Custom assembling system

Disclaimer: Github's markdown may not render properly LaTeX equations of this readme, so, please, take a look at its `.pdf` version. Otherwise, we advise you to download the `.md` file and render it locally on your machine.

Context

We consider the following typical non-linear problem in our study:

Find $\underline{u} \in V$, such that

$$F(\underline{u}) = \int_\Omega \underline{\underline{\sigma}}(\underline{u}) : \underline{\underline{\varepsilon}}(\underline{v}) dx - \int_\Omega \underline{f}\underline{v} dx = 0 \quad \forall \underline{v} \in V,$$

where an expression of $\underline{\underline{\sigma}}(\underline{u})$ is non linear and cannot be defined via UFL. Let us show you some examples

- $\underline{\underline{\sigma}}(\underline{u}) = f(\varepsilon_\mathrm{I}, \varepsilon_\mathrm{II}, \varepsilon_\mathrm{III})$,
- $\underline{\underline{\sigma}}(\underline{u}) = \underset{\alpha}{\mathrm{argmin}} g(\underline{\underline{\varepsilon}}(\underline{u}), \alpha)$,

where $\varepsilon_\mathrm{I}, \varepsilon_\mathrm{II}, \varepsilon_\mathrm{III}$ are eigenvalues of $\underline{\underline{\varepsilon}}$ and g some scalar function.

As seen in the second example, $\underline{\underline{\sigma}}(\underline{u})$ can also implicitly depend on the value of other scalar, vector or even tensorial quantities, α here. The latter do not necessarily need to be represented in a finite-element function space. They shall just be computed during the assembling procedure when evaluating the expression $\underline{\underline{\sigma}}(\underline{u})$ pointwise.

In addition, in order to use a standard Newton method to find the solution of $F(\underline{u}) = 0$, we also need to compute the derivative of $\underline{\underline{\sigma}}$ with respect to $\underline{u}$. The latter may also depend on some internal variables α of $\underline{\underline{\sigma}}$. Thus, it is necessary to have a fonctionality which will allow to do a simultaneous calculation of $\underline{\underline{\sigma}}$ and its derivative during assembling procedure. D In practice, in the previous examples $\underline{\underline{\sigma}}$ depends directly on $\underline{\underline{\varepsilon}} = \frac{1}{2}(\nabla \underline{u} + \nabla \underline{u}^T)$. As a result, it is more natural to consider the non-linear expression as a function of $\underline{\underline{\varepsilon}}$ directly, provide an expression for $\dfrac{d\underline{\underline{\sigma}}}{d\underline{\underline{\varepsilon}}}$ and evaluate $\dfrac{d\underline{\underline{\sigma}}}{d\underline{u}}$ by the chain rule (letting UFL handle the $\dfrac{d\underline{\underline{\varepsilon}}}{d\underline{u}}$ part).

As the result we require the following features:

1. To define nonlinear expressions of variational problems in more complex way than it's allowed by UFL
2. To let an "oracle" provide values of such an expression and its derivative(s)
3. To call this oracle on-the-fly during the assembly to avoid unnecessary loops, precomputations and memory allocations

The following text describes our own view of these features implementation.

Custom assembling

Following the concept of the custom assembler, which uses the power of `numba` and `cffi` python libraries, we implemented our own version of custom assembler, where we can change the main loop of the assembling procedure.

CustomFunction

We would like to introduce a concept of `CustomFunction` (for lack of a better name), which is essential for our study. Let us consider the next simple variational problem

$$\int_\Omega g \cdot uvdx, \quad \forall \underline{v} \in V,$$

where $\underline{u}$ is a trial function, $\underline{v}$ is a test function and the function g is an expression. For this moment we must use `fem.Function` class to implement this variational form. Knowing the exact UFL expression of g we can calculate its values on every element using the interpolation procedure of `fem.Expression` class. So we save all values of g in one global vector. The goal is to have a possibility to calculate g expression, no matter how difficult it is, in every element node (for instance, in every gauss point, if we define g on a quadrature element) during the assembling procedure.

We introduce a new entity named as `CustomFunction` (or `CustomExpression`). It 1. inherits `fem.Function` 2. contains a method `eval`, which will be called inside of the assemble loop and calculates the function local values

DummyFunction

Besides `CustomFunction` we need an other entity. Every `fem.Function` object stores its values globally, but we would like to avoid such a waste of memory updating the function value during the assembling procedure. Let us consider the previous variational form, where g contains its local-element values now. If there is one local value of g (only 1 gauss point), g will be `fem.Constant`, but we need to store different values of g in every element node (gauss point). So we introduce a concept of `DummyFunction` (or `ElementaryFunction`?), which 1. inherits `fem.Function` 2. allocates the memory for local values only 3. can be updated during assembling procedure

Examples

We implemented elasticity and plasticity problems to explain our ideas by examples.

Elasticity problem

Let's consider a beam stretching with the left side fixed. On the other side we apply displacements. Find $\underline{u} \in V$ s.t.

$$\int_\Omega \underline{\underline{\sigma}}(\underline{\underline{\varepsilon}}(\underline{u})) : \underline{\underline{\varepsilon}}(\underline{v}) d\underline{x} = 0 \quad \forall \underline{v} \in V,$$

$$\partial\Omega_{\text{left}} : u_x = 0,$$
$$(0,0) : u_y = 0,$$
$$\partial\Omega_{\text{right}} : u_x = t \cdot u_{\text{bc}},$$

where u_{bc} is a maximal displacement on the right side of the beam, t is a parameter varying from 0 to 1, and where $\underline{\underline{\sigma}}(\underline{\underline{\varepsilon}})$ is our user-defined "oracle". Here we use a simple elastic behaviour:

$$\underline{\underline{\sigma}}(\underline{\underline{\varepsilon}}) = \mathbf{C} : \underline{\underline{\varepsilon}}$$

and for which the derivative is:

$$\frac{d\underline{\underline{\sigma}}}{d\underline{\underline{\varepsilon}}} = \mathbf{C}$$

where $\mathbf{C}$ is the stiffness matrix.

Let's focus on the key points. In this "naive" example the derivative is constant, but in general non-linear models, it's value will directly depend on the local value of $\underline{\underline{\varepsilon}}$. We would like to change this value at every assembling step. In our terms, it is a `DummyFunction`. Obviously, $\underline{\underline{\sigma}}$ is the `CustomFunction`, which depends on $\underline{\underline{\varepsilon}}$.

```
q_dsigma = ca.DummyFunction(VQT, name='stiffness') # tensor C
q_sigma = ca.CustomFunction(VQV, eps(u), [q_dsigma], get_eval)  # sigma^n
```

In the `CustomFunction` constructor we observe three arguments. The first one is the UFL-expression of its variable $\underline{\underline{\varepsilon}}$ here. It will be compiled via ffcx and will be sent as "tabulated" expression to a numba function, which performs the calculation of `q_sigma`. The second argument is a list of `q_sigma` coefficients (`fem.Function` or `DummyFunction`), which take a part in calculations of `q_sigma`. The third argument contains a function generating a `CustomFunction` method `eval`, which will be called during the assembling. It describes every step of local calculation of $\underline{\underline{\sigma}}$.

Besides the local implementation of new entities we need to change the assembling procedure loop to describe explicitly the interaction between different coefficients of linear and bilinear forms. It allows us to write a quite general custom assembler,

which will work for any kind non-linear problem. Thus we have to define two additional numba functions to calculate local values of forms kernels coefficients (see the code below).

```
@numba.njit(fastmath=True)
def local_assembling_b(cell, coeffs_values_global_b, coeffs_coeff_values_b, coeffs_dummy_val
    sigma_local = coeffs_values_global_b[0][cell]

    output_values = coeffs_eval_b[0](sigma_local,
                                     u_local,
                                     coeffs_constants_b[0],
                                     geometry, entity_local_index, perm)

    coeffs_b = sigma_local

    for i in range(len(coeffs_dummy_values_b)):
        coeffs_dummy_values_b[i][:] = output_values[i] #C update

    return coeffs_b

@numba.njit(fastmath=True)
def local_assembling_A(coeffs_dummy_values_b):
    coeffs_A = coeffs_dummy_values_b[0]
    return coeffs_A
```

Plasticity problem

The elasticity case is trivial and doesn't show clearly our demands by the described above features. Therefore we present here a standard non-linear problem from our scientific domain - a plasticity one.

The full description of the problem and its implementation on a legacy version of Fenics is introduced here. Note that for this very specific example, everything can still be expressed in UFL directly. However, in general, this is no longer the case, as one may have to solve a nonlinear equation at each integration point to obtain the expression of stresses and plasticity variables.

We focus on the following variational problem only: Find $\underline{\Delta u} \in V$ s.t.

$$\int_\Omega \underline{\underline{\sigma_{n+1}}}(\underline{\underline{\varepsilon}}(\underline{\Delta u})) : \underline{\underline{\varepsilon}}(\underline{v})dx - q \int_{\partial\Omega_{\text{inside}}} \underline{n} \cdot \underline{v}dx = 0, \quad \forall \underline{v} \in V,$$

where $\underline{\Delta u}$ is a displacement increment between two load steps, $\underline{\underline{\sigma}}_{n+1}$ is the current stress tensor which depends on the previous stress $\underline{\underline{\sigma}}_n$ and the previous plastic strain p_n and which is implicitly defined as the solution to the following equations:

$$\underline{\underline{\sigma_{\text{elas}}}} = \underline{\underline{\sigma}}_n + \mathbf{C} : \underline{\underline{\Delta\varepsilon}}, \quad \sigma^{\text{eq}}_{\text{elas}} = \sqrt{\frac{3}{2}\underline{\underline{s}} : \underline{\underline{s}}}$$

$$\underline{\underline{s}} = \mathsf{dev}\, \underline{\underline{\sigma_{\text{elas}}}}$$

$$f_{\text{elas}} = \sigma^{\text{eq}}_{\text{elas}} - \sigma_0 - Hp_n$$

$$\Delta p = \frac{< f_{\text{elas}} >_+}{3\mu + H},$$

$$\beta = \frac{3\mu}{\sigma^{\text{eq}}_{\text{elas}}} \Delta p$$

$$\underline{\underline{n}} = \frac{\underline{\underline{s}}}{\sigma^{\text{eq}}_{\text{elas}}}$$

$$\underline{\underline{\sigma}}_{n+1} = \underline{\underline{\sigma_{\text{elas}}}} - \beta \underline{\underline{s}}$$

$$< f >_+ = \begin{cases} f, & f > 0, \\ 0, & \text{otherwise} \end{cases}$$

where $\underline{\underline{\Delta\varepsilon}} = \underline{\underline{\varepsilon}}(\underline{\Delta u})$ is the total strain increment.

The corresponding derivative of the non-linear expression $\underline{\underline{\sigma}}_{n+1}(\underline{\underline{\Delta\varepsilon}})$ is given by:

$$\frac{d\underline{\underline{\sigma}}_{n+1}}{d\underline{\underline{\Delta\varepsilon}}} = \mathbf{C}^{\text{tang}}(\underline{\underline{\Delta\varepsilon}}) = \mathbf{C} - 3\mu \left(\frac{3\mu}{3\mu + H} - \beta \right) \underline{\underline{n}} \otimes \underline{\underline{n}} - 2\mu\beta \mathbf{DEV}$$

In contrast to the elasticity problem the tangent stiffness depends here on $\underline{\underline{\Delta\varepsilon}}$ and has different values in every gauss point. Since it's value is needed only for computing the global jacobian matrix, we would like to avoid an allocation of such a global tensorial field. This justifies to use the concept of `DummyFunction` for $\mathbf{C}^{\text{tang}}$.

We can conclude, that the fields $\underline{\underline{\sigma_{n+1}}} = \underline{\underline{\sigma_{n+1}}}(\underline{\underline{\Delta\varepsilon}}, \beta, \underline{\underline{n}}, dp, p_n, \underline{\underline{\sigma_n}})$ and $\mathbf{C}^{\text{tang}} = \mathbf{C}^{\text{tang}}(\beta, \underline{\underline{n}})$ depend on the common variables β and $\underline{\underline{n}}$. With the legacy implementation, it was necessary to allocate additional space for them and calculate $\underline{\underline{\sigma_{n+1}}}$ and $\mathbf{C}^{\text{tang}}$ separately, but now we can combine their local evaluations.

In comparison with the elasticity case the `CustomFunction sig` has more dependent fields. Look at the code below

```
C_tang = ca.DummyFunction(QT, name='tangent') # tensor C_tang
sig = ca.CustomFunction(W, eps(Du), [C_tang, p, dp, sig_old], get_eval) # sigma_n
```

As it was expected, the local evaluation of the `CustomFunction` becomes more complex

```
def get_eval(self:ca.CustomFunction):
    tabulated_eps = self.tabulated_input_expression
    n_gauss_points = len(self.input_expression.X)
    local_shape = self.local_shape
    C_tang_shape = self.tangent.shape

    @numba.njit(fastmath=True)
    def eval(sigma_current_local, sigma_old_local, p_old_local, dp_local, coeffs_values, cor
        deps_local = np.zeros(n_gauss_points*3*3, dtype=PETSc.ScalarType)

        C_tang_local = np.zeros((n_gauss_points, *C_tang_shape), dtype=PETSc.ScalarType)

        sigma_old = sigma_old_local.reshape((n_gauss_points, *local_shape))
        sigma_new = sigma_current_local.reshape((n_gauss_points, *local_shape))

        tabulated_eps(ca.ffi.from_buffer(deps_local),
                      ca.ffi.from_buffer(coeffs_values),
                      ca.ffi.from_buffer(constants_values),
                      ca.ffi.from_buffer(coordinates), ca.ffi.from_buffer(local_index), ca.f

        deps_local = deps_local.reshape((n_gauss_points, 3, 3))

        n_elas = np.zeros((3, 3), dtype=PETSc.ScalarType)
        beta = np.zeros(1, dtype=PETSc.ScalarType)
        dp = np.zeros(1, dtype=PETSc.ScalarType)

        for q in range(n_gauss_points):
            sig_n = as_3D_array(sigma_old[q])
            sig_elas = sig_n + sigma(deps_local[q])
            s = sig_elas - np.trace(sig_elas)*I/3.
            sig_eq = np.sqrt(3./2. * inner(s, s))
            f_elas = sig_eq - sig0 - H*p_old_local[q]
            f_elas_plus = ppos(f_elas)
            dp[:] = f_elas_plus/(3*mu_+H)

            sig_eq += TPV # for the case when sig_eq is equal to 0.0
            n_elas[:,:] = s/sig_eq*f_elas_plus/f_elas
            beta[:] = 3*mu_*dp/sig_eq

            new_sig = sig_elas - beta*s
```

```
            sigma_new[q][:] = np.asarray([new_sig[0, 0], new_sig[1, 1], new_sig[2, 2], new_s
            dp_local[q] = dp[0]

            C_tang_local[q][:] = get_C_tang(beta, n_elas)

        return [C_tang_local.flatten()]
    return eval
```

Thus it can been seen more clearly the dependance of the tensor $\mathbf{C}^{\text{tang}}$ on the calculation of the tensor $\underline{\underline{\sigma}}_{n+1}$.

Summarize

We developed our own custom assembler which makes use of two new entities. This allows us to save memory, avoid unnecessary global *a priori* evaluations and do instead on-the-fly evaluation during the assembly. More importantly, this allows to deal with more complex mathematical expressions, which can be implicitly defined, where the UFL functionality is quite limited. Thanks to `numba` and `cffi` python libraries and some FenicsX features, we can implement our ideas by way of efficient code. Our realization doesn't claim to be the most efficient one. So, if you have any comments about it, don't hesitate to share them with us!

Miscellaneous

Here you find the table, which contains the time needed to solve the problem and the appropriate JIT overhead.

Mesh	Time (s)	Elements nb.	Nodes nb.	JIT overhead (s)
Coarse	2.7	1478	811	7.5
Medium	14	5716	3000	7.8
Fine	100	25897	13251	4.9

We can conclude from this table, that the time spent on the JIT compilation operations is quite negligible, if we consider dense meshes.