A Tool for the Implementation of Open Neural Network Exchange Models in Functional Mockup Units

Michele Urbani[1] Michele Bolognese[1] Luca Pratticò[1] Matteo Testi[1]

[1]Hydrogen Technologies and Resilient Energy Systems Unit, Sustainable Energy Centre, Fondazione Bruno Kessler, Via Sommarive 18, 38123, Trento, Italy, {murbani,mbolognese,prattico,testi}@fbk.eu

Abstract

The Functional Mock-up Interface (FMI) standard is a flagship in the co-simulation and model exchange domain. However, the integration of graph-based computational models—particularly neural networks—into Functional Mock-up Units (FMUs) has remained a technical challenge due to interoperability and platform-specific limitations. To address this, we propose ONNX2FMU, a command-line Python tool that facilitates the deployment of Open Neural Network Exchange (ONNX) models into FMUs. According to FMI's good practices, ONNX2FMU generates C source code to wrap ONNX models in Functional Mockup Units, supports FMI versions 2.0 and 3.0, and provides multi-platform compilation capabilities. The tool simplifies the mapping process between model description and ONNX model inputs and outputs via JSON files, ensuring accessibility and flexibility. This paper presents the tool architecture and methodology and showcases its applicability through illustrative examples, including a reduced-order model powered by a recurrent neural network.

Keywords: ONNX, FMI, FMU, ML, AI

1 Introduction

Machine learning (ML) and deep learning (DL) have gained momentum in engineering and science in recent years. There are countless applications of inferential models in engineering: For example, in engineering and manufacturing (Malik et al. 2018), process system engineering (Lee, Shin, and Realff 2018), chemical engineering (Schweidtmann et al. 2021), smart energy systems (Lund et al. 2017), and fluid mechanics (Brunton, Noack, and Koumoutsakos 2020). Better and more efficient computational infrastructures, together with a never-seen-before amount of data, call for the exploitation of these sources of information in simulations (Rueden et al. 2020; Bertsch, Jarmolowitz, and Lenord 2024). However, incorporating ML models into FMUs requires specialised tooling. The existence of several ML frameworks, each with its own scope and peculiarities, advocates for a standard model exchange format. The Open Neural Network Exchange (ONNX) format (*Open Neural Network Exchange* 2025) enables one to describe an ML model through a set of standard operators and ships with its runtime. Several ML frameworks implement their methods to export a model to the ONNX format, thus partly bridging the gap of model interoperability. Therefore, the challenge is developing a tool to write the C code that maps the model's input and outputs to the FMI's C API specifications. Software licensing must guarantee the usability of the tool for many purposes to foster the widespread adoption of inferential models in simulations.

Commercial tools that can wrap ML models into FMUs already exist. Simcenter ROM (*Simcenter Reduced Order Modeling software* 2025) is a software for developing Reduced Order Models (ROM) that presents the FMU export feature, which allows a user to wrap an ML-based ROM in an FMU. Since ML is simply a model order reduction technique (among a few others) in Simcenter ROM, the choice of the ML framework is arbitrary and, to the eye of the user, it is irrelevant as far as the model's accuracy is satisfactory. The generation of an FMU is, therefore, purely functional and does not allow the user any customisation. SimuLink (*Simulink - Simulation and Model-Based Design* 2025) is a commercial software that exports simulation models as FMUs by generating the C code from a set of proprietary primitives. Neural networks, a type of ML model, can be included in a Simulink simulation workflow and shipped in an FMU. The software suite (*NEURECO: Automatic Parsimonious Neural Networks Generation* 2025) presents itself as the *automatic parsimonious neural networks factory*. It can produce neural network-based models and export them to C code, ONNX, and FMUs.

Software licensing is a key feature when selecting a tool. The availability of open-source, open-access software is always valuable to the community. Most open-source software is hosted on platforms like GitHub and GitLab, where we carried out a structured search for ML to FMU wrapping tools. The keywords used for the search were "ONNX" and "FMU", which returned only one relevant result on the GitHub platform. The search results are confirmed by a check of the "Tools" section of the FMI website. MLFMU (*mlfmu* 2025) is the only open-source, open-access tool that enables wrapping ML models to FMUs, and a team of contributors from DNV develops it. The MLFMU's workflow foresees the possibility of exporting an ONNX model with only one input and one output array, to which the physical model quantities need

to be mapped. Input and output mapping is declared using a JSON file, which plays the role of a user interface. The core functionality of MLFMU is the generation of the FMI model description and the C++ code generator to link the ONNX runtime to the FMU. For the generation of C++ code, MLFMU rely on another open source project called "cppfmu". Conan is used as a package manager.

Proposing a new tool for integrating ML models in FMU co-simulation is important because it enables the use of graph-based computational models, e.g., ML models and neural networks. Furthermore, the proposed tool would be a valuable open-access, open-source resource to the Modelica community.

We introduce ONNX2FMU, a command-line interface (CLI)-based Python tool that generates C source code for embedding ONNX models into FMUs, which can be compiled on any target platform. The tool bridges the gap between ML model deployment and FMI-compliant simulation, offering an efficient and user-friendly pipeline. Compared to the existing alternatives, ONNX2FMU:

- Directly generates C source code, improving performance and control over the generation process. Although the use of the C programming language is not mandatory, we preferred to stick with the FMI Standard's indication that the "FMI API is written in C, not C++, to avoid problems with compiler and linker dependent behaviour, and to enable the use of FMUs on embedded systems" (Modelica Association 2025).
- In addition to the FMI 2.0, it supports FMI 3.0, including extended data types.
- Handles models with multiple inputs and outputs mapped to the FMU model according to an arbitrary naming convention.

Generating C code directly exposes the user to the source code in a single file where FMI "get" and "set" functions are declared together with the link to the ONNX runtime. ONNX2FMU is designed for developers who want to bootstrap the creation of FMUs to deliver graph-based computational models, data scientists and researchers operating in model order reduction, and DevOps who need to automate the creation of pipelines of tools. Being distributed as open-source, the tool is highly customizable, and it allows the separation of the C code generation stage and the compilation stage, thus enabling the inspection and modification of the FMU source code. The tool comes with a set of unit tests and a pipeline for continuous integration that guarantees the integrity of the code and its cross-platform compatibility.

An analysis of requirements for the proposed tool and the adopted methodology for creating the tool is outlined in Section 2. Section 3 presents the results and shows two examples where ONNX2FMU is applied. Conclusions close the paper in Section 4.

2 Methodology

Requirements analysis The inspiring principle for the creation of ONNX2FMU was the simplicity of use. This means relieving the user from making many decisions during the tool usage, reducing the number of steps to the final result, and designing a clear interface for mapping ONNX model inputs and outputs to the FMI model description.

FMI standard compliance. The FMI Standard is now in its third major release (Junghanns et al. 2021), the FMI 3.0 version, which differs from the FMI 2.0 version in several aspects. In FMI 3.0, the *scheduled execution* interface type has been added to the *model exchange* and *co-simulation* types, allowing the activation of individual model partitions by an external scheduler. Model partitioning capabilities for concurrency on single-core CPUs are not of interest for executing ML models in a simulation; therefore, we excluded the possibility of generating the scheduled execution interface type. In general, the execution of ML models does not require the FMU to be equipped with a solver. A straightforward choice is hence to implement the model exchange interface type. Since a few modelling environments, e.g., Modelon Impact, allow us to import only co-simulation FMUs, the usability range of a tool for ML model encapsulation is significantly widened by implementing the co-simulation interface. The co-simulation interface brings the burden of having a solver within it that remains unused.

In the first version of ONNX2FMU, we focus on the FMI 2.0 and 3.0 Standard versions. The FMI 2.0 is the industry standard, supported by more than 90% of the tools published on the FMI Standard website[1]. Supporting the FMI 3.0 contributes to the state of the art in ML model implementation in the FMI standard. The two versions of the standard present meaningful differences in the data types that can be specified in the model description: The FMI 2.0 supports only four data types, i.e., float, integer, string, and boolean, whereas the FMI 3.0 extends these data types to support a broader spectrum of data precision and signature. Relevant data types to the scope of this project are the `float` type for the FMI 2.0 and the `Float32` and `Float64` types for the FMI 3.0. All other data types are considered according to the ONNX specifications discussed in the next paragraph. The variability and causality of variables are restricted to the combinations allowed by the standard. If not specified by the user, the causality of model entries is determined by their connections in the ONNX graph model, which divides entries into inputs and outputs. The available variability options for input and output causalities are "discrete" and "continuous". The FMU developer should carefully set the appropriate variability according to the behaviour of the model to be encapsulated; in the co-simulation mode, the capability flag `providesIntermediateUpdate` should be set accordingly. The combinations of platforms and operating

[1] https://fmi-standard.org/tools/

DOI 10.3384/ecp218645

systems in the following are generally supported:

- win32, win64, linux32, linux64, and aarch64-darwin for the FMI 2.0;
- x86_64-darwin, aarch64-darwin, x86-linux, x86_64-linux, aarch64-linux, x86-windows, x86_64-windows, and aarch64-windows for the FMI 3.0.

ONNX standard. The Open Neural Network Exchange (*Open Neural Network Exchange* 2025) is a standard for the interoperability and exchange of ML models. It defines a set of operators typically used in ML and DL models and a common format for moving models through different ML frameworks, tools, runtimes, and compilers.

Deploying an ML model to production requires replicating part of the ecosystem used to train the model. We refer to this ecosystem using the word *runtime*, that is, the code that enables running inference (and training) on a target hardware platform. In addition to guaranteeing interoperability, converting a model to the ONNX format allows developers to use optimised runtime libraries from different providers, see, e.g., those listed in (*Open Neural Network Exchange* 2025). A runtime can be defined in any language suitable for the production application, C, Java, Python, JavaScript, C#, WebAssembly, or ARM. We chose the ONNX runtime among the many available options because of its licensing, the large community of contributors to the project, and the availability of C code examples that allowed us to bootstrap the development of the tool.

Data types conversions. ML models require support for floating point numbers in single or double precision, and eventually, they may include other data types. An ML model supporting single-precision floating point numbers requires casting double-precision inputs/outputs, with a potential loss of accuracy of the model. The model data type dictates how the tool must manage data types internally: If a model is single-precision, a double-precision float must be converted before and after inference; otherwise, no casting operations are required. The same behaviour is adopted for integers.

Inputs, outputs, and naming conventions. A generic ML model can have multiple inputs and output nodes, whose elements must be mapped to the FMI model description. The FMI 2.0 version foresees only scalar variables in the model description. Therefore, arrays must be flattened, and a naming convention is required to map scalars to their original position. A logical choice seems to use the array name suffixed with the element indexes separated by a specific character. Conversely, the FMI 3.0 allows both the declaration of scalar and array variables in the model description. Declaring array variables produces a more compact and comprehensible model description, but poses a hurdle to mapping variables to their physical meaning.

User experience. The interaction with the tool should consist of two subsequent stages. The mapping of the ONNX model entries to the FMI model description occurs first. During this stage, the user can define the variability associated with ONNX model arrays, if different from `continuous`, and, optionally, a description of each element of an array. The latter will likely be shown by the graphical user interface (GUI) of the importer software, thus easing the connection of the model with other components of the simulation.

Once a model mapping scheme and an ONNX model are available, an FMU model can be generated. There are many possible approaches to FMU generation, ranging from a GUI to Python scripting. A GUI would consist of only a few buttons to trigger the generation process, and it would be more effective if it integrated the model mapping declaration. Interaction via CLI is simple to develop and effective in integrating within a pipeline of operations. FMU generation via Python scripting is always possible by calling the tool's functions directly.

3 Results

The result of this research is ONNX2FMU. The design choices are explained in the following with examples.

3.1 Getting started

ONNX2FMUis written in Python to ease code usability, interaction with ONNX models, and to exploit a suite of available open source tools. A Python interpreter is required, and ONNX2FMU has been tested for versions greater than or equal to 3.10. ONNX2FMU is distributed as a Python package available in PyPI (*PyPI, Python Package Index* 2025), and can be installed with

```
pip install onnx2fmu
```

from the command line. To generate FMUs, ONNX2FMU requires a C compiler for the target platform to be installed on the machine and added to the system `PATH` variable.

3.2 FMU model mapping

ONNX2FMU can handle ONNX models presenting multiple input and output nodes. The description of entry variables is organised hierarchically, including global model information and input/output node descriptions. The JSON file format has been chosen for model mapping due to its suitability in representing hierarchically organised information. In the input and output sections, nodes are identified using the same name used in the ONNX model; this allows ONNX2FMU to map the information correctly. A user can specify the variability of each node, which is set to continuous by default. Node variability should be handled carefully because it might cause sequence-based models to return wrong results when they are not called with the right time step. Array elements can be enriched with a description, which is shown in the FMI

model description and might be shown by the importer UI. Descriptions of entries help the user to link them with their physical meaning. If the list of descriptions is provided, it must match the elements in the array; otherwise, a warning is returned, and no description will be present for the elements of that array. The array notation provided by the FMI 3.0 Standard is left out of the scope of this work because it makes it difficult to connect array entries with their physical meaning. However, the authors acknowledge that array notation could be useful when an ONNX model presents several inputs and outputs; for this reason, it will be addressed in a future release of ONNX2FMU.

3.3 FMU build

The FMU build workflow starts with the target model to encapsulate and the model mapping JSON file. If not specified, the target platform is that of the system where the build process is carried out. A template of files and folders to be populated with ONNX model information was developed. The Python interpreter generates the C code by creating a copy of the FMU template and filling it using Jinja2 (*Jinja Project* 2025), a template engine that populates special placeholders in text files with context variables and renders the final document. The CMake build system (*CMake: A Powerful Software Build System* 2025) is used to build the project, download the latest ONNX runtime version, compile the library, and zip it to an FMI-compliant archive. For separation of concerns, ONNX2FMU takes care of template filling and ONNX model information retrieval using Python, whereas compilation issues and packaging are handled by CMake, which natively supports the build and compilation toolchain.

The Python CLI is realised using *typer* (FastAPI 2025), a Python package that creates simple command line applications with documentation, autocompletion, and typing support.

Validation and testing are the last two steps of ONNX2FMU's workflow, which is depicted in Figure 1. The automated validation of the FMU is made possible by the FMPy (*FMPy* 2025) package. Testing a generated FMU can hardly be automated because of the case-specific nature of ML models, which requires appropriate data.

3.4 Examples

Use cases are presented to show the functionalities and the potential of ONNX2FMU in the following paragraphs.

A simple model. To showcase the flexibility of ONNX2FMU, we present how a generic graph-based model can be wrapped in an FMU. The simple model takes three scalar inputs x_1, x_2, x_3 and returns a vector output y. In the model, two calculations occur: $x_4 = x_1 + x_2$ and $x_5 = x_2 - x_3$, and the results are concatenated in $y = [x_4, x_5]$ and returned as output. A graphical representation of the model is depicted in Figure 2, which was obtained using the Netron web app[2]. A Python pseudo-code example for creating the model using the PyTorch framework is reported in Listing 1.

Listing 1. Python code for the creation of a simple graph-based computational model using PyTorch.

```
class ExampleModel(nn.Module):

  def __init__(self):
    super(ExampleModel, self).__init__()

  def forward(self, x1, x2, x3):
    x4 = x1 + x2
    x5 = x2 - x3
    x = torch.cat([x4, x5], dim=0)
    return x
```

Exporting the model is easy in PyTorch as in other ML frameworks and usually, it can be done in a few lines of code. Models can be exported to the ONNX format with or without weights; ONNX aims to define the model structure primarily through its set of primitives. ONNX models are not necessarily shipped with weights either because they do not need them, as in the case of the model above, or because weights might be application-specific.

Once the ONNX model has been exported, inputs and outputs are mapped to the FMU model description. This is achieved through the JSON schema in Listing 2.

Listing 2. The JSON mapping between inputs and outputs of the simple model.

```
{
"name": "simple-model",
"description": "Example description.",
"FMIVersion": "2.0",
"input": [
    {"name": "x_1",
     "description": "A scalar input"},
    {"name": "x_2",
     "description": "A scalar input"},
    {"name": "x_3",
     "description": "A scalar input"}],
"output": [
    {"name": "y",
     "description": "The output array",
     "names": ["output1", "output2"]}]
}
```

Notice that the output array schema presents the `"names"` item, which allows users to describe each array entry in the FMI model description. Finally, FMU generation is triggered via the command line by providing the ONNX model and description as follows

```
onnx2fmu build \\
    model.onnx \\
    modelDescription.json
```

Time series prediction. Predicting temperature variations in a solid exposed to a heat source is a common engineering problem. A partial differential equation (PDE)-based model (Bergman 2011) can describe a temperature field variation in, e.g., an ideal 2D solid of interest. The

[2] https://netron.app/

DOI 10.3384/ecp218645

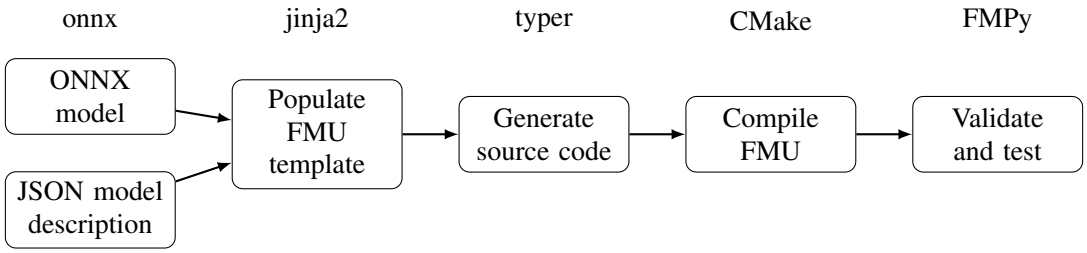

Figure 1. The workflow to embed an ONNX model in an FMU with ONNX2FMU.

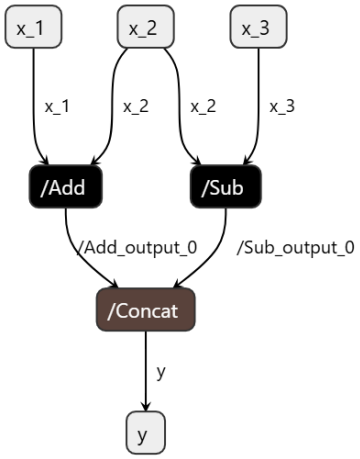

Figure 2. A simple graph-based model for summation and subtraction of inputs.

temperature field equation is

$$\frac{\partial T}{\partial t} = \alpha \left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right) + T_0, \tag{1}$$

where T is the temperature, t is the time, α is the thermal conductivity of the material, and x and y are the coordinates of a point in the solid. Equation 1 is discretised in time and space and resolved by a custom Python model that is reported in the ONNX2FMU documentation[3]. In the following, we define $\alpha = 0.9$, an initial temperature $T_0 = 100°\text{C}$, and $x, y \in [0, 1]$.

In some engineering applications, we may be interested in predicting the temperature variation at a point of the solid, e.g., $x_a = 0.5, y_a = 0.5$. A surrogate model of the studied solid would avoid resolving the full-scale model to know the temperature only in (x_a, y_a), thus saving a meaningful computational effort. A simple way to create a surrogate model for predicting temperature variations is to train a neural network using a time series of heat source variation $u(t)$ as inputs and temperature variations $y(t)$ as outputs. We use a Recurrent Neural Network (RNN) (Bengio, Goodfellow, Courville, et al. 2017) to extrapolate the temperature in (x_a, y_a). A dataset of temperature trajectories $y(t)$ in (x_a, y_a) is generated resolving (1) with the boundary condition

$$u(t) = (-1)^j 20j \sin(t/8 + j\pi/8) + T_0,$$

applied on the side with $x = 0.0$, where j is the index of a trial input profile, for $t \in [0, 25]$ seconds. An example of the applied temperature profile $u(t)$ and temperature profile $y(t)$ in (x_a, y_a) obtained by simulating the system using the custom Python script is depicted in Figure 3.

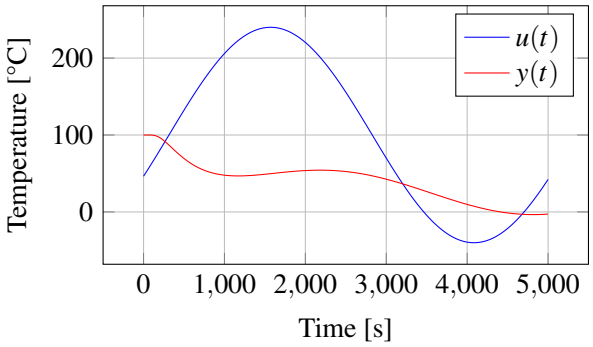

Figure 3. Input temperature profile and the temperature in the centre of the plate.

The model was trained on a dataset with $M = 40$ trajectories, each with 5000 time points sampled at a 1/200 s^{-1} frequency. Data were organised in tuples $(\mathbf{x}, y)$, where $\mathbf{x}$ was a sequence with $R = 200$ consecutive time points, and y was the $(R+1)$-th time point. The mean squared error

$$MSE = \frac{1}{N} \sum_{i}^{N} (y_i - \hat{y}_i)^2$$

measures the predictive power of the RNN, where $\hat{y}_i$ is the value predicted by the RNN, i is the sample index, and N is the number of samples. The efficiency of the RNN model improves if normalised data are used for training. However, it is impractical to deal with normalisation issues in a simulation environment; it would be more useful to provide unscaled data directly with the model. This issue can be easily resolved by adding normalisation and denormalisation steps to the network as shown in Listing 3.

Listing 3. Python code of the RNN model.

```
class HeatRNN(nn.Module):
    def __init__(self, input_size,
        hidden_size, output_size,
        num_layers, norm_params):
```

[3] github.com/HyRES-FBK/onnx2fmu/

```
        super(HeatRNN, self).__init__()
        x_min, x_max, y_min, y_max =
            norm_params
        self.register_buffer("x_min", torch
            .tensor(x_min))
        self.register_buffer("x_max", torch
            .tensor(x_max))
        self.register_buffer("y_min", torch
            .tensor(y_min))
        self.register_buffer("y_max", torch
            .tensor(y_max))
        self.rnn = nn.RNN(input_size,
            hidden_size, num_layers,
            batch_first=True)
        self.fc = nn.Linear(hidden_size,
            output_size)

    def forward(self, x, h=None):
        x = (x - self.x_min) / (self.x_max
            - self.x_min)
        out, _ = self.rnn(x, h)
        out = self.fc(out)
        out = out * (self.y_max - self.
            y_min) + self.y_min
        return out
```

In the code Listing 3, the normalisation parameters are passed as inputs when the network is initialised and stored as *buffers*, that is, they will be part of the network state when exported to the ONNX format, and they will remain untouched by PyTorch optimization algorithms during training. The RNN is initialised with 2 hidden layers, an input size of R, and a 64-unit hidden layer. The RNN output is further processed by a linear layer that maps the RNN output to the output scalar. Figure 4 shows the behaviour of the average value of *MSE* calculated over the training and validation data sets with the number of training epochs. The effectiveness of the trained model was satisfactory for this example after 1000 epochs using the Adam optimiser with a learning rate of 10^{-5}.

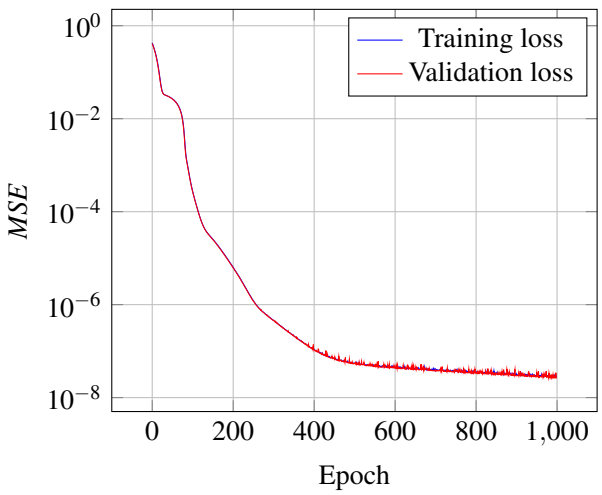

Figure 4. The *MSE* behavior through training epochs.

Finally, the trained RNN model is exported in ONNX format and encapsulated in an FMU for testing. The FMU was tested using FMPy (*FMPy* 2025), a Python toolkit for FMU validation and simulation. An example of the prediction produced by the RNN model is shown in Figure 5. In FMPy, a model can be simulated by providing inputs

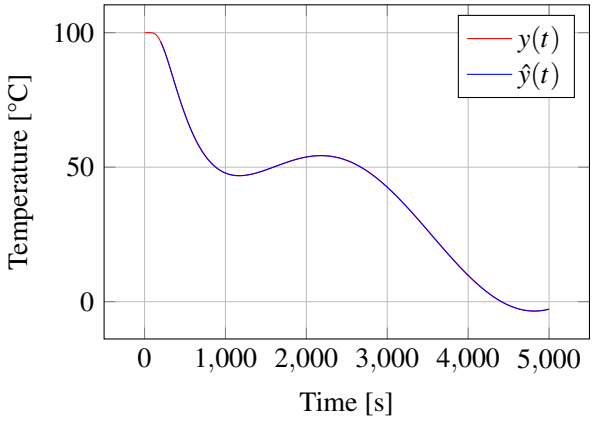

Figure 5. An example of the predicted and analytically obtained temperature profiles in (x_a, y_a).

directly from a text (CSV) file, which simplifies simulation setup when the RNN model presents $R = 200$ inputs. The recursion in the RNN model is carried out by providing R times the number of model variable values at each time step. However, we acknowledge that:

- the length of the input array, $R = 200$, makes the initialisation of such a model impractical and time-consuming in a GUI-based simulator, and
- the repetition of values along rows is memory-wise inefficient.

This aspect will be improved in the next major release of ONNX2FMUby implementing feedback loops declared in the model description.

In this simple example, we showed how a powerful model such as an RNN can be wrapped in an FMU with a few lines of code. The code of the example is released in GitHub[4].

4 Concluding remarks

We presented ONNX2FMU, a tool for converting ONNX models into FMI-compliant FMUs. The tool supports multiple FMI versions and cross-platform compilation and simplifies the deployment of ML models in simulation environments. Future work includes optimising runtime performance, integrating a GUI to favour a broader tool adoption, implementing FMI 3.0 array notation, and develop support for stateful models using feedback loops directly through the model declaration file.

[4] github.com/HyRES-FBK/onnx2fmu/blob/master/tests/example3/generate-example-model.ipynb

DOI 10.3384/ecp218645

References

Bengio, Yoshua, Ian Goodfellow, Aaron Courville, et al. (2017). *Deep learning*. Vol. 1. MIT press Cambridge, MA, USA.

Bergman, Theodore L (2011). *Fundamentals of heat and mass transfer*. John Wiley & Sons.

Bertsch, Christian, Fabian Jarmolowitz, and Oliver Lenord (2024). "Using the Functional Mock-up Interface (FMI) for AI applications". In: *Center for Model-Based Cyber-Physical Product Development* 33.18. DOI: 10.13140/RG.2.2.35730.47043.

Brunton, Steven L., Bernd R. Noack, and Petros Koumoutsakos (2020). "Machine Learning for Fluid Mechanics". In: *Annual Review of Fluid Mechanics* 52.Volume 52, 2020, pp. 477–508. ISSN: 1545-4479. DOI: 10.1146/annurev-fluid-010719-060214.

CMake: A Powerful Software Build System (2025). URL: https://cmake.org/ (visited on 2025-03-26).

FastAPI (2025). *Typer: A library for building CLI applications*. URL: https://typer.tiangolo.com/. (accessed: 28.04.2025).

FMPy (2025). URL: https://fmpy.readthedocs.io/en/latest/ (visited on 2025-03-27).

Jinja Project (2025). URL: https://jinja.palletsprojects.com/en/stable/ (visited on 2025-03-26).

Junghanns, Andreas et al. (2021). "The functional mock-up interface 3.0-new features enabling new applications". In: *Modelica conferences*, pp. 17–26. DOI: 10.3384/ecp2118117.

Lee, Jay H., Joohyun Shin, and Matthew J. Realff (2018). "Machine learning: Overview of the recent progresses and implications for the process systems engineering field". In: *Computers & Chemical Engineering* 114. FOCAPO/CPC 2017, pp. 111–121. ISSN: 0098-1354. DOI: 10.1016/j.compchemeng.2017.10.008.

Lund, Henrik et al. (2017). "Smart energy and smart energy systems". In: *Energy* 137, pp. 556–565. ISSN: 0360-5442. DOI: 10.1016/j.energy.2017.05.123.

Malik, Hasmat et al. (2018). "Applications of artificial intelligence techniques in engineering". In: *Sigma* 1. DOI: https://doi.org/10.1007/s10845-021-01771-6.

mlfmu (2025). URL: https://github.com/dnv-opensource/mlfmu (visited on 2025-04-06).

Modelica Association (2025). *Functional Mock-up Interface Specification*. URL: https://fmi-standard.org/docs/3.0/. (accessed: 24.03.2025).

NEURECO: Automatic Parsimonious Neural Networks Generation (2025). URL: https://www.adagos.com/neureco (visited on 2025-03-27).

Open Neural Network Exchange (2025). URL: https://onnx.ai/ (visited on 2025-04-06).

PyPI, Python Package Index (2025). URL: https://pypi.org/ (visited on 2025-03-26).

Rueden, Laura von et al. (2020). "Combining Machine Learning and Simulation to a Hybrid Modelling Approach: Current and Future Directions". In: *Advances in Intelligent Data Analysis XVIII*. Ed. by Michael R. Berthold, Ad Feelders, and Georg Krempl. Cham: Springer International Publishing, pp. 548–560. ISBN: 978-3-030-44584-3.

Schweidtmann, Artur M. et al. (2021). "Machine Learning in Chemical Engineering: A Perspective". In: *Chemie Ingenieur Technik* 93.12, pp. 2029–2039. DOI: 10.1002/cite.202100083.

Simcenter Reduced Order Modeling software (2025). URL: https://plm.sw.siemens.com/en-US/simcenter/integration-solutions/reduced-order-modeling/ (visited on 2025-03-27).

Simulink - Simulation and Model-Based Design (2025). URL: https://www.mathworks.com/products/simulink.html (visited on 2025-03-27).