FONS: a Python framework for simulating nonlinear inflatable actuator networks

Arne Baeyens*, Bert Van Raemdonck*, Edoardo Milana, Dominiek Reynaerts and Benjamin Gorissen

Abstract— **Soft robots designed within a conventional robotic framework typically consist of individually addressable compliant actuators that are merged together into a deformable body. For inflatable soft robots, this comes at a high cost of tethering which drastically limits their autonomy and versatility. This cost can be decreased by connecting multiple actuators in a fluidic network and partially offloading control to the passive interactions within the network. This type of morphological control necessitates some of the elements in the network to have nonlinear characteristics. However a standardized simulation framework for such networks is lacking. Here, we introduce the open-source python library FONS (Fluidic object-oriented network simulator), a tool for simulating fluidic interactions in lumped fluidic networks of arbitrary size. It is compatible with both gaseous and liquid fluids and supports analytical, simulated and measured characteristics for all components. These components can be defined using a library of standard components or can be implemented as custom objects following a modular object-oriented framework. We show that FONS is capable of simulating a multitude of systems with highly nonlinear components exhibiting morphological control.**

I. Introduction

In traditional robots every degree of freedom is actuated by a dedicated motor, and control software dictates the goal trajectory in joint space. In soft robotics, it is challenging to reproduce a precise trajectory because every actuator is a continuous structure with infinite degrees of freedom with only a single controlled input [1]. However, underactuation also enables individual actuators to automatically adapt to the environment in useful ways in a paradigm called morphological control [2]. On the one hand, many soft robots embrace morphological control on the level of individual components. For example, soft bending actuators can grasp objects of unknown shape without sensory feedback. On the other hand, at the system level these robots often still follow the traditional architecture where every individual actuator is controlled explicitly. For inflatable soft robots, the weight of a plethora of pressure supply tubes and valves, that are necessary in this architecture, excessively loads the compliant body of the robot and is detrimental for the performance [3], [4]. To overcome these limitations, researchers have implemented morphological control on a system level as well, allowing multiple actuators to be controlled via a single input.

For systems of inflatable actuators, one source of morphological control are actuators with peak-and-valley pressure-volume (PV) curves. When connected to a pressure input, these actuators inflate in a discrete sequence in the order of increasing peaks and deflate in order of decreasing valleys [5]–[7]. This mechanism was applied in a self-propelling endoscope [8], a trotting robot [9], and artificial cilia [10]. Another source of morphological control can be found in the fluidic resistances between actuators. This resistance can be constant [11] or can change with the direction of the flow [12] and with the pressure differential over the component [13], which can be harnessed to generate locomotion from a single pressure input. A persistent trend in these systems is that more rich nonlinearities produce more complex behavior, allowing a greater reduction in the control hardware.

For all of these systems, designing them in function of a desired behavior requires precise tuning of the component characteristics. This tuning can be perfected experimentally by iteratively prototyping robots with different parameters (top arrow on Fig. 1), but that approach is often too costly and time intensive. Another approach is fully in-silico with multi-physics simulations featuring turbulent flows, large deformations and fluid-structure interactions, but such models are not mature enough to be integrated in a design loop. The most practical way to design fluidic systems with morphological control is through lumped modeling. In this framework, the robot is discretized into a network of components described by few variables. Usually, it is sufficient to only consider the fluidic interactions between these components governed by Kirchoff equations in terms of pressures and flows [7], [9]. These variables can be mapped to the resulting deformations using component specific mechanical models, but this does not affect the interaction between the components. Consequently, a lumped fluidic network model is highly modular so simulated and measured characteristics can be combined to produce accurate and fast simulations of the system over time (bottom arrow on Fig. 1).

Nearly all publications on morphologically controlled inflatable soft robots employ lumped network models to great success. However, each of these publications uses a dedicated simulator tailored to the specific needs of the analyzed systems. As an alternative, software packages like spice [14], modelica [15] and simulink [16] allow to analyze generalized networks, but they do not include domain knowledge on fluidic soft robots. In conclusion, no modeling tool exists with all of the following features necessary for the general design of nonlinear fluidic networks for morphological control:

- *(i)* Support for networks of arbitrary size and connectivity.
- *(ii)* Support for inflatable actuators [5], [11], [17], [18], interconnections [13], [19], sources and fluids with arbitrary properties
- *(ii)* Straightforward syntax and structure allowing non-experts to easily define new components and write interfaces to other programs.

* These authors contributed equally to this work.

All authors are with Department of Mechanical Engineering, KU Leuven, Leuven, Belgium and members of Flanders Make. Edoardo Milana is also with livMaTs@FIT, University of Freiburg, Freiburg, Germany

Corresponding author: benjamin.gorissen@kuleuven.be

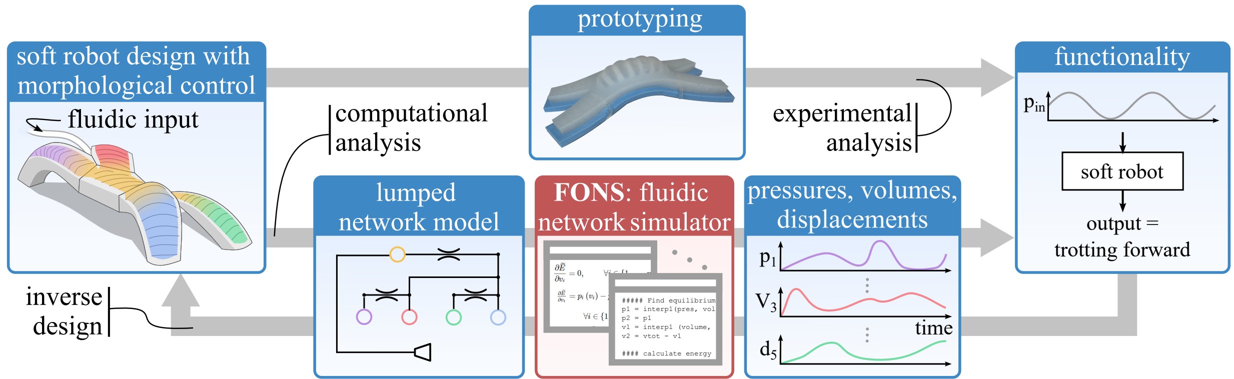

Fig. 1. Designing soft robots with morphological control. The response of a soft robot to a fluidic input can be characterized experimentally on a prototype (top) or computationally on a lumped network model (bottom). The latter approach is orders of magnitude faster than the former, which allows for inverse design of a robot with a certain desired functionality

To bridge this gap, we introduce FONS (Fluidic Object-oriented Network Simulator), an open-source Python toolbox that incorporates all these features. Given the relation between pressure and flow for each component in a soft robot, it predicts the change of these variables over time, which can be mapped to the deformation of the soft robot. Its use is limited to systems that can be clearly separated into components that interact with each other through Kirchoff equations but do not feature mechanical coupling or spatially varying flow fields at the interface. However, since this simplification applies for most soft robots, FONS is a key tool that assists soft robot researchers in designing soft robots that use morphological control in the fluidic domain to generate useful deformation sequences with a minimum of inputs. In the next section, we describe the internal structure of FONS by means of a practical example code. Afterwards, we present the results of a number of simulations run on a nonlinear fluidic network with FONS. They show how the majority of behaviors described in literature on morphological control can be reproduced by changing a few lines of code. Lastly we experimentally benchmark the performance of FONS using a recently published artificial cilia system that exhibits morphological control [10].

II. Methods

FONS takes inspiration from general lumped network simulators to create a specialized tool for soft robots. From these general simulators, FONS takes the network analysis theory and implements it in a transparent way by having a separate Python object for every concept found in that theory. This means that, in a top-down perspective, a simulation in FONS is carried out by a `Simulation` object operating on a `System` object. Such a system represents a fluidic network and is a collection of `Component` and `Node` objects. Components can be thought of as physical actuators, valves, sources, etc. while nodes are an abstract representation of a location where these objects are connected together. In FONS, a component thus has a set of `Terminal` objects which represent their input/output ports to the external world, while a node records which terminals are connected together. Finally, on the lowest level, each terminal features a pair of fundamental `Variable` objects. These fundamental variables are *through* variables, which sum up to zero for all terminals connected to a node, and *across* variables, which are equal for all terminals connected to a node.

The specialization of this general framework towards fluidic networks occurs at the level of the `Component` object. At this level, a straightforward definition of component characteristics can be done by using the through and across variables which for fluidic networks correspond to mass flow rate and pressure, respectively. Moreover, these characteristic can be modified based on the `Fluid` object that is attributed to the component, which is a unique feature of fluidic networks. As such different characteristics can be encoded based on the characteristic of this fluid object. Finally, FONS comes with a library of common fluidic components which can be used as is, or as building blocks to define new components thanks to the object-oriented implementation.

To explain how this works in practice, we analyze an elementary fluidic network consisting of a pressure source (one terminal), a pressure supply tube (two terminals) and an inflatable actuator (one terminal) (see Fig. 2). This network is modeled in FONS using a 44-line script (see section V), which we examine line by line in the following sections.

A. Initialization (lines 1-3)

FONS can be installed with PIP, the package installer for Python, using the command `pip install fonsim`. To use it, it has to be imported as is done on line 3 of the script.

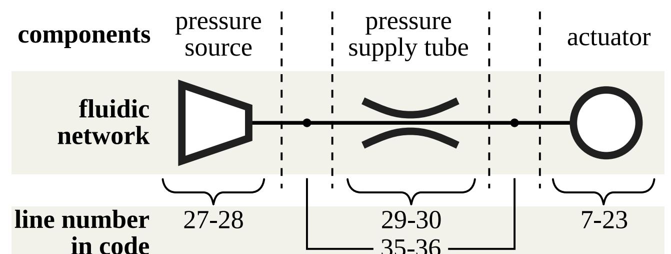

Fig. 2. Example of a rudimentary fluidic network that is analyzed using FONS. The Python code that is needed to generate this example can be found at the end of this paper, where 'line number in code' refers to.

B. Component class definition (lines 7-23)

For the purpose of illustration, the example script defines a simplified version of the `Actuator` component. In FONS, this definition consists of four steps. First, a class is created as a child of the generic `Component` class on lines 7-9. This generic class provides an interface to the fluidic system, which needs a unique label, and an interface to query the value of the component variables after simulation (line 44).

In the second step, the architecture of the component is defined by declaring the relevant variables and distributing them among terminals. Variables are defined by declaring the physical quantity that they represent (pressure, massflow and mass) and their role within the context of network analysis (lines 10-12). These roles can be as *through* and *across* variables for the terminals or as additional *local* variables. Local variables can act as notable intermediate quantities in the component equations (e.g. Reynolds number of flow in a tube) or can represent the internal state of a component (e.g. accumulated fluid mass in the `Actuator` class). Additional arguments in the definition of a variable are a label for ease of use later on and an optional value for initializing the solver. In the example, the pressure is initialized to atmospheric pressure (line 4) because FONS deals in absolute pressures. The *through* and *across* variables are then assigned to the terminal labeled `a` (line 14), while the *local* variables are stored directly in the component (line 15).

The third and fourth step consist of describing the internal relations between the defined variables. In general, these relations take the shape of a differential-algebraic system of equations. In FONS, the purely algebraic equations and the discretized equivalents of the differential equations in this system are implemented in two different functions. All purely algebraic equations are implemented in a function (lines 17-18) assigned to the component `evaluate` method (line 19). This function refers to variables by their labels and can access the simulation time through a variable `t`. Contrary to existing simulators for fluidic networks, all equations are expressed in a residual formulation , which supports implicit equations and is compatible with common numerical solvers. In the example, the `evaluate` method of the `Actuator` class contains a single equation interpolating a measured pressure-volume characteristic represented by a dedicated `PVCurve` class (line 32). Residuals can also be generated by mathematical functions in Python or by external scripts called from within the `evaluate` function. Moreover, because of the object-oriented framework, a component can inherit `evaluate` functions from one or multiple library components. These functions can then be reused or combined to easily build advanced components.

In the final step, all component relations expressed as ordinary differential equations (ode) are discretized and implemented in the `update_state` method (lines 21-23). This function returns the values in the next time increment for all component variables that were added with the `set_states` method (line 15). This means that all ode's have to be structured as a system of explicit first-order ode's and have to be numerically integrated over time step `dt`. In the example, line 22 corresponds to the equation $dm/dt = m_f$. For both the `evaluate` and `update_state` method, the FONS solver requires the derivative of the equations with respect to the input variables. The user can either provide an exact formula for this derivative manually or use the component methods `auto` (line 19) and `auto_state` (line 23) to generate a finite difference approximation for the derivatives of `evaluate` and `update_state`, respectively.

C. Component instances (lines 26-32)

Both custom classes and classes from the component library are parametrized templates, so they need to be instantiated with unique labels and concrete parameters before they are added to the system (lines 26-32). For the actuator, one such parameter is the PV characteristic defined by data in a csv file. The pressure source (lines 27-28) is a library component with as parameter a function of the simulation time `t` that returns a pressure profile in terms of absolute pressure. For the example, this profile is a step function. The flow restrictor (lines 29-30) is a library component for a circular tube with a given length and diameter. It implements the Darcy-Weisbach equation with the friction factor given by Poiseuille's law for laminar flow [9] and by the Haaland approximation [20] for turbulent flow. Depending on the fluid that flows through the tube, these equations have different parameters (e.g. viscosity) and forms (e.g. compressible or incompressible flow). Therefore, many components have multiple `evaluate` and `update_state` functions for different types of fluids. The appropriate equations are selected by methods of the `Component` class based on a provided `Fluid` object that contains both physical parameters and qualitative aspects (e.g. incompressibility) of a fluid. Standard fluids like water and air are built into FONS.

D. Component interconnections (lines 35-36)

When a component is added to a system, a `Node` object is generated for every component terminal. Connections between components are then created by merging different `Node` objects together using the `system.connect` method. The inputs of this method are the labels of components and optionally also of their terminals. In case no terminal labels are provided, FONS selects the first unconnected terminal of every component. Consequently, on line 36 in the example the actuator is automatically connected to the other node of the tube than the pressure source.

E. Simulation and results (lines 39-44)

The final step in a network analysis with FONS is to create a `Simulation` object on the `System` instance (line 39). The simulation first generates a mapping between all `Variable` objects present in the system and two vectors $\mathbf{x}_d$ and $\mathbf{x}_a$. $\mathbf{x}_d$ contains the values for all variables that are updated by the discretized differential equations in the `update_state` methods of the components and $\mathbf{x}_a$ contains the values for all other variables that are purely algebraic. Internally, these variables are referenced by the `states` and `arguments` attributes of the different components, respectively. Next, the simulation constructs a network matrix $\mathbf{J_n}$ implementing the network equations. This matrix remains constant throughout the course of the simulation. It

has as many columns as $\mathbf{x_a}$ has elements. For every `Node` object to which N terminals are connected, $\mathbf{J_n}$ contains one row stating that the sum of all N *through* variables should equal zero to conserve mass and contains $N-1$ rows equating all N *across* variables to each other.

When the simulation is started (line 40), $\mathbf{x}_a$ and $\mathbf{x}_d$ are updated iteratively by a `Solver` object associated with the simulation. In every iteration, the solver calls the `evaluate_equations` method of the `Simulation` object. This method implements an implicit Euler scheme by first updating $\mathbf{x}_d$ and then evaluating the component equation residuals with the updated variables using the `update_state` and `evaluate` method of every component, respectively. The resulting residual vectors are assembled in a single vector $\mathbf{f}$ together with the residual of the network equations $\mathbf{J}_n\mathbf{x}_a$. Apart from the residual, it also assembles the total derivative $\mathbf{J}_a$ of $\mathbf{f}$ with respect to $\mathbf{x}_a$ using the derivatives provided by all `evaluate` and `update_state` equations. Finally, `evaluate_equations` returns $\mathbf{f}$ and $\mathbf{J}_a$ to the solver which produces the next iteration of $\mathbf{x}_a$.

Since the `Simulation` object provides a straightforward interface to the system of equations, any nonlinear equation solver can be used at the back end. Out of the box, FONS comes with two `Solver` objects. One implements time stepping with a constant time increment. The other adapts the time increment based on the time derivatives of the variables and the convergence rate for increased robustness to component instabilities. Both `Solver` objects use the Newton-Raphson algorithm to solve the system of equations at every time instant. Because this algorithm is robust against noise on the provided derivatives, the solver converges even for unfiltered measurement data containing moderate amounts of noise. Finally, after running the simulation all time values are accessible in the `Simulation.times` attribute (line 43) and the values for the variables at those time instants can be queried using the `Component.get` method (line 44).

III. Results and Discussions

To illustrate the capabilities of FONS, we use it to analyze some model problems in Figs. 3–5. For all model problems we consider a network architecture with a volumetric flow source, two actuators and two flow restrictors connecting all components together (architecture in panel A). We subject this network architecture to nine different simulations. In every simulation, the source pumps fluid into the system at a constant volumetric flow rate of $1.8\,\mathrm{mL/s}$, then pauses, and finally pumps the fluid out of the system at the same rate. This input causes the internal volume of the two actuators V_{act1} and V_{act2} to increase and decrease in a certain sequence (panel C). In the D panels, we visualize this sequence by tracing the path in $\{V_{act1}, V_{act2}\}$-space over time for every simulation. If this path encloses an area, the sequence is asymmetric between inflation and deflation and otherwise it is symmetric. Because of the flexibility and robustness of FONS, swapping out the characteristics of individual components with a small amount of code yields a range of qualitatively different sequencing behaviors that covers the majority of literature on morphological control.

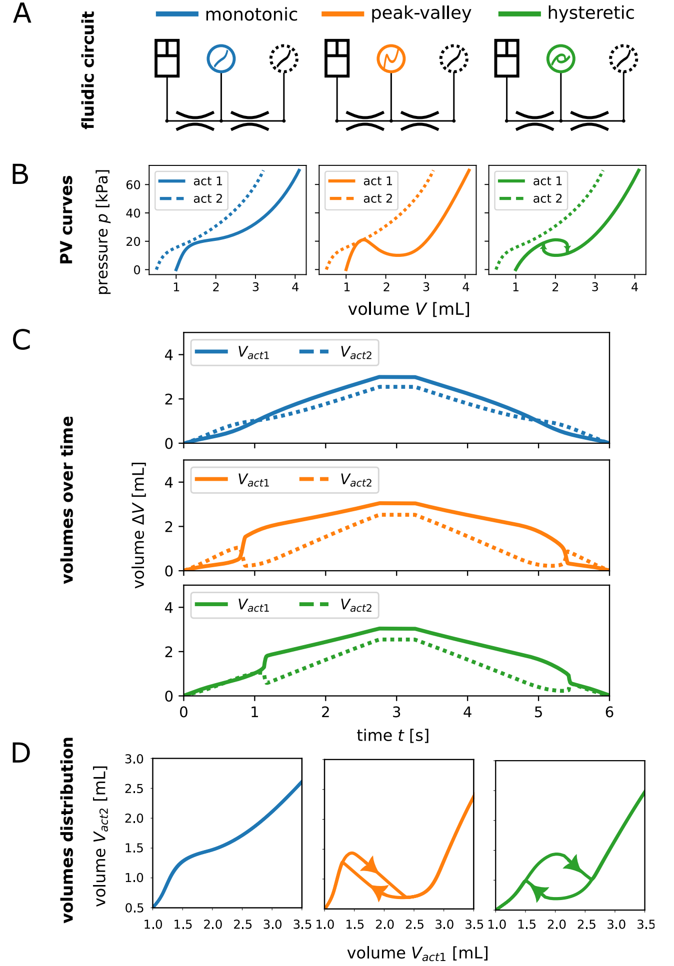

Fig. 3. FONS simulation results for a system with actuator characteristics encoded in a monotonic (blue), peak-valley (orange) and hysteretic (green) pressure-volume (PV) curve.

Fig. 3 presents three simulations illustrating the effect of changing the pressure-volume (PV) characteristic of a single actuator, which in FONS can be easily done by pointing to a different '.csv' file in the actuator instance creation. In all three simulations, the flow restrictors have a negligible resistance and actuator 2 has a monotonic PV characteristic typical for a soft inflatable bending actuator (Fig. 3B). If actuator 1 has a similar monotonic characteristic (blue), the actuators inflate in a symmetrical sequence determined by their relative compliances [21]. However, if the PV curve of actuator 1 features a peak and a valley (orange), unstable exchanges of volume between the actuators occur. Those instabilities occur at different points on inflation and deflation, so the symmetry is broken [5]. Finally, the asymmetry increases further if actuator 1 becomes hysteretic (green) with different PV curves for inflation and deflation separated by snapping events where volume is redistributed within the actuator itself. Here we refer to [18] for actuators that have this hysteretic behaviour. Despite the highly dynamic snapping events, the each simulation finishes within $0.5\,\mathrm{s}$.

Another way to obtain asymmetric sequences is to harness the dynamic pressure drop across a flow restrictor. This is illustrated in Fig. 4 for three systems where the PV curves for

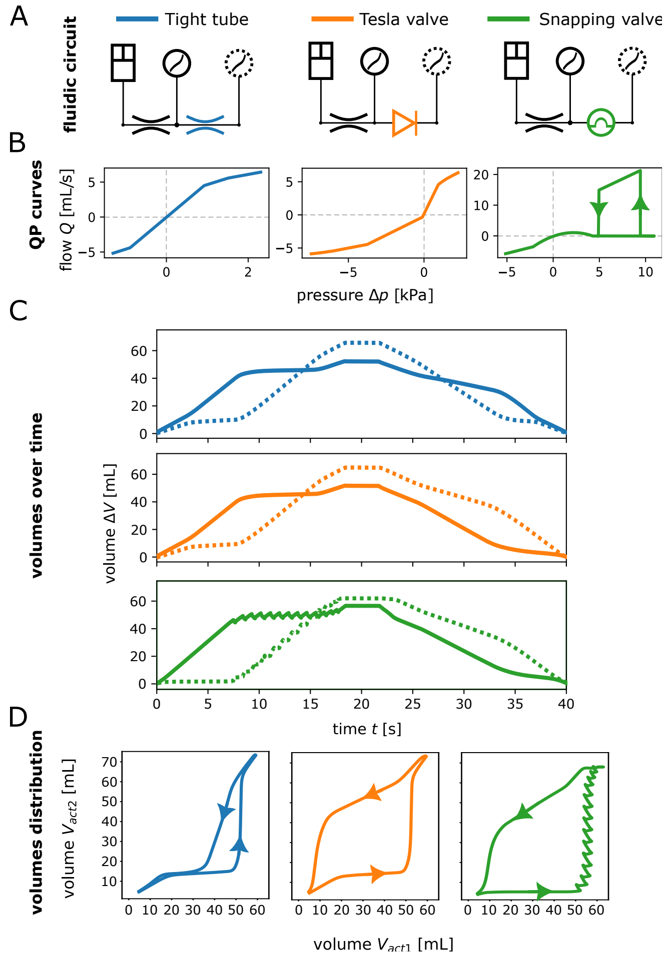

Fig. 4. FONS simulation results for a system where two actuators are connected by a tight tube (blue), an asymmetric tesla valve (orange) and a hysteretic snapping valve (green) with different relations between volumetric flow (Q) and pressure drop (Δp).

actuator 1 and 2 always feature a plateau (as in the monotonic system in Fig. 3) that is lower for 1 than for 2 and where the pressure-flow characteristic of the restrictor is varied. If the flow restrictor is modelled analytically as a tight circular tube with 2.5 mm ID (blue), the viscous energy dissipation in the tube creates a lag between V_{act1} and V_{act2}, which was used for sequencing in [11]. Next, we consider a Tesla valve with the same flow resistance as the circular tube if the fluid flows from actuator 1 to actuator 2, but a higher resistance if the fluid flows in the opposite direction (orange) [22]. This results in the same lag on inflation, but a larger lag on deflation. A third kind of flow restrictor is a snapping valve (green) which allows fluid to flow between the two actuators in bursts on inflation and acts like a tight flow restrictor on deflation, which was used in [13]. In FONS, all these components are either included in a standard library or they can be defined easily by building on top of this library.

Fluidic networks also have the unique property that they function differently depending on what fluid they are filled with. This is illustrated by three simulations in Fig. 5, where the peak in the PV curve of actuator 1 is slightly higher than the peak in actuator 2 and the valley of 1 is much lower than that of 2. This system also contains tight tubes as flow restrictors and an accumulator with a fixed internal volume of 100 mL. For a compressible fluid like air (blue), the accumulator is able to instantaneously supply fluid to a snapping actuator. This allows both actuators to snap independently from each other. For an incompressible fluid like water (orange), however, this is no longer possible and the volumes of both actuators are coupled with each other. Therefore, if a snapping event suddenly increases the volume in one actuator, the volume in the other actuator decreases by the same amount [5]. Finally, for a viscous fluid like ethylene glycol (green), the viscous damping in the flow restrictors eliminates these fast snaps. It also generates a pressure drop between the actuators that is bigger than the difference in peaks during inflation but smaller than the difference in valleys during deflation. Therefore, the sequence is modified for inflation but not for deflation. These three examples differ by a single line in a FONS simulation, highlighting the ability to quickly explore a multitude of actuation behaviors.

Finally, in Fig. 6 we show that the results produced by FONS are also quantitatively accurate. It compares the simulated and experimental behavior of a system with four bending actuators with a nonlinear PV characteristic connected through flow restrictors in response to a trapezoidal pressure signal generated by a pressure source (Fig. 6A). Details about the design of and the experimental study on this system are reported in [10]. For the components used in that study, we experimentally measure the PV characteristics of the individual actuators, the pressure-flow curve of a flow restrictor and the input signal generated by the pressure source (Fig. 6B). These characteristics are processed with a low-pass filter and interpolated in the `evaluate`-method

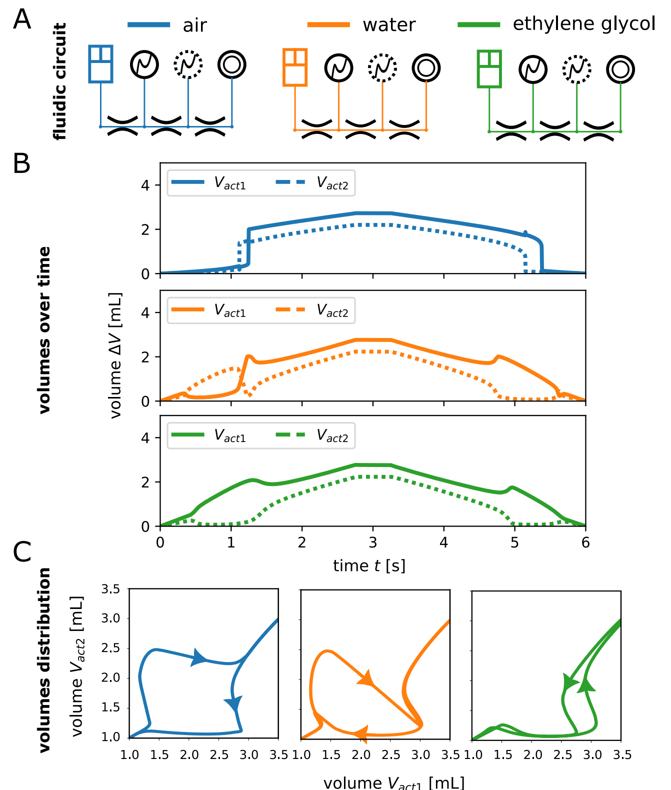

Fig. 5. FONS simulation results for the same system and components in case it is filled with air (blue), water (orange) or ethylene glycol (green)

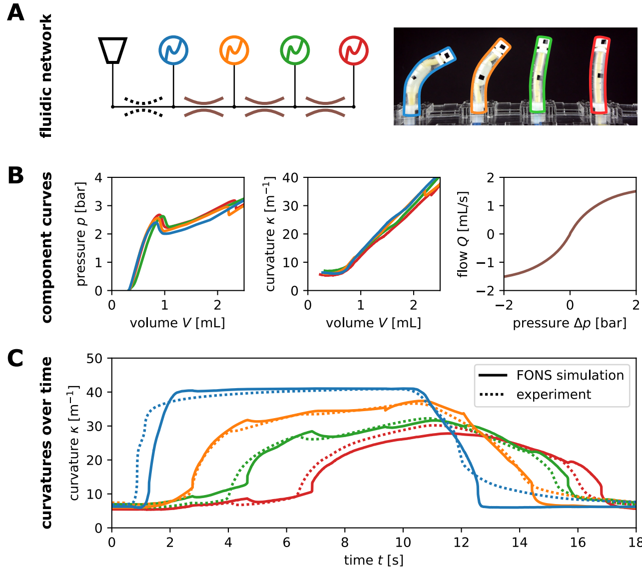

Fig. 6. (A) Architecture of a pressure-controlled network with four nonlinear bending actuators. (B) Experimentally measured characteristics of the actuators and restrictors. (C) Comparison between experimentally measured curvatures [10] and a FONS simulation for the same system.

of the corresponding components in FONS. Moreover, we measure the curvature of each actuator in function of its internal volume. After running FONS, we use these characteristics to map the change in volume of each actuator to a curvature that can be compared to the experimental results of [10]. Even though FONS does not model the inertia or visco-elasticity of the actuators, there is a good agreement between the simulation and the experiment (Fig. 6C).

IV. Conclusions

FONS is a python software library available at `https://pypi.org/project/fonsim/`. It provides a suite of tools to easily define lumped fluidic networks with nonlinear components and to accurately simulate their fluidic interactions. The FONS source code follows a transparent object-oriented framework with a library of standard components that can be extended by the user to incorporate new fluidic components. Moreover, FONS features a straightforward high-level interface with other programs. In combination with robust and fast solving, this allows for integration in ever more performant design algorithms. As such, we envision FONS to be not just a source of inspiration in designing novel fluidic systems with morphological control, but also a key tool for their inverse design.

References

[1] D. Rus and M. T. Tolley, "Design, fabrication and control of soft robots," *Nature*, vol. 521, no. 7553, pp. 467–475, may 2015.

[2] R. Pfeifer and J. Bongard, "How the Body Shapes the Way We Think," in *How Body Shapes W. We Think*. The MIT Press, 2006.

[3] C. A. Aubin, B. Gorissen *et al.*, "Towards enduring autonomous robots via embodied energy," *Nature*, vol. 602, no. 7897, 2022.

[4] B. Gorissen, D. Reynaerts, S. Konishi, K. Yoshida, J.-W. Kim, and M. De Volder, "Elastic Inflatable Actuators for Soft Robotic Applications," *Adv. Mater.*, p. 1604977, sep 2017.

[5] J. T. B. Overvelde, T. Kloek, J. J. a. D'haen, and K. Bertoldi, "Amplifying the response of soft actuators by harnessing snap-through instabilities." *PNAS*, vol. 112, no. 35, 2015.

[6] L. Hines, K. Petersen, and M. Sitti, "Inflated soft actuators with reversible stable deformations," *Adv. Mater.*, vol. 28, no. 19, 2016.

[7] E. Ben-Haim, L. Salem, Y. Or, and A. D. Gat, "Single-input control of multiple fluid-driven elastic actuators via interaction between bistability and viscosity," *Soft Robotics*, vol. 7, no. 2, 2020.

[8] D. Glozman, N. Hassidov, M. Senesh, and M. Shoham, "A self-propelled inflatable earthworm-like endoscope actuated by single supply line," *IEEE Trans. Biomed. Eng.*, vol. 57, no. 6, 2010.

[9] B. Gorissen, E. Milana, A. Baeyens, E. Broeders, J. Christiaens, K. Collin, D. Reynaerts, and M. De Volder, "Hardware Sequencing of Inflatable Nonlinear Actuators for Autonomous Soft Robots," *Adv. Mater.*, vol. 31, no. 3, pp. 1–7, 2019.

[10] E. Milana, B. Van Raemdonck, A. S. Casla, M. De Volder, D. Reynaerts, and B. Gorissen, "Morphological control of cilia-inspired asymmetric movements using nonlinear soft inflatable actuators," *Frontiers in Robotics and AI*, vol. 8, 2021.

[11] N. Vasios, A. J. Gross, S. Soifer, J. T. Overvelde, and K. Bertoldi, "Harnessing Viscous Flow to Simplify the Actuation of Fluidic Soft Robots," *Soft Robot.*, vol. 7, no. 1, 2019.

[12] L. Jin, A. E. Forte, and K. Bertoldi, "Mechanical valves for on-board flow control of inflatable robots," *Adv. Science*, vol. 8, no. 21, 2021.

[13] L. C. van Laake, J. de Vries, S. M. Kani, and J. T. Overvelde, "A fluidic relaxation oscillator for reprogrammable sequential actuation in soft robots," *Matter*, vol. 5, no. 9, pp. 2898–2917, 2022.

[14] A. Vladimirescu, *The SPICE book*. Wiley New York, 1994.

[15] P. Fritzson and V. Engelson, "Modelica—a unified object-oriented language for system modeling and simulation," in *European Conference on Object-Oriented Programming*. Springer, 1998, pp. 67–90.

[16] J. B. Dabney and T. L. Harman, *Mastering simulink*. Pearson/Prentice Hall Upper Saddle River, 2004, vol. 230.

[17] B. Mosadegh, P. Polygerinos, C. Keplinger, S. Wennstedt, R. F. Shepherd, U. Gupta, J. Shim, K. Bertoldi, C. J. Walsh, and G. M. Whitesides, "Pneumatic networks for soft robotics that actuate rapidly," *Adv. Funct. Mater.*, vol. 24, no. 15, pp. 2163–2170, 2014.

[18] B. Gorissen, D. Melancon, N. Vasios, M. Torbati, and K. Bertoldi, "Inflat. soft jumper inspired by shell snapping," *Sci.Robot*, vol. 5, 2020.

[19] A. A. Stanley, A. Amini, C. Glick, N. Usevitch, Y. Mengüç, and S. J. Keller, "Lumped-parameter response time models for pneumatic circuit dynamics," *J. Dyn. Syst. Meas. Contr.*, vol. 143, no. 5, 2021.

[20] S. E. Haaland, "Simple and explicit formulas for the friction factor in turbulent pipe flow," 1983.

[21] T. J. Jones, E. Jambon-Puillet, J. Marthelot, and P.-T. Brun, "Bubble casting soft robotics," *Nature*, vol. 599, no. 7884, 2021.

[22] S. Zhang, S. Winoto, and H. Low, "Performance simulations of tesla microfluidic valves," in *Int. Conf. on Integration and Commercialization of Micro and Nanosystems*, vol. 42657, 2007, pp. 15–19.

V. FONS example in less than 50 lines of code

```
1  #!/usr/bin/env python3
2  # git hash: 7f50c2c9b9f072169e986547cc1f79a5de746d4d
3  import fonsim as fons
4  p0 = fons.pressure_atmospheric
5
6  # === Create the components ===
7  class Actuator(fons.Component):
8      def __init__(self, label, fluid, pvcurve):
9          super().__init__(label)
10         p_ =  fons.Variable('pressure', 'across', label='p', initial_value=p0)
11         mf_ = fons.Variable('massflow', 'through', label='mf')
12         m_ =  fons.Variable('mass', 'local', label='mass', initial_value=1e-3)
13
14         self.set_terminals(fons.Terminal('a', [p_, mf_]))
15         self.set_states(m_)
16
17         def evaluate(t, p, mass):
18             return [p - pvcurve.fdf_volume(volume=mass/fluid.rho)[0]]
19         self.evaluate = self.auto(evaluate)
20
21         def update_state(dt, mass, mf):
22             return {'mass': mass + mf * dt}
23         self.update_state = self.auto_state(update_state)
24
25 # === Create the fluidic system ===
26 system = fons.System()
27 system += fons.PressureSource('source',
28                               pressure=lambda t: p0 + (.3e5 if t<.5 else 0))
29 system += fons.CircularTube(label='tube', fluid=fons.water,
30                             length=0.60, diameter=1e-3)
31 system += Actuator('actu', fluid=fons.water,
32                    pvcurve=fons.pvcurve.PVCurve('actPeakValleyHigh.csv'))
33
34 # Connect the components to each other
35 system.connect('tube', 'source')
36 system.connect('tube', 'actu')
37
38 # == Simulate the system ==
39 sim = fons.Simulation(system, duration=1.)
40 sim.run()
41
42 # Put results in local variables (e.g. for visualization later)
43 t = sim.times
44 p_actu = (system.get('actu').get('pressure') - p0) * 1e-5
```