Tutorial Handbook Part 1
The Basic Scheme

Contents

Introduction

The goal of the exercise sessions of this workshop is that you implement your own FFT-based micromechanics solver. To do so, you may follow this *Exercise Handbook*, which contains two parts.
In this first part, we implement the necessary data structure, the nonlocal operator and finish with the implementation of the *Basic Scheme*.
The second part contains various steps to optimize your code for real-world use and to adapt to other problems, such as thermal conductivity problems. This handbook contains various checkpoints for you to verify your code.

Checkpoint 0 Before you start, be sure to have a basic knowledge of the Python programming language as well as a general idea of how object-oriented programming works. If not, please go through the introductory material you received beforehand.

1 The data structure

1.1 Microstructure images

Usually, effective properties of microstructured materials are computed on a microstructure image. This image tells us exactly how the composite is built up, i.e., at which material is present at each voxel (a 3D pixel). The image may come from a μ-CT scan of a real composite or might be synthetically generated (how to do that is out of the scope of this course and for now - we just *magically* have these images at hand). You have been provided with some microstructure images, which are stored as `.mha` files in the `Microstructures` subfolder of `FFT_Solver_Template`. Take a peek at them by opening these files in Paraview. An example of a microstructure is shown in Figure 1.

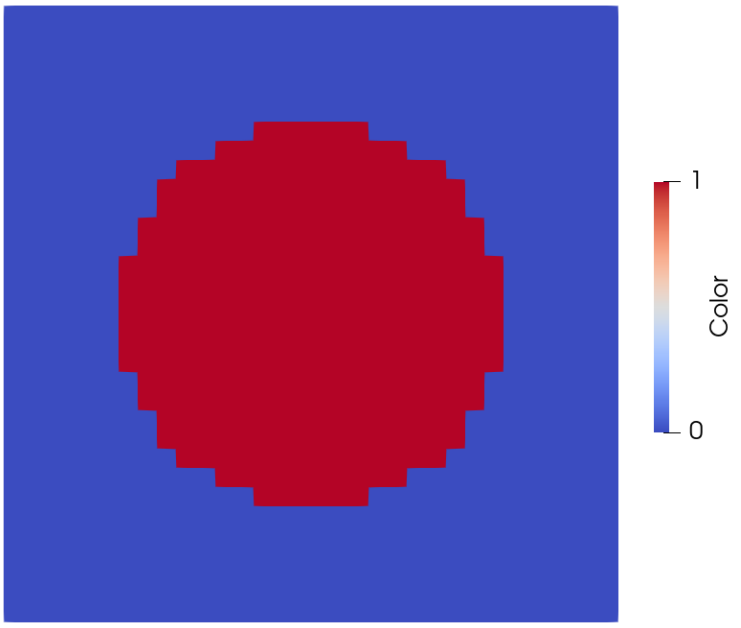

Figure 1: Slice through a cubic 3D microstructure image with a single spherical inclusion. The two different materials (or colors) are distinguished by integer values. Here, 0 refers to the blue (matrix) material and 1 refers to the red material of the sphere.

Checkpoint 1 Create a microstructure object

```
Microstructure = MicrostructureImage()
Microstructure.readColors(microstructure_path)
```

with the provided `MicrostructureImage` class and take a look at `microstructure.colors`, which is a 3D array containing the data which material can be found at voxel $[i, j, k]$. As a reference, for `ball_32.mha`, the color at `[0,0,0]` should be 0 and for `[16,16,16]` it should be 1.

1.2 Linear elastic materials

For the time being, let us restrict to linear elastic isotropic materials which are fully characterized by their Young's modulus E and Poisson's ratio ν. Create a class `LinearElasticIsotropicMaterial` that reads the material parameters in the constructor and stores them along with Lamé's constants λ and μ and the bulk modulus

K (see `https://en.wikipedia.org/wiki/Shear_modulus` for the conversion formulas). This class should have a method `computeStress` that computes the stress for any given strain **at one voxel**.
Additionally, we need to implement a reference material. Create a class `ReferenceMaterial` that inherits from the linear elastic material, but is initialized with a list/dictionary of all materials. For the basic scheme, the optimal reference material of a two phase microstructure containing isotropic phases with bulk and shear moduli computes as

```
eigenvalues = [3*material_0.K, 2*material_0.mu, 3*material_1.K, 2*material_1.
mu]
alpha_0 = 0.5*(max(eigenvalues) + min(eigenvalues))
referenceMaterial.mu = 0.5*alpha_0
referenceMaterial.la = 0.
```

Be sure to choose either the Voigt or Mandel notation and stick with it for all exercises this week. For a refresher on the notation of strain and stress tensors, refer to the hints in Section 1.7.

Checkpoint 2 Check your implementation with the following code (values are rounded):

```
material_0 = LinearElasticIsotropicMaterial(E = 2.1e3, nu = 0.3)
material_1 = LinearElasticIsotropicMaterial(E = 72e3, nu = 0.22)
referenceMaterial = ReferenceMaterial(material_0,material_1)
strain = np.array([1,0,0.1,0,0,0.5])
stress = material_1.computeStress(strain)
tau = stress - referenceMaterial.computeStress(strain)
print(stress)
print(tau)
==========================================================================
array([84519.9, 25503.5, 31405.2,     0. ,     0. , 14754.1])
array([19426.5, 25503.5, 24895.8,     0. ,     0. , -1519.3])
```

1.3 The Field class

In contrast to classical finite element schemes, FFT-based methods avoid assembling matrices and rather perform the critical operations in-place, i.e., without the need to allocate additional memory. In particular, rather large number of degrees of freedom may be treated by FFT-based methods.
To allevate the burden of handling the FFT application yourself, we provide a `Field` class which is located in the file `Field/Field.py`. This class performs the FFT in-place, i.e., the transformed and the real field share the same space in memory.
Objects of the `Field` class will be used to store our strain or stress fields. Based on the microstructure image with a shape $N_x \times N_y \times N_z$, objects of the field have the shape

$6 \times N_x \times N_y \times N_z$ in real space. Such field contain the six components of either the stress or the strain field in Voigt/Mandel notation, see section 1.7
The entries of our `Field` can be accessed either in real space via `real_field` or after transformation in frequency space via `cpx_field`. After initializing, the `Field` comprises only zeros. A uniform initial value may be assigned to all voxels using the `inialize` method.

Checkpoint 3 Check the following functionality:

```
bc_strain = [0.1,0,0,0,0,0]
myfield = Field(microstructure.colors)
myfield.initialize(bc_strain)
print(f"Initial strain: {myfield.real_field[:,0,0,0]}")
print(f"Initial strain: {myfield.real_field[:,5,3,4]}")
```

1.4 The material applicator

With our Field object, our materials and reference material, as well as our microstructure image at hand, we are ready to code the application of the material law. Create a class `MaterialApplicator` that stores the individual materials and microstructure image. The class should have a method `computeStress` that takes a `Field` object and returns the corresponding stress field *out of place´*. Make sure that you apply the correct material law at each voxel.
Additionally, the `MaterialApplicator` should handle your in-place computation of the polarization $\tau(x) = \mathtt{S}(x, \varepsilon(x)) - \mathbb{C}^0 : \varepsilon(x)$. Overwriting a field on the currently occupied memory takes the form

```
myfield.real_field[:] = stress - referenceMaterial.computeStress(myfield)
```

in Python.

1.5 Writing a field to a file

To take a look at the stress field we just created as well as to save our computational ouptut in the end, we may write a field to a file and investigate it in Paraview. Use the `PyEVTKOutput` class from `Image/WriteOutput.py`.

Checkpoint 4 Test your `MaterialApplicator` with the constant stress field we just created:

```
mapl = MaterialApplicator(microstructure, material_1, material_2)
stress_field = mapl.compute_stress_field(myfield)
```

```
mapl.computePolarization(myfield)
```

Output the `stress_field` and inspect it by opening the `output.vti` file with Paraview.

```
output = PyEVTKOutput(output_folder)
output.addStress(stress_field)
output.write()
```

The stresses (`stress_field.real_field`) vary from voxel to voxel, depending on the material at the given location.

1.6 Fast Fourier Transform (FFT)

As we mentioned earlier, the `Field` class is associated with an in-place FFT which transforms the field from real space to the frequency domain. This is essential in order to apply the nonlocal Γ operator, whose action is *local* in Fourier space.
The `fft` method transforms your field to the frequency domain. You may access this complex field `.cpx_field`. This field has four indices, and the last three indices correspond to the frequency instead of the voxel position. Caution has to be taken, as the complex field has a smaller number of entries on its last axis (the shape is $6 \times N_x \times N_y \times (N_z/2+1)$) because a complex number comprises two real numbers). To get back to the spatial domain, we may use the Inverse Fast Fourier Transform (IFFT), which can be accessed by the `ifft` method. You can check whether your field is currently in the frequency or spatial domain by accessing the `status` property of the `Field` class.

Checkpoint 5 Transform your field to the Fourier space. Take a look at the complex field.

```
myfield.fft()
print(f'Status after FFT: {myfield.status}')
print(f'Shape: {myfield.cpx_field.shape}')
```

In the lectures, the zeroth frequency stores the mean value of the field. However, in the chosen FFT framework, this is handled differently (for performance reasons). To extract the mean value you need to divide the value at frequency zero by the voxel count:

```
print(f'strain_0 = {myfield.cpx_field[:,0,0,0]/np.prod(Microstructure.colors.
  shape)}')
```

Implementing the nonlocal Γ operator requires you to know the accomodating Fourier frequencies for each index. It is useful to precompute the frequency vectors

```
xfreq = np.fft.fftfreq(Microstructure.colors.shape[0])
yfreq = np.fft.fftfreq(Microstructure.colors.shape[1])
zfreq = np.fft.rfftfreq(Microstructure.colors.shape[2]).
```

With these vectors at hand, the frequency corresponding to our field at `[i,j,k]` may be computed as

```
xi = np.array([ xfreq[i], yfreq[j], zfreq[k] ]).
```

1.7 Additional hints

Several hints and remarks might be helpful for the implementation:

- A useful form of Hooke's law $\sigma = \mathbb{C} : \varepsilon$ for isotropic linear elastic materials is

$$\sigma = 2\mu\,\varepsilon + \lambda \mathrm{tr}(\varepsilon)\,\mathrm{Id}, \quad \mathrm{Id} = \mathrm{diag}(1,1,1)$$

  with Lamé constants μ and λ.

- We further restrict to a reference material of the form $\mathbb{C}^0 = \alpha_0\,\mathrm{Id}$, i.e., $\mu_0 = 0.5\,\alpha_0$ and $\lambda_0 = 0$. Be aware of Voigt/Mandel notation (see next item).

- Before you start your implementation, you should decide whether you want to operate in Voigt or Mandel notation. The solution you were given is written in Voigt notation.

  Voigt notation:

$$\varepsilon \mathrel{\hat{=}} \begin{bmatrix} \varepsilon_{11} \\ \varepsilon_{22} \\ \varepsilon_{33} \\ 2\,\varepsilon_{23} \\ 2\,\varepsilon_{13} \\ 2\,\varepsilon_{12} \end{bmatrix} \quad \text{and} \quad \sigma \mathrel{\hat{=}} \begin{bmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{33} \\ \sigma_{23} \\ \sigma_{13} \\ \sigma_{12} \end{bmatrix}.$$

  Mandel notation:

$$\varepsilon \mathrel{\hat{=}} \begin{bmatrix} \varepsilon_{11} \\ \varepsilon_{22} \\ \varepsilon_{33} \\ \sqrt{2}\,\varepsilon_{23} \\ \sqrt{2}\,\varepsilon_{13} \\ \sqrt{2}\,\varepsilon_{12} \end{bmatrix} \quad \text{and} \quad \sigma \mathrel{\hat{=}} \begin{bmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{33} \\ \sqrt{2}\sigma_{23} \\ \sqrt{2}\sigma_{13} \\ \sqrt{2}\sigma_{12} \end{bmatrix}.$$

2 The Basic Scheme

2.1 The non-local operator

We are taking large steps towards the implementation of the Basic Scheme. Let us recap: At this moment, we can

- read a microstructure image,
- create a strain/stress field,
- apply a linear elastic material law,
- compute the stress polarization,
- perform an FFT on a field,
- write a field to a file.

Let us take a look at a single iteration of the Basic Scheme (Algorithm 1).

Algorithm 1 A single iteration of the Basic Scheme

1: $\sigma(x) \leftarrow \mathbb{C}(x) : \varepsilon(x)$ (elastic stress)
2: $\tau(x) \leftarrow \sigma(x) - \mathbb{C}^0 : \varepsilon(x)$ (polarization)
3: $\hat{\tau} \leftarrow \mathrm{FFT}(\tau)$
4: $\hat{\varepsilon}(0) \leftarrow \bar{\varepsilon}$ (macroscopic boundary condition)
5: $\hat{\varepsilon}(\xi) \leftarrow -\Gamma^0(\xi) : \hat{\tau}(\xi) \quad \forall \xi \neq 0$ (nonlinear operator)
6: $\varepsilon \leftarrow \mathrm{IFFT}(\hat{\varepsilon})$

Apparently, there is still one key ingredient missing: the non-local operator Γ^0. For a given reference material with Lamé constants λ_0 and μ_0 , applying the operator Γ^0 to a symmetric field S with $\hat{S} = \mathrm{FFT}(S)$ reads

$$\widehat{\Gamma^0 : S}(\xi) = \xi \otimes_s \left[\left(\frac{\mathrm{Id}}{\mu^0 \|\xi\|^2} - \frac{\mu^0 + \lambda^0}{\mu^0 (2\mu^0 + \lambda^0)} \frac{\xi \otimes \xi}{\|\xi\|^4} \right) \xi \cdot \hat{S}(\xi) \right] \tag{1}$$

with the frequency vector $\xi \neq 0$. By $\otimes_s$ we denote the symmetric dyadic product, i.e., $a \otimes_s b = 1/2(a \otimes b + b \otimes a)$.

2.2 Implementation steps for the non-local operator

The expression (1) for Γ^0 may seem a little daunting at first. However, we can decompose the application of Γ^0 into a series of rather simple steps. It seems beneficial to define a `GammaOperator` class which has knowledge about the dimension of your problem (i.e.,

colors.shape) and your reference material. Thus, you can precompute the frequencies once and for all, and define a method which encodes the application of Γ^0 for all frequencies.
Keep in mind that your field S, which is a stress polarization, is stored in some kind of Voigt or Mandel notation. Additionally be aware that we are dealing with complex-valued arrays. You may, for instance, create a complex numpy array with six entries as follows:

```
complex_zeros = np.zeros(6, dtype=np.complex128)
```

To compute the application of Γ^0 for a single frequency, implement the following three steps:

1. As $\hat{S}(\xi)$ is symmetric, $\xi \cdot \hat{S}(\xi) = \hat{S}(\xi) \cdot \xi$ is a simple matrix-vector product
$$\hat{f} = \hat{S}(\xi) \cdot \xi.$$

2. We denote the Green's operator G^0 by
$$\hat{G}^0(\xi) = -\frac{\mathrm{Id}}{\mu^0 \|\xi\|^2} + \frac{\mu^0 + \lambda^0}{\mu^0(2\mu^0 + \lambda^0)} \frac{\xi \otimes \xi}{\|\xi\|^4}, \quad \text{or} \quad \hat{G}^0_{ab}(\xi) = -\frac{\delta_{ab}}{\mu^0 \|\xi\|^2} + \frac{\mu^0 + \lambda^0}{\mu^0(2\mu^0 + \lambda^0)} \frac{\xi_a \xi_b}{\|\xi\|^4}$$
in tensor and index notation, respectively. Hence we may write the next step as
$$\hat{u} = -\hat{G}^0(\xi) \cdot \hat{f} \quad \text{or} \quad \hat{u}_a = -\hat{G}^0_{ab}(\xi) \hat{f}_b,$$
i.e., another matrix-vector product.

3. Last but not least, we compute the symmetrized dyadic product
$$\widehat{\Gamma^0 : S}(\xi) = \xi \otimes_s \hat{u}, \quad \text{or} \quad \widehat{\Gamma^0 : S}_{ab}(\xi) = \frac{1}{2}(\xi_a \hat{u}_b + \xi_b \hat{u}_a),$$
which is our final result.

Additionally, we need to set the correct mean value, i.e. $\hat{\varepsilon}(0) = \bar{\varepsilon}$, corresponding to the macroscopic boundary condition. Note that with the chosen FFT implementation, a factor of $N_x \cdot N_y \cdot N_z$ is required. Remember to apply Γ^0 for every frequency.

Checkpoint 6 To test your implementation, perform a single step of the Basic Scheme (Algorithm 1) with a prescribed strain ($\bar{\varepsilon}$) of `[0.1,0,0,0,0,0]` on the simple laminate microstructure `27_voxels_2_colors.mha`, see Figure 2. Material 0 has the parameters $E = 2.1\,\mathrm{GPa}$ and $\nu = 0.3$ and material 1 has the parameters $E = 72\,\mathrm{GPa}$ and $\nu = 0.22$. The optimal reference material is given by $\alpha^0 = 65.093\,\mathrm{GPa}$.
First, compute the *polarization*, i.e., the difference between the actual stress and the stress obtained from the reference stiffness in-place (see Line 1, Algorithm 1). Then perform the FFT and apply Γ^0 as well as the boundary condition. Finally apply the IFFT. Check with the following results:

$\sigma[:,0,0,0] = [28.26923077, 12.11538462, 12.11538462, 0., 0., 0.]$
$\tau[:,2,0,0] = [171.07998559, 231.8501171, 231.8501171, 0., 0., 0.]$ before FFT
$\hat{\tau}[:,1,0,0] = [-7143.7, -1977.6, -1977.6, 0., 0., 0.]$ (real components of `cpx_field` after FFT)
$\varepsilon[:,0,0,0] = [0.01812929, 0., 0., 0., 0., 0.]$ (after IFFT).

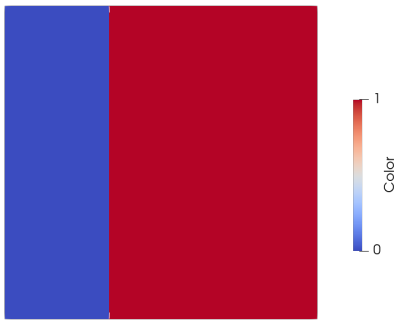

Figure 2: Slice view through the middle of one of the simpler 3D microstructures (`27_voxels_2_colors.mha`). This is a laminate consisting of a $3 \times 3 \times 3$ grid, with an uneven distribution of the materials.

2.3 The residual

The last ingredient to finalize the implementation of the basic scheme is a convergence criterion. The Basic Scheme is considered as converged provided the evaluated stress field σ is sufficiently divergence free. This may be quantified via

$$\frac{\|\mathbb{C}^0 : (\varepsilon^{k+1} - \varepsilon^k)\|}{\|\langle \sigma(\varepsilon^k) \rangle_Q\|} < \texttt{tol}, \tag{2}$$

where $\mathbb{C}^0$ denotes the reference material and the norm $\|\cdot\|$ is arises from the inner product $(S, T) = \frac{1}{|Q|} \int_Q S : T \, dx$.
It might be sensible to define a `Residual` class. Furthermore, it is useful to cache the previous iterate of ε.

2.4 Bringing it all together

The complete Basic Scheme is summarized in Algorithm 2. For efficiency reasons the quantity $\langle \sigma \rangle_Q$ can be computed without additional effort after Line 5 of Algorithm 2 via $\langle \sigma \rangle_Q = \hat{\tau}(0) + \alpha_0 \, \bar{\varepsilon}$ (keeping in mind both the Voigt notation, as well as the normalization factor for mean values in the frequency domain).

Checkpoint 7 To validate your complete Basic Scheme, use the same parameters as in Checkpoint 6. Additionally, use a tolerance of $\texttt{tol} = 10^{-5}$. The converged ε_{11} strain is

Algorithm 2 The Basic Scheme

while $k <$ maxit and res $>$ `tol` **do**
  $\varepsilon_{\text{old}} \leftarrow \varepsilon$
  $\sigma(x) \leftarrow \mathbb{C}(x) : \varepsilon(x)$
  $\tau(x) \leftarrow \sigma(x) - \mathbb{C}^0 : \varepsilon(x)$
  $\hat{\tau} \leftarrow \mathrm{FFT}(\tau)$
  $\hat{\varepsilon}(0) \leftarrow \bar{\varepsilon}$
  $\hat{\varepsilon}(\xi) \leftarrow -\Gamma^0(\xi) : \hat{\tau}(\xi)$
  $\varepsilon \leftarrow \mathrm{IFFT}(\hat{\varepsilon})$
  $\text{res} \leftarrow \alpha_0 \frac{\| \varepsilon - \varepsilon_{\text{old}} \|_{L^2}}{\| \langle \sigma \rangle_Y \|_{L^2}}$
end while

shown in Figure 3 and the converged solution (after ≈ 24) iterations should look like this:

$$\begin{aligned} \varepsilon[:,0,0,0] &= [0.0281, 0., 0., 0., 0., 0.] \quad \text{(rounded)} \\ \varepsilon[:,2,0,0] &= [0.001, 0., 0., 0., 0., 0.] \quad \text{(rounded)} \\ \langle \sigma \rangle_Q &= [79.3, 26.2, 26.2, 0., 0., 0.]\,\text{MPa}. \end{aligned}$$

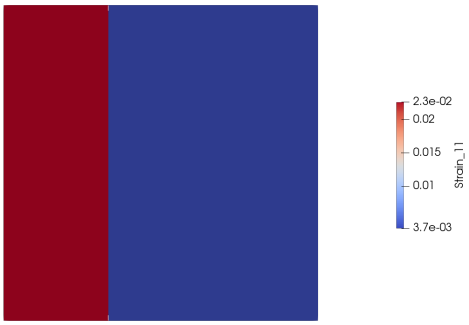

Figure 3: ε_{11} on a slice of the 3D microstructure (`27_voxels_2_colors.mha`).

You might notice that your implementation appears to be quite slow when you scale up the problem size (for instance by considering the `ball_32.mha` microstructure). We will handle this problem, as well as the implementation of additional solvers and discretizations in Part 2 of the Tutorial Handbook.