

# tvopt: A Python Framework for Time-Varying Optimization

Nicola Bastianello

**Abstract**—This paper introduces `tvopt`, a Python framework for prototyping and benchmarking time-varying (or online) optimization algorithms. The paper first describes the theoretical approach that informed the development of `tvopt`. Then it discusses the different components of the framework and their use for modeling and solving time-varying optimization problems. In particular, `tvopt` provides functionalities for defining both centralized and distributed online problems, and a collection of built-in algorithms to solve them, for example gradient-based methods, ADMM and other splitting methods. Moreover, the framework implements prediction strategies to improve the accuracy of the online solvers. The paper then proposes some numerical results on a benchmark problem and discusses their implementation using `tvopt`. The code for `tvopt` is available at [1].

## I. INTRODUCTION

In recent years, *time-varying (or online) optimization* has received increasing interest from the optimization, control, and learning communities [2]–[4]. Indeed, in many applications technological advances brought about a shift from traditional optimization problems to problems that have a dynamic nature, *e.g.* because they depend on streaming sources of data. Static optimization techniques then need to be revisited and adapted in order to provide reliable, on-the-fly algorithms for solving time-varying problems.

The goal of this paper is to propose `tvopt`, a framework written in Python for prototyping and benchmarking of online optimization algorithms, and to facilitate this shift from a static to a dynamic optimization context. The idea indeed is to provide all the necessary tools to model time-varying optimization problems, and to implement suitable solution algorithms and analyze their performance.

Formally, time-varying optimization problems can be modeled as follows:

$$\mathbf{x}^*(t) = \arg \min_{\mathbf{x} \in \mathbb{R}^n} F(\mathbf{x}; t) \quad (1)$$

where  $F : \mathbb{R}^n \times \mathbb{R}_+ \rightarrow \mathbb{R} \cup \{+\infty\}$  is a cost function that varies over time. For example, we may have  $F(\mathbf{x}; t) = \|\mathbf{A}\mathbf{x} - \mathbf{b}(t)\|^2/2$  to solve a regression task that employs time-varying observations  $\mathbf{b}(t) = \mathbf{A}\mathbf{y}(t) + \mathbf{e}(t)$  of a signal  $\mathbf{y}(t)$ , affected by additive noise  $\mathbf{e}(t)$ .

We are also interested in the *distributed* counterpart of (1), defined as

$$\begin{aligned} \mathbf{x}^*(t) = \arg \min_{\mathbf{x}_i \in \mathbb{R}^n} \sum_{i=1}^N F_i(\mathbf{x}_i; t) \\ \text{s.t. } \mathbf{x}_i = \mathbf{x}_j, \text{ if } i, j \text{ connected} \end{aligned} \quad (2)$$

where  $N$  is the number of agents cooperating towards the solution of (2), and each  $F_i$  is the private local cost of agent

$i = 1, \dots, N$ . For example, a multi-agent system of robots may encode a coordination task (*e.g.* moving in formation) as the distributed optimization problem, which is inherently time-varying due to the dynamic nature of the system.

Different online optimization algorithms have been proposed, both for centralized problems *e.g.* [5], [6], and in decentralized scenarios [7]–[9]. An interesting approach to developing online algorithms is that of *prediction-correction*, proposed in [6], [10] and extended in [11]. The main idea is to exploit past information on the problem to improve the solution accuracy of the algorithm.

Applications in which online algorithms are required range from signal and image processing, to control, and smart grids.

We refer to the surveys [3], [4] for an in-depth literature review and a discussion of the different applications.

**Contribution:** This paper describes the `tvopt` framework and the theoretical approach that informed its design. The paper describes the main features of `tvopt`, which can be summarized as follows:

- *problem modeling*: the framework provides an object-oriented approach to modeling and defining online optimization problems, both the costs and constraints;
- *decentralized problems*: moreover, `tvopt` offers tailored tools to model multi-agent networks and decentralized problems;
- *solvers*: `tvopt` implements widely used solution algorithms for different classes of unconstrained and constrained problems, both centralized and decentralized.

The paper also presents some results of numerical simulations performed on a benchmark problem, and shows and discusses their implementation using the tools of `tvopt`.

**Paper organization:** Section II discusses the theoretical approach that informs the design of `tvopt`. Section III describes the main components of the framework and their use for online optimization, while section IV presents additional tools for simulating distributed online problems. Section V concludes with a numerical example implemented using `tvopt`, and some simulations results.

## II. TIME-VARYING OPTIMIZATION

In this section we review the approach to time-varying optimization that informed the design of the `tvopt` framework. We refer the interested reader to the theoretical framework developed in [11] and the surveys [3], [4] for more details.

### A. Problem formulation

At the foundation of `tvopt` is a *discrete-time* approach to online optimization, see *e.g.* [3], which samples (1) in the following sequence of (static) problems:

$$\mathbf{x}^*(t_k) = \arg \min_{\mathbf{x} \in \mathbb{R}^n} F(\mathbf{x}; t_k) \quad (3)$$

where  $t_k, k \in \mathbb{N}$ , are the sampling instants and  $T_s = t_{k+1} - t_k$  is a chosen sampling time. This approach is opposed to a *continuous-time* one, see e.g. [12].

The goal then is to track the *optimal trajectory* given by the sequence  $\{\mathbf{x}^*(t_k)\}_{k \in \mathbb{N}}$  of minimizers of the sampled costs. However, the dynamic nature of the problem implies that the optima can be tracked only approximately, since a limited computational time (upper bounded by  $T_s$ ) is available to solve each problem in the sequence.

In order to illustrate this framework, consider the following examples.

*Example 1:* Let  $\mathbf{y}(t)$  be a signal to be reconstructed from the noisy observations  $\mathbf{b}(t_k) = \mathbf{A}\mathbf{y}(t_k) + \mathbf{e}(t_k)$ , with  $\mathbf{e}(t_k)$  denoting random noise. In this case we can define  $F(\mathbf{x}; t_k) = f(\mathbf{x}; t_k) + g(\mathbf{x})$  where  $f(\mathbf{x}; t_k) = (1/2) \|\mathbf{A}\mathbf{x} - \mathbf{b}(t_k)\|^2$  fits the observed data, and  $g(\mathbf{x}) = w \|\mathbf{x}\|_1$  promotes sparsity.

*Example 2:* In model predictive control (MPC), a control law is designed by solving a sequence of optimization problems which vary over time, since they depend on the states of a dynamical system. Thus MPC can be cast as a time-varying optimization problem and solved using tools developed in this framework. See [4] for an overview.

### B. Solvers

As remarked above, each problem in the sequence (3) needs to be (approximately) solved. We thus introduce the concept of *solver*, by which we mean any recursive algorithm that can be applied to the sampled problems.

Due to the limited computational time that is available between the observation of a problem and the next, in general we cannot solve exactly each problem in the sequence. Rather, we apply a finite number of steps of the solver to each sampled problem, denoted by  $N_C \in \mathbb{N}$ . This yields an approximate solution of the problems and thus leads to an approximate tracking of the optimal trajectory.

*Example 3:* Depending on the structure of the problem, a wide array of solvers can be used. For example, if  $F(\mathbf{x}; t_k)$  is smooth, then a gradient method is a suitable solver. Or, for a composite problem,  $F(\mathbf{x}; t_k) = f(\mathbf{x}; t_k) + g(\mathbf{x}; t_k)$ , we can apply a proximal gradient method.

### C. Prediction

As proposed in [6], [10] and further extended in [11], the knowledge of past sampled problems can be exploited to improve the tracking of the optimal trajectory. Indeed, the information collected up to time  $t_k$  can be used to shape a *prediction* of the (as yet unobserved) problem at time  $t_{k+1}$ . Then an approximate solution of the predicted problem can be used to *warm-start* the solver<sup>1</sup> when applied to the actual problem. As proved in [11], this approach allows to reduce the tracking error.

*Example 4:* A very simple prediction strategy is to choose  $\hat{F}(\mathbf{x}; t_{k+1}) = F(\mathbf{x}; t_k)$ , where  $\hat{F}$  denotes the prediction. Another, called *extrapolation*, chooses  $\hat{F}(\mathbf{x}; t_{k+1}) = 2F(\mathbf{x}; t_k) - F(\mathbf{x}; t_{k-1})$ .

<sup>1</sup>That is, the approximate prediction solution is used as initial condition for the solver applied to problem at time  $t_{k+1}$ .

To conclude this section, we detail the steps of a prediction-correction method for solving (3), see Figure 1, and refer to [11] for further details.

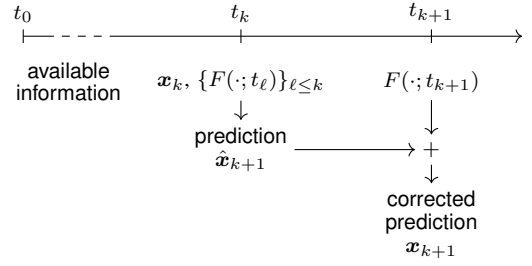


Fig. 1. The prediction-correction scheme.

- 1) *Initialization*: choose the sampling time  $T_s$ , a prediction strategy, and the solver and its parameters.

At each sampling time  $t_k, k \in \mathbb{N}$ :

- 2) *Prediction*: (i) predict the problem by computing  $\hat{F}(\cdot; t_k)$ , and (ii) solve it approximately with  $N_P \in \mathbb{N}$  steps of the solver, which yields the prediction  $\hat{\mathbf{x}}_{k+1}$ .
- 3) *Correction*: (i) sample the new problem at time  $t_{k+1}$ , and (ii) solve it approximately with  $N_C \in \mathbb{N}$  steps of the solver, using  $\hat{\mathbf{x}}_{k+1}$  as initial condition. The result is denoted by  $\mathbf{x}_{k+1}$ .

The framework delineated above can be particularized by omitting either the prediction or correction step. We remark that omitting the correction step yields the approach usually employed in *online learning* [2].

## III. TVOPT FRAMEWORK

In this section we describe the main components of `tvopt` and their application to online optimization as reviewed in the previous section; see [13] for the full documentation. The framework can be conceptually divided in the following:

- *Problem formulation*: the sub-modules `sets` and `costs`, which implement an object-oriented approach to defining time-varying problems. The sub-module `networks` can be used alongside the previous two to define distributed problems.
- *Prediction*: the sub-module `prediction` is provided for approximating future problems based on the problems observed in the past.
- *Solvers*: the sub-modules `solvers` and `distributed_solvers` implement a wide range of solvers that can be applied to (3).

In the following we describe in more details the sub-modules of `tvopt`, while section IV will discuss the specific tools implemented for online distributed optimization.

### A. Sets

This sub-module implements the `Set` objects which are used to define the domain of the cost objects. In particular, a `Set` is defined as a subset of  $\mathbb{R}^{n_1 \times n_2 \times \dots}$ , for some  $n_1, n_2, \dots \in \mathbb{N}$ . `Set` objects are then characterized by the dimensions  $n_1, n_2, \dots$  of the underlying space, which are stored in the attribute `shape`.

In this sub-module and in the following the unknown  $\mathbf{x}$  of problem (3) is modeled as a Numpy ndarray of proper size [14]. In `tvopt`, then, sets are built to be compatible with NumPy's ndarrays, and to use their broadcasting rules.

*Remark 1:* We remark that the most commonly used domains in online optimization are  $\mathbb{R}^n$  and  $\mathbb{R}^{n_1 \times n_2}$ , the latter for example can be used for image processing without the need to vectorize. The definition of `Set` objects with more than two dimensions can however be useful as well, for example in the distributed scenario discussed in section IV.

The other element of a `Set` definition is its `contains` method, which returns `True` if an input  $\mathbf{x}$  is in the `Set`, `False` otherwise. When defining a `Set`, the `projection` method should also be implemented, which, given an input  $\mathbf{x}$ , returns its projection onto the set, defined as

$$\text{proj}_{\mathbb{C}}(\mathbf{x}) = \arg \min_{\mathbf{y} \in \mathbb{C}} \|\mathbf{y} - \mathbf{x}\|^2$$

with  $\mathbb{C}$  the `Set`.

Finally, `Set` objects provide a `check_input` method which verifies if a given array  $\mathbf{x}$  fits the dimension of the set (possibly reshaping it). This is useful to implement validity checks on the inputs inside a `Cost` method (see section III-B).

*Operations:* The `contains` method can be accessed using the Python reserved keyword `in`. `Set` objects can be modified via the `scale` and `translate` methods. We can also define intersections of `Sets` by summing them (that is, using the `+` operator), in which case an approximate projection onto the intersection is implemented using the method of alternating projections (MAP) [15]<sup>2</sup>.

*Built-ins:* Different `Sets` are implemented, for example: the whole space  $\mathbb{R}^{n_1 \times n_2 \times \dots}$ , ball and box sets, and half-spaces. A particular built-in set is `T`, which defines the set  $\{t_k \in \mathbb{R}_+, k \in \mathbb{N}\}$  of sampling instants.

### B. Costs

The sub-module `costs` implements the `Cost` object to define time-varying cost functions

$$F : \mathbb{R}^{n_1 \times n_2 \times \dots} \times \mathbb{R}_+ \rightarrow \mathbb{R} \cup \{+\infty\}$$

or, as a sub-case, static costs. A cost is characterized by the `dom` and (optionally) `time` attributes, which point to `Sets` for  $\mathbb{R}^{n_1 \times n_2 \times \dots}$  and the sampling times  $\{t_k \in \mathbb{R}_+, k \in \mathbb{N}\}$ .

`Cost` objects are then defined by the function, `gradient` and `hessian` methods, where `gradient` returns a (sub-)gradient evaluation and `hessian` is implemented only if the cost is twice differentiable.

For example, we evaluate the (sub-)gradient of a function  $F$  as  $F.\text{gradient}(\mathbf{x}, t)$ .

The costs then provide a `proximal` method, which computes:

$$\text{prox}_{\rho F(\cdot; t_k)}(\mathbf{x}) = \arg \min_{\mathbf{y} \in \mathbb{R}^{n_1 \times n_2 \times \dots}} \{F(\mathbf{y}; t_k) + \|\mathbf{y} - \mathbf{x}\|^2 / (2\rho)\}$$

using either a gradient or Newton method, depending on the smoothness of  $F$ . If a closed form proximal is available, e.g.

<sup>2</sup>We remark that MAP returns a point in the intersection, not the actual projection. However, in practice MAP is faster than methods that are proven to return the projection, see [15].

for quadratic costs or  $\ell_1$  norms, then this method should be overwritten.

Time-varying costs provide the `time_derivative` method which computes, using backward finite differences [16], derivatives of  $F$  (or of its gradient and Hessian) w.r.t. time. For example the time-derivative of the gradient is approximated by

$$\hat{\nabla}_{t\mathbf{x}} F(\mathbf{x}; t_k) = (\nabla_{\mathbf{x}} F(\mathbf{x}; t_k) - \nabla_{\mathbf{x}} F(\mathbf{x}; t_{k-1})) / T_s.$$

Further, the costs have a `sample` method that returns a static cost representing  $F$  at a chosen sampling instant  $t_k$ .

*Operations:* Costs can be scaled by a scalar and elevated to a given power, and they can be summed and multiplied by other costs.

*Built-ins:* Some of the `Cost` objects that are implemented are  $\|\cdot\|_1$ ,  $\|\cdot\|_\infty$ , quadratic cost, Huber loss, and the indicator function of any given `Set`. The benchmark dynamic costs proposed in [6, section IV.A] and [17, section III.B] are implemented. Dynamic costs can also be defined from a sequence of static costs using `DiscreteDynamicCost`.

### C. Prediction

The sub-module `predictions` implements the `Prediction` object for approximating future costs from previously sampled costs. The object stores a dynamic `Cost` to be predicted and, through the method `update`, uses information on the cost up to a specified time  $t_k$  to shape a prediction. The object behaves like a static cost, in the sense that it exposes the `function`, `gradient`, *etc.* methods of the (static) predicted cost.

*Built-ins:* The sub-module implements the Taylor expansion-based and extrapolation-based prediction strategies studied in [11].

### D. Solvers

The sub-module `solvers` implements a selection of algorithms for solving different classes of static problems. The solvers are Python functions that are passed a static problem (in the form of a dictionary), the number of iterations to be applied, and any required parameters, such as step-sizes. The functions then return an approximate solution.

All the solvers provided by `tvopt` are not tailored to any specific choice of cost functions. Instead, they are defined to exploit the common template of `Cost` objects by calling e.g. their `gradient`, without needing to know how `gradient` is actually computed.

*Remark 2:* Notice that the modular design of solvers allows to define costs that for example inexactly compute `gradient` using a zero-th order approximation, without the need to implement a different solver.

There are two implementation choices that underlie the `solvers` module. First of all, solvers are designed to solve a static problem, since in `tvopt` we model a time-varying problem as a sequence of static, sampled ones. We recall that dynamic costs provide the `sample` method. As a by-product, this also allows to employ `tvopt` for prototyping and benchmarking static optimization algorithms.

The second design choice is to define solvers as functions, rather than objects, in order to provide a more flexible and

efficient implementation. Indeed, in the course of solving (3) a solver will be applied to several static problems and (possibly) using different parameters for each of them. As a consequence, defining a solver object is not very different from using a function, since the attributes (*e.g.* problem and parameters) of the object would need to be changed often, cluttering the syntax.

**Metrics:** The sub-module `utils` provides the implementation of different metrics for evaluating the performance of a solver. Let  $\{\mathbf{x}_k\}_{k \in \mathbb{N}}$  be the sequence generated by a solver applied to (3). The available metrics are: *fixed point residual* defined as  $\{\|\mathbf{x}_k - \mathbf{x}_{k-1}\|\}_{k \geq 1}$ ; *tracking error* defined as  $\{\|\mathbf{x}_k - \mathbf{x}^*(t_k)\|\}$ ; and *regret* defined as  $\{\frac{1}{k+1} \sum_{j=0}^k F(\mathbf{x}_j; t_j) - F(\mathbf{x}^*(t_j); t_j)\}_{k \in \mathbb{N}}$ .

**Built-ins:** Examples of built-in solvers are gradient method, proximal point algorithm, forward-backward<sup>3</sup> and Peaceman-Rachford splittings, dual ascent, ADMM. The documentation [13] lists all solvers with appropriate references.

#### E. Constrained optimization

We conclude this section by discussing how `tvopt` can be used to solve online constrained optimization problems.

A first class of constraints that can be implemented is  $\mathbf{x} \in \mathbb{C}$  where  $\mathbb{C}$  is a non-empty, closed, and convex set. Indeed, given a `Set` object representing  $\mathbb{C}$ , we can define the indicator function of the set using an `Indicator` cost object. The indicator is 0 for  $\mathbf{x} \in \mathbb{C}$  and  $+\infty$  for  $\mathbf{x} \notin \mathbb{C}$ , and its proximal operator coincides with a projection onto  $\mathbb{C}$ . Indicator functions can for example appear as the non-smooth term  $g$  in a composite optimization problem  $F(\mathbf{x}; t) = f(\mathbf{x}; t) + g(\mathbf{x}; t)$ .

Equality and inequality constraints  $g_i(\mathbf{x}; t) = 0$ ,  $i = 1, \dots, m$ , and  $h_i(\mathbf{x}; t) \leq 0$ ,  $i = 1, \dots, p$  can also be defined making use of `Cost` objects. The costs can then be used to define the Lagrangian of the constrained problem in order to apply primal-dual solvers [18].

A particular class of constrained problems that can be modeled using `tvopt` is the following:

$$\begin{aligned} \mathbf{x}^*(t_k), \mathbf{y}^*(t_k) = & \arg \min_{\mathbf{x} \in \mathbb{R}^n, \mathbf{y} \in \mathbb{R}^m} \{f(\mathbf{x}; t_k) + g(\mathbf{y}; t_k)\} \\ \text{s.t. } & \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{y} = \mathbf{c} \end{aligned} \quad (4)$$

where  $\mathbf{A} \in \mathbb{R}^{p \times n}$ ,  $\mathbf{B} \in \mathbb{R}^{p \times m}$ ,  $\mathbf{c} \in \mathbb{R}^p$ . As discussed in [11] and references therein, problem (4) can be solved by formulating its dual and applying suitable solvers to it. `tvopt` provides the following *dual* solvers: dual ascent, method of multipliers, ADMM, and dual forward-backward splitting. Moreover, a distributed version of dual ascent and ADMM are also implemented.

Notice that the constraints data  $\mathbf{A}$ ,  $\mathbf{B}$ ,  $\mathbf{c}$  can be defined using NumPy's `ndarrays`, owing to the fact that the unknowns of an optimization problem are modeled as compatible arrays.

## IV. DISTRIBUTED ONLINE OPTIMIZATION

This section describes the features of `tvopt` that allow to model and solve distributed online problems. As in the case

of centralized problems, we consider a sequence of samples from (2):

$$\begin{aligned} \mathbf{x}^*(t_k) = & \arg \min_{\mathbf{x}_i \in \mathbb{R}^n} \sum_{i=1}^N F_i(\mathbf{x}_i; t_k) \\ \text{s.t. } & \mathbf{x}_i = \mathbf{x}_j, \text{ if } i, j \text{ connected.} \end{aligned} \quad (5)$$

#### A. Networks

The sub-module `networks` defines the `Network` objects that model the connectivity pattern of a multi-agent system cooperating towards the solution of (5). The network is created from a given adjacency matrix.

A network implements the exchange of information between agents via the methods `send` and `receive`. In particular, `send` is called specifying a sender, receiver, and the packet to be exchanged. After calling `send`, the transmitted packet can be accessed using `receive`, which by default performs a destructive read of the message.

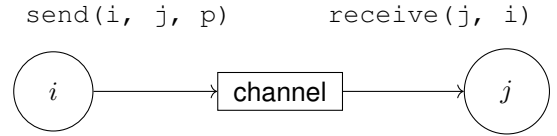


Fig. 2. A scheme representing agent-to-agent communication, with node  $i$  sending packet  $p$  to node  $j$ . The channel may be for example lossy or noisy.

The network implements also a `broadcast` method, using which an agents transmits the same message to all its neighbors. And a `consensus` method, which performs a consensus mixing of given local states using the `send` and `receive` methods.

In general, to define a new type of network (see *e.g.* the built-ins) it is sufficient to overwrite the `send` method.

**Built-ins:** The built-in `Network` class models a standard loss-less network. Other types of networks available are a lossy network (transmissions may randomly fail), and noisy or quantized networks (which add Gaussian noise to or quantize the transmissions, respectively).

The sub-module provides also a number of built-in functions for generating the adjacency matrix of different types of graphs, for example: random, circulant or complete graphs.

#### B. Costs

Formulating distributed optimization problems is done using the `SeparableCost` object defined in `costs`, which models a cost  $F : \mathbb{R}^{n_1 \times n_2 \times \dots \times N} \times \mathbb{R}_+ \rightarrow \mathbb{R} \cup \{+\infty\}$ :

$$F(\mathbf{x}; t) = \sum_{i=1}^N F_i(\mathbf{x}_i; t)$$

with  $F_i$  the local cost function of the  $i$ -th agent. The cost is created from a list of static or dynamic local costs. Notice that the last dimension of  $F$ 's domain is the number of agents, using the flexibility of `Set` objects that allow for multiple dimensions.

`SeparableCost` implements all the methods described in section III-B, with the difference that the outputs are arranged in an array with the last dimension indexing the

<sup>3</sup>Also called proximal gradient method.

agents. This choice allows for an easier access of the evaluation of each cost  $F_i$ . For example, if  $F$  is separable, then the result of  $F.function(x, t)$  will be  $[F_1(x_1, t), \dots, F_N(x_N, t)] \in \mathbb{R}^{1 \times N}$ .

A SeparableCost also allows to evaluate *e.g.* the gradient of a single component function  $F_i$  by specifying the argument  $i$ .

### C. Solvers

The sub-module `distributed_solvers` then provides built-in implementations of several distributed solvers. The difference with the centralized solvers described in section III-D is that these functions also require to be passed a Network object to perform agent-to-agent communications.

The built-in solvers are primal methods, *e.g.* DPGM [19]; primal-dual methods based on gradient tracking strategies, *e.g.* [20], [21]; and dual methods, *e.g.* dual decomposition and ADMM [22].

`tvopt` also provides different functions to solve *average consensus* using different protocols, for example gossip consensus [23].

## V. NUMERICAL EXAMPLES

The following sections presents a centralized example of online optimization benchmark and an example of distributed linear regression. A step-by-step discussion of the code is presented alongside some numerical results.

### A. Benchmark example

The benchmark problem was proposed in [6, section IV.A] and is characterized by

$$F(x; t) = (x - \cos(\omega t))^2 / 2 + \epsilon \log(1 + \exp(\varphi x))$$

with  $\omega = 0.02\pi$ ,  $\epsilon = 7.5$ , and  $\varphi = 1.75$ . We test the prediction-correction framework (see [11]) using the extrapolation-based prediction  $\hat{F}(x; t_{k+1}) = 2F(x; t_k) - F(x; t_{k-1})^4$ .

*Setup:* Defining the cost requires fixing the sampling time  $T_s$  and a time horizon.

```
from tvopt import costs, prediction, solvers

# sampling time and time horizon
t_s, t_max = 0.1, 1e4

# cost function
f = costs.DynamicExample_1D(t_s, t_max)
```

We also define the simulation parameters, with `num_pred` and `num_corr` representing  $N_P$  and  $N_C$ . The solver we will use is a gradient method, so we define its step-size.

```
# num. of prediction and correction steps
num_pred, num_corr = 5, 5

step = 0.2 # gradient method's step-size
```

*Prediction:* We define the extrapolation-based prediction (cf. Example 4) with

```
f_hat = prediction.ExtrapolationPrediction(f, 2)
```

where the argument 2 specifies the number of past costs to use for computing  $\hat{f}$ .

<sup>4</sup>The alternative Taylor expansion-based prediction is also implemented in the examples section of [1].

*Solution:* We then apply the prediction-correction solver as follows.

```
x, x_hat = 0, 0

for k in range(f.time.num_samples):

    # correction
    p = {"f": f.sample(t_s*k)} # correction problem
    x = solvers.gradient(p, x_0=x_hat, step=step,
                        num_iter=num_corr)

    # prediction
    f_hat.update(t_s*k)
    p = {"f": f_hat} # prediction problem
    x_hat = solvers.gradient(p, x_0=x, step=step,
                           num_iter=num_pred)
```

In the code, we update  $x_k$  and  $\hat{x}_k$  during the correction and prediction steps, respectively. Moreover, notice that the dynamic cost  $f$  is sampled every iteration, and that the prediction  $\hat{f}$  is consequently updated. The correction and prediction problem are defined with a dictionary.

Figure 3 depicts the evolution of the tracking error  $\{\|x_k - x^*(t_k)\|\}_{k \in \mathbb{N}}$  for the prediction-correction method discussed above. The method is compared with a correction-only strategy that does not employ a prediction to warm-start the solver at each sampling time.

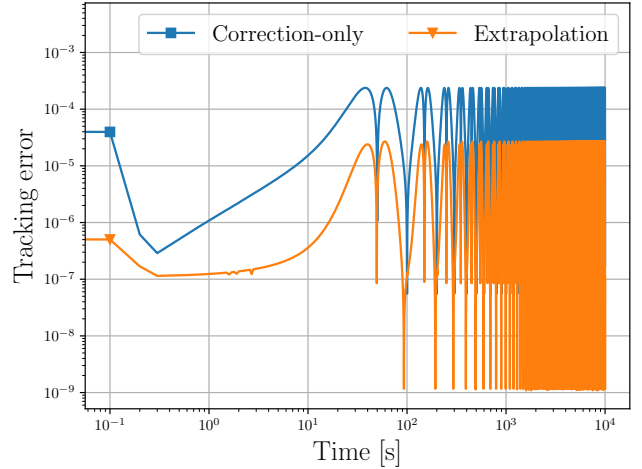


Fig. 3. Tracking error for different online optimization methods.

The results show that prediction has a valuable effect in improving the performance of the online solver, and `tvopt` provides an easy way to experiment with prediction strategies.

### B. Distributed linear regression

The problem is (5) with  $N = 25$  agents and with local costs

$$F_i(x_i; t) = (a_i x_i - b_i(t_k))^2 / 2$$

where  $b_i(t_k) = a_i y(t_k) + e_i(t_k)$  and  $y(t_k)$  is a sinusoidal signal with  $e_i(t_k)$  a Gaussian noise of variance  $\sigma^2 = 10^{-2}$ . The solver employed is DGD [24]. We report a sample of the code in the following.

The network can be created as follows:



```
# adjacency matrix
adj_mat = networks.random_graph(N, 0.5)
# network
net = networks.Network(adj_mat)
```

and the distributed, online solver is implemented with:

```
x = x0 # initial condition
for k in range(f.time.num_samples):
    # problem creation
    problem = {"f":f.sample(t_s*k), "network":net}
    # distributed solver
    x = distributed_solvers.dpgm
        (problem, step, x_0=x, num_iter=num_iter)
```

In Figure 4 we report the fixed point residual (defined as  $\{\|x_k - x_{k-1}\|\}_{k \in \mathbb{N}}$ ) for different graph topologies. We remark that the random graph has  $\sim 225$  edges and thus is the more connected of the four topologies, which explains the fact that it achieves the better results.

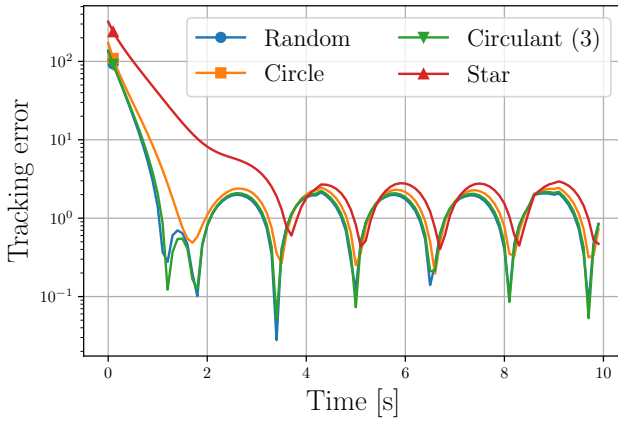


Fig. 4. Fixed point residual  $\{\|x_k - x_{k-1}\|\}_{k \in \mathbb{N}}$  for different graph topologies.

#### ACKNOWLEDGMENT

The author would like to thank Dr. Andrea Simonetto, Prof. Ruggero Carli, and Prof. Emiliano Dall’Anese for the many valuable discussions.

#### REFERENCES

- [1] N. Bastianello, “Code for tvopt,” <https://github.com/nicola-bastianello/tvopt>, 2020.
- [2] S. Shalev-Shwartz, “Online Learning and Online Convex Optimization,” *Foundations and Trends® in Machine Learning*, vol. 4, no. 2, pp. 107–194, 2011.
- [3] E. Dall’Anese, A. Simonetto, S. Becker, and L. Madden, “Optimization and Learning With Information Streams: Time-varying algorithms and applications,” *IEEE Signal Processing Magazine*, vol. 37, no. 3, pp. 71–83, May 2020.
- [4] A. Simonetto, E. Dall’Anese, S. Paternain, G. Leus, and G. B. Giannakis, “Time-Varying Convex Optimization: Time-Structured Algorithms and Applications,” *Proceedings of the IEEE*, vol. 108, no. 11, pp. 2032–2048, Nov. 2020.
- [5] E. C. Hall and R. M. Willett, “Online Convex Optimization in Dynamic Environments,” *IEEE Journal of Selected Topics in Signal Processing*, vol. 9, no. 4, pp. 647–662, June 2015.
- [6] A. Simonetto, A. Mokhtari, A. Koppel, G. Leus, and A. Ribeiro, “A Class of Prediction-Correction Methods for Time-Varying Convex Optimization,” *IEEE Transactions on Signal Processing*, vol. 64, no. 17, pp. 4576–4591, Sept. 2016.

- [7] Q. Ling and A. Ribeiro, “Decentralized Dynamic Optimization Through the Alternating Direction Method of Multipliers,” *IEEE Transactions on Signal Processing*, vol. 62, no. 5, pp. 1185–1197, Mar. 2014.
- [8] A. Simonetto, A. Koppel, A. Mokhtari, G. Leus, and A. Ribeiro, “Decentralized Prediction-Correction Methods for Networked Time-Varying Convex Optimization,” *IEEE Transactions on Automatic Control*, vol. 62, no. 11, pp. 5724–5738, Nov. 2017.
- [9] N. Bastianello, A. Simonetto, and R. Carli, “Distributed Prediction-Correction ADMM for Time-Varying Convex Optimization,” in *54th Asilomar Conference on Signals, Systems and Computers*, Nov. 2020.
- [10] A. Simonetto and E. Dall’Anese, “Prediction-Correction Algorithms for Time-Varying Constrained Optimization,” *IEEE Transactions on Signal Processing*, vol. 65, no. 20, pp. 5481–5494, Oct. 2017.
- [11] N. Bastianello, A. Simonetto, and R. Carli, “Primal and Dual Prediction-Correction Methods for Time-Varying Convex Optimization,” *arXiv:2004.11709 [cs, math]*, Oct. 2020. [Online]. Available: <http://arxiv.org/abs/2004.11709>
- [12] M. Fazlyab, S. Paternain, V. M. Preciado, and A. Ribeiro, “Prediction-Correction Interior-Point Method for Time-Varying Convex Optimization,” *IEEE Transactions on Automatic Control*, vol. 63, no. 7, pp. 1973–1986, July 2018.
- [13] N. Bastianello, “Documentation for tvopt,” <https://tvopt.readthedocs.io>, 2020.
- [14] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, “Array programming with NumPy,” *Nature*, vol. 585, no. 7825, pp. 357–362, Sept. 2020.
- [15] H. Bauschke and V. Koch, “Projection Methods: Swiss Army Knives for Solving Feasibility and Best Approximation Problems with Halfspaces,” in *Contemporary Mathematics*, S. Reich and A. Zaslavski, Eds. Providence, Rhode Island: American Mathematical Society, 2015, vol. 636, pp. 1–40.
- [16] A. Quarteroni, R. Sacco, and F. Saleri, *Numerical mathematics*, 2nd ed., ser. Texts in applied mathematics. Berlin ; New York: Springer, 2007, no. 37.
- [17] Y. Zhang, Z. Qi, B. Qiu, M. Yang, and M. Xiao, “Zeroing Neural Dynamics and Models for Various Time-Varying Problems Solving with ZLSF Models as Minimization-Type and Euler-Type Special Cases [Research Frontier],” *IEEE Computational Intelligence Magazine*, vol. 14, no. 3, pp. 52–60, Aug. 2019.
- [18] S. Boyd and L. Vandenberghe, *Convex optimization*. Cambridge university press, 2004.
- [19] N. Bastianello, A. Ajalloeian, and E. Dall’Anese, “Distributed and Inexact Proximal Gradient Method for Online Convex Optimization,” *arXiv:2001.00870 [math]*, Aug. 2020. [Online]. Available: <http://arxiv.org/abs/2001.00870>
- [20] W. Shi, Q. Ling, G. Wu, and W. Yin, “A Proximal Gradient Algorithm for Decentralized Composite Optimization,” *IEEE Transactions on Signal Processing*, vol. 63, no. 22, pp. 6013–6023, Nov. 2015.
- [21] S. A. Alghunaim, E. Ryu, K. Yuan, and A. H. Sayed, “Decentralized Proximal Gradient Algorithms with Linear Convergence Rates,” *IEEE Transactions on Automatic Control*, 2020.
- [22] N. Bastianello, R. Carli, L. Schenato, and M. Todescato, “Asynchronous Distributed Optimization over Lossy Networks via Relaxed ADMM: Stability and Linear Convergence,” *IEEE Transactions on Automatic Control (to appear)*, 2020.
- [23] T. Aysal, M. Yildiz, A. Sarwate, and A. Scaglione, “Broadcast Gossip Algorithms for Consensus,” *IEEE Transactions on Signal Processing*, vol. 57, no. 7, pp. 2748–2761, July 2009.
- [24] K. Yuan, Q. Ling, and W. Yin, “On the Convergence of Decentralized Gradient Descent,” *SIAM Journal on Optimization*, vol. 26, no. 3, pp. 1835–1854, Jan. 2016.