27/28th International Conference
ENGINEERING MECHANICS 2022
Milovy, Czech Republic, May 9 –12, 2022
Paper **#35,** pp. 73–76, doi: 10.21495/51-2-73

LEVERAGING PYTHON TENSOR CONTRACTION PACKAGES FOR EVALUATING FINITE ELEMENT WEAK FORMS

Cimrman R. *

Abstract: *A new implementation of finite element matrix evaluation functions in the finite element code SfePy is introduced, leveraging several Python tensor contraction packages that implement a general function for evaluating expressions given using the Einstein summation convention. An example of the new weak form implementation is shown, and then results of a numerical study are presented comparing performance of the new implementation, the original implementation, and FEniCS suite.*

Keywords: Finite element method, Tensor contractions, Weak form transpiler, Numerical study.

1. Introduction

Computing weak form integrals in individual finite elements can be a bottleneck in the FE assembling process, especially when higher order approximations are in use. This was the case of the freely available finite element package SfePy (Simple Finite Elements in Python) (Cimrman et al., 2019), co-developed by the author, which has been mostly used with low-order finite elements — the higher-order approximations, although available, were not implemented very efficiently in terms of the elapsed time as well as the computer memory usage.

Python, being an interpreted language, has very slow loops. In order to be fast, SfePy employs two basic strategies. The first one is using vectorized operations enabled by NumPy array data structure (Harris et al., 2020) for the calculations. When a vectorization is not possible or it would be too difficult, the other strategy applies: critical parts of the code are implemented in C and Cython (Behnel et al., 2011) languages. The two strategies come with a cost: the vectorization, to be efficient, needs to be applied to data large enough, and using a low level language complicates the implementation. In multiscale and multiphysical simulations (the principal application of SfePy), e.g. in the field of biomechanics (Rohan et al., 2021), it is often necessary to implement several new weak forms (called terms in the code), and while it is not very difficult, still it is a hurdle for users with non-programmer background.

To make implementing weak forms easier, we decided to exploit existing Python tensor contraction packages that implement a general function, called in this text *einsum*, for evaluating expression given a description of operands using the Einstein summation convention — a standard and well known notation common in the continuum mechanics and finite element contexts. The considered packages were: NumPy (Harris et al., 2020) (the basic implementation and some optimizations from opt_einsum); opt_einsum (Smith and Gray, 2018) (state-of-the-art contraction optimization strategies); Dask (Dask Development Team, 2016) (parallel out-of-core calculations with very large data); JAX (Bradbury et al., 2021) (the JIT (just-in-time) compilation and possible parallel execution or automatic GPU transfer). In (Cimrman, 2021a) a transpiler** was proposed for translating generalized einsum-like expressions to the actual einsum expressions for evaluating multilinear finite element weak forms. A desirable side effect of this approach was an efficient (in terms of elapsed time) support for higher-order approximations, as was shown in the article (all obtained data are available online (Cimrman, 2021b)).

* Robert Cimrman, New Technologies - Research Centre and Faculty of Applied Sciences, University of West Bohemia, Plzeň; CZ, cimrman3@ntc.zcu.cz

** A transpiler translates input in a language to another language that works at approximately the same level of abstraction, unlike a traditional compiler that translates from a higher level programming language to a lower level programming language.

In this contribution we first show an example of a weak form implementation that uses the transpiler, and then present results of a new numerical study comparing performance of the transpiler-based terms, and the original implementation.

2. Example weak form implementation

The transpiler allows straightforward definitions of multi-linear finite element weak forms such as those given in Tab. 1 (a dot denotes a partial derivative, 0 stands for scalars). It supports several einsum evaluation backends based on the packages given above, arbitrary memory layout of operands, easy automatic differentiation due to (multi-)linearity of the considered weak forms and various evaluation modes.

In Listing 1 the full implementation of the Navier-Stokes convection term is shown. This is in stark contrast to the original implementation, that uses about the same amount of Python code, but additionally more than 200 lines of C, not counting the Cython wrapping code — all this is removed by the transpiler (1100 lines of Python including alternative evaluation function implementations used for benchmarking).

description	definition	weak form expression
weak Laplacian	`('0.i,0.i', v, u)`	$\int_T \frac{\partial v}{\partial x_i} \cdot \frac{\partial u}{\partial x_i}$
Navier-Stokes convection	`('i,i.j,j', v, u, u)`	$\int_T v_i \frac{\partial u_i}{\partial x_j} u_j$

Tab. 1: Examples of multi-linear weak form definitions.

```
class EConvectTerm(ETermBase):
    """
    Nonlinear convective term.
    """
    name = 'de_convect'
    arg_types = (('virtual', 'state'),
                 ('parameter_1', 'parameter_2'))
    arg_shapes = {'virtual' : ('D', 'state'), 'state' : 'D',
                  'parameter_1' : 'D', 'parameter_2' : 'D'}
    modes = ('weak', 'eval')

    def get_function(self, virtual, state, mode=None, term_mode=None,
                     diff_var=None, **kwargs):
        return self.make_function(
            'i,i.j,j', virtual, state, state, diff_var=diff_var,
        )
```

Listing 1: The Navier-Stokes convection term implementation.

3. Numerical study: comparing new and original terms

The performance measurements were executed on a Linux workstation with the AMD Ryzen Threadripper 1920X 12-Core Processor, 32 GB RAM and Python version 3.9.5, NumPy 1.22.1, JAX 0.2.27, SfePy 2021.4+git.25152f5d, FEniCS 2019.1.0 (Anaconda package py39hf3d152e_26). The calculations were limited to a single CPU using the affinity setting. The original and new term implementations in SfePy were compared to each other and also to a widely acknowledged and used package FEniCS (Logg et al., 2012) to provide a broader context.

The results are summarized in Figs. 1, 2 (the highest values for each color-coded approximation order are annotated by arrows) and Tab. 2. Only two of the available transpiler backends are considered: jax (a JIT compilation), and numpy_loop (a loop in Python over cells, evaluate using NumPy's `einsum()`). Both backends offer a significant speed-up w.r.t. the original implementation for higher order approximations. The jax backend is the most memory demanding, but performs well also for low orders, while the numpy_loop has the same memory footprint as the original implementation, but it is slow for low orders.

4. Conclusion

A new implementation of multilinear weak forms in the FE code SfePy, based on a transpiler of generalized einsum-like expressions was described and its performance demonstrated using a numerical study.

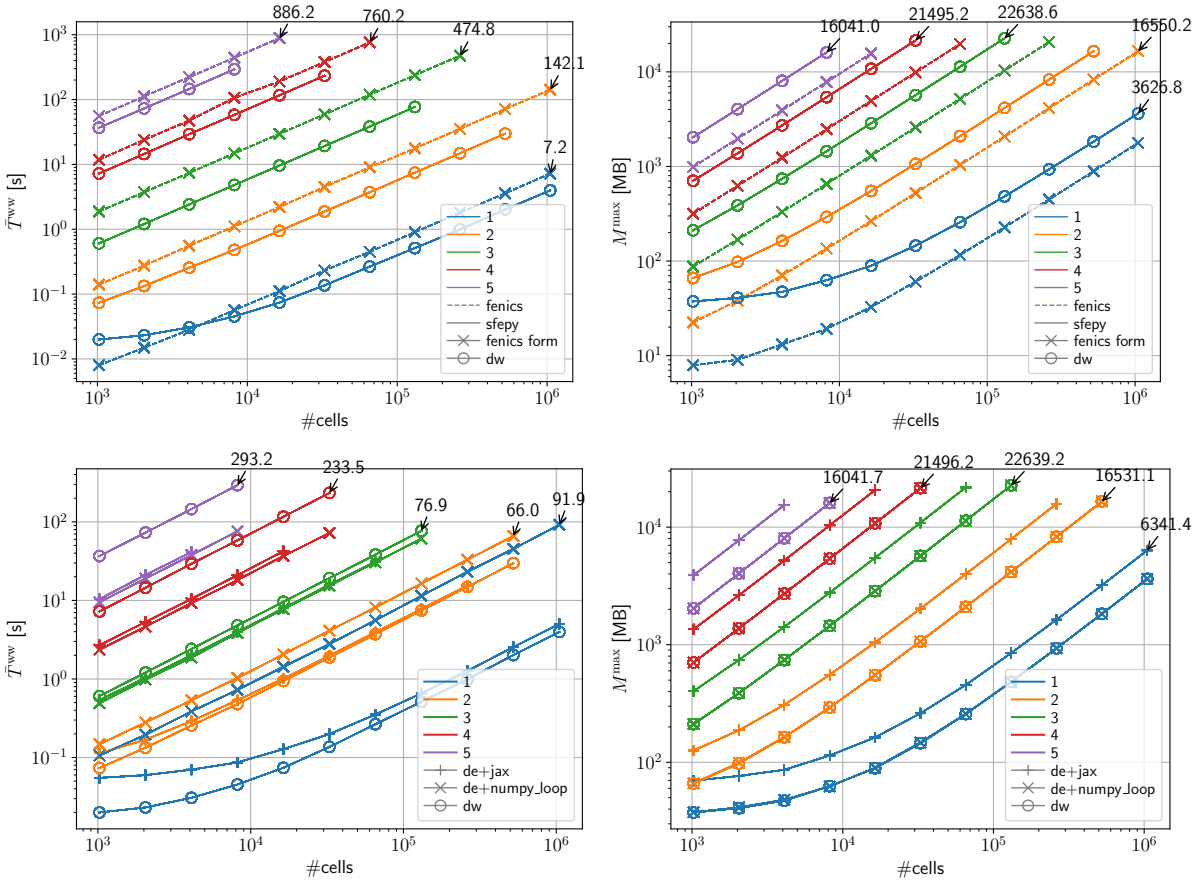

Fig. 1: Performance of the weak Laplacian matrix evaluations for various approximation orders: top: original term (circles), FEniCS (crosses), bottom: new term with jax (+), numpy_loop (x) backends, original term (circles). Left: the elapsed time mean without the worst case $\bar{T}^{ww}$, right: the memory consumption M^{max}.

Acknowledgments

The work was supported from European Regional Development Fund — Project "Application of Modern Technologies in Medicine and Industry" (No. CZ.02.1.01/0.0/0.0/17_048/0007280).

References

Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D., and Smith, K. (2011) Cython: The best of both worlds. *Computing in Science Engineering*, 13, 2, pp. 31 –39.

Bradbury, J., Frostig, R., and et al. (2021) JAX: composable transformations of Python+NumPy programs. `https://github.com/google/jax`. ver. 0.2.9.

Cimrman, R. (2021a) Fast evaluation of finite element weak forms using python tensor contraction packages. *Advances in Engineering Software*, 159.

Cimrman, R. (2021b) Performance measurements of Python tensor contraction packages in the finite element context. `https://doi.org/10.5281/zenodo.4750560`.

Cimrman, R., Lukeš, V., and Rohan, E. (2019) Multiscale finite element calculations in python using sfepy. *Advances in Computational Mathematics*.

Dask Development Team (2016) *Dask: Library for dynamic task scheduling*.

Harris, C. R., Millman, K. J., and et al. (2020) Array programming with NumPy. *Nature*, 585, 7825, pp. 357–362.

Logg, A., Mardal, K.-A., and Wells, G., eds (2012) *Automated Solution of Differential Equations by the Finite Element Method: The FEniCS Book*. Lecture Notes in Computational Science and Engineering. Springer-Verlag, Berlin Heidelberg.

Rohan, E., Turjanicová, J., and Lukeš, V. (2021) Multiscale modelling and simulations of tissue perfusion using the Biot-Darcy-Brinkman model. *Computers & Structures*, 251, pp. 106404.

Smith, D. G. A. and Gray, J. (2018) opt_einsum - a python package for optimizing contraction order for einsum-like expressions. *Journal of Open Source Software*, 3, 26, pp. 753.

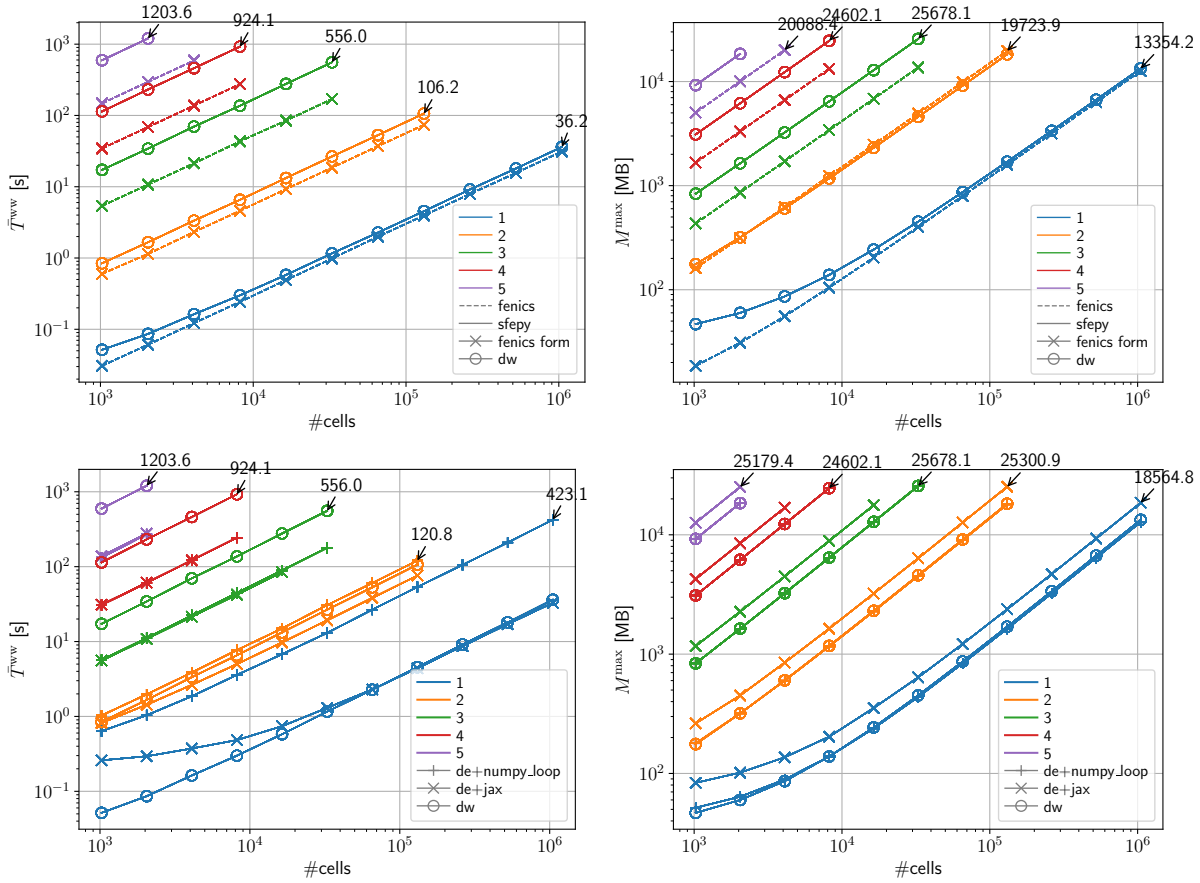

Fig. 2: Performance of the Navier-Stokes convection term matrix evaluations for various approximation orders: top: original term (circles), FEniCS (crosses), bottom: new term with jax (+), numpy_loop (x) backends, original term (circles). Left: the elapsed time mean without the worst case $\bar{T}^{\mathrm{ww}}$, right: the memory consumption M^{max}.

	Laplacian				
order	1	2	3	4	5
$\mathrm{med}(\bar{T}^{\mathrm{ww}}_{\mathrm{sfepy-dw}}/\bar{T}^{\mathrm{ww}}_{\mathrm{fenics}})$	0.59	0.42	0.32	0.61	0.66
$\mathrm{med}(\bar{T}^{\mathrm{ww}}_{\mathrm{sfepy-de-jax}}/\bar{T}^{\mathrm{ww}}_{\mathrm{fenics}})$	0.86	0.46	0.27	0.22	0.19
$\mathrm{med}(\bar{T}^{\mathrm{ww}}_{\mathrm{sfepy-de-numpy\_loop}}/\bar{T}^{\mathrm{ww}}_{\mathrm{fenics}})$	12.83	0.93	0.26	0.19	0.17
$\mathrm{med}(M^{\mathrm{max}}_{\mathrm{sfepy-dw}}/M^{\mathrm{max}}_{\mathrm{fenics}})$	2.40	2.04	2.20	2.18	2.05
$\mathrm{med}(M^{\mathrm{max}}_{\mathrm{sfepy-de-jax}}/M^{\mathrm{max}}_{\mathrm{fenics}})$	4.31	3.92	4.23	4.19	3.93
$\mathrm{med}(M^{\mathrm{max}}_{\mathrm{sfepy-de-numpy\_loop}}/M^{\mathrm{max}}_{\mathrm{fenics}})$	2.43	2.04	2.20	2.18	2.05
	NS convective				
order	1	2	3	4	5
$\mathrm{med}(\bar{T}^{\mathrm{ww}}_{\mathrm{sfepy-dw}}/\bar{T}^{\mathrm{ww}}_{\mathrm{fenics}})$	1.18	1.43	3.23	3.36	4.05
$\mathrm{med}(\bar{T}^{\mathrm{ww}}_{\mathrm{sfepy-de-jax}}/\bar{T}^{\mathrm{ww}}_{\mathrm{fenics}})$	1.32	1.07	1.01	0.89	0.93
$\mathrm{med}(\bar{T}^{\mathrm{ww}}_{\mathrm{sfepy-de-numpy\_loop}}/\bar{T}^{\mathrm{ww}}_{\mathrm{fenics}})$	13.56	1.66	1.05	0.89	0.90
$\mathrm{med}(M^{\mathrm{max}}_{\mathrm{sfepy-dw}}/M^{\mathrm{max}}_{\mathrm{fenics}})$	1.12	0.94	1.89	1.86	1.83
$\mathrm{med}(M^{\mathrm{max}}_{\mathrm{sfepy-de-jax}}/M^{\mathrm{max}}_{\mathrm{fenics}})$	1.59	1.31	2.61	2.55	2.51
$\mathrm{med}(M^{\mathrm{max}}_{\mathrm{sfepy-de-numpy\_loop}}/M^{\mathrm{max}}_{\mathrm{fenics}})$	1.09	0.93	1.88	1.85	1.83

Tab. 2: The SfePy/FEniCS ratios of the elapsed time mean without the worst case $\bar{T}^{\mathrm{ww}}$ and the memory consumption M^{max} for the weak Laplacian and Navier-Stokes convective term weak forms for various approximation orders. The values correspond to medians over various numbers of mesh cells, see Figs. 1, 2. Original SfePy terms data are denoted by "sfepy-dw", the multilinear terms with jax and numpy_loop backends are denoted by "sfepy-de-jax" and "sfepy-de-numpy_loop", respectively.