model-learning

Francesco Regazzoni

Mar 08, 2022

CONTENTS

`model-learning` is a library for model learning and model reduction, written in `MATLAB`. The main methods and algorithms implemented in the library have been introduced in the papers [1,2], wherein they are derived and documented.

CHAPTER

ONE

CONTENTS

1.1 Installation

To install the library, follow these steps.

- Run `addpaths.m`.
- Enjoy.

Note: The data generated by the library are by default stored into a folder `data`, contained in the main folder of the library. If you want to modify this location, just copy in the main folder the file `options_example.ini` into `options.ini` (not git tracked) and customize the field `datapath`.

Note: A collection of pre-trained ANNs is available in the repo model-learning_data. To use them, clone or download the repo and make the `datapath` field point to the path of the repo (or alternatively copy the content of the repo into the `datapath` folder).

1.2 Python wrapper

The library provides a Python interface to deploy the trained model. The module can be loaded as:

```
import pyModelLearning
```

To install it, you can choose among the following options.

1.2.1 Option 1: Install the library by setuptools

From the main folder of this repository run:

```
$ pip install . --use-feature=in-tree-build
```

1.2.2 Option 2: Add the library path to Python paths

Linux / macOS

Add to your `.bashrc`:

```
export PYTHONPATH="${PYTHONPATH}:/path/to/model-learning"
```

Windows

Add the path of `model-learning` to the `PYTHONPATH` environment variable. Alternatively, you can write the path of `model-learning` inside a `pth` file, as described here.

1.2.3 Option 3: Add the library path within your python script

Put the following lines at the beginning of your Python script:

```
import sys
sys.path.append("/path/to/model-learning")
import pyModelLearning
```

1.3 Quickstart

The repository contains several examples of usage of the library, some of which are documented and described in [1]. Each example is contained into a dedicated folder in `\examples` (e.g. `\examples\wave_oned`). The standard structure for an *example* envisages:

- The *problem* specifications, through an `.ini` file (e.g. `\examples\wave_oned\wave1D.ini`).
- The high-fidelity *model* definition, through a `.m` file (e.g. `\examples\wave_oned\wave1D_getmodel.ini`).
- A script that generates training and testing *datasets* (e.g. `\examples\wave_oned\wave1D_generate_tests.ini`) and stores them into the `datapath` folder.
- One or more configuration files to train a reduced model, with the format `opt_*.ini` (e.g. `\examples\wave_oned\opt_wave1D.ini`). To use them, execute the command:

  ```
  model_learn('opt_wave1D.ini')
  ```

- One or more scripts to analyze and postprocess the results obtained with the trained models and the HF model.

To create a new *example*, create a new folder in the folder `\examples`.

This documentation also contains a *Tutorial* to get acquainted with the library.

1.4 Notes about the conventions

1.4.1 Function arguments

Functions typically accept a list of mandatory arguments, briefly documented in the function help. Optional arguments are typically passed by a structure array `opt`. In the first lines of the function, default values are typically assigned to undefined optional arguments. These lines thus provide a list of the possible optional arguments.

1.4.2 Variable names

In the code the names of the variables concerning the models refelct the convention used in [1]. Specifically, models are denoted as follows:

$$\frac{d\mathbf{x}}{dt} = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t), \boldsymbol{\alpha})$$
$$\mathbf{y}(t) = \mathbf{g}(\mathbf{x}(t))$$

where t is time, $\mathbf{x}(t)$ is the internal state, $\mathbf{u}(t)$ (if present) is the time-dependent input, $\boldsymbol{\alpha}$ (if present) is a constant parameter, $\mathbf{y}(t)$ is the output.

1.5 Tutorial

Position the `Matlab` current directory into the folder `\examples\tutorial`. The tutorial consists into two examples:

1.5.1 Learning models without inter-individual variability

Consider the nonlinear transmission line circuit example of Sec. 4.2 of [1]. The learning pipeline consists in the steps listed below. This tutorial allows to follow the pipeline step by step, but the full list of commands are collected in the unique file `\examples\tutorial\NTL1_tutorial.m`.

Definig the problem

A *problem* mainly consists in the definition of the input-output variables (how many they are, which are the bounds for each variable). Notice that a problem does not contain any information about the map linking inputs to outputs: such map is rather defined by a *model*, which is a different concept, treated in the next section. Indeed, for a given problem (in this case, the problem is linking the input current of the circuit to the output voltage) one may have several models.

Problems are defined through `.ini` files. The full list of tags to define a problem is available in `\examples\problem_example.ini`.

The problem of the current example is defined in `\examples\tutorial\NTL1.ini`. A problem may contain a handle to generate the high-fidelity model associated to the problem, as we will see in the next section.

Definig the high-fidelity model

A *model* can be regarded as a map from a time-dependent input to a time-dependent output, according to the definition of Sec. 2.1 of [1]. Each model is associated to a problem. Notice that a model comprises both the mathematical model and its discretization: it corresponds thus to the concept of **numerical model**.

Models are represented by MATLAB structs, that are typically generated by ad-hoc functions, for better versatility. In this example, the high-fidelity model is generated by the function `\examples\tutorial\NTL1_getmodel.m`. The model struct should contain the following fields:

Basic fields

- `model.nX`: number of internal states
- `model.x0`: initial state
- `model.dt`: integration time step

Model dynamics

The model dynamics can be defined in different ways (field `model.advance_type`), depending on the features of the model (linear, nonlinear, ecc.). In this example, the model is defined through its right-hand side and it is solved by Explicit Euler scheme (`model.advance_type = 'nonlinear_explicit'`). The right-hand side is defined in the field `model.f`, which implements Eq. (32) of [1].

Model output

Also the output of the model can be defined in different ways (field `model.output_type`). In this case, it is of type `'insidestate'`, which means that the output is given by the first `model.nY` internal states (this corresponds to the input-inside-the-state approach of [1]).

Note: In alternative to the implementation contained in `\examples\tutorial\NTL1_getmodel.m`, models can be defined as black-boxes (rather than defining the number of variables, initial state, dynamics, etc.). This is the only possibility when the model is defined by an external software. In this case, the user only needs to write a wrapper for the external call, and bind inputs and outputs. An example is contained in `\examples\tutorial\NTL1_getmodel_blackbox.m`. Notice that all you need to specify is the flag `model.blackbox = 1` and the handler function `output = model.forward_function(test, options)` (see the examples for more details).

Once the problem and the model have been defined, we can load them with the following commands:

```
problem = problem_get('tutorial','NTL1.ini');
HFmod = problem.get_model(problem);
```

To employ the model to perform simulations, first define a *test struct*:

```
test_solve.tt = [0 10];
test_solve.uu = @(t) .5*(1+cos(2*pi*t/10));
```

Then, solve the model as follows:

```
figure();
output = model_solve(test_solve,HFmod,struct('do_plot',1));
```

The struct `output` contains the *output test struct* of the simulation.

Generating training datasets

With the following lines, we use the high-fidelity model `HFmod` to generate two training *datasets*. With the option `do_save = 1` the datasets are stored into an automatically generated path inside the data folder defined in `options.ini` (see *Installation*), according the example name and problem name.

```
rng('default') % for reproducibility

opt_gen.do_plot = 1;
opt_gen.do_save = 1;
opt_gen.optRandomU.time_scale = .02;

opt_gen.outFile = 'samples_rnd.mat';
dataset_generate_random(HFmod,100,opt_gen);

opt_gen.constant = 1;
opt_gen.wait_init = 1;
opt_gen.wait_init_time_wait = .2;
opt_gen.wait_init_time_raise = 0;
opt_gen.outFile = 'samples_step.mat';
dataset_generate_random(HFmod,50,opt_gen);
```

The first *dataset* contains 100 *tests* associated to random inputs. The second one contains 50 *tests* associated to step inputs.

It is possible to extract subsets from datasets and combine them. With the following code, for instance, we create a dataset by combining the first three *tests* of the step responses generated before with the first 8 random responses.:

```
dataset_def.problem = problem;
dataset_def.type = 'file';
dataset_def.source = 'samples_step.mat;1:3|samples_rnd.mat;1:8';
train_dataset = dataset_get(dataset_def);
```

To plot the dataset, type:

```
dataset_plot(train_dataset,problem)
```

Training the ANN

Training specifications are defined into an option file. The full list of available options, with relative documentation, can be found in `\mor_ANN_blackbox\opt_example.ini`.

The option file for the current example is contained into `\examples\tutorial\NTL1_opt.ini`. We now comment the main fields:

- `Problem\dataset_source_train` and `Problem\dataset_source_tests` contain the specifications for the training and the test datasets, with the same sitax used before
- `Model\N`: number of states in the learned model
- `ANN\layF`: number of neurons in the hidden layers of the ANN

To train the network, run the following command, which will stop afer 100 training epochs:

```
model_learn('NTL1_opt.ini')
```

The trained model is stored in an automatically generated path. A name is automatically assigned to the trained model: it appears at the beginning of training but it is also copied to the clipboard just after it appears in the MATLAB console. Paste it somewhere to be able to load the trained model later. The name is generated according to the main settings of learning plus a time stamp (e.g. 'test_int_N2_hlayF5_dof32_2019-02-28_11-07-14')

Using the trained model

To load the learned model, run (after replacing the learned model name with the one you stored before):

```
ANNmod = read_model_fromfile(problem,'test_int_N2_hlayF5_dof32_2019-02-28_11-07-14');
```

The struct ANNmod is a *model struct*, as much as HFmod: it can indeed be employed to perform the same actions as the high-fidelity model:

```
figure();
output = model_solve(test_solve,ANNmod,struct('do_plot',1));
```

Moreover, being a spacial type of model (a model whose right-hand side is defined by an ANN), it has additional features. For instance, the ANN can be visualized, by running `ANNmod.visualize()`.

To evaluate the level of approximation of the learned model, first load a test *dataset*:

```
dataset_def.problem = problem;
dataset_def.type = 'file';
dataset_def.source = 'samples_step.mat;11:50|samples_rnd.mat;21:100';
test_dataset = dataset_get(dataset_def);
```

Then, evaluate the error with the high-fidelity model (it must be zero):

```
model_compute_error(HFmod, test_dataset);
```

and with the learned model:

```
model_compute_error(ANNmod, test_dataset);
```

1.5.2 Learning models with inter-individual variability

In the example considered in the previous section, all the indivuals in the training set shared the same dynamics and the same initial state. We now consider the case of **inter-individuals variability**, that is when the individuals may differ for either the dynamics (more precisely, we suppose that the dynamics follows a common law, dependent on parameters, which may be be different for different individuals) or the intial state, or both.

We consider in particular the following exponential model:

$$\begin{aligned} \frac{dx}{dt} &= \alpha x(t) \\ x(0) &= x_0 \end{aligned}$$

where the output is x itself and the initial state x_0 and the parameter α may be different for different individuals.

Notice that such object is not a **model** according to the definition of Sec. 2.1 of [1], since it does not uniquely defines a map from the space of inputs (which is empty in this case) to the space of outputs. Thus, we refer to it as a **metamodel** since it defines rather a family of models.

Definig the problem

The problem is defined in `\examples\tutorial\expmod.ini`. With respect to the previous example, we have two new fields:

- `fixed_x0 = 0`: the individuals do not share the same initial state
- `samples_variability = 1`: the law of evolution may depend on parameters, different for each individual.

Definig the high-fidelity model

The model (or better, the metamodel) is defined in `\examples\tutorial\expmod_getmodel.m`. With respect to the previous example, we notice the following differences:

- Also the number of parameters (`nA`) and their bounds are defined
- The right-hand side depends on the parameter, thus it is assigned into `f_alpha`
- The derivatives of the right-hand side w.r.t. the state and the parameters are provided. This is not mandatory, unless the model will be later used for data assimilation purposes.

We can now load the model:

```
problem = problem_get('tutorial','expmod.ini');
HFmod = problem.get_model(problem);
```

Notice that we cannot test directly the model, since it is just a metamodel. First, we have to **particularize** it, i.e. assigning initial condtion (e.g. `x_0 = 1.1`) and parameter (e.g. `a = 0.5`):

```
HFmod_particularized = metamodel_particularize(HFmod, 1.1, .5);
```

Now, we can solve the model as before:

```
figure()
model_solve(struct('tt',[0 1]), HFmod_particularized, struct('do_plot',1))
```

Otherwise, with the following command, we assign a random initial state and a random parameter, and show the result:

```
model_show_example(HFmod);
```

Generating training datasets

As before, with the following commands we generate a training dataset. In this case, initial state and parameters will be randomly generated.:

```
rng('default')

opt_gen.do_plot = 1;
opt_gen.do_save = 1;
opt_gen.T = 1;
opt_gen.outFile = 'samples.mat';
dataset_generate_random(HFmod,100,opt_gen);
```

Training the ANN

As before, we define the learning specifications in the file `\examples\tutorial\expmod_opt.ini`. The main differences are:

- We must specify the number of parameters in the learned model, by `Model\N_alpha`
- We add some noise, by `Problem\noise_y`

To train the network, run:

```
model_learn('expmod_opt.ini')
```

Using the trained model

As before, the learned model can be loaded by:

```
ANNmod = read_model_fromfile(problem,'test_int_N1_hlayF3_dof13_2019-02-28_17-51-59');
```

Notice that the learned model is also a metamodel! To use it, it should first be particularized. Otherwise, one can show a possible evolution of an individual with the command:

```
model_show_example(ANNmod);
```

When the parameter is just one, the original and learned parameters can be compared by:

```
model_alpha_plot(ANNmod);
```

Using data assimilation on the learned model

To test how Extended Kalman Filter performs on a given metamodel and for a given level of noise, the command `da_test` can be used. It can be used both with the high-fidelity and learned model, since they are metamodels:

```
noise = 1e-3;
da_test(HFmod, noise);
da_test(ANNmod, noise);
```

The following command instead, performs the following steps:

- It randomly generates an indidual and simulates its evolution by means of the high-fidelity model;
- It adds the prescribed amount of noise;
- For the first half of the time span of the simulation, it performs data assimilation through the learned model, to estimate the parameter of the individual and its state;
- It uses those two estimated values to predict the future evolution of the individual and computes the error.

```
opt_ep.mod_HF = HFmod;
opt_ep.obs_err = 1e-3;
opt_ep.pause_each_test = 1;
opt_ep.do_plot = 1;
da_estimate_predict(problem,ANNmod,opt_ep);
```

By pressing ENTER, the process is reapeated for other individuals and the mean error is computed.

With the following command instead, the same process is repeated for different noise levels (without stopping at each individual), and the convergence plot is shown.:

```
opt_ep.mod_HF = HFmod;
opt_ep.obs_err = 10.^-(1:8);
opt_ep.pause_each_test = 0;
opt_ep.do_plot = 0;
opt_ep.n_tests = 20;
da_estimate_predict(problem,ANNmod,opt_ep);
```

1.6 Documentation

1.6.1 The folder `example`

What is an example?

In this library, an **example** is simply a logical container that groups the files corresponding to a apecific application of the library. It corresponds to a folder contained in `/examples`. For instance, each of the test cases considered in [1] has its own example folder.

How to create a new example?

If you want to develop your own application with the library `model-learning`, you simply have to create a folder inside `/examples`.

Note: Inside `/examples`, folders starting with the prefix `app_` (the standard prefix for custom applications) are automatically gitignored.

1.6.2 The struct `problem`

What is a problem?

In this library, a **problem** corresponds to some physical, social, economical **time-dependent phenomenon** that we aim at describing in mathematical terms. It mainly consists in the definition of the input-output variables (how many they are, which are the bounds for each variable). Notice that a problem does not contain any information about the map linking inputs to outputs: such map is rather defined by a *model*, which is a different concept. Indeed, for a given problem one may have several models.

Each problem must be contained in an *example*. Notice that, however, every *example* can contain more that one problem.

How is a problem represented?

A problem is represented as a MATLAB struct.

To easiest (and safest) way of generating a problem struct is to define its features through an `.ini` file. The full list of tags to define a problem is available in `\examples\problem_example.ini`. Then, the problem `struct` can be generated by the command:

```
problem = problem_get('EXAMPLE_NAME','PROBLEM_DEFINITION.ini');
```

1.6.3 The struct `model`

What is a model?

A *model* can be regarded as a map from a time-dependent input to a time-dependent output, according to the definition of Sec. 2.1 of [1]. Each model is associated to a *problem*. Notice that a model comprises both the mathematical model and its discretization: it corresponds thus to the concept of **numerical model**.

How is a model represented?

A model is represented as a MATLAB struct. The struct must contain a field `model.problem`, containing the associated *problem struct*.

Models can be of two types: black-box models or explicitly-defined models.

Black-box models

Black-box models are characterized by the field `model.blackbox = 1`. The input-output map is defined through a function handle `model.forward_function`, with the following signature:

```
output = forward_function(test, options)
```

This function receives an input *test struct* and returns an output *test struct*. An example of definition of such function is contained in `\examples\tutorial\NTL1_getmodel_blackbox.m`.

Black-box models are useful when one wants to build a wrapper to an external software.

Explicitly-defined models

Explicitly-defined models contain the explicit definition of the properties of the model. In this case, the model struct must contain the following fields:

Basic fields

- `model.nX`: number of internal states
- `model.x0`: initial state
- `model.dt`: integration time step

Model dynamics

The model dynamics can be defined in different ways, according to the field `model.advance_type`, that takes the following values:

- `'solveonestep'`: the most generic case. It requires a function `x = model.solveonestep(x,u,dt)` that advances the state of one time step.
- `'solveonestep_linear'`: like `'solveonestep'`, but this entails the solution of a linear system $Kx = r$. It requires a function `[K,r] = model.solveonestep(x,u,dt)`.
- `'nonlinear_explicit'`: explicit Euler advance, with generic right-hand side, defined by the function `rhs = model.f(x,u)`.
- `'linear_CN_uaffine'`: generic linear model, with Crank-Nicolson advance, with affine dependence on $\mathbf{u}(t)$.
- `'linear_advance'`: generic linear model.
- `'linear_advance_timeexplicit'`: generic linear model, with explicit dependence on Δt.
- `'linear_advance_uaffine'`: generic linear model, with affine dependence on $\mathbf{u}(t)$.
- `'linear_advance_timeexplicit_uaffine'`: generic linear model, with explicit dependence on Δt and with affine dependence on $\mathbf{u}(t)$.

For the definition of the fields required according to the field `model.advance_type`, see the function `core/model_solve.m`

Model output

Also the output of the model can be defined in different ways, according to the field `model.output_type`, that takes the following values:

- `'insidestate'`: the output is given by the first `nY` internal states (i.e. `y = x(1:nY)`).
- `'nonlinear'`: the output is a non-linear function of the state: `y = model.g(x)`.
- `'linear'`: the output is a non-linear function of the state: `y = model.G * x + model.g0`.

Note: The first case corresponds to the *input-inside-the-state approach* of [1], while the other two correspond to the *input-outside-the-state approach*.

How to create or load a model struct?

The best-practice is that of creating models by means of ad-hoc functions. Examples are contained in `\examples\tutorial\NTL1_getmodel_blackbox.m` (black-box case) and `\examples\tutorial\NTL1_getmodel.m` (explicitly-defined case).

Such model constructors, that receive in input the *problem struct*, can be used in the `.ini` file defining a problem to define the high-fidelity model handler. In this case, the high-fidelity model associated with a problem can be obtained, for example, with the folowing commands:

```
problem = problem_get('tutorial','NTL1.ini');
HFmod = problem.get_model(problem);
```

Also reduced models based on ANNs are **models**, in the sense of the definition of this page. To load a trained model, you can use the following function:

```
ANNmod = read_model_fromfile(problem,'FOLDER_OF_TRAINED_MODEL');
```

What can I do with a model struct?

Models structs can be used to perform simulations. With the following command, for instance, we employ the model defined in the struct `model` to obtain the output *test struct* `test_output` associated with the input *test struct* `test_input` and we plot the solution:

```
figure();
test_output = model_solve(test_input, model, struct('do_plot',1));
```

1.6.4 The struct `test`

What is a test?

A **test** represents a time-dependent input $\mathbf{u}(t)$ in an interval $[0, T]$ and, optionally, the corresponding output $\mathbf{y}(t)$ and state $\mathbf{x}(t)$.

How is a test represented?

A test is represented as a MATLAB struct. The time is defined in the field `test.tt`, while the input, output and state are defined in the fields `test.uu`, `test.yy`, `test.xx`, respectively.

A test can be defined analytically, as in the following example:

```
test.tt = [0, 10];
test.uu = @(t) sin(t);
```

Otherwise, the test can be defined by a set of discrete values, as in the following example:

```
test.tt = linspace(1,10,200);
test.uu = sin(test.tt);
```

In case of vector-valued input (i.e. `nU > 1` in the *problem* definition), each entries of the input corresponds to a row of `test.uu` (the same convention if employed also for `test.yy` and `test.xx`), that is:

```
test.uu = @(t) [sin(t); cos(t)];
```

or

```
test.uu = [sin(test.tt); cos(test.tt)];
```

In case the value of `test.uu` for a time step not contained in `test.tt` is needed, a linear interpolation is performed.

Note: Both the *input* and the *output* of a simulation are represented as test structs. For example, with the following line one can obtain the solution corresponding the input $\mathbf{u}(t) = \sin(t)$ in the interval $t \in [0, 10]$ for the model represented by the *struct* `model`:

```
test_input.tt = linspace(1,10,200);
test_input.uu = sin(test.tt);
test_output = model_solve(test_input, model);
```

In this example, both `test_input` and `test_output` are test structs. However, while the former contains only the fields `test_input.tt` and `test_input.uu`, the latter contains the fields `test_output.tt`, `test_output.uu` and `test_output.yy`. Additionally, by specifying the option `save_x` as follows:

```
test_output = model_solve(test_input, model, struct('save_x', 1));
```

the struct `test_output` also contains the field `test_output.xx`.

1.6.5 The struct `dataset`

What is a dataset?

A **dataset** is a collection of *tests*.

How is a dataset represented?

A dataset is represented as a MATLAB cell array. Each cell is a *test struct*. Notice that the tests of the same dataset can be defined on different time intervals $[0, T]$.

How to generate a dataset?

A dataset can be manually generated. For instance, the following code creates a dataset with three tests:

```
% first test
dataset{1}.tt = linspace(0,10,250);
dataset{1}.uu = sin(dataset{1}.tt);
dataset{1}.yy = cos(dataset{1}.tt);

% second test
dataset{2}.tt = linspace(0,20,250);
dataset{2}.uu = exp(dataset{2}.tt);
dataset{2}.yy = 2 + dataset{2}.tt;

% third test
dataset{3}.tt = linspace(0,10,250);
dataset{3}.uu = 0 * dataset{3}.tt + pi;
dataset{3}.yy = tanh(dataset{3}.tt);
```

The following example, instead, creates a dataset with three tests, each one being the solution of the model defined in the *struct* `model`:

```
test.tt = [0, 10];

% first test
test.uu = @(t) sin(t);
dataset{1} = model_solve(test, model);

% second test
test.uu = @(t) cos(t);
dataset{2} = model_solve(test, model);
```

(continues on next page)

(continued from previous page)

```
% third test
test.uu = @(t) sin(t) + cos(t);
dataset{3} = model_solve(test, model);
```

Equivalentely, you can first generate a dataset containing only the inputs, and then obtain the outputs associated with the model defined in the *struct* `model` with a single command, thanks to the function `dataset_generate`. The following code provides the same results as the previous one:

```
dataset_input{1}.tt = [0, 10];
dataset_input{1}.uu = @(t) sin(t);

dataset_input{2}.tt = [0, 10];
dataset_input{2}.uu = @(t) cos(t);

dataset_input{3}.tt = [0, 10];
dataset_input{3}.uu = @(t) sin(t) + cos(t);

dataset = dataset_generate(model, dataset_input);
```

To generate a dataset with the goal of training an ANN-based model, it is useful to employ random inputs. This operation can be easily performed with the following command:

```
dataset = dataset_generate_random(model, 100)
```

that generates a dataset with 100 *tests*, where the inputs $\mathbf{u}_j(t)$, for $j = 1, \dots, 100$ are generated by an algorithm of random time-series generation (see `/tools/get_random_time_course.m`).

With the following command, instead, we generate a dataset with 20 tests associated with random constant inputs (i.e. $\mathbf{u}_j(t) \equiv \overline{\mathbf{u}}_j$ for $j = 1, \dots, 100$), where the values of $\overline{\mathbf{u}}_j$ are obtained by Monte Carlo sampling of the input space defined in the *problem struct*:

```
dataset = dataset_generate_random(model, 20, struct('constant', 1));
```

By specifying the option `lhs = 1`, the values of $\overline{\mathbf{u}}_j$ are generated by latin hypercube sampling:

```
dataset = dataset_generate_random(model, 20, struct('constant', 1, 'lhs', 1));
```

How to save and load a dataset?

Sometimes it is useful to give a name to a dataset and to save it, so that it can be later reused. This can be done with the following function:

```
dataset_save(problem, dataset, 'my_dataset.mat')
```

When the dataset is generated through the functions `dataset_generate` or `dataset_generate_random`, it can be directly stored by passing the options `do_save = 1` and `outFile = 'FILENAME.mat'`. For instance, with the following code a dataset with 100 random *tests* is stored:

```
opt_gen.do_save = 1;
opt_gen.outFile = 'samples_rnd.mat';
dataset_generate_random(model, 100, opt_gen);
```

Datasets are stored in an automatically generated path inside the data folder defined in `options.ini` (see *Installation*), under the name of `'samples_rnd.mat'`. Notice that each *problem* has its own path (that can be found in `problem.dir_data`): this entails that the same dataset name can be used for different problems without any conflict. On the other hand, if a dataset with the same name has been already defined for the same problem, it is overwritten by the new one.

The following code loads a previously saved dataset:

```
dataset_def.problem = problem;
dataset_def.type = 'file';
dataset_def.source = 'samples_rnd.mat';
train_dataset = dataset_get(dataset_def);
```

It is possible to load a subset of a dataset with the following sintax, that loads only the tests number 2, 3, 5, 6, 7 and 8:

```
dataset_def.source = 'samples_rnd.mat;[2,3,5:8]';
```

It is also possible to combine datasets in a single dataset:

```
dataset_def.source = 'samples_step.mat;[2,3,5:8]|samples_rnd.mat;1:8';
```

How to plot a dataset?

To plot the dataset, type:

```
dataset_plot(train_dataset, problem)
```

CHAPTER

TWO

REFERENCES

[1] F. Regazzoni, L. Dede', A. Quarteroni Machine learning for fast and reliable solution of time-dependent differential equations, *Journal of Computational Physics* (2019)

[2] F. Regazzoni, D. Chapelle, P. Moireau Combining Data Assimilation and Machine Learning to build data-driven models for unknown long time dynamics - Applications in cardiovascular modeling, *International Journal for Numerical Methods in Biomedical Engineering* (2021)

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX