Efficient determination of zero-crossings in noisy real-life time series

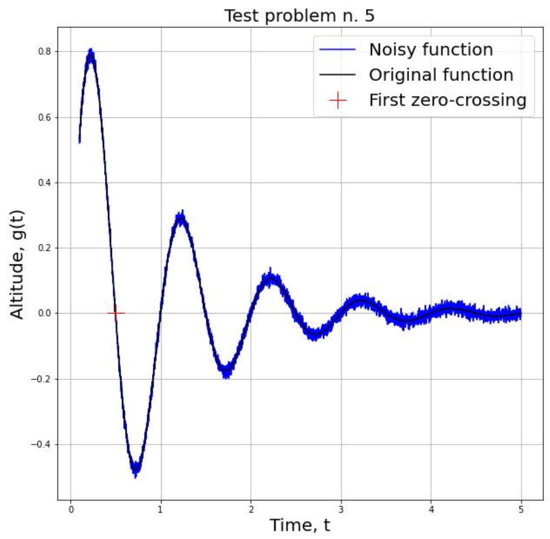
Marat S. Mukhametzhanov

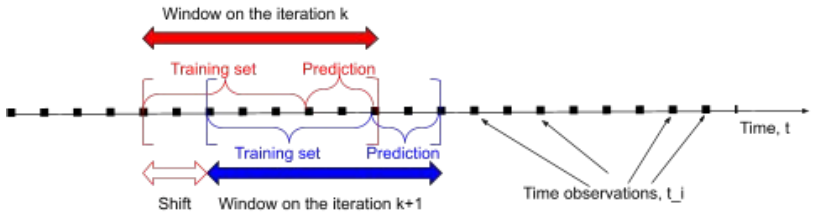
Problem statement

The first zero-crossing of *g(t)* is determined, $t \in [t_0, T],\ g(t_0) > 0.$

The main process:

1. Get the first N observations (N=1000 was used): $(t_0, g(t_0)), (t_1, g(t_1)), \ldots, (t_{N-1}, g(t_{N-1})).$
   The system should be "stable" at the initial time: $g(t_0) > 0$
2. Predict M values of the time series (M is adaptively estimated during the search) using the last N observations: $(t_N, \hat{g}(t_N)), (t_{N+1}, \hat{g}(t_{N+1})), \ldots, (t_{N+M-1}, \hat{g}(t_{N+M-1})).$
3. Check the zero-crossings among the predicted observations.
4. Get new data: $(t_N, g(t_N)), (t_{N+1}, g(t_{N+1})), \ldots, (t_{N+M-1}, g(t_{N+M-1})).$
5. Repeat the process taking the last N observations to train the models

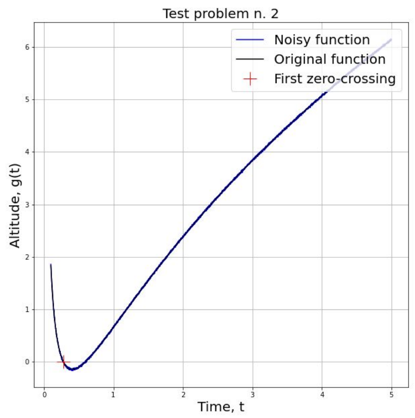


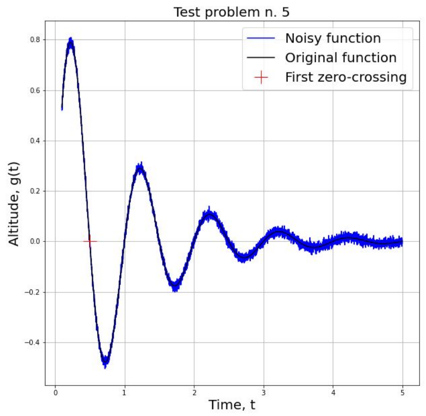


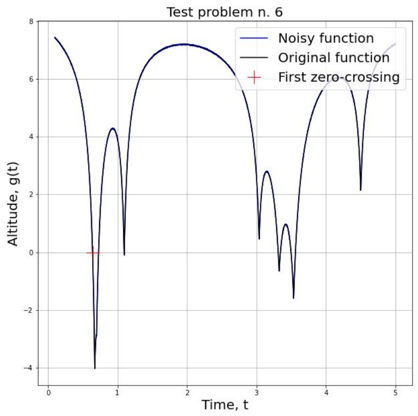
Used Software

- All methods have been implemented in Python 3.7 using Jupyter notebooks in Google Colaboratory.
- *Numpy*, *Pyspark*, *Scikit-learn* and *Tensorflow* with *Keras* have been used mainly. In detail for each step:
    - Initial data exploration: *numpy, matplotlib.pyplot, random, math*.
    - ETL: *pyspark* (with *SparkConf, SparkContext, SparkSession, ml.linalg.Vectors*) is added.
    - Feature engineering: *pyspark's ml.feature.PolynomialExpansion* and *ml.feature.MinMaxScaler, sklearn.preprocessing*'s *PolynomialFeatures* and *MinMaxScaler* have been added.
    - Model definition: *pyspark.ml.regression.LinearRegression, sklearn.linear_model.LinearRegression, keras.models.Sequential* and *keras.models.Dense* have been added.
    - Model training: *sklearn.metrics.r2_score* and *keras.callbacks.EarlyStopping* have been added.
    - Model evaluation and deployment: no additional libraries are required (*os* and *warnings* can be added optionally)

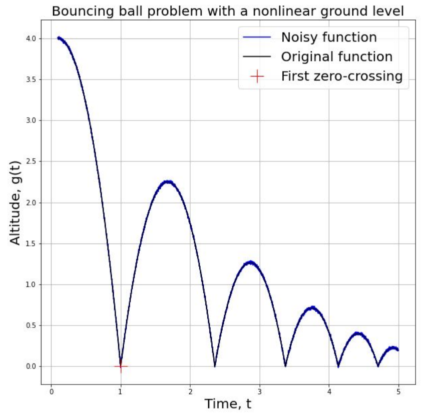
Data & Exploration: test functions and their quality

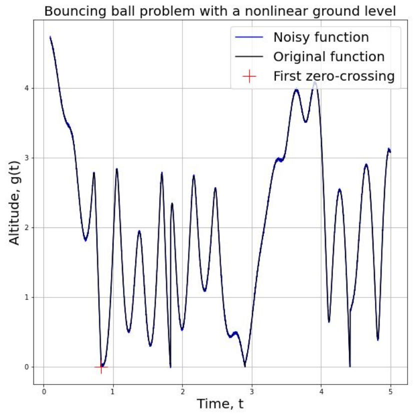






- The first 10 test problems from Casado, Garcia, Sergeyev (2003) have been used for evaluating the models (examples of functions number 2, 5 and 6 are presented).





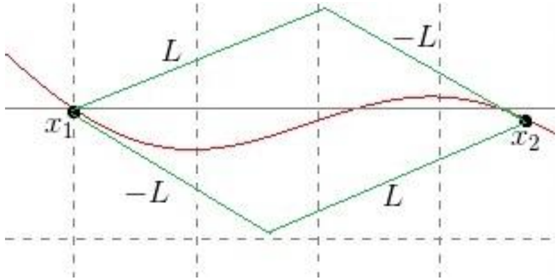
- Bouncing ball hybrid system with two different case studies have been also implemented: with a constant ground and difficult nonlinear ground level.

- A random gaussian noise has been added to all function evaluations in order to complicate the problems.

Data pre-processing: Lipschitz condition

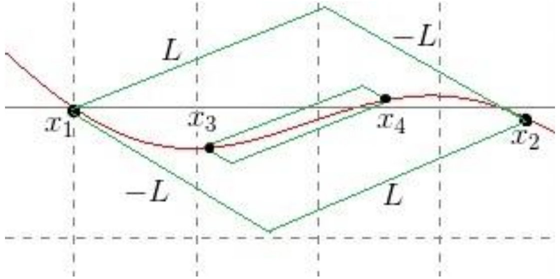
The objective function g(t) is supposed to be Lipschitz-continuous over the search interval:

$$|g(t^I) - g(t^{II})| \leq L \cdot |t^I - t^{II}|, t^I, t^{II} \in [t_0, T], 0 < L < \infty,$$



The meaning of the Lipschitz condition:

- For a physical system:
  the change of the energy in the system is always limited.
- From the numerical point of view, measure of the conditioning:
  - High Lipschitz constant <-> ill-conditioned system
  - Small Lipschitz constant <-> well-conditioned system.
- LC can be used to measure how large should be the prediction set:
  - Large prediction intervals <-> low accuracy, but high computational efficiency
  - Small prediction intervals <-> high accuracy, but low computational efficiency



Data pre-processing: Lipschitz constant estimation

- The objective function g(t) is supposed to be Lipschitz-continuous over the search interval:

$$|g(t^I) - g(t^{II})| \leq L \cdot |t^I - t^{II}|, t^I, t^{II} \in [t_0, T], 0 < L < \infty,$$

- Prediction interval length: $\Delta t \leq \varepsilon / L$. The number M of prediction steps:

$$M = \frac{\Delta t}{h}, h = \frac{1}{N} \sum_{i=1}^{N-1} (t_i - t_{i-1}), \rightarrow M \leq \varepsilon / (h \cdot L),$$

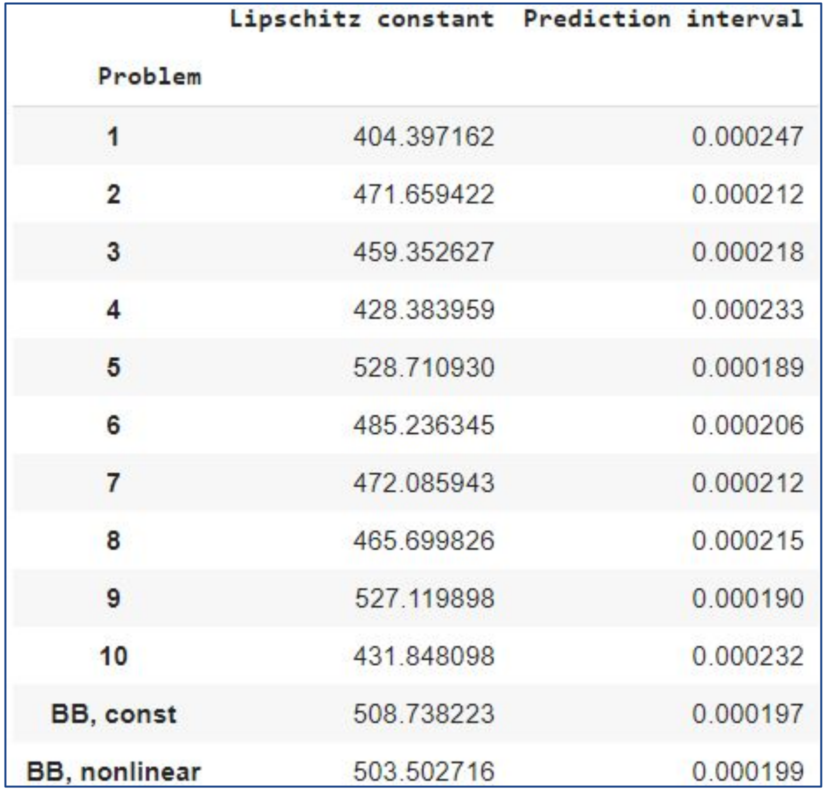
- I.e., in order to guarantee that the function g(t) will not change for more than ε, we can predict no more than ε/(hL) steps or no more than (ε/L) time (ε = 0.1 has been used in the present project).

- Adaptive Lipschitz constant estimate (see, e.g., this paper) over the initial training set:

$$L \approx \max_{t^I \neq q t^{II}} \frac{|g(t^I) - g(t^{II})|}{|t^I - t^{II}|}, t^I, t^{II} \in \{t_0, t_1, ..., t_{N-1}\}$$

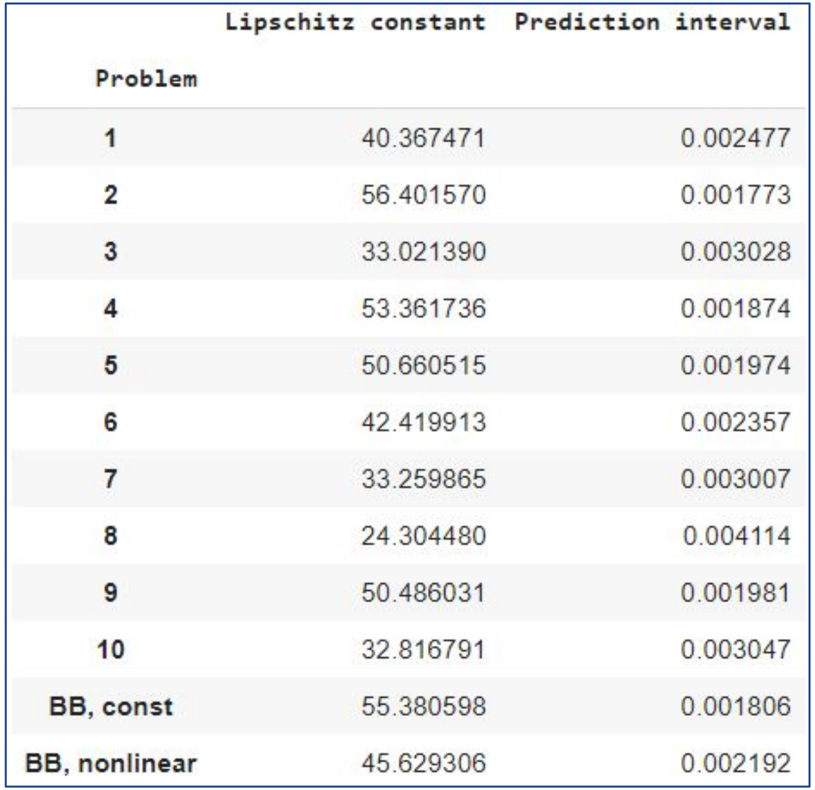
Data pre-processing: Lipschitz constant & prediction

Taking each evaluation of the function:
too sensitive to noise ->
large Lipschitz constants, small prediction intervals



Problem	Lipschitz constant	Prediction interval
1	404.397162	0.000247
2	471.659422	0.000212
3	459.352627	0.000218
4	428.383959	0.000233
5	528.710930	0.000189
6	485.236345	0.000206
7	472.085943	0.000212
8	465.699826	0.000215
9	527.119898	0.000190
10	431.848098	0.000232
BB, const	508.738223	0.000197
BB, nonlinear	503.502716	0.000199

Taking only a part of the training set:
less sensitive to noise ->
smaller Lipschitz constants, bigger prediction intervals

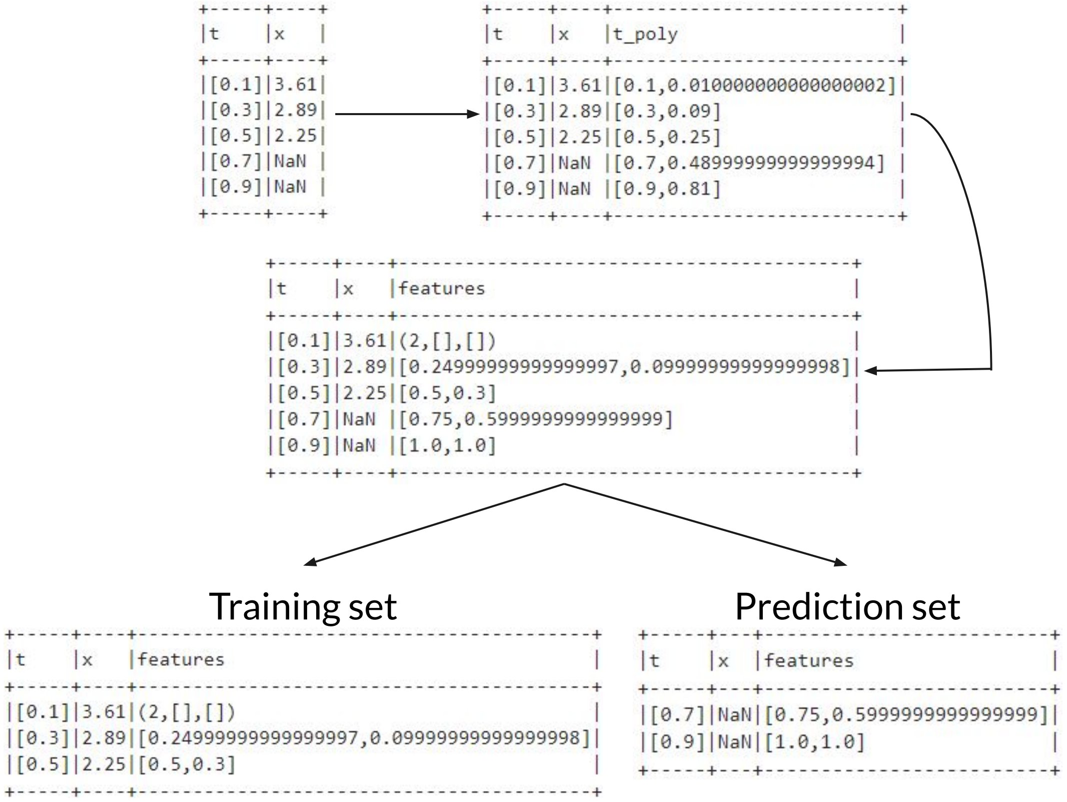


Problem	Lipschitz constant	Prediction interval
1	40.367471	0.002477
2	56.401570	0.001773
3	33.021390	0.003028
4	53.361736	0.001874
5	50.660515	0.001974
6	42.419913	0.002357
7	33.259865	0.003007
8	24.304480	0.004114
9	50.486031	0.001981
10	32.816791	0.003047
BB, const	55.380598	0.001806
BB, nonlinear	45.629306	0.002192

ETL and Feature Engineering

- Every time the new data arrives, it is checked to be numeric (all time observations t_k, where g(t_k) = NaN, are excluded).
- Array $t = [t_k, t_{k+1}, \ldots, t_{k+N-1}, \ldots, t_{k+N+M-1}]$ is transformed into polynomial features without constant (the first N values are used for training, the last M ones: for prediction). The number of degrees is determined after the model definition (see next slides).
- Obtained polynomial features are scaled into [0,1] using MinMaxScaler, since high/low degrees can have different impact:
  - $\text{if, e.g., } t = 10, \text{ then } t^{10} = 10^{10} >> t^1 = 10$
  - $\text{if, e.g., } t = 0.1, \text{ then } t^{10} = 10^{-10} << t^1 = 0.1$
- Obtained set is divided into training (the first N rows/values) and prediction (the last M rows/values) sets, which are used directly for the training/prediction.

Example: N = 3, M = 2, degrees = 2, columns: t, x=g(t).
The values of g(t) are unknown for the prediction set (initialized as NaNs)



```
+-----+----+
|t    |x   |
+-----+----+
|[0.1]|3.61|
|[0.3]|2.89|
|[0.5]|2.25|
|[0.7]|NaN |
|[0.9]|NaN |
+-----+----+
```

```
+-----+----+-------------------------+
|t    |x   |t_poly                   |
+-----+----+-------------------------+
|[0.1]|3.61|[0.1,0.010000000000000002]|
|[0.3]|2.89|[0.3,0.09]               |
|[0.5]|2.25|[0.5,0.25]               |
|[0.7]|NaN |[0.7,0.48999999999999994] |
|[0.9]|NaN |[0.9,0.81]               |
+-----+----+-------------------------+
```

```
+-----+----+---------------------------------------+
|t    |x   |features                               |
+-----+----+---------------------------------------+
|[0.1]|3.61|(2,[],[])                              |
|[0.3]|2.89|[0.24999999999999997,0.09999999999999998]|
|[0.5]|2.25|[0.5,0.3]                              |
|[0.7]|NaN |[0.75,0.5999999999999999]              |
|[0.9]|NaN |[1.0,1.0]                              |
+-----+----+---------------------------------------+
```

Training set

```
+-----+----+---------------------------------------+
|t    |x   |features                               |
+-----+----+---------------------------------------+
|[0.1]|3.61|(2,[],[])                              |
|[0.3]|2.89|[0.24999999999999997,0.09999999999999998]|
|[0.5]|2.25|[0.5,0.3]                              |
+-----+----+---------------------------------------+
```

Prediction set

```
+-----+---+-------------------------+
|t    |x  |features                 |
+-----+---+-------------------------+
|[0.7]|NaN|[0.75,0.5999999999999999]|
|[0.9]|NaN|[1.0,1.0]                |
+-----+---+-------------------------+
```

Model definition

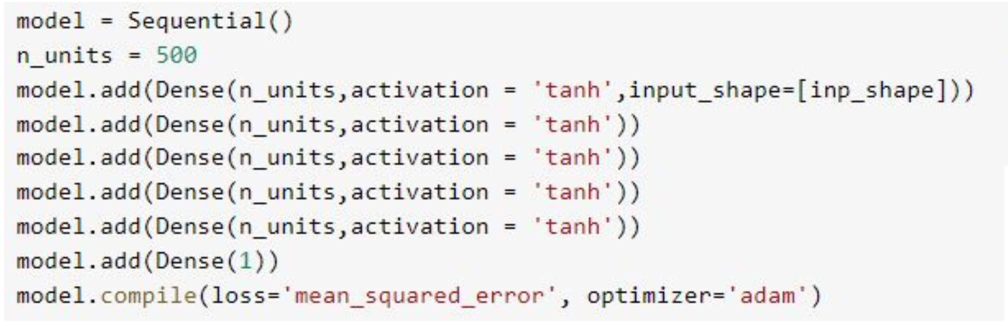
Two Machine Learning and one Deep Learning techniques have been implemented:

1. Machine Learning: linear regression after polynomial transformation and scaling:
   a. in Python by Scikit-learn and Numpy.
   b. in Python by Apache Spark and SparkML.

$$\hat{g}_i = \beta_0 + \beta_1 T_{1,i} + \ldots + \beta_D T_{D,i}, \quad T_{j,i} = (t_i^j - \min_i(t_i^j))/(\max_i(t_i^j) - \min_i(t_i^j)), j = 1, \ldots, D$$

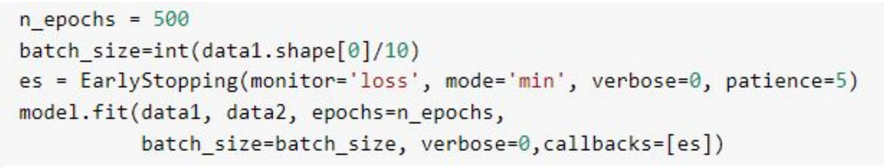
2. Deep Learning: sequential model with dense neural networks in Python by Tensorflow and Keras :
   a. Five dense layers with 500 units and the “tanh” activation function for each layer have been used (all other parameters have been set to their default values).
   b. The models have been compiled under the mean squared error loss function and “Adam” optimizer.
   c. 500 epochs, batch size = N/10 (N is the training set length), and early stopping with the patience parameter set equal to 5 have been used for fitting the models.

DL Model definition:



```
model = Sequential()
n_units = 500
model.add(Dense(n_units,activation = 'tanh',input_shape=[inp_shape]))
model.add(Dense(n_units,activation = 'tanh'))
model.add(Dense(n_units,activation = 'tanh'))
model.add(Dense(n_units,activation = 'tanh'))
model.add(Dense(n_units,activation = 'tanh'))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
```

DL Model training



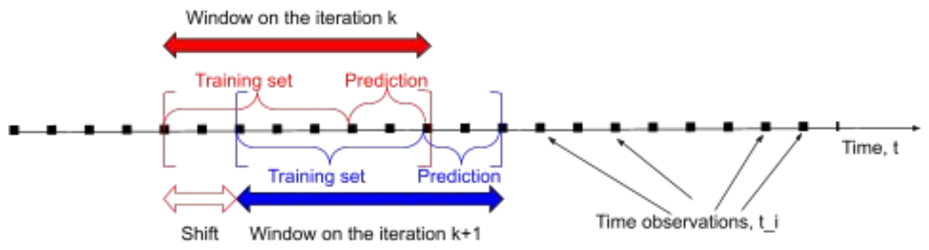
```
n_epochs = 500
batch_size=int(data1.shape[0]/10)
es = EarlyStopping(monitor='loss', mode='min', verbose=0, patience=5)
model.fit(data1, data2, epochs=n_epochs,
          batch_size=batch_size, verbose=0,callbacks=[es])
```

Model training: setting parameters

- The number M of prediction steps is determined as follows (see Exploration section):
  - The Lipschitz constant is estimated on the N known observations:
    $$L \approx \max_{t^I \neq q t^{II}} \frac{|g(t^I)-g(t^{II})|}{|t^I - t^{II}|},\ t^I, t^{II} \in \{t_0, t_1, \ldots, t_{N-1}\}$$
  - After that, the value M is estimated (ε= 0.1 has been used):
    $$M = \mathrm{int}(\varepsilon/(h \cdot L)),\ h = \frac{1}{N}\sum_{i=1}^{N-1}(t_i - t_{i-1})$$
  - After that, M is updated as M:=max(N,min(10,M)), i.e., if M was previously estimated too small (less than 10 prediction steps), then it is fixed equal to 10 (this value was chosen empirically). If M was too large (greater than N), then we fix M equal to N.
- Then, the number D of degrees for polynomial transformation of the features is also determined (see the ETL & Feature engineering):
  - Since at this moment we have only the first N observations, i.e., the training set, then, in order to determine the optimal number of degrees, we subdivide our training set into two sets with N1 and N2 = N-N1 observations, where N2 = min(N/2,M), i.e., we use min(N/2,M) observations for testing the initial model at this step.
  - Then, the model is fitted using N1 observations and polynomial degree from 1 to 30. The R2 value on the testing set using N2 observations is calculated for each value of the degree. The degree number D, which gives the highest R2 value is then selected as the number of degrees for the polynomial features for the whole model.

Model training

- Once the values M and D have been estimated, the selected model is trained using the last N known observations. New M observations are predicted. Zero-crossings are checked on the predicted set. All necessary actions are performed, if there has been determined zero-crossing.
- After that, new real M observations arrive to the system. The results obtained during the last training-prediction step can be evaluated.
- The training-prediction is repeated using the last N known observations as training set and predicting new M observations until the stopping criteria: e.g., when the time T was reached.
- To obtain better and more stable results, M and D can be re-evaluated during the simulation after one or several training-prediction cycles (higher computational costs, but higher accuracy) or can be fixed for the whole simulation process (lower computational costs, but lower accuracy).



Model evaluation

- Each model has been evaluated on the test problems as follows:
  - For each test problem, the first zero-crossing t* is known a priori.
  - If at the iteration k of the selected model's training-prediction steps there has been determined a zero-crossing among the predicted values at time t_k, such that:

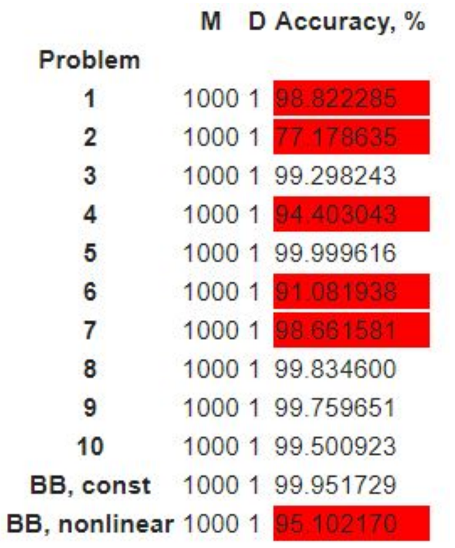
    $$100 \cdot (1 - RE) \geq 99, \quad RE = \frac{|t_k - t^*|}{|t^*|},$$

    where RE is the relative error with respect to t*, then the problem is considered to be solved.
  - Otherwise, the problem is considered to be unsolved.
  - The models and their parameters have been chosen and fitted in such the way to solve all test problems with the same parameters.
- Relative error RE represents the error in % (if multiplied by 100) with respect to the real a priori known value (used frequently in testing numerical algorithms, see, e.g., Molinaro&Sergeyev (2001)).
- Relative accuracy RA = 100(1-RE), can be also used in order to estimate the errors: RA > 99% is equivalent to RE < 0.01 or 1%.

Linear model performance between different approaches

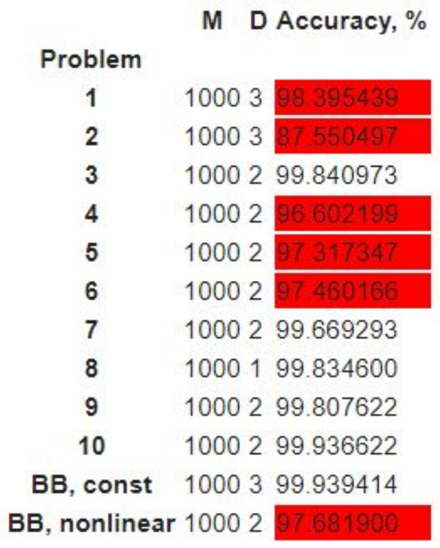
Number of prediction steps (M) and Degree of polynomial features (D): a priori choice vs. proposed adaptive
Problem is considered to be solved if Accuracy > 99%

Both M and D are fixed a priori:
M = size of training set
D = 1 (no polynomial features)
6 problems unsolved



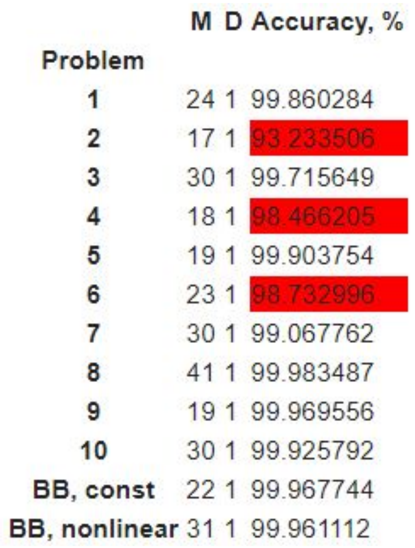
Problem	M	D	Accuracy, %
1	1000	1	98.822285
2	1000	1	77.178635
3	1000	1	99.298243
4	1000	1	94.403043
5	1000	1	99.999616
6	1000	1	91.081938
7	1000	1	98.661581
8	1000	1	99.834600
9	1000	1	99.759651
10	1000	1	99.500923
BB, const	1000	1	99.951729
BB, nonlinear	1000	1	95.102170

M is fixed, D is adaptive:
M = size of training set
6 problems unsolved
(but accuracy is better)



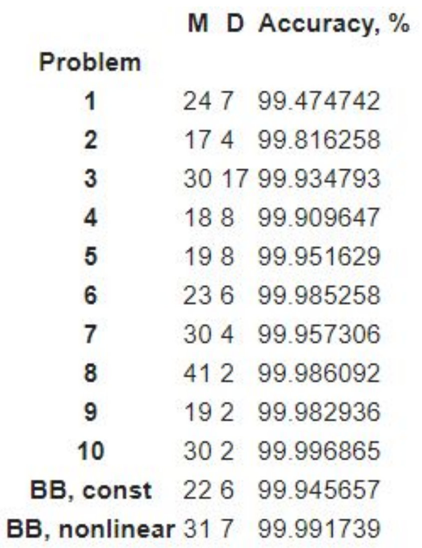
Problem	M	D	Accuracy, %
1	1000	3	98.395439
2	1000	3	87.550497
3	1000	2	99.840973
4	1000	2	96.602199
5	1000	2	97.317347
6	1000	2	97.460166
7	1000	2	99.669293
8	1000	1	99.834600
9	1000	2	99.807622
10	1000	2	99.936622
BB, const	1000	3	99.939414
BB, nonlinear	1000	2	97.681900

M is adaptive, D is fixed:
D=1 (no polynomial features)
3 problems unsolved
(but accuracy is even better)



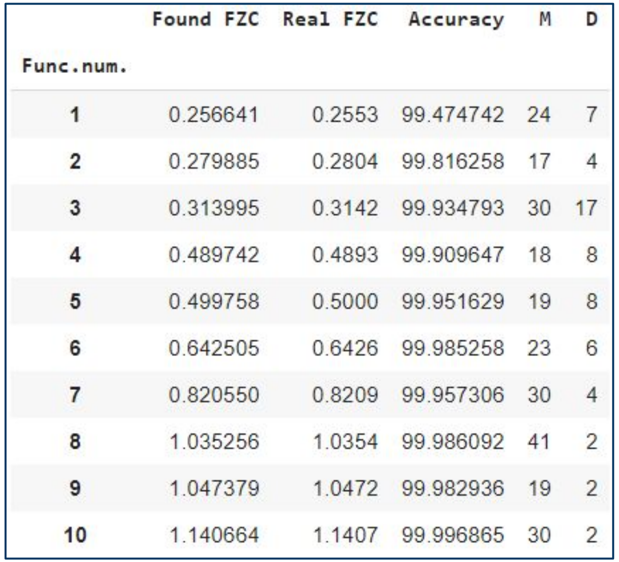
Problem	M	D	Accuracy, %
1	24	1	99.860284
2	17	1	93.233506
3	30	1	99.715649
4	18	1	98.466205
5	19	1	99.903754
6	23	1	98.732996
7	30	1	99.067762
8	41	1	99.983487
9	19	1	99.969556
10	30	1	99.925792
BB, const	22	1	99.967744
BB, nonlinear	31	1	99.961112

Both M and D are adaptive:
All problems are solved successfully

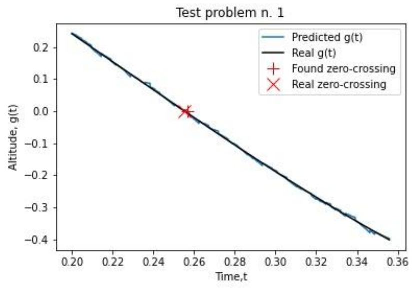


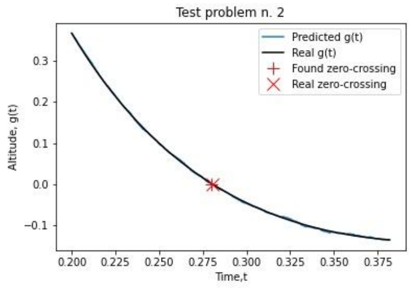
Problem	M	D	Accuracy, %
1	24	7	99.474742
2	17	4	99.816258
3	30	17	99.934793
4	18	8	99.909647
5	19	8	99.951629
6	23	6	99.985258
7	30	4	99.957306
8	41	2	99.986092
9	19	2	99.982936
10	30	2	99.996865
BB, const	22	6	99.945657
BB, nonlinear	31	7	99.991739

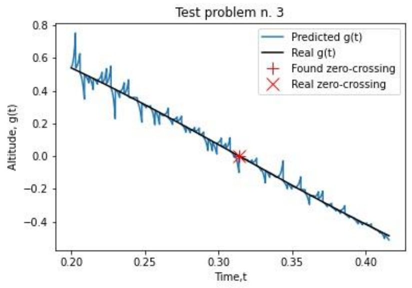
Machine Learning Results: 10 test problems

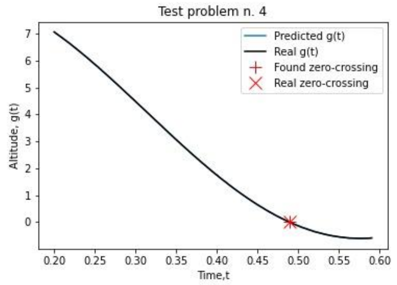


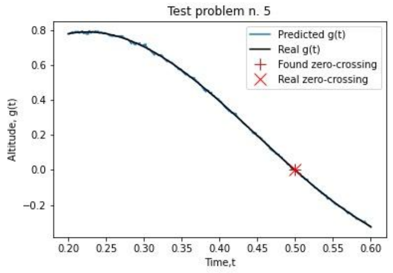
Func.num.	Found FZC	Real FZC	Accuracy	M	D
1	0.256641	0.2553	99.474742	24	7
2	0.279885	0.2804	99.816258	17	4
3	0.313995	0.3142	99.934793	30	17
4	0.489742	0.4893	99.909647	18	8
5	0.499758	0.5000	99.951629	19	8
6	0.642505	0.6426	99.985258	23	6
7	0.820550	0.8209	99.957306	30	4
8	1.035256	1.0354	99.986092	41	2
9	1.047379	1.0472	99.982936	19	2
10	1.140664	1.1407	99.996865	30	2

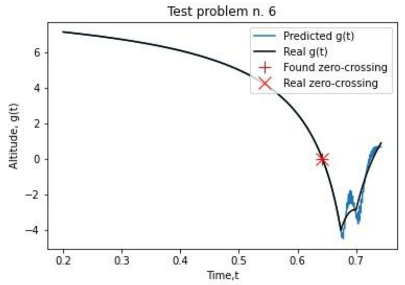


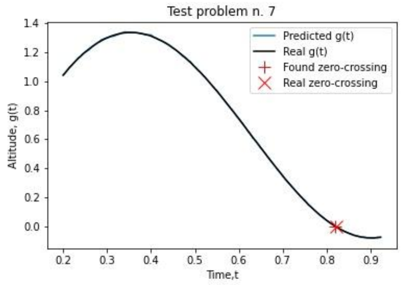


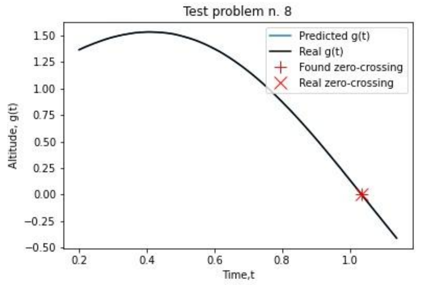


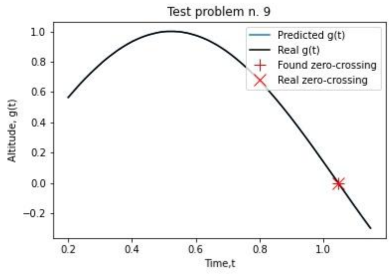


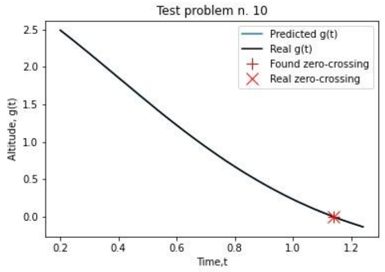




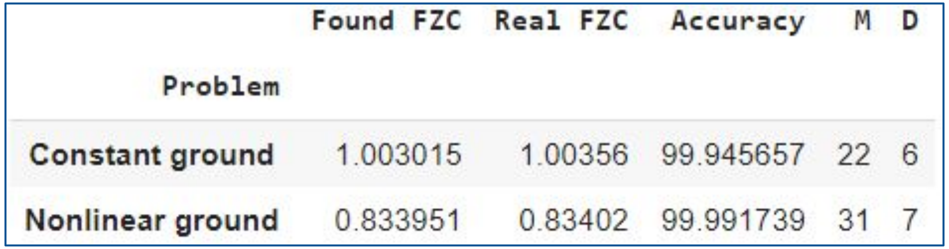




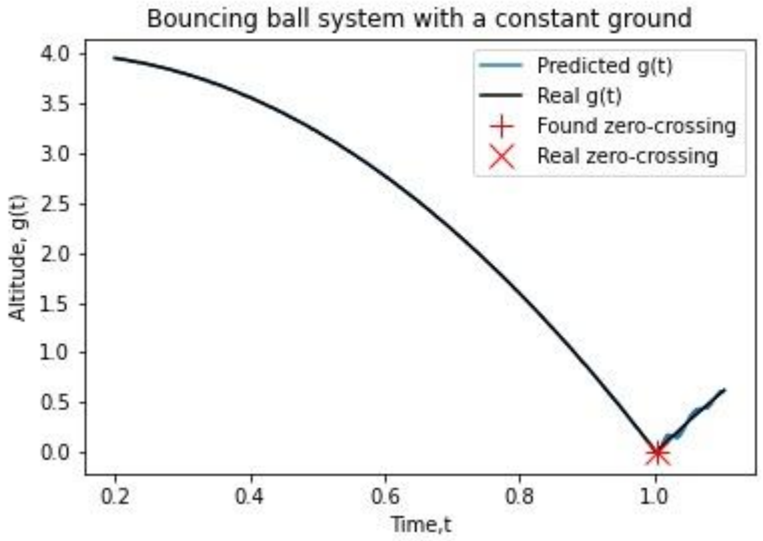


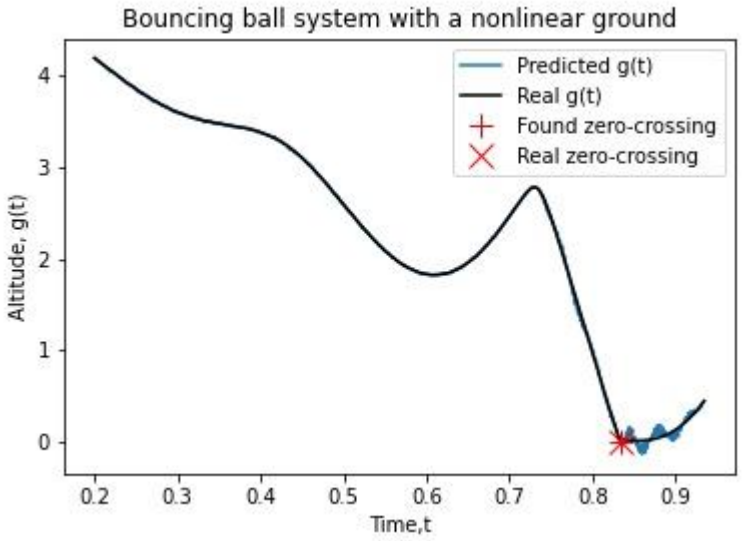


Machine Learning Results: bouncing ball systems

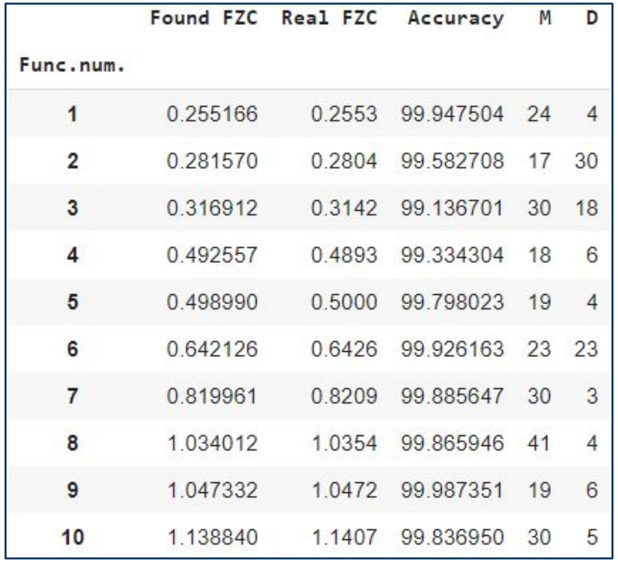


Problem	Found FZC	Real FZC	Accuracy	M	D
Constant ground	1.003015	1.00356	99.945657	22	6
Nonlinear ground	0.833951	0.83402	99.991739	31	7

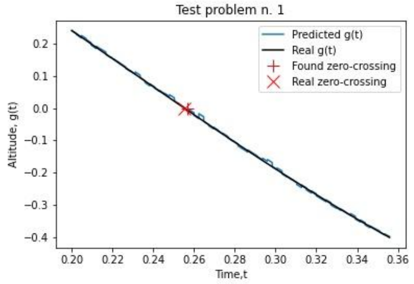


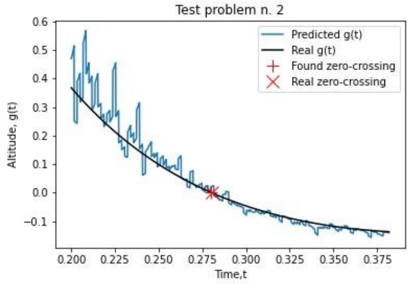


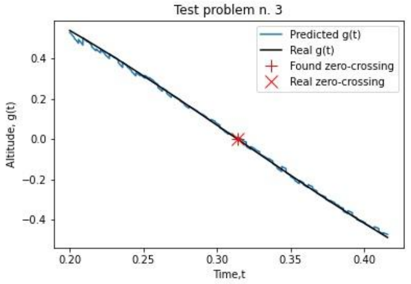
Deep Learning Results: 10 test problems

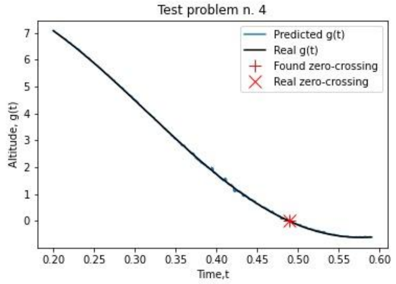


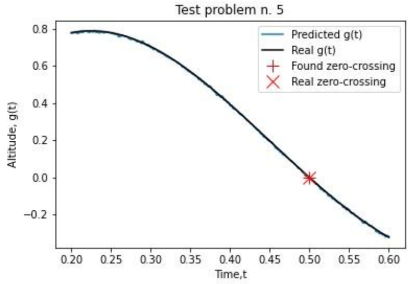
Func.num.	Found FZC	Real FZC	Accuracy	M	D
1	0.255166	0.2553	99.947504	24	4
2	0.281570	0.2804	99.582708	17	30
3	0.316912	0.3142	99.136701	30	18
4	0.492557	0.4893	99.334304	18	6
5	0.498990	0.5000	99.798023	19	4
6	0.642126	0.6426	99.926163	23	23
7	0.819961	0.8209	99.885647	30	3
8	1.034012	1.0354	99.865946	41	4
9	1.047332	1.0472	99.987351	19	6
10	1.138840	1.1407	99.836950	30	5

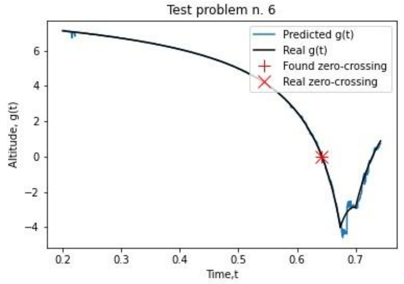


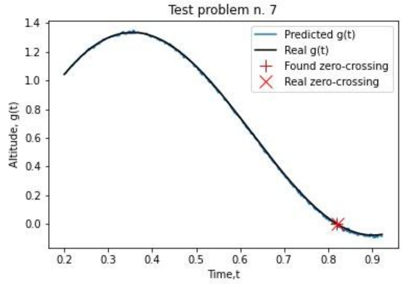


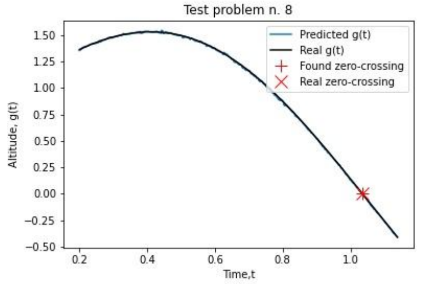


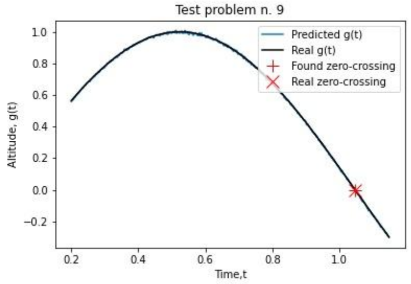


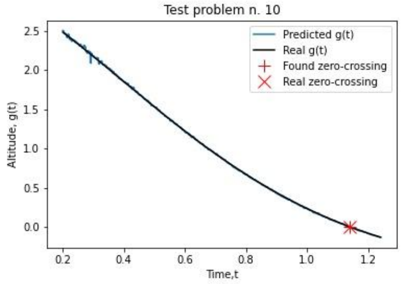




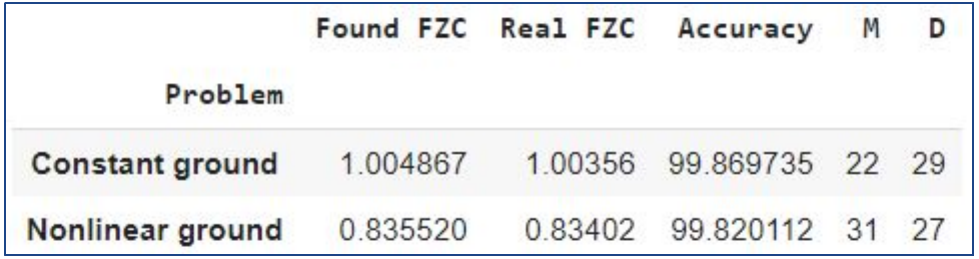




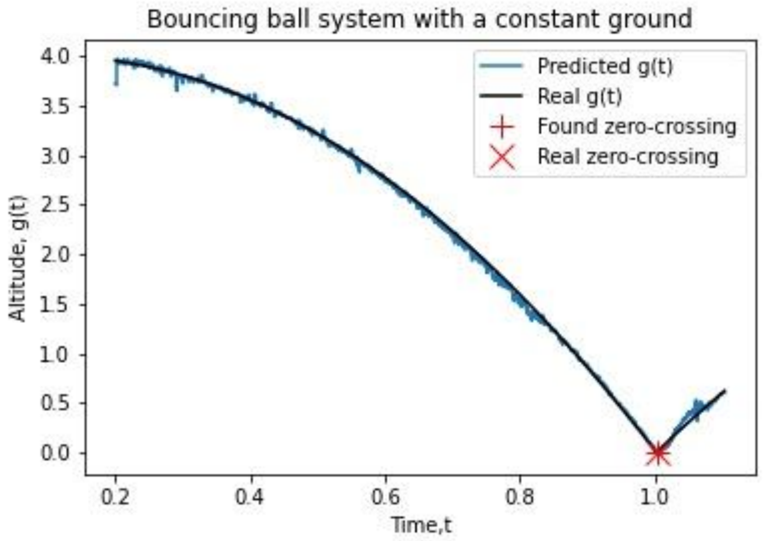


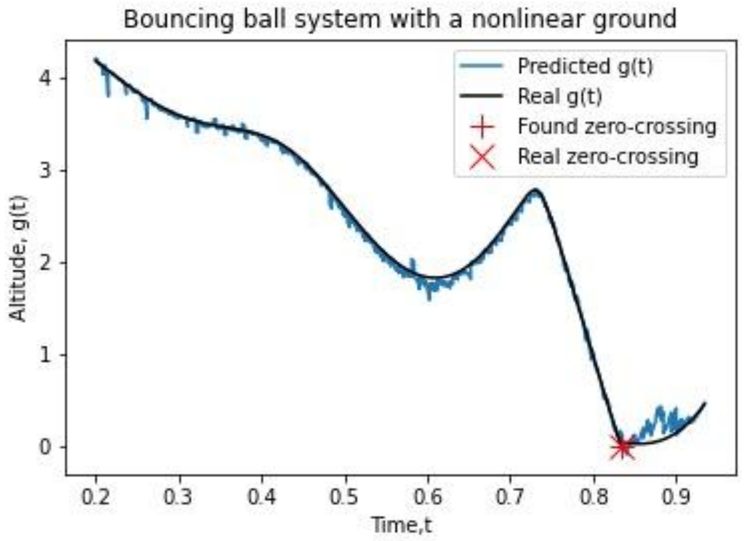


Deep Learning Results: bouncing ball systems



Problem	Found FZC	Real FZC	Accuracy	M	D
Constant ground	1.004867	1.00356	99.869735	22	29
Nonlinear ground	0.835520	0.83402	99.820112	31	27



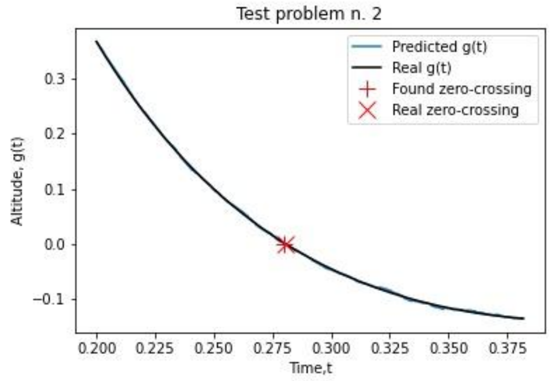


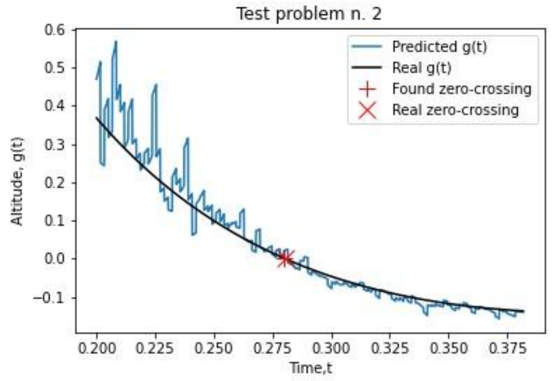
Discussion

Test problem n. 2:

ML: accurate results

DL: overfitting

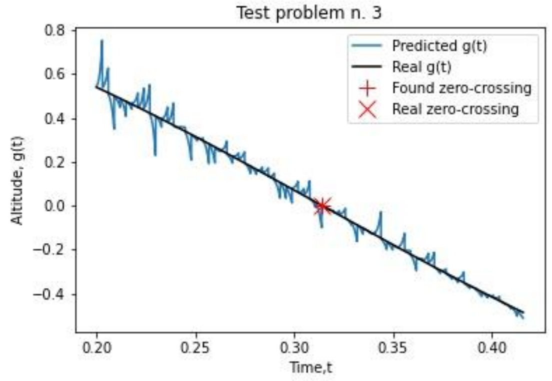


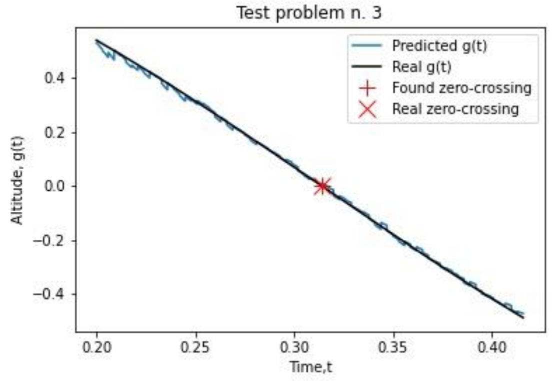


Test problem n. 3:

ML: overfitting

DL: accurate results





Model deployment: interactive Jupyter notebooks

- *[Zero-crossings_in_time_series].import_libraries.python.ipynb*
  Optional notebook for importing of all necessary libraries once.
- *[Zero-crossings_in_time_series].data_exploration.python.ipynb*
  Data exploration. Contains the implementations of all test problems, the function for Lipschitz constant estimation and all initial exploration plots.
- *[Zero-crossings_in_time_series].etl.python.ipynb*
  Extract-Transform-Load. Contains the functions for generating/getting new time observations and the objective function evaluations. Can be connected to an external dataset. All function values are checked to be numeric.
- *[Zero-crossings_in_time_series].feature_eng.python.ipynb*
  Feature engineering. Contains the function `prepare_features', which transforms the vector t into polynomial features and then scales the obtained features into [0,1]. The result is divided into training set and prediction set.
- *[Zero-crossings_in_time_series].model_def.python.ipynb*
  Model definition. Compiles the selected model with the predefined parameters.
- *[Zero-crossings_in_time_series].model_train.python.ipynb*
  Model training. Contains the functions required for model fitting and prediction. Contains the function `find_degrees`, which returns the optimal number of degrees for polynomial features.
- *[Zero-crossings_in_time_series].model_evaluate.python.ipynb*
  Model evaluation. Contains the function for checking zero-crossings and computation of the relative accuracy (if the real value t* is provided).
- *[Zero-crossings_in_time_series].model_deployment.python.ipynb*
  Model deployment. An example of the application of the project. Contains the loading of all notebooks defined above and the function for the full simulation cycle including all steps defined above.

Conclusion

- Machine and Deep learning techniques can be successfully used for checking zero-crossings of a time series “on-the-fly”.
- Deep learning techniques have much higher computational costs, but allow to use sophisticated models combining linear regression, dense layers, LSTM’s, etc., thus, obtaining any high-precision models of any desired accuracy.
- Simple linear regression after a polynomial transformation and scaling is still efficient and can be successfully used in case of limited computational resources.
- More stable and accurate results can be obtained mixing up different models: e.g., prediction of DNNs with the output of a simple linear model.
- Implemented notebooks can be easily embedded into real-life applications, where efficient determination of zero-crossings is required.
- The main parameters of the implemented methods, i.e., the number of degrees for polynomial features and the number of prediction steps are fixed automatically in an adaptive way.
- More sophisticated, accurate and efficient methods will be implemented in the future research, combining different deep learning techniques with each other and with machine learning methods and efficiently using high-performance computing, e.g., parallel computations.