has been presented in [FaMa17].

**Warning:** this algorithm is still under development

```python
get_result()
```

Return the actual value of dual and primal averaged variable

Returns: value of primal running average, value of dual variable

Return type: tuple (dual, primal) of numpy.ndarray

```python
run(iterations=1000, stepsize=0.1, verbose=False, callback_iter=None)
```

Run the algorithm for a given number of iterations

Parameters:

• **iterations** (int) – Number of iterations. Defaults to 1000.
• **stepsize** (Union[float, Callable]) – If a float is given as input, the stepsize is constant. If a function is given, it must take an iteration \(k\) as input and output the corresponding stepsize. Defaults to 0.1.
**verbose** (bool) – If True print some information during the evolution of the algorithm. Defaults to False.

**callback_iter** (Optional[Callable]) – callback function to be called at the end of each iteration. Must take an iteration k as input. Defaults to None.

**Raises**

- **TypeError** – The number of iterations must be an int
- **TypeError** – The stepsize must be a float or a callable

**Return type** ndarray

**Returns** return a tuple (lambda, x_hat) with the sequence of dual and primal estimates if enable_log=True.

### Distributed Primal Decomposition

```python
class disrupt.algorithms.primal_decomp.PrimalDecomposition(agent, initial_condition, enable_log=False):
    Bases: disrupt.algorithms.algorithm.Algorithm

Distributed primal decomposition.

From the perspective of agent \(i\) the algorithm works as follows.

Initialization: \(y_0^i\) such that \(\sum_{i=1}^N y_0^i = 0\)

For \(k = 0, 1, \ldots\)

- Compute \((x^k_i, \rho^k_i, \mu^k_i)\) as a primal-dual optimal solution of

- Gather \(\mu^k_j\) from \(j \in \mathcal{N}_i\) and update

\[
y^{k+1}_i = y^k_i + \alpha^k \sum_{j \in \mathcal{N}_i} (\mu^k_i - \mu^k_j)
\]

where \(x_i \in \mathbb{R}^{n_i}, \mu_i, y_i \in \mathbb{R}^S, X_i \subseteq \mathbb{R}^{n_i}\) for all \(i\) and \(\alpha^k\) is a positive stepsize.

The algorithm has been presented in ???.

**get_result** (return_runavg=False)

Return the current value of primal solution, allocation and cost

**Parameters**

- **return_runavg** (bool) – whether or not to return also running average of allocation. Defaults to False.

**Returns** value of primal solution, allocation, cost (if return_runavg = False) tuple (primal, allocation, allocation_avg cost) if return_runavg = True

**Return type** tuple (primal, allocation, cost) of numpy.ndarray

**iterate_run** (stepsize, M, update_runavg, event, **kwargs)

Run a single iterate of the algorithm

**Parameters**

- **stepsize** (float)
- **M** (float)
- **update_runavg** (bool)
- **event**

**run** (iterations=1000, stepsize=0.1, M=1000.0, verbose=False, callback_iter=None, compute_runavg=False, runavg_start_iter=0, event=None, **kwargs)

Run the algorithm for a given number of iterations

**Parameters**

- **iterations** (int) – Number of iterations. Defaults to 1000.
• **stepsize** *(Union[float, Callable])* – If a float is given as input, the stepsize is constant. If a function is given, it must take an iteration k as input and output the corresponding stepsize. Defaults to 0.1.

• **M** *(float)* – Value of the parameter M. Defaults to 1000.

• **verbose** *(bool)* – If True print some information during the evolution of the algorithm. Defaults to False.

• **callback_iter** *(Optional[Callable])* – callback function to be called at the end of each iteration. Must take an iteration k as input. Defaults to None.

• **compute_runavg** *(bool)* – whether or not to compute also running average of allocation. Defaults to False.

• **runavg_start_iter** *(int)* – specifies when to start computing running average (applies only if compute_runavg = True). Defaults to 0.

**Raises**

• **TypeError** – The number of iterations must be an int

• **TypeError** – The stepsize must be a float or a callable

• **TypeError** – The parameter M must be a float

**Return type** *ndarray*

**Returns** return a tuple (x, y, J) with the sequence of primal solutions allocation estimates and cost if enable_log=True. If compute_runavg=True, then return (x, y, y_avg, J)

**Distributed Primal Decomposition for MILPs**

```python
class disropt.algorithms.primal_decomp_milp.PrimalDecompositionMILP (agent, initial_condition, enable_log=False, restriction=None, finite_time_add_restriction=0)
```

Bases: *disropt.algorithms.primal_decomp.PrimalDecomposition*

Distributed primal decomposition for MILPs.

- **compute_restriction** *(iterations, graph_diam)*
- **compute_violating_y** *(allocation)*
- **iterate_run** *(stepsize, M, update_runavg, event, **kwargs)*
  Run a single iterate of the algorithm
- **mixed_integer_solution** ()
- **run** *(n_agents, iterations=1000, stepsize=0.1, M=1000.0, verbose=False, callback_iter=None, fast_mode=True, max_cutting_planes=1000, milp_solver=None, extra_allocation=None, use_runavg=False, runavg_start_iter=0, event=None, max_consensus_iterations=None, max_consensus_graph_diam=None, **kwargs)*
  Run the algorithm for a given number of iterations

**Parameters**
• **iterations** *(int)* – Number of iterations. Defaults to 1000.

• **stepsize** *(Union[float, Callable]*) – If a float is given as input, the stepsize is constant. If a function is given, it must take an iteration \( k \) as input and output the corresponding stepsize. Defaults to 0.1.

• **\( M \)** *(float)* – Value of the parameter \( M \). Defaults to 1000.

• **verbose** *(bool)* – If True print some information during the evolution of the algorithm. Defaults to False.

• **callback_iter** *(Optional[Callable]*) – callback function to be called at the end of each iteration. Must take an iteration \( k \) as input. Defaults to None.

• **fast_mode** *(bool)* – If True, a mixed-integer solution is computed at each iteration, otherwise only at the last iteration. Defaults to True.

• **max_cutting_planes** *(int)* – maximum number of cutting planes for local solver. Defaults to 1000.

• **milp_solver** *(Optional[str]*) – MILP solver to use. Defaults to None (use default solver).

**Raises**

• **TypeError** – The number of iterations must be an int

• **TypeError** – The stepsize must be a float or a callable

• **TypeError** – The parameter \( M \) must be a float

**Return type** `ndarray`

**Returns** return a tuple \((x, y)\) with the sequence of primal solutions and allocation estimates if `enable_log=True`. If `fast_mode=True`, the primal solutions are those of the convexified problem, otherwise they are the mixed-integer estimates.

### ASYMM (beta)

**class** `disropt.algorithms.asymm.ASYMM(agent, graph_diameter, initial_condition, enable_log=False, **kwargs)`

**Bases:** `disropt.algorithms.misc.AsynchronousLogicAnd`

Asynchronous Distributed Method of Multipliers [FaGa19b]

See [FaGa19b] for the details.

**Warning:** This algorithm is currently under development

**dual_update_step()**

**get_result()**

  Return the value of the solution

**primal_update_step()**

**reset_step()**

  Reset the matrix \( S \) and update \( e \)

**run**(running_time=10.0)

  Run the algorithm
Parameters `maximum_running_time` – Maximum running time. Defaults to 1.

Raises `TypeError` – maximum running time must be a float

References

5.3.4 Set Membership Algorithms

Distributed Set Membership

class `disropt.algorithms.setmembership.SetMembership`(`agent`, `initial_condition`, `enable_log=False`)

Bases: `disropt.algorithms.consensus.Consensus`

Distributed Set Membership Algorithm [FaGa18]

From the perspective of agent $i$ the algorithm works as follows. For $k = 0, 1, \ldots$

\[
X_{i}^{k+1} = X_{i}^{k} \cap M_{i}^{k+1}
\]

\[
z_{i}^{k} = \sum_{j=1}^{N} w_{ij} x_{j}^{k}
\]

\[
x_{i}^{k+1} = \Pi_{X_{i}^{k+1}}[z_{i}^{k}]
\]

where $x_{i}, z_{i} \in \mathbb{R}^{n}, M_{i}^{k+1}, X_{i}^{k+1} \subseteq \mathbb{R}^{n}$ are the current feasible (measurement) set and the feasible (parameter) set respectively, and $\Pi_{X}$ denotes the projection operator over the set $X$.

Parameters

- `agent (Agent)` – agent to execute the algorithm (must be a Agent)
- `initial_condition (numpy.ndarray)` – initial condition
- `enable_log (bool)` – True for enabling log

agent
agent to execute the algorithm

Type `Agent`

x0
initial condition

Type `numpy.ndarray`

x
current value of the local solution

Type `numpy.ndarray`

shape
shape of the variable

Type `tuple`

x_neigh
dictionary containing the local solution of the (in-)neighbors

Type `dict`

enable_log
True for enabling log

Type `bool`
**disropt Documentation, Release 0.1.8**

### Type

bool

### measure()()

Takes a new measurement and updates parameter_set

### set_measure_generator(generator)

set the measure generator

**Parameters**

generator (Callable) – measure generator

---

**Asynchronous Distributed Set Membership**

```python
class disropt.algorithms.setmembership.AsynchronousSetMembership(agent, initial_condition, **kwargs)
```

Bases: disropt.algorithms.consensus.AsynchronousConsensus

Asynchronous Distributed Set Membership Algorithm [FaGa19]

**Parameters**

- **agent** (Agent) – agent to execute the algorithm (must be a Agent)
- **initial_condition** (numpy.ndarray) – initial condition
- **enable_log** (bool) – True for enabling log

**agent**

agent to execute the algorithm

**Type**

Agent

**x0**

initial condition

**Type**

numpy.ndarray

**x**

current value of the local solution

**Type**

numpy.ndarray

**shape**

shape of the variable

**Type**

tuple

**x_neigh**

dictionary containing the local solution of the (in-)neighbors

**Type**

dict

**enable_log**

True for enabling log

**Type**

bool

**timestamp_sequence**

list of timestamps

**Type**

list

**measure()**

Takes a new measurement and updates parameter_set
set_measure_generator (generator)
set the measure generator

Parameters generator (Callable) – measure generator

References

5.3.5 Constraint Exchange Algorithms

Constraints Consensus

class disropt.algorithms.constraintexchange.ConstraintsConsensus (agent, enable_log=False)

Bases: disropt.algorithms.algorithm.Algorithm

Constraints Consensus Algorithm [NoBu11]
This algorithm solves convex and abstract programs in the form

\[ x \]
\[ B \]
\[ \text{shape} \]
\[ x_{\text{neigh}} \]
\[ \text{sequence}_x \]

Parameters

\begin{itemize}
\item agent (Agent) – agent to execute the algorithm
\item enable_log (bool) – True to enable log
\end{itemize}

compare_constr \((a, b)\)
Compare two constraints to check whether they are equal

compute_basis \((\text{constraints})\)
Compute a (minimal) basis for the given constraint list

constr_to_dict \((\text{constraints})\)
Convert constraint list to dictionary

dict_to_constr \((\text{dictio})\)
Convert dictionary to constraint list

5.3. Algorithms
get_basis()
Return agent basis

get_result()
Return the current value of x

Returns value of x

Return type numpy.ndarray

iterate_run(event)
Run a single iterate of the algorithm

run(iterations=100, verbose=False, event=None, **kwargs)
Run the algorithm for a given number of iterations

Parameters
- iterations (int, optional) – Number of iterations. Defaults to 100.
- verbose (bool) – If True print some information during the evolution of the algorithm. Defaults to False.

Returns ndarray

Returns the sequence of computed solutions if enable_log=True.

unique_constr(constraints)
Remove redundant constraints from given constraint list

Distributed Simplex

class disrupt.algorithms.constraintexchange.DistributedSimplex(agent, problem_size=None, local_indices=None, enable_log=False, stop_iterations=None, big_M=500.0)

Bases: disrupt.algorithms.algorithm.Algorithm

Distributed Simplex Algorithm [BuNo11]
This algorithm solves linear programs in standard form. When reading the variable agent.problem.constraints, this class only considers equality constraints. Other constraints are discarded.

x
current value of the complete solution

Type numpy.ndarray

J
current value of the cost

Type float

x_basic
current value of the basic solution

Type numpy.ndarray
B
  basis associated to the local solution
  Type numpy.ndarray

n_constr
  number of constraints of the problem
  Type tuple

B_neigh
  dictionary containing the local bases of (in-)neighbors
  Type dict

A_init
  initial constraint matrix
  Type numpy.ndarray

b_init
  initial constraint vector
  Type numpy.ndarray

c_init
  initial cost vector
  Type numpy.ndarray

sequence_x
  sequence of solutions
  Type numpy.ndarray

sequence_J
  sequence of costs
  Type numpy.ndarray

Parameters

- agent (Agent) – agent to execute the algorithm
- problem_size (list) – total number of variables in the network. Defaults to None. If both problem_size and local_indices is provided, the complete solution vector will be computed.
- local_indices (list) – indices of the agent’s variables in the network, starting from 0. Defaults to None. If both problem_size and local_indices is provided, the complete solution vector will be computed.
- enable_log (bool) – True to enable log
- stop_iterations (int) – iterations with constant solution to stop algorithm. Defaults to None (disabled).
- big_M (float) – cost of big-M variables. Defaults to 500.

check_index_consistency ()
  Check consistency of local indices, problem_size and constraint matrix

get_basis ()
  Return current basis

5.3. Algorithms
get_result ()
Return the current value of the solution

Returns value of primal solution, primal basic solution, dual solution, cost

Return type tuple of nd.ndarray (primal, primal_basic, dual, cost)

initialize ()
Evaluate a first solution and basis starting from agent’s constraints through the Big-M method

iterate_run (event)
Run a single iterate of the algorithm

read_problem_data ()
Read local problem data from agent.problem. The data is saved in order to be solved as a standard form problem.

run (iterations=100, verbose=False, event=None, **kwargs)
Run the algorithm for a given number of iterations

Parameters
  * iterations (int, optional) – Maximum number of iterations. Defaults to 100.
  * verbose (bool) – If True print some information during the evolution of the algorithm. Defaults to False.

Returns return a tuple (x, J) with the sequence of solutions and costs if enable_log=True.

Dual Distributed Simplex

class disrupt.algorithms.constraintexchange.DualDistributedSimplex (agent, num_constraints=None, local_indices=None, enable_log=False, stop_iterations=None, big_M=500.0)

Bases: disrupt.algorithms.constraintexchange.DistributedSimplex

Distributed Simplex Algorithm on dual problem [BuNo11]

This algorithm solves linear programs of the form This class runs the Distributed Simplex algorithm on the (standard form) dual problem.

Parameters
  * agent (Agent) – agent to execute the algorithm
  * num_constraints (list) – total number of constraints in the network. Defaults to None. If both num_constraints and local_indices is provided, the complete dual solution vector will be computed.
  * local_indices (list) – indices of the agent’s constraints in the network, starting from 0. Defaults to None. If both num_constraints and local_indices is provided, the complete dual solution vector will be computed.
• **enable_log** (*bool*) – True to enable log
• **stop_iterations** (*int*) – iterations with constant solution to stop algorithm. Defaults to None (disabled).
• **big_M** (*float*) – cost of big-M variables. Defaults to 500.

**check_index_consistency** ()
Check consistency of local indices, num_constraints and constraint matrix

**get_result** ()
Return the current value of the solution

  **Returns**
  value of primal solution, dual basic solution, dual solution, cost

  **Return type**
  tuple of nd.ndarray (primal, dual_basic, dual, cost)

**read_problem_data** ()
Read local problem data from agent.problem. The data is saved in order to be solved as a standard form problem.

**References**

5.3.6 Miscellaneous Algorithms

**Distributed Logic-And**

**class** `disropt.algorithms.misc.LogicAnd` *(agent, graph_diameter, flag=False, enable_log=False, **kwargs)*

Logic-And algorithm. It can be used for checking in a distributed way if a certain condition (corresponding to flag=True in the algorithm) is satisfied by all the agents in the network. Details can be found in [FaGa19a]

  **Parameters**

  • **agent** (*Agent*) – Agent
  • **graph_diameter** (*int*) – diameter of the graph representing the network
  • **flag** (*bool*, *optional*) – local flag value. Defaults to False.
  • **enable_log** (*bool*, *optional*) – True for enabling log. Defaults to False.

**change_flag** (*new_flag*)
Change the local flag

  **Parameters**
  new_flag (*bool*) – new flag

**check_stop** ()
Check the last row of S

  **Returns**
  True if last row contains only ones. Meaning that all have the flag True

  **Return type**
  bool

**force_matrix_update** ()
Force the matrix S to have all ones in the last row

**iterate_run** ()
Run an iterate

**matrix_reset** ()
matrix_update()
    Update the matrix S

run (maximum_iterations=100, verbose=False)
    Run the algorithm

    Parameters
    • maximum_iterations (int) – Maximum number of iterations. Defaults to 100.
    • verbose (bool) – If True print some information during the evolution of the algorithm. Defaults to False.

    Raises TypeError – maximum iterations must be an int

update_column (neighbor, column)
    Update a column of the matrix corresponding to a neighbor

    Parameters
    • neighbor (Any) – neighbor
    • column (ndarray) – column value

    Raises
    • TypeError – second argument must be a numpy.ndarray with shape (graph_diameter, )
    • ValueError – second argument must be a numpy.ndarray with shape (graph_diameter, )

Asynchronous Distributed Logic-And

class disrupt.algorithms.misc.AsynchronousLogicAnd (agent, graph_diameter, flag=False, enable_log=False, **kwargs)

    Bases: disrupt.algorithms.misc.LogicAnd

    Asynchronous Logic-And algorithm. It can be used for checking in a distributed way if a certain condition (corresponding to flag=True in the algorithm) is satisfied by all the agents in the network. Details can be found in [FaGa19a]

    Parameters
    • agent (Agent) – Agent
    • graph_diameter (int) – diameter of the graph representing the network
    • flag (bool, optional) – local flag value. Defaults to False.
    • enable_log (bool, optional) – True for enabling log. Defaults to False.

iterate_run()
    Run an iterate

run (maximum_running_time=1)
    Run the algorithm

    Parameters maximum_running_time (float) – Maximum running time. Defaults to 1.

    Raises TypeError – maximum running time must be a float
Distributed Max-Consensus

```python
class disropt.algorithms.misc.MaxConsensus(agent, x0, graph_diameter=None, enable_log=False, **kwargs):
    Bases: disropt.algorithms.algorithm.Algorithm

    Max-Consensus algorithm. It computes the entry-wise maximum of a numpy array by using only neighboring
    communication.

    Parameters
    • agent (Agent) – Agent
    • x0 (np.ndarray) – local initial condition
    • graph_diameter (int, optional) – diameter of the graph representing the network
    • enable_log (bool, optional) – True to enable log. Defaults to False.

    get_result()
    Return the value of the solution

    iterate_run()
    Run an iterate

    run(iterations=100, verbose=False)
    Run the algorithm

    Parameters
    • iterations (int) – Maximum number of iterations. Defaults to 100.
    • verbose (bool) – If True print some information during the evolution of the algorithm. 
      Defaults to False.

    Raises TypeError – maximum iterations must be an int
```

References

Algorithm

```python
class disropt.algorithms.Algorithm(agent, enable_log=False, **kwargs):
    Bases: object

    Algorithm abstract class

    Parameters
    • agent (Agent) – agent to execute the algorithm
    • enable_log (bool) – True for enabling log

    agent
    agent to execute the algorithm
    Type Agent

    sequence
    sequence of data generated by the algorithm
    Type numpy.ndarray

    enable_log
    True for enabling log
```

5.3. Algorithms
disropt Documentation, Release 0.1.8

**5.4 Functions**

Functions are used to represent objective functions and constraints in optimization problems (see the tutorial *Objective functions and constraints*).

### 5.4.1 Abstract Function

AbstractFunction

```python
class disropt.functions.abstract_function.AbstractFunction
    Bases: object

    AbstractFunction class. This should be the parent of all specific (objective) functions.

    input_shape
        shape of the input of the function
        Type tuple

    output_shape
        shape of the output of the function
        Type tuple

    differentiable
        True if the function is differentiable
        Type bool

    affine
        True if the function is affine
        Type bool

    quadratic
        True if the function is quadratic
        Type bool

    eval(x)
        Evaluate the function at a point x
        Parameters x (ndarray) – input point
        Return type ndarray

    jacobian(x, **kwargs)
        Evaluate the jacobian of the function at a point x
        Parameters x (ndarray) – input point
        Return type ndarray
```

5.4 Functions

Functions are used to represent objective functions and constraints in optimization problems (see the tutorial Objective functions and constraints).

5.4.1 Abstract Function

AbstractFunction

```python
class disropt.functions.abstract_function.AbstractFunction
    Bases: object

    AbstractFunction class. This should be the parent of all specific (objective) functions.

    input_shape
        shape of the input of the function
        Type tuple

    output_shape
        shape of the output of the function
        Type tuple

    differentiable
        True if the function is differentiable
        Type bool

    affine
        True if the function is affine
        Type bool

    quadratic
        True if the function is quadratic
        Type bool

    eval(x)
        Evaluate the function at a point x
        Parameters x (ndarray) – input point
        Return type ndarray

    jacobian(x, **kwargs)
        Evaluate the jacobian of the function at a point x
        Parameters x (ndarray) – input point
        Return type ndarray
```
subgradient \((x, **kwargs)\)
Evaluate the subgradient of the function at a point \(x\)

**Parameters**
- \(x\) (ndarray) – input point

**Raises** `ValueError` – subgradient is defined only for functions with scalar output

**Return type** ndarray

get_parameters()

hessian \((x, **kwargs)\)
Evaluate the hessian of the function at a point \(x\)

**Parameters**
- \(x\) (ndarray) – input point

**Return type** ndarray

property is_affine
property is_differentiable
property is_quadratic

### 5.4.2 Basic Functions

Here is the list of the implemented basic mathematical functions.

**Variable**

```python
class disropt.functions.variable.Variable(n)
```

**Bases:** `disropt.functions.affine_form.AffineForm`

Variable, basic function

\[ f(x) = x \]

with \(x \in \mathbb{R}^n\)

**Parameters**
- \(n\) (int) – dimension of the decision variable: \((n,1)\)

**Raises** `TypeError` – input dimension must be an int

eval \((x)\)
Evaluate the function at a point \(x\)

**Parameters**
- \(x\) (ndarray) – input point

**Return type** ndarray

**AffineForm**

```python
class disropt.functions.affine_form.AffineForm(fn, A=None, b=None)
```

**Bases:** `disropt.functions.abstract_function.AbstractFunction`

Makes an affine transformation

\[ f(x) = \langle A, x \rangle + b = A^T x + b \]

with \(A \in \mathbb{R}^{n \times m}, b \in \mathbb{R}^m\) and \(x : \mathbb{R}^n\). It can also be instantiated as:
\[ A \odot x + b \]

Parameters

- **\(fn\)** ([AbstractFunction]) – input function
- **\(A\)** ([numpy.ndarray]) – input matrix
- **\(b\)** ([numpy.ndarray]) – input bias

Raises

- **TypeError** – first argument must be a AbstractFunction object
- **TypeError** – second argument must be numpy.ndarray
- **ValueError** – the number of columns of \(A\) must be equal to the number of rows of the output of \(fn\) –

\[ \text{eval}(x) \]
Evaluate the function at a point \(x\)

Parameters **\(x\)** ([ndarray]) – input point

Return type **ndarray**

\[ \text{get_parameters}() \]

QuadraticForm

class disropt.functions.quadratic_form.QuadraticForm(fn, P=None, q=None, r=None)

Bases: disropt.functions.abstract_function.AbstractFunction

Quadratic form

\[ f(x) = x^T P x + q^T x + r \]

with \(P \in \mathbb{R}^{n \times n}, q \in \mathbb{R}^n, r \in \mathbb{R}\) and \(x : \mathbb{R}^n\).

Parameters

- **\(fn\)** ([AbstractFunction]) – input function
- **\(P\)** ([numpy.ndarray, optional]) – input matrix. Defaults to None (identity).
- **\(q\)** ([numpy.ndarray, optional]) – input vector. Defaults to None (zero).
- **\(r\)** ([numpy.ndarray, optional]) – input bias. Defaults to None (zero).

Raises

- **TypeError** – First argument must be a AbstractFunction object
- **TypeError** – Second argument must be a numpy.ndarray
- **ValueError** – Input matrix must be a square matrix
- **ValueError** – Dimension mismatch. Input matrix must have shape compliant with the function output shape

\[ \text{eval}(x) \]
Evaluate the function at a point \(x\)
Parameters \textbf{x} (ndarray) – input point

Return type ndarray

get_parameters()
SquaredNorm

```python
class disropt.functions.squared_norm.SquaredNorm(fn, order=None, axis=None)
Bases: disropt.functions.abstract_function.AbstractFunction

Squared norm (supported norms are 1, 2, inf)

\[ f(x) = \|x\|^2 \]

with \( x \in \mathbb{R}^n \).

Parameters
- \texttt{fn} (AbstractFunction) – input function
- \texttt{order} (int, optional) – order of the norm. Can be 1, 2 or np.inf. Defaults to 2.

Raises
- TypeError – input must be a function object
- NotImplementedError – only 1, 2 and inf norms are currently supported

\texttt{eval(x)}
Evaluate the function at a point \( x \)

Parameters
- x (ndarray) – input point

Return type
ndarray
```

Log

```python
class disropt.functions.log.Log(fn)
Bases: disropt.functions.abstract_function.AbstractFunction

Natural log function (elementwise)

\[ f(x) = \log(x) \]

with \( x \in \mathbb{R}^n \).

Parameters
- \texttt{fn} (AbstractFunction) – input function

Raises
- TypeError – input must be a AbstractFunction object

\texttt{eval(x)}
Evaluate the function at a point \( x \)

Parameters
- x (ndarray) – input point

Return type
ndarray
```

Exp

```python
class disropt.functions.exp.Exp(fn)
Bases: disropt.functions.abstract_function.AbstractFunction

Exponential function (elementwise)

\[ f(x) = e^x \]

with \( x \in \mathbb{R}^n \).
```
Parameters \( fn \) (AbstractFunction) – input function

Raises \texttt{TypeError} – input must be a AbstractFunction object

\texttt{eval}(x)
Evaluate the function at a point \( x \)

Parameters \( x \) (ndarray) – input point

Return type ndarray

Logistic

\begin{verbatim}
class disropt.functions.logistic.Logistic(fn)
    Bases: disropt.functions.abstract_function.AbstractFunction

    Logistic function (elementwise)
    \[ f(x) = \log(1 + e^x) \]
    with \( x \in \mathbb{R}^n \).

    Parameters \( fn \) (AbstractFunction) – input function
    Raises \texttt{TypeError} – input must be a AbstractFunction object

    \texttt{eval}(x)
    Evaluate the function at a point \( x \)

    Parameters \( x \) (ndarray) – input point
    Return type ndarray
\end{verbatim}

Min

\begin{verbatim}
class disropt.functions.min.Min(f1, f2)
    Bases: disropt.functions.max.Max

    Min function (elementwise)
    \[ f(x, y) = \min(x, y) \]
    with \( x, y \in \mathbb{R}^n \).

    Parameters
    \begin{itemize}
      \item \( f1 \) (AbstractFunction) – input function
      \item \( f2 \) (AbstractFunction) – input function
    \end{itemize}

    Raises
    \begin{itemize}
      \item \texttt{ValueError} – input must be a AbstractFunction object
      \item \texttt{ValueError} – functions must have the same input/output shapes
    \end{itemize}

    \texttt{eval}(x)
    Evaluate the function at a point \( x \)

    Parameters \( x \) (ndarray) – input point
    Return type ndarray
\end{verbatim}

5.4. Functions
Max

```python
class disropt.functions.max.Max(f1, f2)
    Bases: disropt.functions.abstract_function.AbstractFunction

    Max function (elementwise)
    
    \[ f(x, y) = \max(x, y) \]
    with \( x, y : \mathbb{R}^n \).

    Parameters
    • \( f1 \) (AbstractFunction) – input function
    • \( f2 \) (AbstractFunction) – input function

    Raises
    • TypeError – input must be a AbstractFunction object
    • ValueError – functions must have the same input/output shapes

eval(x)
    Evaluate the function at a point x
    
    Parameters \( x \) (ndarray) – input point
    
    Return type ndarray
```

Square

```python
class disropt.functions.square.Square(fn)
    Bases: disropt.functions.abstract_function.AbstractFunction

    Square function (elementwise)
    
    \[ f(x) = x^2 \]
    with \( x : \mathbb{R}^n \).

    Parameters \( fn \) (AbstractFunction) – input function

    Raises TypeError – input must be a AbstractFunction object

eval(x)
    Evaluate the function at a point x
    
    Parameters \( x \) (ndarray) – input point
    
    Return type ndarray
```

Power

```python
class disropt.functions.power.Power(fn, exponent)
    Bases: disropt.functions.abstract_function.AbstractFunction

    Power function (elementwise)
    
    \[ f(x) = x^\alpha \]
    with \( x : \mathbb{R}^n, \alpha : \mathbb{R} \).
```
Parameters `fn` (AbstractFunction) – input function

Raises `TypeError` – input must be a AbstractFunction object

**eval**(*x*)
Evaluate the function at a point *x*

Parameters `x` (ndarray) – input point

Return type `ndarray`

### 5.4.3 Special Functions

**Stochastic Function**

class `disropt.functions.stochastic_function.StochasticFunction`(*fn_list*, `probabilities`)

Bases: `disropt.functions.abstract_function.AbstractFunction`

Stochastic function

\[
f(x) = \mathbb{E}[h(x)]
\]

with \(x : \mathbb{R}^n\).

The `random_batch` method extract a batch from the function and `batch_subgradient`, `batch_jacobian` and `batch_hessian` methods return a subgradient, jacobian and hessian computed at on the last batch.

Parameters

- `fn_list` *(list)* – list of AbstractFunction objects
- `probabilities` *(list, optional)* – list with the probabilities of drawing each function. Default is None which leads to uniform probabilities

Raises

- `TypeError` – `fn_list` input must be a list of functions
- `ValueError` – All functions must have the same input/output shape
- `TypeError` – probabilities argument must be a list of floats
- `ValueError` – inputs must have the same length
- `ValueError` – provided probabilities must sum to 1
- `NotImplementedError` – only 1, 2 and inf norms are currently supported

**eval**(*x*)
Evaluate the function at a point *x*

Parameters `x` (ndarray) – input point

Return type `ndarray`

**jacobian**(*x*, **kwargs)
Evaluate the jacobian of the the function at a point *x*

Parameters `x` (ndarray) – input point

Return type `ndarray`
subgradient\( (x, **kwargs) \)
Evaluate the subgradient of the function at a point \( x \)

**Parameters**
- \( x \) (ndarray) – input point

**Raises**
- ValueError – subgradient is defined only for functions with scalar output

**Return type**
- ndarray

batch_hessian\( (x) \)
evaluate the hessian on the current batch

**Parameters**
- \( x \) (np.ndarray) – point

**Returns**
- hessian

**Return type**
- numpy.ndarray

batch_jacobian\( (x) \)
evaluate the jacobian on the current batch

**Parameters**
- \( x \) (np.ndarray) – point

**Returns**
- jacobian

**Return type**
- numpy.ndarray

batch_subgradient\( (x) \)
evaluate the subgradient on the current batch

**Parameters**
- \( x \) (np.ndarray) – point

**Raises**
- ValueError – Only functions with scalar output have a subgradient

**Returns**
- subgradient

**Return type**
- numpy.ndarray

random_batch \( (batch_size=1) \)
generate a random batch from the function.

**Parameters**
- batch_size (int, optional) – batch size. Defaults to 1.

### Function With Extended Variable

class disrupt.functions.extended_function.ExtendedFunction \( (fn, n\_var=1, axis=-1, pos=0) \)

**Bases:** disrupt.functions.abstract_function.AbstractFunction

Function with extended variable

\[ f(x, y) = x \]

with \( x \in \mathbb{R}^n, y \in \mathbb{R}^m \)

**Parameters**
- \( fn \) (AbstractFunction) – input function
- \( n\_var \) (int) – number of additional variables. Defaults to 1
- \( axis \) (int) – axis along which the additional variables are appended. Defaults to -1 (the last valid one)
- \( pos \) (int) – position index of the old variable vector. Defaults to 0
Raises

- `TypeError` – fn must be a AbstractFunction
- `TypeError` – n_var must be a positive int
- `TypeError` – axis must be int

### 5.4.4 SubmodularFn

class disropt.functions.submodular_func.SubmodularFn(input_shape)

```python
Bases: disropt.functions.abstract_function.AbstractFunction
```

Submodular AbstractFunction abstract class

**V**

ground set

**input_shape**

cardinality of the ground set

#### Parameters

**input_shape** – cardinality of the ground set

#### Raises

`ValueError` – input_shape must be an integer positive number

#### greedy_polyhedron(self, w)

Greedy algorithm for finding a maximizer $x$ of

$$
\max_{x \in B(F)} w^T x
$$

(5.3)

where $B(F)$ is the base polyhedron associated to the submodular function $F$

##### Parameters

**w** – cost direction

##### Returns

maximizer of (5.3)

##### Return type

`numpy.ndarray`

#### Raises

`ValueError` – Input must be a `numpy.ndarray` with input_shape elements

#### subgradient(self, x)

Evaluate a subgradient of the Lovasz extension of the submodular function at $x$

##### Parameters

**x** – vector

##### Returns

subgradient of the Lovasz extension of $F$ at $x$

##### Return type

`numpy.ndarray`

#### Raises

`ValueError` – Input must be a `numpy.ndarray` with input_shape elements

#### blocksubgradient(x, block)

Evaluate a subgradient of the Lovasz extension of the submodular function at $x$

##### Parameters

- **x** – vector
- **block** – vector

##### Returns

block subgradient of the Lovasz extension of $F$ at $x$

##### Return type

`numpy.ndarray`

#### Raises

`ValueError` – Input must be a `numpy.ndarray` with input_shape elements
eval(set)
Evaluate the submodular function at a given set

Parameters
- set (numpy.ndarray (dtype='int')) – vector

5.5 Constraints

5.5.1 Generic constraints

AbstractConstraint (Abstract class)

class disropt.constraints.constraints.AbstractConstraint
Bases: object

Abstract class for expressing constraints
eval()

Constraint

class disropt.constraints.constraints.Constraint (fn, sign='==')
Bases: disropt.constraints.constraints.AbstractConstraint

Constraint build from a AbstractFunction object. Constraints are represented in the canonical forms $f(x) = 0$ and $f(x) \leq 0$.

Parameters
- fn (AbstractFunction) – constraint function
- sign (bool) – type of constraint: “==”, “<” or “>”

fn
constraint function

Type AbstractFunction

sign
type of constraint: “==”, “<” or “>”

Type bool

input_shape
input space dimensions

Type tuple

output_shape
output space dimensions

Type tuple

eval(x)
Evaluate the constraint function at a point x

Parameters
- x (ndarray) – input point

Return type

bool

property function
get_parameters()
Return the parameters of the function if it is affine or quadratic

Returns  A, b for affine constraints, P, q, r for quadratic

Return type  tuple

property is_affine
Return true if the function is affine.

Returns  true if the function is affine

Return type  bool

property is_equality

property is_inequality

property is_quadratic
Return true if the function is affine.

Returns  true if the function is affine

Return type  bool

projection(x)
Compute the projection of a point onto the set defined by the constraint. The constraint should be convex.

Parameters  x (ndarray) – point to be projected

Returns  projected point

Return type  numpy.ndarray

ExtendedConstraint

class  disropt.constraints.extended_constraint.ExtendedConstraint (fn, sign=’==’)
Bases: disropt.constraints.constraints.Constraint

Constraint with extended variable

Parameters

• constr (Constraint or list of Constraint) – original constraint(s)
• n_var – number of additional variables. Defaults to 1
• axis – axis along which the additional variables are appended. Defaults to -1 (the last valid one)
• pos – position index of the old variable vector. Defaults to 0

Raises

• TypeError – fn must be a Constraint object or a list of Constraint objects
• TypeError – n_var must be a positive int
• TypeError – axis must be int

5.5. Constraints
5.5.2 Sets (with project method)

AbstractSet

```python
class disropt.constraints.projection_sets.AbstractSet
    Bases: object
    Abstract constraint set

    projection(x)
    Project a point onto the set

    to_constraints()
    Convert the set in a list of Constraints
```

Box

```python
class disropt.constraints.projection_sets.Box(lower_bound=None, upper_bound=None)
    Bases: disropt.constraints.projection_sets.AbstractSet
    Box set
    \( X = \{ x | l \leq x \leq u \} \) with \( l, x, u \in \mathbb{R}^n \)

    Parameters
    • **lower_bound** (*numpy.ndarray*, *optional*) – array with lower bounds for each axis. Defaults to None.
    • **upper_bound** (*numpy.ndarray*, *optional*) – array with upper bounds for each axis. Defaults to None.

    lower_bound
    lower bounds

    Type *numpy.ndarray*

    upper_bound
    upper bounds

    Type *numpy.ndarray*

    input_shape
    space dimensions

    Type *tuple*

    intersection(box)
    Compute the intersection with another box

    Parameters box (Box) – box to compute the intersection with

    Raises
    • **ValueError** – Only intersection with another box is supported
    • **ValueError** – The two boxex must have the same input_shape

    projection(x)
    Project a point onto the box

    Parameters x (*numpy.ndarray*) – Point to be projected
Returns projected point

Return type numpy.ndarray

to_constraints()
Convert the set in a list of Constraints

Strip

class disropt.constraints.projection_sets.Strip(regressor, shift, width)
Bases: disropt.constraints.projection_sets.AbstractSet

Strip set
\[ X = \{ x \mid -w \leq |a^\top x - s| \leq w \} \]
with \( a, x \in \mathbb{R}^n \) and \( w \in \mathbb{R} \)

Parameters

- **regressor** (numpy.ndarray) – regressor of the strip
- **shift** (float) – shift from the origin
- **width** (float) – width of the strip

regressor
regressor of the strip

Type numpy.ndarray

shift
shift from the origin

Type float

upper
upper border (shift + half width)

Type float

lower
lower border (shift - half width)

Type float

input_shape
space dimensions

Type tuple

intersection(strip)
Compute the intersection with another strip

Parameters **strip** (strip) – strip to compute the intersection with

Raises

- **ValueError** – Only intersection with another strip is supported
- **ValueError** – The two strips must have the same regressor

projection(x)
Project a point onto the strip

Parameters **x** (numpy.ndarray) – Point to be projected

Returns projected point

5.5. Constraints
Return type  numpy.ndarray
to_constraints()  
Convert the set in a list of Constraints

Circle
class disropt.constraints.projection_sets.Circle(center, radius)
Bases: disropt.constraints.projection_sets.AbstractSet
Circle set
X = \{x \mid \|x - c\| \leq r\} \text{ with } x, c \in \mathbb{R}^n \text{ and } r \in \mathbb{R}

Parameters
• center (numpy.ndarray) – center of the circle
• radius (float) – radius of the circle
center
center of the circle
Type  numpy.ndarray
radius
radius of the circle
Type  float
input_shape
space dimensions
Type  tuple
intersection(circle)
Compute the intersection with another circle
Parameters circle (Circle) – circle to compute the intersection with
Raises
• ValueError – Only intersection with another circle is supported
• ValueError – The two circles must have the same center
projection(x)
Project a point onto the circle
Parameters x (numpy.ndarray) – Point to be projected
Returns  projected point
Return type  numpy.ndarray
to_constraints()  
Convert the set in a list of Constraints
CircularSector

class disropt.constraints.projection_sets.CircularSector(vertex, angle, radius, width)

Circular sector set.

Parameters

• vertex (numpy.ndarray) – vertex of the circular sector
• angle (float) – direction of the circular sector (in rad)
• radius (float) – radius of the circular sector
• width (float) – width of the circular sector (in rad)

vertex

vertex of the circular sector

Type numpy.ndarray

angle

direction of the circular sector (in rad)

Type float

radius

radius of the circular sector

Type float

h_angle

left border (from the vertex pov) of the circular sector (in rad)

Type float

l_angle

right border (from the vertex pov) of the circular sector (in rad)

Type float

input_shape

space dimensions

Type tuple

intersection (circular_sector)

Compute the intersection with another circular sector

Parameters circular_sector (CircularSector) – circula sector to compute the intersec-
tion with

Raises

• ValueError – Only intersection with another circular_sector is supported
• ValueError – The two circular_sector must have the same vertex

projection (x)

Project a point onto the circular sector

Parameters x (numpy.ndarray) – Point to be projected

Returns projected point

Return type numpy.ndarray
to_constraints()

Convert the set in a list of Constraints

## 5.6 Problem Classes

### 5.6.1 Problem

class disrupt.problems.Problem(objective_function=None, constraints=None, force_general_problem=False, **kwargs)

**Bases:** object

A generic optimization problem.

- **minimize** \( f(x) \)
- subject to \( g(x) \leq 0 \)
- \( h(x) = 0 \)

**Parameters**

- **objective_function (AbstractFunction, optional)** – objective function. 
  Defaults to None.

- **constraints (list, optional)** – constraints. Defaults to None.

**objective_function**

Objective function to be minimized

- **Type** Function

**constraints**

constraints

- **Type** list

**input_shape**

dimension of optimization variable

- **Type** tuple

**output_shape**

output shape

- **Type** tuple

**add_constraint (fn)**

Add a new constraint

- **Parameters** fn (Union[AbstractSet, Constraint]) – new constraint

- **Raises** TypeError – constraints must be AbstractSet or Constraint

**project_on_constraint_set (x)**

Compute the projection of a point onto the constraint set of the problem

- **Parameters** x (ndarray) – point to project

- **Returns** projected point

- **Return type** numpy.ndarray

**set_objective_function (fn)**

Set the objective function
Parameters `fn` *(AbstractFunction)* – objective function

Raises **TypeError** – input must be a AbstractFunction object

```python
solve(solver='cvxpy', return_only_solution=True)
```

Solve the problem

Returns `solution`

Return type `numpy.ndarray`

### 5.6.2 LinearProblem

```python
class disropt.problems.LinearProblem(objective_function, constraints=None, **kwargs)
```

Bases: `disropt.problems.linear_problem.Problem`

Solve a Linear programming problem defined as:

\[
\begin{align*}
\text{minimize } & c^\top x \\
\text{subject to } & Gx \leq h \\
& Ax = b
\end{align*}
\]

```python
set_objective_function(objective_function)
```

set the objective function

Parameters `objective_function` *(AffineForm)* – objective function

Raises **TypeError** – Objective function must be a AffineForm with output_shape=(1,1)

```python
solve(initial_value=None, solver='glpk', return_only_solution=True)
```

Solve the problem

Parameters

- `initial_value` *(numpy.ndarray), optional* – Initial value for warm start. Defaults to None.
- `solver` *(str, optional)* – Solver to use ['glpk', 'cvxopt']. Defaults to 'glpk'.

Raises **ValueError** – Unsupported solver

Returns `solution`

Return type `numpy.ndarray`

### 5.6.3 MixedIntegerLinearProblem

```python
class disropt.problems.MixedIntegerLinearProblem(objective_function, constraints=None, integer_vars=None, binary_vars=None, **kwargs)
```

Bases: `disropt.problems.linear_problem.LinearProblem`

Solve a Mixed-Integer Linear programming problem defined as:

\[
\begin{align*}
\text{minimize } & c^\top x \\
\text{subject to } & Gx \leq h \\
& Ax = b \\
& x_k \in \mathbb{Z}, \forall k \in I \\
& x_k \in \{0,1\}, \forall k \in B
\end{align*}
\]
where $I$ is the set of integer variable indexes and $B$ is the set of binary variable indexes.

```python
def solve(initial_value=None, solver='glpk', return_only_solution=True)
    Solve the problem

    Parameters
    • initial_value (numpy.ndarray, optional) – Initial value for warm start.
      Defaults to None. Not available in GLPK
    • solver (str, optional) – Solver to use ['glpk', 'gurobi']. Defaults to 'glpk'.

    Raises ValueError – Unsupported solver

    Returns solution

    Return type numpy.ndarray
```

## 5.6.4 ConvexifiedMILP

```python
class ConvexifiedMILP(disropt.problems.milp.MixedIntegerLinearProblem,
                       objective_function, y, A, constraints=None, integer_vars=None,
                       binary_vars=None, **kwargs)
Bases: disropt.problems.milp.MixedIntegerLinearProblem

Solve a convexified Mixed-Integer Linear Problem of the form:

$$
\begin{align*}
\text{minimize} & \quad c^\top x + M \rho \\
\text{subject to} & \quad x \in \text{conv}(X), \quad \rho \geq 0 \\
& \quad Ax \leq y + \rho 1
\end{align*}
$$

where the set $X$ is a compact mixed-integer polyhedral set defined by equality and inequality constraints. Moreover, $\rho$ is a scalar, $y \in \mathbb{R}^m$ is a vector and $A \in \mathbb{R}^{m \times n}$ is a constraint matrix.

```python
def solve(M=None, milp_solver=None, initial_dual_solution=None, return_only_solution=True, y=None,
          max_cutting_planes=None, cutting_planes=None, threshold_convergence=1e-08,
          max_iterations=1000)
    Solve the problem using a custom dual cutting-plane algorithm

    Parameters
    • M (float) – value of the parameter $M$
    • milp_solver (str, optional) – MILP solver to use. Defaults to None (use default solver).
    • initial_dual_solution (numpy.ndarray, optional) – Initial dual value for warm start. Defaults to None.
    • return_only_solution (bool, optional) – if True, returns only solution, otherwise returns more information. Defaults to True.
    • y (np.ndarray, optional) – value to override the current $y$. Defaults to None (keep current value).
    • max_cutting_planes (int, optional) – maximum number of stored cutting planes. Defaults to None (disabled).
    • cutting_planes (numpy.ndarray, optional) – cutting planes for warm start previously returned by this function. Defaults to None.
    • threshold_convergence (float, optional) – threshold for convergence detection. Defaults to 1e-8.
• **max_iterations**(int, optional) – maximum number of iterations performed by algorithm. Defaults to 1e3.

Returns primal solution tuple (x, rho) if return_only_solution = True, otherwise (primal solution tuple, optimal cost, dual solution, cutting planes, number of iterations)

Return type tuple

### 5.6.5 QuadraticProblem

class disropt.problems.QuadraticProblem(objective_function, constraints=None, is_pos_def=True, **kwargs)

Bases: disropt.problems.problem.Problem

Solve a Quadratic programming problem defined as:

$$
\begin{align*}
\text{minimize} & \quad x^T P x + q^T x + r \\
\text{subject to} & \quad G x \leq h \\
& \quad A x = b
\end{align*}
$$

Quadratic problems are currently solved by using CVXOPT or OSQP [https://osqp.org](https://osqp.org).

Parameters

• **objective_function**(QuadraticForm) – objective function

• **constraints**(list, optional) – list of constraints. Defaults to None.

• **is_pos_def**(bool) – True if P is (semi)positive definite. Defaults to True.

set_constraints(constraints)

Set the constraints

Parameters constraints(list) – list of constraints

Raises TypeError – a list of affine Constraints must be provided

set_objective_function(objective_function)

set the objective function

Parameters objective_function(QuadraticForm) – objective function

Raises TypeError – Objective function must be a QuadraticForm

solve(initial_value=None, solver='osqp', return_only_solution=True)

Solve the problem

Parameters

• **initial_value**(numpy.ndarray), optional) – Initial value for warm start. Defaults to None.

• **solver**(str, optional) – Solver to use (‘osqp’ or ‘cvxopt’). Defaults to ‘osqp’.

Raises ValueError – Unsupported solver, only ‘osqp’ and ‘cvxopt’ are currently supported

Returns solution

Return type numpy.ndarray

5.6. Problem Classes
5.6.6 ProjectionProblem

class disropt.problems.ProjectionProblem(*args, **kwargs)
    Bases: disropt.problems.problem.Problem

    Computes the projection of a point onto some constraints, i.e., it solves
    \[
    \begin{align*}
    & \text{minimize} \quad \frac{1}{2} \|x - p\|^2 \\
    & \text{subject to} \quad f_k(x) \leq 0, \ k = 1, \ldots
    \end{align*}
    \]

    Parameters
    
    - **constraints_list** (list) – list of constraints
    - **point** (numpy.ndarray) – point \( p \) to project

    solve()
    solve the problem

    Returns
    solution

    Return type
    numpy.ndarray

5.6.7 ConstraintCoupledProblem

class disropt.problems.ConstraintCoupledProblem(objective_function=None, constraints=None, coupling_function=None, **kwargs)
    Bases: disropt.problems.problem.Problem

    A local part of a constraint-coupled problem.

    Parameters
    
    - **objective_function** (AbstractFunction, optional) – Local objective function. Defaults to None.
    - **constraints** (list, optional) – Local constraints. Defaults to None.
    - **coupling_function** (AbstractFunction, optional) – Local function contributing to coupling constraints. Defaults to None.

    objective_function
    Objective function to be minimized

    Type
    Function

    constraints
    Local constraints

    Type
    AbstractSet or Constraint

    coupling_function
    Local function contributing to coupling constraints

    Type
    Function

    set_coupling_function(fn)
    Set the coupling constraint function

    Parameters
    fn (AbstractFunction) – coupling constraint function

    Raises
    TypeError – input must be a AbstractFunction
5.6.8 ConstraintCoupledMILP

class disropt.problems.ConstraintCoupledMILP(objective_function, coupling_function, constraints=None, integer_vars=None, binary_vars=None, **kwargs)


5.7 Utilities

5.7.1 Graph generation

MPIgraph

class disropt.utils.graph_constructor.MPIgraph(graph_type=None, in_weight_matrix_type=None, out_weight_matrix_type=None, **kwargs)

Bases: object

Create a graph on the network

Parameters

- graph_type (str, optional) – type of graph (‘complete’, ‘random_binomial’). Defaults to None (complete).

get_local_info()

return the local info available at the agent

Returns local_rank, in_neighbors, out_neighbors, in_weights, out_weights,

Return type tuple

ring_graph

disropt.utils.graph_constructor.ring_graph(N, link_type='undirected')

construct a ring graph

Parameters

- N (int) – number of agents
- link_type (str, optional) – ‘directed’ or ‘undirected’. Defaults to ‘undirected’.

Returns adjacency matrix

Return type numpy.ndarray
**binomial_random_graph**

disropt.utils.graph_constructor.binomial_random_graph(N, p=None, seed=None, link_type='undirected')

construct a random binomial graph

**Parameters**

- N (int) – number of agents
- p (float, optional) – link probability. Defaults to None (=1).
- seed (int, optional) – [description]. Defaults to None (=1).
- link_type (str, optional) – ‘directed’ or ‘undirected’. Defaults to ‘undirected’.

**Returns** adjacency matrix

**Return type** numpy.ndarray

**binomial_random_graph_sequence**

disropt.utils.graph_constructor.binomial_random_graph_sequence(under_adj, n_graphs, p=0.1, period=None, seed=None, link_type='undirected')

Construct a sequence of random binomial graphs starting from a given underlying graph

**Parameters**

- under_adj (2D numpy.ndarray) – Adjacency matrix of underlying graph.
- n_graphs (int) – number of graphs in the returned sequence.
- p (float or 2D numpy.ndarray, optional) – link probability. Defaults to None (=0.1).
- period (int, optional) – T-connectivity period of the returned sequence (obtained artificially with cycles). Defaults to None (no T-connectivity).
- seed (int, optional) – [description]. Defaults to None (=1).
- link_type (str, optional) – ‘directed’ or ‘undirected’. Defaults to ‘undirected’.

**Returns** sequence of adjacency matrices

**Return type** tuple(numpy.ndarray)

### 5.7.2 Weighted adjacency matrices

**metropolis_hastings**

disropt.utils.graph_constructor.metropolis_hastings(Adj, link_type='undirected')

Construct a weight matrix using the Metropolis-Hastings method

**Parameters** Adj (numpy.ndarray) – Adjacency matrix

**Returns** weighted adjacency matrix

**Return type** numpy.ndarray
row_stochastic_matrix

disropt.utils.graph_constructor.row_stochastic_matrix(Adj, weights_type='uniform')
Construct a row-stochastic weighted adjacency matrix

Parameters

• Adj (numpy.ndarray) – Adjacency matrix

Returns

weighted adjacency matrix

Return type

numpy.ndarray
column_stochastic_matrix

disropt.utils.graph_constructor.column_stochastic_matrix(Adj, weights_type='uniform')
Construct a column-stochastic weighted adjacency matrix

Parameters

• Adj (numpy.ndarray) – Adjacency matrix

Returns

weighted adjacency matrix

Return type

numpy.ndarray

5.7.3 Matrix properties

is_pos_def

disropt.utils.utilities.is_pos_def(P)
check if a matrix is positive definite

Parameters

• P (numpy.ndarray) – matrix

Returns

Return type bool

is_semi_pos_def

disropt.utils.utilities.is_semi_pos_def(P)
check if a matrix is positive semi-definite

Parameters

• P (numpy.ndarray) – matrix

Returns

Return type bool

check_symmetric

disropt.utils.utilities.check_symmetric(A, rtol=1e-05, atol=1e-08)
check if a matrix is symmetric

Parameters

• A (numpy.ndarray) – matrix

• rtol (float) – Defaults to 1e-05.

• atol (float) – Defaults to 1e-08.
5.7.4 Linear Programming

generate_LP

disropt.utils.LP_utils.generate_LP(n_var, n_constr, radius, direction='min', constr_form='ineq')

Generate a feasible and not unbounded Linear Program and return problem data (cost, constraints, solution)

TODO add reference

Parameters

- n_var (int) – number of optimization variables
- n_constr (int) – number of constraints
- radius (float) – size of feasible set (for constr_form = ‘ineq’), size of dual feasible set (for constr_form = ‘eq’)
- direction (str, optional) – optimization direction - either ‘max’ (for maximization) or ‘min’ (for minimization). Defaults to ‘min’
- constr_form (str, optional) – form of constraints - either ‘ineq’ (for inequality: \(Ax \leq b\)) or ‘eq’ (for standard form: \(Ax = b, x \geq 0\)). Defaults to ‘ineq’

Returns

- c (numpy.ndarray): cost vector
- A (numpy.ndarray): constraint matrix
- b (numpy.ndarray): constraint vector (right-hand side)
- solution (numpy.ndarray): optimal solution of problem

Return type tuple
6.1 Optimization Problems

The `Problem` class allows one to define optimization problems of various types. Consider the following problem:

\[
\begin{align*}
\text{minimize} & \quad \|A^T x - b\| \\
\text{subject to} & \quad x \geq 0
\end{align*}
\]

with \( x \in \mathbb{R}^4 \). We can define it as:

```python
import numpy as np
from disropt.problems import Problem
from disropt.functions import Variable, Norm

x = Variable(4)
A = np.random.randn(n, n)
b = np.random.randn(n, 1)
obj = Norm(A @ x - b)
constr = x >= 0
pb = Problem(objective_function = obj, constraints = constr)
```

If the problem is convex, it can be solved as:

```python
solution = pb.solve()
```

Generic (convex) nonlinear problems of the form

\[
\begin{align*}
\text{minimize} & \quad f(x) \\
\text{subject to} & \quad g(x) \leq 0 \\
& \quad h(x) = 0
\end{align*}
\]

are solved through the `cvxpy` solver (when possible), or with the `cvxopt` solver, while more structured problems (LPs and QPs) can be solved through other solvers (osqp and glpk). The integration with other solvers will be provided in future releases. LPs and QPs can be directly defined through specific classes (`LinearProblem` and `QuadraticProblem`). However, the `Problem` class is capable to recognize LPs and QPs, which are automatically converted into the appropriate format.
6.1.1 Projection onto the constraints set

Projecting a point onto the constraints set of a problem is often required in distributed optimization algorithms. The method `project_on_constraint_set` is available to do this:

```python
projected_point = pb.project_on_constraint_set(pt)
```

6.2 Implementing custom functions

Custom functions can be easily implemented and integrated with already defined functions. They can be built on the `AbstractFunction` class, by overloading the `eval` method. Subgradients, jacobians and hessians are usually automatically computed through `autograd`. If they cannot be computed, then the `_alternative_jacobian` and `_alternative_hessian` method must be implemented too.

WARNING: jacobian of custom functions should be implemented by using the numerator layout convention.
ACKNOWLEDGEMENTS

This result is part of a project that has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No 638992 - OPT4SMART).

http://www.opt4smart.eu


INDEX

A

A_init (disropt.algorithms.constraintexchange.DistributedSimplex attribute), 95
Abs (class in disropt.functions.abs), 103
AbstractConstraint (class in disropt.constraints.constraints), 110
AbstractFunction (class in disropt.functions.abstract_function), 100
AbstractSet (class in disropt.constraints.projection_sets), 112
add_constraint () (disropt.problems.Problem method), 116
ADMM (class in disropt.algorithms.admm), 86
affine (disropt.functions.abstract_function.AbstractFunction attribute), 100
AffineForm (class in disropt.functions.affine_form), 101
Agent (class in disropt.agents), 71
agent (disropt.algorithms.Algorithm attribute), 99
agent (disropt.algorithms.consensus.AsynchronousConsensus attribute), 77
agent (disropt.algorithms.consensus.Consensus attribue), 75
agent (disropt.algorithms.consensus.PushSumConsensus attribute), 79
agent (disropt.algorithms.gradient_tracking.GradientTracking attribute), 83
agent (disropt.algorithms.setmembership.AsynchronousSetMembership attribute), 92
agent (disropt.algorithms.setmembership.SetMembership attribute), 91
Algorithm (class in disropt.algorithms), 99
angle (disropt.constraints.projection_sets.CircularSector attribute), 115
ASYMM (class in disropt.algorithms.asymm), 90
AsynchronousConsensus (class in disropt.algorithms.consensus), 76
AsynchronousLogicAnd (class in disropt.algorithms.misc), 98
AsynchronousSetMembership (class in disropt.algorithms.setmembership), 92

B

B (disropt.algorithms.constraintexchange.DistributedSimplex attribute), 93
b_init (disropt.algorithms.constraintexchange.DistributedSimplex attribute), 94
B_neigh (disropt.algorithms.constraintexchange.DistributedSimplex attribute), 95
batch_hessian () (disropt.functions.stochastic_function.StochasticFunction method), 108
batch_jacobian () (disropt.functions.stochastic_function.StochasticFunction method), 108
batch_subgradient () (disropt.functions.stochastic_function.StochasticFunction method), 108
binomial_random_graph () (in module disropt.utils.graph_constructor), 122
binomial_random_graph_sequence () (in module disropt.utils.graph_constructor), 122
BlockConsensus (class in disropt.algorithms.consensus), 78
blocksubgradient () (disropt.functions.submodular_func.SubmodularFn method), 109
BlockSubgradientMethod (class in disropt.algorithms.subgradient), 82
Box (class in disropt.constraints.projection_sets), 112

C

c_init (disropt.algorithms.constraintexchange.DistributedSimplex attribute), 95
center (disropt.constraints.projection_sets.Circle attribute), 114
change_flag () (disropt.algorithms.misc.LogicAnd method), 97
check_index_consistency () (disropt.algorithms.constraintexchange.DistributedSimplex method), 95
check_index_consistency () (dis-
eval() (disropt.functions.quadratic_form.QuadraticForm method), 102
eval() (disropt.functions.square.Square method), 106
eval() (disropt.functions.squared_norm.SquaredNorm method), 104
eval() (disropt.functions.stochastic_function.StochasticFunction method), 107
eval() (disropt.functions.submodular_func.SubmodularFn method), 110
eval() (disropt.functions.variable.Variable method), 101
Exp (class in disropt.functions.exp), 104
ExtendedConstraint (class in disropt.constraints.extended_constraint), 111
ExtendedFunction (class in disropt.functions.extended_function), 108

F

class fn (disropt.constraints.constraints.Constraint attribute), 110
force_computation_time (disropt.algorithms.consensus.AsynchronousConsensus attribute), 78
force_matrix_update() (disropt.algorithms.misc.LogicAnd method), 97
force_sleep (disropt.algorithms.consensus.AsynchronousConsensus attribute), 77
force_unreliable_links (disropt.algorithms.consensus.AsynchronousConsensus attribute), 78
function() (disropt.constraints.constraints.Constraint property), 110

G

class generate_LP () (in module disropt.utils.LP_utils), 124
class get_basis() (disropt.algorithms.constraintexchange.ConstraintsConsensus method), 94
class get_basis() (disropt.algorithms.constraintexchange.DistributedSimplex method), 95
class get_local_info() (disropt.utils.graph_constructor.MPIgraph method), 121
class get_parameters() (disropt.constraints.constraints.Constraint method), 110
class get_parameters() (disropt.functions.abstract_function.AbstractFunction method), 101
class get_parameters() (disropt.functions.affine_form.AffineForm method), 102
class get_result() (disropt.algorithms.admm.ADMM method), 103
class get_result() (disropt.functions.quadratic_form.QuadraticForm method), 103
get_result() (disropt.algorithms.Algorithm method), 100
get_result() (disropt.algorithms.asymm.ASYMM method), 90
get_result() (disropt.algorithms.consensus.AsynchronousConsensus method), 78
get_result() (disropt.algorithms.consensus.BlockConsensus method), 79
get_result() (disropt.algorithms.consensus.Consensus method), 76
get_result() (disropt.algorithms.consensus.PushSumConsensus method), 80
get_result() (disropt.algorithms.constraintexchange.ConstraintsConsensus method), 94
get_result() (disropt.algorithms.constraintexchange.DistributedSimplex method), 95
get_result() (disropt.algorithms.constraintexchange.DualDistributedSimplex method), 97
get_result() (disropt.algorithms.dual_decomp.DualDecomposition method), 85
get_result() (disropt.algorithms.dual_subgradient.DualSubgradientMethod method), 87
get_result() (disropt.algorithms.gradient_tracking.DirectedGradientTracking method), 84
get_result() (disropt.algorithms.gradient_tracking.GradientTracking method), 84
get_result() (disropt.algorithms.misc.MaxConsensus method), 99
get_result() (disropt.algorithms.primal_decomp.PrimalDecomposition method), 88
GradientTracking (class in disropt.algorithms.gradient_tracking), 83
greedy_polyhedron() (disropt.functions.submodular_func.SubmodularFn method), 109

H

class h_angle (disropt.constraints.projection_sets.CircularSector attribute), 115
hessian() (disropt.functions.abstract_function.AbstractFunction attribute), 101

I

class id (disropt.agents.Agent attribute), 71
in_neighbors (disropt.agents.Agent attribute), 71
in_weights (disropt.agents.Agent attribute), 72
initialize() (disropt.algorithms.constraintexchange.DistributedSimplex method), 96
initialize_algorithm() (disropt.algorithms.admm.ADMM method), 86
input_shape (disropt.constraints.constraints.Constraint attribute), 110
input_shape (disropt.constraints.projection_sets.Box attribute), 112
input_shape (disropt.constraints.projection_sets.Circle attribute), 114
input_shape (disropt.constraints.projection_sets.CircularSector attribute), 115
input_shape (disropt.constraints.projection_sets.Strip attribute), 113
input_shape (disropt.functions.abstract_function.AbstractFunction attribute), 100
input_shape (disropt.functions.submodular_fn SubmodularFn attribute), 109
input_shape (disropt.problems.Problem attribute), 116
intersection() (disropt.constraints.projection_sets.Box method), 112
intersection() (disropt.constraints.projection_sets.Circle method), 114
intersection() (disropt.constraints.projection_sets.CircularSector method), 115
intersection() (disropt.constraints.projection_sets.Strip method), 113
is_inequality() (disropt.constraints.constraints.Constraint property), 111
is_infinity() (disropt.functions.abstract_function.AbstractFunction property), 101
is_pos_def () (in module disropt.utils.utilities), 123
is_quadratic() (disropt.constraints.constraints.Constraint property), 111
is_quadratic () (disropt.functions.abstract_function.AbstractFunction property), 101
is_semi_pos_def () (in module disropt.utils.utilities), 123
iterate_run() (disropt.algorithms.admm.ADMM method), 86
iterate_run() (disropt.algorithms.consensus.AsynchronousConsensus method), 78
iterate_run() (disropt.algorithms.consensus.BlockConsensus method), 79
iterate_run() (disropt.algorithms.consensus.Consensus method), 76
iterate_run() (disropt.algorithms.consensus.PushSumConsensus method), 80
iterate_run() (disropt.algorithms.constraintexchange.ConstraintsConsensus method), 94
iterate_run() (disropt.algorithms.gradient_tracking.DirectedGradientTracking method), 85
iterate_run() (disropt.algorithms.gradient_tracking.GradientTracking method), 84
iterate_run() (disropt.algorithms.misc.AsynchronousLogicAnd method), 98
iterate_run() (disropt.algorithms.misc.LogicAnd method), 97
iterate_run() (disropt.algorithms.misc.MaxConsensus method), 99
iterate_run() (disropt.algorithms.primal_decomp.PrimalDecomposition method), 88
iterate_run() (disropt.algorithms.primal_decomp_milp.PrimalDecompositionMILP method), 89
LinearProblem (class in disropt.problems), 117
link_failure_probability (disropt.algorithms.consensus.AsynchronousConsensus attribute), 78
Log (class in disropt.functions.log), 104
LogicAnd (class in disropt.algorithms.misc), 97
Logistic (class in disropt.functions.logistic), 105
lower (disropt.constraints.projection_sets.Strip attribute), 113
lower_bound (disropt.constraints.projection_sets.Box attribute), 112
M
matrix_reset() (disropt.algorithms.misc.LogicAnd method), 97
matrix_update() (disropt.algorithms.misc.LogicAnd method), 97
Max (class in disropt.functions.max), 106
MaxConsensus (class in disropt.algorithms.misc), 99
maximum_computation_time (disropt.algorithms.consensus.AsynchronousConsensus attribute), 78
maximum_sleep (disropt.algorithms.consensus.AsynchronousConsensus attribute), 77
measure() (disropt.algorithms.setmembership.AsynchronousSetMembership method), 92
measure() (disropt.algorithms.setmembership.SetMembership method), 92
metropolis_hastings() (in module disropt.utils.graph_constructor), 122
Min (class in disropt.functions.min), 105
mixed_integer_solution() (disropt.algorithms.primal_decomp_milp.PrimalDecompositionMILP method), 89
MixedIntegerLinearProblem (class in disropt.problems), 117
MPICommunicator (class in disropt.communicators), 74
MPIgraph (class in disropt.utils.graph_constructor), 121
N
n_constr (disropt.algorithms.constraintexchange.DistributedSimplex attribute), 95
neighbors_exchange() (disropt.agents.Agent method), 72
neighbors_exchange() (disropt.communicators.Communicator method), 73
neighbors_exchange() (disropt.communicators.MPICommunicator method), 74
neighbors_receive() (disropt.communicators.Communicator method), 73
neighbors_receive() (disropt.communicators.MPICommunicator method), 74
neighbors_receive() (disropt.agents.Agent method), 72
neighbors_receive() (disropt.communicators.Communicator method), 73
neighbors_receive() (disropt.communicators.MPICommunicator method), 74
neighbors_send() (disropt.agents.Agent method), 72
neighbors_send() (disropt.communicators.Communicator method), 73
neighbors_send() (disropt.communicators.MPICommunicator method), 74
neighbors_receive() (disropt.communicators.MPICommunicator method), 74
neighbors_receive() (disropt.agents.Agent method), 72
neighbors_send() (disropt.communicators.Communicator method), 73
neighbors_send() (disropt.communicators.MPICommunicator method), 74
neighbors_receive() (disropt.communicators.MPICommunicator method), 74
objective_function (disropt.problems.ConstraintCoupledProblem attribute), 120
out_neighbors (disropt.agents.Agent attribute), 72
out_weights (disropt.agents.Agent attribute), 72
output_shape (disropt.constraints.constraints.Constraint attribute), 110
output_shape (disropt.functions.abstract_function.AbstractFunction attribute), 100
output_shape (disropt.problems.Problem attribute), 116
P
Power (class in disropt.functions.power), 106
primal_update_step() (disropt.algorithms.asymm.ASYM method), 90
PrimalDecomposition (class in disropt.algorithms.primal_decomp), 88
PrimalDecompositionMILP (class in disropt.algorithms.primal_decomp_milp), 89
Problem (class in disropt.problems), 116
problem (disropt.agents.Agent attribute), 72
project_on_constraint_set() (disropt.problems.Problem method), 116
projection() (disropt.constraints.constraints.Constraint method), 111
Index 135
set_problem() (disropt.agents.Agent method), 72
set_weights() (disropt.agents.Agent method), 73
Submembership (class in disropt.algorithms.setmembership), 91
SubmodularFn (class in disropt.functions.submodular_func), 109
timestamp_sequence (disropt.algorithms.setmembership.AsynchronousSetMembership attribute), 92
timestamp_sequence_awake (disropt.algorithms.consensus.AsynchronousConsensus attribute), 77
timestamp_sequence_sleep (disropt.algorithms.consensus.AsynchronousConsensus attribute), 77
to_constraints() (disropt.constraints.projection_sets.AbstractSet method), 112
to_constraints() (disropt.constraints.projection_sets.Box method), 113
to_constraints() (disropt.constraints.projection_sets.Circle method), 114
to_constraints() (disropt.constraints.projection_sets.CircularSector method), 116
to_constraints() (disropt.constraints.projection_sets.Strip method), 114
unique_constr() (disropt.algorithms.constraintexchange.ConstraintsConsensus method), 94
update_column() (disropt.algorithms.constraintexchange.ConstraintsConsensus method), 94
upper (disropt.constraints.projection_sets.Strip attribute), 113
upper_bound (disropt.constraints.projection_sets.Box attribute), 112
V
V (disropt.functions.submodular_func.SubmodularFn attribute), 109
variable (class in disropt.functions.variable), 101
vertex (disropt.constraints.projection_sets.CircularSector attribute), 115
X
x (disropt.algorithms.consensus.AsynchronousConsensus attribute), 77
x (disropt.algorithms.consensus.Consensus attribute), 75
x (disropt.algorithms.constraintexchange.ConstraintsConsensus attribute), 93

Index 137
\textcolor{red}{\times (disropt.algorithms.constraintexchange.DistributedSimplex} \\
\textcolor{red}{\text{\hspace{1cm} attribute), 94}} \\
\textcolor{red}{\times (disropt.algorithms.gradient\_tracking.GradientTracking} \\
\textcolor{red}{\text{\hspace{1cm} attribute), 83}} \\
\textcolor{red}{\times (disropt.algorithms.setmembership.AsynchronousSetMembership} \\
\textcolor{red}{\text{\hspace{1cm} attribute), 92}} \\
\textcolor{red}{\times (disropt.algorithms.setmembership.SetMembership \text{\hspace{1cm} attribute), 91}} \\
\textcolor{red}{\times_0 (disropt.algorithms.consensus.AsynchronousConsensus} \\
\textcolor{red}{\text{\hspace{1cm} attribute), 77}} \\
\textcolor{red}{\times_0 (disropt.algorithms.consensus.Consensus \text{\hspace{1cm} attribute), 75}} \\
\textcolor{red}{\times_0 (disropt.algorithms.gradient\_tracking.GradientTracking} \\
\textcolor{red}{\text{\hspace{1cm} attribute), 83}} \\
\textcolor{red}{\times_0 (disropt.algorithms.setmembership.AsynchronousSetMembership} \\
\textcolor{red}{\text{\hspace{1cm} attribute), 92}} \\
\textcolor{red}{\times_0 (disropt.algorithms.setmembership.SetMembership \text{\hspace{1cm} attribute), 91}} \\
\textcolor{red}{\times_{\text{\_basic}} (disropt.algorithms.constraintexchange.DistributedSimplex} \\
\textcolor{red}{\text{\hspace{1cm} attribute), 94}} \\
\textcolor{red}{\times_{\text{\_neigh}} (disropt.algorithms.consensus.AsynchronousConsensus} \\
\textcolor{red}{\text{\hspace{1cm} attribute), 77}} \\
\textcolor{red}{\times_{\text{\_neigh}} (disropt.algorithms.consensus.Consensus \text{\hspace{1cm} attribute), 75}} \\
\textcolor{red}{\times_{\text{\_neigh}} (disropt.algorithms.consensus.PushSumConsensus} \\
\textcolor{red}{\text{\hspace{1cm} attribute), 80}} \\
\textcolor{red}{\times_{\text{\_neigh}} (disropt.algorithms.constraintexchange.ConstraintsConsensus} \\
\textcolor{red}{\text{\hspace{1cm} attribute), 93}} \\
\textcolor{red}{\times_{\text{\_neigh}} (disropt.algorithms.gradient\_tracking.GradientTracking} \\
\textcolor{red}{\text{\hspace{1cm} attribute), 83}} \\
\textcolor{red}{\times_{\text{\_neigh}} (disropt.algorithms.setmembership.AsynchronousSetMembership} \\
\textcolor{red}{\text{\hspace{1cm} attribute), 92}} \\
\textcolor{red}{\times_{\text{\_neigh}} (disropt.algorithms.setmembership.SetMembership} \\
\textcolor{red}{\text{\hspace{1cm} attribute), 91}} \\
\textcolor{red}{Y} \\
\textcolor{red}{\text{\_neigh}} (disropt.algorithms.consensus.PushSumConsensus} \\
\textcolor{red}{\text{\hspace{1cm} attribute), 80}} \\
\textcolor{red}{Z} \\
\textcolor{red}{z (disropt.algorithms.consensus.PushSumConsensus \text{\hspace{1cm} attribute), 80}} \\
\textcolor{red}{z_0 (disropt.algorithms.consensus.PushSumConsensus \text{\hspace{1cm} attribute), 80}}