Matlab Code for Morphological Computation for Soft Bodies

This is a short description on how to use the Matlab simulation code developed for the publications [1] [»] and [2] [»]. To understand the code and its purpose, you will have to have read those papers.

1 Idea of the Approach

The underlying idea of [1] and [2] was to develop theoretical models that are able to describe the computational power of generic physical (soft) bodies. It is based on the concept of morphological computation [4], which is based on the observation that biological systems have a certain morphology that seems to carry out computations that are beneficial for interaction with the environment and other agents. The theoretical models of Hauser et al. are based on a machine learning technique called *Reservoir Computing*, see [5]. In this case the physical body serves as a reservoir and, hence, as a computational resource.

The idea of the toolbox was to simulate generic models of real physical, compliant bodies. The results are networks of nonlinear spring-damper systems and masses. These networks are initialized with random parameters like the nonlinear properties of the springs (i.e., parameters of the nonlinear stiffness and damping functions described by polynomials of order three – see [1] and [2] for more details). The inputs to the systems are applied to randomly chosen input nodes (i.e., masses of the network). The feedback is applied in the same manner. The readout from the system is defined as the set of current lengths[1] of all springs.

[1]Note that the code also allows to use current positions of all masses as the readout.

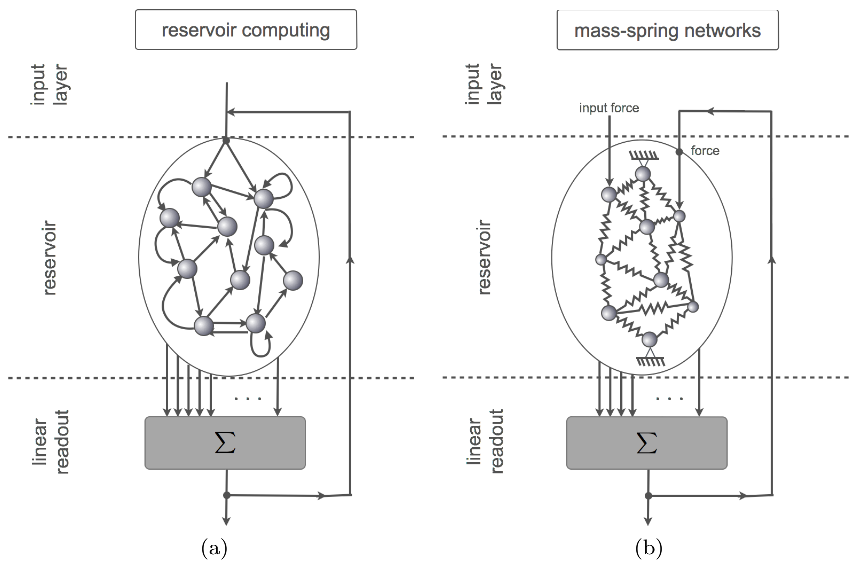

Figure 1: Schematic of (a) the generic Reservoir Computing setup and (b) the corresponding Morphological Computation setup used in [1] and [2].

2 Description of Files

Here follows a short description of the folders and files:

- folder **data_NARMA**:
  - **NARMA-L2.mat**: example learning data
  - **make_NARMA_data.m**: Matlab code to produce learning data based on a NARMA (nonlinear autoregressive–moving-average) system
- folder **data_quad**:
  - **quad_e=7.mat**: example learning data based on a "quadratic limit cycle" as describe in Figure 5e in [2]
  - **plot_one_period.m**: function to plot one period of this limit cycle
- folder **data_vanderPol**:
  - **make_vdPol_data.m:** Matlab code to produce learning data of a van der Pol system. This can be easily adapted to make your own learning data

  - **ode_van_der_Pol_sd.m:** Matlab function used to simulate a van der Pol system. You can adapt it easily to produce other non-linear limit cycles.
  - **plot_one_period_van_der_Pol.m:** function to plot the obtained data
  - **vanderPol.mat:** example learning data

- folder **data_Volterra**:
  - **make_Volterra_data.m:** Matlab code to produce data for the Volterra task (quadratic kernel) as describe in [1].
  - **gauss_kernel_2D.m:** code to define the used kernel (in this case a Gaussian kernel)
  - **volterra.mat:** example learning data

- folder **helping_functions**:
  - **e_distance.m**: calculates the euclidean distance between two points (to get current spring lengths)
  - **learn_linear_model.m**: learns a linear model with linear regression to emulate a given input/output data set (to evaluate the computational power of the morphology, see [1, 2])
  - **make_simulation_movie.m**: function to produce an .avi movie of the simulated network
  - **nth_point.m**: function used for **plot_one_period.m** (reduces the size of the figure file)
  - **plot_graph.m**: plots the networks with color coding (as in Fig. 7 of [1])
  - **rand_in_range.m**: draws random numbers from a given range
  - **rand_in_range_exp.m**: draws random numbers from a given range with an exponential distribution.
  - **rhs_mass_sys.m**: function used for the Runge-Kutta algorithm
  - **rk4ode2.m**: Runge-Kutta algorithm used for the physical simulation
  - **step_response.m**: produces a step response (or impulse response) of a given network

- **main folder**:
  - **init_ms_sys_data.m**: initializes a data structure with default values that defines the properties and parameter ranges for a network. The returned data structure can be manipulated and is fed to **init_ms_sys_net.m**.

- **init_ms_sys_net.m**: initializes a random network based on the properties described in a data structure
- **init_path.m**: this has to be adapted by the user. It adds the right paths to the Matlab path
- **ode_simple_ms_sys.m**: function needed for the physical simulation
- **simulate_ms_sys.m**: main physical simulation function
- **learning_Volterra.m**: demonstration of learning to emulate a Volterra operator as described in [1].
- **learning_NARMA.m**: demonstration of learning to emulate a NARMA system as described in [1].
- **learning_quad_generator.m**: demonstration of learning to produce autonomously a nonlinear limit cycle.

3 Simple Demonstration

Before you start you have to add the folder of the demo and its subfolders to the current Matlab path so that Matlab is able to find the functions. You have to adapt the local paths in the function **init_path.m** accordingly and call the function.

You can then start the demonstration by calling **learning_Volterra.m**, which

- Loads Volterra data (input-output data set)
- Sets appropriate values like ranges for the dynamical parameters, number of masses, etc.
- Produces a random network (based on these parameter ranges)
- Simulates it in open loop with the learning data
- Learns the optimal output weights with linear regression
- Simulates the network with the optimal weights and new testing input data
- Compares the system output to the target output and presents the results

Take a look at the code and the comments therein to understand the individual parts. You can run **learning_NARMA.m** and and **learning_quad_generator.m** to see how such a setup can also be used to learn to emulate a NARMA system and to produce robustly and autonomously a nonlinear limit cycle respectively.

4 What else can you do

Here are a couple of steps you could try out:

- Try and test different parameters in the data structure (see **init_ms_sys_data.m** for more information) and use **step_response.m** and **plot_graph.m** to see visually the differences in the responses of the networks and their topographies.

- Investigate different parameter ranges and how they change the performance (i.e., MSE) for the Volterra and NARMA task:

  - Size of the network ***data.num*** (i.e. number of masses)
  - Parameter ranges for the stiffness and damping functions ***data.k_lim*** and ***data.d_lim***
  - Change the size of the used learning data ***len***

- Investigate different parameter ranges and their influence on the performance (i.e., MSE) and the robustness of a learned nonlinear limit cycle (e.g., the quadratic limit cycle). When does it get unstable, i.e, the closed loop system drifts away.

  - Size of the network ***data.num*** (i.e. number of masses)
  - Parameter ranges for the stiffness and damping functions ***data.k_lim*** and ***data.d_lim***
  - Amplitude of the applied noise during learning

- Write a Matlab code to test the robustness of a learned limit cycle. For example, in [2] (see Fig. 6) we used following perturbations:

  - Hold one output to a fixed value for certain time (Fig 6a,b)
  - Apply strong forces to randomly chosen nodes for a certain time (Fig 6c,d)
  - Apply strong noise to the readout (i.e., add noise to the current spring lengths) (Fig 6e,f)
  - Come up with your own idea how to perturb the system. What could be a “natural” way of perturbation if you think of the networks as a real body (part) of a real soft robot?

- Compare performance (for the NARMA and Volterra task) to the case when no morphology (i.e., no body) is available. You can use the function **learn_linear_model.m** – compare Fig 9 in [1].

- make your own learning data and use it to train a morphological computation setup, for example:

  - Use a different kernel for the Volterra series (non-Gaussian, adding terms of higher order, etc.)
  - Define different NARMA systems (see e.g., Equ. 7 in [1])
  - Define another nonlinear limit cycle (e.g., van der Pol – see corresponding subfolder)
  - Design your own learning task that deals with multiple inputs

- Design a *multitasking* setup (as in Section 4.3 of [1])
- Think about what the properties of our mass-spring networks relate to properties in a real-world soft robot.

If you are looking for more inspiration and what is possible, we refer you to our publication "Morphological Computation – The Physical Body as a Computational Resource" [3].

If you have any troubles or if you find a bug, please, let me know.

hhauser@ifi.uzh.ch

References

[1] Helmut Hauser, Auke Ijspeert, Rudolf Füchslin, Rolf Pfeifer, and Wolfgang Maass. Towards a theoretical foundation for morphological computation with compliant bodies. *Biological Cybernetics*, 105:355–370, January 2011. Issue 5.

[2] Helmut Hauser, Auke Ijspeert, Rudolf Füchslin, Rolf Pfeifer, and Wolfgang Maass. The role of feedback in morphological computation with compliant bodies. *Biological Cybernetics*, 106(10):1–19, 2012.

[3] Helmut Hauser, Kohei Nakajima, and Rudolf M. Füchslin. Morphological computation – the body as a computational resource. In Helmut Hauser, Rudolf M. Füchslin, and Rolf Pfeifer, editors, *E-book on Opinions and Outlooks on Morphological Computation*, chapter 20, pages 226–244. 2014.

[4] Rolf Pfeifer, Max Lungarella, and Fumiya Iida. Self-organization, embodiment, and biologically inspired robotics. *Science*, 318:1088–1093, November 2007.

[5] Benjamin Schrauwen, David Verstraeten, and Jan Van Campenhout. An overview of reservoir computing: theory, applications and implementations. In *Proceedings of the 15th European Symposium on Artificial Neural Networks*, pages 471–482, 2007.