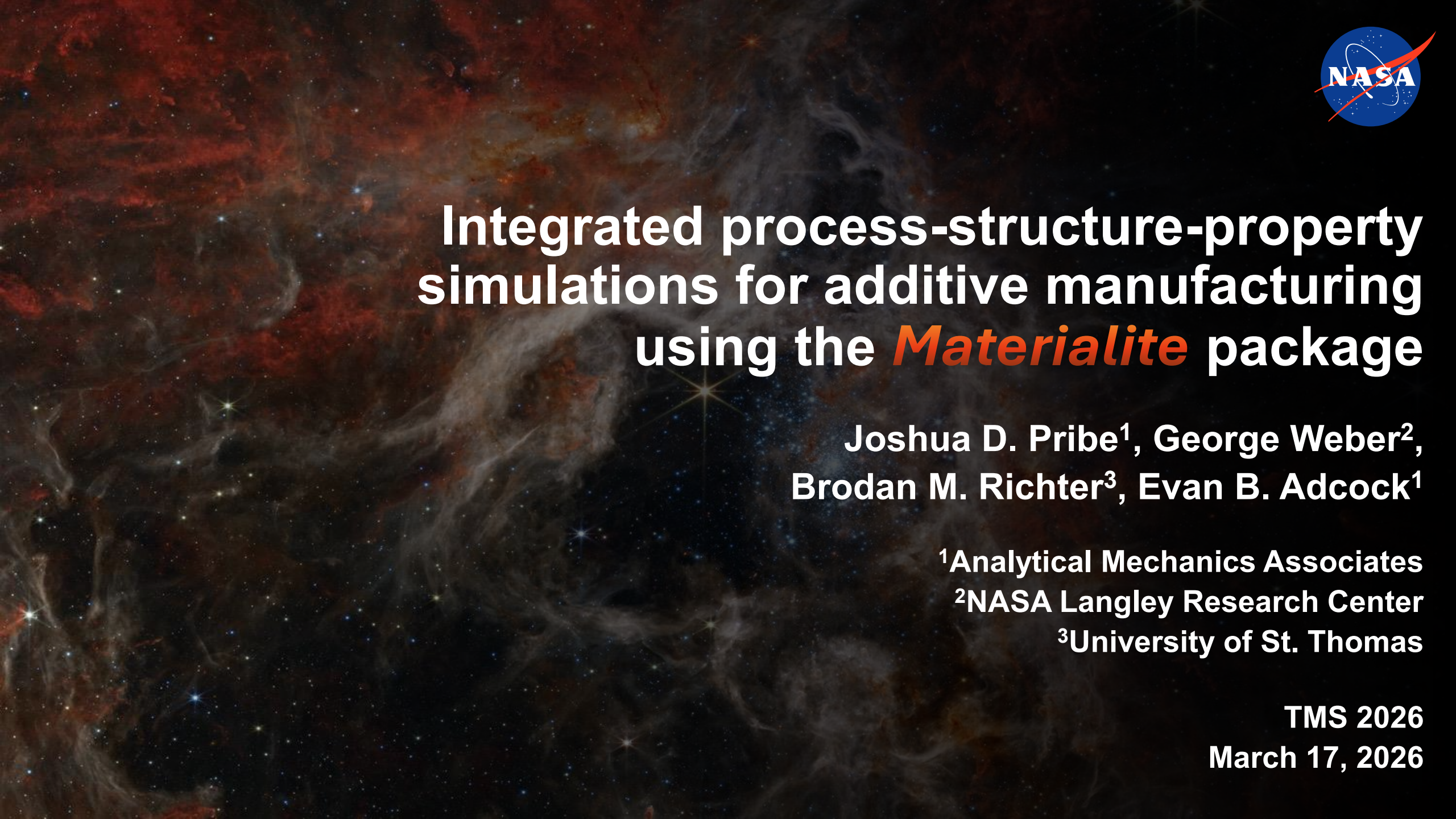


Integrated process-structure-property simulations for additive manufacturing using the *Materialite* package

Joshua D. Pribe[1], George Weber[2], Brodan M. Richter[3], Evan B. Adcock[1]

[1]Analytical Mechanics Associates
[2]NASA Langley Research Center
[3]University of St. Thomas

TMS 2026
March 17, 2026

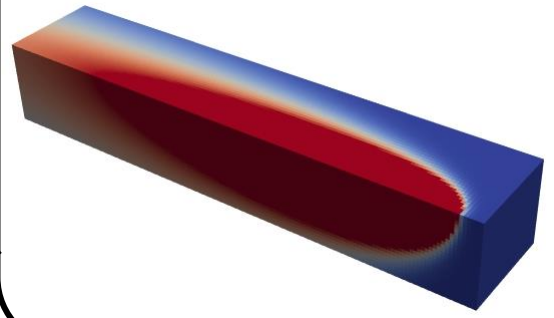
Process-structure-property (PSP) simulations

- PSP simulations: understand how additive manufacturing (AM) process variations affect mechanical behavior
- Goals:
  - Process parameter selection
  - New alloy and component design
  - Augment next-gen Q&C processes

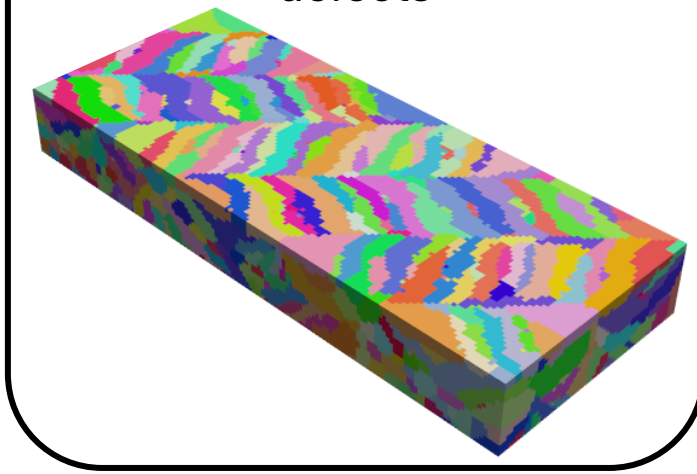
- Huge variety of modeling tools → difficult to couple simulation modalities in practice…

Q&C: qualification and certification

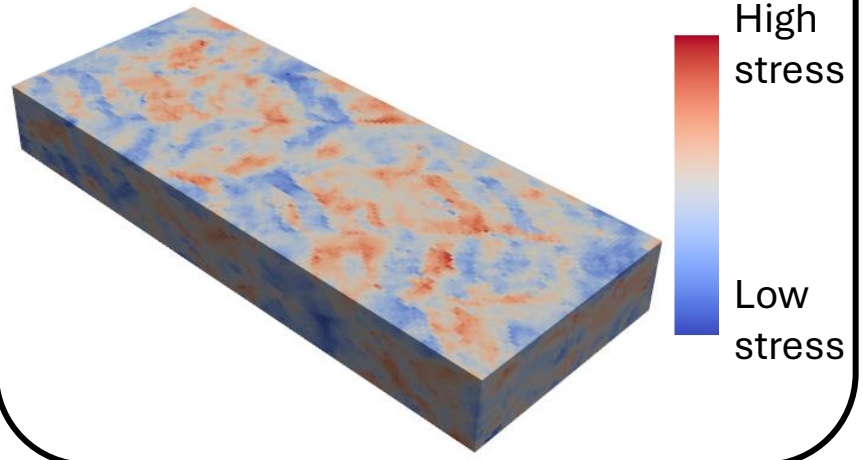
Thermal model: melt pool, temperature field



Process-structure model: solidification, microstructure, defects



Structure-property model: stress and strain fields, fatigue and fracture

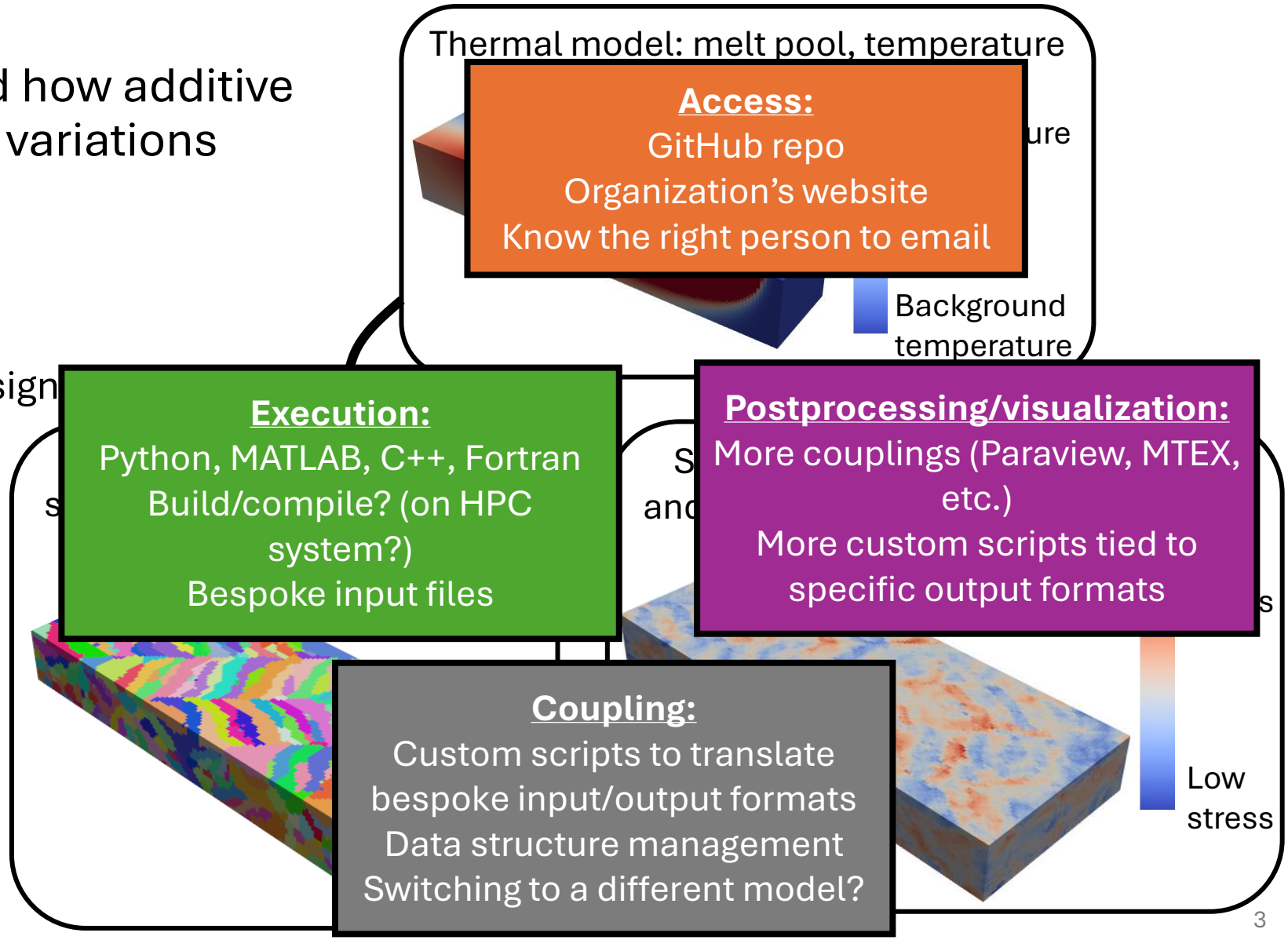


Process-structure-property (PSP) simulations

- PSP simulations: understand how additive manufacturing (AM) process variations affect mechanical behavior
- Goals:
  - Process parameter selection
  - New alloy and component design
  - Augment next-gen Q&C processes

- Huge variety of modeling tools → difficult to couple simulation modalities in practice…

HPC: high-performance computing



Materialite overview

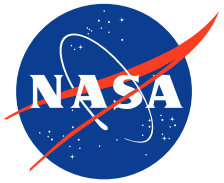
- Implemented in Python
- Unified data structure
- Streamlined workflows through method chaining
- Built-in plotting
- Simplified interfaces for operations like tensor math and crystallography
- Data exchange with established tools (e.g., DREAM.3D)
- **Enables fast testing and verification (e.g., Jupyter notebook)**

```python
from materialite import Material

material = Material()

material = (
    material.run(am_process_model)
    .run(grain_coarsening_model)
    .run(mechanical_model)
)

material.plot("grain")
```

```python
from materialite import Orientation,
    Order2SymmetricTensor
```

```python
material = import_dream3d("microstructure.dream3d", ...)
```

Outline



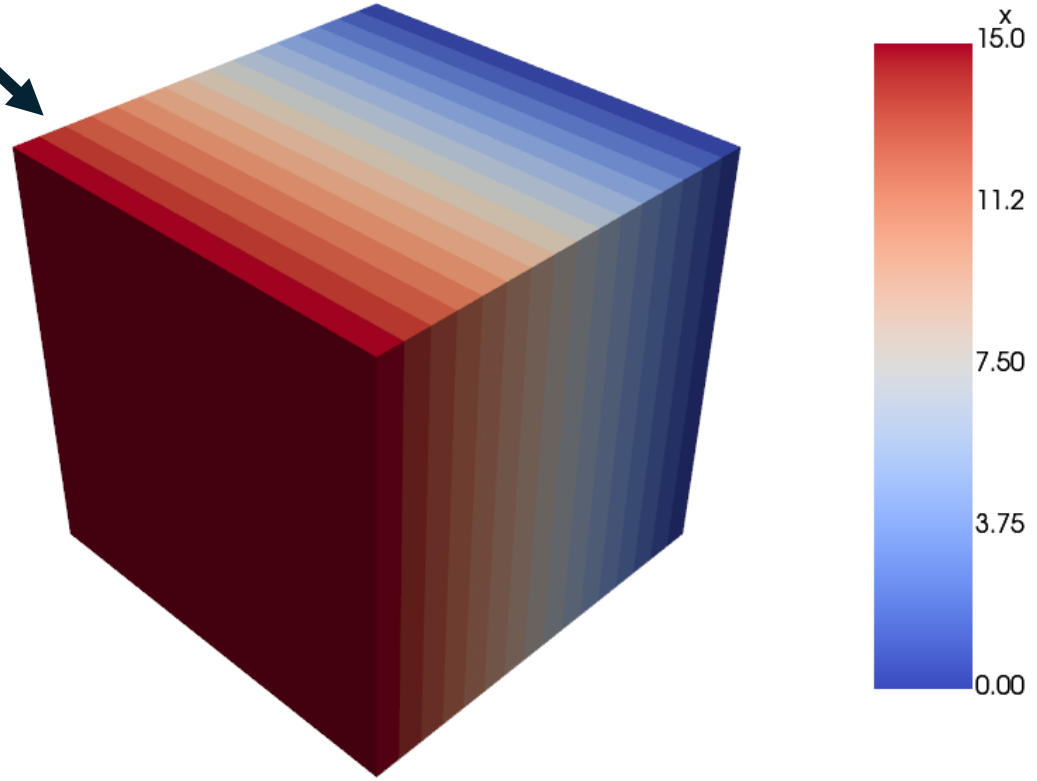
- ***Materialite*** features and workflow
  - `Material` basics
  - PSP simulations

- Example applications
  - Microstructures from synchrotron
  - Crystal plasticity calibration study

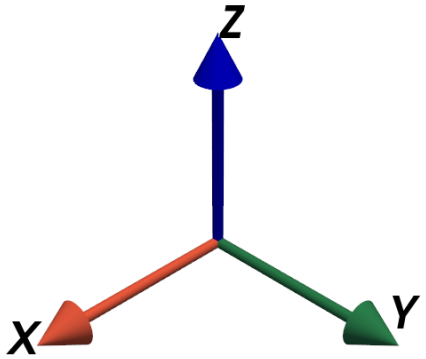
`Material` basics

- `Material` contains fields on a regular 3-D grid

```
from materialite import Material
material = Material()
material.plot("x")
```



$16 \times 16 \times 16$ grid, colored by *x* location



`Material` basics

- `Material` contains fields on a regular 3-D grid

```
from materialite import Material
material = Material()
material.get_fields()
```

Pandas `DataFrame` containing pointwise field information

	x	y	z	x_id	y_id	z_id
0	0	0	0	0	0	0
1	0	0	1	0	0	1
2	0	0	2	0	0	2
3	0	0	3	0	0	3
4	0	0	4	0	0	4
...	...	...	...	...	...	...
4091	15	15	11	15	15	11
4092	15	15	12	15	15	12
4093	15	15	13	15	15	13
4094	15	15	14	15	15	14
4095	15	15	15	15	15	15

4096 rows × 6 columns

Grid spacing; set using `spacing`

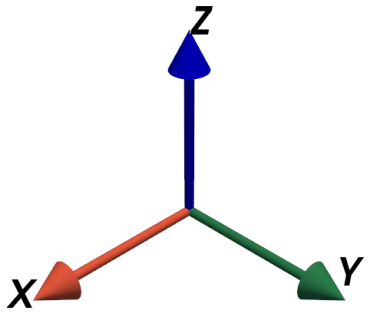
Number of points; set using `dimensions`

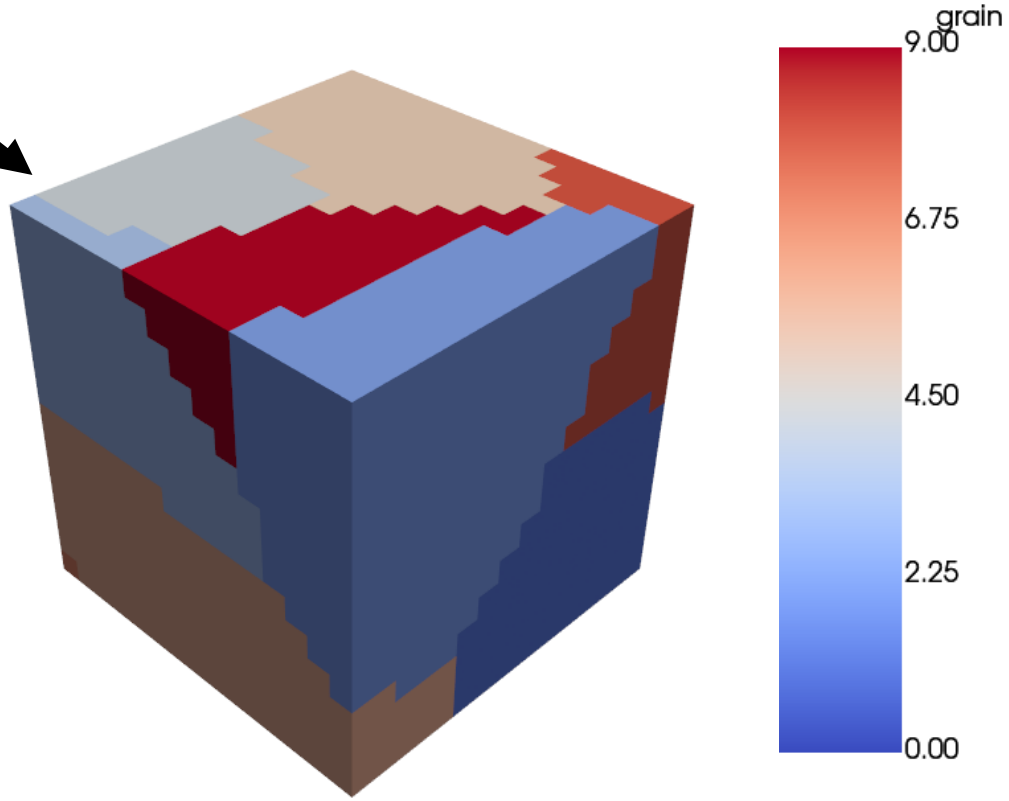
`Material` basics

- `Material` contains fields on a regular 3-D grid
- Most operations involve manipulating fields

```
from materialite import Material
material = Material()
material = material.create_voronoi(
    num_regions=10, label="grain")
material.plot("grain")
```

Voronoi tessellation with 10 grains

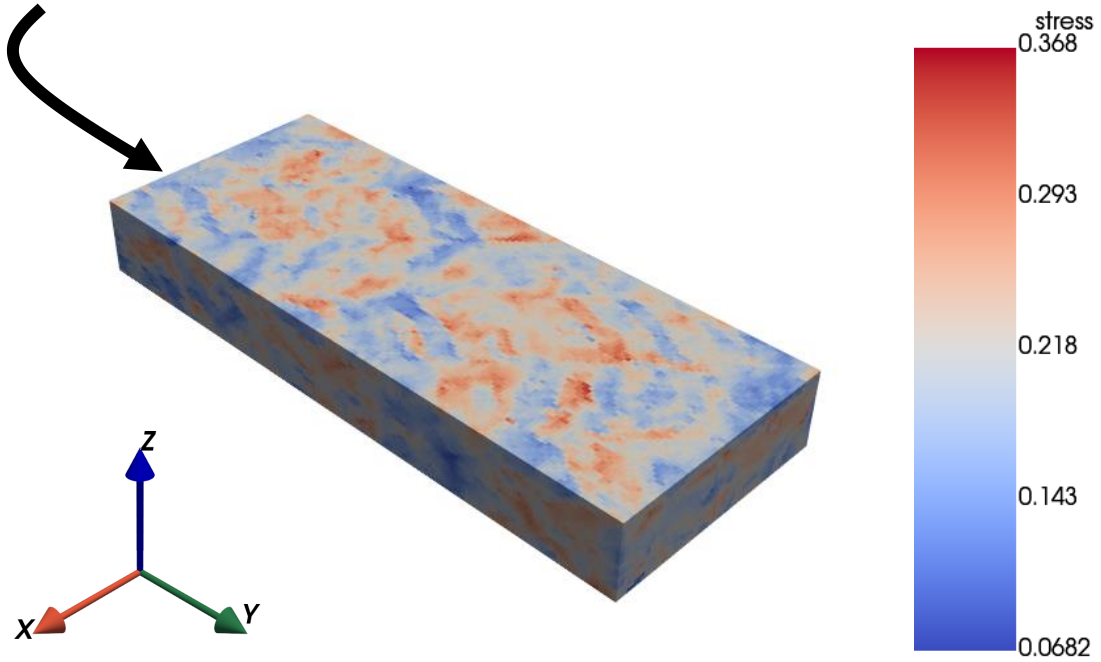




`Material` basics

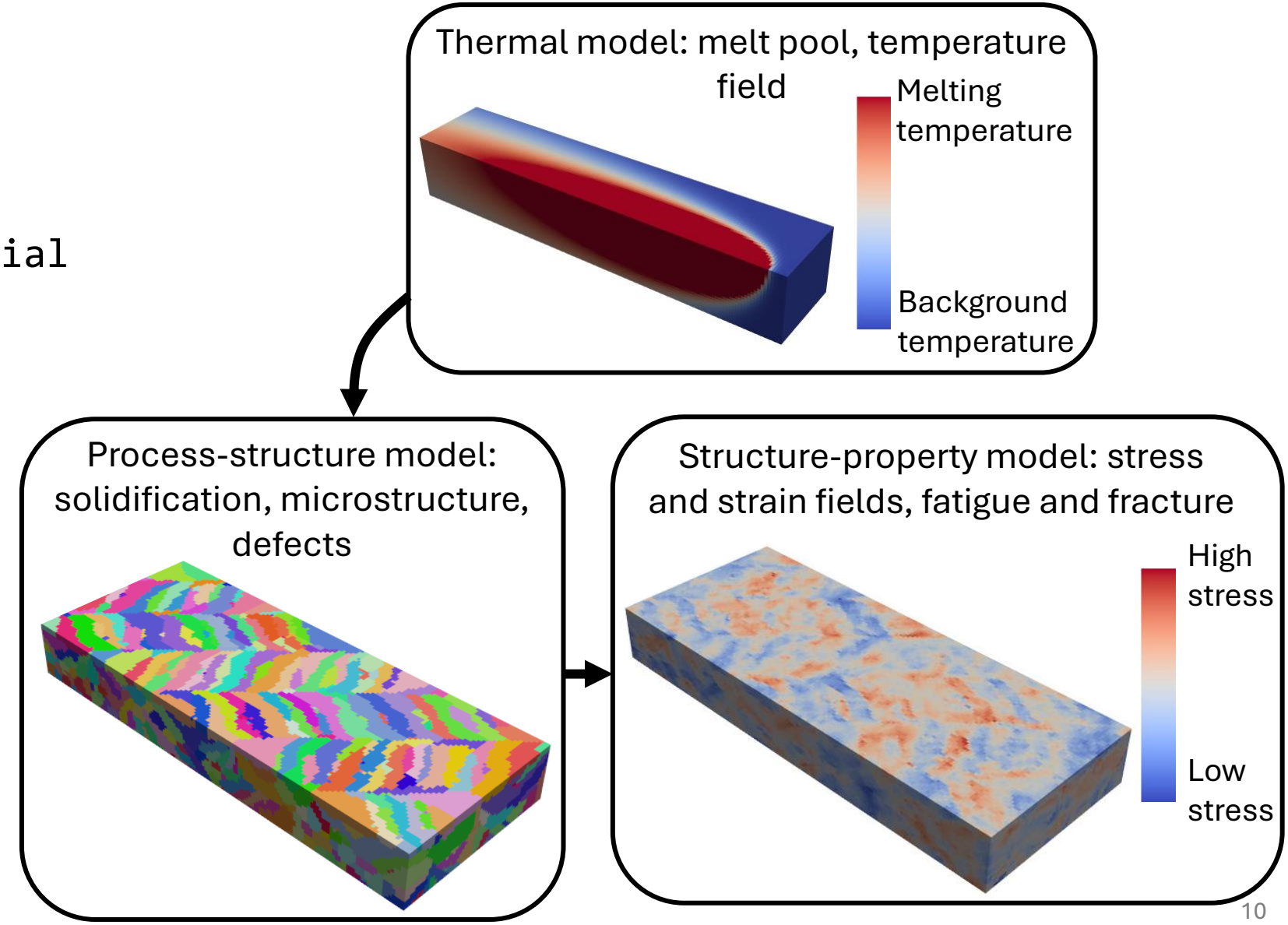
- `Material` contains fields on a regular 3-D grid
- Most operations involve manipulating fields
- Models return a new `Material` with updated fields
  - Allows for method chaining, similar to Pandas `DataFrames`

```
material = (
    Material()
    .create_voronoi(num_regions=10, label="grain")
    .run(am_process_model)
    .run(grain_coarsening_model)
    .run(mechanical_model)
)
material.plot("stress", component=1)
```



PSP simulations in *Materialite*

- Basic workflow:
  - Define models
  - Create a `Material`
  - Run the models on the `Material`
- Linkages become method chains
- Plots represent `Material` fields using `material.plot(…)`



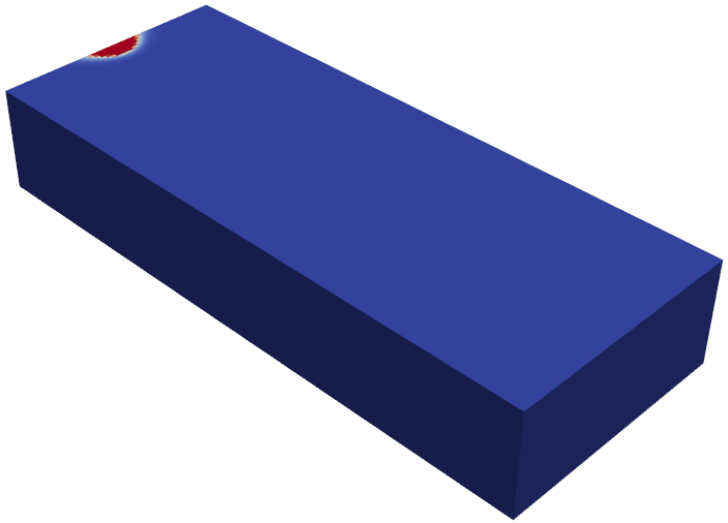
PSP simulations: Process-structure model

Process-structure model: Rosenthal equation + Potts Monte Carlo-based solidification and coarsening[1]

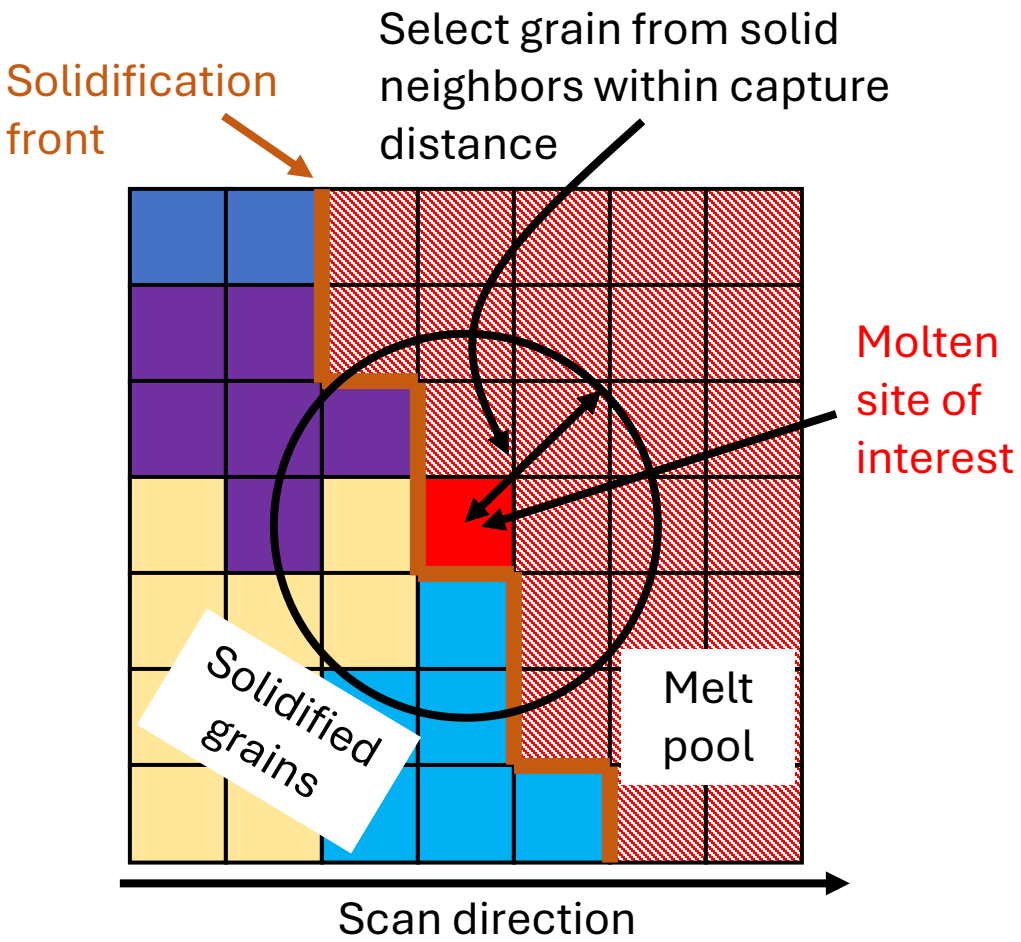
Melting temperature

Background temperature

Single scan animation



Solidification schematic



[1]T.M. Rodgers et al., Addit Manuf 41 (2021) 101953. https://doi.org/10.1016/j.addma.2021.101953

PSP simulations: Process-structure model

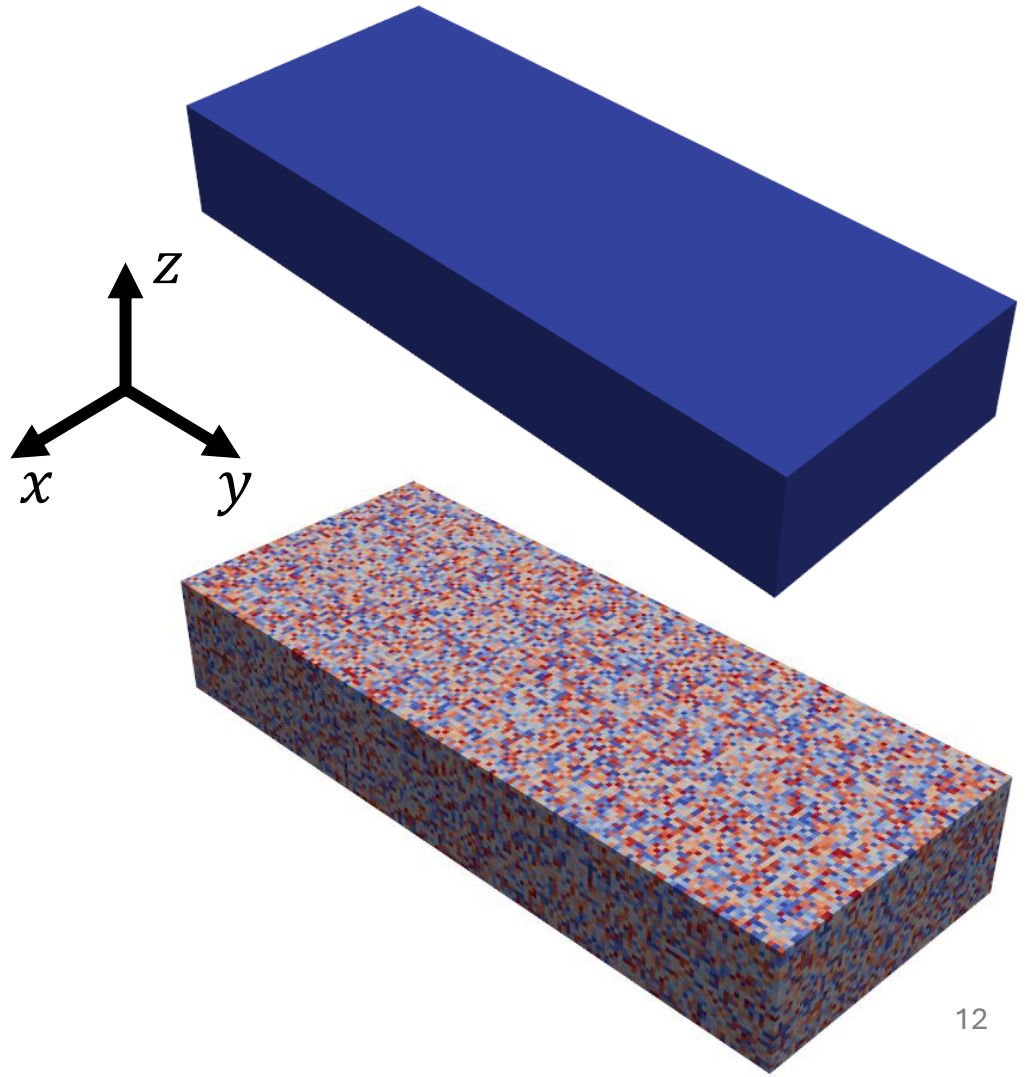
Define scan strategy using `LaserPath`

```python
from materialite.models import LaserPath
path1 = LaserPath.angled_raster_scan(
    angle=67.0, z_height=120.0e-6, **path_inputs
)

path2 = LaserPath.angled_raster_scan(
    angle=134.0, z_height=160.0e-6,
    start_time=path1.total_time, **path_inputs
)
laser_path = LaserPath.from_list([path1, path2])
```

Constant parameters (laser power, scan speed, domain size, etc.)

Put in a loop with defined layer thickness (z_height) and hatch angle (angle) to generate a larger build

Two-layer animation



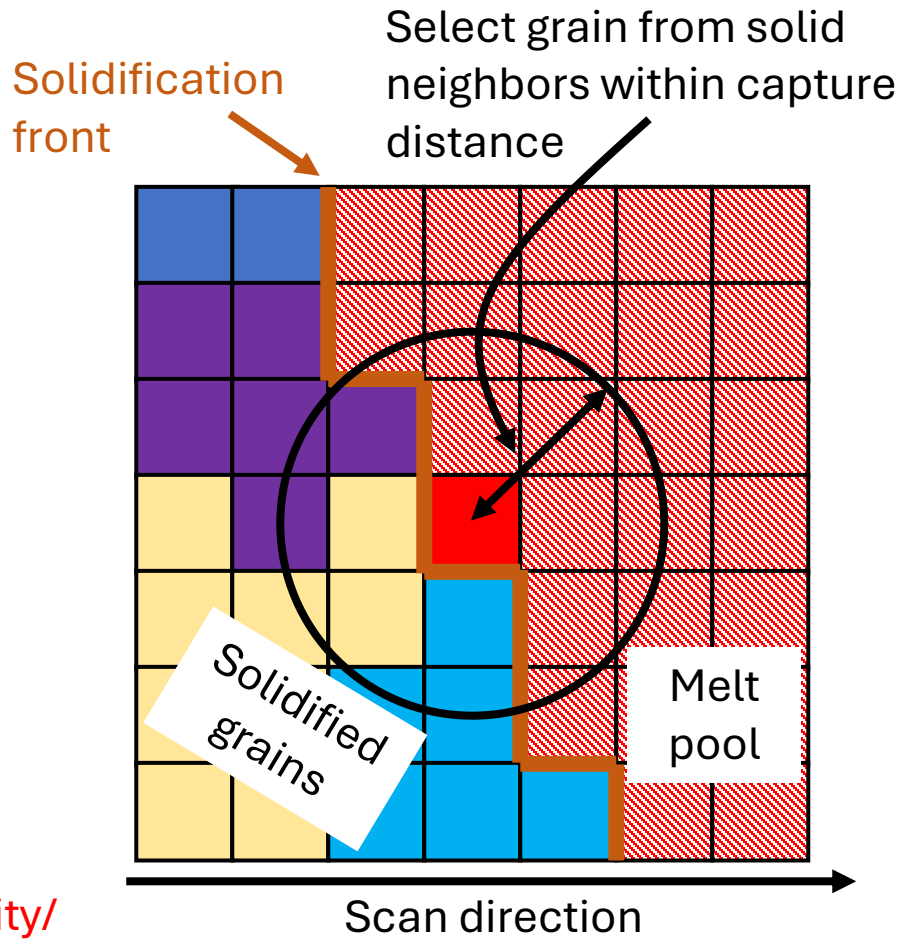
PSP simulations: Process-structure model

- Thermal + solidification model: `RosenthalSolidificationModel`
- Inputs: `LaserPath`, material properties, and time incrementation parameters

```
from materialite.models import RosenthalSolidificationModel

ps_model = RosenthalSolidificationModel(
    laser_path=laser_path,
    initial_temperature=300.0,
    thermophysical_properties=thermophysical_properties,
    solidification_properties=solidification_properties,
    time_step_duration=5.2e-6,
    output_times=output_times,
)
```

Dictionaries with thermal conductivity/ diffusivity, undercooling constants, etc.

Solidification schematic

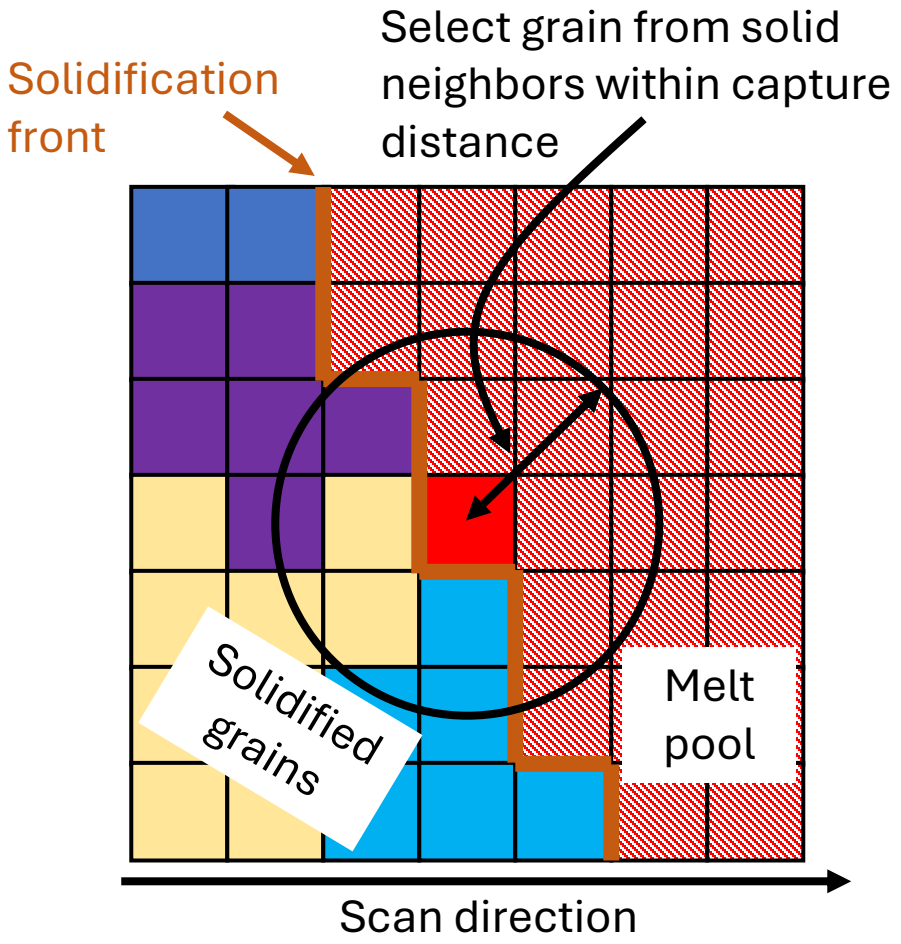


PSP simulations: Process-structure model

- Thermal + solidification model: `RosenthalSolidificationModel`
- Inputs: `LaserPath`, material properties, and time incrementation parameters
- More complex thermal models available (e.g., convolution-based methods)

"Sensitivity of Crystallographic Texture Strength to Enthalpy of Fusion and Freezing Range During Powder Bed Fusion Additive Manufacturing"
Wednesday, 9:05 AM
Speaker: Brodan Richter
Room: 23C
Location: SDCC

Solidification schematic

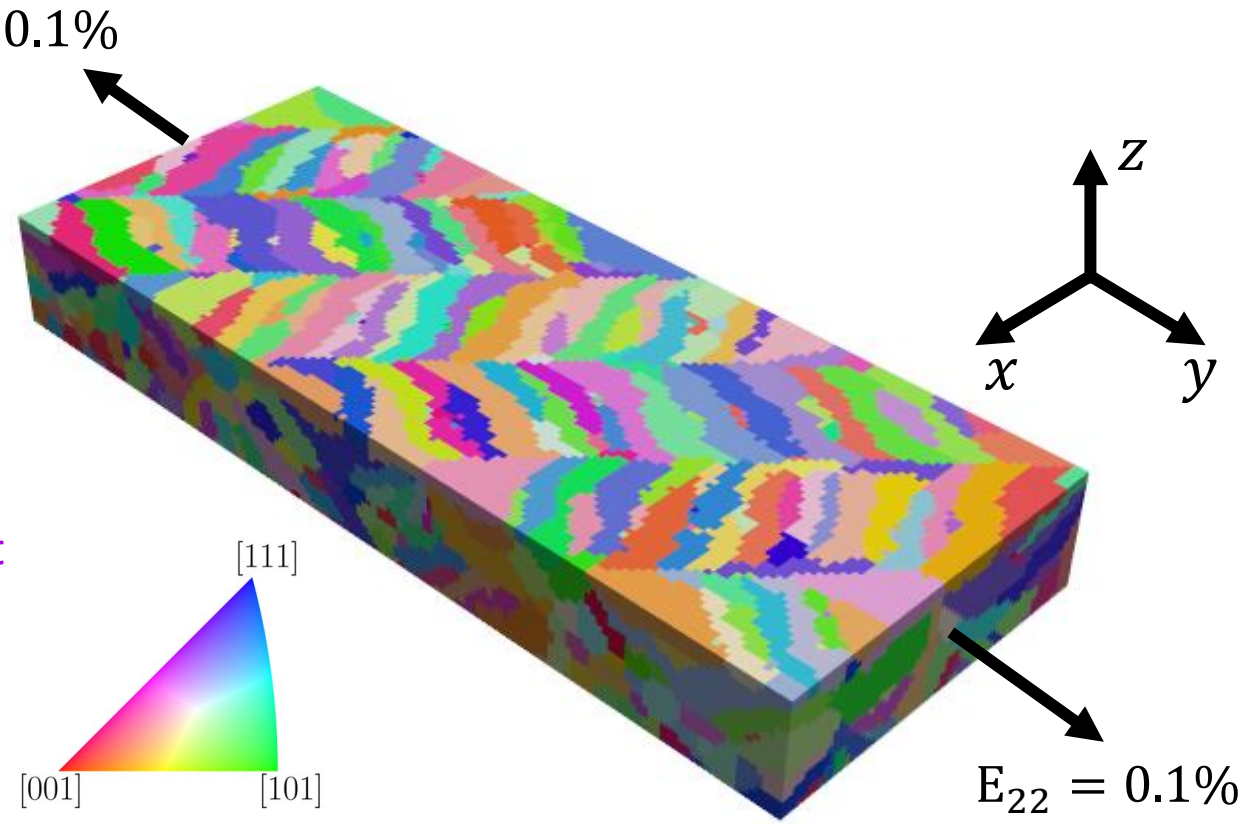


PSP simulations: Structure-property model

- Fast Fourier transform (FFT)-based mechanical solver[1,2]
- Uniaxial tension in *y* direction; all other mean stress components = 0
- In ***Materialite***: constitutive model, applied load, and solver

```
from materialite.models.small_strain_fft import
Elastic, LoadSchedule, SmallStrainFFT
```

Inverse pole figure map

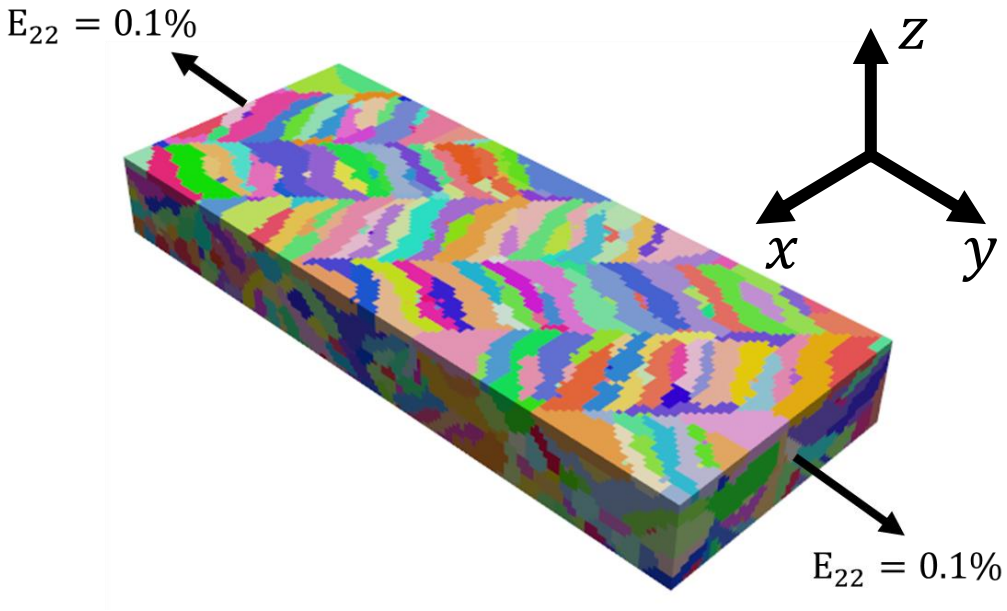


[1]J. Zeman et al., A finite element perspective on nonlinear FFT-based micromechanical simulations, Int J Numer Meth Engr 111 (2017) 903–926. https://doi.org/10.1002/nme.5481.

[2]T.W.J. de Geus et al., Finite strain FFT-based non-linear solvers made simple, Comput Methods Appl Mech Eng 318 (2017) 412–430. https://doi.org/10.1016/j.cma.2016.12.032.

PSP simulations: Structure-property model

Constitutive model (`Elastic`) and applied loads (`LoadSchedule`) are inputs to `SmallStrainFFT`



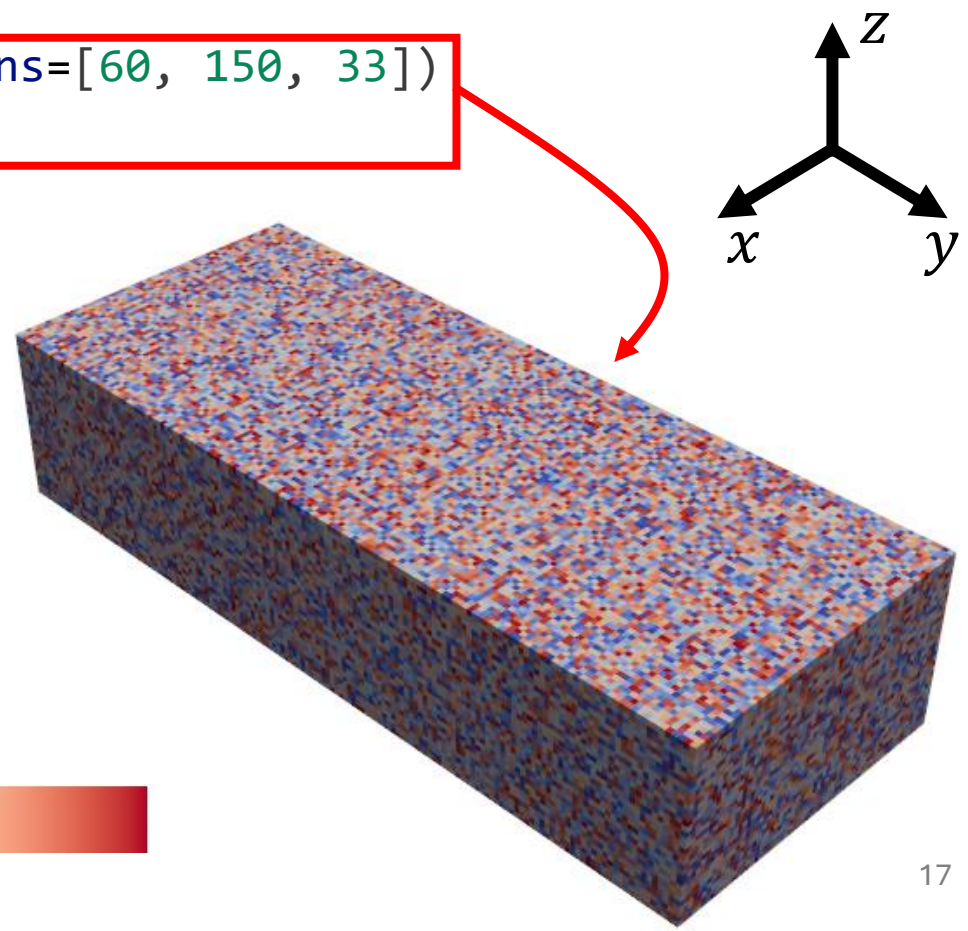

```
stiffness = Order4SymmetricTensor.from_cubic_constants(
    C11=243.3, C12=156.7, C44=117.8
)
elastic_model = Elastic(stiffness)
load_schedule = LoadSchedule.from_constant_uniaxial_strain_rate(
    magnitude=1.0, direction="y"
)
sp_model = SmallStrainFFT(
    load_schedule=load_schedule, end_time=0.001, constitutive_model=elastic_model
)
```

Built-in constructors for common cases; can supply functions for arbitrary load history

PSP simulations: Results

Put everything together:

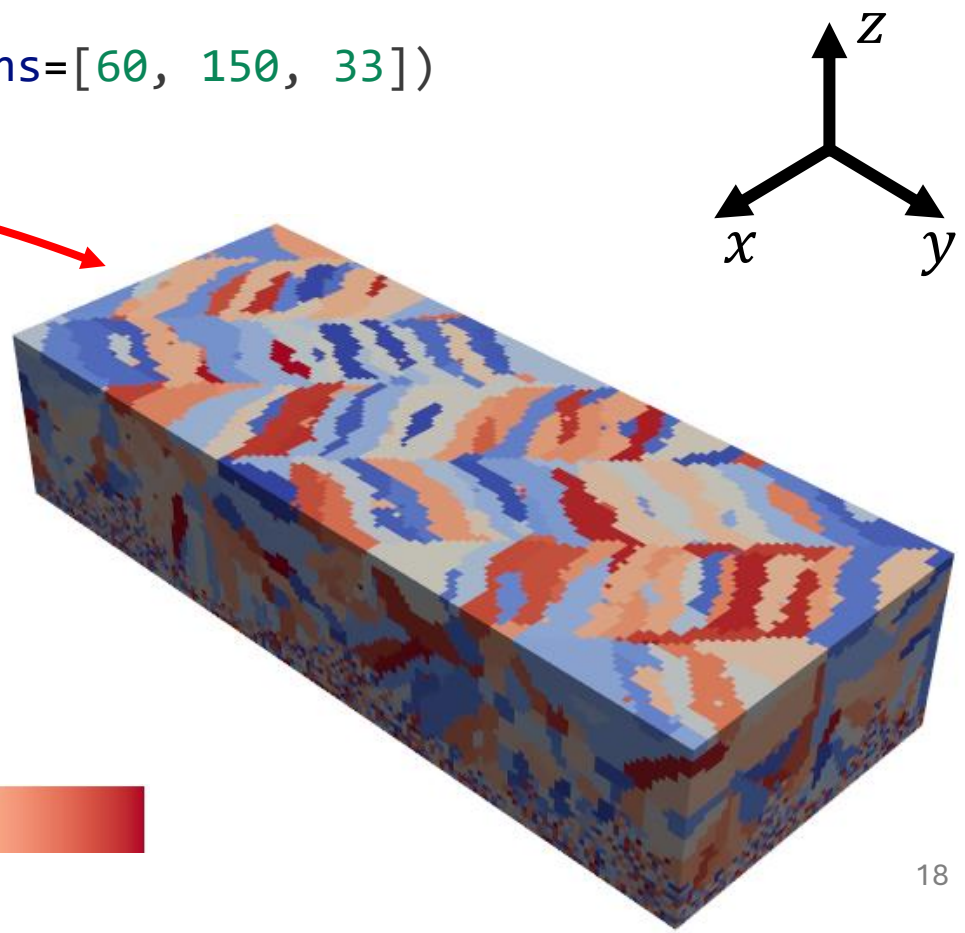
```
material = (
    Material(spacing=[5.0e-6, 5.0e-6, 5.0e-6], dimensions=[60, 150, 33])
    .create_random_integer_field("grain", 1, 10000)
    .run(ps_model)
    .crop_by_range(z_range=[70.0e-6, np.inf])
    .assign_random_orientations(region_label="grain")
    .run(add_ipf_colors_field,
        specimen_frame_direction=[0, 1, 0])
    .run(sp_model)
)
```



PSP simulations: Results

Put everything together:

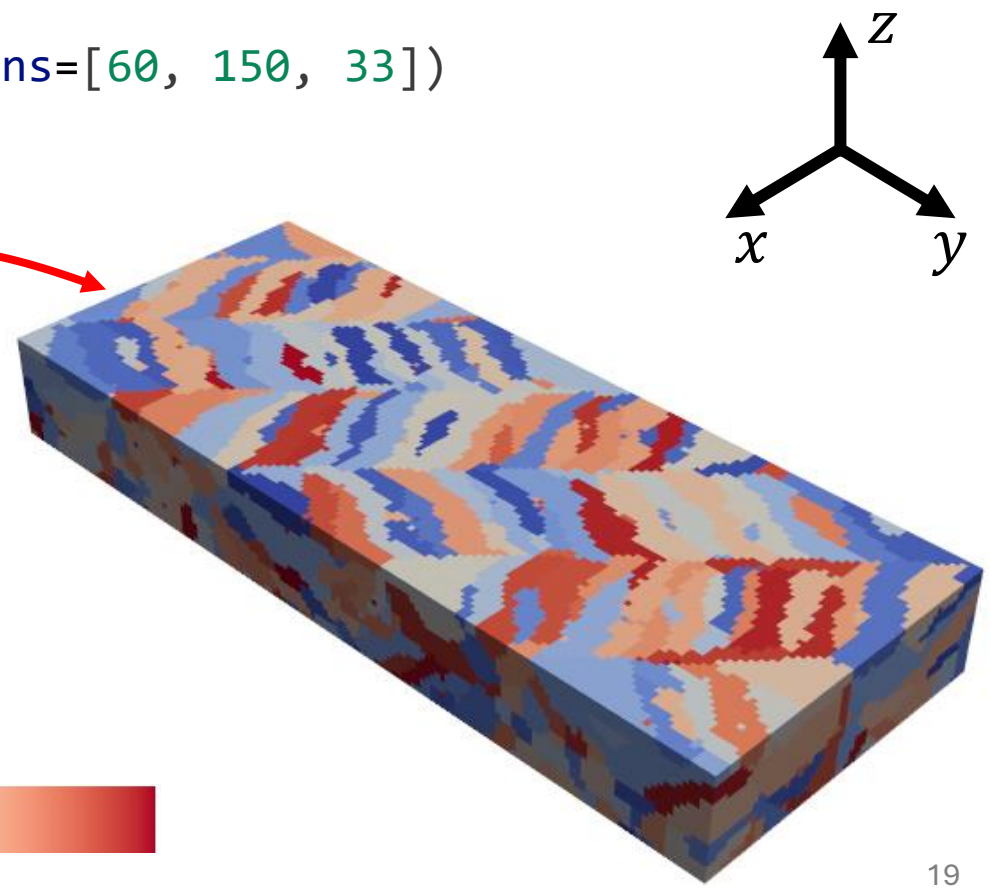
```
material = (
    Material(spacing=[5.0e-6, 5.0e-6, 5.0e-6], dimensions=[60, 150, 33])
    .create_random_integer_field("grain", 1, 10000)
    .run(ps_model)
    .crop_by_range(z_range=[70.0e-6, np.inf])
    .assign_random_orientations(region_label="grain")
    .run(add_ipf_colors_field,
        specimen_frame_direction=[0, 1, 0])
    .run(sp_model)
)
```



PSP simulations: Results

Put everything together:

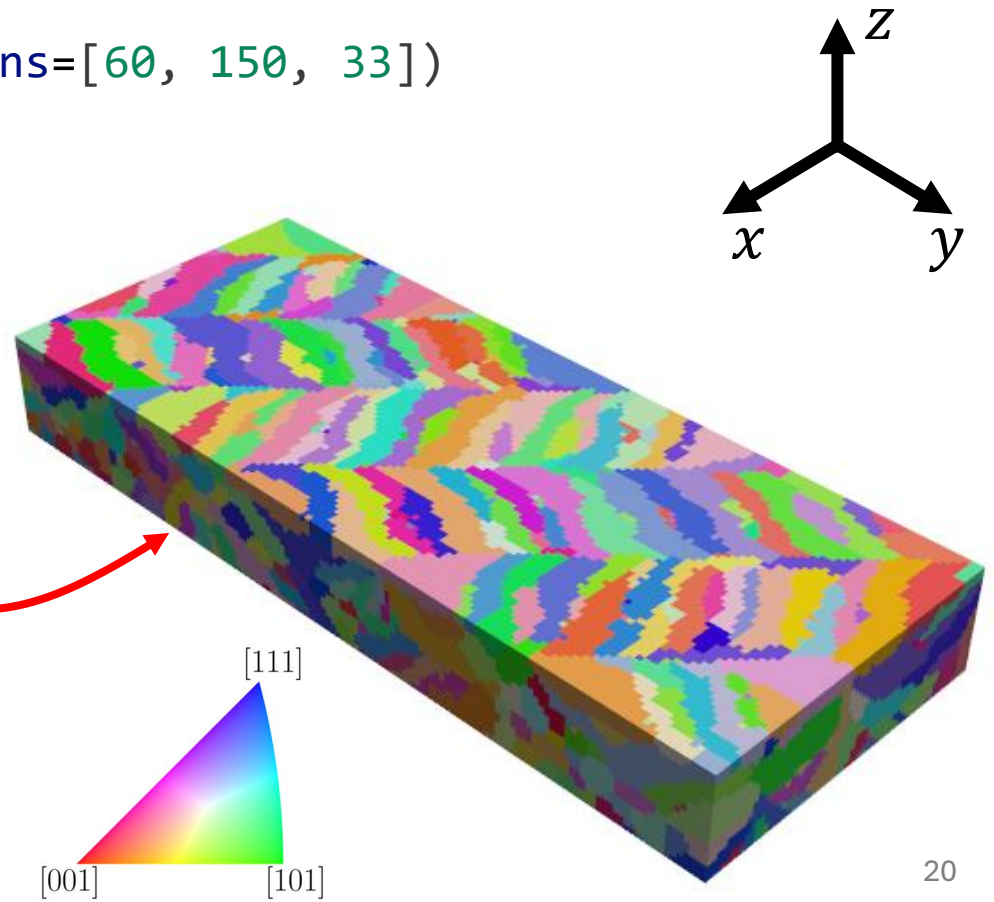
```
material = (
    Material(spacing=[5.0e-6, 5.0e-6, 5.0e-6], dimensions=[60, 150, 33])
    .create_random_integer_field("grain", 1, 10000)
    .run(ps_model)
    .crop_by_range(z_range=[70.0e-6, np.inf])
    .assign_random_orientations(region_label="grain")
    .run(add_ipf_colors_field,
        specimen_frame_direction=[0, 1, 0])
    .run(sp_model)
)
```



PSP simulations: Results

Put everything together:

```
material = (
    Material(spacing=[5.0e-6, 5.0e-6, 5.0e-6], dimensions=[60, 150, 33])
    .create_random_integer_field("grain", 1, 10000)
    .run(ps_model)
    .crop_by_range(z_range=[70.0e-6, np.inf])
    .assign_random_orientations(region_label="grain")
    .run(add_ipf_colors_field,
        specimen_frame_direction=[0, 1, 0])
    .run(sp_model)
)
```

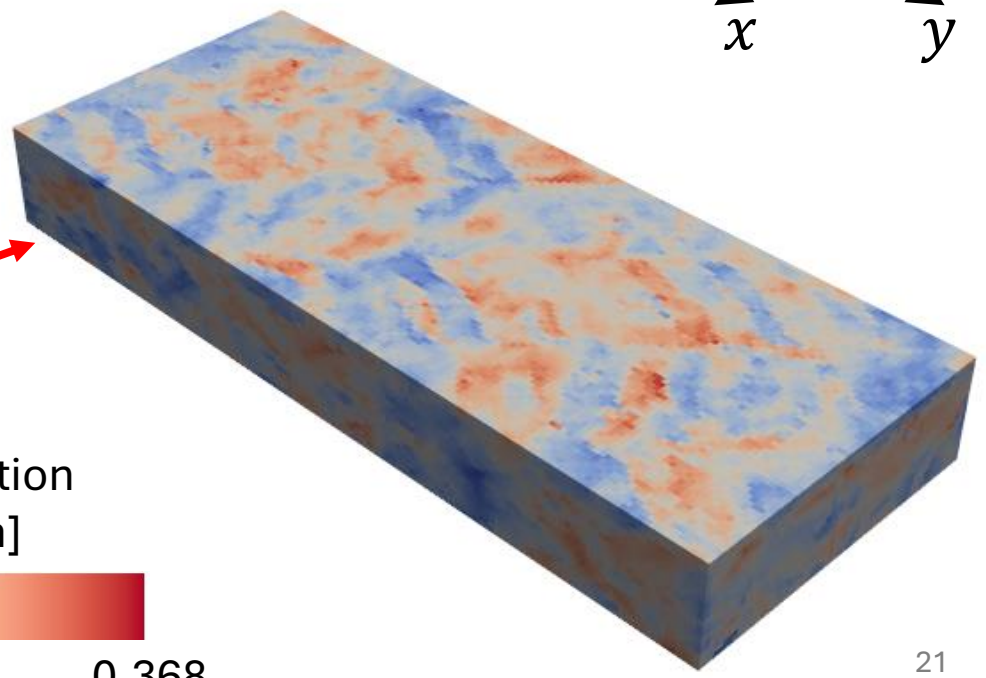


`add_ipf_colors_field`: ***Materialite*** utility function to add IPF (inverse pole figure) color as a `Material` field

PSP simulations: Results

Put everything together:

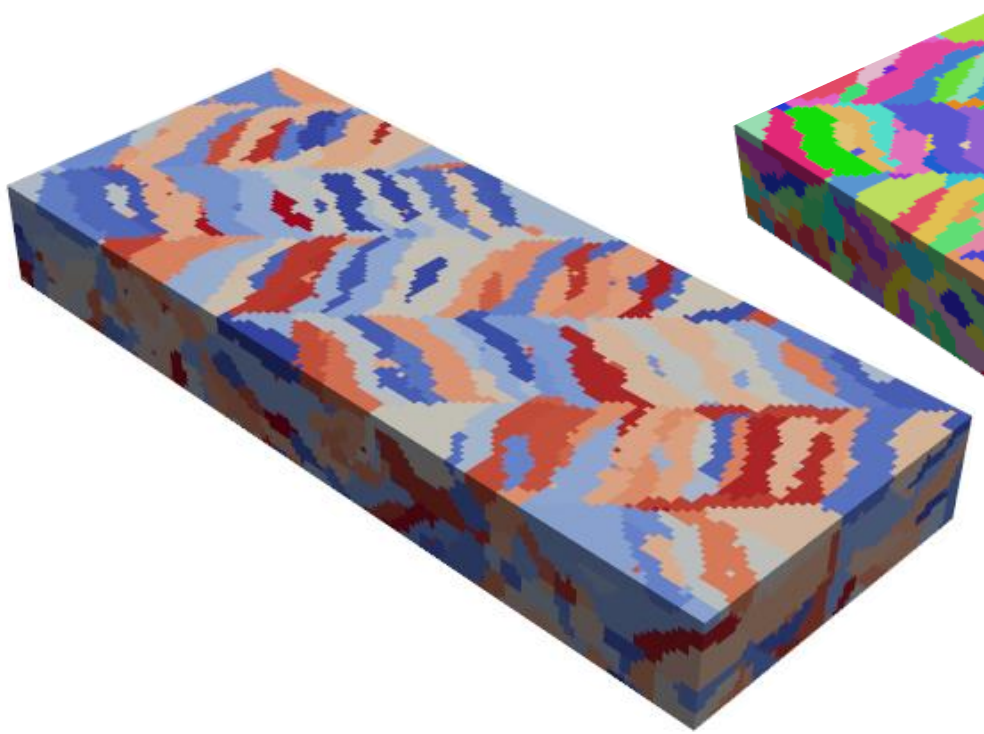
```
material = (
    Material(spacing=[5.0e-6, 5.0e-6, 5.0e-6], dimensions=[60, 150, 33])
    .create_random_integer_field("grain", 1, 10000)
    .run(ps_model)
    .crop_by_range(z_range=[70.0e-6, np.inf])
    .assign_random_orientations(region_label="grain")
    .run(add_ipf_colors_field,
        specimen_frame_direction=[0, 1, 0])
    .run(sp_model)
)
```



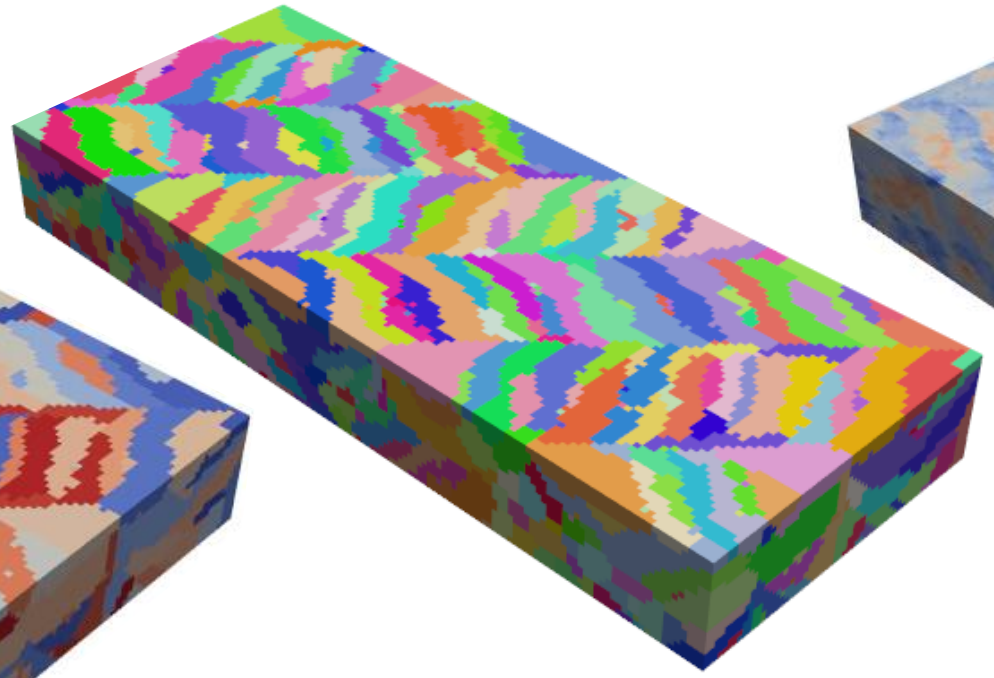
PSP simulations: Plotting

All plots come from one-line commands and render in a Jupyter notebook

```
material.plot("grain")
```

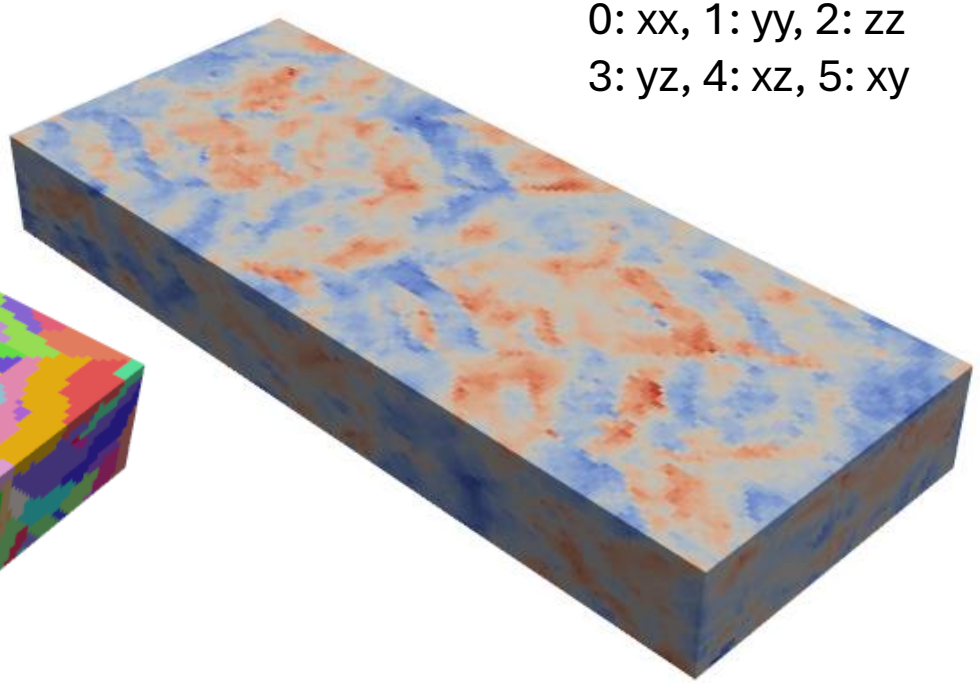


```
material.plot("ipf_color",
    kind="ipf_map")
```



```
material.plot(“stress",
    component=1)
```

0: xx, 1: yy, 2: zz
3: yz, 4: xz, 5: xy

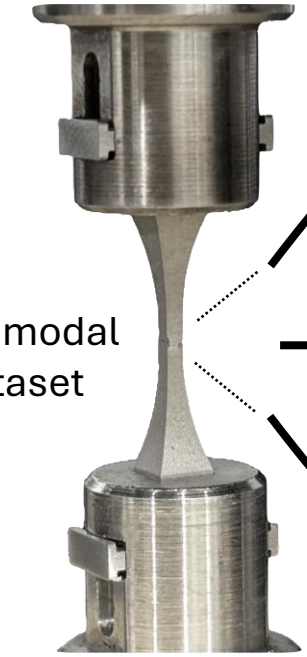


Outline

- ***Materialite*** features and workflow
  - `Material` basics
  - Process-structure-property simulations

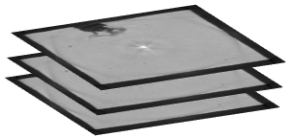
- Example applications
  - Microstructures from synchrotron
  - Crystal plasticity calibration study

Microstructures from synchrotron

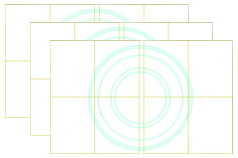


Multimodal dataset

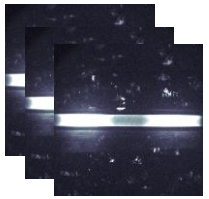
In-situ fatigue loading experiment at CHESS



X-ray computed tomography



Far-field high energy diffraction



Near-field high energy diffraction

```
material.apply(<scikit-image>)
```

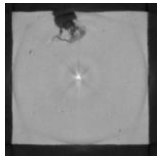
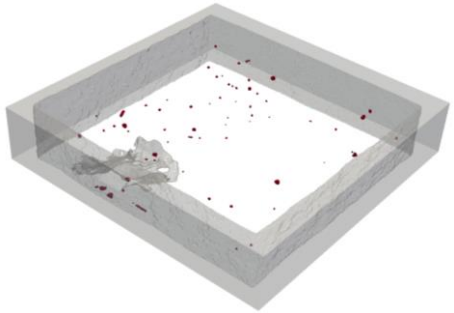
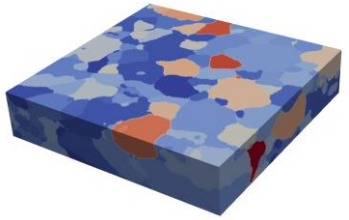


Image processing and segmentation



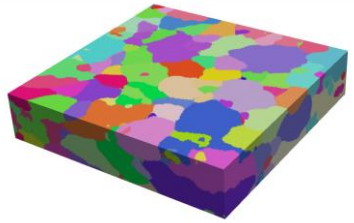
Crack and pore morphology

```
material = import_hexrd_grains(...)
```



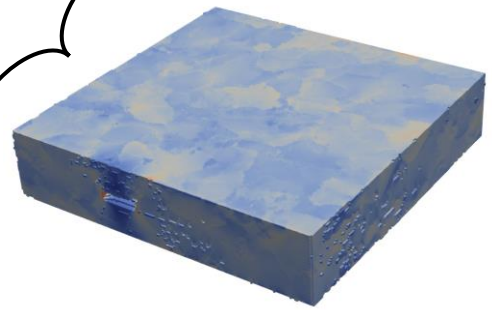
Grain-averaged strain

```
material = import_hexrd_near_field(...)
```



Grain orientation

Data Registration



Micromechanical simulation

"Integrating 3D Synchrotron Crack Growth Data With Crystal Plasticity Simulations Using the Open-Source Materialite Framework"
Today (Tuesday), 2:30 PM
Speaker: George Weber
Room: Sapphire 411B, Location: Hilton

CHESS: Cornell High-Energy Synchrotron Source

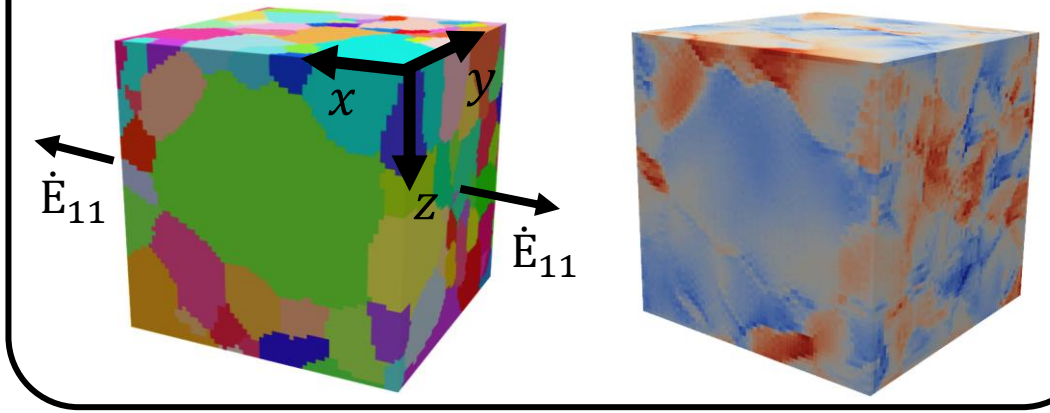
Crystal plasticity calibration

- Goal: probabilistic calibration of crystal plasticity parameters using Bayesian inference
- Challenge: uncertainty and non-uniqueness with global measurement data only
- Solution: use both global and local data in the calibration

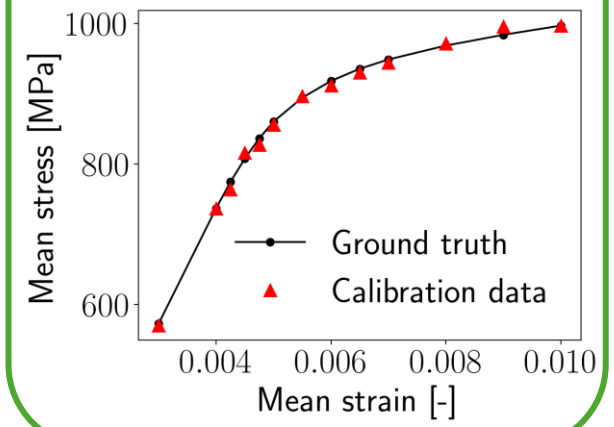
Calibration parameters

Rate exponent, m Initial CRSS, g_0 Hardening coefficient, H CRSS ratio, g_0 / g_1 (g_1 is the saturation stress)

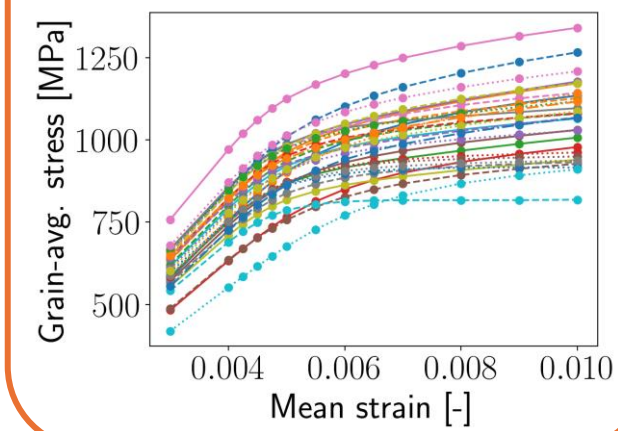
Crystal plasticity simulations (***Materialite***)



Global data (stress-strain curve)



Local data (grain-average stresses)

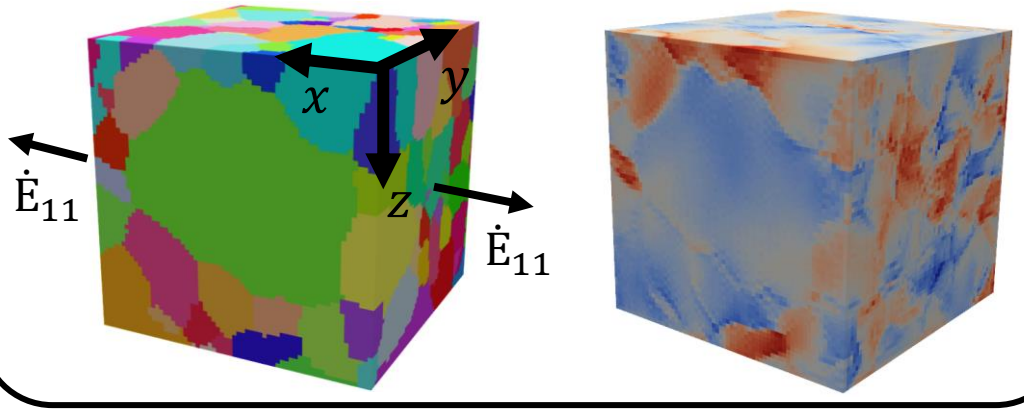


CRSS: critical resolved shear stress

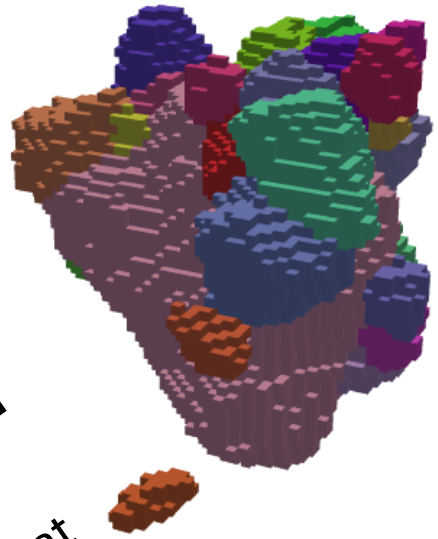
Crystal plasticity calibration: Procedure

- Generate synthetic data

Crystal plasticity simulations (***Materialite***)

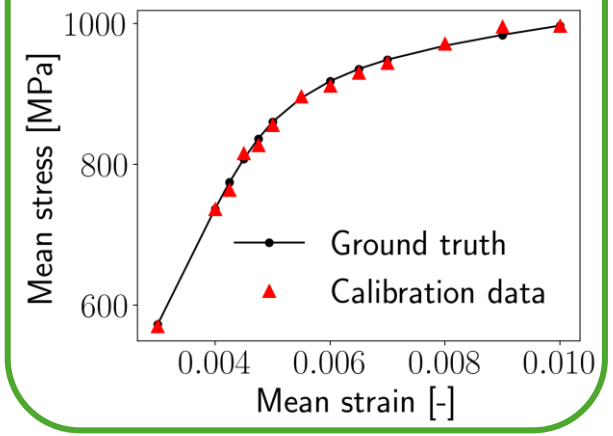


32 grains of interest

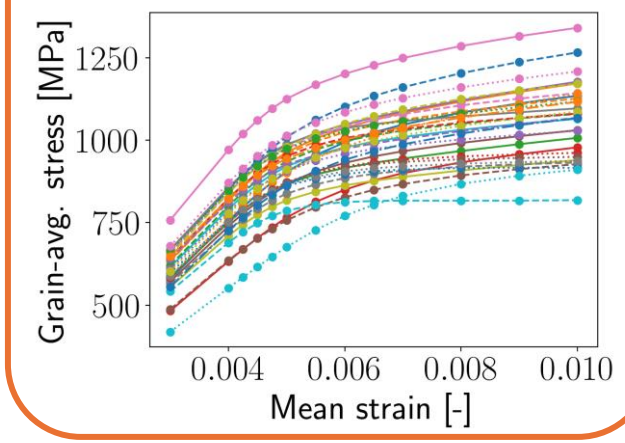


Extract grain-average stresses at each time point

Synthetic global data



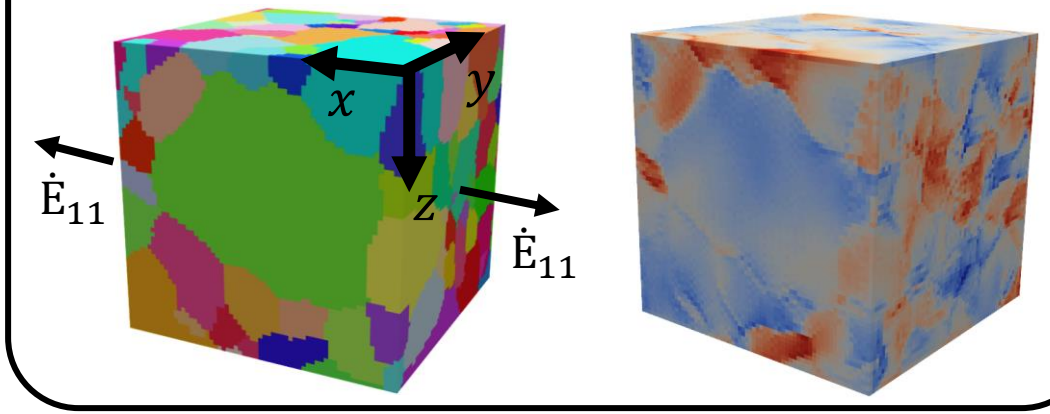
Synthetic local data



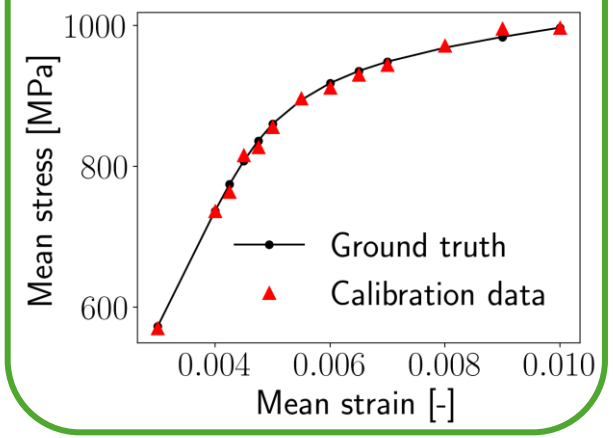
Crystal plasticity calibration: Procedure

- Generate synthetic data

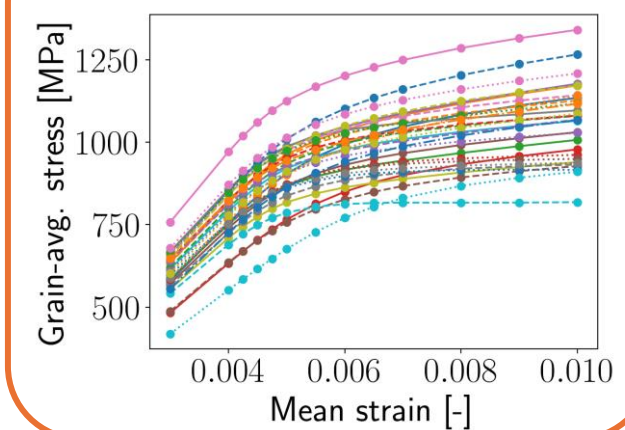
Crystal plasticity simulations (***Materialite***)



Synthetic global data



Synthetic local data



Constitutive model for `SmallStrainFFT` simulations

```
constitutive_model = ElasticViscoplastic(
    stiffness=stiffness,
    slip_systems=SlipSystem.octahedral(),
    reference_slip_rate=0.004,
    rate_exponent=m,
    slip_resistance=g0,
    hardening_function=armstrong_frederick,
    hardening_parameters={
        "direct_hardening": H,
        "dynamic_recovery": H_d,
    },
)
```

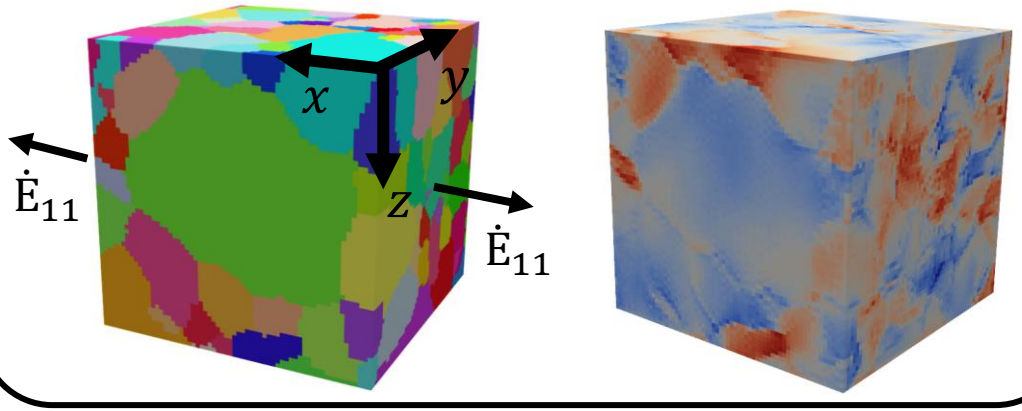
Note: dynamic recovery coefficient inferred from saturation stress

Crystal plasticity calibration: Procedure

- Generate synthetic data
- Parallelized sequential Monte Carlo calibration

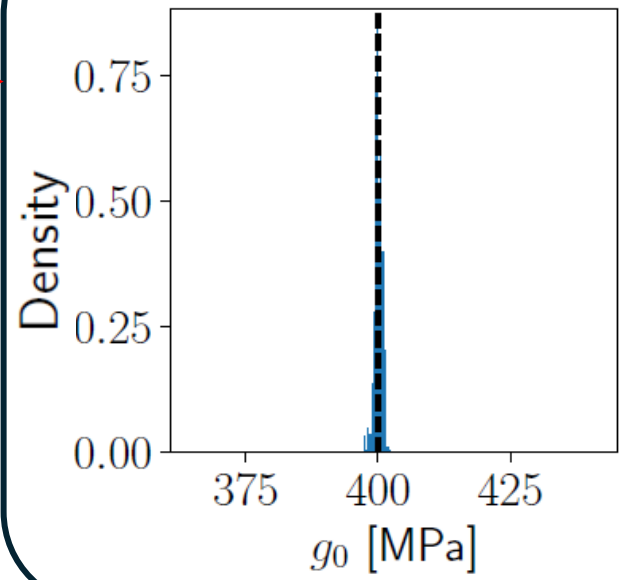
Sequential Monte Carlo calibration

Crystal plasticity simulations (***Materialite***)

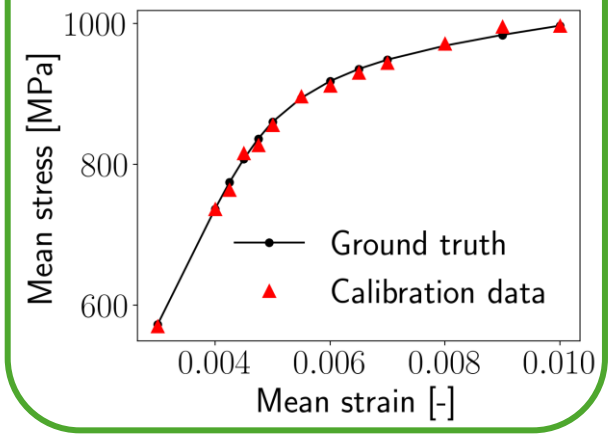


1,288 samples; ~3.75 days

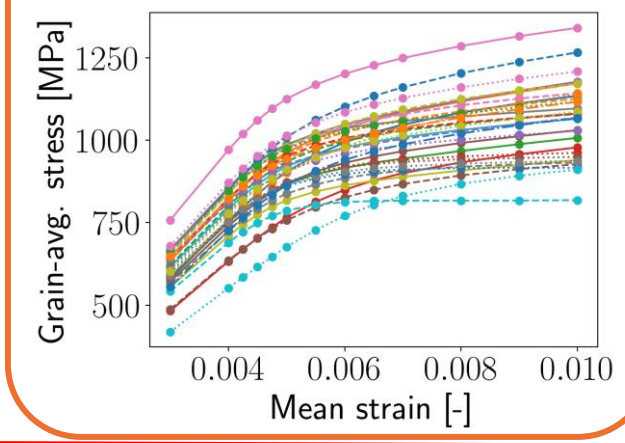
Posterior distributions



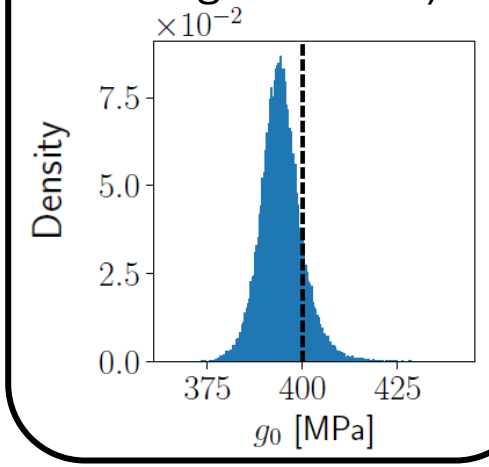
Synthetic global data



Synthetic local data



Priors (from surrogate with global data)

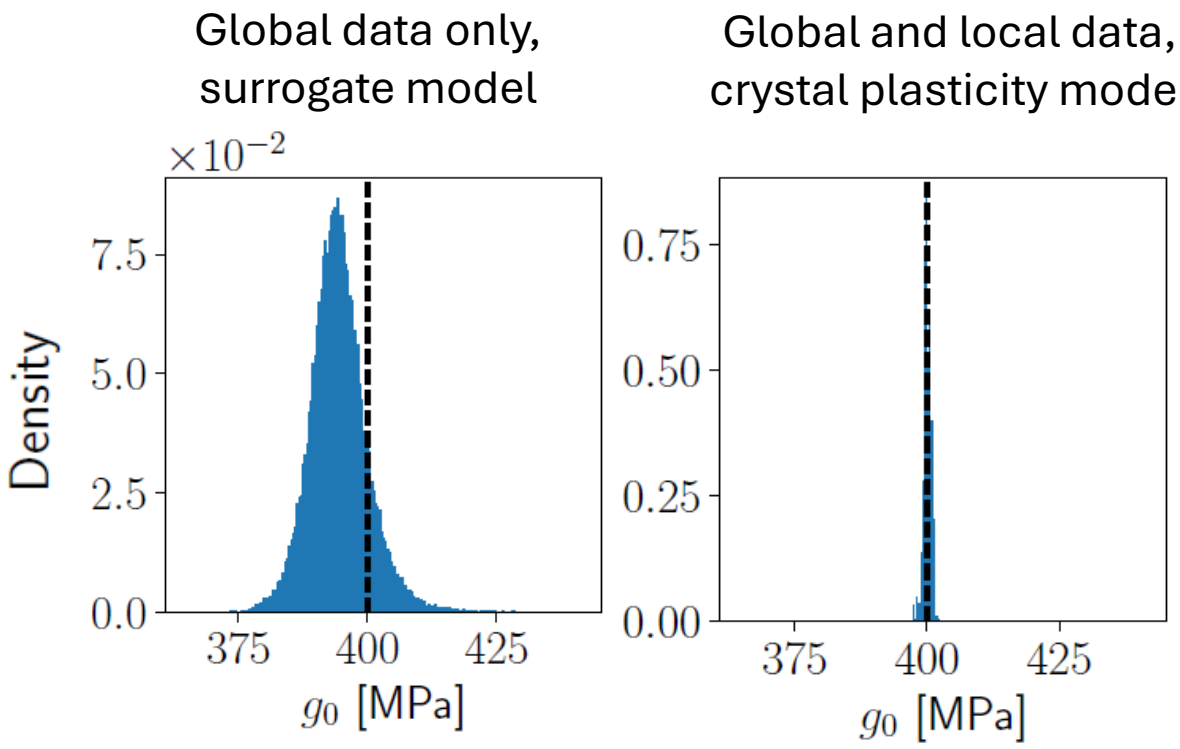


Crystal plasticity calibration: Outcomes

Key outcomes:

- Local measurement data increases accuracy and reduces uncertainty

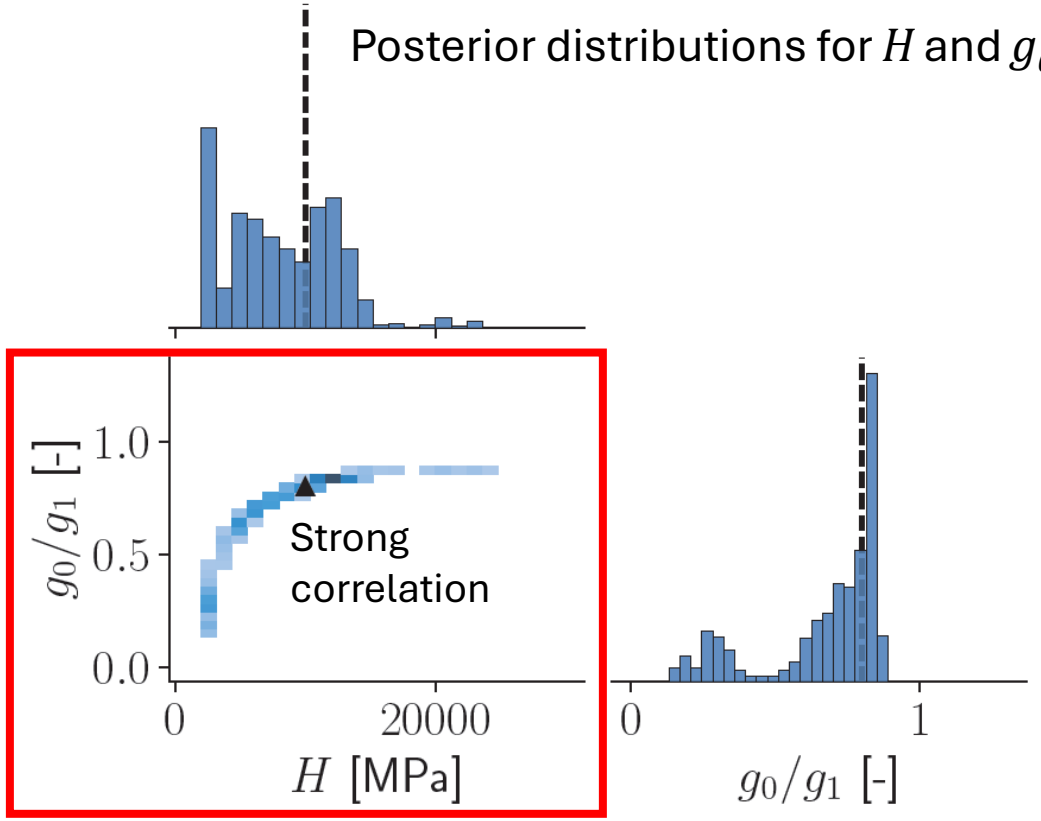
Posterior distributions for initial CRSS, g_0



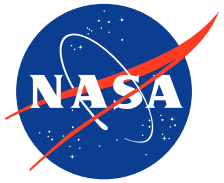
Crystal plasticity calibration: Outcomes

Key outcomes:

- Local measurement data increases accuracy and reduces uncertainty
- Correlations between parameters → identifiability challenges, even with local measurement data



Crystal plasticity calibration: Outcomes



Key outcomes:

- Local measurement data increases accuracy and reduces uncertainty
- Correlations between parameters → identifiability challenges, even with local measurement data
- Quality-of-life features in ***Materialite*** simplify the problem setup
  - Adaptive time step in FFT solver
  - Outputs at arbitrary user-specified times
  - Easily extract indices corresponding to each grain of interest and use to compute grain-average stresses
- Sequential Monte Carlo enables tractable calibration with the full crystal plasticity model
  - Stage 2 required ~120,000 simulations
  - Parallelized sampling → ~3.75 days on 644 cores versus ~2400 days of CPU time

Summary

- Coupling models/codes for PSP simulations can be challenging and time consuming
  - Common roadblocks: installation, compilation, linking bespoke data structures and output formats, verification

- ***Materialite*** unifies operations with the `Material` data structure
  - Enables method chaining, built-in plotting, and **rapid testing and verification**

- Applications to real problems:
  - Synchrotron microstructures: import data, visualize microstructure and porosity, run a mechanical simulation
  - Crystal plasticity calibration: run large numbers of simulations in parallel for sequential Monte Carlo

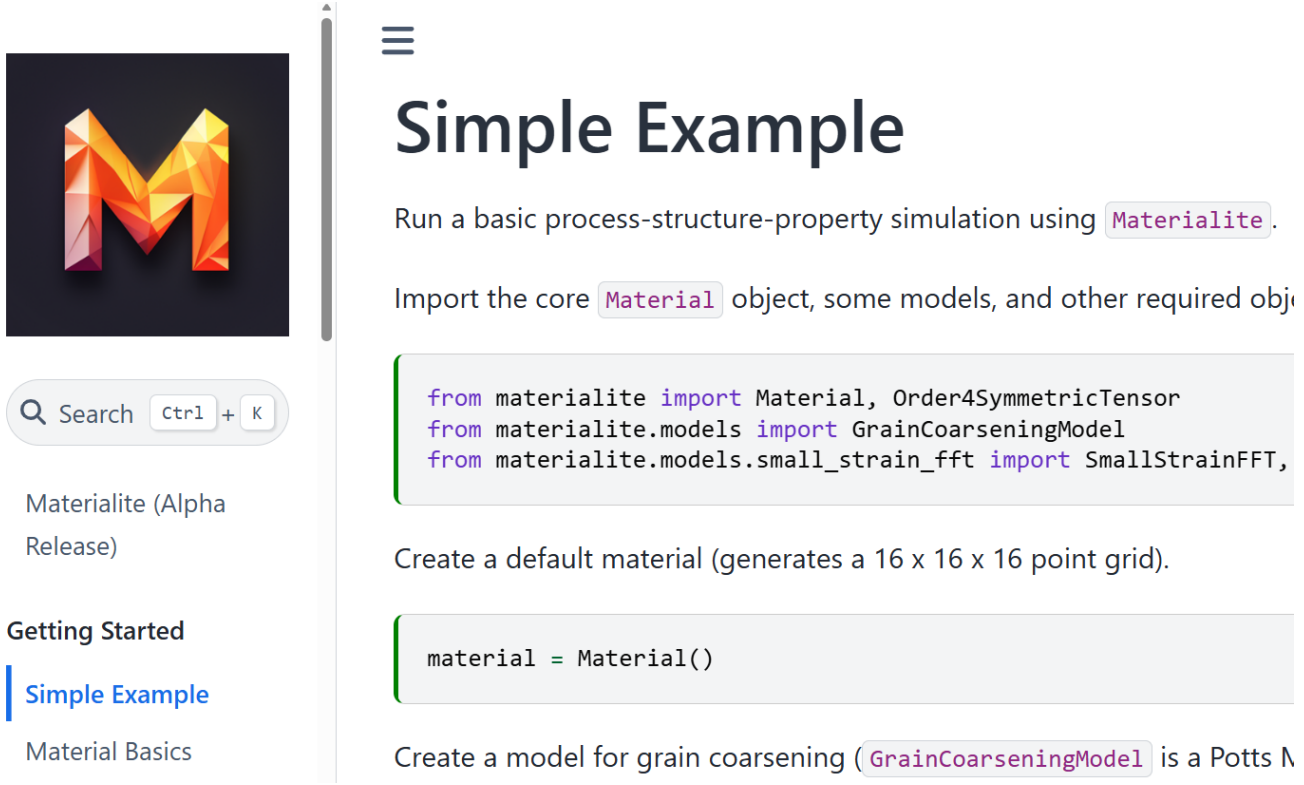
Acknowledgements

- This work was supported by the NASA Aeronautics Research Mission Directorate (ARMD) Transformational Tools and Technologies (TTT) project

- Contact information: joshua.pribe@nasa.gov

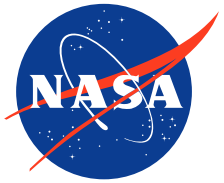
Materialite available at https://github.com/nasa/materialite

Jupyter notebook-based documentation



Backup slides

Tensors



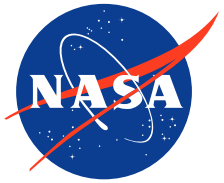
- Tensor math can require messy NumPy operations or for loops
- May need to transform between representations (Cartesian versus Voigt/Mandel)
- May need to change syntax for individual versus pointwise computations

$$\boldsymbol{\sigma} = \mathbb{C}\boldsymbol{\varepsilon}$$

$$\tau = \mathbf{M} : \boldsymbol{\sigma}$$

```
R = np.array([[...], [...], [...]])
cartesian_strain = voigt_to_cartesian(strain)
crystal_strain = cartesian_to_voigt(R @ cartesian_strain @ R.T)
crystal_stress = stiffness @ crystal_strain
resolved_shear_stress = np.tensordot(schmid_tensor,
cartesian_crystal_stress, axes=2)
```

Tensors



- Code operations mirror written/symbolic tensor operations
- Code is the same for tensors representing single values, multiple space/time points, slip systems in a crystal, etc.

```
orientation = Orientation.from_euler_angles([phi1, Phi, phi2])
crystal_strain = strain.to_crystal_frame(orientation)
crystal_stress = stiffness @ crystal_strain
resolved_shear_stress = schmid_tensor * crystal_stress
```

- `Orientation`: Materialite object that manages orientations
- Based on crystallography (i.e., represents a basis transformation from the specimen frame to the crystal frame)
- Utility functions to construct `Orientations` from different input representations (e.g., Euler angles, rotation matrices, quaternions, Miller indices)
- All `Tensor` objects equipped with `to_crystal_frame` and `to_specimen_frame` methods (idea is to explicitly say what the method is doing since these operations often are confusing)

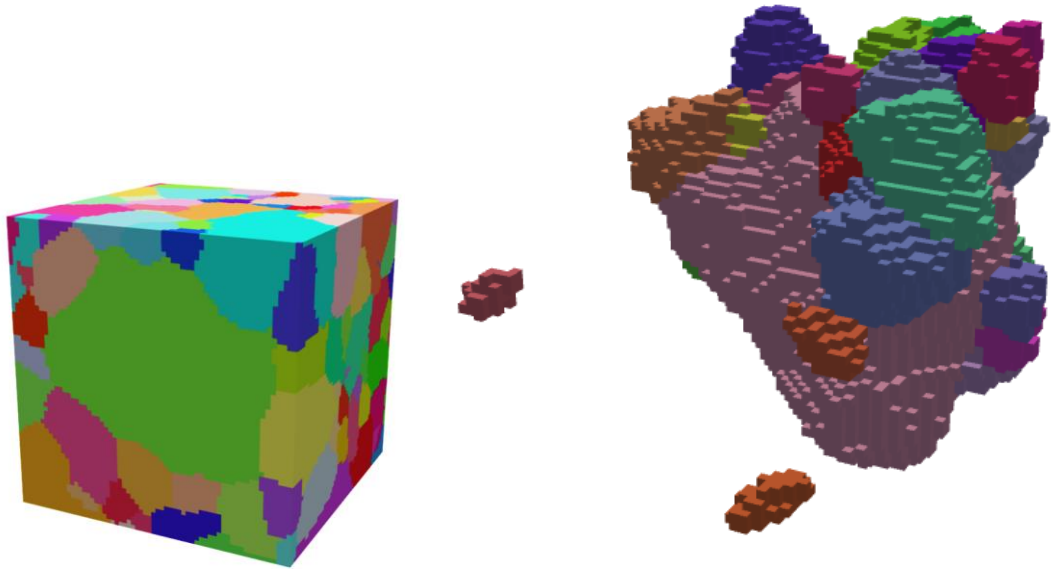
Grain-average quantities

Extract global mean stress and grain-average stresses at each time step in a simulation

```
grain_indices = material.get_region_indices("grain")
for grain in grains_of_interest:
    indices = grain_indices[grain]
    grain_average_stress = stress[indices].mean()
global_mean_stress = stress.mean()
```

Dictionary of field indices corresponding to each grain in the `Material`

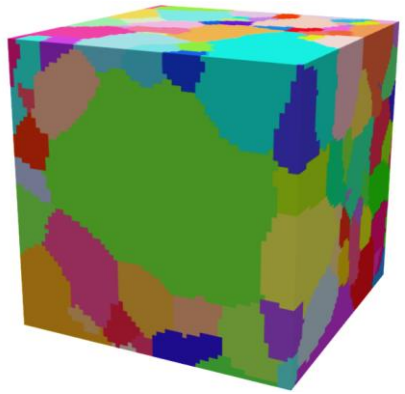
Materialite `Tensor` with stress at the time step (can be indexed and has utility operations like `mean`)



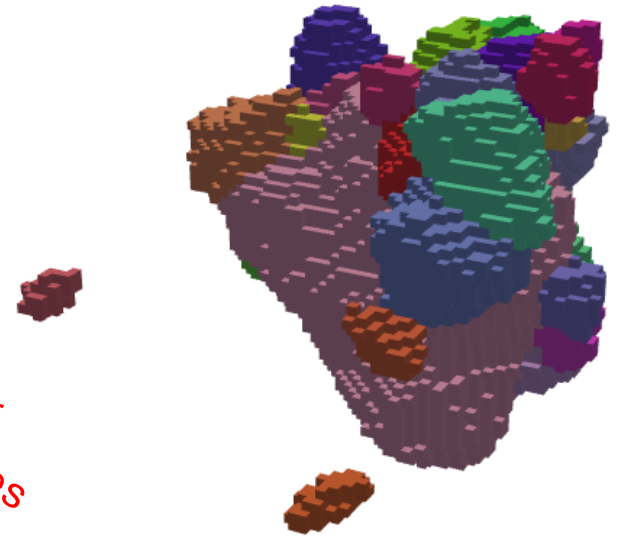
```
model = SmallStrainFFT(
    load_schedule,
    end_time=2.5,
    initial_time_increment=0.25,
    constitutive_model=constitutive_model,
)
material.run(model, output_times=output_times, postprocessor=postprocessor)
```

Grain-average quantities

Extract global mean stress and grain-average stresses at each time step in a simulation



```
grain_indices = material.get_region_indices("grain")
for grain in grains_of_interest:
    indices = grain_indices[grain]
    grain_average_stress = stress[indices].mean()
global_mean_stress = stress.mean()
```



User-prescribed postprocessing function: operate on state variables in each time step to obtain quantities of interest to store

```
model = SmallStrainFFT(
    load_schedule,
    end_time=2.5,
    initial_time_increment=0.25,
    constitutive_model=constitutive_model,
)
material.run(model, output_times=output_times, postprocessor=postprocessor)
```

List of times to store outputs from the simulation