An Interprocess Communication based Integration of AI User Materials into LS-DYNA

Joachim Sprave[1], Tobias Erhart[2], André Haufe[2]

[1]Mercedes-Benz AG, 059-L423, D-71032 Böblingen, Germany
[2]DYNAmore GmbH, Industriestrasse 2, D-70565 Stuttgart, Germany

Machine Learning (ML) driven material models have been investigated for some time now. The respective ML models can be trained outside of the Finite Element solvers by means of given strain paths and corresponding stress results which have either been recorded from simulations, drawn from distributions, or even measured from hardware tests. The trained models can be easily evaluated regarding their performance based on an-other set of strain paths with results that have not been presented to the model during training. When a model has reached a promising prediction accuracy on the validation data, the natural next step is to test it in a finite element simulation by integrating the trained model as a user material. Unfortunately, coupling ML models trained by established AI frameworks, such as TensorFlow or PyTorch, is not a trivial task. Solvers are typically stand-alone programs written in a compiled language such as Fortran or C/C++. For LS-DYNA, there are three canonical ways to use trained ML material models as user materials: Reimplementation as a user material subroutine (UMAT) in Fortran or C/C++, linking LS-DYNA with an existing framework, and interprocess communication (IPC) with an external ML model. In this work we present an IPC UMAT for LS-DYNA on Unix-like systems. A dummy UMAT is provided that handles the communication, as well as boilerplate code in Python that serves as a wrapper to call a trained model for each integration point or batch of integration points, mimicking the logic of simple and vectorized UMATs in Fortran. Examples are given how to use Keras, PyTorch, and Scikit-learn models.

1 Material Models using Machine Learning Models

Today, material models for state-of-the-art finite element solvers are built on a solid base of decades of research in continuum mechanics. Most material models and their parameters are being developed in a highly optimized, mostly standardized process, with standardized hardware tests, standardized measurement methods, and standardized software tools. Still, this takes time and only works for materials that are not too far away from known materials with respect to their mechanical properties and deformation behavior. The final result of this calibration process is a piece of code and respective parameters. For LS-DYNA, this would be a Fortran subroutine that implements a function which, in the simplest case for incremental metallic materials, computes from a stress tensor, a strain tensor, and some history variables an updated stress tensor and updated history variables. Formally this can be written as $f : \mathbb{R}^{12+\text{nhv}} \to \mathbb{R}^{6+\text{nhv}}$ where nvh is the number of history variables. Note that in LS-DYNA, effective plastic strain is treated separately from other history variables.

Since material models are just multi-variate real functions, they can be as well treated as black boxes and learned by data-driven algorithms. Although it is beyond the scope of this publication to discuss the details of data-driven constitutive models, two reasons to create these models are mentioned here: Firstly, when a new material enters the market, often there are no established and trusted constitutive models available. Deriving such a model from hardware tests only can deliver a first and rough material model. Hence, often a full understanding and hence the representation of the mechanical properties and behavior of this material is not possible. Secondly, data-driven models based on Neural Networks (NN) are suited for specialized hardware as for instance GPUs and can be faster in execution during simulations. If just speed is the objective, AI material models may be trained from data generated using existing constitutive models, e.g. [4]. For new materials, no existing material models could be available to generate such data. In this case, the training data must be extracted from hardware tests. This is especially challenging because the most significant output of material models, namely the stresses, cannot be measured from tests directly. An approach to overcome this problem can be found in [3].

2 Coupling LS-DYNA with ML Frameworks

For simple ML models, reimplementation of the models as user defined materials in a finite element code can be an option. Feedforward neural networks, for example, consist of matrix multiplications and non-linear activations functions only. When the weights are given as result of a training outside of LS-DYNA, an inference model that serves as a user material can be easily implemented. But when it comes to more sophisticated ML models, such as convolutional neural networks, recurrent neural networks, or the most recent achievement in sequence modelling, so called transformers, the models used in the training can become very large and very complex. In this case reimplementation is time-consuming and error-prone. It is still hard to beat performance-wise, so this is probably the way to go for models which finally go into production releases. However, for developing and testing, and optimizing models, a more flexible workflow is required.

Lightweight Linkable Libraries

Technically, the most obvious way to couple LS-DYNA with an ML framework or library is linking the ML code to the LS-DYNA binaries, either statically or as modules. This requires the ML code to be Fortran-friendly, meaning that it is already written in Fortran, e.g. [2, 5] or in another compiled language such as C or C++, e.g. [6] that can be easily linked to Fortran code. In this case one can simply call the trained models from an otherwise empty user material subroutine. Hence the ML model is part of the solver, so one receives the full performance of the framework. On the other hand, the ML code shares the address space with the solver, meaning that the correctness and stability of the simulation depends on the combined binaries. Still, if the trained models accuracy is sufficient, this approach can deliver a performance compared to individually coded models.

Bridging to State-Of-The-Art Machine Leaning Frameworks

Instead of relatively light-weight libraries which can be linked to LS-DYNA directly, it is also possible to couple full-featured state-of-the-art frameworks by means of bridging. Bridging means that there is some code that translates the ML frameworks API to Fortran calls. Examples for bridges are the Fortran-Keras-Bridge (FKB) [8] for Keras/TensorFlow, or PyTorch Fortran bindings [1] for PyTorch. These bridges require a careful setup and a matching version of the ML framework. Linking against a code monster like LS-DYNA also has its pitfalls. But once the setup works, one can use most – in case of FKB – or – with the PyTorch Fortran Bindings – all of the layer types available in the framework. The models can be trained with full support of tools that have been developed in the ML domain.

Interprocess communication (IPC)

A dummy UMAT forks a process that communicates with a process written in a scripting language such as Python. Within the Python script, arbitrary frameworks can be used, including, but not restricted to, PyTorch, TensorFlow, and Scikit-Learn. One can even write traditional material models based on continuum mechanics and test them without recompiling and relinking LS-DYNA. Compared to linking and bridging, this approach is hopelessly bad regarding performance. On the other hand, a trained model as well as a hand-written material model can be made available for LS-DYNA simulations in a couple of minutes.

3 pipespy: An IPC-UMAT Suite for LS-DYNA

In LS-DYNA, user materials are Fortran subroutines which are being linked to the main solver either statically or dynamically. In any case, when the solver runs it calls a user material as a subroutine inside of its address space. In order to use an ML model instead of a classical code based constitutive law, the UMAT serves as a converter and dispatcher. Data received from the LS-DYNA main program, like strain increments and history variables, must be filtered and converted into a data format that can be processed by the linked ML model. The results coming back from the ML model must then be converted into the format expected as output from user materials, i.e. updates stresses and history variables. In case of bridging with an ML framework, LS-DYNA, the

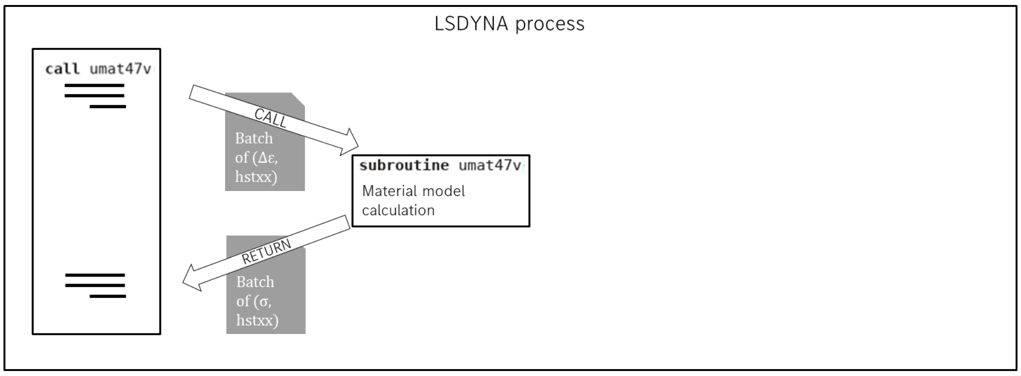


Figure 1: Essential control flow for a user material in LS-DYNA. The main program calls a subroutine (here: umat47v) passing strain increments, the current stress tensor, the history variables, and other information, e.g. time step, material constants etc. for a batch of integration points to the called subroutine. The material model subroutine returns the stress tensors for the next time step and updated history variables.

bridging software, and the ML framework must be linked together into a single executable. While the PyTorch Fortran bindings [1] make use of a PyTorch runtime library, thereby providing the complete functionality of PyTorch, the *Fortran-Keras-Bridge* (FKB) translates Keras trained models to *neural-fortran* [5]. This means that FKB is limited to network architectures and layer types implemented in *neural-fortran*. LS-DYNA and the ML framework share the same address space. In contrast,

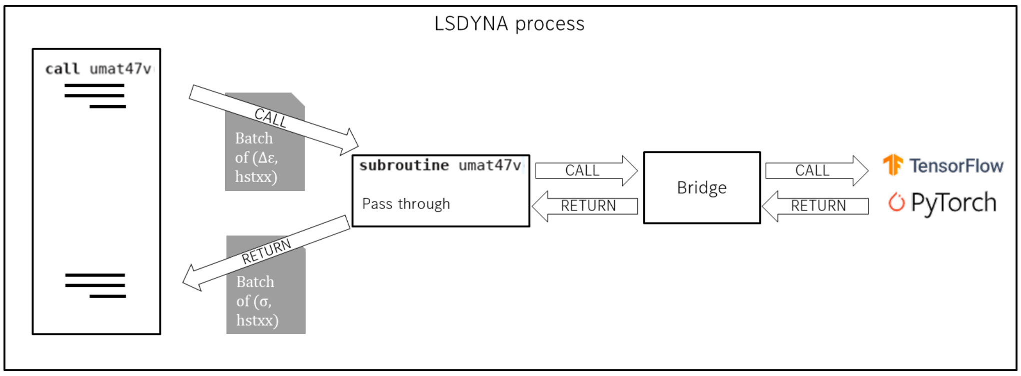


Figure 2: Essential control flow for an ML user material using bridging. After optionally filtering and scaling the data, the material model passes its input data to the bridge. The bridge routine then passes the data to the actual ML material model which has be trained outside of LS-DYNA, and returns the model's response to the user material which again returns it to the main program in a proper format, i.e. as stresses and history variables. Since everything is linked to a common executable program, there is no parallelism involved.

an IPC coupling of LS-DYNA with Python requires only a dummy user material that handles the communcation. When this is available, anything that can be invoked by a Python function can serve as a material model. This means especially that users do not have to worry about compilers, linkers, makefiles etc.

4 LS-DYNA Integration of the IPC Interface

The development of the actual IPC material model presented here was initially started in the public funded project AIAX [7] and has been further development in the project AIMM (Artificial Intelligence for Materials Modelling, see Acknowledgements). As of now, this software is a proof of concept within the project consortium and will not be part of an official release.

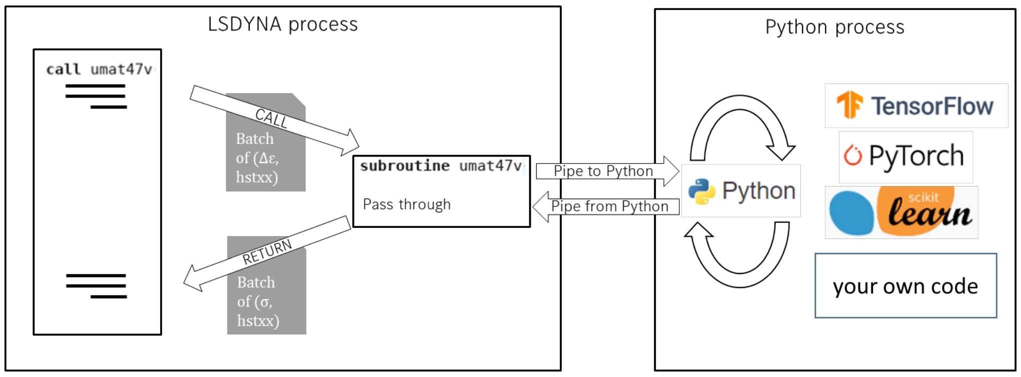


Figure 3: Essential control flow for a user material in LS-DYNA using interprocess communication. After optionally filtering and scaling the data, the user material subroutine writes its inputs into an IPC channel – here: an anonymous pipe – and listens to a corresponding return channel until the results have been delivered. On the other end of the pipes, a separate Python process performs an endless loop listening on the first pipe, processing the data, and writing results to the second pipe.

5 Example: Linear-Elastic Material Model in Python

Boilerplate Code

On the Python side, the pipespy suite provides the pipespy.py code that handles the communication, and a simple template called predict.py as boilerplate code for developing Python material models. The heart of the code in Listing 1 is the `apply` function that instantiates a `DynaIPC` object, and then goes into an endless communication loop. For most Python materials models, only the `predict` function must be replaced by code calculating a material model. In this chapter, an example is shown that implements a linear-elastic material model in Python and makes it available in LS-DYNA by means of interprocess communication.

```python
import sys
import numpy as np
from pipespy import DynaIPC

def predict(Xin):
    # your material model code goes here
    # result = [sigmas, plastic strains, histvars] for solids
    # result = [sigmas, thickness strain, plastic strains, histvars] for shells
    return result

def apply(n_histvar):
    sys.stderr.write(" ------- UMAT IPC Interface (Python): waiting for data\n")
    sys.stderr.write(f" ------- UMAT IPC Interface (Python): {n_histvar} histvars\n")
    ipc = DynaIPC()

    while True:
        X0 = ipc.get_batch()            # get a batch of data from DYNA
        outputs = predict(X0)           # calculate the answer of the material model
        ipc.put_batch(outputs)          # return sigmas to DYNA

def main():
    apply(int(sys.argv[1]))

if __name__ == '__main__':
    main()
```

Listing 1: Boilerplate code for a solid material model using pipespy. For shells, the thickness strain is expected as an additional return value. In the `apply` function, a DynaIPC object is created that handles all the commucation. Besides providing methods to receive and send batches of data, it also provides members `tt`, `ipt`, and `idevels`, holding the current simulation time, integration point, and the list of element IDs from the last call.

MAT001 Python Equivalent

While the interface presented here was developed with ML models in mind, it can also be seen as a playground to develop material models in Python. A simple Hookean material law, for example, consists of just a couple of lines of code in Python, as shown in Listing 2.

```python
emod = 70.0
rnue = 0.30
gmod = emod/(2.*(1.+rnue))
bkmod = emod/(3.*(1.-2.*rnue))

def mat001(umat_input_vector):
    sig123 = umat_input_vector[:3]
    sig456 = umat_input_vector[3:6]
    d123 = umat_input_vector[6:9]
    d456 = umat_input_vector[9:12]
    epsvol= np.sum(d123)/3.
    dpress = 3.0*bkmod*epsvol
    sig123 = sig123 + dpress + 2.0*gmod*(d123-epsvol)
    sig456 = sig456+gmod*d456
    return np.array([sig123,sig456]).flatten()
```

Listing 2: A linear-elastic material model written in Python. It is assumed that matrices and vectors are numpy arrays. For the sake of readability, this code has not been optimized. This can be used instead of a Neural Network to test the interface.

For the sake of efficiency, the IPC user material of pipespy has been implemented as a vectorized UMAT. Hence the data sent to Python is a batch of UMAT inputs for a certain number of integration points. The example in Listing 2 could be further optimized to handle these batches in parallel. To keep this example simple, we just call the `mat001` routine in a loop for each integration point and concatenate the results, as shown in Listing 3.

```python
def predictHook(Xin):
    Yout = []
    for row in Xin:
        Yout.append(predict_single(row))
    zero_plstrain = np.zeros([len(Yout),1])
    return np.hstack([Yout, zero_plstrain])
```

Listing 3: Wrapper function to loop over a batch of integration points. Since pipespy always expects effective plastic strain updates, a column of zeros is added to the result.

Now it is easy to integrate this material model into the boilerplate code in Listing 1. We simply replace the `predict` function by the code from Listings 2 and 3.

Running the MAT001 Example

With the Python part of the example developed in the previous section, there are two missing pieces in the puzzle: Adding an IPC UMAT to a keyword file, and, finally, running a simulation. The keyword file entry is a typical `MAT_USER_DEFINED_MATERIAL_MODELS` card. Density, Poisson's ratio, Young's modulus, as well as bulk and shear moduli are required by LS-DYNA for step size control and contact handling. The material number (mt) calls UMAT 47 that should contain the IPC specific calls. The number of history variables `nhv` must match the respective Python implementation. The width of the array `X0` in line 17 of Listing 1 is $13+\texttt{nhv}$[1].

```
*MAT_USER_DEFINED_MATERIAL_MODELS
$      mid        ro        mt       lmc       nhv    iortho     ibulk        ig
      1027.85000E-6        47        10         0         0         9        10
$    ivect     ifail    itherm    ihyper      ieos
         1         0         0         0         0         0
$     emod      rnue  crv/tble      eppf     eppfr
      70.0       0.3       0.0       0.0       0.0       0.0       0.0       0.0
$     bulk         g
      58.4      26.9       0.0       0.0       0.0       0.0       0.0       0.0
$
*PART
$#                                                                         title
Cantilever
$#     pid     secid       mid     eosid      hgid      grav    adpopt      tmid
         1         1       102         0         0         0         0         0
```

Listing 4: User material card for the IPC UMAT. While the material ID (here: 102) can be chosen arbitrarily by the user, the material model number `mt` is hard-coded to 47 in this proof of concept. `ivect` must be 1 because the IPC UMAT is implemented as a vectorized model. The number of history variable `nhv` must be set according to the amount of data processed by the Python code. Be aware that currently, this is not being checked internally and the user must take care of the correct numbers for `nhv`.

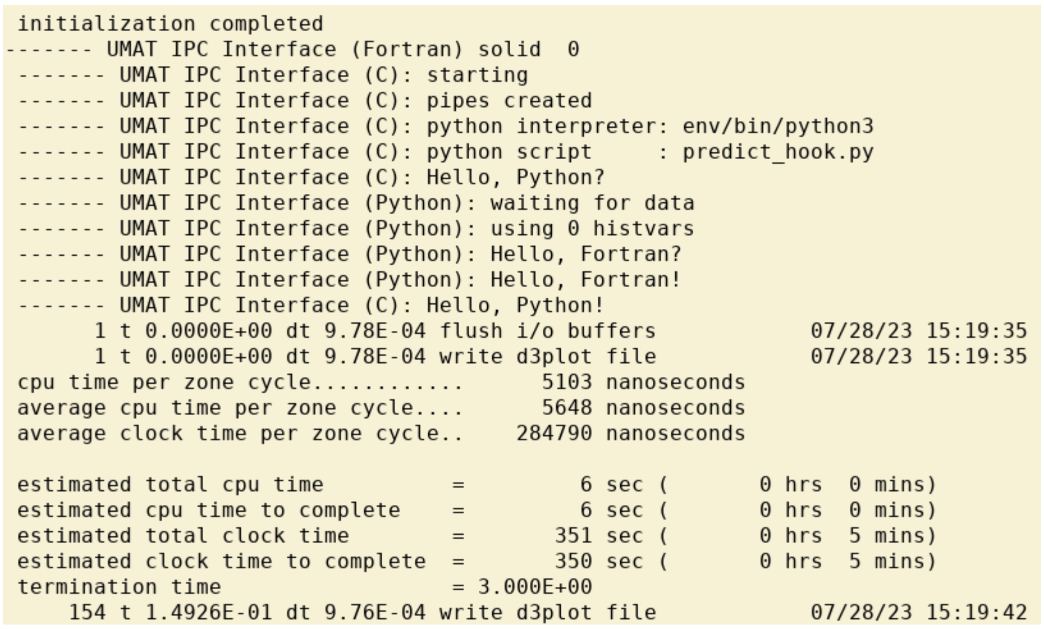
[1] $13 = 6$ stresses, 6 strain increments, 1 plastic strain

Before starting a simulation, two environment variables must be set to inform the LS-DYNA module about the IPC details. The first variable is called `AIAX_PYTHON` and must be set to the path of the Python interpreter to be used. This can be a system-wide install interpreter as well as one from a The second variable `AIAX_SCRIPT` must be set to the path of the Python script that contains the actual model. This script should be based on the the boierplate code in Listing 1.

```
1 AIAX_PYTHON=env/bin/python3
2 export AIAX_PYTHON
3 AIAX_SCRIPT=predict_hook.py
4 export AIAX_SCRIPT
5 mpirun ls-dyna_mpp_d_R12_platformmpi_sharelib i=Cantilever.key
```

Listing 5: Setting the environment variables for pipespy in a bash shell and running LS-DYNA. Note that pipespy is currently implemented as a module for the MPI version of LS-DYNA. For the present proof of concept, a special executable fpr LS-DYNA is required (see line no. 5).

If everything is setup correctly, LS-DYNA reports the successful negotiation between the solver and pipespy, as shown in Figure 4.



```
 initialization completed
------- UMAT IPC Interface (Fortran) solid  0
 ------- UMAT IPC Interface (C): starting
 ------- UMAT IPC Interface (C): pipes created
 ------- UMAT IPC Interface (C): python interpreter: env/bin/python3
 ------- UMAT IPC Interface (C): python script     : predict_hook.py
 ------- UMAT IPC Interface (C): Hello, Python?
 ------- UMAT IPC Interface (Python): waiting for data
 ------- UMAT IPC Interface (Python): using 0 histvars
 ------- UMAT IPC Interface (Python): Hello, Fortran?
 ------- UMAT IPC Interface (Python): Hello, Fortran!
 ------- UMAT IPC Interface (C): Hello, Python!
       1 t 0.0000E+00 dt 9.78E-04 flush i/o buffers            07/28/23 15:19:35
       1 t 0.0000E+00 dt 9.78E-04 write d3plot file            07/28/23 15:19:35
 cpu time per zone cycle...........      5103 nanoseconds
 average cpu time per zone cycle....      5648 nanoseconds
 average clock time per zone cycle..    284790 nanoseconds

 estimated total cpu time          =         6 sec (       0 hrs  0 mins)
 estimated cpu time to complete    =         6 sec (       0 hrs  0 mins)
 estimated total clock time        =       351 sec (       0 hrs  5 mins)
 estimated clock time to complete  =       350 sec (       0 hrs  5 mins)
 termination time                  = 3.000E+00
     154 t 1.4926E-01 dt 9.76E-04 write d3plot file            07/28/23 15:19:42
```

Figure 4: Output from a run of LS-DYNA with a pipespy IPC UMAT.

6 Summary and Outlook

A material model interface for LS-DYNA has been introduced that allows the integration of arbitrary Python code as material models. Although it has been developed for testing data-driven models trained using state-of-the-art AI frameworks, it can also be used to write and test conventional material models in Python, e.g. for educational purposes. While it is clear that this interface cannot compete with conventional user materials w.r.t. execution speed, it has not been investigated how it compares to other approaches. This will be subject of further research.

Acknowledgements

The contributions of Joachim Sprave and André Haufe were supported by the Federal Ministry for Economic Affairs and Climate Action (BMWK) due to an enactment of the German Bundestag under Grant No. 19I20024 within the project AIMM (Artificial Intelligence for Materials Modelling).

References

[1] Dmitry Alexeev. Pytorch fortran bindings. https://github.com/alexeedm/pytorch-fortran, 2022.

[2] Javier Bernal and Jose Torres-Jimenez. Sagrad: A program for neural network training with simulated annealing and the conjugate gradient method, 2015-06-17 2015.

[3] Pauline Böhringer, Daniel Sommer, Thomas Haase, Martin Barteczko, Joachim Sprave, Markus Stoll, Celalettin Karadogan, David Koch, Peter Middendorf, and Mathias Liewald. A strategy to train machine learning material models for finite element simulations on data acquirable from physical experiments. *Computer Methods in Applied Mechanics and Engineering*, 406:115894, 2023.

[4] Colin Bonatti and Dirk Mohr. One for all: Universal material model based on minimal state-space neural networks. *Science Advances*, 7(26):eabf3658, 2021.

[5] Milan Curcic. A parallel fortran framework for neural networks and deep learning, 2019.

[6] Ryan R. Curtin, Marcus Edel, Omar Shrit, Shubham Agrawal, Suryoday Basak, James J. Balamuta, Ryan Birmingham, Kartik Dutt, Dirk Eddelbuettel, Rishabh Garg, Shikhar Jaiswal, Aakash Kaushik, Sangyeon Kim, Anjishnu Mukherjee, Nanubala Gnana Sai, Nippun Sharma, Yashwant Singh Parihar, Roshan Swain, and Conrad Sanderson. mlpack 4: a fast, header-only c++ machine learning library. *Journal of Open Source Software*, 8(82):5026, 2023.

[7] Sarah Eberhard, Said Jamei, Joachim Sprave, Norbert Dölle, Steven Peters, Carmen Maria Krahe, Alexander Jacob, Tom Stähr, Fabio Echsler Minguillon, Constantin Hofmann, Leonard Overbeck, Louis Schäfer, Magnus Kandler, Gisela Lanza, Milan Marinov, Henrik Oppermann, Nicolas Schönborn, Jawad Tayyub, Volker Frey, David Koch, André Haufe, Thanh Binh Bui, Paul-Ludwig Winkler, and Klaus-Robert Müller. Verbundprojekt aiax: Machine-learning-driven engineering - cax goes aiax : gemeinsamer schlussbericht zum verbundprojekt : Laufzeit vom 01.08.2018 bis 31.07.2021. Technical report, Mercedes-Benz AG , Wbk Institut für Produktionstechnik , USU-Software AG , Endress+Hauser GmbH+Co. , DYNAmore GmbH , Technische Universität Berlin, Institut für Softwaretechnik und Theoretische Informatik, Stuttgart, 2021.

[8] Jordan Ott, Mike Pritchard, Natalie Best, Erik Linstead, Milan Curcic, and Pierre Baldi. A fortran-keras deep learning bridge for scientific computing, 2020.