Available online at www.sciencedirect.com
ScienceDirect
IFAC PapersOnLine 53-2 (2020) 2666–2671

DISROPT: a Python Framework for Distributed Optimization

Francesco Farina, Andrea Camisa, Andrea Testa, Ivano Notarnicola, Giuseppe Notarstefano

Department of Electrical, Electronic and Information Engineering, Alma Mater Studiorum Università di Bologna, Bologna, Italy
{franc.farina, a.camisa, a.testa, ivano.notarnicola, giuseppe.notarstefano}@unibo.it.

Abstract: In this paper we introduce DISROPT, a Python package for distributed optimization over networks. We focus on cooperative set-ups in which an optimization problem must be solved by peer-to-peer processors (without central coordinators) that have access only to partial knowledge of the entire problem. To reflect this, agents in DISROPT are modeled as entities that are initialized with their local knowledge of the problem. Agents then run local routines and communicate with each other to solve the global optimization problem. A simple syntax has been designed to allow for an easy modeling of the problems. The package comes with many distributed optimization algorithms that are already embedded. Moreover, the package provides full-fledged functionalities for communication and local computation, which can be used to design and implement new algorithms. DISROPT is available at `github.com/disropt/disropt` under the GPL license, with a complete documentation and many examples.

1. INTRODUCTION

In recent years, distributed learning and control over networks have gained a growing attention. Most problems arising in these contexts can be formulated as distributed optimization problems, whose solution calls for the design of tailored strategies. The main idea of distributed optimization is to solve an optimization problem over a network of computing units, also called agents. Each agent can perform local computation and can exchange information only with its neighbors in the network. Typically, the problem to be solved is assumed to have a given structure, while the communication network can be unstructured. Each agent knows only a portion of the global optimization problem, so that finding a solution to the network-wide problem requires cooperation with the other agents. A distributed algorithm consists of an iterative procedure in which each agent alternates communication and computation phases with the aim of eventually finding a solution to the problem. The recent monograph Notarstefano et al. (2019) provides a comprehensive overview of the most common approaches for distributed optimization, together with the theoretical analysis of the main schemes in their basic version.

In this paper we present DISROPT, a Python package designed to run distributed optimization algorithms over peer-to-peer networks of processors. In the last years, several toolboxes have been developed in order to solve optimization problems using *centralized* algorithms. Examples of toolboxes written in C are OSQP Stellato et al. (2018), and GLPK Makhorin (2008). As for packages developed in C++, nonlinear optimization problems can be solved by using OPT++ Meza (1994). In the context of nonlinear optimization, we mention ACADO Houska et al. (2011), which deals with optimal control, and IPOPT Biegler and Zavala (2009), which solves large-scale problems. More recent interpreted languages, such as Matlab and Python, do not have the performance of low-level compiled languages such as C and C++, however they are more expressive and often easier to use. Well-known Matlab packages for optimization are YALMIP Löfberg (2004) and CVX Grant and Boyd (2014). Notable Python packages for convex optimization are CVXPY Diamond and Boyd (2016) and CVXOPT Andersen et al. (2015). Based on CVXPY, the toolbox SNAPVX Hallac et al. (2017) allows for the solution of large-scale convex problems defined over graphs by exploiting their structure. An extension of Diamond and Boyd (2016) to optimize convex objectives over nonconvex domains using heuristics is NCVX Diamond et al. (2018). Other well-known packages based on the recent programming language Julia are OPTIM, CONVEX.JL and JUMP Mogensen and Riseth (2018), Udell et al. (2014), Dunning et al. (2017). A Julia package for stochastic optimization is Huchette et al. (2014), which implements a parallel solver.

None of the above references provides direct capabilities to solve optimization problems over networks using distributed algorithms. The aim of DISROPT is to bridge this gap. The package is designed as follows. When a distributed algorithm is executed, many (possibly spatially distributed) processes are created. Each process corresponds to an agent in the network, has its own memory space with its private data, runs its own set of instructions and cooperates with the other agents through a message-passing paradigm. Consistently with the distributed model, there is no central coordinator. The package provides a comprehensive framework for all the typical tasks that must be performed by distributed optimization algorithms.

⋆ 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).

10.1016/j.ifacol.2020.12.382

In particular, the package allows for both synchronous and asynchronous communication over custom networks of peer-to-peer agents. Moreover, it provides an easy-to-use interface to represent, for each agent, the local knowledge of the global optimization problem to be solved and to run distributed optimization algorithms via a streamlined interface. Local optimization problems can be also solved (if needed). The tools provided by DISROPT let the user to easily design new distributed optimization algorithms by using an intuitive syntax and to solve several classes of problems arising both in distributed control and machine learning frameworks.

The paper is organized as follows. In Section 2, the distributed optimization framework is introduced. The architecture of DISROPT is presented in Section 3 and a numerical computation on three example scenarios is provided in Section 4.

2. DISTRIBUTED OPTIMIZATION SET-UPS

In this section, we introduce the optimization set-ups considered in DISROPT. 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,

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & x \in X, \end{aligned}$$

where $x \in \mathbb{R}^d$ is called optimization variable, $f : \mathbb{R}^d \to \mathbb{R}$ is called cost function and $X \subseteq \mathbb{R}^d$ describes the problem constraints. The optimization problem is assumed to be feasible, to have finite optimal cost and to admit at least an optimal solution, which is usually denoted as $x^\star$. The optimal solution is a vector satisfying all the constraints and attaining the optimal cost. If the problem is nonconvex, $x^\star$ can be any (feasible) stationary point.

Distributed optimization problems arising in applications usually enjoy a proper structure in their mathematical formulation. In DISROPT, three different optimization set-ups are considered and are detailed next.

2.1 Cost-coupled Set-up

In the *cost-coupled optimization set-up*, the cost function is expressed as the sum of local contributions f_i and all of them depend on a common optimization variable x. Formally, the set-up is

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \sum_{i=1}^{N} f_i(x) \\ \text{subject to} \quad & x \in X, \end{aligned} \tag{1}$$

where $x \in \mathbb{R}^d$ and $X \subseteq \mathbb{R}^d$. The global constraint set X is known to all the agents, while $f_i : \mathbb{R}^d \to \mathbb{R}$ is assumed to be known by agent i only, for all $i \in \{1, \dots, N\}$.

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 assumed to be known by agent i only, for all $i \in \{1, \dots, N\}$.

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

2.2 Common-cost Set-up

In the *common-cost 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

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & x \in \bigcap_{i=1}^{N} X_i \end{aligned} \tag{2}$$

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 known by agent i only, for all $i \in \{1, \dots, N\}$.

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

2.3 Constraint-coupled Set-up

In the *constraint-coupled optimization set-up*, the cost function is expressed as the sum of local contributions f_i that depend on a local optimization variable x_i. The variables must satisfy local constraints and global coupling constraints among all of them. Formally, the set-up is

$$\begin{aligned} \underset{x_1,\dots,x_N}{\text{minimize}} \quad & \sum_{i=1}^{N} f_i(x_i) \\ \text{subject to} \quad & x_i \in X_i, \quad i \in \{1, \dots, N\} \\ & \sum_{i=1}^{N} g_i(x_i) \leq 0 \end{aligned} \tag{3}$$

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 $i \in \{1, \dots, N\}$. Here the symbol $\leq$ is also used to denote component-wise inequality for vectors. Therefore, the optimization variable consists of the stack of all the variables x_i, namely the vector $(x_1, \dots, x_N)$. The quantities with the subscript i are assumed to be known by agent i only, for all $i \in \{1, \dots, N\}$. 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 asymptotically computes its portion $x_i^\star \in X_i$ of an optimal solution $(x_1^\star, \dots, x_N^\star)$ of the optimization problem, thereby satisfying also the coupling constraints $\sum_{i=1}^{N} g_i(x_i^\star) \leq 0$.

3. SOFTWARE ARCHITECTURE AND BASIC SYNTAX

In this section, we present the architecture of the package, which replicates the typical structure of a distributed algorithm, see Figure 1. The main entities of a distributed scenario are the agents (with their local information), the communication network, and the local routines of the distributed algorithm.

In DISROPT, these components are represented with an object-oriented framework. In the remainder of this section, we provide a brief description of the main classes and of the semantics of DISROPT.

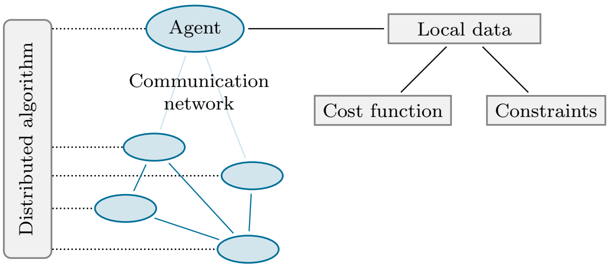

Fig. 1. Distributed scenario architecture. Agents are equipped with their local information and interact with each other through the communication network to run the distributed algorithm.

3.1 Agent

An instance of the *Agent* class represents a single computing unit in the network. This class is equipped with the list of the neighboring agents. It is also embedded with an instance of the class *Problem* (detailed next), which describes the local knowledge of the global optimization problem.

Suppose that we want to instantiate the agent with index 1 of the network in Figure 2. The in-neighbors of agent 1

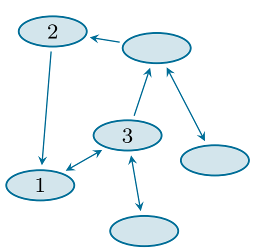

Fig. 2. Example of a network of 6 agents.

are agents 2 and 3 while the unique out-neighbor is agent 3. The following Python code can be used to accomplish the task:

```
from disropt.agents import Agent

in_nbrs = [2, 3]
out_nbrs = [3]
agent = Agent(in_nbrs, out_nbrs)
```

3.2 Communication Network

The *Communicator* class handles communication among the network agents. It allows agents to send and receive data from neighbors in a synchronous/asynchronous, time-invariant/time-varying fashion. In the current release, communication over the Message Passing Interface (MPI) is supported, however custom communication protocols can be implemented as well.

As an example, to perform synchronous communication to exchange a two-dimensional vector with neighbors, the syntax is

```
from disropt.communicators import MPICommunicator

vect = numpy.random.rand(2, 1)
comm = MPICommunicator()
exch_data = comm.neighbors_exchange(vect, in_nbrs,
     out_nbrs, dict_neigh=False)
```

The flag `dict_neigh` set to `False` means that the same object is sent to all the neighbors, while `exch_data` is a dictionary containing all the vectors received from the neighbors with their corresponding indices.

An instance of this class is embedded in the Agent class which also manages the list of in- and out-neighbors. Therefore, the previous code can also be restated as follows:

```
vect = numpy.random.rand(2, 1)
exch_data = agent.neighbors_exchange(vect)
```

Note that the provided communication features are typically required only during the algorithm implementation.

3.3 Distributed Algorithm

The class *Algorithm* aims to characterize the behavior and the local steps of distributed algorithms. This class reads the local problem data, handles the data received by neighbors and updates the local solution estimate. Specializations of this class correspond to different distributed algorithms. We implemented several algorithms corresponding to the different distributed optimization set-ups of Section 2. An exhaustive list of the currently implemented algorithms is provided in Table 1. References to the implemented algorithms can be found in Notarstefano et al. (2019).

For example, in order to run the distributed subgradient method for 100 iterations, the Python code is:

```
from disropt.algorithms import SubgradientMethod

x0 = numpy.random.randn(2, 1)
algorithm = SubgradientMethod(agent=agent,
     initial_condition=x0)
algorithm.run(iterations=100)
```

Notice that we are assuming that the `agent` is already equipped with the local problem information, which can be done as described in the next subsection.

Then, it is possible to run the distributed algorithm over a network of N agents by executing:

```
mpirun -np N python <source_file.py>
```

3.4 Local Agent Data

The class *Problem* is used to model the locally available data of the global optimization problem. It is embedded with an objective function (an instance of the class *Function*) and a list of constraints (a list of objects of the class *Constraint*). The class should be initialized according to the specific distributed optimization set-up and must be provided to the class *Agent*.

For instance, suppose that, in a cost-coupled set-up, the agent knows the following function and constraint,

$$f_i(x) = \|x\|^2, \qquad X_i = \{x \in \mathbb{R}^2 \mid -1 \leq x \leq 1\}.$$

Table 1. Overview of implemented algorithms. *=next release.

Algorithm	Communication	Time-varying	Block-wise	Constraints	Non-smooth	Asynchronous
Cost-coupled						
Distributed subgradient	Directed	✓	✓	Global	✓	✓
Gradient tracking	Directed	✓	✓*	No		
Distributed dual decomposition	Undirected			Local	✓	✓*
Distributed ADMM	Undirected			Local	✓	
ASYMM	Undirected		✓	Local		✓
Common cost						
Constraints consensus	Directed	✓		Local		✓*
Distributed set membership	Directed	✓		Local		✓
Constraint-coupled						
Distributed Dual Subgradient	Directed			Local	✓	
Distributed Primal Decomposition	Undirected			Local	✓	

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)
agent.set_problem(problem)
```

Since many distributed algorithms require the solution of small optimization problems, this class is also able to solve the problem described by the cost function and the constraints through the method `solve()`, which currently relies upon CVXOPT, GLPK, OSPQ and CVXPY.

4. EXAMPLE USAGE

In this section we present three illustrative examples on how DISROPT can be used to solve a cost-coupled (cf. Sec. 2.1), a common-cost (cf. Sec. 2.2) and a constraint-coupled optimization problem (see Sec. 2.3), respectively. In particular, we consider a distributed classification problem where the training points are spread among the agents and a distributed microgrid control problem.

4.1 Distributed Classification via Logistic Loss

The classification problem consists in dividing a set of points, representing data in a feature space, into two clusters, by means of a separating hyperplane. The purpose of the agents is to cooperatively estimate the parameters of the hyperplane. In Figure 3, a bidimensional example is reported. The classification task corresponds to computing the parameters of the line (in red) separating triangles from circles.

Problem formulation Let us consider N agents contributing to the d-dimensional classification problem as follows. Each agent i is equipped with m_i points $p_{i,1}, \dots, p_{i,m_i} \in \mathbb{R}^d$. 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 $j \in \{1, \dots, m_i\}$ and $i \in \{1, \dots, N\}$. 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 following optimization problem

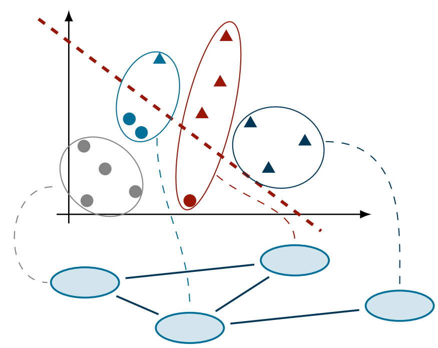

Fig. 3. Illustration of a distributed classification problem.

$$\underset{w,b}{\text{minimize}} \quad \sum_{i=1}^{N} \sum_{j=1}^{m_i} \log\left(1 + e^{-(w^\top p_{i,j} + b)\ell_{i,j}}\right) + \frac{C}{2}\|w\|^2,$$

where $C > 0$ is the regularization parameter. As we already mentioned, this is a cost-coupled problem of the type (1), where each local cost function f_i is appropriately defined [1]. The goal is to make agents agree on a common solution $(w^\star, b^\star)$, so that all of them can compute the separating hyperplane as $\{z \in \mathbb{R}^d \mid (w^\star)^\top z + b^\star = 0\}$.

Simulation results We consider a bidimensional sample space ($d = 2$). Each agent i generates a total of m_i points of both labels, with m_i a random number between 4 and 10. 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 $C = 10$.

We run a comparative study of the distributed subgradient algorithm and the gradient tracking algorithm, with $N = 20$ agents and 20 000 iterations.

As for the step size, we use the following rules: constant step-size $\alpha^t = 0.001$ for gradient tracking and diminishing step-size $\alpha^t = (1/t)^{0.6}$ or distributed subgradient.

The simulation results are reported in Figures 4 and 5. It can be seen that, for both the distributed algorithms, the solution and cost error go to zero (although with different rates).

[1] The regularization term can be appropriately split among the agents so that $f_i(x) = \sum_{j=1}^{m_i} \log\left(1 + e^{-(w^\top p_{i,j} + b)\ell_{i,j}}\right) + \frac{C}{2N}\|w\|^2$.

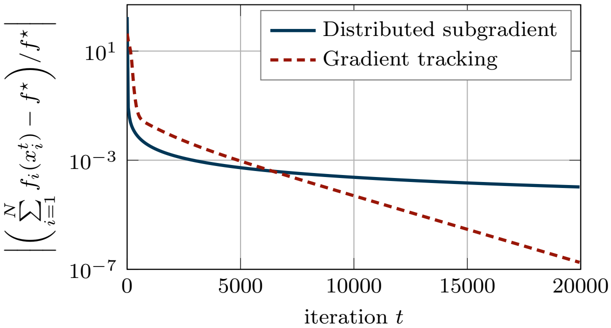

Fig. 4. Distributed classification: normalized cost error between the locally computed solution estimates and the optimal cost.

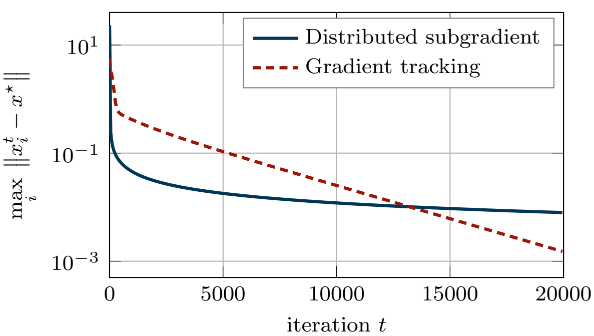

Fig. 5. Distributed classification: maximum error between the local solution estimates and the optimal solution.

4.2 Distributed Classification via Support Vector Machines

In this example, we consider again the distributed classification problem and we apply a different strategy to solve it.

Problem formulation If the data are known to be divided into two clusters that can be exactly separated by a hyperplane, the previous distributed classification problem can be recast as a *hard-margin* SVM problem, i.e.,

$$\begin{aligned} \underset{w,b}{\text{minimize}} \quad & \frac{1}{2}\|w\|^2 \\ \text{subject to} \quad & \ell_{i,j}(w^\top p_{i,j} + b) \geq 1, \quad \forall j,\ \forall i. \end{aligned} \tag{4}$$

This is a common-cost problem of the type (2), in which the constraint set of agent i is given by

$$X_i = \{(w,b) \mid \ell_{i,j}(w^\top p_{i,j} + b) \geq 1,\ j = 1,\ldots,m_i\}.$$

Simulation results The problem data are generated as described in the previous example (see Section 4.1). We apply the Constraints Consensus algorithm, with $N = 20$ agents for 10 iterations. Figure 6 depicts the evolution of the cost error computed at the local solution estimate of each agent. As expected from the theoretical analysis (see, e.g., (Notarstefano et al., 2019, Chapter 4)), all the agents converge to a cost-optimal solution in finite time. Moreover, let us define the maximum constraint violation of the solution computed by agent i at iteration t,

$$\phi_i^t = 1 - \max_{k,j} \ell_{k,j}\big((w_i^t)^\top p_{k,j} + b_i^t\big).$$

In Figure 7, it is shown that ϕ_i^t goes to 0 for all agents, meaning that the solution retrieved by the agents concurrently satisfies all the constraints of problem (4) in a finite numbe of iterations.

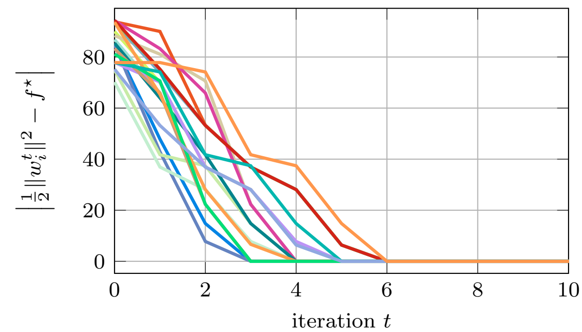

Fig. 6. Distributed classification via SVM: cost error between the locally computed solution estimates and the optimal cost.

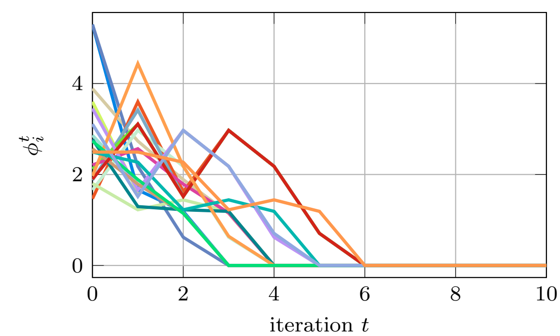

Fig. 7. Distributed classification via SVM: maximum constraints violation of the locally computed solution estimates.

4.3 Microgrid Control

In a microgrid control problem, a network of dynamical systems cooperate in order to compute an optimal "power profile" while satisfying both local constraints (e.g., local dynamics or bounds on the state variables) and global constraints (e.g., compliance with a common resource budget). We assume that agents are interested in optimizing their profiles over a given discrete-time horizon $\{0, 1, \ldots, S\}$, for some $S \in \mathbb{N}$.

Problem formulation Formally, we consider N dynamical units in which each agent i has state $x_i(k) \in \mathbb{R}$ and input $u_i(k) \in \mathbb{R}$ for all $k \in \{0, 1, \ldots, S\}$. For all $i \in \{1, \ldots, N\}$, the states and the inputs must satisfy a linear dynamics

$$x_i(k+1) = A_i x_i(k) + B_i u_i(k), \qquad k \in \{0, 1, \ldots, S\}$$

for given $x_i(0) \in \mathbb{R}$ and given matrices A_i and B_i of suitable dimensions. The constraint-coupled optimization problem can be cast as follows

$$\begin{aligned} \underset{x,u}{\text{minimize}} \quad & \sum_{k=1}^{S}\sum_{i=1}^{N} \ell_i(x_i(k), u_i(k)) \\ \text{subject to} \quad & x_i(k+1) = A_i x_i(k) + B_i u_i(k), \quad \forall k, \forall i \\ & \sum_{i=1}^{N} (C_i x_i(k) + D_i u_i(k)) \leq h_k, \quad \forall k. \end{aligned}$$

where h_k are entries of a given vector $h \in \mathbb{R}^S$, C_i and D_i are matrices of suitable dimensions, and x and u are the collections of all the states and inputs of the agents. The last line can interpreted as a constraint on the output map of the local systems.

More details can be found in Notarstefano et al. (2019).

Simulation results We run a comparative study of distributed dual subgradient and distributed primal decomposition with $N = 20$ agents, $S = 8$ coupling constraints and $20,000$ iterations. For both algorithms, we use a diminishing step-size rule $\alpha^t = (1/t)^{0.6}$.

The simulation results are reported in Figures 8 and 9. For both the distributed algorithms, the cost error goes to zero with sublinear rate. In this example, for the dual subgradient algorithm, the local solution estimates are within the coupling constraints after less than $5,000$ iterations, while for the distributed primal decomposition the solution estimates are always feasible.

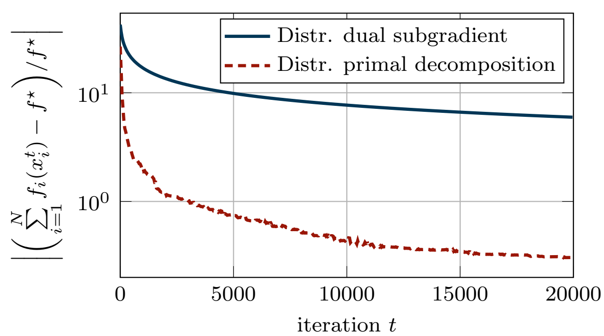

Fig. 8. Microgrid control: normalized cost error between the locally computed solution estimates and the optimal cost.

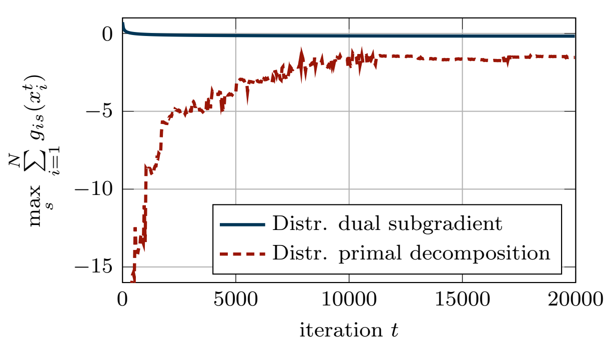

Fig. 9. Microgrid control: coupling constraint value for the computed solution estimates. Solutions are feasible if the line is below zero.

5. CONCLUSIONS

In this paper, we introduced DISROPT, a Python package for distributed optimization over peer-to-peer networks. The package allows users to define and solve optimization problems through distributed optimization algorithms. We presented the software architecture of the package together with simulation results over example application scenarios.

REFERENCES

Andersen, M., Dahl, J., and Vandenberghe, L. (2015). CVXOPT: Python software for convex optimization, version 1.2. URL `https://cvxopt.org`.

Biegler, L.T. and Zavala, V.M. (2009). Large-scale nonlinear programming using IPOPT: An integrating framework for enterprise-wide dynamic optimization. *Computers & Chemical Engineering*, 33(3), 575–582.

Diamond, S. and Boyd, S. (2016). CVXPY: A Python-embedded modeling language for convex optimization. *Journal of Machine Learning Research*, 17(83), 1–5.

Diamond, S., Takapoui, R., and Boyd, S. (2018). A general system for heuristic minimization of convex functions over non-convex sets. *Optimization Methods and Software*, 33(1), 165–193.

Dunning, I., Huchette, J., and Lubin, M. (2017). JuMP: A modeling language for mathematical optimization. *SIAM Review*, 59(2), 295–320.

Grant, M. and Boyd, S. (2014). CVX: Matlab software for disciplined convex programming, version 2.1. `http://cvxr.com/cvx`.

Hallac, D., Wong, C., Diamond, S., Sharang, A., Sosic, R., Boyd, S., and Leskovec, J. (2017). SnapVX: A network-based convex optimization solver. *The Journal of Machine Learning Research*, 18(1), 110–114.

Houska, B., Ferreau, H., and Diehl, M. (2011). ACADO Toolkit – An Open Source Framework for Automatic Control and Dynamic Optimization. *Optimal Control Applications and Methods*, 32(3), 298–312.

Huchette, J., Lubin, M., and Petra, C. (2014). Parallel algebraic modeling for stochastic optimization. In *Proceedings of the 1st First Workshop for High Performance Technical Computing in Dynamic Languages*, 29–35. IEEE Press.

Löfberg, J. (2004). YALMIP: A toolbox for modeling and optimization in MATLAB. In *Proceedings of the CACSD Conference*, volume 3. Taipei, Taiwan.

Makhorin, A. (2008). GLPK (GNU linear programming kit). URL `http://www.gnu.org/s/glpk/glpk.html`.

Meza, J.C. (1994). OPT++: An object-oriented class library for nonlinear optimization. Technical report, Sandia National Labs., Livermore, CA (United States).

Mogensen, P.K. and Riseth, A.N. (2018). Optim: A mathematical optimization package for Julia. *Journal of Open Source Software*, 3(24).

Notarstefano, G., Notarnicola, I., and Camisa, A. (2019). Distributed optimization for smart cyber-physical networks. *Foundations and Trends® in Systems and Control*, 7(3), 253–383.

Stellato, B., Banjac, G., Goulart, P., Bemporad, A., and Boyd, S. (2018). OSQP: An operator splitting solver for quadratic programs. In *UKACC International Conference on Control (CONTROL)*, 339–339. IEEE.

Udell, M., Mohan, K., Zeng, D., Hong, J., Diamond, S., and Boyd, S. (2014). Convex optimization in Julia. In *Proceedings of the 1st First Workshop for High Performance Technical Computing in Dynamic Languages*, 18–28. IEEE Press.