

IBM Advanced Data Science Professional Certificate

Capstone Project

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

Architectural Decisions Document

author:

Marat S. Mukhametzhanov

January 2021

Table of Contents

[Table of Contents](#)

[Problem statement](#)

[Architectural Components Overview](#)

[Data Source](#)

[Technology Choice](#)

[Justification](#)

[Enterprise Data](#)

[Technology Choice](#)

[Justification](#)

[Streaming analytics](#)

[Technology Choice](#)

[Justification](#)

[Data Integration](#)

[Technology Choice](#)

[Justification](#)

[Data Repository](#)

[Technology Choice](#)

[Justification](#)

[Discovery and Exploration](#)

[Technology Choice](#)

[Justification](#)

[Actionable Insights](#)

[Technology Choice](#)

[Justification](#)

[Applications / Data Products](#)

[Technology Choice](#)

[Justification](#)

[Security, Information Governance and Systems Management](#)

[Technology Choice](#)

[Justification](#)

1 Problem statement

The project is dedicated to an efficient determination of zero-crossings in numerical simulation of noisy real-life processes. The main problem can be briefly described as follows. There is a difficult nonlinear black-box function $g(t, x(t))$, which depends on time t and a vector of variables $x(t) = (x_1(t), x_2(t), \dots, x_n(t))$, each of them depends on t as well. The system variables x can be also unknown a priori, each value $x(t)$ at a concrete time t can be obtained after a numerical simulation/solution to ordinary differential equations/taking the information from sensors, etc. So, the function $g(t, x)$ is a black-box, i.e., its analytical representation is unknown to a user. Moreover, e.g., if the values of $x(t)$ arrive with a small random error, then this function becomes noisy.

When the function $g(t, x)$ becomes equal to zero, there can occur an event: e.g., the system works correctly only when $g(t, x) > 0$, while if $g(t, x)$ becomes smaller or equal to 0, the system's behavior is incorrect (see, e.g., [Casado, Garcia, Sergeyev \(2003\)](#) or [Molinaro, Sergeyev \(2001\)](#)) and the main system's unit can even explode or break (for example, if $x(t)$ = the glucose level of a diabetic patient in mg/dl, then $g(t, x) = x(t) - c$, where $c = 40$ is a critical value, can be used. In this case, $g(t, x)$ is noisy, nonlinear (with respect to the time t), and the case $g(t, x) < 0$ can be extremely dangerous for the patient's life, so it should be predicted with a high accuracy and fast). Another example is a numerical simulation of a hybrid system: when a predefined function $g(t, x(t))$ becomes equal to zero, there occurs a discrete event (for example, a [bouncing ball system](#)). The function $g(t, x)$ is also nonlinear with respect to t , can be noisy and can have a very complicated structure (e.g., the ground level in the bouncing ball system). In this case, an incorrect determination of zero-crossings leads to an incorrect simulation (e.g., in this well-known bouncing ball problem it can lead to a ball bouncing below the ground, which is impossible from the physical point of view).

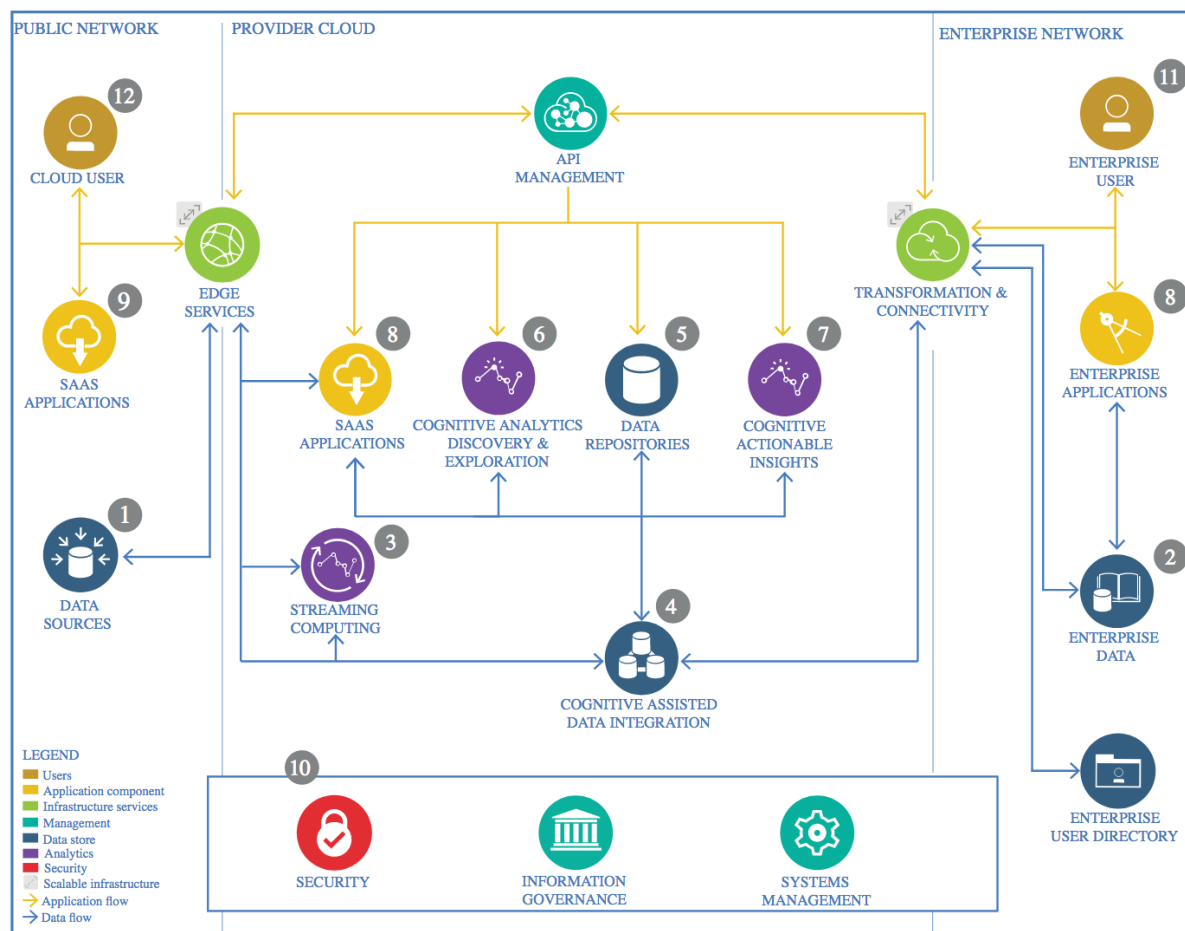
The problem of determination of zero-crossings on-the-fly is very difficult. There are several approaches to determine them (e.g., using global optimization methods from the above mentioned papers or using dynamic stepsize solvers like Dormand-Prince method), but they are not quite efficient for a generic real-life problem, since they require a possibility to calculate the function at any point t , which is not always possible. Moreover, at this moment, machine and deep learning techniques have never been used for these problems, so it can be also challenging. In the present study, I will try to apply several machine and deep learning techniques to predict the zero-crossings on-the-fly, i.e., during the simulation (in such type of problems, it's always supposed that the value of $g(t, x(t))$ at $t = t_0$, i.e., at the initial time point, is always greater than 0). Just for simplicity, we will refer to $g(t, x(t))$ as to $g(t)$ hereinafter.

The obtained results are very important from the practical point of view, since they can be used in any field, where a correct determination of zero-crossings is required: from medicine to transportations in Space. For this reason, the present study raises the interest from a very vast number of stakeholders: from pharmaceutical companies to space agencies (e.g., NASA or SpaceX). The choice of the present use case is also substantiated by a vast personal knowledge of the domain: a vast section of my Ph.D. dissertation has been dedicated to the

problem of a determination of zero-crossings as well as several my co-authored papers in international journals.

All codes, models and procedures have been implemented in Python 3.7 using the following standard Python libraries: Numpy, Math, Matplotlib, Scikit-Learn, Pyspark, SparkML, Tensorflow and Keras.

2 Architectural Components Overview



IBM Data and Analytics Reference Architecture. Source: IBM Corporation

2.1 Data Source

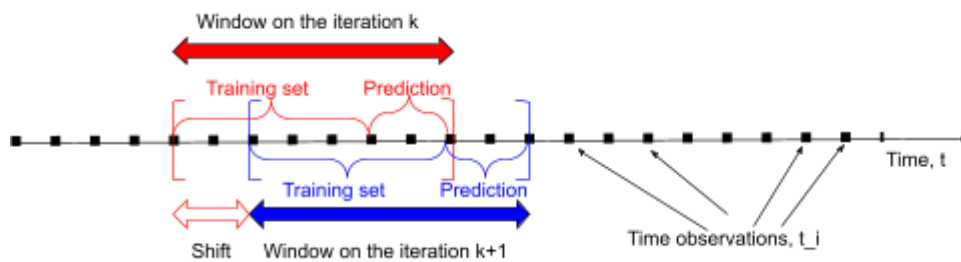
2.1.1 Technology Choice

Two types of data are used in this project:

- The data explicitly used for training each model on-the-fly during the simulation
- Test problems for evaluating the models.

The first 10 test problems from [Casado, Garcia, Sergeyev \(2003\)](#) have been used for evaluating the models. These functions are easy to implement, so they have been all implemented in Python as dictionaries with the key being the number of the problem. Moreover, two real-life problems from a well-known bouncing ball hybrid system have been also implemented in a similar way as a python class.

For each test problem, the respective time series are generated using constant time steps on a predefined time interval. The time interval is subdivided into windows with $N+M$ time steps, where the first N time steps are used for training the model, while the last M values are used for the prediction:



After each iteration, the window is moved on the right by the number of prediction steps (i.e., the shift is equal to the size of the prediction set). The observations can arrive in any numeric format in real time as, e.g., sensor data. If the number of prediction steps M is greater than 1, then there is no necessity of handling the data in real time, but it can be stored in any online or offline storage (e.g., in Cloud Object Store or in a local filesystem as a .csv file) and then, when all M new observations are ready, they should be moved as standard numeric numpy arrays to the model. Since the present project is supposed to be of general-purpose, then specific technologies for working with particular sensors or filesystems are not used here.

2.1.2 Justification

Since it is required to develop a system for an efficient determination of zero-crossings on-the-fly, then there is no a priori given dataset that can be used to train the models. The respective training set can be obtained during the first steps of simulation, since it is always supposed that $g(t, x(t)) > 0$ for t around the initial time point t_0 . Then, when new data will arrive, it will be also used for updating the model.

However, for a model evaluation we always need some test set. For this reason, we will use the first ten test functions taken from [Casado, Garcia, Sergeyev \(2003\)](#). These test functions are already widely used for testing the zero-crossing determination algorithms. They have absolutely different structures and behaviors, so the obtained results are representative. Moreover, several test problems of difficult nonlinear and noisy hybrid systems are also used for evaluation of the built models. In particular, the bouncing ball system is implemented with several nonlinear ground level functions. So, the respective datasets used in the development, evaluation and deployment of the models are the following:

- Ten test functions from [Casado, Garcia, Sergeyev \(2003\)](#) implemented on the search interval $[0.1, 7]$. The exact values of the first zero-crossings for each test function are taken from the same source.
- The standard one-dimensional bouncing ball system from [this page](#) with two different ground level functions $r(t)$ are also taken. These real-life examples are much more difficult w.r.t. the previous test problems, since the zero is not crossed, but only touched by the functions $g(t)$ in these cases. So, the methods of a higher precision are needed.
- Each test problem is simulated over the interval $[t_0, T]$ with a grid using a constant stepsize h (the same for all test problems). Then, the first N values of the function $g(t)$ are generated (it is required that the function $g(t)$ should be greater than 0 on these observations. If there is no such a guarantee, then a smaller stepsize h should be used). These N ($N = 1000$ has been used in the experiments) observations are used as training set to predict M future observations.

Zero-crossings are determined on the predicted observations (e.g., if one of them is greater than zero and another one is lower than zero, then there is definitely at least one zero-crossing). After that, M new real observations are added to the dataset and the model is updated using always the last N observations and predicting new M observations. So, the window of our time series forecasting is $N+M$, where N observations are used for the training from the past and M observations are predicted (the window can be non symmetrical, since M and N can be different).

Finally, since the proposed project can be used not only for a particular type of problems (e.g., prediction of sensor data), but for any other problem, where a high-precision determination of zero-crossings is required (for example, numerical simulation of hybrid systems, global optimization problems or other numerical modeling problems) they are handled as standard numpy arrays without using a particular technology of working with sensors or real-time streaming data. A necessary additional technology can be easily added moving the data from the sensors or a simulated process to the model transforming it to standard numpy numeric arrays.

2.2 Enterprise Data

2.2.1 Technology Choice

Cloud-based solutions are not necessary in the present project, but they can be added, if it is required not only to predict the time series, but also to handle all previous history. In the latter case, there can be large amounts of data, which can be saved on the cloud. But this issue is not related to the present project.

2.2.2 Justification

Since the models use only a finite and relatively small window of data at each iteration (e.g., working with a 1-D or 2-D arrays with thousands of elements is not costly), then there is no necessity of storing the data on the cloud. Moreover, there is also no necessity of storing the trained models, since the time series can be non stationary and a good model, e.g., trained on the interval of time t in $[0, 0.1]$, can be inaccurate on the interval $[0.5, 0.6]$.

2.3 Streaming analytics

2.3.1 Technology Choice

Batch processing without any specific technology is used in the present work.

2.3.2 Justification

The main methodology of working with time-series is batch processing. As it has been already mentioned, the time series is divided into finite windows, where the models are launched and trained. Again, the reference systems can be of a different type, so there is no unique efficient choice of a technology to use (for example, in numerical modeling the data can be processed fast without any additional technology with a good latency and fault tolerance). The present project is easily extendable to a particular technology (e.g., Apache Spark Structured Streaming or NodeRED).

2.4 Data Integration

2.4.1 Technology Choice

ETL steps are subdivided into three different choices, each of them is related to a concrete technology and model used for prediction.

1. Polynomial regression in Python using Scikit-learn:
 - 1.1. Data arrives into models as two numpy arrays t and x , where both t and x are $(N+M)$ -dimensional numeric arrays of the time observations t_i and the values of the function $g(t_i)$, respectively. The values of x are checked to be numbers and not NaN's. All non-numeric values are excluded.
 - 1.2. The array t is transformed into polynomial features using scikit-learn, i.e., into the matrix $(N+M) \times (\text{degrees}+1)$ of the following type: $[1, t, t^2, \dots, t^{\text{degrees}}]$. The number of degrees is determined later, during the model definition.
 - 1.3. The polynomial features are normalized and scaled into the interval $[0,1]$ using MinMaxScaler. Obtained features are used directly for training and prediction.
2. Polynomial regression using Apache Spark (all necessary functions are the ones of the pyspark.ml library):
 - 2.1. Data arrives into models as a spark dataframe with two columns "t" and "x". The values of x are checked to be numbers and not NaN's. All non-numeric values are excluded.
 - 2.2. The column "t" is transformed into the column of polynomial features entitled as "features". The number of degrees is determined later, during the model definition.

- 2.3. The polynomial features are normalized and scaled into the interval [0,1] using MinMaxScaler. Obtained features are used directly for training and prediction.

3. Neural Networks using tensorflow and keras:

- 3.1. Data arrives into models as two numpy arrays t and x , where both t and x are $(N+M)$ -dimensional numeric arrays of the time observations t_i and the values of the function $g(t_i)$, respectively. The values of x are checked to be numbers and not NaN's. All non-numeric values are excluded.
- 3.2. The array t is transformed into polynomial features using scikit-learn just for simplicity. The number of degrees is determined later, during the model definition.
- 3.3. The polynomial features are normalized and scaled into the interval [0,1] using MinMaxScaler of scikit-learn. Obtained features are used directly for training and prediction.

2.4.2 Justification

Since the main topic of the present project is a forecasting of a univariate time series, then there are only two arrays or columns ("t" and "x") of data available at the initial stage. The simplest and the most efficient way to treat these data consists of using simple numpy arrays. However, if the Apache Spark is available, then it can be also useful to handle these data as spark dataframes or rdd's, because Spark allows to join the data in real-time processing efficiently. Since a further transformation of data can be required, then the spark dataframes are chosen, when spark context is available, otherwise simple numpy arrays are used. The values of "x" are checked to be numeric, since sometimes, due to a technical error in reading the information from sensors, it can be NaN. The values of "t" are not checked, since they are already supposed to be numeric by default (and they can be even generated a priori).

Once the data (as numpy arrays or spark dataframe) is received, it should be transformed into polynomial features. This choice can be substantiated by a wide using of polynomial or polynomial-based approximations in many sources and by a personal knowledge of the domain (see, e.g., the following recent papers on the approximations based on the Taylor expansion: [Iavernaro et al. \(2021\)](#), [Falcone et al. \(2020\)](#), [Amodio et al. \(2017\)](#)). So, as it has been shown in many sources (see, e.g., [Falcone et al. \(2020\)](#)), a polynomial approximation can give a good accuracy for relatively small time intervals. Again, if the data is given as numpy arrays, then it can be transformed using simple scikit-learn library (the method "PolynomialFeatures"), while for Spark context there exists the respective pyspark's method PolynomialExpansion.

However, especially if the polynomial degree is high, then a further normalization is required. For example, if $t = 0.5$, then the 10-th degree polynomial features will be, respectively, [1, 0.5, 0.25, 0.125, 0.0625, 0.03125, 0.015625, 0.0078125, 0.00390625, 0.00195312], i.e., we see that the coefficients of t and t^{10} have qualitatively different impacts on the output. For $t = 10$, e.g., there will be generated the following polynomial features: [1,10,100,1000,...,10000000000], i.e., the impact of t^{10} is much higher than the

one of t or even of the constant term t^0 . For this issue, the polynomial features are all scaled into the interval $[0,1]$ using the MinMaxScaler. No further feature engineering is needed, the obtained features can be used directly to predict the values. At this moment, all necessary transformations are simple and require only basic knowledge of the Python programming language.

2.5 Data Repository

2.5.1 Technology Choice

There is no need to use a particular data repository. Any simple cloud or local repository can be used for storing the local python files.

2.5.2 Justification

The main goal of the present project is to develop a fast and efficient system for prediction of time series in order to determine their zero-crossings. For this purpose, there is no necessity of a particular data repository. All models lose their actuality right after obtaining new information from the time series, so there is no necessity to store old values or models. The developed models and python files are lite and don't require a specific repository.

2.6 Discovery and Exploration

2.6.1 Technology Choice

Plots of all test problems are useful to discover a potential complexity of the problem (simple matplotlib plots are used for this purpose).

Lipschitz constants can be estimated for the objective function on the first N evaluations. The Lipschitz constant L can be easily estimated as $L \approx \max_{i=1, \dots, N} |g(t_i) - g(t_{i-1})| / (t_i - t_{i-1})$. However, in presence of noise, this estimate can

be too sensitive to the noise, so we will use a more rough but stable estimate over only each 10-th time point, i.e., over the set $[t_{10}, t_{20}, t_{30}, \dots, t_{N-10}]$, which is less sensitive to the noise, but still efficient for the original $g(t)$. The number of prediction steps can be estimated using the Lipschitz condition: $M \leq \varepsilon / (L \cdot \Delta t)$, where ε ($\varepsilon > 0$) is a pre-defined desirable accuracy for $g(t)$.

2.6.2 Justification

Lipschitz-continuous functions arise frequently in real-life applications. The Lipschitz condition for a physical system means that the change of the energy in the system is always limited. From the numerical point of view, the Lipschitz condition can be used also for measuring the conditioning of the problem: if the Lipschitz constant is high, then for a small difference in time there can be large changes in the function, while if the constant is small, then to small differences in time correspond small changes in the function. In our case, this means that we can use the Lipschitz condition to measure how large should be the prediction set: if we will predict the values of the time series for too long time period, then the accuracy can be very poor, while very small prediction sets lead to frequent re-training

of the model and, thus, to a computational inefficiency. Formally, the Lipschitz condition for a univariate function $g(t)$ can be written as follows: $|g(t_1) - g(t_2)| \leq L \cdot |t_1 - t_2|$, where L , $0 < L < \infty$, is the Lipschitz constant over the interval $[t_0, t_0 + N \cdot \Delta t]$, where t_0 , N , Δt are the initial time, the number of steps for the training and the time stepsize, respectively (t_1, t_2 belong to this interval as well). In this case, if we want the function $g(t)$ to be changed not more than ε ($\varepsilon > 0$), then $|t_1 - t_2| \leq \varepsilon/L$, i.e., $M \cdot \Delta t \leq \varepsilon/L$, from where one can obtain $M \leq \varepsilon/(L \cdot \Delta t)$. Since we want only to limit the changes of the values of $g(t)$ on the prediction interval, then we simply use $\varepsilon = 0.1$. In this case, the changes of the function $g(t)$ are always limited on the prediction interval and are not too high, so its approximation can be efficient. It should be only mentioned that for functions having bigger Lipschitz constants, the prediction intervals will be smaller, while the functions having small Lipschitz constants (i.e., well-conditioned functions) will have larger prediction intervals.

In a vast literature, there exist several ways to estimate the Lipschitz constant (see, e.g., [Sergeyev et al. \(2016\)](#)), so the simplest, but efficient way is to estimate it using simple finite differences $L \approx \max_{i_1 \neq i_2} |g(t_{i_1}) - g(t_{i_2})| / (t_{i_1} - t_{i_2})$.

2.7 Actionable Insights

2.7.1 Technology Choice

Three models are deployed in the present project.

1. Linear regression in Python using Numpy and Scikit-learn with the polynomial features.
2. Linear regression in Python using Apache Spark and SparkML.
3. Dense Neural Networks in Python using Tensorflow and Keras. Five dense layers with 500 units and the “tanh” activation function for each layer are used (all other parameters have been set to their default values). The models have been compiled under the mean squared error loss function and “Adam” optimizer. Finally, 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.

For each model, polynomial transformation of the features is first performed (see the ETL section). The number of degrees is determined on the first training set as follows. First, one of the previously described three models is selected. Then, the number of prediction steps M is determined using the Lipschitz constant estimation as described previously. After that, M is fixed as $\max(N, \min(10, M))$, i.e., if M is too small (less than 10 prediction steps), then it is fixed equal to 10 (this value was chosen empirically). If M is too large (greater than N), then we fix M equal to N . Finally, 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 N_1 and $N_2 = N - N_1$ observations, where $N_2 = \min(N/2, M)$, 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, which gives the highest R2 value is then selected as the number of degrees for the polynomial features for the whole model.

Finally, for each model, the results are evaluated as follows. If there has been determined a zero-crossing at t_k such that $100 \cdot (1 - |t_k - t^*|/|t^*|) \geq 99$, where t^* is the a priori known first zero-crossing for the test problem, 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.

2.7.2 Justification

The first model is the simplest choice when approximating the time series. It requires only a basic knowledge of Python including Numpy and Scikit-learn. All commands and methods are simple, but still efficient, so there is almost nothing to describe at this point.

The second model is equal to the first one from the qualitative point of view, but can be more efficient from the computational point of view, since it uses Apache Spark being an efficient parallel computation framework.

Finally, the third model is related to Deep learning instead of Machine learning and uses the Tensorflow and Keras in Python. It also requires a basic knowledge of these libraries.

The third model is much more complicated with respect to a simple linear regression, since it has much more parameters to be tuned: the number of the layers (and their types), the number of units in each layer, the activation function, optimizer, number of epochs, batch size are just the most important ones, which affect more on the performance.

- The number of layers: higher the number of layers, more accurate will be the results, but an overfitting is possible. At this point, there is no necessity to use a high number of the layers, so only 3 layers have been used (empirically it has been obtained that 3 layers is the most stable value). Simple but still efficient Dense layers have been chosen for this purpose being more efficient at this point than the LSTM layers (after preliminary empirical research).
- Number of units in each layer: higher the number of units, more accurate will be the results, but again, an overfitting is possible. So, empirically, the number of units = 100 has been chosen as the best value.
- The activation function: it has been obtained that the activation function affects a lot on the velocity of the convergence over the epochs. It has been also obtained that the “tanh” function gives the fastest convergence, so it has been used for all layers.
- Optimizer: it has been also obtained that the optimizer affects a lot on the convergence over the epochs, so the Adam optimizer has been chosen as the best one for the present problem.
- Number of epochs: higher number of epochs, more accurate will be results, but only until some limit, after which the results are not improving more. Moreover, after this

limit (unknown a priori), the overfitting is possible, giving higher results over the training set, but poor results over the prediction set. So, a high number of epochs (500) with early stopping using patience = 5 has been used. This means that the training is stopped if the result is not improved for 5 consecutive epochs.

- Batch size: this parameter affects the performance of the model as well. It has been obtained empirically that the value N/10 is the best one for the present problem.

The R2-value is chosen to determine the optimal number of degrees in polynomial features, since it is the most widely used criterion in linear regression for determination of the number of features. Other criteria can be also used instead: for example, the adjusted R2 value on the whole N observations of the training set (in this case we don't need to divide it into two sets). But, empirical results show that the R2 and Adjusted-R2 values can be too high for the training set (always around 0.99-1), which can be also caused by an overfitting. To overcome this issue, the standard R2-value is used, but on the test set, instead of the training set.

Finally, the results of the models are evaluated using the relative errors (ER) and the relative accuracy (AR, in %): $ER = |t_k - t^*|/|t^*|$, $AR = 100 \cdot (1 - ER)$, where t^* is the known a priori first zero-crossing for each test problem and t_k is the approximated zero-crossing closest to t^* (due to the presence of noise and approximation of the function $g(t)$, there can be found false zero-crossings during the prediction, but this case just means that the system needs to be simulated with more attention and does not affect the real simulation). The justification of this choice is the following.

First, as it has been mentioned previously, the unit simulated as the time series under the study works correctly only when $g(t) > 0$. When $g(t)$ becomes negative or equal to zero, there can occur a discrete event and/or even explosion or destruction of the unit or its parts. For this point, it is strictly important to determine the first zero-crossing correctly. Otherwise, there is no sense to continue the simulation, if the first zero-crossing was passed.

Second, the indicated relative error is already widely used in the literature related to the problem (see, e.g., [Molinaro, Sergeyev \(2001\)](#)). Since the relative error indicates the error in % (if multiplied by 100), then the relative accuracy AR can be also used in this case to indicate the accuracy of determining the zero-crossing correctly. If the model allows to determine the first zero-crossing correctly.

2.8 Applications / Data Products

2.8.1 Technology Choice

The final application is presented as python codes that can be easily applied for merging to the real-life application. A series of Jupyter notebooks has been developed: each notebook is related to the respective step of the process: libraries loading, initial exploration, ETL, feature engineering, model definition, model training, model evaluation, and model deployment. The last notebook is created as an example for the whole simulation process: it calls all other notebooks loading the procedures and functions defined within them and defines two simple procedures: one for an example of the whole simulation process and another for the final evaluation, visualization and saving of the results.

2.8.2 Justification

Since the project is supposed to be general-purpose, then a particular sector-specific technology is not needed at this point. The simplest choice is to implement a simple, but efficient series of Jupyter notebooks in order to make it then possible to use all implemented models in practice for any application of the studied type.

2.9 Security, Information Governance and Systems Management

2.9.1 Technology Choice

Security, Information governance and systems management are not required in this project.

2.9.2 Justification

The main output or product of the present project is a simple software, i.e., python notebooks. It does not require any additional security or information governance technologies, since the input of this class is just the time series.