VirtualArena: An Object-Oriented MATLAB Toolkit for Control System Design and Simulation

Andrea Alessandretti[1], A. Pedro Aguiar[1] and Colin N. Jones[2]

Abstract

This paper presents an open-source object-oriented MATLAB toolkit for control system design and system simulation. The objective of the toolkit is to reduce the time required for the design and validation of a control architecture while at the same time increasing the reliability, modularity, and reusability of each of its components and fostering collaborative design and sharing of the developed components. To reduce the development time, a set of ready-to-use functions that are commonly required by control design processes is provided, such as automatic generation of Extended Kalman Filters, discretization, and many others. Moreover, we define a set of common interfaces to integrate the different standard components. The toolkit is introduced by means of a practical example, starting from the modeling of a planar Unmanned Aerial Vehicle, implementation of a two state-feedback controllers (one simple but nonlinear and another more complex using a Model Predictive Control approach), automatic generation of a state estimator, simulation, and remote network control over a Local Area Network.

I. Introduction

A common process for the design of model-based control algorithms can be summarized in the following design phases: first, a dynamic model of the system is designed that explains the evolution of the state and output of the system starting from an initial condition and subject to an input signal. In standard model-based control design, such model is then used to design a control input that steers the state, or output, of the system to the desired configuration. Often the controller requires information about the state of the system, but only output measurements are available. In this case, a state observer is designed to estimate the state of the system from the input and output signals. At this point, to assess the performance of the overall architecture via simulation, the closed-loop system is simulated with a more realistic model, e.g., including state and output disturbances, and with the state observer in order. If the control specifications are met, the controller is implemented on the real system, otherwise, the different components are re-designed and the overall process is repeated.

A correct implementation of such a process requires a wide set of expertise in many fields, but often, we wish to contribute to a specific aspect of the overall architecture and rely on experts for the other details. For instance, if we are interested in testing a novel control algorithm, we would like to relay on pre-existing accurate models, advanced state observers, and testing routines.

Although, implementing each component using works from the technical literature is often prohibitive in terms of time and know-how required for implementing algorithms relying on a strong technical background in specific fields. In this aspect, of great help are the occasional research groups producing toolboxes or making code publicly available online. Some notable contributions in the field of control are YALMIP [10] with the implementation of unified framework for optimization and optimization-based control/estimation design with interfaces to numerous software, ACADO [9] for optimization-based control/estimation design including control strategies for real-time control implementation, FORCES [6] with custom code generation for deployment of optimization-based controllers on various embedded platforms, MPT [8] with focus on for parametric optimization and computational geometry. Although, these tools are often specialized in the design of specific components of a control architecture and the simulation more complex control structure is still a challenge, e.g., multiple systems with state controllers, observers, measurement systems, etc.. On the other hand, advanced frameworks, e.g., [5] able to support experimental operations on multi-agent systems, are available. Although necessary for experimental validation, these frameworks often rely on low-level programming languages and complex architectures that can be unnecessarily difficult if one is interested in the design and analysis of a control strategy.

Motivated by these observations, the proposed toolkit provides a MATLAB framework for control system design and simulation. It consists of a set of object classes, defining common interfaces to connect standard components in the control architecture (such as of dynamical models, controllers, observers, etc.), as well as a set of ready-to-use implementations of such classes, including discretization methods, auto-generation of Extended Kalman Filters, and many other components.

In the proposed platform, fully exploiting the property of object-oriented design, it is possible to obtain a software architecture that is modular, easy-to-reuse and maintain, therefore fostering collaborative design of control algorithms, with multiple operators developing independent and easy-to-connect objects, that eventually results in software that is easy to extend and share.

[1] Faculty of Engineering, University of Porto (FEUP), Porto, Portugal. {andrea.alessandretti,pedro.aguiar}@fe.up.pt

[2] École Polytéchnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland. andrea.alessandretti@epfl.ch colin.jones@epfl.ch

This work was partially supported by FCT R&D Unit SYSTEC - POCI-01-0145-FEDER-006933/SYSTEC funded by ERDF—COMPETE2020—FCT/MEC—PT2020 and project STRIDE - NORTE-01-0145-FEDER-000033, funded by ERDF—NORTE 2020.

VirtualArena toolkit is available at the Github page [1].

The remainder of the paper is structured as follows: Section II presents an illustrative example starting from the model definition of an Unmanned Aerial Vehicle (UAV) to the design of a state-feedback controller, a state observer, simulation of closed-loop with the system subject to noise, and implementation on a remote system using a MATLAB-in-the-loop network control strategy. Section III illustrates the key benefit and main features of the VirtualArena toolkit, followed by Section IV with some conclusion and pointing out future developments.

II. Control system design and simulation in VirtualArena

This section introduces VirtualArena toolkit by illustrating all the steps for the implementation a common control architecture.

A. Model and state feedback controller definition

Let I be an inertial coordinate frame and B be a body coordinate frame attached to an UAV. The pair $(p(t), R(t)) \in SE(2)$[1] denote the configuration of the vehicle, position and orientation, where $R(t)$ is the rotation matrix from body to inertial coordinates. Now, let $(v(t), \Omega(\omega(t))) \in SE(2)$ be the twist that defines the velocity of the vehicle, linear and angular, where the matrix $\Omega(\omega(t))$ is the skew-symmetric matrix associated to the angular velocity $\omega(t) \in \mathbb{R}$, defined as $\Omega(\omega) := \begin{pmatrix} 0 & -\omega \\ \omega & 0 \end{pmatrix}$. The kinematic model of the body frame satisfies

$$\dot{p}(t) = R(t)v(t), \qquad \dot{R}(t) = R(t)\Omega(\omega(t)). \tag{1a}$$

The underactuation of the UAV is captured by choosing the control input

$$u(t) := \begin{pmatrix} v_1(t) & \omega(t) \end{pmatrix}', \tag{1b}$$

with $v(t) = \begin{pmatrix} v_1(t) & 0 \end{pmatrix}' \in \mathbb{R}^2$, that only consists of forward and angular velocity. The planar model (1) captures the under-actuated nature of a wide range of vehicles and is often used for the control of a set of UAV moving in at 2-D plane, i.e., relying on an on-board inner controller for altitude stabilization.

Consider the *control point* that we wish to control

$$c(t) := p(t) + R(t)\epsilon \tag{2}$$

that is a constant point in the body frame placed at a constant distance $\epsilon = (\epsilon_1, \epsilon_2)' \in \mathbb{R}^2$ from the center of rotation of the body frame B. For a desired trajectory $c_d : \mathbb{R} \to \mathbb{R}^2$ for the control point (2), it is possible to show (see [3], [2]) that the control law

$$u(t) = \Delta^{-1}(R'\dot{c}_d - Ke) \tag{3}$$

$$e(t) = R'(c(t) - c_d(t)) \tag{4}$$

with $\Delta = \begin{pmatrix} 1, & -\epsilon_2 \\ 0, & \epsilon_1 \end{pmatrix}$ for any $\epsilon_1 \neq 0$ stabilizes the origin of the tracking error space $e = 0$. The following code is used to simulated the closed-loop (1) with (3) with associated plots displayed in Fig. 1.

```
dt = 0.05;

%% Unicycle Model
sys = ICtSystem(...
    'StateEquation', @(t,x,u,varargin) [
    u(1)*cos(x(3));
    u(1)*sin(x(3));
    u(2)],...
    'nx',3,'nu',2 ...
);

sys.initialCondition = {[15;15;-pi/2],-[15;15;-pi/2],[15;-15;pi],[-15;15;-pi/2]};

sys.controller = TrackingController_ECC13(...
    @(t) 10*[sin(0.1*t); cos(0.1*t)],...% c
    @(t)    [cos(0.1*t);-sin(0.1*t)],...% cDot
    eye(2)                          ,...% K
    [1;0] ); % epsilon

va = VirtualArena(sys,...
    'StoppingCriteria'  , @(t,sysList)t>70,...
```

[1]For a given $n \in \mathbb{N}$, $SE(n)$ denotes the Cartesian product of $\mathbb{R}^n$ with the group $SO(n)$ of $n \times n$ rotation matrices and $se(n)$ denotes the Cartesian product of $\mathbb{R}^n$ with the space $so(n)$ of $n \times n$ skew-symmetric matrices.

```
22      'DiscretizationStep', dt,...
23      'PlottingStep'      , 1, ...
24      'StepPlotFunction'  , @ex01StepPlotFunction);
25
26  log = va.run();
```

It is worth noticing that the simulation is automatically executed multiple times, one for each initial condition of the system specified in Line 12. This is generally a desirable feature to test via simulation the behavior of the controller in different scenarios. The letter `I` in the name `ICtSystem` in Line 4, and in many other default classes in VirtualArena, stands for "inline", i.e., that can be defined inline without the need of creating a separate file. This is in contrast, for instance, to the controller `TrackingController_ECC13`. In fact, due to its complexity, the latter is not declared using the class `IController`, available in VirtualArena, but it is defined in the separate file `TrackingController_ECC13.m` reported next.

```
classdef TrackingController_ECC13 < Controller

properties
    epsilon,cdes,cdesDot,K,PinvE
end

methods
function obj = TrackingController_ECC13(cdes,cdesDot,K,epsilon)

    obj = obj@Controller();

    obj.cdes      = cdes;
    obj.cdesDot   = cdesDot;
    obj.K       = K;
    obj.epsilon = epsilon;

    E   = [[1;0],[-epsilon(2);epsilon(1)]];

    if not(rank(E)==size(E,1))
        error('System␣not␣controllable,␣try␣a␣different␣epsilon');
    end

    obj.PinvE = inv(E);
end

function u = computeInput(obj,t,x)
    R = [cos(x(3)),-sin(x(3));
         sin(x(3)),cos(x(3))];
    e = obj.computeError(t,x);
    u = -obj.PinvE*(obj.K*e-R'*obj.cdesDot(t));
end

function e = computeError(obj,t,x)
    p = x(1:2);
    R = [cos(x(3)),-sin(x(3));
         sin(x(3)),cos(x(3))];
    e = R'*(p-obj.cdes(t))+obj.epsilon;
end end end
```

The `StepPlotFunction` is defined in the separate file `ex01StepPlotFunction.m`.

```
function h = ex01StepPlotFunction(sysList,log,plot_handles,k)
  logX = log{1}.stateTrajectory(:,1:k);hold on;
  h    = plot(logX(1,:),logX(2,:));grid on;
end
```

By default, VirtualArena logs time, state trajectories, input trajectories, and whenever applicable, the trajectories associated with the internal state of the controller and state observer. Each log is associated with an object subclass of `Log`, e.g., by default VirtualArena loads the loggers, `TimeLog()`, `StateLog()`, `InputLog()`,`ControllerStateLog()`, and `ObserverStateLog()`, all subclass of `Log`. Custom logs can be defined for specific applications, e.g., to log tracking error, Lyapunov function value, etc.

B. *Output feedback and EKF design*

The previous example considers the state feedback case. This section assumes that the state of the vehicle is observed with the observation model $y(t) = p(t)$, i.e., that captures, for instance, the case where only the position, and not the heading, is available for control design. An Extended Kalman Filter (EKF, see, e.g., [4]) is designed on the vehicle model to estimate the state of the system, and then the same state feedback controller of the previous section is adopted for the motion control.

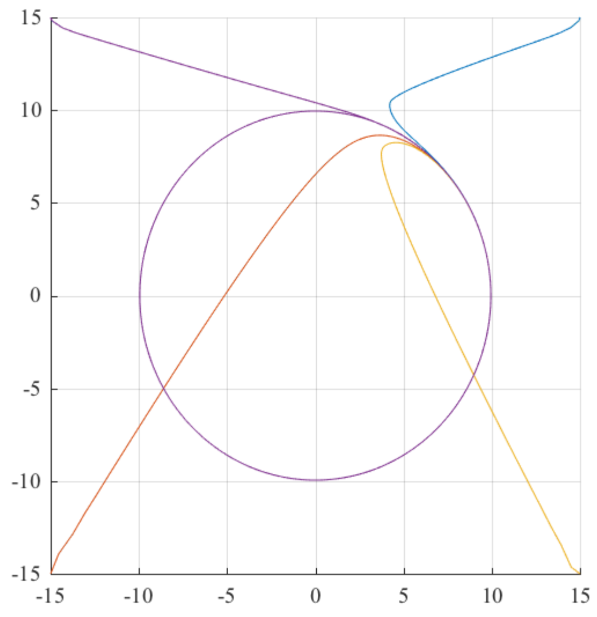

Fig. 1. Plot of simulation in Section II-A. Closed-loop state trajectories for the system simulated from four different initial conditions.

One way to define an output of the system in VirtualArena is to add the `OutputEquation` in the inline definition of the system as follows.

```
...
sys = ICtSystem(...
    ...
    'OutputEquation',@(t,x,varargin)x(1:2),...
    'ny',2,...
    ...
);
...
```

The automatic generation of the EKF with initial covariance matrix $P_0 = 10I_{3\times 3}$, state noise covariance matrix $Q = \text{diag}(([0.1, 0.1, \pi/4])/3)^2$, and output noise covariance matrix $R = \text{diag}(([0.1, 0.1])/3)^2$, where $D = \text{diag}(v)$ denotes a diagonal matrix D with $D_{ij} = v_i$ and $I_{n\times n}$ represents the identity matrix of size n, is obtained as follows.

```
...
% System with state and input noise covariance matrices
Q = diag(([0.1,0.1,pi/4])/3)^2;
R = diag(([0.1,0.1])/3)^2;

% Model with additive noise
realSystem = ICtSystem(...
    'StateEquation',@(t,x,u,varargin) sys.f(t,x,u,varargin{:})+chol(Q)*randn(3,1),...
    'OutputEquation',  @(t,x,varargin) sys.h(t,x,varargin{:})+chol(R)*randn(2,1),...
    'ny', 2,'nx',3,'nu',2);

% Initial conditions for the system ...
realSystem.initialCondition =  repmat({[15;15;-pi/2],-[15;15;-pi/2], [15;-15;-pi],[-15;15;-pi
    /2]},1,10);

% ... and associated initial conditions for the observer
for ii = 1:length(realSystem.initialCondition)
    x0Filter{ii} = [
    realSystem.initialCondition{ii}  +5*randn(3,1);         %xHat(0)
    10*reshape(eye(3),9,1) ]; %P(0)
end

realSystem.stateObserver = EkfFilter(...
  DiscretizedSystem(sys,dt),...
    'StateNoiseMatrix' , dt*Q,...
    'OutputNoiseMatrix', R,...
    'InitialCondition' , x0Filter);
```

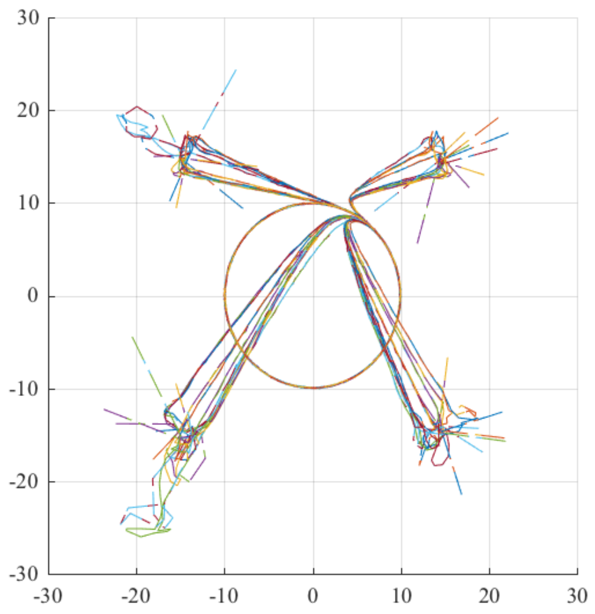

Fig. 2. Plot of simulation in Section II-B.Closed-loop state trajectories for the system with disturbance simulated from forty different initial conditions. Each simulation is associated with a different realization of the noise signal. The continuous line denotes the state of the system and the dashed line its estimate.

```
27
28  ...
29
30  realSystem.controller = TrackingController_ECC13(... % <<< attached to the realSystem
31  ...
32  va = VirtualArena(realSystem,... % <<< simulate realSystem
```

Since the Extended Kalman Filter is defined for discrete-time system, in Line 23 `DiscretizedSystem(sys,dt)` is used to discretized the system using Runge-Kutta 4, which is the default discretization method. Different, and possibly custom, discretization methods can be specified, e.g., `DiscretizedSystem(sys,dt,EulerForward())`. The object `EkfFilter` automatically computes the linearization of the system that will be evaluated by VirtuaArena around the closed-loop state and input trajectory. The processes of system discretization, linearization, and observer design for nonlinear systems are often involving and time-consuming steps toward the validation of control law. In this example, VirtualArena aims to minimize the associated effort to directly target the analysis of the control law.

In order to observe the effect of the noise the system is initialized utilizing the same set of initial conditions from the previous example repeated ten times with added noise, leading to a total of forty simulations.

C. *Model Predictive Control implementation*

Building on the previous examples, the controller `TrackingController_ECC13` is replaced with an Model Predictive Control (MPC) control law designed to drive the error vector (4) to the origin. Specifically, we consider the MPC tracking controller from [3] where the terminal cost and terminal set is re-designed as in [2] to obtain global convergence. The following definition of open-loop MPC problem is considered in the definition of the MPC controller.

Definition 1 (Open-Loop MPC Problem): Given a pair $(k, z) \in \mathbb{R}_{\geq k_0} \times \mathcal{X}(k)$ and an integer horizon length $N > 1$, the open-loop MPC optimization problem $\mathcal{P}(k, z)$ consists of finding the optimal control sequence $\bar{\boldsymbol{u}}^* := \{u^*(k), u^*(k+1), \ldots, u^*(k+N-1)\}$ that solves

$$\begin{aligned} J_N^*(k,z) = \min_{\bar{\boldsymbol{u}}} \; & J_N(k,z,\bar{\boldsymbol{u}}) \\ \text{s.t.} \quad & \bar{x}(i+1) = f(i,\bar{x}(i),\bar{u}(i),0), && i \in \mathbb{Z}_{k:k+N-1} \\ & (\bar{x}(i),\bar{u}(i)) \in \mathcal{X}(i) \times \mathcal{U}(i), && i \in \mathbb{Z}_{k:k+N-1} \\ & \bar{x}(k) = z, \\ & \bar{x}(k+N) \in \mathcal{X}_{aux}(k+N) \end{aligned}$$

with, $\bar{\boldsymbol{u}} = \{\bar{u}(k), \bar{u}(k+1), \dots, \bar{u}(k+N-1)\}$ and

$$J_N(k, z, \bar{\boldsymbol{u}}) := \sum_{i=k}^{k+N-1} l(i, \bar{x}(i), \bar{u}(i)) \\ + m(k+N, \bar{x}(k+N)), \tag{5a}$$

where the *finite horizon cost* $J_N(\cdot)$, which corresponds to the *performance index* of the MPC controller, is composed of the *stage cost* $l : \mathbb{Z}_{\geq k_0} \times \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$ and the *terminal cost* $m : \mathbb{Z}_{\geq k_0} \times \mathbb{R}^n \to \mathbb{R}$, where the last is defined over the time-varying *terminal set* $\mathcal{X}_{aux} : \mathbb{Z}_{\geq k_0} \rightrightarrows \mathbb{R}^n$. $\square$

In order to make explicit the dependence of the optimal solution to the parameters of the MPC open-loop optimization problem, the term $\bar{u}^*(i; \hat{k}, \hat{x})$ denotes the i-th input of the optimal input sequence $\bar{\boldsymbol{u}}^*$ computed by solving $\mathcal{P}(\hat{k}, \hat{x})$.

A standard MPC controller is obtained by solving at every time step $k \geq k_0$ the open-loop MPC problem and applying the first input of the optimal input trajectory to the system as follows

$$u(k) = k_{MPC}(k, x(k)) := \bar{u}^*(k; k, x(k)). \tag{6}$$

Following [2], choosing the performance index with

$$l(k, x, u) = \|e(t)\|^2, \qquad m(k, x, u) = 0.3333\|e(t)\|^3, \tag{7}$$

the input constraint set

$$\mathcal{U}(t) = \left\{ u : \begin{matrix} -1.1 \leq v_1 \leq 1.1 \\ -1.1 \leq \omega \leq 1.1 \end{matrix} \right\}$$

and the state constraint sets $\mathcal{X}(t) = \mathcal{X}_{aux}(t) = \mathbb{R}^3$ satisfies the sufficient conditions for global convergence of the error vector to the origin. In VirtualArena, this MPC controller can be defined as follows:

```
 1 ...
 2 auxiliaryControlLaw=TrackingController_ECC13( ... )
 3
 4 e=@(t,x)auxiliaryControlLaw.computeError(t,x);
 5
 6 mpcOp = ICtMpcOp( ...
 7     'System'              , sys,...
 8     'HorizonLength'       , 2*dt,...
 9     'StageConstraints'    , BoxSet( -[1.1;1.1],4:5,[1.1;1.1],4:5,5),... % on the variable z=[x;u];
10     'StageCost'           , @(t,x,u,varargin) e(t,x)'* e(t,x),...
11     'TerminalCost'        , @(t,x,varargin) 0.3333*(e(t,x)'* e(t,x))^(3/2)...
12     );
13
14 dtMpcOp = DiscretizedMpcOp(mpcOp,dt);
15
16 dtRealSystem=DiscretizedSystem(realSystem,dt);
17
18 dtRealSystem.controller = MpcController(...
19     'MpcOp'        , dtMpcOp ,...
20     'MpcOpSolver'  , FminconMpcOpSolver(...
21         'MpcOp', dtMpcOp,...
22         'UseSymbolicEvaluation',1...
23         ) ...
24     );
25 ...
26
27 va = VirtualArena(dtRealSystem,... % <<< Discrete time simulation
28     'StoppingCriteria'   , @(t,sysList)t>70/dt,...
29     'PlottingStep'       , 1/dt, ...
30     'StepPlotFunction'   , @ex02StepPlotFunction ...
31     );
```

The `MpcOpSolver` utilizes in this simulation is the `FminconMpcOpSolver` that is designed, building on the `fmincon` function of MATLAB, to solve discrete time MPC optimization problems. Because of this, in Line 17 the MPC optimization problem is discretized before being assigned as parameter to the `MpcController` in charge of implementing the receding horizon strategy (6).

It is worth noticing that the implementation of MPC scheme is generally a complex task that motivated a wide set of tools from the technical literature. In this example, VirtualArena is used to design in few lines of code an MPC controller for a constrained nonlinear system. Whereas in this implementation considers the `FminconMpcOpSolver` object to solve the MPC problem, VirtualArena defines the general interface `MpcOpSolver` that can be used to interface existing MPC solver from the literature, such as ACADO [9] and FORCES [6], or to custom ones.

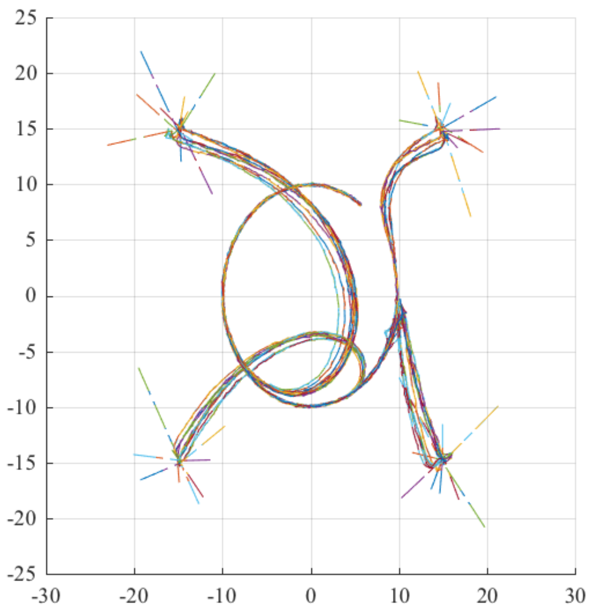

Fig. 3. Plot of simulation in Section II-C. Closed-loop state trajectories for the system with disturbance simulated from four different initial conditions, ten times each. Each simulation is associated with a different realization of the noise signal. The continuous line denotes the state of the system and the dashed line its estimate.

D. *Controlling remote system via UDP over LAN network*

In the previous sections, we were able with a small effort to start from a model of a UAV, generate an Extended Kalman Filter able to estimate the full state of the vehicle only using position measurements, and design and simulate an MPC controller to steer the position of the control point to a desired trajectory. In this section, we use the same code developed in the previous section to control a remote system over an Internet connection.

A basic network control strategy (NCS) consists of reading the measurements received from a remote sensor on board of the vehicle, computing the control input, sending the control input to a remote actuator, and iterating the process. Since VirtualArena captures the dynamics of the vehicle in a dedicated object, in order to implement a MATLAB-in-the-loop NCS strategy it is enough to replace the `realSystem`in the previous example with an object that, instead of simulating the dynamical model of the system, communicate with the real one.

This is done in the system `ex04RemoteUnicycle.m` available at the GitHub page of the toolbox that measures the position of the remote vehicle once every 0.2 sec with the 80% probability of measurement loss. Thus, it is enough to attach the previously designed controllers and observer to the `ex04RemoteUnicycle` as follows.

```
realSystem = ex04RemoteUnicycle(...
    'RemoteIp','127.0.0.1','RemotePort',20001,...
    'LocalIp' ,'127.0.0.1','LocalPort',20002);

realSystem.stateObserver =  EkfFilter(DtSystem(sys,dt),...
...
realSystem.controller = MpcController(...
```

At this point, before running the file `ex04runme_UnicycleWithEKFandMPCRealSystem.m` we can run the python script `ex04RealSystem.py` that emulates the real system.

Fig. 4 shows the associated closed-loop trajectory. Notice that, in contrast to the previous examples, the real state of the vehicle is not displayed, since not available, and only the measurements the state estimate is shown.

This section only presents an illustrative example of a basic NCS. Although, for the control of a real system with fast dynamics, delays, and dropouts in the communication link, it is opportune to design robust NCS schemes.

Model Predictive Control, with its ability to generate future state and input prediction, is particularly suitable for this kind of applications. In fact, instead of sending only the input associated with the current time, it is possible to send the whole state and input predicted trajectory to a "smart" actuator that will apply open-loop the input trajectory to the system (or track the associated state trajectory) till a new state-input trajectory pair is received. We refer the reader to, e.g., [7] for more insight on the topic.

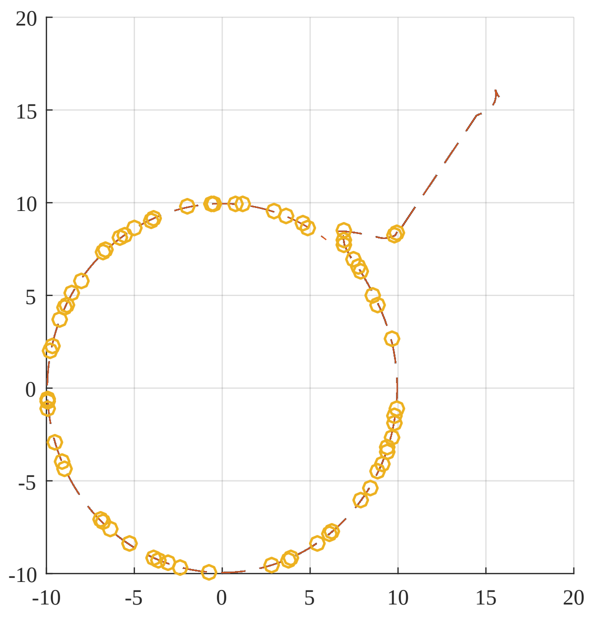

Fig. 4. Plot of simulation in Section II-D. The dashed line denote the state trajectory estimate and the circles denote the measurements received from the remote system.

E. Distributed control: a consensus algorithm

In this section, we deviate from the previous example of vehicle control to illustrate the design of a distributed controller where, in contrast to what discussed above, the control input is computed as a function of the local state and of the state of the neighbouring systems according to a pre-defined network topology.

Consider a set of $N > 0$ single integrator systems

$$\dot{x}(t) = u(t), x(t_0) = x_0, \qquad\qquad i = 1, \ldots, N. \tag{8}$$

where $x(t) \in \mathbb{R}$ and $u(t) \in \mathbb{R}$ are the state and the input vectors at time $t \geq t_0$, respectively, and x_0 is the initial state vector $x(t_0)$ evaluated at the initial time $t_0 \in \mathbb{R}$ of the generic agent i. The agents communicate among each others according to a communication graph $\mathcal{G} := (\mathcal{V}, \mathcal{E})$, where the vertex set $\mathcal{V}$ collects all the indexes of the agents, that is, $\mathcal{V} = 1, \ldots, N$, and the edge set $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is such that $(i, j) \in \mathcal{E}$ if and only if the system i can access $\gamma^{[j]}(t)$. Therefore, agent i can access from their neighborhoods $j \in \mathcal{N} := \{j : (i, j) \in \mathcal{E}\}$ at time t the coordination vectors $x_{\mathcal{N}}(t) := \{x^{[j]}(t) : j \in \mathcal{N}\}$. Moreover, let $A := [a_{ij}]$ be the adjacency matrix of the communication graph $\mathcal{G}$, such that $a_{ij} = 1$ for all $j \in \mathcal{N}$ and $a_{ij} = 0$ otherwise. Applying the consensus control law

$$u(t) = \sum_{j \in \mathcal{N}} (x^{[j]}(t) - x(t)) \tag{9}$$

it is possible to show that the state of all the system will converge to a common value, i.e., to consensus [11].

```
N = 5;

for i = 1:N

    v{i} = ICtSystem('StateEquation',@(t,x,u,varargin)u,'nx',1,'nu',1);
    v{i}.controller = ex05BasicConsensusController();
    v{i}.initialCondition = i;

end

%% Network
% Adjacency matrix - loop
A            = zeros(N);
A(1,4)       = 1;
A(2:N,1:N-1) = eye(N-1);

s1 = IAgentSensor(@(t,agentId,agent,sensedAgentIds,sensedAgents)sensedAgent.x);
```

```

Ah = @(t) A;

%% VirtualArena
a = VirtualArena(v,...
    'StoppingCriteria'  , @(t,as)t>10,...
    'SensorsNetwork'    , {s1,Ah},...
    'DiscretizationStep', 0.1,...
    'PlottingStep'      , 1);

ret = a.run();
```

Specifically, this class specifies the measurements that the `agent`, with associated `agentId`, obtains from the neighbouring agents `sensedAgents`, with the associated `sensedAgentIds`, obtained by VirtualArena from the, possibly time varying, adjacency matrix `A`. In the specific example of `ex5StateSensor`, each agent reads the state of the `detectableAgentsList`. Using this network measurement model, the controller (9) is implemented as follows.

```
classdef ex05BasicConsensusController < Controller
methods
    function u = computeInput(obj,t,x,readings)
        nNeigh = length(readings{1});
        u = 0;
        for i =1:nNeigh
            u = u+(readings{1}{i} - x)/nNeigh;
        end
    end
end
end
```

In general, is possible to specify multiple network sensors. In the latter case, `readings{i}{j}` refers to the j-th measurament provided by the i-th sensor.

III. Key benefits and main features

From the previous examples, it is possible to identify the main features and benefits of the proposed platform.

A. *Modularity, reusability, and maintenance*

VirtualArena promotes the development of a modular structure, where each component (e.g., a controller, a dynamical model...) is self-contained and satisfies pre-defined interfaces. Therefore, it is easy to switch among different components of the same kind, e.g., as illustrated in Section II-C, it is easy to switch among controllers. To give an example, each (state-less) controller in VirtualArena is defined as a subclass of the class `Controller` with abstract method `u = computeInput(t,x)`. In this case, each controller needs to implement the abstract method `u = computeInput(t,x)` which will be used by VirtualArena in the simulation phase, independently from the specific controller in hand. As a consequence, once an object is developed, the knowledge of its implementation is not necessary for its use. This abstraction allows to reuse/share components and easily maintain the architecture by simply replacing selected blocks. Fig. 5 shows a simplified class-diagram of the main classes of the toolbox.

B. *Collaborative design and test*

When multiple developers are involved in the design of a control architecture, the formalization of common interfaces is required to connect the different components of the architecture. VirtualArena defines such interfaces, facilitating the assignment of the design of modules to different developers.

C. *Dissemination and extensibility*

The effort and the technological and theoretical background required for the implementation of advanced control strategies are two of the main obstacles inhibiting the dissemination of new research results in industrial applications. VirtualArena, allowing the development of modular controllers, promotes the sharing of new algorithms among the users and provides functions to install external modules from third-party repositories. For instance, the following code will automatically retrieve a code from a third party archive located at the remote `URL` and import specific folders to the MATLAB path for future use.

```
vaInstall(URL)
```

D. *Implemented control-oriented common functions*

There are many common components/procedures required in the simulation/design of control systems (e.g., discretization procedures, linearization procedures, EKF design, data logging architecture, etc.). VirtualaArena comes with an increasing number of predefined functionalities devoted to making the development process easier. Next, a list the main functionalities currently available in VirtualArena are presented. We refer to the on-line/function documentation for the use of specific functions.

1) Simulation: The first set of features regards tools for the simulation of the control architecture.

- Time discretization methods. VirtualArena comes with different time discretization methods build-in such as `EulerForward`, `RK4`, `MatlabSolver`, that build on the `ode` solvers of MATLAB. Each discretization methods is defined as a subclass of `Integrator`. Other, and possible custom, discretisation methods can be created extending the latter class.
- Logging management system. Easy-to-use logging management system with custom logging modules subclass of the abstract class `Log`. Implemented logs are `TimeLog`, `StateLog`, `ObserverStateLog`, `MeasurementsLog`, `InputLog`, `ControllerStateLog`, `ILog`.
- Multiple simulations. Easy-to-use system to run multiple simulations with different initial conditions and initial settings (e.g., different initial conditions, different controller parameters, etc.) implementing the abstract class `MultiRun`.
- Multi-agent systems. Simulation of a communication/sensor network for multi-agent systems implementing the abstract class `Sensor`: `RangeFinder`, `AgentSensor`, `IAgentSensor`.
- Simultaneous simulation of a network of multiple vehicles.

2) System definition and manipulation: VirtualArena contains a set of interfaces to define dynamical systems provides a set of methods to manipulate these systems.

- Definition. Each system is defined as subclass of `GenericSystem`. The two main implementations of the dynamical systems are `CtSystem` and `DtSystem`.
- A `CtSystem` can be automatically discretized to create a `DtSystem`.
- A `DynamicalSystem`, either `CtSystem` or `DtSystem`, can be automatically linearized, where the computation of the jacobian matrices required for the linearization are automatically computed via Symbolic MATLAB or via sampling.

3) State estimation: As illustrated in Section II-B and using the linearization methods described above, VirtualArena provides automatic generation and simulations of state observers.

- Automatic generation of Extended Kalman Filter for discrete-time dynamical systems in `EkfFilter`.
- Automatic generation of Extended Kalman-Bucy Filter for continuous-time dynamical systems in `EkbfFilter`
- Support for custom observers.

4) Model predictive control: A set of functionalities is provided to define and customize MPC controllers.

- Definition of continuous-time and discrete-time MPC optimization problem, `CtMpcOp` and `DtMpcOp` respectively, subclasses of `MpcOp`.
- Definition of abstract class for MPC solver `MpcOpSolver` and warm-start strategies `WarmStart`.
- Implementation of a discrete-time MPC solver using `fmincon` in `FminconMpcOpSolver` (subclass of `MpcOpSolver`).
- Implementation of different warm-start strategies `ZerosWarmStart`, `ShiftAndAppendZeroWarmStart`, `AuxLawWarmStart`, `ShiftAndHoldWarmStart`, and `ShiftAndAppendAuxLawWarmStart`.

5) Motion control of underactuated vehicle: On the modelling of dynamical system, VirtualArena provides some models of underactuated vehicle with different state-space representations.

- Generic dynamical model representing `Unicycle` and the 3-D version `UAV` subclasses of `UnderactuatedVehicle`.
- Different representations of attitude using quaternions and rotation matrices available in `UnderactuatedVehicle`.

6) Supported controller: Both state-less controller and controller with internal dynamical model are supported.

- Controller without internal dynamical model subclass of `Controller`.
- Discrete time controller with internal dynamical model, subclass of `DtSytem`.
- Continuous time controller with internal dynamical model, subclass of `CtSytem`.

IV. Conclusion and Future Works

This paper introduced the VirtualArena toolbox highlighting main features, keys benefits, and illustrating by example the use of the toolbox for the simulation and single-agent/multi-agent systems and evaluation of control schemes via simulations.

Future works consider further extensions of the toolkit in the direction of multi-agent control and estimation as well as the production of interfaces for existing third-party toolbox of interest, such as, e.g., fast solver for MPC optimization problem.

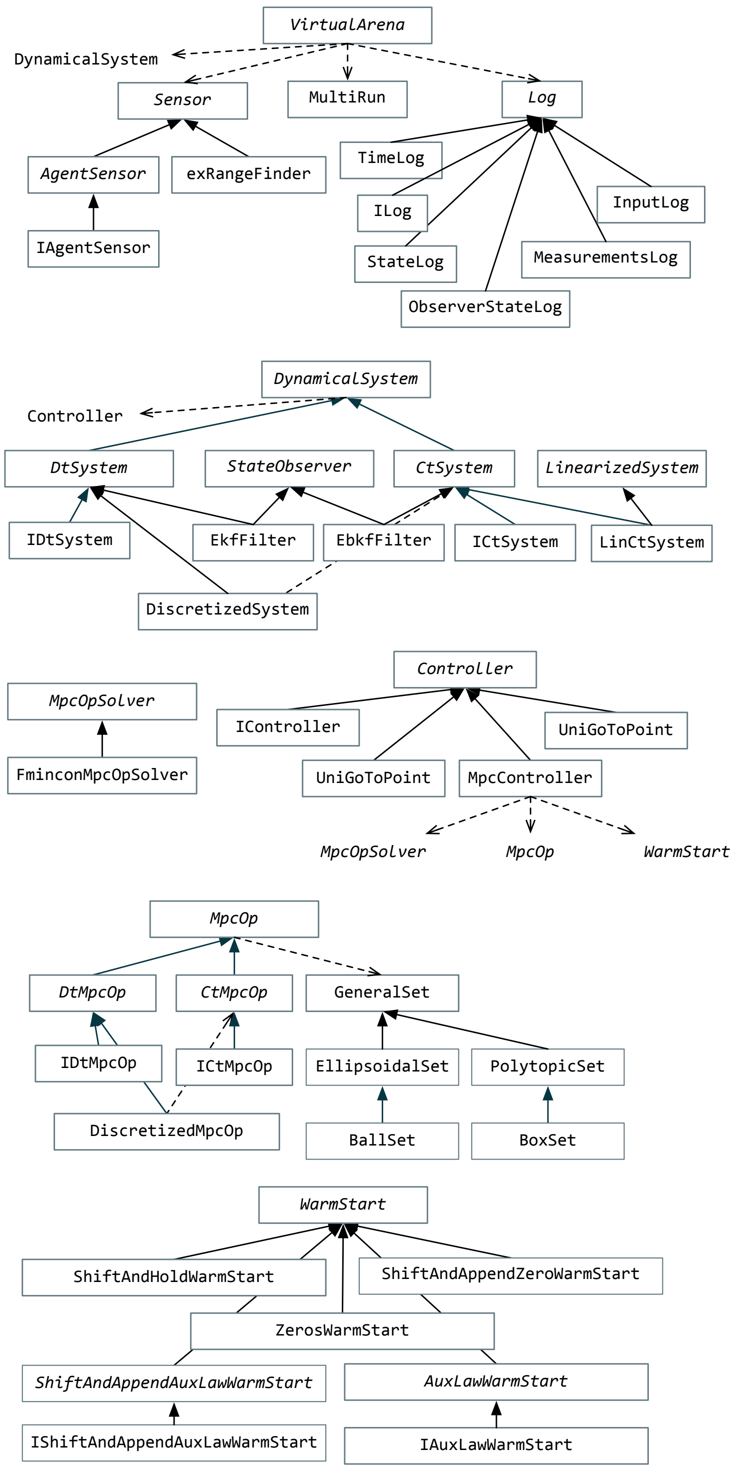

Fig. 5. Simplified class diagram of the main components of VirtualArena. The solid arrow denotes inheritance/implementation, where a class A points to a class B if A is a subclass of B and implements its abstract methods in the case of abstract class B (denoted in italic). The dashed arrow denotes dependency, where if A points to B then A “uses” B to function.

REFERENCES

[1] A. Alessandretti. VirtualArena Toolkit, Github page: github.com/andreaalessandretti/VirtualArena, 2014.

[2] A. Alessandretti and A. P. Aguiar. A distributed Model Predictive Control scheme for coordinated output regulation. In *Proc. of the 20th IFAC World World Congress*, Toulouse, France, 2017.

[3] A. Alessandretti, A. P. Aguiar, and C. N. Jones. Trajectory-tracking and path-following controllers for constrained underactuated vehicles using Model Predictive Control. In *Proc. of the 2013 European Control Conference*, pages 1371–1376, 2013.

[4] B. D. O. Anderson and J. B. Moore. *Optimal Filtering*, volume 16. Dover Publications, 2005.

[5] P. Dias, S. L. Fraga, R. Gomes, G. Gonçalves, F. L. Pereira, J. Pinto, and J. Sousa. Neptus - a framework to support multiple vehicle operation. In *Europe Oceans 2005*. IEEE, IEEE, 2005.

[6] A. Domahidi and J. Jerez. FORCES Professional, jul 2014.

[7] R. Findeisen and P. Varutti. Stabilizing nonlinear predictive control over nondeterministic communication networks. In *Nonlinear model predictive control*, pages 167–179. Springer, 2009.

[8] M. Herceg, M. Kvasnica, C. N. Jones, and M. Morari. Multi-parametric toolbox 3.0. In *Proc. of the 2013 European Control Conference (ECC),*, pages 502–510, 2013.

[9] B. Houska, H. J. Ferreau, and M. Diehl. ACADO toolkit-An open-source framework for automatic control and dynamic optimization. *Optimal Control Applications and Methods*, 32(3):298–312, 2011.

[10] J. Löfberg. YALMIP : A Toolbox for Modeling and Optimization in MATLAB. In *Proc. of the CACSD Conf.*, Taipei, Taiwan, 2004.

[11] R. Olfati-Saber, J. A. Fax, and R. M. Murray. Consensus and Cooperation in Networked Multi-Agent Systems. *Proceedings of the IEEE*, 95(1):215–233, jan 2007.