ARL-TN-1221 • Aug 2024

DEVCOM ARMY RESEARCH LABORATORY

Python Vectorized User MATerial (PyVUMAT) Model 2.0 User's Guide

by Joshua C Crone

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

ARL-TN-1221 ● AUG 2024

Python Vectorized User MATerial (PyVUMAT) Model 2.0 User's Guide

by Joshua C Crone
DEVCOM Army Research Laboratory

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
August 2024	Technical Note	April–August 2024

4. TITLE AND SUBTITLE
Python Vectorized User MATerial (PyVUMAT) Model 2.0 User's Guide

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Joshua C Crone

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
DEVCOM Army Research Laboratory
ATTN: FCDD-RLA-NA
Aberdeen Proving Ground, MD 21005-5066

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TN-1221

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
DISTRIBUTION STATEMENT A. Approved for public release: distribution is unlimited.

13. SUPPLEMENTARY NOTES
Primary author's email: <joshua.crone.civ@army.mil>; ORCID: 0000-0003-0856-9149.

14. ABSTRACT
The Python Vectorized User MATerial (PyVUMAT) framework enables users to develop custom material models in Python and seamlessly integrate them into finite element analysis simulations. Python can reduce development time significantly compared with compiled languages (i.e., C, C++, and Fortran) by leveraging the vast collection of freely available Python packages. This is especially true for the emerging field of machine learning (ML)-based material models, where established Python packages such as PyTorch, TensorFlow, and JAX can drastically simplify ML model development. Direct use of these packages will allow ML material model development to keep pace with the rapid advancements in the ML community. PyVUMAT may also lower the barrier to creating novel material models for researchers who are unfamiliar with C, C++, or Fortran. The first half of this user's guide provides the necessary information to start developing and running PyVUMAT models. The latter sections provide additional guidance on improving functionality and performance.

15. SUBJECT TERMS
constitutive models; machine learning; Python; finite element analysis; VUMAT; Network, Cyber, and Computational Sciences; Sciences of Extreme Materials

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Joshua C Crone
Unclassified	Unclassified	Unclassified	UU	39	19b. TELEPHONE NUMBER (Include area code) 410-306-2156

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18

Contents

List of Figures

List of Tables

1. Introduction

The use of machine learning (ML) for the development of material models is a rapidly growing area of study within computational structural mechanics. This follows the rapid rise in ML across many fields of science and engineering made possible, in part, by the ecosystem of user-friendly and robust Python packages. These Python packages enable users to quickly design, train, test, and refine ML models. ML theory lacks the maturity to identify the appropriate model architecture, hyperparameters, and training set composition a priori, making an iterative design process crucial to ML model development. Modern Python packages simplify and often automate the process of optimizing architecture, hyperparameters, and training data for particular applications.

Unfortunately, the integration of ML-based material models into finite element analysis (FEA) tools is time consuming and labor intensive, breaking the ML development paradigm of rapidly iterating through potential models. Most FEA codes require reimplementation of ML models, either directly into the source code, if available, or through a supported subroutine interface such as the Abaqus Vectorized User MATerial (VUMAT) interface.[1] In either case, the ML model must be reimplemented from scratch, typically in C, C++, or Fortran. Given the complexity of modern ML architectures, this can be a long and error-prone task. In addition, the significant investments by major tech companies to optimize Python-based ML models for CPUs and graphics processing units (GPUs) are lost when forced to reimplement the models. Furthermore, loading the trained weights of an ML model within an FEA simulation is nontrivial. Models consist of a complex hierarchy of layers containing thousands to billions of parameters.

To bring rapid and agile ML material model development to FEA applications, we developed the Python Vectorized User MATerial (PyVUMAT) framework to enable direct integration of Python-based material models. Models created with PyVUMAT are compatible with any FEA tool supporting the Abaqus VUMAT interface. Material model developers can now leverage the functionality, usability, and performance of modern Python packages, drastically reducing the iterative design process for ML model development. PyVUMAT may also provide significant reductions in time and effort in the development of non-ML material models through the use of packages such as NumPy and SciPy to simplify equation-based models or Pandas

for models relying on lookups and interpolations of complex databases.

Sections 3–6 of this guide constitute the required reading to start developing PyVUMAT models. Sections 7 and 8 go into detail on design and performance considerations to improve PyVUMAT models. While beneficial to getting the most out of PyVUMAT, new users should not be intimidated by the additional content and can save it until after they have gained experience with PyVUMAT.

2. Release Notes

The following lists new features and changes made to PyVUMAT since the 1.0 release:

- Index ordering of 2-D NumPy arrays have been swapped to be consistent with Fortran ordering. The first index now corresponds to the material point and the second index corresponds to the component of the tensor. For example, `stressOld[i,j]` is the j^{th} component of stress for material point `i`. This change requires modification to the 2-D output variables to indicate Fortran memory layout. This is accomplished in `pyvumat/driver.py` through `np.asfortranarray()`. **Note: Any PyVUMAT model written with the index ordering of PyVUMAT 1.0 should be modified to account for this index swap.** Alternatively, PyVUMAT can be compiled with the macro `C_ORDERING` defined to revert back to the original ordering. This will make PyVUMAT 2.0 compatible with Python models developed for PyVUMAT 1.0. Definition of this macro can be accomplished with the `-DC_ORDERING` compiler flag or by defining it explicitly in `pyVUMAT.cpp`.

- Input arrays are read-only. If a user needs to modify the values of an input array, they must create a copy and the changes will only be made to the local copy of the array.

- C++ functionality in `pyVUMAT.cpp` has been replaced with C for better cross-platform compatibility. Since Abaqus defaults to a C++ compiler, `extern ''C''` is still included and the library is still compiled with a C++ compiler.

- Array sizes are explicitly passed to Python. It is no longer necessary to infer values such as nblock, ndir, and nshr from the shapes of the input arrays.

- An example to run PyVUMAT in Abaqus is provided in `pyvumat/svk/abaqus_example`. It includes a simple Abaqus input file, example INI configuration files, and the skeleton of a run script, which will require modification by the user but gives the basic steps of setting up and running the example.

3. Overview

This user's guide assumes a general familiarity with writing and executing traditional VUMAT models. The reader is directed to the Abaqus documentation[2] and numerous online resources for details on the VUMAT interface and usage. This user's guide focuses on aspects of writing and executing a PyVUMAT model that differ from traditional VUMATs.

The PyVUMAT consists of three components:

1. A Python function created by the user that defines the material model function.
2. A Python class called `Driver` (defined in `pyvumat/driver.py`) that calls the user-defined Python model.
3. A C++ function (defined in `pyVUMAT.cpp`) that interfaces with the FEA code through the standard VUMAT interface. This function creates an instance of the `Driver` class, sends the VUMAT input arguments to Python through the `Driver` class, converts the return arguments from Python, and sends the return arguments to the FEA code through the VUMAT interface.

The majority of the effort in creating a Python-based material model with PyVUMAT is in developing component 1. Minor modifications are required of the Python `Driver` class. Sections 4 and 7 provide basic and detailed guidance, respectively, for creating a new model and modifying the `Driver` class. The C++ function (component 3) should be treated as a black box by the user; no modifications are required to this function when developing new models.

The user-defined material model function (component 1) can be quite general. The only constraints are the format of the input arguments provided by the C++ function and the format of the return values expected by the C++ function. The inputs are passed as a Python dictionary, with a function declaration similar to

```
def user_function(**kwargs):
```

where the keys of the `kwargs` dictionary are the parameter names in the VUMAT interface and the values are scalars or NumPy arrays with the corresponding data. Stress and strain tensors are passed as flattened vectors with ordering (11, 22, 33, 12, 23, 31) for symmetric tensors and (11, 22, 33, 12, 23, 31, 21, 32, 13) for non-symmetric tensors. This is standard for all VUMAT models. The size of the arrays are passed to the Python function and can be inferred by the NumPy arrays. Array sizes include the following:

- N = number of material points provided by the VUMAT function call
- n_{dir} = number of direct components in a symmetric tensor
- n_{shr} = number of indirect components in a symmetric tensor
- n_{field} = number of user-defined external field variables
- n_{props} = number of user-defined properties
- n_{state} = number of user-defined state variables

See Tables 1 and 2 for a brief description of all key/value pairs passed through the input argument. The reader is referred to the VUMAT documentation for additional information on the physical meaning of the input parameters.

Table 1 Scalar inputs passed from the VUMAT to PyVUMAT through a dictionary argument

Key	Value
dt	Time increment
lanneal	Flag indicating call is during annealing process
nblock	N
ndir	n_{dir}
nfieldv	n_{field}
nprops	n_{props}
nshr	n_{shr}
nstatev	n_{state}
stepTime	Time since start of step
totalTime	Current time

Table 2 Input arrays passed from the VUMAT to PyVUMAT through a dictionary argument

Key	Value	Dimension	Shape
charLength	Characteristic element length	1	N
coordMp	Material point coordinates	2	$N \times n_{dir}$
defgradNew	Deformation gradient at the end of the increment	2	$N \times (n_{dir} + 2n_{shr})$
defgradOld	Deformation gradient at the start of the increment	2	$N \times (n_{dir} + 2n_{shr})$
density	Density	1	N
enerInelasOld	Dissipated inelastic energy at start of increment	1	N
enerInternOld	Internal energy at start of increment	1	N
fieldNew	Values of user-defined field variables at end of increment	2	$N \times n_{field}$
fieldOld	Values of user-defined field variables at start of increment	2	$N \times n_{field}$
props	User-defined properties	1	n_{props}
relSpinInc	Incremental relative rotation vector	2	$N \times n_{shr}$
stateOld	State variables at start of increment	2	$N \times n_{state}$
strainInc	Strain increment	2	$N \times (n_{dir} + n_{shr})$
stressOld	Stress at start of increment	2	$N \times (n_{dir} + n_{shr})$
stretchNew	Stretch tensor at end of increment	2	$N \times (n_{dir} + n_{shr})$
stretchOld	Stretch tensor at start of increment	2	$N \times (n_{dir} + n_{shr})$
tempNew	Temperature at end of increment	1	N
tempOld	Temperature at start of increment	1	N

The function's return arguments are four separate NumPy arrays, not a Python dictionary. The return statement should follow the form

```
return stressNew, stateNew, enerInternNew, enerInelasNew
```

where the names of the variables are up to the user but the order of the quantities must be consistent. See Table 3 for a brief description of the output parameters. The

reader is referred to the VUMAT documentation for additional information on the physical meaning of the output parameters.

There is tremendous flexibility in the functionality of the model beyond these constraints on the input and output argument formats. The examples in this user's guide only scratch the surface of the capabilities and flexibility provided by a PyVUMAT model.

Table 3 Output arguments that must be returned from any PyVUMAT model

Variable	Value	Dimension	Shape
stressNew	Stress at end of increment	2	$N \times (n_{dir} + n_{shr})$
stateNew	State variables at end of increment	2	$N \times n_{state}$
enerInternNew	Internal energy at end of increment	1	N
enerInelasNew	Dissipated inelastic energy at end of increment	1	N

4. Quick-Start Guide to Creating a PyVUMAT Model

This section describes the process of creating a simple PyVUMAT model. After reading, a user should have a basic understanding of developing a model; however, the reader is encouraged to read Sections 7 and 8 for additional tips and considerations to improve the functionality, flexibility, and performance of PyVUMAT models.

As described in Section 3, the primary task in developing a PyVUMAT material model is the creation of a Python function that can be called by the `Driver` to compute the output arguments listed in Table 3. In practice, encapsulating this function in a class is an easy way to store a "state" from one function call to the next. This simplifies the process of performing one-time initialization tasks such as parsing an input file or loading an ML model. However, this is not required; a standalone function is fully compatible with PyVUMAT.

Figure 1 contains the source code for a simple ML model to demonstrate the key features. In this example, the material model is a pretrained fully connected neural network (FCNN) with one hidden layer containing 100 nodes. The FCNN is designed to predict the six components of the symmetric 3-D corotational Cauchy

stress tensor given the six components of the symmetric 3-D stretch tensor. The code starts with the declaration of the class in line 8, which can take any name. Lines 9–26 define the constructor, always called `__init__()` in Python, which is evaluated once during the first call to the VUMAT from the FEA code. The constructor can take a file path as an argument, which is a configuration file specified by the user at run time to provide additional parameters to the PyVUMAT function (see details in Sections 6 and 7). In this example, the constructor performs the one-time task of loading the pretrained model. First, the FCNN is defined in line 12. PyTorch is employed here, but any ML Python package should be compatible. Then the configuration file is parsed to find the location of a file containing the pretrained weights in lines 18–20. An initialization (INI) file format is assumed, containing a section called `[Model]` with a property key called `modelfilepath`. This is explained in more detail in Section 7. The pretrained weights are loaded into the FCNN in line 23. Finally, the FCNN is set to "eval" mode in preparation for inference evaluations, completing the initialization.

Lines 28–53 of Fig. 1 show the definition of the material model function. The function can take any name, but must be consistent with the function call in the PyVUMAT `Driver` class. In line 31, the required input arguments from the VUMAT are extracted from the dictionary `kwargs`. In this simple model, the current stretch tensor is the only required input, but all quantities listed in Tables 1 and 2 are available to the user. In line 37, the NumPy array is converted to a PyTorch tensor. The $N \times (n_{dir} + n_{shr})$ array is already in the correct shape for batch inferencing in PyTorch, where the batch dimension is assumed to be the first dimension. The specifics of the input preprocessing will depend on the ML package employed in the model. In practice, the inputs are likely to be scaled, but that is ignored in this simple example. The output of the FCNN model is computed in line 40 and converted back to a NumPY array, as expected by the C++ PyVUMAT function, in line 44. Once again, the specifics of this postprocessing will depend on the Python package and any scaling that was performed during training of the FCNN weights. All of the output parameters listed in Table 3 must be returned by the material model. In this simple example, the energy terms are not updated and there are no state variables associated with the model. Therefore, the function simply returns the corresponding input values from the start of the time increment.

```python
# File: pyvumat/simple_ML_vumat.py
#
# Import necessary modules
import numpy as np
import torch
import configparser

class UserVumat:
    def __init__(self,config_file):

        # Create the ML model
        self.ml_model = torch.nn.Sequential(torch.nn.Linear(6,100),
                                            torch.nn.ReLU(),
                                            torch.nn.Linear(100,6))

        # Parse the INI config file and get the file path
        # to the trained weights
        parser = configparser.ConfigParser()
        parser.read(config_file)
        model_file = parser.get('Model','modelfilename')

        # Load the trained weights
        self.ml_model.load_state_dict(torch.load(model_file))

        # Set model to eval mode
        self.ml_model.eval()

    def evaluate(self, **kwargs):

        # Extract the required arguments from the keywords
        stretchNew = kwargs['stretchNew']

        # Evaluate the predicted output
        with torch.no_grad():

            # Convert from NumPy array to PyTorch tensor
            input = torch.from_numpy(stretchNew)

            # Predict stress
            output = self.ml_model(input)

            # Convert the ouput back to a NumPy array and
            # update new stress.
            stressNew = output.numpy()

        # Only the stress is updated in this simple model so we
        # return the old values for the other terms
        stateOld = kwargs['stateOld']
        enerInternOld = kwargs['enerInternOld']
        enerInelasOld = kwargs['enerInelasOld']

        # Return the output arguments of the VUMAT
        return stressNew, stateOld, enerInternOld, enerInelasOld
```

Fig. 1 A simple PyVUMAT model using an FCNN in PyTorch to predict stresses. A file like this would be created by a PyVUMAT user to develop their own Python-based material model.

This model illustrates the ease at which ML material models can be implemented, requiring fewer than 25 lines of code when ignoring comments. It also illustrates the ease at which users could test multiple models in FEA simulations. One could train a number of models with the same architecture but different learning rates, optimization schemes, or training datasets without requiring any changes to this function. The user only has to change the `modelfilename` parameter in the configuration file read at runtime. Even more powerful, the architecture can be changed with a single line of code. The width and depth of the FCNN, the types of layers, and the activation functions can be changed simply by modifying line 12. These changes would require significant effort if developed from scratch in a C, C++, or Fortran function.

The final step in creating a new PyVUMAT model is to modify the `Driver` class in `pyvumat/driver.py` to call the new function. Figure 2 shows an implementation of the `Driver` class to call the simple ML model described above. First, the PyVUMAT model must be imported, as done in line 2. In this example, the model was created in a file called `simple_ML_vumat.py` and placed within the `pyvumat` package directory. However, the file can be placed anywhere, as long as the `PYTHONPATH` environment variable is set or another mechanism is in place to ensure it can be imported successfully. In the constructor of the `Driver` class, the class defining the PyVUMAT material model is instantiated. If the model is implemented as a standalone function, `Driver.__init__()` can be left blank. In line 11, the specific user-defined function to compute the VUMAT output arguments must be specified. This is all that is required to integrate a Python-based PyVUMAT model into the standard VUMAT interface. Lines 14–20 do not need to be modified by the user. Section 7 contains an example and discussion of options to increase the functionality of the `Driver` class.

```python
# Import module with user-defined VUMAT
from pyvumat.simple_ML_vumat import UserVumat

class Driver:
    def __init__(self,config_file):
        # Instantiate object of user-defined class
        self.vumat_model = UserVumat(config_file)

    def evaluate(self, **kwargs):
        # Specify PyVUMAT function
        user_function = self.vumat_model.evaluate

        # Call PyVUMAT function
        stress, state, e_intern, e_inelas = user_function(**kwargs)

        # Set memory layout of 2D arrays to Fortran
        stress = np.asfortranarray(stress)
        state = np.asfortranarray(state)

        return stress, state, e_intern, e_inelas
```

Fig. 2 An example implementation of the `pyvumat/driver.py` file to use the PyVUMAT material model from Fig. 1

5. How to Build

Common FEA codes have three methods of incorporating user-defined VUMAT functions:

1. Compiled separately as a library that is linked into the FEA tool at run time
2. Compiled into the source code of the FEA tool
3. Compiled by the FEA tool at run time

The user is directed to the documentation of the specific FEA tool to determine the appropriate method and for details on the procedure. This section focuses on the aspects of these procedures that are specific to the PyVUMAT.

For any of the three methods listed above, the critical task is providing the following paths:

- The include directory containing `Python.h`. This directory will be referred to as `<PYTHON_INC_DIR>`. This can be determined by the command `python -c ''from sysconfig import get_path; print(get_path('include'))''`.

- The include directory containing `numpy/arrayobject.h`. This directory will be referred to a `<NUMPY_INC_DIR>`. This can be determined by the command `python -c 'from numpy import get_include; print(get_include())'`.
- The library directory containing `libpythonX.Y.so` where `X` and `Y` are the major and minor versions, respectively, of the Python build. This directory will be referred to as `<PYTHON_LIB_DIR>`. Typically, this will be located in `<PYTHON_INC_DIR/../../lib>`. If unsure, start by searching the parent directories of the paths listed as outputs from the command `python -c ''import sys; print('\n'.join(sys.path))''`.

How these paths are passed during compilation will depend on the system, compiler, and FEA code. A makefile is provided in the PyVUMAT source to compile the standalone library for method 1. The makefile is located in the top-level source directory of PyVUMAT. The variables described above must be set to the appropriate paths at the top of the makefile. After editing the makefile to set the appropriate paths, run `make` at the top-level source directory. The build was successful if the library file `libpyvumat.so` was created. This library can be used by FEA tools that link to external libraries to incorporate a user-defined VUMAT.

The process of compiling PyVUMAT into the FEA source code (method 2) will depend on the specifics of the FEA code. Most likely, `pyVUMAT.cpp` will be copied into the FEA source directory (and possibly renamed) prior to compilation of the FEA tool. Adding the include and library paths, discussed above, to the compiler will depend on the build system of the FEA code.

Abaqus is an example of an FEA tool that uses run-time compilation (method 3). At run time, the `pyVUMAT.cpp` file is passed as an argument to the execution command with `user=<full path to pyVUMAT.cpp>`. See the Abaqus manual for more details on the `user` argument. The additional include and library directories can be passed via a local `abaqus_v6.env` file by appending flags to the `compile_cpp` and `link_sl` parameters. The default values for a particular Abaqus installation are found through the command `abaqus information=environment`. Copy the lines associated with `compile_cpp` and `link_sl` from the output of this command to a new file called `abaqus_v6.env`. In this local environment file, append `'-I<PYTHON_INC_DIR>'`,

'-I<NUMPY_INC_DIR>' to the end of the flags of the compile_cpp parameter. Append '-L<PYTHON_LIB_DIR>','-lpythonX.Y' to the end of the link_sl flags, where X and Y are the major and minor versions, respectively, of the Python build. If this abaqus_v6.env is in the user's home or working directory, it will overwrite the default parameters used by Abaqus. See the Abaqus manual for more information on environment files.

When using PyVUMAT with Abaqus, the user should use the Python headers and library in the Abaqus install to ensure compatibility. The include and library directories can be identified by searching within the Abaqus install directory for Python.h, arrayobject.h, and libpythonX.Y.so (with the appropriate X and Y version numbers). When installing additional Python packages (e.g., PyTorch, scikit-learn), users should use a Python version as close as possible to the version used in Abaqus. Note that Abaqus releases prior to 2024 only support Python 2. If a user's model requires Python 3 (as required in many modern ML packages), then the model will only work with releases of Abaqus starting in 2024.

6. How to Run

Running an FEA simulation tool with a PyVUMAT model will follow the same procedure as using a traditional VUMAT model. This procedure will be specific to the individual FEA code. In addition, the PYTHONPATH environment variable should be prepended to include the PyVUMAT top-level source directory, which contains pyvumat/driver.py. If the user-defined PyVUMAT model is located in another directory, that path should be prepended to PYTHONPATH as well. The C++ component of PyVUMAT checks for a configuration file using the environment variable PYVUMAT_CONF_FILE. If this environment variable is defined by the user, the file path is passed to the Driver class for use by the material model. Use of this configuration file is shown in lines 18–20 of Fig. 1, with additional usage discussed in Section 7.

An example to run PyVUMAT in Abaqus is provided in pyvumat/svk/abaqus_example/. It includes a simple Abaqus input file for performing uniaxial compression on a unit cube represented by eight elements. Example INI configuration files are provided to run the models described in Section 7. A run script will require modification by the user depending on the specifics of their system, but gives the basic steps of setting up and running the example.

7. Advanced Guide to Creating a Model

The flexibility provided by the PyVUMAT is too open-ended to be fully captured in this guide. However, a use case is provided to highlight some additional functionality while hopefully spurring creativity in the reader. The use case is the Saint Venant–Kirchhoff (SVK) hyperelastic material model. Although this is a trivially simple model, it can highlight a number of features and design choices a user may employ to get the most out of their PyVUMAT model. It also serves as a way for users to test PyVUMAT "out of the box" against a material model available in most FEA tools. Both analytical and ML-based implementations are provided in the subdirectory `pyvumat/svk/` and discussed in this section.

7.1 Analytical Hyperelastic Model

The SVK hyperelastic material model computes the corotational Cauchy stress ($\hat{\boldsymbol{\sigma}}$) as a function of the stretch tensor ($\boldsymbol{U}$), Young's modulus (E), and Poisson's ratio (ν) through the following equation:

$$\begin{aligned}
\hat{\boldsymbol{\sigma}} &= \frac{\boldsymbol{U}\boldsymbol{S}\boldsymbol{U}}{J} \\
\boldsymbol{S} &= \frac{E}{1+\nu}\left[\boldsymbol{E} + \frac{\text{trace}(\boldsymbol{E})\nu}{1-2\nu}\boldsymbol{I}\right] \\
\boldsymbol{E} &= \frac{1}{2}\left(\boldsymbol{U}\boldsymbol{U} - \boldsymbol{I}\right) \\
J &= \det(\boldsymbol{U})
\end{aligned} \tag{1}$$

The analytical SVK model is implemented in the file `svk_vumat.py`. Multiple implementations for computing $\hat{\boldsymbol{\sigma}}$ by Eq. 1 are provided. These implementations, shown in Fig. 3, include varying levels of vectorization to improve performance. The first function, `compute_stress_loop()`, loops over each material point, converts the corresponding flattened representation of $\boldsymbol{U}$ with Abaqus ordering to a 3×3 matrix, and then performs the various matrix operations provided in NumPy to compute $\hat{\boldsymbol{\sigma}}$. The matrix representation of $\hat{\boldsymbol{\sigma}}$ is then converted back to the flattened representation for symmetric matrices before returning the values. This is the most straightforward approach to implementing a model (i.e., without any vectorization). However, many NumPy operations can leverage vectorization to speed up calculations by operating on many matrices at once. The second function, `compute_stress_mat_vec()`, converts the $N \times 6$ 2-D ar-

ray of stretch tensors for all material points into a $N \times 3 \times 3$ array to take advantage of the vectorized linear algebra operations in NumPy. The third function, `compute_stress_abq_vec()`, uses utility functions provided in PyVUMAT to perform vectorized matrix–matrix multiplication (`abq_mat_mult()`) and vectorized determinant calculations (`abq_det()`). These utility functions maintain the flattened structure and Abaqus ordering of 3-D symmetric and 3-D nonsymmetric tensors, eliminating the need to convert to and from 3×3 matrices. Utility functions for 2-D tensors are not available at this time, but may be provided in a later release. The performance of these implementations is discussed in Section 8.

The analytical SVK model is implemented in a class called `SvkVumat`, shown in Fig. 4. The constructor for the `SvkVumat` class takes an optional configuration file path to chose an implementation for computing stress. The `evaluate()` function is straightforward, with two new aspects not seen in the simple ML model of Section 4. First, the elastic constants are stored in the props array, which must be extracted from the input arguments. This model assumes there are two components to the props array containing E and ν. Since Python performs bounds checking automatically, an error will be thrown if the props array defined in the input file of the FEA tool is not large enough. These types of error handling, inherent in Python, are another benefit of prototyping non-ML material models in Python. Second, even though state variables are not required for the SVK model, two state variables are updated for demonstration purposes. These values are also extracted from the input arguments and copied to a new array so any modifications to the new state variable values do not affect the old state variables.

7.2 ML-Based Hyperelastic Model

Development of an ML-based material model starts with identifying the inputs and outputs, which will be a subset of the input and output arguments of the VUMAT listed in Tables 1–3. In practice, identifying the appropriate subset of parameters may be nontrivial, especially when there is limited knowledge of the underlying physics. However, in the SVK model, the subsets are known explicitly from Eq. 1. The inputs are the six components of $\boldsymbol{U}$ and the scalars E and ν. However, Eq. 1 reveals that the stress scales linearly with E, meaning this scaling can be performed outside of the ML model; therefore, the model requires seven input parameters. For the purposes of this demonstration, the material model ignores the energy terms and there are no state variables; therefore, the only outputs are the six components of $\hat{\boldsymbol{\sigma}}$.

```python
from pyvumat.abq_mat_utils import *

def compute_stress_loop(stretch, lame1, two_mu):
    # Storage for temporary matricies
    num_points = stretch.shape[0]
    I = np.identity(3)
    U = np.zeros((3,3))
    E = np.zeros((3,3))
    S = np.zeros((3,3))
    stress_mat = np.zeros((3,3))
    stress = np.zeros((num_points,6))

    # Loop through material points and compute stress
    for i in range(num_points):
        U[0,0], U[1,1] = stretch[i,0], stretch[i,1]
        U[2,2], U[0,1] = stretch[i,2], stretch[i,3]
        U[1,2], U[2,0] = stretch[i,4], stretch[i,5]
        U[1,0], U[2,1], U[0,2] = U[0,1], U[1,2], U[2,0]

        E = 0.5*(np.dot(U,U) - I)
        S = lame1*np.trace(E)*I + two_mu*E
        stress_mat = np.dot(np.dot(U,S),U.T)/np.linalg.det(U)
        stress[i,0], stress[i,1] = stress_mat[0,0], stress_mat[1,1]
        stress[i,2], stress[i,3] = stress_mat[2,2], stress_mat[0,1]
        stress[i,4], stress[i,5] = stress_mat[1,2], stress_mat[2,0]
    return stress

def compute_stress_mat_vec(stretch, lame1, two_mu):
    # Put stretch tensor (U) in [N, 3, 3] array
    U = abq_vec_to_mat(stretch)

    # Compute Green-Lagrange strain (E = 1/2 (U.U - I)
    I = np.identity(3).reshape(1,3,3)
    E = 0.5 * (np.matmul(U,U) - I)
    trace_E = np.trace(E, axis1=1, axis2=2).reshape(-1,1,1)

    # compute 2nd PK stress (S)
    S = two_mu*E  +  lame1*trace_E*I

    # Convert from 2nd PK (S) to corotational Cauchy = U.S.U/J
    det_U = np.linalg.det(U).reshape((-1,1))
    stress_mat = np.matmul(np.matmul(U,S),U)
    return abq_mat_to_symm_vec(stress_mat)*(1.0/det_U)

def compute_stress_abq_vec(stretch, lame1, two_mu):
    # Compute Green-Lagrange strain E = 1/2 (U.U - I)
    E = abq_mat_mult(stretch,stretch)
    E[:,:3] -= 1.0
    E *= 0.5
    trace_E = np.sum(E[:,:3], axis=1).reshape((-1,1))

    # Compute 2nd PK stress (S)
    stress = two_mu*E
    stress[:,:3] += lame1*trace_E

    # Convert from 2nd PK (S) to corotational Cauchy = U.S.U/J
    J = abq_det(stretch).reshape(-1,1)
    cauchy = abq_mat_mult(abq_mat_mult(stretch,stress),stretch)/J
    return cauchy[:,:6]
```

Fig. 3 Three methods for computing stress ($\hat{\sigma}$) from Eq. 1

```python
class SvkVumat:
    def __init__(self,config_file=None):

        if config_file is None:
            self.stress_func = compute_stress_abq_vec
        else:
            # Parse the options from the ini file
            parser = configparser.ConfigParser()
            parser.read(config_file)

            # Chose the function to compute stress
            func_name = parser.get('Model','StressFunction')
            if func_name == 'loop':
                self.stress_func = compute_stress_loop
            elif func_name == 'matrix_vectorize':
                self.stress_func = compute_stress_mat_vec
            elif func_name == 'abaqus_vectorize':
                self.stress_func = compute_stress_abq_vec
            else:
                print("\n Error: unknown function name \n")
                sys.stdout.flush()
                sys.exit()

    def evaluate(self, **kwargs):

        # Extract required arguments from the keywords
        props = kwargs['props']
        stretchNew = kwargs['stretchNew']
        stateOld = kwargs['stateOld']

        # Get the elastic properties
        youngs_mod = props[0]
        nu = props[1]
        two_mu = youngs_mod/(1+nu)
        lame1 = two_mu*nu/(1.0-2.0*nu)

        # Create storage for return values
        num_points = stretchNew.shape[0]
        stateNew = stateOld.copy()
        enerInternNew = np.zeros(num_points)
        enerInelasNew = np.zeros(num_points)

        # Compute the stress using the implementation
        # specified in the configuration file
        stressNew = self.stress_func(stretchNew,
                                     lame1,
                                     two_mu)

        # Update dummy state variables
        stateNew[:,0] += 1.0
        stateNew[:,1] -= 1.0

        return stressNew, stateNew, enerInternNew, enerInelasNew
```

Fig. 4 Implementation of the analytical SVK material model in Python. This class is defined in `pyvumat/svk/svk_vumat.py`**.**

The Python script `GenSvkTraining.py` is provided to generate training data. The source code is presented in Fig. 5. At the top of the file are various arguments that can be modified by the user to control how many data points are generated, the minimum and maximum values for $\boldsymbol{U}$, the minimum and maximum values for ν, and the name of the output file where the training data will be saved. Of note in this script is the use of the analytical SVK PyVUMAT model described above to compute the target stresses. Models developed in PyVUMAT can be easily instantiated and called outside of traditional FEA simulations tools, as shown in lines 23 and 40. Furthermore, only the input arguments relevant to that model must be passed through the input dictionary, as shown in line 36. This makes a PyVUMAT model useful in a variety of applications and significantly easier to debug.

The options for implementing the training and inference of an ML material model are almost limitless. The purpose of this section is to show only one such approach, with a focus on minimizing the effort and potential for user error when iterating over many ML architectures, training datasets, and hyperparameters. The key to achieving this is abstracting the creation of the ML architecture and the scaling of input and output arguments to a centralized module that will be used during training and when performing inference in the PyVUMAT model. The specifics of the implementation will depend on the ML Python package and ML architecture chosen by the user, but the general approach should apply broadly. An example of this abstraction for an FCNN is contained in `pyvumat/svk/fcnn.py` and shown in Fig. 6. The FCNN architecture will be used for the ML-based SVK example. The FCNN architecture is defined by the `FCNN` class, which takes a list containing the number of nodes in each layer of the neural network (NN). The number of layers and nodes per layer is adjustable, but the number of nodes in the first and last layers must correspond to the input and output dimensions, respectively.

The second component to the module is a class to perform three primary tasks:

1. Create the architecture, either from scratch to perform training or loaded from a saved file to perform inference;
2. Scale and inverse scale the input and output data; and
3. Save all relevant parameters (weights, scaling constants, parameters to recreate ML architecture, etc.) to a file so a trained model can be reinstantiated in the PyVUMAT material model.

```python
import numpy as np
from pyvumat.svk.svk_vumat import SvkVumat

# Define Input Parameters
num_points = 20000
strain_low, strain_high = (-0.01, 0.01)
poisson_low, poisson_high = (0.05, 0.45)
youngsMod = 1.0

# Output file name
out_file = 'Data_SVK_strain1_20K.csv'

# Generate training data
# U = [Uxx, Uyy, Uzz, Uxy, Uyz, Uxz]
stretch = np.random.rand(num_points,6)
stretch = strain_low + (strain_high-strain_low)*stretch
stretch[:,:3] += 1.0
poisson = np.random.rand(num_points,1)
poisson = poisson_low + (poisson_high-poisson_low)*poisson

# Instantiate the model
config_file = None
model = SvkVumat(config_file)

# Evaluate stress at strains
stateOld = np.zeros((num_points,2))
stress = np.zeros((num_points,6))

# Since the Poisson's ratio (and therefore props array) is
# different for each point, we can not evaluate the stresses
# in one batch. We need to evaluate them separately, passing in
# the corresponding Poisson's ratio.
for i in range(num_points):
    props = np.array([youngsMod, poisson[i,0]])
    stretchNew = stretch[i].reshape(1,6)
    kwargs = {'stretchNew':stretchNew,
              'props':props,
              'stateOld':stateOld}

    stressNew, _, _, _ = model.evaluate(**kwargs)

    stress[i,:] = stressNew.reshape(6)

# Write the input and output data to a file
index = np.arange(num_points).reshape(-1,1)
data = np.hstack((index, poisson, stretch, stress))
header = ("Index, Poisson's Ratio, Uxx, Uyy, Uzz, Uxy, Uyz, Uxz," +
          "Sxx, Syy, Szz, Sxy, Syz, Sxz")
np.savetxt(out_file,data,delimiter=', ',header=header)
```

Fig. 5 A Python script to generate training data for an ML model of SVK provided in the source code as the file `pyvumat/svk/GenSvkTraining.py`

In this example, these tasks are performed in the `FCNN_Driver` class. The constructor for `FCNN_Driver` can take parsed command-line arguments from the argparse module to create a new model for training. It can also take a file where the width and depth of the NN, trained weights, and scaling constants are stored to re-instantiate a trained model for inference. Methods are provided to process the input and output data by converting the data to a format compatible with the ML Python package (PyTorch tensors in this example) and scaling the data, either by determining the scaling constants on the fly during training or using the pretrained constants stored in the saved model file. Note that nothing in the `FCNN` and `FCNN_Driver` classes are specific to the SVK model. These classes could be used with any FCNN-based material model.

```python
class FCNN(nn.Module):
    def __init__(self, layers_sizes):
        super(FCNN, self).__init__()
        # construct the layers for the feed-forward NN
        layers = []
        for j in range(len(layers_sizes) - 1):
            layers.append(nn.Linear(layers_sizes[j],
                                    layers_sizes[j+1]))
            layers.append(nn.ReLU())

        # Remove activation from the output layer
        layers.pop()
        self.layers = nn.Sequential(*layers)

    def forward(self, input):
        return self.layers(input)

class FCNN_Driver():
    def __init__(self, cmd_args=None,saved_file=None):
        # Check for gpu
        self.device = torch.device('cuda:0' if torch.cuda.is_available()
                                   else 'cpu')
        self.type = torch.float32

        # Initialize the scalers
        self.input_scaler = StandardScaler()
        self.output_scaler = StandardScaler()

        # Get arguments for FCNN model from the command line (cmd_args) or
        # saved from previously trained model (saved_file)
        if cmd_args is not None:
            # Create the NN
            self.layers_sizes = cmd_args.layers_sizes
            self.nn = FCNN(self.layers_sizes)
        elif saved_file is not None:
            saved_model = torch.load(saved_file,
                                     map_location=self.device)

            # Load the size of each NN layer
            self.layers_sizes = saved_model['layers_sizes']

            # Create the NN
            self.nn = FCNN(self.layers_sizes)

            # Load weights & biases
            self.nn.load_state_dict(saved_model['model_state'])

            # Load the parameters for scaling the inputs and outputs
            # that were fit to the training data
            self.input_scaler.__setstate__(saved_model['input_scale'])
            self.output_scaler.__setstate__(saved_model['output_scale'])
        else:
            print("Error: Constructor for FCNN_Driver requires either",
                  "cmd_args or saved_file argument")
            sys.exit(1)

        # Convert model to appropriate type
        self.nn.to(self.type).to(self.device)
```

1

```python
    # Save the model
    def save(self,out_file_name):
        model_data = {"layers_sizes": self.layers_sizes,
                      "input_scale": self.input_scaler.__getstate__(),
                      "output_scale": self.output_scaler.__getstate__(),
                      "model_state": self.nn.state_dict()}
        torch.save(model_data,out_file_name)
        return

    # Convert tensor to appropriate type and move to appropriate device
    def to(self,tensor):
        return tensor.to(self.type).to(self.device)

    def process_data(self,data,scaler,expected_dim,fit_scaler=False):
        batch_size, dim = data.shape
        if not dim == expected_dim:
            print("Error: inconsistent dimensions when preprocessing")
            print(dim, expected_dim)
            sys.exit(1)

        # Fit the scaler if requested
        if fit_scaler:
            scaler.fit(data)

        # Scale the data
        return_data = scaler.transform(data)

        # Convert to torch tensor
        return self.to(torch.from_numpy(return_data).view(batch_size,
                                                          dim))

    def process_input(self,input_data,fit_scaler=False):
        return self.process_data(input_data, self.input_scaler,
                                 self.layers_sizes[0],fit_scaler)

    def process_output(self,output_data,fit_scaler=False):
        return self.process_data(output_data, self.output_scaler,
                                 self.layers_sizes[-1],fit_scaler)

    def inverse_scale_input(self,scaled_input):
        return self.input_scaler.inverse_transform(scaled_input)

    def inverse_scale_output(self,scaled_output):
        return self.output_scaler.inverse_transform(scaled_output)
```

2

Fig. 6 Implementation of a generic FCNN designed to be used both for training of FCNNs and for inference within a PyVUMAT model. These classes are defined in `pyvumat/svk/fcnn.py`.

The general FCNN model is trained with the Python script `TrainFCNN.py`. The script has command-line arguments allowing the user to vary the architecture (number of layers and nodes per layer), training hyperparameters (minimum and maximum learning rates, number of epochs, batch size), and training dataset (file containing data, train/test split). From this, a user can vary any combination of these parameters and store the resulting trained models as separate files with the "`-o`" command-line argument. Figure 7 contains the start and end of the source code for `TrainFCNN.py`. Much of the implementation for training the ML model will depend on the ML Python package employed and is therefore ignored in Fig. 7 for the sake of space and readability of this guide. At the beginning of the script, the command-line arguments are defined, parsed, and passed to the `FCNN_Driver` constructor. This creates a new FCNN model that is ready for training. At the end of the script, the model architecture, trained weights, and scaling constants are saved to a PyTorch file as defined in the `FCNN_Driver.save()` function. Once again, nothing in this file is specific to the SVK model. Application to the SVK model will occur by passing the training data generated by `GenSvkTraining.py` and by ensuring the number of nodes in the first and last layers of the FCNN are 7 and 6, respectively. An example command for training on the SVK data with the default hyperparameters is

```
python TrainFCNN.py -i Data_SVK_strain1_20K.csv \
--layers-sizes 7 25 100 25 6 -o SVK_layers7-25-100-25-6
```

This will create output files `SVK_layers7-25-100-25-6_error.csv` logging the training and test errors after each epoch and `SVK_layers7-25-100-25-6_model.pth` containing the saved model to be read in the PyVUMAT function.

Now all of the pieces are in place to write the ML-based PyVUMAT for the SVK model. This model is implemented in `pyvumat/svk/svknn_vumat.py` and shown in Fig. 8. As discussed in Section 4, the material model function is encapsulated in a class called `SvkNnVumat` so that parsing the configuration file and loading the ML model are only performed once. The tasks in the constructor mirror the example in Section 4 (Fig. 1) with an additional verbosity flag to demonstrate another way to leverage the configuration file and the creation of the `FCNN_Driver` object. Here, the file saved from training is passed to the constructor to reinitialize the trained FCNN model.

```python
import numpy as np
import argparse
import torch
from pyvumat.svk.fcnn import FCNN_Driver

# Construct the argument parse and parse the command line arguments
ap = argparse.ArgumentParser(description='Train a simple fully ' +
                             'connected NN with PyTorch')

ap.add_argument("--layers-sizes", type=int, required=True, nargs='+',
                help="List of number of nodes for each layer")
ap.add_argument("-i","--in-file", type=str, required=True,
                help="Name of input file")
ap.add_argument("-o","--out-file", type=str, required=True,
                help="Prefix of output files")
ap.add_argument("--epochs", default=1000, type=int,
                help="Number of epochs [1000]")
ap.add_argument("--batch", default=100, type=int,
                help="Batch size [100]")
ap.add_argument("--max-lr", default=0.001, type=float,
                help="Max learning rate [0.001]")
ap.add_argument("--min-lr", default=1.0e-6, type=float,
                help="Min learning rate [1.0e-6]")
ap.add_argument("--train-ratio", default=0.9, type=float,
                help="Ratio of data to use for training [0.9]")
ap.add_argument("--test-ratio", default=0.1, type=float,
                help="Ratio of data to use for testing [0.1]")
args = ap.parse_args()

# Build the model using the command line arguments
model = FCNN_Driver(cmd_args=args)

# Read the files containing training and test data
raw_data = np.loadtxt(args.in_file,
                      skiprows=1, delimiter=',')

                      .
                      .
                      .

# Save the model to a file
model.save(args.out_file+"_model.pth")
```

Fig. 7 The start and end of a Python script to train an FCNN. The middle section of this script is ignored as it is specific to training in PyTorch. However, the full code is available in `pyvumat/svk/TrainFCNN.py`**.**

```python
from pyvumat.svk.fcnn import FCNN_Driver

class SvkNnVumat:
    def __init__(self,config_file):
        # Read the configuration file
        parser = configparser.ConfigParser()
        parser.read(config_file)
        model_file = parser.get('Model','modelfilename')
        self.verbose = parser.getint('Model','verbose',
                                     fallback=0)

        # Create the model, load the trained weights and
        # scaling parameters
        self.model = FCNN_Driver(saved_file=model_file)

        # Set model to eval mode
        self.model.nn.eval()

        if self.verbose > 0:
            print(self.model.nn, flush=True)

    def evaluate(self, **kwargs):
        # Extract the required arguments from the keywords
        stretchNew = kwargs['stretchNew']
        props = kwargs['props']
        youngsMod = props[0]

        # Add Poisson's ratio as an input to the model
        num_points = stretchNew.shape[0]
        poisson = np.full((num_points,1),props[1])
        input = np.hstack((poisson,
                           stretchNew))

        #Evaluate the predicted output
        with torch.no_grad():
            # Convert the input to a PyTorch tensor and perform
            # scaling consistent with scaling of training data.
            pyt_input = self.model.process_input(input)

            # Predict the stress from the ML model
            stress_out = self.model.nn(pyt_input)

            # Apply inverse scaling on stress
            stressNew = self.model.inverse_scale_output(stress_out)

        # Scale by E since training was done with E=1
        stressNew *= youngsMod

        # Only the stress is updated in this model so we return the
        # old values for the other terms
        stateOld = kwargs['stateOld']
        enerInternOld = kwargs['enerInternOld']
        enerInelasOld = kwargs['enerInelasOld']

        return stressNew, stateOld, enerInternOld, enerInelasOld
```

Fig. 8 Implementation of an ML-based SVK model using PyTorch and an FCNN architecture. The class is defined in `pyvumat/svk/svknn_vumat.py`**.**

The `evaluate()` function computes the VUMAT output arguments (Table 3) as a function of the VUMAT input arguments (Tables 1 and 2). In addition to the procedures discussed in Section 3, this function requires the props array to determine the elastic constants and combines ν and $\boldsymbol{U}$ to define the seven input arguments for the FCNN model. Scaling of the input and inverse scaling of the output are performed through `FCNN_Driver` to ensure consistency with the scaling performed during training. Conversion between NumPy arrays and PyTorch tensors are also performed in these scaling functions. The simplicity and flexibility of this function showcases the benefits of writing an ML-based VUMAT in Python, rather than implementing it from scratch in C, C++, or Fortran. Most likely, the latter approach would still require something similar to the Python codes defining the FCNN architecture and to train the model's weights (Figs. 6 and 7). However, the VUMAT function to perform inference would require hundreds to thousands of lines of code, significant effort to debug, and complex logic for generalizability. This is in sharp contrast to the approximately 60 lines of code presented in Fig. 8.

7.3 Modifying the Driver

The final step in creating both the analytical and ML-based SVK models is modifying the `Driver` class called by the C++ PyVUMAT function. Figure 9 contains an implementation of this `Driver` class that allows for the model to be specified at run time through an INI configuration file using the keyword `model` within the section `[Driver]`. This Driver was written to work with Python 2 and 3 so that it is compatible with older versions of Abaqus (discussed in Section 5); however, only the analytical SVK model was implemented with Python 2 compatibility. The ML-based models require a version PyTorch that is compatible with Python 3.

Example configuration files using the INI format are provided in Fig. 10. Figure 10a shows a configuration file to use the analytical SVK model with the function to compute stress based on vectorizing operations with the flattened vector representation and Abaqus ordering. Figure 10b shows a configuration file that specifies the FCNN-based SVK model and passes a file path for a trained PyTorch file. From this example configuration file, it should be clear that numerous models can be trained and the only modification required to test these models in an FEA tool is modifying line 7 (the `ModelFileName` keyword) in the INI configuration file, which can be specified at run time.

```python
import sys
if sys.version_info[0] == 2:
    import ConfigParser as configparser
else:
    import configparser

# Import user-defined models
from pyvumat.svk.svk_vumat import SvkVumat

# These modules won't work without Python 3 and PyTorch.
# It's ok to skip imports if models are not used.
try:
    from pyvumat.simple_ml_vumat import UserVumat
    from pyvumat.svk.svknn_vumat import SvkNnVumat
except:
    print("WARNING: Could not load Python 3 Modules")
    sys.stdout.flush()
    pass

class Driver:

    def __init__(self,config_file):
        # Parse the options from the ini file
        parser = configparser.ConfigParser()
        parser.read(config_file)

        # Choose the model
        model_type = parser.get('Driver','model')
        if model_type == 'simple_ml':
            self.model = UserVumat(config_file)
        elif model_type == 'svk':
            self.model = SvkVumat(config_file)
        elif model_type == 'svknn':
            self.model = SvkNnVumat(config_file)
        else:
            print("\n Error: unknown model type \n")
            sys.stdout.flush()
            sys.exit()

    def evaluate(self, **kwargs):
        # Specify PyVUMAT function
        user_function = self.model.evaluate

        # Call PyVUMAT function
        stress, state, e_intern, e_inelas = user_function(**kwargs)

        # Set memory layout of 2D arrays to Fortran
        stress = np.asfortranarray(stress)
        state = np.asfortranarray(state)

        return stress, state, e_intern, e_inelas
```

Fig. 9 An example of `pyvumat/driver.py` modified to use any of the PyVUMAT models presented in this guide

```
# File: pyvumat_conf_svk.ini
[Driver]
Model = svk

[Model]
StressFunction = abaqus_vectorize
```

(a)

```
# File: pyvumat_conf_svkNN.ini
[Driver]
Model = svknn

[Model]
Verbose = 1
ModelFileName = ./trainedModels/Svk_layers7-25-100-25-6_model.pth
```

(b)

Fig. 10 Example configuration files for use with the `Driver` class and PyVUMAT models presented in this guide. (a) Uses the analytical SVK model and a vectorized implementation of the stress computation. (b) Uses FCNN-based SVK model. These files are passed to `Driver` by defining the `PYVUMAT_CONF_FILE` environment variable at run time.

8. Performance

Some overhead costs are unavoidable with PyVUMAT due to the added conversions to and from NumPy arrays. In addition, Python execution times will be slower than C, C++, or Fortran. However, the philosophy behind PyVUMAT is that the ability to rapidly prototype and refine material models outweighs the loss in speed, particularly in research settings. This section explores the computational performance to quantify this tradeoff for the test case of the SVK PyVUMAT models discussed in Section 7. The results were generated from PyVUMAT 1.0, but the changes to version 2.0 have a negligible effect on computational performance.

The metric of interest for assessing performance is the wall-clock time to perform the material model evaluations for one time step of an FEA simulation. This time does not include the cost of file input/output, parallel communication, contact algorithms, or any other cost outside of the material model evaluation. Reported timings are averaged over 10,000 time steps. Simulations were performed on an HPE SGI 8600 system with Intel Skylake processors clocked at 2.7 GHz at the Navy DOD Supercomputing Resource Center. The FEA simulations were performed with the

Arbitrary Lagrangian–Eulerian 3-D (ALE3D) simulation tool.[3]

The wall-clock times per time step with increasing number of elements are plotted in Fig. 11 for simulations on a single CPU processor. The benchmark model is an SVK VUMAT implemented directly in C++. At low element counts, the C++ implementation is 4 to 16 times faster. However, all implementations achieve times of 10^{-3} s or better for small element counts. This regime is not particularly interesting from a performance standpoint. Most large-scale simulations, where performance is important, will have at least thousands of elements per process. In this regime, the PyVUMAT SVK implementation that loops over each material point is predictably the slowest, but the other PyVUMAT SVK models show similar execution times to the C++ implementation. All models show a linear scaling with more than a few thousand elements. Figure 12 contains the wall-clock time per time step normalized by the cost of the C++ implementation. This provides a quantitative assessment of the added cost of Python. For large element counts, the loop-based implementation of SVK is 15 to 20 times slower than the C++ implementation. Both vectorization schemes show significant speedup at high element counts. The vectorization scheme that maintains the Abaqus flattened representation of tensors achieves speeds that are only 15% to 20% slower than the C++ implementation. The ML-based model achieves performance comparable to the analytical model at high element counts, where the batch optimizations in PyTorch take effect.

Parallel FEA simulations were also performed for a fixed simulation size of 2.1 million elements. The average wall-clock time per time step is shown in Fig. 13 with increasing numbers of CPU processes. Excellent parallel scalability is demonstrated on thousands of processes. While this result is not entirely surprising, as communication costs are not included in this timing, it does show that thousands of Python evaluations can efficiently run concurrently on separate processes.

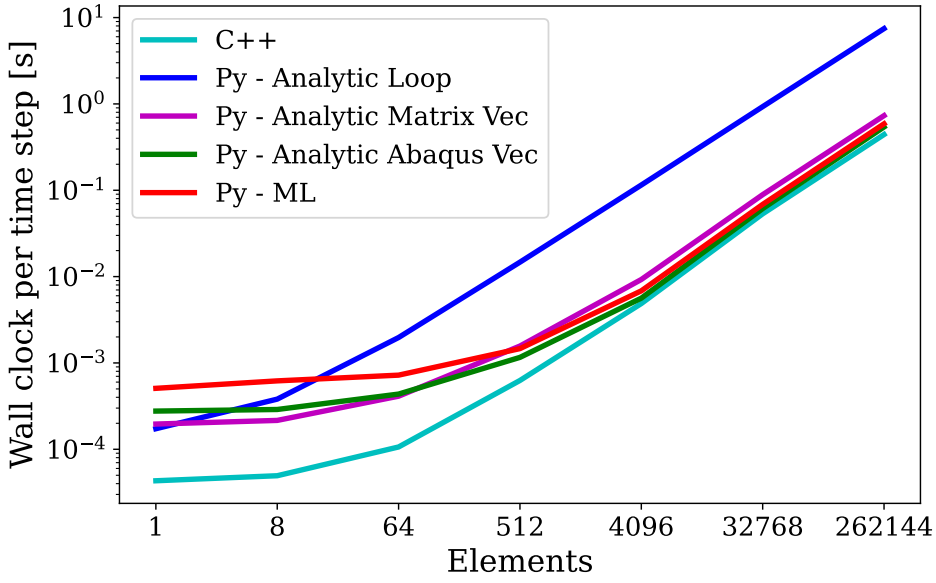

Fig. 11 Average wall-clock time per time step to evaluate the material model for various implementations of the SVK hyperelastic model. Simulations were performed on a single CPU process.

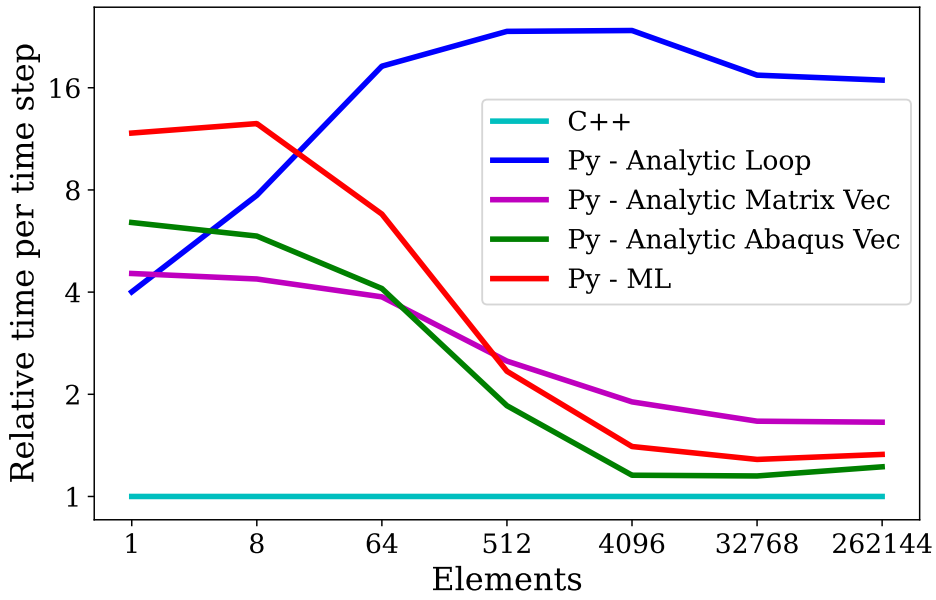

Fig. 12 Timings from Fig. 11 normalized by the C++ implementation of the SVK model. This shows the increase in computational cost of each PyVUMAT mode.

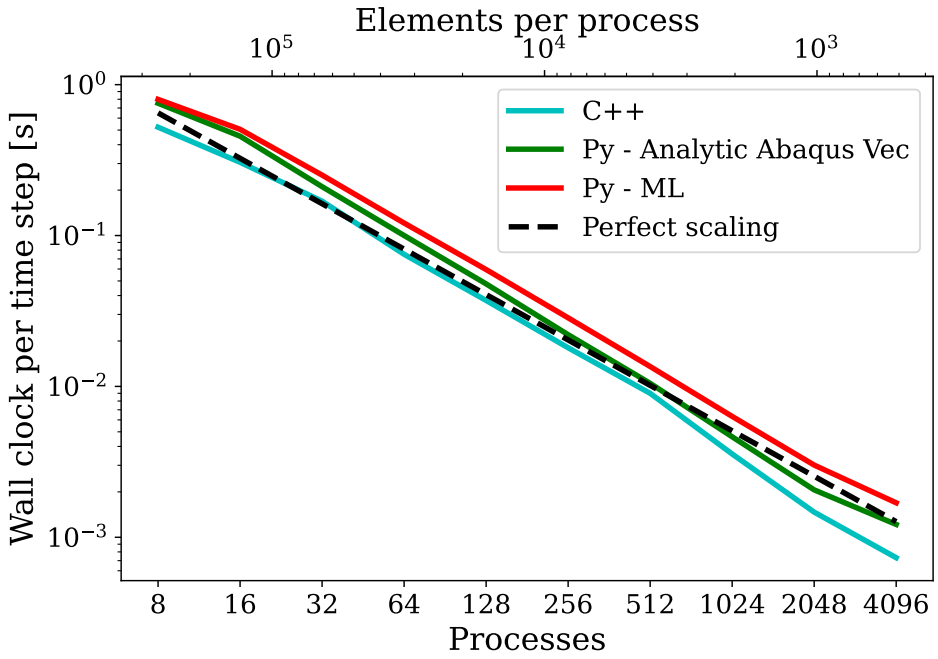

Fig. 13 Average wall-clock time per time step of parallel FEA simulations with fixed number of elements at 2.1 million

The specific performance of PyVUMAT models will depend on many factors. The purpose of this section is simply to demonstrate that the cost of PyVUMAT models are not prohibitively high, particularly for research settings. There are many situations where the significant reduction in development time provided by PyVUMAT will outweigh the added computational cost. And as demonstrated in this section, the computational cost can be reduced to a small increase over compiled languages through careful use of vectorization capabilities in Python. Furthermore, Python packages such as CuPy or JAX may improve performance further by seamlessly integrating GPU acceleration with minimal effort by the developer.

9. Conclusion

The rapid growth in ML is due, in part, to the ability to quickly and easily develop ML models and deploy them. Whether it is PyTorch, TensorFlow, JAX, or any of the countless packages openly available, Python is the programming language of choice for ML. This is the motivation behind PyVUMAT—to bring the features and usability of Python into computational mechanics. As shown in this guide, users can create Python-based material models following minimal constraints on the input and output arguments and then easily integrate them into the PyVUMAT framework. Once integrated, running a PyVUMAT model is nearly identical to running a traditional VUMAT model. The reduction in development time and increased flexibility of PyVUMAT models over those written in compiled languages can be tremendous, as demonstrated in Section 7. Furthermore, there has been significant investment by tech companies and individuals over the past few years to improve the performance of Python. These improvements reduce the trade-off between ease of development and computational cost that might be expected when switching from C, C++, or Fortran to Python.

PyVUMAT is an example of how the functionality of FEA tools can increase dramatically through the support of the VUMAT interface. FEA codes that do not support VUMATs, or only have partial support, are missing out on the opportunity to instantly add ML modeling capabilities to their software. Furthermore, VUMATs provide a clear transition path from academic research to government laboratories. Material models can be developed and tested in commercial FEA codes such as Abaqus and integrated directly into government codes. PyVUMAT also supports the development of simple and easy-to-use code that would make the handoff from academia to government laboratories significantly easier.

The content of this user's guide merely covers a small sample of what can be accomplished with PyVUMAT. There are hundreds of thousands of Python packages providing opportunities to bring new functionality, performance, and usability to Python-based material models.

10. References

1. Hibbitt D, Karlsson B, Sorensen P. Abaqus standard user's manual. Hibbitt, Karlsson & Sorensen, Inc; 1997.

2. Dassault Systèmes. Abaqus User Subroutines Reference Guide. http://130.149.89.49:2080/v6.14/books/sub/default.htm?startat=ch01s02asb20.html#sub-rtn-uexpmat; 2014.

3. Nobel CR, Anderson AT, Barton NR, Bramwell JA, Capps A, Chang MH, Chou JJ, Dawson DM, Diana ER, Dunn TA, et al. ALE3D: an arbitrary Lagrangian-Eulerian multi-physics code. Lawrence Livermore National Laboratory (US); 2017. Report No.: LLNL-TR-732040.

List of Symbols, Abbreviations, and Acronyms

TERMS:

1-D	one-dimensional
2-D	two-dimensional
3-D	three-dimensional
ALE3D	Arbitrary Lagrangian-Eulerian 3-D
CPU	central processing unit
DOD	Department of Defense
FCNN	fully connected neural network
FEA	finite element analysis
GPU	graphics processing unit
INI	initialization
ML	machine learning
NN	neural network
PyVUMAT	Python Vectorized User MATerial
SVK	Saint Venant–Kirchhoff
VUMAT	Vectorized User MATerial

(U) MATHEMATICAL SYMBOLS:

$\hat{\boldsymbol{\sigma}}$	corotational Cauchy stress tensor
$\boldsymbol{U}$	stretch tensor
E	Young's modulus
ν	Poisson's ratio

1 (PDF)	DEFENSE TECHNICAL INFORMATION CTR DTIC OCA
1 (PDF)	DEVCOM ARL FCDD RLB CI TECH LIB