NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

FYP INTERIM REPORT

TITLE: State-of-the-art AI integration methodologies & framework.

Name: Sherwin Samson.

Matriculation Number: U2020911J.

Date: 29/01/2024.

<u>Table Of Contents:</u>

1. Project Summary:

Currently, the integration of AI components is complex and requires ad-hoc developments that are error-prone and repetitive. The result can be systems that do not work as expected as an integrated whole. Indeed, it is not enough for the individual pieces to be functional in a software system. It is also key that their interfaces are well-defined, that components can communicate consistently for the integrated system to function as expected. For example, if one sub-system provides an unexpected input to the next system, that may lead to wrong behaviours and potential catastrophic failure.

We aim to evaluate the state-of-the-art framework and methods available for the integration of AI systems and develop a simple application applying them. We will consider applications in the domain of Hybrid AI (also physics-inspired AI) for Smart Cities.

2. The Spring System Model:

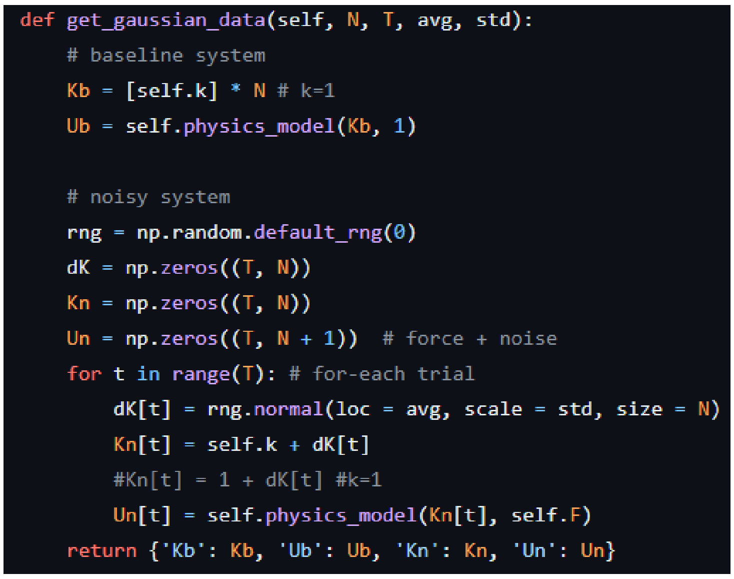
a. The Spring System is intended as an illustration of the Hybrid AI approach. Hybrid AI refers to systems that blend aspects of data-based (traditional black-box AI) with aspects of rule-based systems (logic or physics models). In particular, here we use a physics model as a surrogate/replacement of the real system that we do not have access to.

b. We considered a system of springs subject to Hooke's law. Hooke's law states that, for a spring with one end attached to a fixed object and another free end that is subject to some force F. If u is the length of extension/compression of the spring from its original position at rest and k is the spring stiffness, we have that $F = ku$ (Wikipedia, 2023).

c. First, we developed this physics model (input K, output U, $K = f(U)$). The system of spring is composed of a series of springs with a certain stiffness K. When a force F is applied, these springs show a displacement U.

3. The Regression Model:

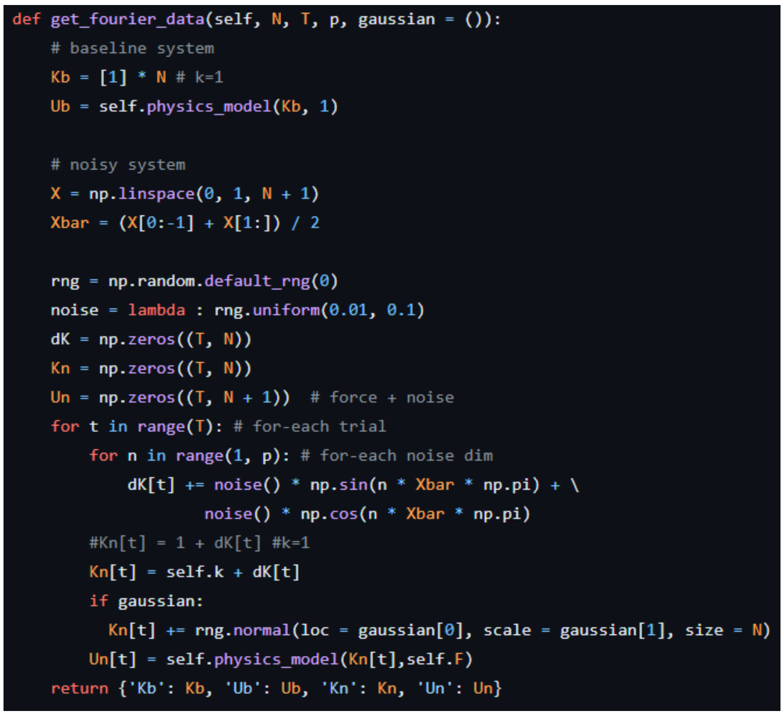
a. Then, using the data from the physics model (and including perturbations to make it more realistic), we train a data model to solve the inverse problem (input U, output K, $K = f^{-1}(U)$).
b. To train this data model, we first randomly generated 1000 sets of data following the physics model formula along with perturbations like Gaussian or Fourier noise added to the system to get the outputs.



```python
def get_gaussian_data(self, N, T, avg, std):
    # baseline system
    Kb = [self.k] * N # k=1
    Ub = self.physics_model(Kb, 1)

    # noisy system
    rng = np.random.default_rng(0)
    dK = np.zeros((T, N))
    Kn = np.zeros((T, N))
    Un = np.zeros((T, N + 1))  # force + noise
    for t in range(T): # for-each trial
        dK[t] = rng.normal(loc = avg, scale = std, size = N)
        Kn[t] = self.k + dK[t]
        #Kn[t] = 1 + dK[t] #k=1
        Un[t] = self.physics_model(Kn[t], self.F)
    return {'Kb': Kb, 'Ub': Ub, 'Kn': Kn, 'Un': Un}
```

*[**Image 1**: Adding Gaussian Noise]*



```python
def get_fourier_data(self, N, T, p, gaussian = ()):
    # baseline system
    Kb = [1] * N # k=1
    Ub = self.physics_model(Kb, 1)

    # noisy system
    X = np.linspace(0, 1, N + 1)
    Xbar = (X[0:-1] + X[1:]) / 2

    rng = np.random.default_rng(0)
    noise = lambda : rng.uniform(0.01, 0.1)
    dK = np.zeros((T, N))
    Kn = np.zeros((T, N))
    Un = np.zeros((T, N + 1))  # force + noise
    for t in range(T): # for-each trial
        for n in range(1, p): # for-each noise dim
            dK[t] += noise() * np.sin(n * Xbar * np.pi) + \
                    noise() * np.cos(n * Xbar * np.pi)
        #Kn[t] = 1 + dK[t] #k=1
        Kn[t] = self.k + dK[t]
        if gaussian:
          Kn[t] += rng.normal(loc = gaussian[0], scale = gaussian[1], size = N)
        Un[t] = self.physics_model(Kn[t],self.F)
    return {'Kb': Kb, 'Ub': Ub, 'Kn': Kn, 'Un': Un}
```

*[**Image 2**: Adding Fourier Noise]*

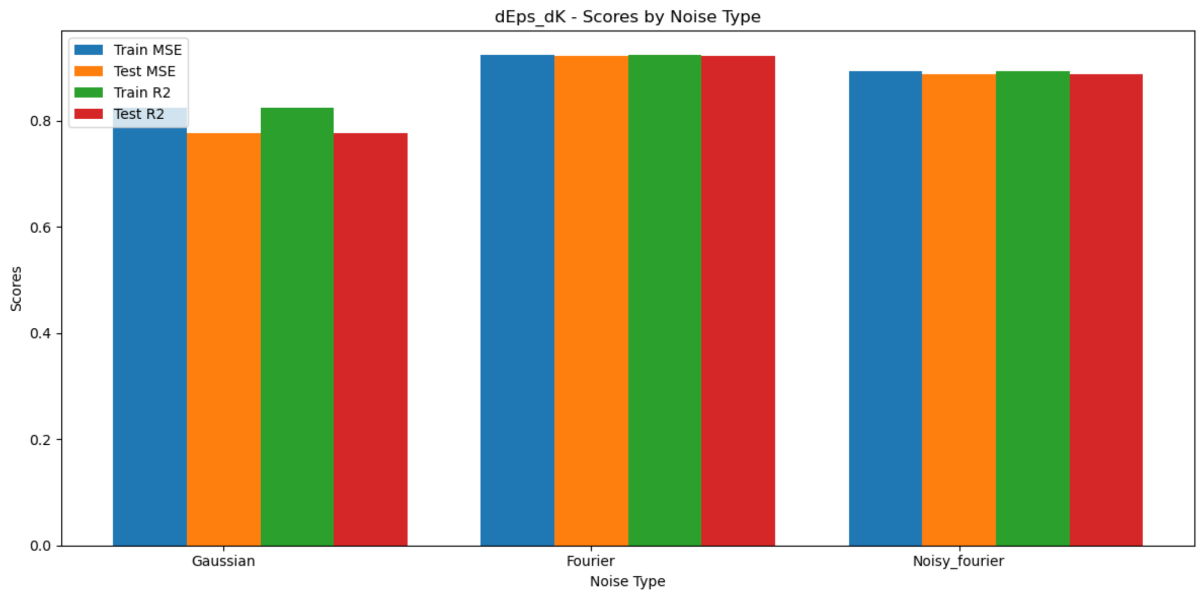
c. Essentially, the parameters for the functions include **N**, the number of elements (100); **T**, representing the number of trials; and **avg** and **std**, which are the mean and standard deviation for the noise to be added.
d. In the baseline system, arrays **Kb** and **Ub** are defined. **Kb** is an array of length **N**, where each element is set to 1, assuming a constant value. **Ub** is then calculated using the **physics_model** function with **Kb** and a fixed force for standardisation, which is set to 1.
e. For the noisy system, the function first initializes a random number generator with a fixed seed to ensure reproducibility. Arrays **dK**, **Kn**, and **Un** are then created to store the noise values, the parameters of the noisy system, and the outputs from the physics model, respectively.

f. During each trial, Gaussian noise is added to the system: **dK[t]** is filled with random values drawn from a Gaussian distribution with the specified mean and standard deviation. This noise is then added to the baseline value of **k**=1to form **Kn[t]**. The **Un[t]** array is then calculated by passing **Kn[t]** and force **F** to the **physics_model**.

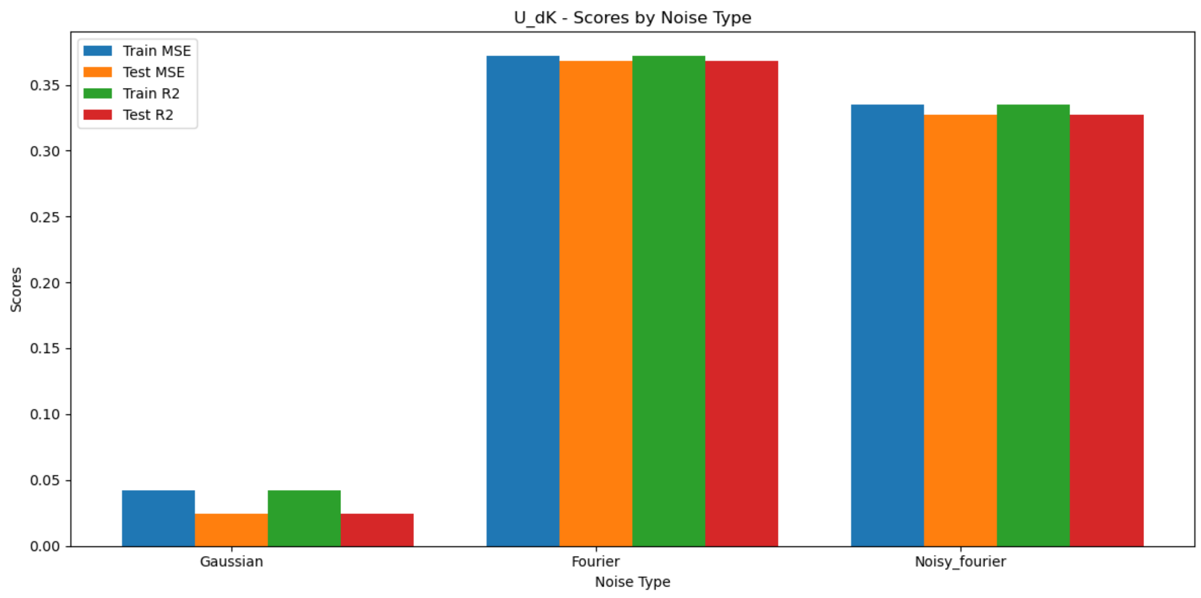
g. Multiple experimentations were done with the varying noise_type and data_transform methods:
    a. dEps/dK with Gaussian, Fourier & Noisy_fourier noise.
    b. U/dK with Gaussian, Fourier & Noisy_fourier noise.
    c. dU/dK with Gaussian, Fourier & Noisy_fourier noise.

h. Subsequently, PCA (Principal Component Analysis) was applied to the input data to reduce its dimensionality while preserving 99% of the explained variance. Then, the linear regression model was trained on the transformed data to compare the results.
    a. (PCA) dEps/dK with Gaussian, Fourier & Noisy_fourier noise.
    b. (PCA) U/dK with Gaussian, Fourier & Noisy_fourier noise.
    c. (PCA) dU/dK with Gaussian, Fourier & Noisy_fourier noise.

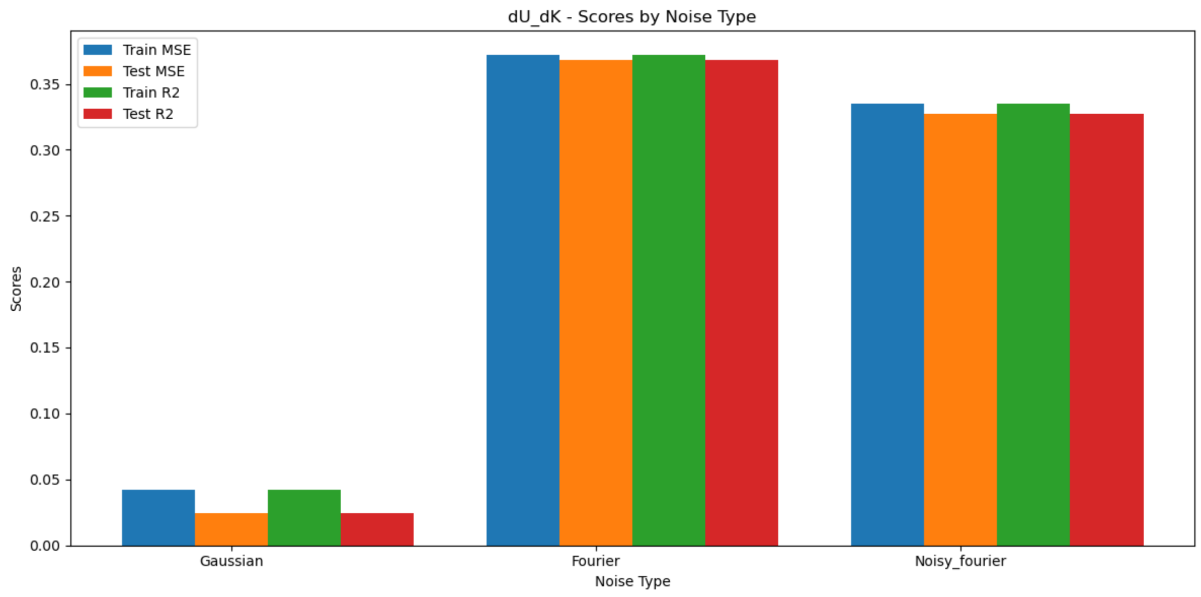
i. Train and test MSE and R2 scores comparison by Noise:



*[**Image 3**: Results of dEps/dK predictions]*



*[**Image 4**: Results of U/dK predictions]*



*[**Image 5**: Results of dU/dK predictions]*

h. As observed, when the inputs were introduced with Gaussian noise, the regression model struggled to predict for all cases.

4. Python based approach:

Creating 2 separate models for the spring system and the regression model and using their class instances to test the simulation.

a. springModel.py:

The first model represented the spring system itself, with its behaviour or function to provide the resulting displacement du in the system, for the given inputs K (Stiffness of Spring) and F (Force applied) based on the formula.

b. regressionModel.py:

The Second model represented the regression model, where a set of 1000 random inputs were used to generate 1000 trials from the spring model's output. Subsequently, a linear regression model was fitted on a portion of these inputs and outputs for training and testing. Finally the mse and r2 scores were collected to validate the accuracy of training and test using linear regression.

c. combinedSimulation.py:

Finally, we test the working of both models in unison. An instance is created for each of the SpringModel and RegressionModel classes, and the simulation is executed. ie: after generating 1000 trials of outputs using the get_data() function from the SpringModel class, the instance of RegressionModel class, trains the model with this data and outputs the resulting train and test scores. The python simulation works perfectly with no issues and validates the same results of model training or prediction.

5. Python-FMU-2 approach:

Experimentation of the Spring System's Predictive Model using FMU in accordance with the FMI standard of co-simulation. This is generally done to simplify the creation, storage, exchange and (re-) use of dynamic system models of different simulation systems for model/software/hardware-in-the-loop simulation and other applications.

The process of creating an FMU using Python-FMU is split into 3 steps:

(I) Generating the project:

(II) Implementing its behaviour:

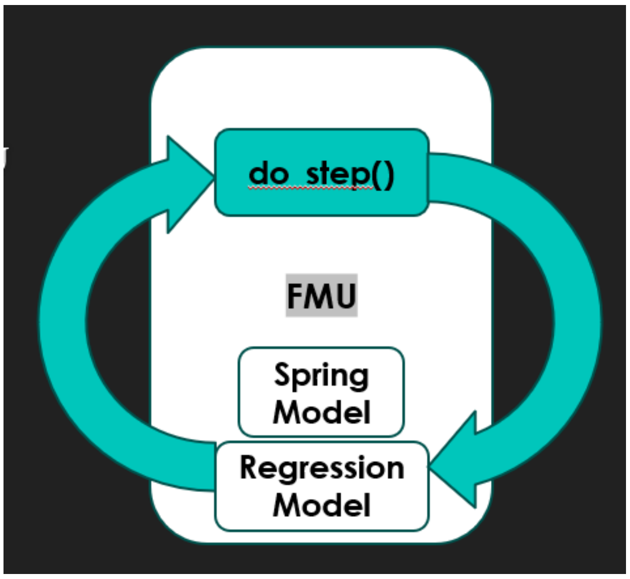
(III) Exporting the project as an FMU.

a. python-fmu2: packaging of python3 code as co-simulation FMUs following FMI version 2.0:

pyfmu.fmi2.fmi2slave is the crucial module in the PyFMU package used for creating and managing FMU (Functional Mock-up Unit) slaves in Python. It includes functionalities for configuring the FMI (Functional Mock-up Interface) slave, managing variables, handling logging, and implementing the FMI co-simulation interface. Key functions include initialising and terminating simulations, stepping through simulation time, and setting or getting various types of variables like integers, reals, booleans, and strings. The module is designed to facilitate the integration of Python code into co-simulation environments compatible with the FMI standard. Started off by reproducing all the available examples given by python-fmu libraries to build and load python functions as an FMU file.

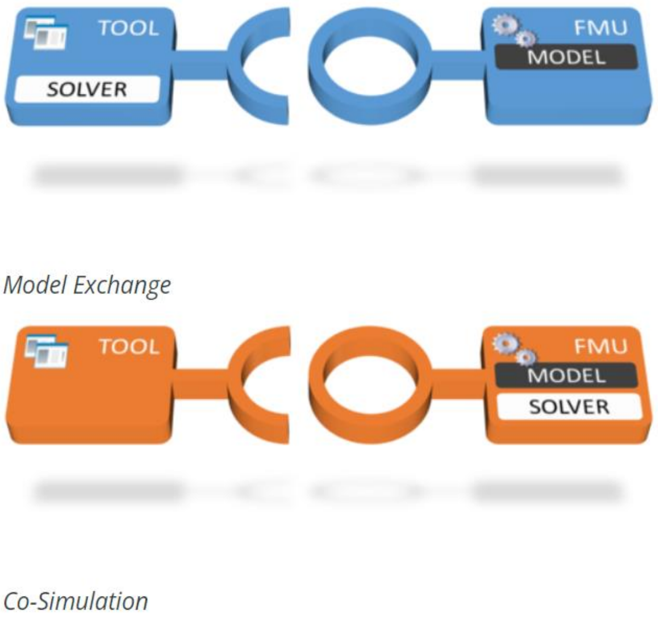
Now that we have generated the ground truth values and results from the Python based approach above, we can validate the FMU based simulation with these results.

1. **Generation of Python-fmu-2 file**:



*[**Image 6**: Spring Model's Python-FMU-2 Pipeline]*

2. **FMI 2.0 Simulation of FMU file**:



*[**Image 7**: Difference between Model Exchange and Co-Simulation FMU]* [1]

The FMI standard supports either co-simulation or model exchange FMUs. Although both FMUs can be simulated, the key difference is due to the location of the FMU's solver.

Simulating Model-Exchange FMU: The numerical solver is supplied by the importing tool. The FMU provides functions to set the state, inputs and to compute the state derivatives. The solver in the importing tool will determine what time steps to use, and how to compute the state at the next time step (Modelon, 2023).

Simulating Co-simulation FMU: The numerical solver is embedded and supplied by the exporting tool. The importing tool will set the inputs, step forward a given time and read the outputs (Modelon, 2023).

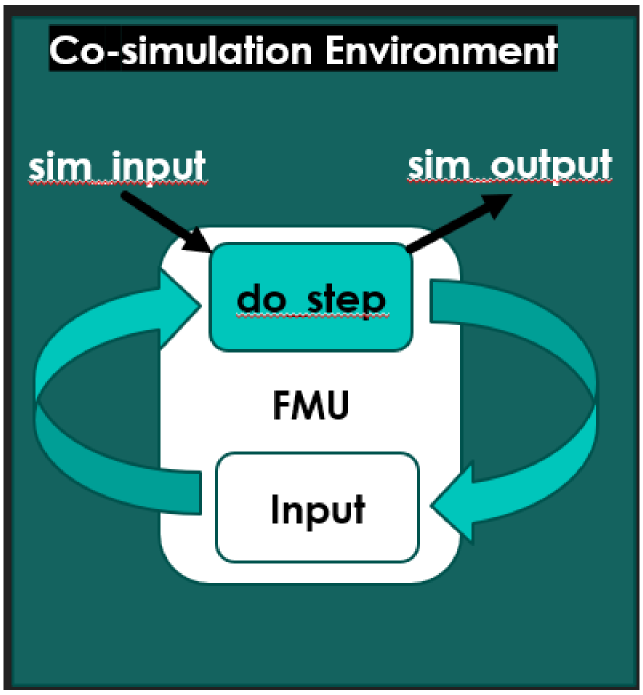
Then, the FMU file is simulated for Co-Simulation. The regression model is pre-trained for an array input with (1, 100) float values from 0-1 (Force =1 with 100 springs in the Spring System with their corresponding stiffness K) and stored locally. Due to this, the prediction is fast for any input without having to train the model repeatedly. A random (1,100) binary array is generated for each runtime.

- The key issue found when running co-simulation of our generated FMU was that python-fmu2, only allows 4 types of variables to be registered. i.e. Boolean, Integer, Real and String.
- This means that array inputs are not recognised unless they are flattened to be individually structured scalar inputs.
- Even when an array with 100 values is flattened and stored as input for simulation, the input calculation is immensely complex both time and space wise.

6. Python-FMU-3 approach:

Hence, python-fmu3 is required for the packaging of python3 code as co-simulation FMUs following FMI version 3.0. With the introduction of python-fmu3 that followed the FMI version 3.0 in 2023, there were some crucial advancements made, mainly the recognition of more variable types: Boolean, Enumeration, Int32, Int64, UInt64, Float64, String and Dimension.

I. Encapsulated FMU Simulation:



*[**Image 8**: Encapsulated Simulation pipeline for Python-FMU-3]*

- With fmu3, the array input for our prediction was able to be registered under Float64 including the necessary dimension of the array.
- The fmu files were simulated for each randomly generated (1,100) input producing unique outputs accordingly.
- Although the inputs were randomly generated, they were introduced during the do_step() function of the fmu simulation for each time step. This means that at every increment of the time step, only preset values during the fmu creation were able to be simulated.
- This use case will be applicable when the inputs are static, but the function's output changes with respect to the time step of simulation.
- However, for our case, the simulation of fmu files should be through the passing of inputs from an external source. As we are assuming the Spring model as a surrogate/replacement of a real system that we do not have access to in a Hybrid AI approach. The inputs have to be passed.

II. External FMU simulation:

1. **Generation of the Python-FMU-3 file**:

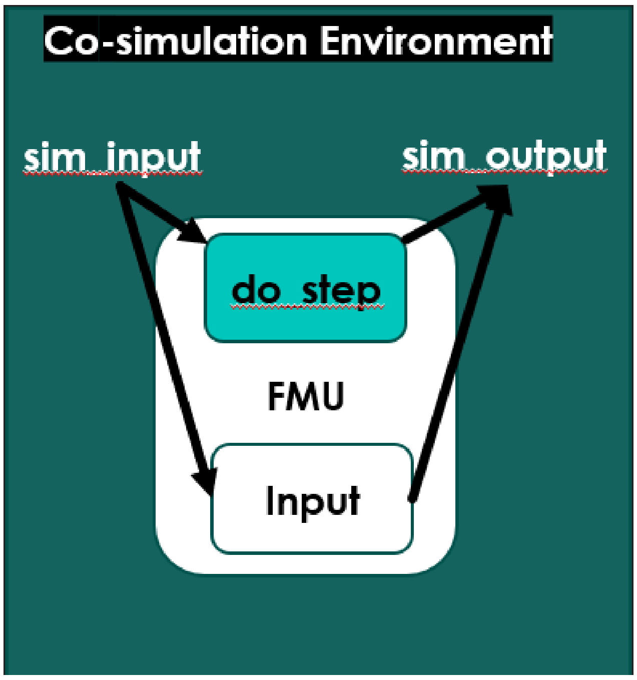
   a. Importing the locally stored pretrained Regression model.

   b. Initialized input and output parameters with values set to zero.

   c. At each time step, the inputs are scaled, and passed to the regression model for prediction.

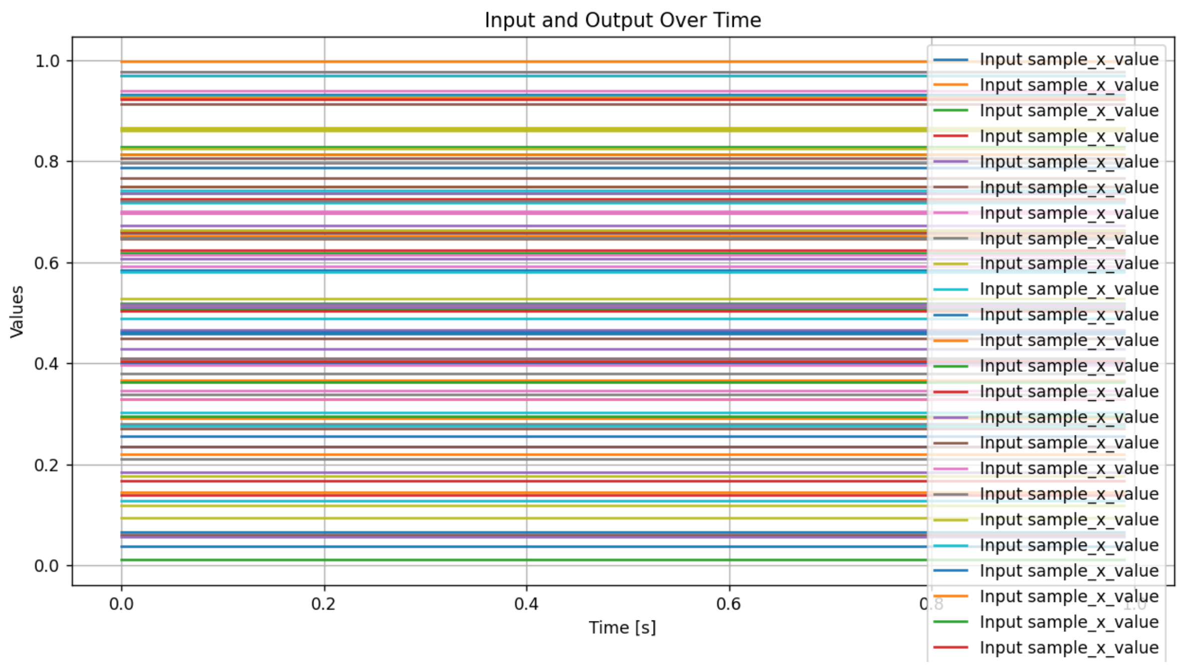
   d. This prediction is fast as it does not train the model repeatedly for each input.

2. **Custom Simulation of the FMU-3 file in fmpy**:



*[**Image 9**: External Simulation pipeline for Python-FMU-3]*

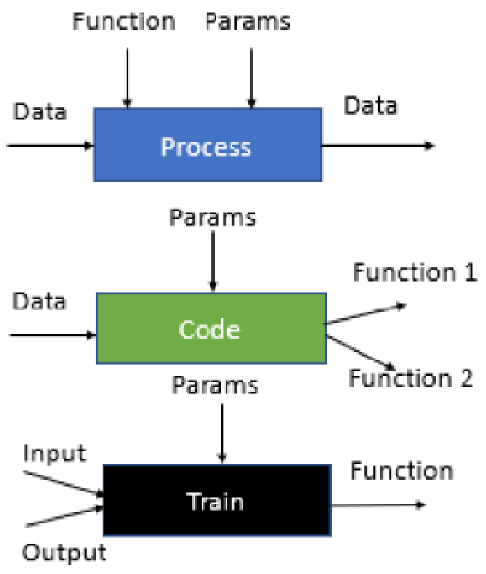
a. Initialise start_time, stop_time and step_size for simulation.
b. Read model_description to extract the FMU's structure, variables, value_references and the model_identifier.
c. Extract and unzip the FMU file to a directory.
d. FMU is instantiated as a slave object and an instance is created.
e. 'test_array' is initialized with (1,100) random numbers to simulate input data.
f. The simulation loop iterates from 'start_time' to 'stop_time' in steps of 'step_size'.
g. At each time step, the 'test_array' is flattened and set as a list for the FMU's input variable 'sample_x'. The resulting output variable 'y_pred' is retrieved and results are appended.
h. FMU instance is terminated, and memory resources are freed.



*[**Image 10**: External Simulation Results vs Time]*

- Although the input variables are initialized during FMU generation, the value of the inputs are only passed during the simulation.
- The results of the prediction of the given inputs are returned at each time step.
- No change in input or output with respect to time.

7. (Current stage) Spring model FMU integration with Kedro:



*[**Image 11**: Example pipeline for an online training model]*

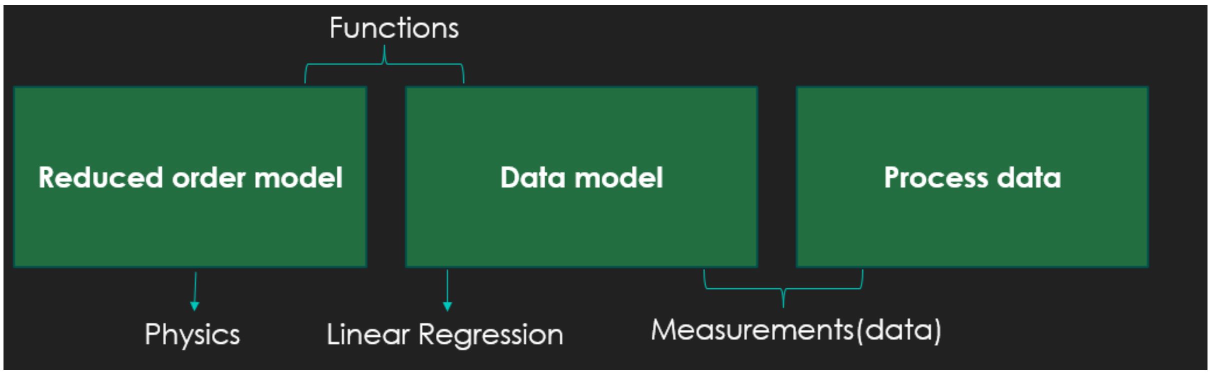
- Kedro, is a builder framework for Hybrid AI models where we can connect separate building blocks together, validate them and execute them.
- We define a set of Objects and Operators that are manipulated by the Pipeline. Objects can either be Data or Function
    - Data: The numerical values to process, the physics inputs for our case.
    - Function: Trained regression models (for traditional AI), the spring system model, etc.

a. Represent Spring system predictive model in the builder.

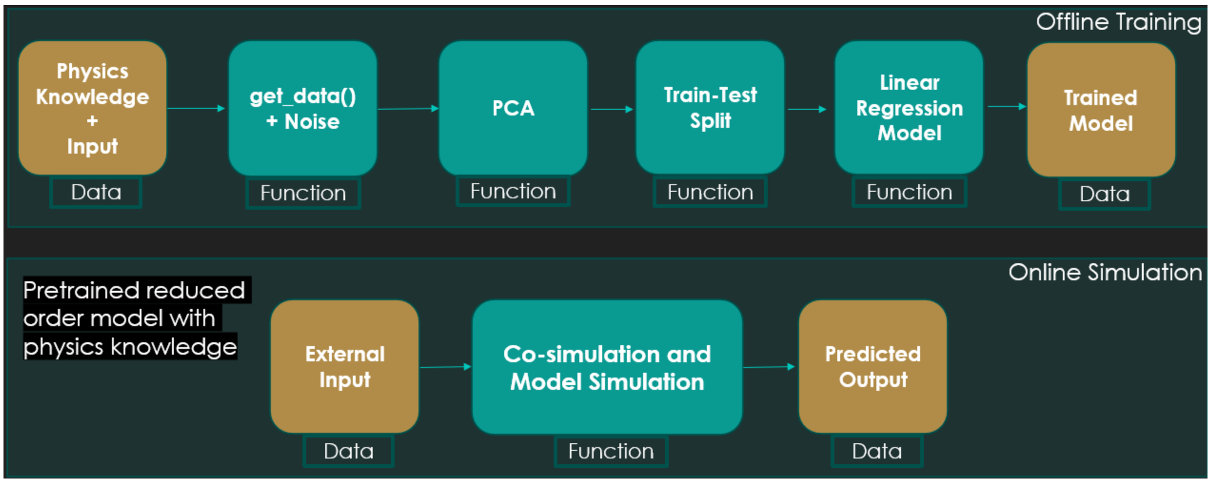
- Offline Training Functions:
    - get_data:
        - input = [“noise_type”, “dimensions”, “num_of_trials”, “prediction_feature”]
        - ouput = [“X”, “Y”]
        - function name = “get_data”
    - pca:
        - input = [“X”, “pca”]
        - output = xform_input_data
        - function name = “pca_function”
    - split:
        - input = [xform_input_data]
        - output = [“X_train”, “X_test”, “Y_train”, “Y_test”]
        - function name = “split_data”
    - train:
        - inputs = [“X_train”, “Y_train”, “parameters”]

        - outputs = [“fit_function”]
        - function name = “train”
    - test_spring_model:
        - inputs = [“fit_function”, “X_test”, “Y_test”]
        - outputs = [“Y_pred”]
        - function name = “test_spring_model”

- Online Simulation Function:
    - simulation [input = external_input_data, output = simulated_prediction, import_fmu = True)



*[**Image 12**: Reduced pipeline for the Builder integration]*



*[**Image 13**: Complete pipeline for both offline and online integration]*

8. (Next Stage) Deployment of Integrated ALPR Model on IOS device:

This stage is to test and demonstrate the working of an integrated kedro pipeline. The Automatic Number Plate Recognition model will be split into an online training use case and an offline simulation use case for a real-life Hybrid AI application.

- First, a pretrained ALPR model will be replicated on FMU.
- Then the FMU generated will be simulated using Kedro.
    1. Model input and output dataflow can be monitored.
    2. Performance of the Kedro pipeline can be tracked.
- Deployment:
    1. The final kedro application will be deployed and tested on an IOS device.

References

[1]: Modelon. (2023, May 18). *FMI standard: Understand the two types of functional mock-up units - CS V. me*. Modelon. https://modelon.com/blog/fmi-functional-mock-up-unit-types/

[2]: Wikimedia Foundation. (2023, December 21). *Hooke's law*. Wikipedia. https://en.wikipedia.org/wiki/Hooke%27s_law#:~:text=In%20physics%2C%20Hooke's%20law%20is,small%20compared%20to%20the%20total