µSpectre Documentation

Release v0.1

Till Junge

Jul 22, 2023

CONTENTS:


µSpectre

CHAPTER

ONE

SUMMARY

Project μSpectre aims at providing an open-source platform for efficient FFT-based continuum mesoscale modelling. It's development is funded by the Swiss National Science Foundation within an Ambizione Project.

Computational continuum mesoscale modelling (or computational homogenisation) involves computing the overall response of a periodic unit cell of material, a so-called representative volume element (RVE), to a given average (i.e., macroscale) strain. Typically, this is done using the finite-element method, even though it is neither able to leverage its main strength, the trivial handling of complex geometries, nor otherwise particularly well suited for periodic problems. An alternative method for modelling periodic RVE, developed by **`Moulinec and Suquet`_**, is based on the fast Fourier transform (FFT). This method has evolved substantially over the last two decades, with particularly important and currently underused improvements in the last two years, see *Zeman et al*, *de Geus et al*.

This new method for the solution of the core problem of computational homogenisation is significantly superior to the FEM in terms of computational cost and memory footprint for most applications, but has not been exploited to its full potential. One major obstacle to the wide adoption of the method is the lack of a robust, validated, open-source code. Hence, researchers choose the well-known and tested FEM that has numerous commercial, open-source or legacy in-house FEM codes.

The goal of this project is to develop μSpectre, an open-source platform for efficient FFT-based continuum mesoscale modelling, which will overcome this obstacle. The project is designed to i) propose a de facto standard implementation for the spectral RVE method that subsequent implementations can be compared to, in order to concentrate the development effort of all interested parties in the field, ii) make μSpectre widely accessible for users by providing language bindings for virtually all relevant popular computing platforms and comprehensive user's manuals in order to help widespread adoption, and, finally iv) make μSpectre eminently modifiable for developers by developing it in the open, with a clean architecture and extensive developer's documentation in order to maximise outside contributions.

Furthermore, this project places great importance on truly reproducible and verifiable science with a credible open data strategy in the firm belief that these qualifiers help to reach and guarantee a high level of scientific quality, difficult to reach otherwise, and to attract outside collaborations and contributions that help boost the scientific output beyond what can be achieved by a single team.

H. Moulinec and P. Suquet. A numerical method for computing the overall response of nonlinear composites with complex microstructure. *Computer Methods in Applied Mechanics and Engineering*, 157(1):69–94, 1998. doi: 10.1016/S0045-7825(97) 00218-1.

J. Zeman, T. W. J. de Geus, J. Vondřejc, R. H. J. Peerlings, and M. G. D. Geers. A finite element perspective on nonlinear FFT-based micromechanical simulations. *International Journal for Numerical Methods in Engineering*, 2017. doi: 10.1002/nme.5481.

T.W.J. de Geus, J. Vondřejc, J. Zeman, R.H.J. Peerlings, M.G.D. Geers. Finite strain FFT-based non-linear solvers made simple, *Computer Methods in Applied Mechanics and Engineering* (318, pp. 412-430), 2017

CHAPTER

TWO

TUTORIALS

2.1 Getting Started

2.1.1 Obtaining µSpectre

µSpectre is hosted on a git repository on gitlab. To clone it, run

```
$ git clone https://gitlab.com/muspectre/muspectre.git
```

or if you prefer identifying yourself using a public ssh-key, run

```
$ git clone git@gitlab.com:muspectre/muspectre.git
```

The latter option requires you to have a user account on gitlab (create).

2.1.2 Building µSpectre

You can compile µSpectre using CMake (3.1.0 or higher). The current (and possibly incomplete list of) dependencies are

- CMake (3.1.0 or higher)
- Boost unit test framework
- FFTW
- git
- Python3 including the header files.
- numpy and scipy.

Recommended:

- Sphinx and Breathe (necessary if you want to build the documentation (turned off by default)
- Eigen (3.3.0 or higher). If you do not install this, it will be downloaded automatically at configuration time, so this is not strictly necessary. The download can be slow, though, so we recommend installing it on your system.
- The CMake curses graphical user interface (`ccmake`).

µSpectre requires a relatively modern compiler as it makes heavy use of C++14
features. It has successfully been compiled and tested using the following compilers under Linux

- gcc-7.2
- gcc-6.4

- gcc-5.4
- clang-6.0
- clang-5.0
- clang-4.0

and using clang-4.0 under MacOS.

It does *not* compile on Intel's most recent compiler, as it is still lacking some C++14 support. Work-arounds are planned, but will have to wait for someone to pick up the task.

To compile, create a build folder and configure the CMake project. If you do this in the folder you cloned in the previous step, it can look for instance like this:

```
$ mkdir build-release
$ cd build-release
$ ccmake ..
```

Then, set the build type to `Release` to produce optimised code. μSpectre makes heavy use of expression templates, so optimisation is paramount. (As an example, the performance difference between code compiled in `Debug` and `Release` is about a factor 40 in simple linear elasticity.)

Finally, compile the library and the tests by running

```
$ make -j <NB-OF-PROCESSES>
```

Warning: When using the `-j` option to compile, be aware that compiling μSpectre uses quite a bit of RAM. If your machine start swapping at compile time, reduce the number of parallel compilations

2.1.3 Running µSpectre

The easiest and intended way of using μSpectre is through its Python bindings. The following simple example computes the response of a two-dimensional stretched periodic RVE cell. The cell consist of a soft matrix with a circular hard inclusion.

More examples both python and c++ executables can be found in the `/bin` folder.

2.1.4 Getting help

μSpectre is in a very early stage of development and the documentation is
currently spotty. Also, there is no FAQ page yet. If you run into trouble, please contact us by opening an issue and someone will answer as soon as possible. You can also check the API *Reference*.

2.1.5 Reporting Bugs

If you think you found a bug, you are probably right. Please report it! The preferred way is for you to create a task on µSpectre's workboard and assign it to user `junge`. Include steps to reproduce the bug if possible. Someone will answer as soon as possible.

2.1.6 Contribute

We welcome contributions both for new features and bug fixes. New features must be documented and have unit tests. Please submit merge requests for review. More detailed guidelines for submissions will follow soon.

2.2 Writing a New Constitutive Law

The abstraction for a constitutive law in μSpectre** is the `Material`, and new such materials must inherit from the class *muSpectre::MaterialBase*. Most often, however, it will be most convenient to inherit from the derived class *muSpectre::MaterialMuSpectre*, as it implements a lot of the machinery that is commonly used by constitutive laws. This section describes how to implement a new constitutive law with internal variables (sometimes also called state variables). The example material implemented here is `MaterialTutorial`, an objective linear elastic law with a distribution of eigenstrains as internal variables. The constitutive law is defined by the relationship between material (or second Piola-Kirchhoff) stress $\mathbf{S}$ and Green-Lagrange strain $\mathbf{E}$

$$\mathbf{S} = \mathbb{C} : (\mathbf{E} - \mathbf{e}) \,, \tag{2.1}$$

$$S_{ij} = C_{ijkl} \left(E_{kl} - e_{kl}\right) , \tag{2.2}$$

$$\tag{2.3}$$

where $\mathbb{C}$ is the elastic stiffness tensor and $\mathbf{e}$ is the local eigenstrain. Note that the implementation sketched out here is the most convenient to quickly get started with using μSpectre**, but not the most efficient one. For a most efficient implementation, refer to the implementation of *muSpectre::MaterialLinearElastic2*.

2.2.1 The `muSpectre::MaterialMuSpectre` class

The class *muSpectre::MaterialMuSpectre* is defined in `material_muSpectre_base.hh` and takes three template parameters;

1. `class Material` is a CRTP and names the material inheriting from it. The reason for this construction is that we want to avoid virtual method calls from *muSpectre::MaterialMuSpectre* to its derived classes. Rather, we want *muSpectre::MaterialMuSpectre* to be able to call methods of the inheriting class directly without runtime overhead.
2. `Dim_t DimS` defines the number of spatial dimensions of the problem, i.e., whether we are dealing with a two- or three-dimensional grid of pixels/voxels.
3. `Dim_t DimM` defines the number of dimensions of our material description. This value will typically be the same as `DimS`, but in cases like generalised plane strain, we can for instance have a three three-dimensional material response in a two-dimensional pixel grid.

The main job of *muSpectre::MaterialMuSpectre* is to

1. loop over all pixels to which this material has been assigned, transform the global gradient $\mathbf{F}$ (or small strain tensor ε) into the new material's required strain measure (e.g., the Green-Lagrange strain tensor $\mathbf{E}$),
2. for each pixel evaluate the constitutive law by calling its `evaluate_stress` (computes the stress response) or `evaluate_stress_tangent` (computes both stress and consistent tangent) method with the local strain and internal variables, and finally

3. transform the stress (and possibly tangent) response from the material's stress measure into first Piola-Kirchhoff stress $\mathbf{P}$ (or Cauchy stress $\boldsymbol{\sigma}$ in small strain).

In order to use these facilities, the new material needs to inherit from *muSpectre::MaterialMuSpectre* (where we calculation of the response) and specialise the type *muSpectre::MaterialMuSpectre_traits* (where we tell *muSpectre::MaterialMuSpectre* how to use the new material). These two steps are described here for our example material.

Specialising the *muSpectre::MaterialMuSpectre_traits* structure ********** This structure is templated by the new material (in this case `MaterialTutorial`) and needs to specify

1. the types used to communicate per-pixel strains, stresses and stiffness tensors to the material (i.e., whether you want to get maps to *Eigen* matrices or raw pointers, or ...). Here we will use the convenient `muSpectre::MatrixFieldMap` for strains and stresses, and `muSpectre::T4MatrixFieldMap` for the stiffness. Look through the classes deriving from `muSpectre::FieldMap` for all available options.
2. the strain measure that is expected (e.g., gradient, Green-Lagrange strain, left Cauchy-Green strain, etc.). Here we will use Green-Lagrange strain. The options are defined by the enum *muSpectre::StrainMeasure*.
3. the stress measure that is computed by the law (e.g., Cauchy, first Piola-Kirchhoff, etc,). Here, it will be first Piola-Kirchhoff stress. The available options are defined by the enum *muSpectre::StressMeasure*.

Our traits look like this (assuming we are in the namespace `muSpectre`:

```
template <Dim_t DimS, Dim_t DimM>
struct MaterialMuSpectre_traits<MaterialTutorial<DimS, DimM>>
{
  //! global field collection
  using GFieldCollection_t = typename
    GlobalFieldCollection<DimS, DimM>;

  //! expected map type for strain fields
  using StrainMap_t = MatrixFieldMap<GFieldCollection_t, Real, DimM, DimM, true>;
  //! expected map type for stress fields
  using StressMap_t = MatrixFieldMap<GFieldCollection_t, Real, DimM, DimM>;
  //! expected map type for tangent stiffness fields
  using TangentMap_t = T4MatrixFieldMap<GFieldCollection_t, Real, DimM>;

  //! declare what type of strain measure your law takes as input
  constexpr static auto strain_measure{StrainMeasure::GreenLagrange};
  //! declare what type of stress measure your law yields as output
  constexpr static auto stress_measure{StressMeasure::PK2};

  //! local field_collections used for internals
  using LFieldColl_t = LocalFieldCollection<DimS, DimM>;
  //! local strain type
  using LStrainMap_t = MatrixFieldMap<LFieldColl_t, Real, DimM, DimM, true>;
  //! elasticity with eigenstrain
  using InternalVariables = std::tuple<LStrainMap_t>;

};
```

2.2.2 Implementing the new material

The new law needs to implement the methods `add_pixel`, `get_internals`, `evaluate_stress`, and `evaluate_stress_tangent`. Below is a commented example header:

```
template <Dim_t DimS, Dim_t DimM>
class MaterialTutorial:
  public MaterialMuSpectre<MaterialTutorial<DimS, DimM>, DimS, DimM>
{
public:
  //! traits of this material
  using traits = MaterialMuSpectre_traits<MaterialTutorial>;

  //! Type of container used for storing eigenstrain
  using InternalVariables = typename traits::InternalVariables;

  //! Construct by name, Young's modulus and Poisson's ratio
  MaterialTutorial(std::string name, Real young, Real poisson);

  /**
   * evaluates second Piola-Kirchhoff stress given the Green-Lagrange
   * strain (or Cauchy stress if called with a small strain tensor)
   */
  template <class s_t, class eigen_s_t>
  inline decltype(auto) evaluate_stress(s_t && E, eigen_s_t && E_eig);

  /**
   * evaluates both second Piola-Kirchhoff stress and stiffness given
   * the Green-Lagrange strain (or Cauchy stress and stiffness if
   * called with a small strain tensor)
   */
  template <class s_t, class eigen_s_t>
  inline decltype(auto)
  evaluate_stress_tangent(s_t &&  E, eigen_s_t && E_eig);

  /**
   * return the internals tuple (needed by `muSpectre::MaterialMuSpectre`)
   */
  InternalVariables & get_internals() {
    return this->internal_variables;};

  /**
   * overload add_pixel to write into eigenstrain
   */
  void add_pixel(const Ccoord_t<DimS> & pixel,
                 const Eigen::Matrix<Real, DimM, DimM> & E_eig);

protected:
  //! stiffness tensor
  T4Mat<Real, DimM> C;
  //! storage for eigenstrain
  using Field_t =
    TensorField<LocalFieldCollection<DimS,DimM>, Real, secondOrder, DimM>;
```

(continues on next page)

(continued from previous page)

```
  Field_t & eigen_field; //!< field of eigenstrains
  //! tuple for iterable eigen_field
  InternalVariables internal_variables;
private:
};
```

A possible implementation for the constructor would be:

```
template <Dim_t DimS, Dim_t DimM>
MaterialTutorial<DimS, DimM>::MaterialTutorial(std::string name,
                                               Real young,
                                               Real poisson)
  :MaterialMuSpectre<MaterialTutorial, DimS, DimM>(name) {

  // Lamé parameters
  Real lambda{young*poisson/((1+poisson)*(1-2*poisson))};
  Real mu{young/(2*(1+poisson))};

  // Kronecker delta
  Eigen::Matrix<Real, DimM, DimM> del{Eigen::Matrix<Real, DimM, DimM>::Identity()};


  // fill the stiffness tensor
  this->C.setZero();
  for (Dim_t i = 0; i < DimM; ++i) {
    for (Dim_t j = 0; j < DimM; ++j) {
      for (Dim_t k = 0; k < DimM; ++k) {
        for (Dim_t l = 0; l < DimM; ++l) {
          get(this->C, i, j, k, l) += (lambda * del(i,j)*del(k,l) +
                                       mu * (del(i,k)*del(j,l) + del(i,l)*del(j,k)));
        }
      }
    }
  }
}
```

as an exercise, you could check how *muSpectre::MaterialLinearElastic1* uses μSpectre**'s materials toolbox (in namespace `MatTB`) to compute $\mathbb{C}$ in a much more convenient fashion. The evaluation of the stress could be (here, we make use of the `Matrices` namespace that defines common tensor algebra operations):

```
template <Dim_t DimS, Dim_t DimM>
template <class s_t, class eigen_s_t>
decltype(auto)
MaterialTutorial<DimS, DimM>::
evaluate_stress(s_t && E, eigen_s_t && E_eig) {
  return Matrices::tens_mult(this->C, E-E_eig);
}
```

The remaining two methods are straight-forward:

```
template <Dim_t DimS, Dim_t DimM>
template <class s_t, class eigen_s_t>
decltype(auto)
```

(continues on next page)

(continued from previous page)

```
MaterialTutorial<DimS, DimM>::
evaluate_stress_tangent(s_t && E, eigen_s_t && E_eig) {
  return return std::make_tuple
        (evaluate_stress(E, E_eig),
         this->C);
}

template <Dim_t DimS, Dim_t DimM>
InternalVariables &
MaterialTutorial<DimS, DimM>::get_internals() {
  return this->internal_variables;
}
```

Note that the methods `evaluate_stress` and `evaluate_stress_tangent` need to be in the header, as both their input parameter types and output type depend on the compile-time context.

CHAPTER

THREE

CODING CONVENTION

3.1 Objectives of the Convention

µSpectre is a collaborative project and these coding conventions aim to make reading and understanding its code as pain-free as possible, while ensuring the four main requirements of the library

1. Versatility
2. Efficiency
3. Reliability
4. Ease-of-use

Versatility requires that the core of the code, i.e., the data structures and fundamental algorithms be written in a generic fashion. The genericity cannot come at the cost of the second requirement – *Efficiency* – which is the reason why the material base classes make extensive use of template metaprogramming and expression templates. *Reliability* can only be enforced through good unit testing with high test coverage, and *ease-of-use* relies on a good documentation for developers and users alike.

Review of submitted code is the main mechanism to enforce the coding conventions.

3.2 Structure

This section contains planned features that are not yet implemented, but that need to be considered during the development effort.

The goal of this section is to define a maintainable and testable architecture for μSpectre. In order to achieve this, the software is segmented in modules that perform one testable task each and are linked through well defined interfaces. This way, when the implementation of a module changes, the other modules do not need to be adapted, as long as the interfaces are respected. In the case of μSpectre, the central task is the evaluation of RVEs, referred to as the core library, and there are the different language bindings, and the FEM plugins. This segmentation to obtain maintainability and testability follows the *Don't repeat yourself!* (DRY) principle stated as *"Every piece of knowledge must have a single, unambiguous, authoritative representation within a system"* by *Hunt (2000)*: Since all algorithms and procedures are implemented only once and only in the core library, there is only one unit test per feature to implement and maintain. Unit tests for the language bindings do not need to retest these features and test merely correct wrapping and proper memory management. An exception to this rule are tests that are more convenient to implement in any of the bound languages rather than in a C++ test (e.g. the FFT module is tested in python against numpy.fft as reference.

Figure *1* shows a schematic of the projected structure and identifies a chain of three modules, where each new module depends on the previous one.

The first module contains only existing external (third-party) FFT implementations and is not part of the development effort for this project. The block is listed for clarity since the choices made here determine the type of machines the final code can run on.

The second module consists of i) the core library, which encapsulates the implementation of the spectral homogenisation method and represents the major projected development and maintenance effort, as well as ii) a set of language bindings. This second module allows single-scale RVE computations directly in most of the popular computing environments. It furthermore allows to rapidly prototype a simulation in a convenient interactive environment such as Jupyter or Matlab, and scale it up to a computing cluster when necessary using the same software.

The third and last module is a collection of plugins for multiple open-source and commercial FEM codes and provides coupled concurrent multiscale computation capabilities in the spirit of FE^2.

3.3 Documentation

There are two types of Documentation for μSpectre: on the one hand, there is this monograph which is supposed to serve as reference manual to understand, and use the library and its extensions, and to look up APIs and data structures. On the other hand, there is in-code documentation helping the developer to understand the role of functions, variables, member (function)s and steps in algorithms.

The in-code documentation uses the syntax of the doxygen documentation generator, as its lightweight markup language is very readable in the code and allows to generate the standalone API documentation in *Reference*.

All lengthier, text-based documentation is written for Sphinx in reStructuredText. This allows to write longer, more expressive texts, such as this convention or the *Tutorials*.

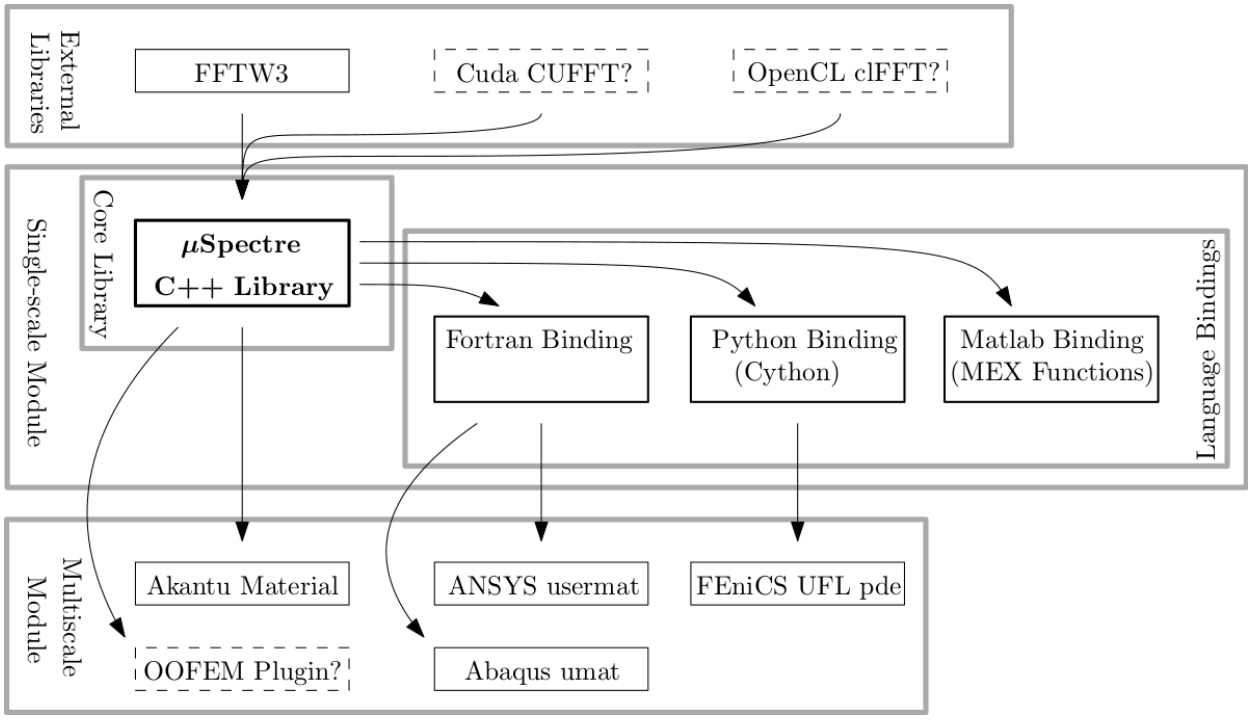

Fig. 1: Figure 1: Principal modules of the platform. Boxes with dashed lines mark optional modules. External libraries refer to established and well-tested existing third-party FFT implementations. The core library μSpectre represents the main development objective of this project and will be written in modern C++14 and wrapped in language bindings for Fortran, Python, and Matlab in order to be usable for single-scale computations by most researchers in their favourite computing environment. Plugins for multiple open-source and commercial FEM codes will use either the core library directly (Akantu, OOFEM), the Fortran language binding (ANSYS, Abaqus), or the Python language binding (FEniCS).

3.4 Testing

Every feature in μSpectre's core library is supposed to be unit tested, and a missing test is considered a bug. Core library features are unit tested in the C++ unit tests (preferred option) or the python unit tests (both within the `tests` folder), because external contributors should not be expected to compile all the language bindings.

The unit tests typically use the Boost unit test framework to define C++ test cases and python's unittest module for python tests. If necessary, standalone tests can be added by contributors, provided that they are added as `ctest` targets to the project's main CMake file. See in the `tests` folder for examples regarding the tests.

3.5 C++ Coding Style and Convention

These are heavily inspired by the Google C++ Style Guide but are *not compatible* with it. These guidelines mostly establish a common vocabulary to write common code and do not give advice for efficient programming practices. For that, follow Scott Meyers book *Effective Modern C++*. As far as possible, the guidelines given in that book are also enforced by the `-Weffc++` compile flag.

The goals of this style guide are:

Style rules should pull their weight
: The benefit of a style rule must be large enough to justify asking all of our engineers to remember it. The benefit is measured relative to the code base we would get without the rule, so a rule against a very harmful practice may still have a small benefit if people are unlikely to do it anyway. This principle mostly explains the rules we don't have, rather than the rules we do: for example, `goto` contravenes many of the following principles, but is already vanishingly rare, so the Style Guide doesn't discuss it.

Optimise for the reader, not the writer
: Our core library (and most individual components submitted to it) is expected to continue for quite some time, and we will hopefully attract more external contributors. As a result, more time will be spent reading most of our code than writing it. We explicitly choose to optimise for the experience of our average contributor reading, maintaining, and debugging code in our code base rather than ease when writing said code. "Leave a trace for the reader" is a particularly common sub-point of this principle: When something surprising or unusual is happening in a snippet of code (for example, use of raw pointers in the `FFTEngine` classes), leaving textual hints for the reader at the point of use is valuable. Use explicit traces of ownership of objects on the heap using smart pointers such as `std::unique_ptr` and `std::shared_ptr`.

Be consistent with existing code
: Using one style consistently through our code base lets us focus on other (more important) issues. Consistency also allows for automation: tools that format your code or adjust your `#includes` only work properly when your code is consistent with the expectations of the tooling. In many cases, rules that are attributed to "Be Consistent" boil down to "Just pick one and stop worrying about it"; the potential value of allowing flexibility on these points is outweighed by the cost of having people argue over them.

Be consistent with the broader C++ community when appropriate
: Consistency with the way other organisations use C++ has value for the same reasons as consistency within our code base. If a feature in the C++ standard solves a problem, or if some idiom is widely known and accepted, that's an argument for using it. However, sometimes standard features and idioms are flawed, or were just designed without our efficiency needs in mind. In those cases (as described below) it's appropriate to constrain or ban standard features.

Avoid surprising or dangerous constructs
: C++ has features that are more surprising or dangerous than one might think at a glance. Some style guide restrictions are in place to prevent falling into these pitfalls. There is a high bar for style guide waivers on such restrictions, because waiving such rules often directly risks compromising program correctness.

Avoid constructs that our average C++ programmer would find tricky or hard to maintain in the constitutive laws and solvers
: C++ has features that may not be generally appropriate because of the complexity they introduce to the code. In the core library, where we make heavy use of template metaprogramming and expression templates for efficiency, it is totally fine to use trickier language constructs, because any benefits of more complex implementation are multiplied widely by usage, and the cost in understanding the complexity does not need to be paid by the average contributor who writes a new material or solver. When in doubt, waivers to rules of this type can be sought by starting an issue.

Concede to optimisation when necessary
: Performance is the overwhelming priority in the **core library** (i.e., data structures and low level algorithms that the typical user relies on often, but rarely uses directly). If performance optimisation is in conflict with other principles in this document, optimise.

3.5.1 Header Files

In general, every `.cc` file should have an associated `.hh` file. There are some common exceptions, such as unit tests and small `.cc` files containing just a `main()` function (e.g., see in the `bin` folder).

Correct use of header files can make a huge difference to the readability, size and performance of your code.

The following rules will guide you through the various pitfalls of using header files.

Self-contained Headers

Header files should be self-contained (compile on their own) and end in `.hh`. There should not be any non-header files that are meant for inclusion.

All header files should be self-contained. Users and refactoring tools should not have to adhere to special conditions to include the header. Specifically, a header should have header guards and include all other headers it needs.

Prefer placing the definitions for inline functions in the same file as their declarations. The definitions of these constructs must be included into every `.cc` file that uses them, or the program may fail to link in some build configurations. If declarations and definitions are in different files, including the former should transitively include the latter. Do not move these definitions to separately included header files (`-inl.hh`); this practice was common in the past, but is no longer allowed.

As an exception, a template that is explicitly instantiated for all relevant sets of template arguments, or that is a private implementation detail of a class, is allowed to be defined in the one and only `.cc` file that instantiates the template, see `material_linear_elastic1.cc` for an example.

The `#define` Guard

All header files should have `#define` guards to prevent multiple inclusion. The format of the symbol name should be `CLASS_NAME_H` (all caps with underscores), where `ClassName` (CamelCase) is the main class declared it the header file.

Make sure to use unique file names to avoid triggering the wrong `#define` guard.

Forward Declarations

Use forward declarations of µSpectre entities where it avoids `includes` and saves compile time.

A "forward declaration" is a declaration of a class, function, or template without an associated definition.

Pros:

- Forward declarations can save compile time, as `#includes` force the compiler to open more files and process more input.
- Forward declarations can save on unnecessary recompilation. `#includes` can force your code to be recompiled more often, due to unrelated changes in the header.

Cons:

- Forward declarations can hide a dependency, allowing user code to skip necessary recompilation when headers change.
- A forward declaration may be broken by subsequent changes to the library. Forward declarations of functions and templates can prevent the header owners from making otherwise-compatible changes to their APIs, such as widening a parameter type, adding a template parameter with a default value, or migrating to a new namespace.
- Forward declaring symbols from namespace std:: yields undefined behaviour.
- It can be difficult to determine whether a forward declaration or a full `#include` is needed. Replacing an `#include` with a forward declaration can silently change the meaning of code:

```
// b.hh:
struct B {};
struct D : B {};

// good_user.cc:
#include "b.hh"
void f(B*);
void f(void*);
void test(D* x) { f(x); }  // calls f(B*)
```

  If the #include was replaced with forward declarations for `B` and `D`, `test()` would call `f(void*)`.

- Forward declaring multiple symbols from a header can be more verbose than simply `#includeing` the header.

Try to avoid forward declarations of entities defined in another project.

Inline Functions

Use inline functions for performance-critical code. Also, templated member functions that that cannot be explicitly instantiated need to be declared inline.

Names and Order of Includes

All of a project's header files should be listed as descendants of the project's source directory without use of UNIX directory shortcuts `.` (the current directory) or `..` (the parent directory). For example, `muSpectre/src/common/ccoord_operations.hh` should be included as:

```
#include <libmugrid/ccoord_operations.hh>
```

Use the following order for includes to avoid hidden dependencies:

1. μSpectre headers
2. A blank line
3. Other libraries' headers
4. A blank line
5. C++ system headers

With this ordering, if a μSpectre header omits any necessary includes, the build will break. Thus, this rule ensures that build breaks show up first for the people working on these files, not for innocent people in different places.

You should include all the headers that define the symbols you rely upon, except in the case of forward declaration. If you rely on symbols from `bar.hh`, don't count on the fact that you included `foo.hh` which (currently) includes `bar.hh`: include `bar.hh` yourself, unless `foo.hh` explicitly demonstrates its intent to provide you the symbols of `bar.hh`. However, any includes present in the related header do not need to be included again in the related `.cc` (i.e., `foo.cc` can rely on `foo.hh`'s includes).

3.5.2 Scoping

Namespaces

With few exceptions, place code in the namespace `muSpectre`. All other (subordinate) namespaces should have unique, expressive names based on their purpose. Do not use using-directives (e.g. `using namespace foo`) within the core library (but feel free to do so in the executables in the `bin` folder). Do not use inline namespaces. For unnamed namespaces, see *Unnamed Namespaces and Static Variables*.

Definition:
Namespaces subdivide the global scope into distinct, named scopes, and so are useful for preventing name collisions in the global scope.

Pros:

- Namespaces provide a method for preventing name conflicts in large programs while allowing most code to use reasonably short names.

  For example, if two different projects have a class `Foo` in the global scope, these symbols may collide at compile time or at runtime. If each project places their code in a namespace, `project1::Foo` and `project2::Foo` are now distinct symbols that do not collide, and code within each project's namespace can continue to refer to Foo without the prefix.

- Inline namespaces automatically place their names in the enclosing scope. Consider the following snippet, for example:

```
namespace outer {
  inline namespace inner {
    void foo();
```

(continues on next page)

(continued from previous page)

```
  }  // namespace inner
}  // namespace outer
```

The expressions `outer::inner::foo()` and `outer::foo()` are interchangeable. Inline namespaces are primarily intended for ABI compatibility across versions.

Cons:

- Inline namespaces, in particular, can be confusing because names aren't actually restricted to the namespace where they are declared. They are only useful as part of some larger versioning policy.
- In some contexts, it's necessary to repeatedly refer to symbols by their fully-qualified names. For deeply-nested namespaces, this can add a lot of clutter.

Decision:

Namespaces should be used as follows:

- Follow the rules on *Namespace Names*.
- Terminate namespaces with comments as shown in the given examples.
- Namespaces wrap the entire source file after includes and forward declarations of classes from other namespaces.

```
// In the .hh file
namespace mynamespace {

  // All declarations are within the namespace scope.
  // Notice the lack of indentation.
  class MyClass {
    public:
     ...
     void Foo();
  };

}  // namespace mynamespace

// In the .cc file
namespace mynamespace {

  // Definition of functions is within scope of the namespace.
  void MyClass::Foo() {
    ...
  }

}  // namespace mynamespace
```

More complex `.cc` files might have additional details, using-declarations.

```
#include "a.h"

    namespace mynamespace {

      using ::foo::bar;

      ...code for mynamespace...    // Code goes against the left margin.
```

(continues on next page)

(continued from previous page)

```
  }  // namespace mynamespace
```

- Do not declare anything in namespace `std`, including forward declarations of standard library classes. Declaring entities in namespace `std` is undefined behaviour, i.e., not portable. To declare entities from the standard library, include the appropriate header file.
- You may not use a *using-directive* to make all names from a namespace available (namespace clobbering).

```
// Forbidden -- This pollutes the namespace.
using namespace foo;
```

- Do not use *namespace aliases* at namespace scope in header files except in explicitly marked internal-only namespaces, because anything imported into a namespace in a header file becomes part of the public API exported by that file.

```
// Shorten access to some commonly used names in .cc files.
namespace baz = ::foo::bar::baz;

// Shorten access to some commonly used names (in a .h file).
namespace librarian {
  namespace impl {  // Internal, not part of the API.
    namespace sidetable = ::pipeline_diagnostics::sidetable;
  }  // namespace impl

  inline void my_inline_function() {
    // namespace alias local to a function (or method).
    namespace baz = ::foo::bar::baz;
    ...
  }
}  // namespace librarian
```

- Do not use inline namespaces.

Unnamed Namespaces and Static Variables

When definitions in a `.cc` file do not need to be referenced outside that file, place them in an unnamed namespace or declare them static. Do not use either of these constructs in `.hh` files.

All declarations can be given internal linkage by placing them in unnamed namespaces. Functions and variables can also be given internal linkage by declaring them static. This means that anything you're declaring can't be accessed from another file. If a different file declares something with the same name, then the two entities are completely independent.

Use of internal linkage in `.cc` files is encouraged for all code that does not need to be referenced elsewhere. Do not use internal linkage in `.hh` files.

Format unnamed namespaces like named namespaces. In the terminating comment, leave the namespace name empty:

```
namespace {
  ...
}  // namespace
```

Nonmember, Static Member, and Global Functions

Prefer placing nonmember functions in a namespace; use completely global functions rarely. Note: placing functions in a namespace keeps them globally accessible, the goal of this is not to suppress the use of non-member functions but rather to avoid polluting the global and `muSpectre` namespace by grouping them together in thematic namespaces, see for instance the namespace `MatTB` in `materials/materials_toolbox.cc`. Do not use a class simply to group static functions, unless they are function templates which need to be partially specialised. Otherwise, static methods of a class should generally be closely related to instances of the class or the class's static data.

Pros:
Nonmember and static member functions can be useful in some situations. Putting nonmember functions in a namespace avoids polluting the global namespace.

Cons:
Nonmember and static member functions may make more sense as members of a new class, especially if they access external resources or have significant dependencies.

Decision:
Sometimes it is useful to define a function not bound to a class instance. Such a function can be either a static member or a nonmember function. Nonmember functions should not depend on external variables, and should nearly always exist in a namespace. Do not create classes only to group static member functions, unless they are function templates which need to be partially specialised; otherwise, this is no different than just giving the function names a common prefix, and such grouping is usually unnecessary anyway.

If you define a nonmember function and it is only needed in its `.cc` file, use *internal linkage* to limit its scope.

Local Variables

Place a function's variables in the narrowest scope possible, and initialise variables in the declaration.

C++ allows you to declare variables anywhere in a function. We encourage you to declare them in as local a scope as possible, and as close to the first use as possible. This makes it easier for the reader to find the declaration and see what type the variable is and what it was initialised to. In particular, initialisation should be used instead of declaration and assignment, e.g.:

```
int i;
i = f();     // Bad -- initialisation separate from declaration.

int j{g()};  // Good -- declaration has initialisation.

std::vector<int> v;
v.push_back(1);  // Prefer initialising using brace initialisation.
v.push_back(2);

std::vector<int> v = {1, 2};  // Good -- v starts initialised.
```

Prefer C++11-style universal initialisation (`int i{0}`) over legacy initialisation (`int i = 0`).

Variables needed for `if`, `while` and `for` statements should normally be declared within those statements, so that such variables are confined to those scopes. E.g.:

```
for (size_t i{0}; i < DimS; ++i) {
  ...
}
```

There is one caveat: if the variable is an object, its constructor is invoked every time it enters scope and is created, and its destructor is invoked every time it goes out of scope.

```
// Inefficient implementation:
for (int i = 0; i < 1000000; ++i) {
  Foo f;  // My ctor and dtor get called 1000000 times each.
  f.do_something(i);
}
```

It may be more efficient to declare such a variable used in a loop outside that loop:

```
Foo f;  // My ctor and dtor get called once each.
for (int i = 0; i < 1000000; ++i) {
  f.do_something(i);
}
```

Static and Global Variables

Objects with static storage duration are forbidden unless they are trivially destructible. Informally this means that the destructor does not do anything, even taking member and base destructors into account. More formally it means that the type has no user-defined or virtual destructor and that all bases and non-static members are trivially destructible. Static function-local variables may use dynamic initialisation. Use of dynamic initialisation for static class member variables or variables at namespace scope is discouraged, but allowed in limited circumstances; see below for details.

As a rule of thumb: a global variable satisfies these requirements if its declaration, considered in isolation, could be `constexpr`.

Definition:
Every object has a *storage duration*, which correlates with its lifetime. Objects with static storage duration live from the point of their initialisation until the end of the program. Such objects appear as variables at namespace scope ("global variables"), as static data members of classes, or as function-local variables that are declared with the `static` specifier. Function-local static variables are initialised when control first passes through their declaration; all other objects with static storage duration are initialised as part of program start-up. All objects with static storage duration are destroyed at program exit (which happens before unjoined threads are terminated).

Initialisation may be *dynamic*, which means that something non-trivial happens during initialisation. (For example, consider a constructor that allocates memory, or a variable that is initialised with the current process ID.) The other kind of initialisation is *static* initialisation. The two aren't quite opposites, though: static initialisation *always* happens to objects with static storage duration (initialising the object either to a given constant or to a representation consisting of all bytes set to zero), whereas dynamic initialisation happens after that, if required.

Pros:
Global and static variables are very useful for a large number of applications: named constants, auxiliary data structures internal to some translation unit, command-line flags, logging, registration mechanisms, background infrastructure, etc.

Cons:
Global and static variables that use dynamic initialisation or have non-trivial destructors create complexity that can easily lead to hard-to-find bugs. Dynamic initialisation is not ordered across translation units, and neither is destruction (except that destruction happens in reverse order of initialisation). When one initialisation refers to another variable with static storage duration, it is possible that this causes an object to be accessed before its lifetime has begun (or after its lifetime has ended). Moreover, when a program starts threads that are not joined at exit, those threads may attempt to access objects after their lifetime has ended if their destructor has already run.

Decision:
Decision on destruction

When destructors are trivial, their execution is not subject to ordering at all (they are effectively not "run"); otherwise we are exposed to the risk of accessing objects after the end of their lifetime. Therefore, we only allow objects with static storage duration if they are trivially destructible. Fundamental types (like pointers and int) are trivially destructible, as are arrays of trivially destructible types. Note that variables marked with `constexpr` are trivially destructible.

```
const int kNum{10};  // allowed

struct X { int n; };
const X kX[]{{1}, {2}, {3}};  // allowed

void foo() {
  static const char* const kMessages[]{"hello", "world"};  // allowed
}

// allowed: constexpr guarantees trivial destructor
constexpr std::array<int, 3> kArray {{1, 2, 3}};
```

```
// bad: non-trivial destructor
const string kFoo("foo");

// bad for the same reason, even though kBar is a reference (the
// rule also applies to lifetime-extended temporary objects)
const string& kBar(StrCat("a", "b", "c"));

void bar() {
  // bad: non-trivial destructor
  static std::map<int, int> kData{{1, 0}, {2, 0}, {3, 0}};
}
```

Note that references are not objects, and thus they are not subject to the constraints on destructibility. The constraint on dynamic initialisation still applies, though. In particular, a function-local static reference of the form `static T& t = *new T;` is allowed.

Decision on initialisation

Initialisation is a more complex topic. This is because we must not only consider whether class constructors execute, but we must also consider the evaluation of the initialiser:

```
int n{5};     // fine
int m{f()};   // ? (depends on f)
Foo x;        // ? (depends on Foo::Foo)
Bar y{g()};   // ? (depends on g and on Bar::Bar)
```

All but the first statement expose us to indeterminate initialisation ordering.

The concept we are looking for is called *constant initialisation* in the formal language of the C++ standard. It means that the initialising expression is a constant expression, and if the object is initialised by a constructor call, then the constructor must be specified as `constexpr`, too:

```
struct Foo { constexpr Foo(int) {} };

int n{5};   // fine, 5 is a constant expression
Foo x(2);   // fine, 2 is a constant expression and the chosen constructor is␣
```

(continues on next page)

(continued from previous page)

```
↪constexpr
Foo a[] { Foo(1), Foo(2), Foo(3) };  // fine
```

Constant initialisation is always allowed. Constant initialisation of static storage duration variables should be marked with `constexpr`. Any non-local static storage duration variable that is not so marked should be presumed to have dynamic initialisation, and reviewed very carefully.

By contrast, the following initialisations are problematic:

```
time_t time(time_t*);      // not ``constexpr``!
int f();                   // not ``constexpr``!
struct Bar { Bar() {} };

time_t m{time(nullptr)};   // initialising expression not a constant expression
Foo y(f());                // ditto
Bar b;                     // chosen constructor Bar::Bar() not ``constexpr``
```

Dynamic initialisation of nonlocal variables is discouraged, and in general it is forbidden. However, we do permit it if no aspect of the program depends on the sequencing of this initialisation with respect to all other initialisations. Under those restrictions, the ordering of the initialisation does not make an observable difference. For example:

```
int p{getpid()};  // allowed, as long as no other static variable
                  // uses p in its own initialisation
```

Dynamic initialisation of static local variables is allowed (and common).

Common patterns

- Global strings: if you require a global or static string constant, consider using a simple character array, or a char pointer to the first element of a string literal. String literals have static storage duration already and are usually sufficient.
- Maps, sets, and other dynamic containers: if you require a static, fixed collection, such as a set to search against or a lookup table, you cannot use the dynamic containers from the standard library as a static variable, since they have non-trivial destructors. Instead, consider a simple array of trivial types, e.g. an array of arrays of `int` (for a "map from `int` to `int`"), or an array of pairs (e.g. pairs of `int` and `const char*`). For small collections, linear search is entirely sufficient (and efficient, due to memory locality). If necessary, keep the collection in sorted order and use a binary search algorithm. If you do really prefer a dynamic container from the standard library, consider using a function-local static pointer, as described below.
- Smart pointers (`std::unique_ptr`, `std::shared_ptr`): smart pointers execute cleanup during destruction and are therefore forbidden. Consider whether your use case fits into one of the other patterns described in this section. One simple solution is to use a plain pointer to a dynamically allocated object and never delete it (see last item).
- Static variables of custom types: if you require `static`, constant data of a type that you need to define yourself, give the type a trivial destructor and a `constexpr` constructor.
- If all else fails, you can create an object dynamically and never delete it by binding the pointer to a function-local static pointer variable: `static const auto* const impl = new T(args...);` (If the initialisation is more complex, it can be moved into a function or lambda expression.)

`thread_local` Variables

`thread_local` variables that aren't declared inside a function must be initialised with a true compile-time constant. Prefer `thread_local` over other ways of defining thread-local data.

Definition:

Starting with C++11, variables can be declared with the `thread_local` specifier:

```
thread_local Foo foo{...};
```

Such a variable is actually a collection of objects, so that when different threads access it, they are actually accessing different objects. `thread_local` variables are much like static storage duration variables in many respects. For instance, they can be declared at namespace scope, inside functions, or as static class members, but not as ordinary class members.

`thread_local` variable instances are initialised much like static variables, except that they must be initialised separately for each thread, rather than once at program startup. This means that `thread_local` variables declared within a function are safe, but other `thread_local` variables are subject to the same initialisation-order issues as static variables (and more besides).

`thread_local` variable instances are destroyed when their thread terminates, so they do not have the destruction-order issues of static variables.

Pros:

- Thread-local data is inherently safe from races (because only one thread can ordinarily access it), which makes `thread_local` useful for concurrent programming.
- `thread_local` is the only standard-supported way of creating thread-local data.

Cons:

- Accessing a `thread_local` variable may trigger execution of an unpredictable and uncontrollable amount of other code.
- `thread_local` variables are effectively global variables, and have all the drawbacks of global variables other than lack of thread-safety.
- The memory consumed by a `thread_local` variable scales with the number of running threads (in the worst case), which can be quite large in a program.
- An ordinary class member cannot be `thread_local`.
- `thread_local` may not be as efficient as certain compiler intrinsics.

Decision:

`thread_local` variables inside a function have no safety concerns, so they can be used without restriction. Note that you can use a function-scope `thread_local` to simulate a class- or namespace-scope `thread_local` by defining a function or static method that exposes it:

```
Foo& MyThreadLocalFoo() {
  thread_local Foo result{ComplicatedInitialisation()};
  return result;
}
```

`thread_local` variables at class or namespace scope must be initialised with a true compile-time constant (i.e. they must have no dynamic initialisation). To enforce this, `thread_local` variables at class or namespace scope must be annotated with `constexpr`:

```
constexpr thread_local Foo foo = ...;
```

`thread_local` should be preferred over other mechanisms for defining thread-local data.

3.5.3 Classes

Classes are the fundamental unit of code in C++. Naturally, we use them extensively. This section lists the main dos and don'ts you should follow when writing a class.

Doing Work in Constructors

Avoid virtual method calls in constructors, and avoid initialisation that can fail if you can't signal an error.

Definition:
It is possible to perform arbitrary initialisation in the body of the constructor.

Pros:

- No need to worry about whether the class has been initialised or not.
- Objects that are fully initialised by constructor call can be const and may also be easier to use with standard containers or algorithms.

Cons:

- If the work calls virtual functions, these calls will not get dispatched to the subclass implementations. Future modification to your class can quietly introduce this problem even if your class is not currently subclassed, causing much confusion.
- There is no easy way for constructors to signal errors, short of crashing the program (not always appropriate) or using exceptions.
- If the work fails, we now have an object whose initialisation code failed, so it may be an unusual state requiring a `bool is_valid()` state checking mechanism (or similar) which is easy to forget to call.
- You cannot take the address of a constructor, so whatever work is done in the constructor cannot easily be handed off to, for example, another thread.

Decision:
Constructors should never call virtual functions. If appropriate for your code , terminating the program may be an appropriate error handling response. Otherwise, consider a factory function or `initialise()` method as described in TotW #42 . Avoid `initialise()` methods on objects with no other states that affect which public methods may be called (semi-constructed objects of this form are particularly hard to work with correctly).

Implicit Conversions

Do not define implicit conversions. Use the `explicit` keyword for conversion operators and single-argument constructors.

Definition:
Implicit conversions allow an object of one type (called the *source type*) to be used where a different type (called the *destination type*) is expected, such as when passing an `int` argument to a function that takes a `double` parameter.

In addition to the implicit conversions defined by the language, users can define their own, by adding appropriate members to the class definition of the source or destination type. An implicit conversion in the source type is defined by a type conversion operator named after the destination type (e.g. `operator bool()`). An implicit conversion in the destination type is defined by a constructor that can take the source type as its only argument (or only argument with no default value).

The `explicit` keyword can be applied to a constructor or (since C++11) a conversion operator, to ensure that it can only be used when the destination type is explicit at the point of use, e.g. with a cast. This applies not only to implicit conversions, but to C++11's list initialisation syntax:

```
class Foo {
  explicit Foo(int x, double y);
  ...
};

void Func(Foo f);

Func({42, 3.14});  // Error
```

This kind of code isn't technically an implicit conversion, but the language treats it as one as far as `explicit` is concerned.

Pros:

- Implicit conversions can make a type more usable and expressive by eliminating the need to explicitly name a type when it's obvious.
- Implicit conversions can be a simpler alternative to overloading, such as when a single function with a `string_view` parameter takes the place of separate overloads for `string` and `const char*`.
- List initialisation syntax is a concise and expressive way of initialising objects.

Cons:

- Implicit conversions can hide type-mismatch bugs, where the destination type does not match the user's expectation, or the user is unaware that any conversion will take place.
- Implicit conversions can make code harder to read, particularly in the presence of overloading, by making it less obvious what code is actually getting called.
- Constructors that take a single argument may accidentally be usable as implicit type conversions, even if they are not intended to do so.
- When a single-argument constructor is not marked `explicit`, there's no reliable way to tell whether it's intended to define an implicit conversion, or the author simply forgot to mark it.
- It's not always clear which type should provide the conversion, and if they both do, the code becomes ambiguous.
- List initialisation can suffer from the same problems if the destination type is implicit, particularly if the list has only a single element.

Decision:

Type conversion operators, and constructors that are callable with a single argument, must be marked `explicit` in the class definition. As an exception, copy and move constructors should not be `explicit`, since they do not perform type conversion. Implicit conversions can sometimes be necessary and appropriate for types that are designed to transparently wrap other types. In that case, raise an issue.

Constructors that cannot be called with a single argument may omit `explicit`. Constructors that take a single `std::initialiser_list` parameter should also omit `explicit`, in order to support copy-initialisation (e.g. `MyType m{1, 2};`).

Copyable and Movable Types

A class's public API should make explicit whether the class is copyable, move-only, or neither copyable nor movable. Support copying and/or moving if these operations are clear and meaningful for your type.

Definition:

A movable type is one that can be initialised and assigned from temporaries.

A copyable type is one that can be initialised or assigned from any other object of the same type (so is also movable by definition), with the stipulation that the value of the source does not change. `std::unique_ptr<int>` is an example of a movable but not copyable type (since the value of the source `std::unique_ptr<int>` must be modified during assignment to the destination). `int` and `string` are examples of movable types that are also copyable. (For `int`, the move and copy operations are the same; for `string`, there exists a move operation that is less expensive than a copy.)

For user-defined types, the copy behaviour is defined by the copy constructor and the copy-assignment operator. Move behaviour is defined by the move constructor and the move-assignment operator, if they exist, or by the copy constructor and the copy-assignment operator otherwise.

The copy/move constructors can be implicitly invoked by the compiler in some situations, e.g. when passing objects by value.

Pros:

Objects of copyable and movable types can be passed and returned by value, which makes APIs simpler, safer, and more general. Unlike when passing objects by pointer or reference, there's no risk of confusion over ownership, lifetime, mutability, and similar issues, and no need to specify them in the contract. It also prevents non-local interactions between the client and the implementation, which makes them easier to understand, maintain, and optimise by the compiler. Further, such objects can be used with generic APIs that require pass-by-value, such as most containers, and they allow for additional flexibility in e.g., type composition.

Copy/move constructors and assignment operators are usually easier to define correctly than alternatives like `clone()`, `copy_from()` or `swap()`, because they can be generated by the compiler, either implicitly or with `= default`. They are concise, and ensure that all data members are copied. Copy and move constructors are also generally more efficient, because they don't require heap allocation or separate initialisation and assignment steps, and they're eligible for optimisations such as copy elision.

Move operations allow the implicit and efficient transfer of resources out of rvalue objects. This allows a plainer coding style in some cases.

Cons:

Some types do not need to be copyable, and providing copy operations for such types can be confusing, nonsensical, or outright incorrect. Types representing singleton objects (Registerer), objects tied to a specific scope (Cleanup), or closely coupled to object identity (Mutex) cannot be copied meaningfully. Copy operations for base class types that are to be used polymorphically are hazardous, because use of them can lead to object slicing. Defaulted or carelessly-implemented copy operations can be incorrect, and the resulting bugs can be confusing and difficult to diagnose.

Copy constructors are invoked implicitly, which makes the invocation easy to miss. This may cause confusion for programmers used to languages where pass-by-reference is conventional or mandatory. It may also encourage excessive copying, which can cause performance problems.

Decision:

Every class's public interface should make explicit which copy and move operations the class supports. This should usually take the form of explicitly declaring and/or deleting the appropriate operations in the public section of the declaration.

Specifically, a copyable class should explicitly declare the copy operations, a move-only class should explicitly declare the move operations, and a non-copyable/movable class should explicitly delete the copy operations. Explicitly declaring or deleting all four copy/move operations is required. If you provide a copy or move assignment

operator, you must also provide the corresponding constructor.

```
class Copyable {
  public:
    //! Default constructor
    Copyable() = delete;

    //! Copy constructor
    Copyable(const Copyable &other);

    //! Move constructor
    Copyable(Copyable &&other) = delete;

    //! Destructor
    virtual ~Copyable() noexcept;

    //! Copy assignment operator
    Copyable& operator=(const Copyable &other);

    //! Move assignment operator
    Copyable& operator=(Copyable &&other) = delete;

  protected:
    ...
  private:
    ...

  };

  class MoveOnly {
  public:
    //! Default constructor
    MoveOnly() = delete;

    //! Copy constructor
    MoveOnly(const MoveOnly &other) = delete;

    //! Move constructor
    MoveOnly(MoveOnly &&other);

    //! Destructor
    virtual ~MoveOnly() noexcept;

    //! Copy assignment operator
    MoveOnly& operator=(const MoveOnly &other) = delete;

    //! Move assignment operator
    MoveOnly& operator=(MoveOnly &&other);

  protected:
    ...
  private:
    ...
```

(continues on next page)

(continued from previous page)

```
};

class NotCopyableNorMovable {
public:
  //! Default constructor
  NotCopyableNorMovable() = delete;

  //! Copy constructor
  NotCopyableNorMovable(const NotCopyableNorMovable &other) = delete;

  //! Move constructor
  NotCopyableNorMovable(NotCopyableNorMovable &&other);

  //! Destructor
  virtual ~NotCopyableNorMovable() noexcept;

  //! Copy assignment operator
  NotCopyableNorMovable& operator=(const NotCopyableNorMovable &other) = delete;

  //! Move assignment operator
  NotCopyableNorMovable& operator=(NotCopyableNorMovable &&other) = delete;

protected:
  ...
private:
  ...
};
```

These declarations/deletions can be omitted only if they are obvious: for example, if a base class isn't copyable or movable, derived classes naturally won't be either. Similarly, a `struct`'s copyability/movability is normally determined by the copyability/movability of its data members. Note that if you explicitly declare or delete any of the copy/move operations, the others are not obvious, and so this paragraph does not apply (in particular, the rules in this section that apply to `class`es also apply to `struct`s that declare or delete any copy/move operations).

A type should not be copyable/movable if it incurs unexpected costs. Move operations for copyable types are strictly a performance optimisation and are a potential source of bugs and complexity, so define them if they have a chance of being more efficient than the corresponding copy operations. If your type provides copy operations, it is recommended that you design your class so that the default implementation of those operations is correct. Remember to review the correctness of any defaulted operations as you would any other code.

Structs vs. Classes

Use a `struct` only for passive objects that carry data or collections of templated static member functions that need to be partially specialised; everything else is a `class`.

The `struct` and `class` keywords behave almost identically in C++. We add our own semantic meanings to each keyword, so you should use the appropriate keyword for the data-type you're defining.

`struct`s should be used for passive objects that carry data, and may have associated constants, but lack any functionality other than access/setting the data members. The accessing/setting of fields is done by directly accessing the fields rather than through method invocations.

Methods should only be used in templated static method-only `struct`s. See, e.g.:

```
//! static inline implementation of Hooke's law
template <Dim_t Dim, class Strain_t, class Tangent_t>
struct Hooke {
  /**
   * compute Lamé's first constant
   * @param young: Young's modulus
   * @param poisson: Poisson's ratio
   */
  inline static constexpr Real
  compute_lambda(const Real & young, const Real & poisson) {
    return convert_elastic_modulus<ElasticModulus::lambda,
                                   ElasticModulus::Young,
                                   ElasticModulus::Poisson>(young, poisson);
  }

  /**
   * compute Lamé's second constant (i.e., shear modulus)
   * @param young: Young's modulus
   * @param poisson: Poisson's ratio
   */
  inline static constexpr Real
  compute_mu(const Real & young, const Real & poisson) {
    return convert_elastic_modulus<ElasticModulus::Shear,
                                   ElasticModulus::Young,
                                   ElasticModulus::Poisson>(young, poisson);
  }

  /**
   * compute the bulk modulus
   * @param young: Young's modulus
   * @param poisson: Poisson's ratio
   */
  inline static constexpr Real
  compute_K(const Real & young, const Real & poisson) {
    return convert_elastic_modulus<ElasticModulus::Bulk,
                                   ElasticModulus::Young,
                                   ElasticModulus::Poisson>(young, poisson);
  }

  /**
   * compute the stiffness tensor
   * @param lambda: Lamé's first constant
   * @param mu: Lamé's second constant (i.e., shear modulus)
   */
  inline static Eigen::TensorFixedSize<Real, Eigen::Sizes<Dim, Dim, Dim, Dim>>
  compute_C(const Real & lambda, const Real & mu) {
    return lambda*Tensors::outer<Dim>(Tensors::I2<Dim>(),Tensors::I2<Dim>()) +
      2*mu*Tensors::I4S<Dim>();
  }

  /**
   * compute the stiffness tensor
   * @param lambda: Lamé's first constant
```

(continues on next page)

(continued from previous page)

```
   * @param mu: Lamé's second constant (i.e., shear modulus)
   */
  inline static T4Mat<Real, Dim>
  compute_C_T4(const Real & lambda, const Real & mu) {
    return lambda*Matrices::Itrac<Dim>() + 2*mu*Matrices::Isymm<Dim>();
  }

  /**
   * return stress
   * @param lambda: First Lamé's constant
   * @param mu: Second Lamé's constant (i.e. shear modulus)
   * @param E: Green-Lagrange or small strain tensor
   */
  template <class s_t>
  inline static decltype(auto)
  evaluate_stress(const Real & lambda, const Real & mu, s_t && E) {
    return E.trace()*lambda * Strain_t::Identity() + 2*mu*E;
  }

  /**
   * return stress and tangent stiffness
   * @param lambda: First Lamé's constant
   * @param mu: Second Lamé's constant (i.e. shear modulus)
   * @param E: Green-Lagrange or small strain tensor
   * @param C: stiffness tensor (Piola-Kirchhoff 2 (or ) w.r.t to `E`)
   */
  template <class s_t>
  inline static decltype(auto)
  evaluate_stress(const Real & lambda, const Real & mu,
                  Tangent_t && C, s_t && E) {
    return std::make_tuple
      (std::move(evaluate_stress(lambda, mu, std::move(E))),
       std::move(C));
  }
};
```

The goal of such static member functions-only `structs` is to instantiate a set of function templates with consistent template parameters without repeating those parameters.

If more functionality is required, a `class` is more appropriate. If in doubt, make it a `class`.

For consistency with STL, you can use `struct` instead of `class` for functors and traits.

Inheritance

Composition is often more appropriate than inheritance. When using inheritance, make it `public`.

Definition:
When a sub-class inherits from a base class, it includes the definitions of all the data and operations that the parent base class defines. In practice, inheritance is used in two major ways in C++: implementation inheritance, in which actual code is inherited by the child, and *interface inheritance*, in which only method names are inherited.

Pros:
Implementation inheritance reduces code size by re-using the base class code as it specializes an existing type.

Because inheritance is a compile-time declaration, you and the compiler can understand the operation and detect errors. Interface inheritance can be used to programmatically enforce that a class expose a particular API. Again, the compiler can detect errors, in this case, when a class does not define a necessary method of the API.

Cons:
For implementation inheritance, because the code implementing a sub-class is spread between the base and the sub-class, it can be more difficult to understand an implementation. The sub-class cannot override functions that are not virtual, so the sub-class cannot change implementation.

Decision:
All inheritance should be `public`. If you want to do private inheritance, you should be including an instance of the base class as a member instead.

Do not overuse implementation inheritance. Composition is often more appropriate. Try to restrict use of inheritance to the "is-a" case: `Bar` subclasses `Foo` if it can reasonably be said that `Bar` "is a kind of" `Foo`.

Limit the use of protected to those member functions that might need to be accessed from subclasses. Note that *data members should be private*.

Explicitly annotate overrides of virtual functions or virtual destructors with exactly one of either the `override` or (less frequently) `override final` specifier. Do not use `virtual` when declaring an `override`. Rationale: A function or destructor marked `override` or `final` that is not an `override` of a base class virtual function will not compile, and this helps catch common errors. The specifiers serve as documentation; if no specifier is present, the reader has to check all ancestors of the class in question to determine if the function or destructor is `virtual` or not.

Multiple Inheritance

Only very rarely is multiple implementation inheritance actually useful. We allow multiple inheritance only when at most one of the base classes has an implementation; all other base classes must be *pure interface* classes.

Definition:
Multiple inheritance allows a sub-class to have more than one base class. We distinguish between base classes that are *pure interfaces* and those that have an *implementation*.

Pros:
Multiple implementation inheritance may let you re-use even more code than single inheritance (see *Inheritance*).

Cons:
Only very rarely is multiple *implementation* inheritance actually useful. When multiple implementation inheritance seems like the solution, you can usually find a different, more explicit, and cleaner solution.

Decision:
Multiple inheritance is allowed only when all superclasses, with the possible exception of the first one, are *pure interfaces*.

Note:
There is an *exception* to this rule on Windows.

Interfaces

Definition:
A class is a pure interface if it meets the following requirements:

- It has only public pure virtual (`= 0`) methods and static methods (but see below for destructor).
- It may not have non-static data members.
- It need not have any constructors defined. If a constructor is provided, it must take no arguments and it must be protected.
- If it is a subclass, it may only be derived from classes that satisfy these conditions.

An interface class can never be directly instantiated because of the pure virtual method(s) it declares. To make sure all implementations of the interface can be destroyed correctly, the interface must also declare a virtual destructor (in an exception to the first rule, this should not be pure). See *Stroustrup, The C++ Programming Language, 4th edition, 2014*, section 20.3 for details.

Operator Overloading

Overload operators judiciously.

Definition:
C++ permits user code to declare overloaded versions of the built-in operators using the `operator` keyword, so long as one of the parameters is a user-defined type. The `operator` keyword also permits user code to define new kinds of literals using `operator""`, and to define type-conversion functions such as `operator bool()`.

Pros:
Operator overloading can make code more concise and intuitive by enabling user-defined types to behave the same as built-in types. Overloaded operators are the idiomatic names for certain operations (e.g. ==, <, =, and <<), and adhering to those conventions can make user-defined types more readable and enable them to interoperate with libraries that expect those names.

User-defined literals are a very concise notation for creating objects of user-defined types.

Cons:

- Providing a correct, consistent, and unsurprising set of operator overloads requires some care, and failure to do so can lead to confusion and bugs.
- Overuse of operators can lead to obfuscated code, particularly if the overloaded operator's semantics don't follow convention.
- The hazards of function overloading apply just as much to operator overloading, if not more so.
- Operator overloads can fool our intuition into thinking that expensive operations are cheap, built-in operations.
- Finding the call sites for overloaded operators may require a search tool that's aware of C++ syntax, rather than e.g. grep.
- If you get the argument type of an overloaded operator wrong, you may get a different overload rather than a compiler error. For example, `foo < bar` may do one thing, while `&foo < &bar` does something totally different.
- Certain operator overloads are inherently hazardous. Overloading unary `&` can cause the same code to have different meanings depending on whether the overload declaration is visible. Overloads of `&&`, `||`, and `,` (comma) cannot match the evaluation-order semantics of the built-in operators.

- Operators are often defined outside the class, so there's a risk of different files introducing different definitions of the same operator. If both definitions are linked into the same binary, this results in undefined behavior, which can manifest as subtle run-time bugs.
- User-defined literals allow the creation of new syntactic forms that are unfamiliar even to experienced C++ programmers.

Decisions:

Define overloaded operators only if their meaning is obvious, unsurprising, and consistent with the corresponding built-in operators. For example, use `|` as a bitwise- or logical-or, not as a shell-style pipe.

Define operators only on your own types. More precisely, define them in the same headers, `.cc` files, and namespaces as the types they operate on. That way, the operators are available wherever the type is, minimising the risk of multiple definitions. If possible, avoid defining operators as templates, because they must satisfy this rule for any possible template arguments. If you define an operator, also define any related operators that make sense, and make sure they are defined consistently. For example, if you overload <, overload all the comparison operators, and make sure < and > never return true for the same arguments.

Prefer to define non-modifying binary operators as non-member functions. If a binary operator is defined as a class member, implicit conversions will apply to the right-hand argument, but not the left-hand one. It will confuse your users if `a < b` compiles but `b < a` doesn't.

Don't go out of your way to avoid defining operator overloads. For example, prefer to define ==, =, and <<, rather than `equals()`, `copy_from()`, and `print_to()`. Conversely, don't define operator overloads just because other libraries expect them. For example, if your type doesn't have a natural ordering, but you want to store it in a `std::set`, use a custom comparator rather than overloading <.

Do not overload &&, `||`, `,` (comma), or unary &.

Type conversion operators are covered in *Implicit Conversions*. The = operator is covered in *Copyable and Movable Types*. Overloading << for use with streams is covered in *Streams*. See also the rules on *function overloading*, which apply to operator overloading as well.

Access Control

Make data members `protected`, unless they are `static const` (and follow the *naming convention for constants*).

Declaration Order

Group similar declarations together, placing `public` parts earlier.

A class definition should usually start with a `public:` section, followed by `protected:`, then `private:`. Omit sections that would be empty.

Within each section, generally prefer grouping similar kinds of declarations together, and generally prefer the following order: types (including `using`, and nested `structs` and `classes`), constants, factory functions, constructors, assignment operators, destructor, all other methods, data members.

Do not put large method definitions inline in the class definition. Trivial, performance-critical, or template methods may be defined inline. See *Inline Functions* for more details.

3.5.4 Functions

Output Parameters

Prefer using return values rather than output parameters. If output-only parameters are used they should appear after input parameters.

The output(s) of a C++ function is/are naturally provided via a (tuple of) return value and sometimes via output parameters.

Prefer using return values and return value tuples over output parameters since they improve readability and oftentimes provide the same or better performance.

Parameters are either input to the function, output from the function, or both. Input parameters are usually values or const references, while output and input/output parameters will be references to non-const.

When ordering function parameters, put all input-only parameters before any output parameters. In particular, do not add new parameters to the end of the function just because they are new; place new input-only parameters before the output parameters.

This is not a hard-and-fast rule. Parameters that are both input and output (often classes/structs) muddy the waters, and, as always, consistency with related functions may require you to bend the rule.

Write Short Functions

Prefer small and focused functions.

We recognise that long functions are sometimes appropriate, so no hard limit is placed on functions length. If a function exceeds about 40 lines, think about whether it can be broken up without harming the structure of the program.

Even if your long function works perfectly now, someone modifying it in a few months may add new behaviour. This could result in bugs that are hard to find. Keeping your functions short and simple makes it easier for other people to read and modify your code.

You could find long and complicated functions when working with some code. Do not be intimidated by modifying existing code: if working with such a function proves to be difficult, you find that errors are hard to debug, or you want to use a piece of it in several different contexts, consider breaking up the function into smaller and more manageable pieces.

Reference Arguments

All input parameters passed by reference must be labelled `const`, Output and input/output parameters can be passed as references, *smart pointers*, or `std::optional`. **There are no raw pointers** within μSpectre, ever.

Definition:
In C, if a function needs to modify a variable, the parameter must use a pointer, e.g., `int foo(int *pval)`. In C++, the function can alternatively declare a reference parameter: `int foo(int &val)`.

Pros:
Defining a parameter as reference avoids ugly code like `(*pval)++`. Necessary for some applications like copy constructors. Makes it clear, unlike with pointers, that a null pointer is not a possible value.

Cons:
References can be confusing to absolute beginners, as they have value syntax but pointer semantics.

Decision:
The one hard rule in μSpectre is that no raw pointers will be tolerated (with the obvious exception of interacting

with third-party APIs). Pointers are to be considered a bug-generating relic of a darker time when `goto` statements were allowed to exist. If you need to mimic the questionable practice of passing a pointer that could be `nullptr` to indicate that there is no value, use `std::optional`.

Function Overloading

Use overloaded functions (including constructors) only if a reader looking at a call site can get a good idea of what is happening without having to first figure out exactly which overload is being called.

Definition:
You may write a function that takes a `const string&` and overload it with another that takes `const char*`. However, in this case consider `std::string_view` instead.

```
class MyClass {
 public:
  void Analyze(const string &text);
  void Analyze(const char *text, size_t textlen);
};
```

Pros:
Overloading can make code more intuitive by allowing an identically-named function to take different arguments. It may be necessary for templated code, and it can be convenient for Visitors.

Cons:
If a function is overloaded by the argument types alone, a reader may have to understand C++'s complex matching rules in order to tell what's going on. Also many people are confused by the semantics of inheritance if a derived class overrides only some of the variants of a function.

Decision:
You may overload a function when there are no semantic differences between variants, or when the differences are clear at the call site.

If you are overloading a function to support variable number of arguments of the same type, consider making it take a STL container so that the user can use an *initialiser list* to specify the arguments.

Default Arguments

Default arguments are allowed on non-virtual functions when the default is guaranteed to always have the same value. Follow the same restrictions as for *function overloading*, and prefer overloaded functions if the readability gained with default arguments doesn't outweigh the downsides below.

Pros:
Often you have a function that uses default values, but occasionally you want to override the defaults. Default parameters allow an easy way to do this without having to define many functions for the rare exceptions. Compared to overloading the function, default arguments have a cleaner syntax, with less boilerplate and a clearer distinction between 'required' and 'optional' arguments.

Cons:
Defaulted arguments are another way to achieve the semantics of overloaded functions, so all the *reasons not to overload functions* apply.

The defaults for arguments in a virtual function call are determined by the static type of the target object, and there's no guarantee that all overrides of a given function declare the same defaults.

Default parameters are re-evaluated at each call site, which can bloat the generated code. Readers may also expect the default's value to be fixed at the declaration instead of varying at each call.

Function pointers are confusing in the presence of default arguments, since the function signature often doesn't match the call signature. Adding function overloads avoids these problems.

Decision:

Default arguments are banned on virtual functions, where they don't work properly, and in cases where the specified default might not evaluate to the same value depending on when it was evaluated. (For example, don't write `void f(int n = counter++);`.)

In some other cases, default arguments can improve the readability of their function declarations enough to overcome the downsides above, so they are allowed.

Trailing Return Type Syntax

Use trailing return types only where using the ordinary syntax (leading return types) is impractical or much less readable.

Definition:

C++ allows two different forms of function declarations. In the older form, the return type appears before the function name. For example:

```
int foo(int x);
```

The new form, introduced in C++11, uses the auto keyword before the function name and a trailing return type after the argument list. For example, the declaration above could equivalently be written:

```
auto foo(int x) -> int;
```

The trailing return type is in the function's scope. This doesn't make a difference for a simple case like int but it matters for more complicated cases, like types declared in class scope or types written in terms of the function parameters.

Pros:

Trailing return types are the only way to explicitly specify the return type of a *lambda expression*. In some cases the compiler is able to deduce a lambda's return type, but not in all cases. Even when the compiler can deduce it automatically, sometimes specifying it explicitly would be clearer for readers.

Sometimes it's easier and more readable to specify a return type after the function's parameter list has already appeared. This is particularly true when the return type depends on template parameters. For example:

```
template <typename T, typename U>
auto add(T t, U u) -> decltype(t + u);
```

versus

```
template <typename T, typename U>
decltype(declval<T&>() + declval<U&>()) add(T t, U u);
```

Decision:

In most cases, continue to use the older style of function declaration where the return type goes before the function name. Use the new trailing-return-type form only in cases where it's required (such as lambdas) or where, by putting the type after the function's parameter list, it allows you to write the type in a much more readable way.

3.5.5 Ownership and linting

There are various tricks and utilities that we use to make C++ code more robust, and various ways we use C++ that may differ from what you see elsewhere.

Ownership and Smart Pointers

Prefer to have single, fixed owners for dynamically allocated objects. Prefer to transfer ownership with smart pointers.

Definition:

"Ownership" is a bookkeeping technique for managing dynamically allocated memory (and other resources). The owner of a dynamically allocated object is an object or function that is responsible for ensuring that it is deleted when no longer needed. Ownership can sometimes be shared, in which case the last owner is typically responsible for deleting it. Even when ownership is not shared, it can be transferred from one piece of code to another.

"Smart" pointers are classes that act like pointers, e.g. by overloading the * and -> operators. Some smart pointer types can be used to automate ownership bookkeeping, to ensure these responsibilities are met. `std::unique_ptr` is a smart pointer type introduced in C++11, which expresses exclusive ownership of a dynamically allocated object; the object is deleted when the `std::unique_ptr` goes out of scope. It cannot be copied, but can be moved to represent ownership transfer. `std::shared_ptr` is a smart pointer type that expresses shared ownership of a dynamically allocated object. `std::shared_ptrs` can be copied; ownership of the object is shared among all copies, and the object is deleted when the last `std::shared_ptr` is destroyed.

Pros:

- It's virtually impossible to manage dynamically allocated memory without some sort of ownership logic.
- Transferring ownership of an object can be cheaper than copying it (if copying it is even possible).
- Transferring ownership can be simpler than 'borrowing' a pointer or reference, because it reduces the need to coordinate the lifetime of the object between the two users.
- Smart pointers can improve readability by making ownership logic explicit, self-documenting, and unambiguous.
- Smart pointers can eliminate manual ownership bookkeeping, simplifying the code and ruling out large classes of errors.
- For const objects, shared ownership can be a simple and efficient alternative to deep copying.

Cons:

- Ownership must be represented and transferred via smart pointers. Pointer semantics are more complicated than value semantics, especially in APIs: you have to worry not just about ownership, but also aliasing, lifetime, and mutability, among other issues.
- The performance costs of value semantics are often overestimated, so the performance benefits of ownership transfer might not justify the readability and complexity costs.
- APIs that transfer ownership force their clients into a single memory management model.
- Code using smart pointers is less explicit about where the resource releases take place.
- Shared ownership can be a tempting alternative to careful ownership design, obfuscating the design of a system.
- Shared ownership requires explicit bookkeeping at run-time, which can be costly.
- In some cases (e.g. cyclic references), objects with shared ownership may never be deleted.

Decision:

If dynamic allocation is necessary, prefer to keep ownership with the code that allocated it. If other code needs momentary access to the object (i.e., there is no risk of the other code accessing it later, after the object may have been destroyed), consider passing it a reference without transferring ownership. Prefer to use `std::unique_ptr` to make ownership transfer explicit. For example:

```
std::unique_ptr<Foo> FooFactory();
void FooConsumer(std::unique_ptr<Foo> ptr);
```

Do not design your code to use shared ownership without a very good reason. One such reason is to avoid expensive copy operations. If you do use shared ownership, prefer to use `std::shared_ptr`.

Never use `std::auto_ptr` it has no longer any value. Instead, use `std::unique_ptr`.

cpplint

Use `cpplint.py` to detect style errors.

`cpplint.py` is a tool that reads a source file and identifies many style errors. It is not perfect, and has both false positives and false negatives, but it is still a valuable tool. False positives can be ignored by putting `// NOLINT` at the end of the line or `// NOLINTNEXTLINE` in the previous line.

3.5.6 Other C++ Features

Rvalue References

Use rvalue references to define move constructors and move assignment operators, or for perfect forwarding.

Definition:

Rvalue references are a type of reference that can only bind to temporary objects. The syntax is similar to traditional reference syntax. For example, `void f(string&& s);` declares a function whose argument is an rvalue reference to a `string`.

Pros:

- Defining a move constructor (a constructor taking an rvalue reference to the class type) makes it possible to move a value instead of copying it. If `v1` is a `std::vector<string>`, for example, then auto `v2(std::move(v1))` will probably just result in some simple pointer manipulation instead of copying a large amount of data. In some cases this can result in a major performance improvement.
- Rvalue references make it possible to write a generic function wrapper that forwards its arguments to another function, and works whether or not its arguments are temporary objects. (This is sometimes called "perfect forwarding".)
- Rvalue references make it possible to implement types that are movable but not copyable, which can be useful for types that have no sensible definition of copying but where you might still want to pass them as function arguments, put them in containers, etc.
- `std::move` is necessary to make effective use of some standard-library types, such as `std::unique_ptr`.

Decision:

Use rvalue references to define move constructors and move assignment operators (as described in *Copyable and Movable Types*) and, in conjunction with `std::forward`, to support perfect forwarding. You may use `std::move` to express moving a value from one object to another rather than copying it.

Friends

We allow use of `friend` classes and functions, within reason.

Friends should usually be defined in the same file so that the reader does not have to look in another file to find uses of the private members of a class. A common use of friend is to have a `FooBuilder` class be a friend of `Foo` so that it can construct the inner state of `Foo` correctly, without exposing this state to the world.

Friends extend, but do not break, the encapsulation boundary of a class. In some cases this is better than making a member public when you want to give only one other class access to it. However, most classes should interact with other classes solely through their public members.

Exceptions

We use C++ exceptions extensively.

Pros:

- Exceptions allow higher levels of an application to decide how to handle "can't happen" failures in deeply nested functions, without the obscuring and error-prone bookkeeping of error codes.
- Exceptions are used by most other modern languages. Using them in C++ would make it more consistent with Python, Java, and the C++ that others are familiar with.
- Some third-party C++ libraries use exceptions, and turning them off internally makes it harder to integrate with those libraries.
- Exceptions are the only way for a constructor to fail. We can simulate this with a factory function or an `initialise()` method, but these require heap allocation or a new "invalid" state, respectively.
- Exceptions are really handy in testing frameworks.

Cons:

- When you add a throw statement to an existing function, you must examine all of its transitive callers. Either they must make at least the basic exception safety guarantee, or they must never catch the exception and be happy with the program terminating as a result. For instance, if `f()` calls `g()` calls `h()`, and `h` throws an exception that `f` catches, `g` has to be careful or it may not clean up properly.
- More generally, exceptions make the control flow of programs difficult to evaluate by looking at code: functions may return in places you don't expect. This causes maintainability and debugging difficulties. You can minimise this cost via some rules on how and where exceptions can be used, but at the cost of more that a developer needs to know and understand.
- Exception safety requires both RAII and different coding practices. Lots of supporting machinery is needed to make writing correct exception-safe code easy. Further, to avoid requiring readers to understand the entire call graph, exception-safe code must isolate logic that writes to persistent state into a "commit" phase. This will have both benefits and costs (perhaps where you're forced to obfuscate code to isolate the commit). Allowing exceptions would force us to always pay those costs even when they're not worth it.
- Turning on exceptions adds data to each binary produced, increasing compile time (probably slightly) and possibly increasing address space pressure.

Decision: On their face, the benefits of using exceptions outweigh the costs, especially in new projects. Especially in a computational project, were we are perfectly happy to terminate if an exception is thrown.

There is an *exception* to this rule (no pun intended) for Windows code.

noexcept

Specify `noexcept` when it is useful and correct.

Definition:
The `noexcept` specifier is used to specify whether a function will throw exceptions or not. If an exception escapes from a function marked `noexcept`, the program crashes via `std::terminate`.

The `noexcept` operator performs a compile-time check that returns true if an expression is declared to not throw any exceptions.

Pros:

- Specifying move constructors as `noexcept` improves performance in some cases, e.g. `std::vector<T>::resize()` moves rather than copies the objects if `T`'s move constructor is `noexcept`.
- Specifying `noexcept` on a function can trigger compiler optimisations in environments where exceptions are enabled, e.g. compiler does not have to generate extra code for stack-unwinding, if it knows that no exceptions can be thrown due to a `noexcept` specifier.

Cons:

- It's hard, if not impossible, to undo `noexcept` because it eliminates a guarantee that callers may be relying on, in ways that are hard to detect.

Decision:
You should use `noexcept` when it is useful for performance if it accurately reflects the intended semantics of your function, i.e. that if an exception is somehow thrown from within the function body then it represents a fatal error. You can assume that `noexcept` on move constructors has a meaningful performance benefit. If you think there is significant performance benefit from specifying `noexcept` on some other function, feel free to use it.

Run-Time Type Information (RTTI)

When possible, avoid using Run Time Type Information (RTTI).

Definition:
RTTI allows a programmer to query the C++ class of an object at run time. This is done by use of `typeid` or `dynamic_cast`.

Cons:
Querying the type of an object at run-time frequently means a design problem. Needing to know the type of an object at runtime is often an indication that the design of your class hierarchy is flawed.

Undisciplined use of RTTI makes code hard to maintain. It can lead to type-based decision trees or switch statements scattered throughout the code, all of which must be examined when making further changes.

Pros:
RTTI can be very useful when interacting with duck-typed languages (like python) and when implementing efficient containers with polymorphic interfaces, see, e.g., µSpectre's `FieldMap` implementation.

RTTI can be useful in some unit tests. For example, it is useful in tests of factory classes where the test has to verify that a newly created object has the expected dynamic type. It is also useful in managing the relationship between objects and their mocks.

RTTI is useful when considering multiple abstract objects. Consider

```
bool Base::Equal(Base* other) = 0;
bool Derived::Equal(Base* other) {
  Derived* that = dynamic_cast<Derived*>(other);
  if (that == nullptr) {
```

(continues on next page)

(continued from previous page)

```
    return false;
  }
  ...
}
```

Decision:

RTTI has legitimate uses but is prone to abuse, so you must be careful when using it. You may use it freely in unit tests, but avoid it when possible in other code. In particular, think twice before using RTTI in new code. If you find yourself needing to write code that behaves differently based on the class of an object, consider one of the following alternatives to querying the type:

- Virtual methods are the preferred way of executing different code paths depending on a specific subclass type. This puts the work within the object itself.
- If the work belongs outside the object and instead in some processing code, consider a double-dispatch solution, such as the Visitor design pattern. This allows a facility outside the object itself to determine the type of class using the built-in type system.

When the logic of a program guarantees that a given instance of a base class is in fact an instance of a particular derived class, then a `dynamic_cast` may be used freely on the object. Usually one can use a `static_cast` as an alternative in such situations.

Decision trees based on type are a strong indication that your code is on the wrong track.

```
if (typeid(*data) == typeid(D1)) {
  ...
} else if (typeid(*data) == typeid(D2)) {
  ...
} else if (typeid(*data) == typeid(D3)) {
...
```

Code such as this usually breaks when additional subclasses are added to the class hierarchy. Moreover, when properties of a subclass change, it is difficult to find and modify all the affected code segments.

Do not hand-implement an RTTI-like workaround. The arguments against RTTI apply just as much to workarounds like class hierarchies with type tags. Moreover, workarounds disguise your true intent.

Casting

Use C++-style casts like `static_cast<float>(double_value)`, or brace initialisation for conversion of arithmetic types like `int64 y{int64{1} << 42}`. Do not use cast formats like `int y{(int)x}` or `int y{int(x)}` (but the latter is okay when invoking a constructor of a class type).

Definition:

C++ introduced a different cast system from C that distinguishes the types of cast operations.

Pros:

The problem with C casts is the ambiguity of the operation; sometimes you are doing a conversion (e.g., `(int)3.5`) and sometimes you are doing a cast (e.g., `(int)"hello"`). Brace initialisation and C++ casts can often help avoid this ambiguity. Additionally, C++ casts are more visible when searching for them.

Cons:

The C++-style cast syntax is verbose

Decision:

Do not use C-style casts. Instead, use these C++-style casts when explicit type conversion is necessary.

- Use brace initialisation to convert arithmetic types (e.g. `int64{x}`). This is the safest approach because code will not compile if conversion can result in information loss. The syntax is also concise.
- Use `static_cast` as the equivalent of a C-style cast that does value conversion, when you need to explicitly up-cast a pointer from a class to its superclass, or when you need to explicitly cast a pointer from a superclass to a subclass. In this last case, you must be sure your object is actually an instance of the subclass.
- Use `const_cast` to remove the `const` qualifier (see *Use of const*). **This indicates a serious design flaw if it happens in µSpectre and is to be considered a bug**. Only use this if third-party libraries force you to.
- Use `reinterpret_cast` to do unsafe conversions of pointer types to and from integer and other pointer types. Use this only if you know what you are doing and you understand the aliasing issues.

See the *RTTI* section for guidance on the use of `dynamic_cast`.

Streams

Use streams where appropriate, and stick to "simple" usages. Overload << for streaming only for types representing values, and write only the user-visible value, not any implementation details.

Definition:
Streams are the standard I/O abstraction in C++, as exemplified by the standard header `<iostream>`.

Pros:
The << and >> stream operators provide an API for formatted I/O that is easily learned, portable, reusable, and extensible. `printf`, by contrast, doesn't even support string, to say nothing of user-defined types, and is very difficult to use portably. `printf` also obliges you to choose among the numerous slightly different versions of that function, and navigate the dozens of conversion specifiers.

Streams provide first-class support for console I/O via `std::cin`, `std::cout`, `std::cerr`, and `std::clog`. The C APIs do as well, but are hampered by the need to manually buffer the input.

Cons:

- Stream formatting can be configured by mutating the state of the stream. Such mutations are persistent, so the behaviour of your code can be affected by the entire previous history of the stream, unless you go out of your way to restore it to a known state every time other code might have touched it. User code can not only modify the built-in state, it can add new state variables and behaviours through a registration system.
- It is difficult to precisely control stream output, due to the above issues, the way code and data are mixed in streaming code, and the use of operator overloading (which may select a different overload than you expect).
- The streams API is subtle and complex, so programmers must develop experience with it in order to use it effectively.
- Resolving the many overloads of << is extremely costly for the compiler. When used pervasively in a large code base, it can consume as much as 20% of the parsing and semantic analysis time.

Decision:
Use streams only when they are the best tool for the job. This is typically the case when the I/O is ad-hoc, local, human-readable, and targeted at other developers rather than end-users. Be consistent with the code around you, and with the code base as a whole; if there's an established tool for your problem, use that tool instead. In particular, logging libraries are usually a better choice than `std::cerr` or `std::clog` for diagnostic output.

Overload << as a streaming operator for your type only if your type represents a value, and << writes out a human-readable string representation of that value. Avoid exposing implementation details in the output of <<; if you need to print object internals for debugging, use named functions instead (a method named `debug_string()` is the most common convention).

Preincrement and Predecrement

Use prefix form (`++i`) of the increment and decrement operators with iterators and other template objects.

Definition:
When a variable is incremented (`++i` or `i++`) or decremented (`--i` or `i--`) and the value of the expression is not used, one must decide whether to pre-increment (decrement) or post-increment (decrement).

Pros:
When the return value is ignored, the "pre" form (`++i`) is never less efficient than the "post" form (`i++`), and is often more efficient. This is because post-increment (or decrement) requires a copy of `i` to be made, which is the value of the expression. If `i` is an iterator or other non-scalar type, copying `i` could be expensive. Since the two types of increment behave the same when the value is ignored, why not just always pre-increment?

Cons:
The tradition developed, in C, of using post-increment when the expression value is not used, especially in for loops. Some find post-increment easier to read, since the "subject" (`i`) precedes the "verb" (`++`), just like in English. This is a dumb tradition and should be abolished.

Decision:
If the return value is ignored, a post-increment (post-decrement) is a bug.

Use of const

Use `const` doggedly whenever it makes is correct. With C++11, `constexpr` is a better choice for some uses of `const`.

Definition:
Declared variables and parameters can be preceded by the keyword `const` to indicate the variables are not changed (e.g., `const int foo`). Class functions can have the `const` qualifier to indicate the function does not change the state of the class member variables (e.g., `class Foo { int Bar(char c) const; };`).

Pros:
Easier for people to understand how variables are being used. Allows the compiler to do better type checking, and, conceivably, generate better code. Helps people convince themselves of program correctness because they know the functions they call are limited in how they can modify your variables. Helps people know what functions are safe to use without locks in multi-threaded programs.

`const` is viral: if you pass a `const` variable to a function, that function must have `const` in its prototype.

Cons:
`const` can be problem when calling library functions, and require `const_cast`.

Decision:
const variables, data members, methods and arguments add a level of compile-time type checking; it is better to detect errors as soon as possible. Therefore we strongly recommend that you use `const` whenever it is possible to do so:

- If a function guarantees that it will not modify an argument passed by reference, the corresponding function parameter should be a reference-to-const (`const T&`).
- Declare methods to be `const` whenever possible. Accessors should almost always be `const`. Other methods should be `const` if they do not modify any data members, do not call any non-`const` methods, and do not return a non-`const` reference to a data member.
- Consider making data members `const` whenever they do not need to be modified after construction.

The `mutable` keyword is allowed but is unsafe when used with threads, so thread safety should be carefully considered first.

Use of `constexpr`

In C++11, use `constexpr` to define true constants or to ensure constant initialisation.

Definition:
Some variables can be declared `constexpr` to indicate the variables are true constants, i.e. fixed at compilation/link time. Some functions and constructors can be declared `constexpr` which enables them to be used in defining a `constexpr` variable.

Pros:
Use of `constexpr` enables definition of constants with floating-point expressions rather than just literals; definition of constants of user-defined types; and definition of constants with function calls.

Decision:
`constexpr` definitions enable a more robust specification of the constant parts of an interface. Use `constexpr` to specify true constants and the functions that support their definitions. You can use `constexpr` to force inlining of functions.

Integer Types

We do not use the built-in C++ integer types in μSpectre, rather the alias `Int`. If a part needs a variable of a different size, use a precise-width integer type from <cstdint>, such as `int16_t`. If your variable represents a value that could ever be greater than or equal to 2^{31} (2GiB), use a 64-bit type such as `int64_t`. Keep in mind that even if your value won't ever be too large for an `Int`, it may be used in intermediate calculations which may require a larger type. When in doubt, choose a larger type.

Definition:
μSpectre does not specify the size of `Int`. Assume it's 32 bits.

Pros:
Uniformity of declaration.

Cons:
The sizes of integral types in C++ can vary based on compiler and architecture.

Decision:
<cstdint> defines types like `int16_t`, `uint32_t`, `int64_t`, etc. You should always use those in preference to short, unsigned long long and the like, when you need a guarantee on the size of an integer. When appropriate, you are welcome to use standard types like `size_t` and `petrify_t`.

We use `Int` very often, for integers we know are not going to be too big, e.g., loop counters. Use plain old `Int` for such things. You should assume that an `Int` is at least 32 bits, but don't assume that it has more than 32 bits. If you need a 64-bit integer type, use `int64_t` or `uint64_t`.

For integers we know can be "big", use `int64_t`.

You should not use the unsigned integer types such as `uint32_t`, unless there is a valid reason such as representing a bit pattern rather than a number, or you need defined overflow modulo 2. In particular, do not use unsigned types to say a number will never be negative. Instead, use assertions for this.

If your code is a container that returns a size, be sure to use a type that will accommodate any possible usage of your container. When in doubt, use a larger type rather than a smaller type.

Use care when converting integer types. Integer conversions and promotions can cause undefined behaviour, leading to security bugs and other problems.

On Unsigned Integers

Unsigned integers are good for representing bitfields and modular arithmetic. Because of historical accident, the C++ standard also uses unsigned integers to represent the size of containers - many members of the standards body believe

this to be a mistake, but it is effectively impossible to fix at this point. The fact that unsigned arithmetic doesn't model the behaviour of a simple integer, but is instead defined by the standard to model modular arithmetic (wrapping around on overflow/underflow), means that a significant class of bugs cannot be diagnosed by the compiler. In other cases, the defined behaviour impedes optimisation.

That said, mixing signedness of integer types is responsible for an equally large class of problems. The best advice we can provide: try to use iterators and containers rather than pointers and sizes, try not to mix signedness, and try to avoid unsigned types (except for representing bitfields or modular arithmetic). Do not use an unsigned type merely to assert that a variable is non-negative.

Preprocessor Macros

Avoid defining macros, especially in headers; prefer inline functions, enums, and const variables. Do not use macros to define pieces of a C++ API. Be aware that if you do not have a **very** good reason to submit code with a macro, it will likely be rejected.

Macros mean that the code you see is not the same as the code the compiler sees. This can introduce unexpected behaviour, especially since macros have global scope.

The problems introduced by macros are especially severe when they are used to define pieces of a C++ API, and still more so for public APIs. Every error message from the compiler when developers incorrectly use that interface now must explain how the macros formed the interface. Refactoring and analysis tools have a dramatically harder time updating the interface. As a consequence, we specifically disallow using macros in this way. For example, avoid patterns like:

```
class WOMBAT_TYPE(Foo) {
  // ...

 public:
  EXPAND_PUBLIC_WOMBAT_API(Foo)

  EXPAND_WOMBAT_COMPARISONS(Foo, ==, <)
};
```

Luckily, macros are not nearly as necessary in C++ as they are in C. Instead of using a macro to inline performance-critical code, use an inline function. Instead of using a macro to store a constant, use a `const` or `constexpr` variable. Instead of using a macro to "abbreviate" a long variable name, use a reference. Instead of using a macro to conditionally compile code … well, don't do that at all (except, of course, for the `#define` guards to prevent double inclusion of header files, and packages such as MPI). It makes testing much more difficult.

Macros can do things these other techniques cannot, and you do see them in the code base, especially in the lower-level libraries. And some of their special features (like stringifying, concatenation, and so forth) are not available through the language proper. But before using a macro, consider carefully whether there's a non-macro way to achieve the same result. If you need to use a macro to define an interface, discuss it with the community in an *issue <https://gitlab.com/muspectre/muspectre/issues>>`_*.

The following usage pattern will avoid many problems with macros; if you use macros, follow it whenever possible:

- Don't define macros in a `.hh` file.
- `#define` macros right before you use them, and `#undef` them right after.
- Do not just `#undef` an existing macro before replacing it with your own; instead, pick a name that's likely to be unique.
- Try not to use macros that expand to unbalanced C++ constructs, or at least document that behaviour well.
- Prefer not using `##` to generate function/class/variable names.

Exporting macros from headers (i.e. defining them in a header without `#undef`ing them before the end of the header) is extremely strongly discouraged. If you do export a macro from a header, it must have a globally unique name. To achieve this, it must be named with a prefix consisting of your project's namespace name (but upper case).

`0` and `nullptr`/`NULL`

Use `0` for integers, `0.` for reals, `nullptr` for pointers, and `'\0'` for chars.

For pointers (address values), there is a choice between `0`, `NULL`, and `nullptr`. µSpectre only accepts `nullptr`, as this provides type-safety.

Use `'\0'` for the null character. Using the correct type makes the code more readable.

sizeof

Prefer `sizeof(varname)` to `sizeof(type)`.

Use `sizeof(varname)` when you take the size of a particular variable. `sizeof(varname)` will update appropriately if someone changes the variable type either now or later. You may use `sizeof(type)` for code unrelated to any particular variable, such as code that manages an external or internal data format where a variable of an appropriate C++ type is not convenient.

```
Struct data;
memset(&data, 0, sizeof(data));

memset(&data, 0, sizeof(Struct));

if (raw_size < sizeof(int)) {
  LOG(ERROR) << "compressed record not big enough for count: " << raw_size;
  return false;
}
```

auto

Use auto to avoid type names that are noisy, obvious, or unimportant - cases where the type doesn't aid in clarity for the reader. Continue to use manifest type declarations only when it helps readability or you wish to override the type (important in the context of expression templates, see Eigen C++11 and the auto keyword).

Pros:

- C++ type names can be long and cumbersome, especially when they involve templates or namespaces.
- Long type names hinder readability.
- When a C++ type name is repeated within a single declaration or a small code region, the repetition hinders readability and breaks the *DRY* principle.
- It is sometimes safer to let the type be specified by the type of the initialisation expression, since that avoids the possibility of unintended copies or type conversions.
- Allows the use of universal references `auto &&` which allow to write efficient template expression code without sacrificing readability.

Cons:

- Sometimes code is clearer when types are manifest, especially when a variable's initialisation depends on things that were declared far away. In expressions like:

```
auto foo = x.add_foo();
auto i = y.Find(key);
```

- it may not be obvious what the resulting types are if the type of `y` isn't very well known, or if `y` was declared many lines earlier.
- Programmers have to understand the difference between `auto` and `const auto&` or they'll get copies when they didn't mean to.

Decision:

`auto` is highly encouraged when it increases readability and reduces redundant code repetitions, particularly as described below. Not using `auto` in these conditions is to be considered a bug. Never initialise an `auto`-typed variable with a braced initialiser list.

Typical example cases where `auto` is appropriate:

- For iterators and other long/cluttery type names, particularly when the type is clear from context (calls to `find`, `begin`, or `end` for instance).
- When the type is clear from local context (in the same expression or within a few lines). Initialisation of a pointer or smart pointer with calls to `new` and `std::make_unique` commonly falls into this category, as does use of `auto` in a range-based loop over a container whose type is spelled out nearby.
- When the type doesn't matter because it isn't being used for anything other than equality comparison.
- When iterating over a map with a range-based loop (because it is often assumed that the correct type is `std::pair<KeyType, ValueType>` whereas it is actually `std::pair<const KeyType, ValueType>`). This is particularly well paired with local key and value aliases for `.first` and `.second` (often const-ref).

```
for (const auto& item : some_map) {
  const KeyType& key = item.first;
  const ValType& value = item.second;
  // The rest of the loop can now just refer to key and value,
  // a reader can see the types in question, and we've avoided
  // the too-common case of extra copies in this iteration.
}
```

Braced Initialiser List

You may use braced initialiser lists.

In C++03, aggregate types (arrays and structs with no constructor) could be initialised with braced initialiser lists.

```
struct Point { int x; int y; };
Point p = {1, 2};
```

In C++11, this syntax was generalised, and any object type can now be created with a braced initialiser list, known as a braced-init-list in the C++ grammar. Here are a few examples of its use.

```
// Vector takes a braced-init-list of elements.
std::vector<string> v{"foo", "bar"};

// Basically the same, ignoring some small technicalities.
// You may choose to use either form.
std::vector<string> v = {"foo", "bar"};
```

(continues on next page)

(continued from previous page)

```
// Usable with 'new' expressions.
auto p = new std::vector<string>{"foo", "bar"};

// A map can take a list of pairs. Nested braced-init-lists work.
std::map<int, string> m = {{1, "one"}, {2, "2"}};

// A braced-init-list can be implicitly converted to a return type.
std::vector<int> test_function() { return {1, 2, 3}; }

// Iterate over a braced-init-list.
for (int i : {-1, -2, -3}) {}

// Call a function using a braced-init-list.
void TestFunction2(std::vector<int> v) {}
TestFunction2({1, 2, 3});
```

A user-defined type can also define a constructor and/or assignment operator that take `std::initialiser_list<T>`, which is automatically created from braced-init-list:

```
class MyType {
 public:
  // std::initialiser_list references the underlying init list.
  // It should be passed by value.
  MyType(std::initialiser_list<int> init_list) {
    for (int i : init_list) append(i);
  }
  MyType& operator=(std::initialiser_list<int> init_list) {
    clear();
    for (int i : init_list) append(i);
  }
};
MyType m{2, 3, 5, 7};
```

Finally, brace initialisation can also call ordinary constructors of data types, even if they do not have `std::initialiser_list<T>` constructors.

```
double d{1.23};
// Calls ordinary constructor as long as MyOtherType has no
// std::initialiser_list constructor.
class MyOtherType {
 public:
  explicit MyOtherType(string);
  MyOtherType(int, string);
};
MyOtherType m = {1, "b"};
// If the constructor is explicit, you can't use the "= {}" form.
MyOtherType m{"b"};
```

Never assign a braced-init-list to an `auto` local variable. In the single element case, what this means can be confusing.

```
auto d = {1.23};        // d is a std::initialiser_list<double>

auto d = double{1.23};  // Good but weird -- d is a double, not a std::initialiser_list.
```

See *Braced Initialiser List Format* for formatting.

Lambda expressions

Use lambda expressions where appropriate. Use explicit captures.

Definition:

Lambda expressions are a concise way of creating anonymous function objects. They're often useful when passing functions as arguments. For example:

```
std::sort(v.begin(), v.end(), [](int x, int y) {
  return Weight(x) < Weight(y);
});
```

They further allow capturing variables from the enclosing scope either explicitly by name, or implicitly using a default capture. Explicit captures require each variable to be listed, as either a value or reference capture:

```
int weight{3};
int sum{0};
// Captures `weight` by value and `sum` by reference.
std::for_each(v.begin(), v.end(), [weight, &sum](int x) {
  sum += weight * x;
});
```

Default captures implicitly capture any variable referenced in the lambda body, including this if any members are used:

```
const std::vector<int> lookup_table = ...;
std::vector<int> indices = ...;
// Captures `lookup_table` by reference, sorts `indices` by the value
// of the associated element in `lookup_table`.
std::sort(indices.begin(), indices.end(), [&](int a, int b) {
  return lookup_table[a] < lookup_table[b];
});
```

Lambdas were introduced in C++11 along with a set of utilities for working with function objects, such as the polymorphic wrapper `std::function`.

Pros:

- Lambdas are much more concise than other ways of defining function objects to be passed to STL algorithms, which can be a readability improvement.
- Appropriate use of default captures can remove redundancy and highlight important exceptions from the default.
- Lambdas, `std::function`, and `std::bind` can be used in combination as a general purpose callback mechanism; they make it easy to write functions that take bound functions as arguments.

Cons:

- Variable capture in lambdas can be a source of dangling-pointer bugs, particularly if a lambda escapes the current scope.
- Default captures by value can be misleading because they do not prevent dangling-pointer bugs. Capturing a pointer by value doesn't cause a deep copy, so it often has the same lifetime issues as capture by reference. This is especially confusing when capturing `this` by value, since the use of `this` is often implicit.

- It's possible for use of lambdas to get out of hand; very long nested anonymous functions can make code harder to understand.

Decision:

- Use lambda expressions where appropriate, with formatting as described below.
- Use explicit captures if the lambda may escape the current scope. For example, instead of:

```
{
  Foo foo;
  ...
  executor->schedule([&] { frobnicate(foo); })
  ...
}
// BAD! The fact that the lambda makes use of a reference to `foo` and
// possibly `this` (if `frobnicate` is a member function) may not be
// apparent on a cursory inspection. If the lambda is invoked after
// the function returns, that would be bad, because both `foo`
// and the enclosing object could have been destroyed.
```

prefer to write:

```
{
   Foo foo;
   ...
   executor->schedule([&foo] { frobnicate(foo); })
   ...
 }
 // BETTER - The compile will fail if `frobnicate` is a member
 // function, and it's clearer that `foo` is dangerously captured by
 // reference.
```

- Do not usese default capture by reference (`[&]`).
- Do not use default capture by value (`[=]`).
- Keep unnamed lambdas short. If a lambda body is more than maybe five lines long, prefer to give the lambda a name, or to use a named function instead of a lambda.
- Specify the return type of the lambda explicitly if that will make it more obvious to readers, as with `auto`.

Template metaprogramming

Template metaprogramming is our tool to obtain both generic and efficient code. It can be complicated, but efficiency is the top priority in the core of μSpectre.

Definition:
Template metaprogramming refers to a family of techniques that exploit the fact that the C++ template instantiation mechanism is Turing complete and can be used to perform arbitrary compile-time computation in the type domain.

Pros:
Template metaprogramming allows extremely flexible interfaces that are type safe and high performance. Facilities like the Boost unit test framework, `std::tuple`, `std::function`, and `Boost.Spirit` would be impossible without it.

Cons:

The techniques used in template metaprogramming are often obscure to anyone but language experts. Code that uses templates in complicated ways is demanding to read, and is hard to debug.

Template metaprogramming often leads to extremely poor compiler time error messages: even if an interface is simple, the complicated implementation details become visible when the user does something wrong.

Template metaprogramming interferes with large scale refactoring by making the job of refactoring tools harder. First, the template code is expanded in multiple contexts, and it's hard to verify that the transformation makes sense in all of them. Second, some refactoring tools work with an AST that only represents the structure of the code after template expansion. It can be difficult to automatically work back to the original source construct that needs to be rewritten.

Decision:

Template metaprogramming sometimes allows cleaner and easier-to-use interfaces than would be possible without it. It's best used in a small number of low level components where the extra maintenance burden is spread out over a large number of uses (i.e., the core of μSpectre, e.g. `MaterialMuSpectre` and the data structures).

If you use template metaprogramming, you should expect to put considerable effort into minimising and isolating the complexity. You should hide metaprogramming as an implementation detail whenever possible, so that user-facing headers are readable, and you should make sure that tricky code is especially well commented. You should carefully document how the code is used, and you should say something about what the "generated" code looks like. Pay extra attention to the error messages that the compiler emits when users make mistakes. The error messages are part of your user interface, and your code should be tweaked as necessary so that the error messages are understandable and actionable from a user point of view.

Boost

We try to depend on Boost as little as possible. The core library should not at all depend on Boost, while the tests use the Boost unit test framework. There is one exception: For users with ancient compilers, Boost is used to emulate `std::optional`. Do not add Boost dependencies.

Definition:

The Boost library collection is a popular collection of peer-reviewed, free, open-source C++ libraries.

Pros:

Boost code is generally very high-quality, is widely portable, and fills many important gaps in the C++ standard library, such as type traits and better binders.

Cons:

Boost can be tricky to install on certain systems

C++14

Use libraries and language extensions from C++14 when appropriate.

C++14 contains significant improvements both to the language and libraries.

Nonstandard Extensions

Nonstandard extensions to C++ may not be used unless needed to fix compiler bugs.

Compilers support various extensions that are not part of standard C++. Such extensions include GCC's `__attribute__`.

Cons:

- Nonstandard extensions do not work in all compilers. Use of nonstandard extensions reduces portability of code.
- Even if they are supported in all targeted compilers, the extensions are often not well-specified, and there may be subtle behaviour differences between compilers.
- Nonstandard extensions add to the language features that a reader must know to understand the code.

Decision:

Do not use nonstandard extensions.

Aliases

Public aliases are for the benefit of an API's user, and should be clearly documented.

Definition:

You can create names that are aliases of other entities:

```
template<class Param>
using Bar = Foo<Param>;
using other_namespace::Foo;
```

In μSpectre, aliases are created with the `using` keyword and never with `typedef`, because it provides a more consistent syntax with the rest of C++ and works with templates.

Like other declarations, aliases declared in a header file are part of that header's public API unless they're in a function definition, in the private portion of a class, or in an explicitly-marked internal namespace. Aliases in such areas or in `.cc` files are implementation details (because client code can't refer to them), and are not restricted by this rule.

Pros:

- Aliases can improve readability by simplifying a long or complicated name.
- Aliases can reduce duplication by naming in one place a type used repeatedly in an API, which might make it easier to change the type later.

Cons:

- When placed in a header where client code can refer to them, aliases increase the number of entities in that header's API, increasing its complexity.
- Clients can easily rely on unintended details of public aliases, making changes difficult.
- It can be tempting to create a public alias that is only intended for use in the implementation, without considering its impact on the API, or on maintainability.
- Aliases can create risk of name collisions
- Aliases can reduce readability by giving a familiar construct an unfamiliar name
- Type aliases can create an unclear API contract: it is unclear whether the alias is guaranteed to be identical to the type it aliases, to have the same API, or only to be usable in specified narrow ways

Decision:

Don't put an alias in your public API just to save typing in the implementation; do so only if you intend it to be used by your clients.

When defining a public alias, document the intent of the new name. This lets the user know whether they can treat the types as substitutable or whether more specific rules must be followed, and can help the implementation retain some degree of freedom to change the alias.

Don't put namespace aliases in your public API. (See also *Namespaces*).

For example, these aliases document how they are intended to be used in client code:

```
namespace mynamespace {
  // Used to store field measurements. DataPoint may change from Bar* to some␣
↪internal type.
  // Client code should treat it as an opaque pointer.
  using DataPoint = foo::Bar*;

  // A set of measurements. Just an alias for user convenience.
  using TimeSeries = std::unordered_set<DataPoint, std::hash<DataPoint>,␣
↪DataPointComparator>;
}  // namespace mynamespace
```

These aliases don't document intended use, and half of them aren't meant for client use:

```
namespace mynamespace {
  // Bad: none of these say how they should be used.
  using DataPoint = foo::Bar*;
  using std::unordered_set;  // Bad: just for local convenience
  using std::hash;           // Bad: just for local convenience
  typedef unordered_set<DataPoint, hash<DataPoint>, DataPointComparator> TimeSeries;
}  // namespace mynamespace
```

However, local convenience aliases are fine in function definitions, private sections of classes, explicitly marked internal namespaces, and in `.cc` files:

```
// In a ``.cc`` file
using foo::Bar;
```

3.5.7 Naming

The most important consistency rules are those that govern naming. The style of a name immediately informs us what sort of thing the named entity is: a type, a variable, a function, a constant, a macro, etc., without requiring us to search for the declaration of that entity. The pattern-matching engine in our brains relies a great deal on these naming rules.

Naming rules are pretty arbitrary, but we feel that consistency is more important than individual preferences in this area, so regardless of whether you find them sensible or not, the rules are the rules.

General Naming Rules

Names should be descriptive; avoid abbreviation.

Give as descriptive a name as possible, within reason. Do not worry about saving horizontal space as it is far more important to make your code immediately understandable by a new reader. Do not use abbreviations that are ambiguous or unfamiliar to readers outside your project, and do not abbreviate by deleting letters within a word. Abbreviations that would be familiar to someone outside your project with relevant domain knowledge are OK. As a rule of thumb, an abbreviation is probably OK if it's listed in Wikipedia.

A few good examples:

```
int price_count_reader;    // No abbreviation.
int nb_params;             // "nb" is a widespread convention.
int nb_dns_connections;    // Most people know what "DNS" stands for.
int lstm_size;             // "LSTM" is a common machine learning abbreviation.
```

A few bad examples

```
int n;                     // Meaningless.
int nerr;                  // Ambiguous abbreviation.
int n_comp_conns;          // Ambiguous abbreviation.
int wgc_connections;       // Only your group knows what this stands for.
int pc_reader;             // Lots of things can be abbreviated "pc".
int cstmr_id;              // Deletes internal letters.
FooBarRequestInfo fbri;    // Not even a word.
```

Note that certain universally-known abbreviations are OK, such as `i` for an iteration variable and `T` for a template parameter.

For some symbols, this style guide recommends names to start with a capital letter and to have a capital letter for each new word (a.k.a. "CamelCase"). When abbreviations appear in such names, prefer to capitalise every letter of the abbreviation (i.e. `FFTEngine`, not `FftEngine`).

Template parameters should follow the naming style for their category: type template parameters should follow the rules for type names, and non-type template parameters should follow the rules for `constexpr` variable names.

File Names

Filenames should be all lowercase and can include underscores (_). File names should indicate their content.

Examples of acceptable file names:

```
my_useful_class.cc # implementation of MyUsefulClass
my_useful_class.hh # interface and inlines of MyUsefulClass
fft_utils.hh       # declarations (header) for a bunch of FFT-related tools
test_my_useful_class.cc // unittests for MyUsefulClass
```

C++ files should end in `.cc` and header files should end in `.hh` (see also the section on *self-contained headers*).

Do not use filenames that already exist in `/usr/include` or widely used libraries, such as `db.hh`.

In general, make your filenames very specific. For example, use `http_server_logs.hh` rather than `logs.hh`. A very common case is to have a pair of files called, e.g., `foo_bar.hh` and `foo_bar.cc`, defining a class called `FooBar`.

Inline functions must be in a `.hh` file. If your inline functions are very short, they should go directly into your `.hh` file.

Type Names

Type names start with a capital letter and have a capital letter for each new word (CamelCase), with no underscores: `MyExcitingClass`, `MyExcitingEnum`.

The names of all types — classes, structs, type aliases, enums, and type template parameters — have the same naming convention. Type names should start with a capital letter and have a capital letter for each new word. No underscores. For example:

```
// classes and structs
class UrlTable { ...
class UrlTableTester { ...
struct UrlTableProperties { ...

// using aliases
using PropertiesMap_t = hash_map<UrlTableProperties *, string>;

// enums
enum UrlTableErrors { ...
```

There are two classes of very useful exception to above rules:

- When using aliases, please append `_t` for alias types `_ptr` for alias (smart) pointers and `_ref` for alias references and `std::reference_wrapper`s.
- In the specific context of type manipulations using the STL's type traits, it can help readability to follow the STL's convention of lowercase `type_names_with_undenscores_t`.

Variable Names

The names of variables (including function parameters) and data members are all lowercase, with underscores between words. For instance: `a_local_variable`, `a_struct_data_member`, `this->a_class_data_member`. Use class members exclusively with explicit mention of the `this` pointer. Common Variable names

For example:

```
string table_name;  // OK - uses underscore.

string tablename;   // Bad - missing underscore.
string tableName;   // Bad - mixed case.
```

`struct` and `class` Data Members

Data members of `structs` and classes, both static and non-static, are named like ordinary nonmember variables.

```
class TableInfo {
  ...
private:
  string unique_name;  // OK - underscore at end.
  static Field_t<FieldCollection> gradient;  // OK.

  string tablename;   // Bad - missing underscore.
};
```

See *Structs vs. Classes* for a discussion of when to use a struct versus a class.

`constexpr` and `const` Names

Variables declared `constexpr` and non-class template parameters are CamelCase, `const` are named like regular variables.

Function Names

Regular functions and methods are named like variables (lowercase `name_with_underscore`).

```
make_field()
get_nb_components()
compute_stresses()
```

Distinguish (member) functions that compute something at non-trivial cost from simple accessors to internal variables, and `constexpr static` accessors:

```
compute_stresses()  // not an accessor, does actual work
get_nb_components() // simple accessor
sdim()              // constexpr compile-time access
```

Namespace Names

The main namespace is `muSpectre`. All subordinate namespaces are CamelCase. Avoid collisions between nested namespaces and well-known top-level namespaces. If a namespace is only used to hide unnecessary internal complications, put it in `namespace internal` or `namespace *_internal` to indicate that these are implementation details that the user does not have to bother with.

Keep in mind that the *rule against abbreviated names* applies to namespaces just as much as variable names. Code inside the namespace seldom needs to mention the namespace name, so there's usually no particular need for abbreviation anyway.

Avoid nested namespaces that match well-known top-level namespaces. Collisions between namespace names can lead to surprising build breaks because of name lookup rules. In particular, do not create any nested `std` namespaces.

Enumerator Names

Enumerators (for both scoped and unscoped enums) should be named like `constexpr` variables (CamelCase).

Preferably, the individual enumerators should be named like constants. However, it is also acceptable to name them like macros. The enumeration name, UrlTableErrors (and AlternateUrlTableErrors), is a type, and therefore mixed case.

```
//! Material laws can declare which type of strain measure they require and
//! µSpectre will provide it
enum class StrainMeasure {
  Gradient,      //!< placement gradient (y/x)
  Infinitesimal, //!< small strain tensor .5(u + u)
  GreenLagrange, //!< Green-Lagrange strain .5(F·F - I)
  Biot,          //!< Biot strain
  Log,           //!< logarithmic strain
  Almansi,       //!< Almansi strain
  RCauchyGreen,  //!< Right Cauchy-Green tensor
  LCauchyGreen,  //!< Left Cauchy-Green tensor
```

(continues on next page)

(continued from previous page)

```
  no_strain_     //!< only for triggering static_assert
};
```

Note that the last case, `StrainMeasure::no_strain_` is deliberately not named like an `enum`, to indicate that it is an invalid state;

Macro Names

You're not really going to define a macro, are you? If you do, they're like this: `MY_MACRO_THAT_SCARES_SMALL_CHILDREN_AND_ADULTS_ALIKE.`

Please see the *description of macros*; in general macros should not be used. However, if they are absolutely needed, then they should be named with all capitals and underscores. Be ready to argue the need for a macro in your submission, and prepare yourself for rejection if you do not have an overwhelmingly convincing reason.

```
#define ROUND(x) ...
#define PI_ROUNDED 3.0
```

Exceptions to Naming Rules

Exceptions are classes and as such follow the *Type Names* rules (CamelCase), but additionally end in `Error`, e.g., `ProjectionError`.

3.5.8 Comments

Though a pain to write, comments are absolutely vital to keeping our code readable. The following rules describe what you should comment and where. But remember: while comments are very important, the best code is self-documenting. Giving sensible names to types and variables is much better than using obscure names that you must then explain through comments.

When writing your comments, write for your audience: the next contributor who will need to understand your code. Be generous — the next one may be you!

Comment Style

Use either the `//` or `/* ... */` syntax for comments that are only relevant within their context (local comments for the reader/maintainer of your code)

Use `//!`, `/** ... */`, and `//!<` syntax for doxygen comments (which will end up being compiled into the API *Reference*), see Doxygen for details on doxygen.

File Comments

Start each file with license boilerplate. e.g., for file `common.hh` authored by John Doe:

```
/**
 * @file   common.hh
 *
 * @author John Doe <John.Do@email.address>
 *
 * @date   01 May 2017
 *
 * @brief  Small prototypes of commonly used types throughout µSpectre
 *
 * @section  LICENSE
 *
 * Copyright © 2017 Till Junge, John Doe
 *
 * µSpectre is free software; you can redistribute it and/or
 * modify it under the terms of the GNU Lesser General Public License as
 * published by the Free Software Foundation, either version 3, or (at
 * your option) any later version.
 *
 * µSpectre is distributed in the hope that it will be useful, but
 * WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
 * General Public License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public License
 * along with µSpectre; see the file COPYING. If not, write to the
 * Free Software Foundation, Inc., 59 Temple Place - Suite 330,
 * Boston, MA 02111-1307, USA.
 *
 * Additional permission under GNU GPL version 3 section 7
 *
 * If you modify this Program, or any covered work, by linking or combining it
 * with proprietary FFT implementations or numerical libraries, containing parts
 * covered by the terms of those libraries' licenses, the licensors of this
 * Program grant you additional permission to convey the resulting work.
 */
```

Note on copyright: it is shared among the writers of the particular file, but chances are that in most cases, at least part of each new file contains ideas or even copied-and-pasted snippets from other files. Give the authors of those files also shared copyright.

File comments describe the contents of a file. If a file declares, implements, or tests exactly one abstraction that is documented by a comment at the point of declaration, file comments are not required. All other files must have file comments.

Every file should contain GPL boilerplate.

Do not duplicate comments in both the `.hh` and the .cc. Duplicated comments diverge.

Class Comments

Every non-obvious class declaration should have an accompanying comment that describes what it is for and how it should be used.

```
/**
 * Virtual base class for all fields. A field represents
 * meta-information for the per-pixel storage for a scalar, vector
 * or tensor quantity and is therefore the abstract class defining
 * the field. It is used for type and size checking at runtime and
 * for storage of polymorphic pointers to fully typed and sized
 * fields. `FieldBase` (and its children) are templated with a
 * specific `FieldCollection` (derived from
 * `muSpectre::FieldCollectionBase`). A `FieldCollection` stores
 * multiple fields that all apply to the same set of
 * pixels. Addressing and managing the data for all pixels is
 * handled by the `FieldCollection`.  Note that `FieldBase` does
 * not know anything about about mathematical operations on the
 * data or how to iterate over all pixels. Mapping the raw data
 * onto for instance Eigen maps and iterating over those is
 * handled by the `FieldMap`.
 */
template <class FieldCollection>
class FieldBase
{
  ...
};
```

The class comment should provide the reader with enough information to know how and when to use the class, as well as any additional considerations necessary to correctly use the class. Document the assumptions the class makes, if any. If an instance of the class can be accessed by multiple threads, take extra care to document the rules and invariants surrounding multithreaded use.

The class comment is often a good place for a small example code snippet demonstrating a simple and focused usage of the class.

When sufficiently separated (e.g. `.hh` and `.cc` files), comments describing the use of the class should go together with its interface definition; comments about the class operation and implementation should accompany the interface definition of the class's methods.

Function Comments

Declaration comments describe use of the function; comments at the definition of a function describe operation.

Function Declarations

Every function declaration should have comments immediately preceding it that describe what the function does and how to use it. These comments should be descriptive ("Opens the file") rather than imperative ("Open the file"); the comment describes the function, it does not tell the function what to do. In general, these comments do not describe how the function performs its task. Instead, that should be left to comments in the function definition.

Types of things to mention in comments at the function declaration:

- What the inputs and outputs are.
- For class member functions: whether the object remembers reference arguments beyond the duration of the method call, and whether it will free them or not.
- If the function allocates memory that the caller must free.
- Whether any of the arguments can be a null pointer.
- If there are any performance implications of how a function is used.
- If the function is re-entrant. What are its synchronisation assumptions?

Here is an example:

```
/** Returns an iterator for this table.  It is the client's
  * responsibility to delete the iterator when it is done with it,
  * and it must not use the iterator once the GargantuanTable object
  * on which the iterator was created has been deleted.
  *
  * The iterator is initially positioned at the beginning of the table.
  *
  * This method is equivalent to:
  *    Iterator* iter = table->NewIterator();
  *    iter->Seek("");
  *    return iter;
  * If you are going to immediately seek to another place in the
  * returned iterator, it will be faster to use NewIterator()
  * and avoid the extra seek.
  */
Iterator get_iterator() const;
```

However, do not be unnecessarily verbose or state the completely obvious.

When documenting function overrides, focus on the specifics of the override itself, rather than repeating the comment from the overridden function. In many of these cases, the override needs no additional documentation and thus only brief comments are required.

When commenting constructors and destructors, remember that the person reading your code knows what constructors and destructors are for, so comments that just say something like "destroys this object" are not useful. Document what constructors do with their arguments (for example, if they take ownership of pointers), and what cleanup the destructor does. If this is trivial, just name it (`default constructor`, `move constructor`, `destructor` etc).

Function Definitions

If there is anything tricky about how a function does its job, the function definition should have an explanatory comment. For example, in the definition comment you might describe any coding tricks you use, give an overview of the steps you go through, or explain why you chose to implement the function in the way you did rather than using a viable alternative. For instance, you might mention why it must acquire a lock for the first half of the function but why it is not needed for the second half.

Note you should not just repeat the comments given with the function declaration, in the `.hh` file or wherever. It's okay to recapitulate briefly what the function does, but the focus of the comments should be on how it does it.

Variable Comments

In general the actual name of the variable should be descriptive enough to give a good idea of what the variable is used for, but we require a short description for the API documentation.

Class Data Members

The purpose of each class data member (also called an instance variable or member variable) must be clear. If there are any invariants (special values, relationships between members, lifetime requirements) not clearly expressed by the type and name, they must be commented. However, if the type and name suffice (`int nb_events;`), a brief "number of events" is a sufficient comment:

```
protected:

  const std::string name; //!< the field's unique name
  const size_t nb_components; //!< number of components per entry

  //! reference to the collection this field belongs to
  const FieldCollection & collection;
  size_t pad_size; //!< size of padding region at end of buffer
```

Global Variables

All global variables should have a comment describing what they are, what they are used for, and (if unclear) why it needs to be global. For example:

```
constexpr Dim_t oneD{1}; //!< constant for a one-dimensional problem
constexpr Dim_t twoD{2}; //!< constant for a two-dimensional problem
constexpr Dim_t threeD{3}; //!< constant for a three-dimensional problem
constexpr Dim_t firstOrder{1}; //!< constant for vectors
constexpr Dim_t secondOrder{2}; //!< constant second-order tensors
constexpr Dim_t fourthOrder{4}; //!< constant fourth-order tensors
```

Implementation Comments

In your implementation you should have comments in tricky, non-obvious, interesting, or important parts of your code. Explanatory Comments

Tricky or complicated code blocks should have comments before them. Example:

```
/* original definition of the operator in de Geus et
 * al. (https://doi.org/10.1016/j.cma.2016.12.032). However,
 * they use a obscure definition of the double contraction
 * between fourth-order and second-order tensors that has a
 * built-in transpose operation (i.e., C = A:B <-> AB =
 * C , note the inverted  instead of ), here, we define
 * the double contraction without the transposition. As a
 * result, this Projection operator produces the transpose of de
 * Geus's */

for (Dim_t im = 0; im < DimS; ++im) {
  for (Dim_t j = 0; j < DimS; ++j) {
    for (Dim_t l = 0; l < DimS; ++l) {
      get(G, im, j, l, im) = xi(j)*xi(l);
    }
  }
}
```

Line Comments

Also, lines that are non-obvious should get a comment at the end of the line. These end-of-line comments should be separated from the code by at least one space. Example:

```
// If we have enough memory, mmap the data portion too.
mmap_budget = max<int64>(0, mmap_budget - index_->length());
if (mmap_budget >= data_size_ && !MmapData(mmap_chunk_bytes, mlock))
  return;  // Error already logged.
```

Note that there are both comments that describe what the code is doing, and comments that mention that an error has already been logged when the function returns.

If you have several comments on subsequent lines, it can often be more readable to line them up:

```
do_something();                   // Comment here so the comments line up.
do_somethingElseThatIsLonger();  // Two spaces between the code and the comment.
{ // One space before comment when opening a new scope is allowed,
  // thus the comment lines up with the following comments and code.
  do_somethingElse();  // Two spaces before line comments normally.
}
std::vector<string> list{
                    // Comments in braced lists describe the next element...
                    "First item",
                    // .. and should be aligned appropriately.
                    "Second item"};
do_something(); /* For trailing block comments, one space is fine. */
```

Self-describing code doesn't need a comment. The comment from the example above would be obvious:

```
if (!IsAlreadyProcessed(element)) {
  Process(element);
}
```

Punctuation, Spelling and Grammar

Pay attention to punctuation, spelling, and grammar; it is easier to read well-written comments than badly written ones.

Comments should be as readable as narrative text, with proper capitalisation and punctuation. In many cases, complete sentences are more readable than sentence fragments. Shorter comments, such as comments at the end of a line of code, can sometimes be less formal, but you should be consistent with your style.

Although it can be frustrating to have a code reviewer point out that you are using a comma when you should be using a semicolon, it is very important that source code maintain a high level of clarity and readability. Proper punctuation, spelling, and grammar help with that goal.

TODO Comments

Use TODO comments for code that is temporary, a short-term solution, or good-enough but not perfect.

TODOs should include the string TODO in all caps, followed by the name, e-mail address, Phabricator task number or other identifier of the person or issue with the best context about the problem referenced by the TODO. The main purpose is to have a consistent TODO that can be searched to find out how to get more details upon request. A TODO is not a commitment that the person referenced will fix the problem. Thus when you create a TODO with a name, it is almost always your name that is given.

```
// TODO(kl@gmail.com): Use a "*" here for concatenation operator.
// TODO(Zeke) change this to use relations.
// TODO(T1234): remove the "Last visitors" feature
```

If your TODO is of the form "At a future date do something" make sure that you either include a very specific date ("Fix by November 2005") or a very specific event ("Remove this code when all clients can handle XML responses.").

Deprecation Comments

Mark deprecated interface points with DEPRECATED comments.

You can mark an interface as deprecated by writing a comment containing the word DEPRECATED in all caps. The comment goes either before the declaration of the interface or on the same line as the declaration.

After the word DEPRECATED, write your name, e-mail address, or other identifier in parentheses.

A deprecation comment must include simple, clear directions for people to fix their call sites. In C++, you can implement a deprecated function as an inline function that calls the new interface point.

Marking an interface point DEPRECATED will not magically cause any call sites to change. If you want people to actually stop using the deprecated facility, you will have to fix the call sites yourself or recruit a crew to help you.

New code should not contain calls to deprecated interface points. Use the new interface point instead. If you cannot understand the directions, find the person who created the deprecation and ask them for help using the new interface point.

3.5.9 Formatting

Coding style and formatting are pretty arbitrary, but a project is much easier to follow if everyone uses the same style. Individuals may not agree with every aspect of the formatting rules, and some of the rules may take some getting used to, but it is important that all project contributors follow the style rules so that they can all read and understand everyone's code easily.

To help you format code correctly, Google has created a settings file for emacs.

Line Length

Each line of text in your code should be at most 80 characters long.

We recognise that this rule is controversial, but so much existing code already adheres to it, and we feel that consistency is important.

Pros:
Those who favour this rule argue that it is rude to force them to resize their windows and there is no need for anything longer. Some folks are used to having several code windows side-by-side, and thus don't have room to widen their windows in any case. People set up their work environment assuming a particular maximum window width, and 80 columns has been the traditional standard. Why change it?

Cons:
Proponents of change argue that a wider line can make code more readable. The 80-column limit is an hidebound throwback to 1960s mainframes; modern equipment has wide screens that can easily show longer lines.

Decision:
80 characters is the maximum.

Exception:
Comment lines can be longer than 80 characters if it is not feasible to split them without harming readability, ease of cut and paste or auto-linking – e.g. if a line contains an example command or a literal URL longer than 80 characters.

Exception:
A raw-string literal may have content that exceeds 80 characters. Except for test code, such literals should appear near the top of a file.

Exception:
An `#include` statement with a long path may exceed 80 columns.

Exception:
You needn't be concerned about *header guards* that exceed the maximum length.

Non-ASCII Characters

In comments and human-readable names in strings, non-ASCII characters should be used where they help readability, and must use UTF-8 formatting, e.g.

```
/**
  *  verification of resultant strains: subscript  for hard and
  *  for soft, N is nb_lays and N is nb_grid_pts, k is contrast
  *
  *     l = l = l + l = l+l
  *  =>  =  N/N +  (N-N)/N
  *
```

(continues on next page)

(continued from previous page)

```
   *   is constant across all layers
   *         =
   *  => E  = E
   *  =>  = 1/k
   *  =>  / (1/k N/N + (N-N)/N) =
   */
constexpr Real factor{1/contrast * Real(nb_lays)/nb_grid_pts[0]
    + 1.-nb_lays/Real(nb_grid_pts[0])};
```

```
template <Dim_t DimS, Dim_t DimM>
MaterialHyperElastoPlastic1<DimS, DimM>::
MaterialHyperElastoPlastic1(std::string name, Real young, Real poisson,
                            Real tau_y0, Real H)
  : Parent{name},
    plast_flow_field("cumulated plastic flow ", this->internal_fields),
    F_prev_field("Previous placement gradient F", this->internal_fields),
    be_prev_field("Previous left Cauchy-Green deformation b",
                  this->internal_fields),
    young{young}, poisson{poisson},
    lambda{Hooke::compute_lambda(young, poisson)},
    mu{Hooke::compute_mu(young, poisson)},
    K{Hooke::compute_K(young, poisson)},
    tau_y0{tau_y0}, H{H},
    // the factor .5 comes from equation (18) in Geers 2003
    // (https://doi.org/10.1016/j.cma.2003.07.014)
    C{0.5*Hooke::compute_C_T4(lambda, mu)},
    internal_variables{F_prev_field.get_map(), be_prev_field.get_map(),
        plast_flow_field.get_map()}
{}
```

You shouldn't use the C++11 `char16_t` and `char32_t` character types, since they're for non-UTF-8 text. For similar reasons you also shouldn't use `wchar_t`.

Spaces vs. Tabs

Use only spaces, and indent 2 spaces at a time.

We use spaces for indentation. Do not use tabs in your code. You should set your editor to emit spaces when you hit the tab key.

Function Declarations and Definitions

Return type on the same line as function name, parameters on the same line if they fit. Wrap parameter lists which do not fit on a single line as you would wrap arguments in a function call.

Functions look like this:

```
ReturnType ClassName::function_name(Type par_name1, Type par_name2) {
  do_something();
}
```

If you have too much text to fit on one line:

```
ReturnType ClassName::really_long_function_name(Type par_name1,
                                                Type par_name2,
                                                Type par_name3) {
  do_something();
}
```

or if you cannot fit even the first parameter, be reasonable, in the spirit of readability:

```
template<class FieldCollection, class EigenArray, class EigenConstArray,
          class EigenPlain, Map_t map_type,  bool ConstField>
typename MatrixLikeFieldMap<FieldCollection, EigenArray, EigenConstArray,
                            EigenPlain, map_type, ConstField>::const_reference
MatrixLikeFieldMap<FieldCollection, EigenArray, EigenConstArray, EigenPlain,
                   map_type, ConstField>::
operator[](const Ccoord & ccoord) const{
  size_t index{};
  index = this->collection.get_index(ccoord);
  return const_reference(this->get_ptr_to_entry(std::move(index)));
}
```

Some points to note:

- Choose good parameter names.
- If you cannot fit the return type and the function name on a single line, break between them.
- If you break after the return type of a function declaration or definition, do not indent.
- There is never a space between the function name and the open parenthesis.
- There is never a space between the parentheses and the parameters.
- The open curly brace is always on the end of the last line of the function declaration, not the start of the next line.
- The close curly brace is either on the last line by itself or on the same line as the open curly brace.
- There should be a space between the close parenthesis and the open curly brace.
- All parameters should be aligned if possible.
- Default indentation is 2 spaces.

Unused parameters that might not be obvious must comment out the variable name in the function definition:

```
class Shape {
 public:
  virtual void rotate(double radians) = 0;
};

class Circle : public Shape {
 public:
  void rotate(double radians) override;
};

void Circle::rotate(double /*radians*/) {}
```

```
// Bad - if someone wants to implement later, it's not clear what the
// variable means.
void Circle::rotate(double) {}
```

Attributes, and macros that expand to attributes, appear at the very beginning of the function declaration or definition, before the return type:

Lambda Expressions

Format parameters and bodies as for any other function, and capture lists like other comma-separated lists.

For by-reference captures, do not leave a space between the ampersand (&) and the variable name.

```
int x = 0;
auto x_plus_n = [&x](int n) -> int { return x + n; }
```

Short lambdas may be written inline as function arguments.

```
std::set<int> blacklist = {7, 8, 9};
std::vector<int> digits = {3, 9, 1, 8, 4, 7, 1};
digits.erase(std::remove_if(digits.begin(), digits.end(), [&blacklist](int i) {
               return blacklist.find(i) != blacklist.end();
             }),
             digits.end());
```

Function Calls

Either write the call all on a single line, wrap the arguments at the parenthesis, or use common sense to help readability. In the absence of other considerations, use the minimum number of lines, including placing multiple arguments on each line where appropriate.

Function calls have the following format:

```
result = do_something(argument1, argument2, argument3);
```

If the arguments do not all fit on one line, they should be broken up onto multiple lines, with each subsequent line aligned with the first argument. Do not add spaces after the open parenthesis or before the close parenthesis:

```
result = do_something(averyveryveryverylongargument1,
                      argument2, argument3);
```

Arguments may optionally all be placed on subsequent lines.

```
if (...) {
   ...
   ...
   if (...) {
     bool result = do_something
       (argument1, argument2,
        argument3, argument4);
     ...
   }
```

Put multiple arguments on a single line to reduce the number of lines necessary for calling a function unless there is a specific readability problem. Some find that formatting with strictly one argument on each line is more readable and simplifies editing of the arguments. However, we prioritise for the reader over the ease of editing arguments, and most readability problems are better addressed with the following techniques.

If having multiple arguments in a single line decreases readability due to the complexity or confusing nature of the expressions that make up some arguments, try creating variables that capture those arguments in a descriptive name:

```
int my_heuristic{scores[x] * y + bases[x]};
result = do_something(my_heuristic, x, y, z);
```

Or put the confusing argument on its own line with an explanatory comment:

```
result = do_something(scores[x] * y + bases[x],  // Score heuristic.
                      x, y, z);
```

If there is still a case where one argument is significantly more readable on its own line, then put it on its own line. The decision should be specific to the argument which is made more readable rather than a general policy.

Sometimes arguments form a structure that is important for readability. In those cases, feel free to format the arguments according to that structure:

```
// Transform the widget by a 3x3 matrix.
my_widget.transform(x1, x2, x3,
                    y1, y2, y3,
                    z1, z2, z3);
```

Braced Initialiser List Format

Format a *braced initialiser list* exactly like you would format a function call in its place.

If the braced list follows a name (e.g. a type or variable name), format as if the {} were the parentheses of a function call with that name. If there is no name, assume a zero-length name.

```
// Examples of braced init list on a single line.
return {foo, bar};
function_call({foo, bar});
std::pair<int, int> p{foo, bar};

// When you have to wrap.
some_function(
    {"assume a zero-length name before {"},
    some_other_function_parameter);
SomeType variable{
    some, other, values,
    {"assume a zero-length name before {"},
    SomeOtherType{
        "Very long string requiring the surrounding breaks.",
        some, other values},
    SomeOtherType{"Slightly shorter string",
                  some, other, values}};
SomeType variable{
    "This is too long to fit all in one line"};
MyType m = {  // Here, you could also break before {.
    superlongvariablename1,
    superlongvariablename2,
    {short, interior, list},
    {interiorwrappinglist,
     interiorwrappinglist2}};
```

Conditionals

```
Prefer no spaces inside parentheses. The if and else keywords belong on separate lines.

if (condition) {  // no spaces inside parentheses
  ...  // 2 space indent.
} else if (...) {  // The else goes on the same line as the closing brace.
  ...
} else {
  ...
}
```

Note that in all cases you must have a space between the if and the open parenthesis. You must also have a space between the close parenthesis and the curly brace.

```
if(condition) {   // Bad - space missing after IF.
if (condition){   // Bad - space missing before {.
if(condition){    // Doubly bad.
```

```
if (condition) {  // Good - proper space after IF and before {.
```

Short conditional statements may be written on one line if this enhances readability. You may use this only when the line is brief and the statement does not use the else clause. You must still use curly braces, as they exclude a particularly dumb class of bugs.

```
if (x == kFoo) {return new Foo()};
if (x == kBar) {return new Bar()};
```

This is not allowed when the if statement has an else:

```
// Not allowed - IF statement on one line when there is an ELSE clause
if (x) {do_this()};
else {do_that()};
```

Loops and Switch Statements

Switch statements must use braces for blocks. Annotate non-trivial fall-through between cases. Empty loop bodies should use `{continue;}`.

`case` blocks in switch statements have curly braces which should be placed as shown below.

If not conditional on an enumerated value, switch statements should always have a default case (in the case of an enumerated value, the compiler will warn you if any values are not handled). If the default case should never execute, treat this as an error. For example:

```
switch (form) {
case Formulation::finite_strain: {
  return StrainMeasure::Gradient;
  break;
}
case Formulation::small_strain: {
  return StrainMeasure::Infinitesimal;
  break;
```

(continues on next page)

(continued from previous page)

```
}
default:
  return StrainMeasure::no_strain_;
  break;
}
```

Braces are required even for single-statement loops.

```
for (int i = 0; i < kSomeNumber; ++i)
   std::cout << "I love you" << std::endl; // Bad!
```

```
for (int i = 0; i < kSomeNumber; ++i) {
  std::cout << "I take it back" << std::endl;
} // Good!
```

Pointer and Reference Expressions

No spaces around period or arrow. Pointer operators may have trailing spaces.

The following are examples of correctly-formatted pointer and reference expressions:

```
x = *p;
x = * p; // also ok
p = &x;
p = & x;
x = r.y;
x = r->y;
```

Note that:

- There are no spaces around the period or arrow when accessing a member.
- Pointer operators have no space after the * or &.

Boolean Expressions

When you have a boolean expression that is longer than the standard line length, be consistent in how you break up the lines.

In this example, the logical AND operator is always at the end of the lines:

```
if (this_one_thing > this_other_thing &&
    a_third_thing == a_fourth_thing &&
    yet_another && last_one) {
  ...
}
```

Note that when the code wraps in this example, both of the && logical AND operators are at the end of the line. This is more common, though wrapping all operators at the beginning of the line is also allowed. Feel free to insert extra parentheses judiciously because they can be very helpful in increasing readability when used appropriately.

Return Values

Do not needlessly surround the return expression with parentheses.

Use parentheses in `return expr;` only where you would use them in `x = expr;`.

```
return result;                  // No parentheses in the simple case.
// Parentheses OK to make a complex expression more readable.
return (some_long_condition &&
        another_condition);
```

```
return (value);                // You wouldn't write var = (value);
return(result);                // return is not a function!
```

Variable and Array Initialisation

Use `{}` when possible, `()` when necessary, avoid `=`.

```
int x(3.5);
int x{3};
string name{"Some Name"};
```

```
string name("Some Name");  // could have used non-narrowing {}
int x = 3;                 // could have used non-narrowing {}
string name = "Some Name"; // could have used non-narrowing {}
```

Be careful when using a braced initialisation list `{...}` on a type with an `std::initialiser_list` constructor. A nonempty braced-init-list prefers the `std::initialiser_list` constructor whenever possible. Note that empty braces `{}` are special, and will call a default constructor if available. To force the non-`std::initialiser_list` constructor, use parentheses instead of braces.

```
std::vector<int> v(100, 1);  // A vector containing 100 items: All 1s.
std::vector<int> v{100, 1};  // A vector containing 2 items: 100 and 1.
```

Also, the brace form prevents narrowing of integral types. This can prevent some types of programming errors.

```
int pi(3.14);  // OK -- pi == 3.
int pi{3.14};  // Compile error: narrowing conversion.
```

Preprocessor Directives

The hash mark that starts a preprocessor directive should always be at the beginning of the line.

Even when preprocessor directives are within the body of indented code, the directives should start at the beginning of the line.

```
// Good - directives at beginning of line
  if (lopsided_score) {
#if DISASTER_PENDING      // Correct -- Starts at beginning of line
    DropEverything();
# if NOTIFY               // OK but not required -- Spaces after #
    NotifyClient();
```

(continues on next page)

(continued from previous page)

```
# endif
#endif
    BackToNormal();
  }
```

```
// Bad - indented directives
  if (lopsided_score) {
    #if DISASTER_PENDING  // Wrong!  The "#if" should be at beginning of line
    DropEverything();
    #endif                // Wrong!  Do not indent "#endif"
    BackToNormal();
  }
```

Class Format

Sections in `public`, `protected` and `private` order, each unindented.

The basic format for a class definition (lacking the comments, see *Class Comments* for a discussion of what comments are needed) is:

```
class MyClass : public OtherClass {
public:      // Note the 1 space indent!
  MyClass();  // Regular 2 space indent.
  explicit MyClass(int var);
  ~MyClass() {}

  void some_function();
  void some_function_that_does_nothing() {
  }

  void set_some_var(int var) { some_var_ = var; }
  int some_var() const { return some_var_; }

protected:
  bool SomeInternalFunction();

  int some_var_;
  int some_other_var_;
};
```

Things to note:

- Any base class name should be on the same line as the subclass name, subject to the 80-column limit.
- The `public:`, `protected:`, and `private:` keywords should not be indented.
- Except for the first instance, these keywords should be preceded by a blank line. This rule is optional in small classes.
- Do not leave a blank line after these keywords.
- The public section should be first, followed by the protected and finally the private section.
- See *Declaration Order* for rules on ordering declarations within each of these sections.

Constructor Initialiser Lists

Constructor initialiser lists can be all on one line or with subsequent lines indented four spaces.

The acceptable formats for initialiser lists are:

```
//  wrap before the colon and indent 2 spaces:
MyClass::MyClass(int var)
  :some_var_(var), some_other_var_(var + 1) {
  do_something();
}

// When the list spans multiple lines, put each member on its own line
// and align them:
MyClass::MyClass(int var)
  :some_var_(var),              // 4 space indent
   some_other_var_(var + 1) {  // lined up
  do_something();
}

// As with any other code block, the close curly can be on the same
// line as the open curly, if it fits.
MyClass::MyClass(int var)
  :some_var_(var) {}
```

Namespace Formatting

The contents of namespaces are indented normally.

Namespaces add an extra level of indentation. For example, use:

```
 namespace {

   void foo() {  // Correct.  Extra indentation within namespace.
     ...
}
```

} // namespace

Indent within a namespace:

```
namespace {

// Wrong!  Not indented when it should not be.
void foo() {
  ...
}

}  // namespace
```

When declaring nested namespaces, put each namespace on its own line.

```
namespace foo {
  namespace bar {
```

Horizontal Whitespace

Use of horizontal whitespace depends on location. Never put trailing whitespace at the end of a line.

General

```
void f(bool b) {  // Open braces should always have a space before them.
  ...
int i{0};  // Semicolons usually have no space before them.
// Spaces inside braces for braced-init-list are optional.  If you use them,
// put them on both sides!
int x[] = { 0 };
int x[] = {0};

// Spaces after the colon in inheritance and initialiser lists.
class Foo: public Bar {
 public:
  // For inline function implementations, put spaces between the braces
  // and the implementation itself.
  Foo(int b) : Bar(), baz_(b) {}  // No spaces inside empty braces.
  void Reset() { baz_ = 0; }  // Spaces separating braces from implementation.
  ...
```

Adding trailing whitespace can cause extra work for others editing the same file, when they merge, as can removing existing trailing whitespace. So: Don't introduce trailing whitespace. Remove it if you're already changing that line, or do it in a separate clean-up operation (preferably when no-one else is working on the file).

Loops and Conditionals

```
if (b) {          // Space after the keyword in conditions and loops.
} else {          // Spaces around else.
}
while (test) {}   // There is usually no space inside parentheses.
switch (i) {
for (int i = 0; i < 5; ++i) {
// Loops and conditions may have spaces inside parentheses, but this
// is rare.  Be consistent.
switch ( i ) {
if ( test ) {
for ( int i{0}; i < 5; ++i ) {
// For loops always have a space after the semicolon.  They may have a space
// before the semicolon, but this is rare.
for ( ; i < 5 ; ++i) {
  ...

// Range-based for loops always have a space before and after the colon.
for (auto x : counts) {
  ...
}
switch (i) {
  case 1:         // No space before colon in a switch case.
```

(continues on next page)

(continued from previous page)

```
  ...
 case 2: break;  // Use a space after a colon if there's code after it.
```

Operators

```
// Assignment operators always have spaces around them.
x = 0;

// Other binary operators usually have spaces around them, but it's
// OK to remove spaces around factors.  Parentheses should have no
// internal padding.
v = w * x + y / z;
v = w*x + y/z;
v = w * (x + z);

// No spaces separating unary operators and their arguments.
x = -5;
++x;
if (x && !y)
...
```

Templates and Casts

```
// No spaces inside the angle brackets (< and >), before
// <, or between >( in a cast
std::vector<string> x;
y = static_cast<char*>(x);

// Spaces between type and pointer are OK, but be consistent.
std::vector<char *> x;
```

Vertical Whitespace

Minimise use of vertical whitespace.

This is more a principle than a rule: don't use blank lines when you don't have to. In particular, don't put more than one or two blank lines between functions, resist starting functions with a blank line, don't end functions with a blank line, and be discriminating with your use of blank lines inside functions.

The basic principle is: The more code that fits on one screen, the easier it is to follow and understand the control flow of the program. Of course, readability can suffer from code being too dense as well as too spread out, so use your judgement. But in general, minimise use of vertical whitespace.

Some rules of thumb to help when blank lines may be useful:

- Blank lines at the beginning or end of a function very rarely help readability.
- Blank lines inside a chain of if-else blocks may well help readability.

3.5.10 Exceptions to the Rules

The coding conventions described above are mandatory. However, like all good rules, these sometimes have exceptions, which we discuss here.

Existing Non-conformant Code

You may diverge from the rules when dealing with code that does not conform to this style guide.

If you find yourself modifying code that was written to specifications other than those presented by this guide, you may have to diverge from these rules in order to stay consistent with the local conventions in that code. If you are in doubt about how to do this, ask the original author or the person currently responsible for the code. Remember that consistency includes local consistency, too.

Windows Code

Just kidding.

3.5.11 Parting Words

Use common sense and BE CONSISTENT.

If you are editing code, take a few minutes to look at the code around you and determine its style. If they use spaces around their if clauses, you should, too. If their comments have little boxes of stars around them, make your comments have little boxes of stars around them too.

The point of having style guidelines is to have a common vocabulary of coding so people can concentrate on what you are saying, rather than on how you are saying it. We present global style rules here so people know the vocabulary. But local style is also important. If code you add to a file looks drastically different from the existing code around it, the discontinuity throws readers out of their rhythm when they go to read it. Try to avoid this.

OK, enough writing about writing code; the code itself is much more interesting. Have fun!

3.6 Python Coding Style

μSpectre is a C++ project with a thin Python wrapping. Try to follow the spirit of the *C++ Coding Style and Convention*, as far as it fits into `pep8`. in case of conflict, follow `pep8`

3.7 References

Hunt (2000) A. Hunt. The pragmatic programmer : from journeyman to master. Addison-Wesley, Reading, Mass, 2000. ISBN 978-0-2016-1622-4.

Meyers (2014) Scott Meyers. Effective Modern C++, O'Reilly Media, November 2014, ISBN 978-1491903995

CHAPTER

FOUR

ORGANISATION OF THE CODE

μSpectre's code base is split in three components which might be separated into different projects in the future, and this logical separation is already apparent in the directory structure. The three components are 1. μGrid 2. μFFT 3. μSpectre proper

At the lowest level, the header-only library μGrid, contains a set of tools to define and interact with mathematical fields discretised on a regular spatial grid as used by the FFT. It is discussed in more detail in its *own section*.

On top of μGrid the library μFFT provides an uniform interface for multiple FFT implementations.

And finally μSpectre itself makes use of the two lower-level libraries and defines all the abstractions and classes to define material behaviours and to solve mechanics problems.

4.1 μGrid

As the lowest-level component of μSpectre, μGrid defines all the commonly used type aliases and data structures used througout the project. The most common aliases are described below, but it is worth having a look at the file grid_common.hh for the details.

4.1.1 Common Type Aliases

All mathematical calculations should use the types.

Warning: doxygengroup: Cannot find group "Scalars" in doxygen xml output for project "µSpectre" from directory: ./doxygenxml

While it is possible to use other types in principle, these are the ones for which all datastructures are tested and which are known to work. Also, other μSpectre developpers will expect and understand these types.

Dimensions are counted using the signed integer type *muGrid::Dim_t*. This is necessary because Eigen uses -1 to signify a dynamic number of dimensions.

The types *muGrid::Rcoord_t* and *muGrid::Ccoord_t* are used to represent real-valued coordinates and integer-valued coordinates (i.e., pixel- or cell-coordinates).

group **Coordinates**

Typedefs

using **Ccoord_t** = std::array<Dim_t, Dim>

Ccoord_t are cell coordinates, i.e. integer coordinates.

using **Rcoord_t** = std::array<Real, Dim>

Real space coordinates.

using **DynCcoord_t** = DynCcoord<threeD>

usually, we should not need omre than three dimensions

using **DynRcoord_t** = DynCcoord<threeD, Real>

usually, we should not need omre than three dimensions

Functions

template<typename **T**, size_t **Dim**>
Eigen::Map<*Eigen*::Matrix<*T*, *Dim*, 1>> **eigen**(std::array<*T*, *Dim*> &coord)

return a Eigen representation of the data stored in a std::array (e.g., for doing vector operations on a coordinate)

template<typename **T**, size_t **Dim**>
Eigen::Map<const *Eigen*::Matrix<*T*, *Dim*, 1>> **eigen**(const std::array<*T*, *Dim*> &coord)

return a constant Eigen representation of the data stored in a std::array (e.g., for doing vector operations on a coordinate)

template<typename **T**, size_t **MaxDim**>
Eigen::Map<*Eigen*::Matrix<*T*, *Eigen*::Dynamic, 1>> **eigen**(DynCcoord<*MaxDim*, *T*> &coord)

return a Eigen representation of the data stored in a std::array (e.g., for doing vector operations on a coordinate)

template<typename **T**, size_t **MaxDim**>
Eigen::Map<const *Eigen*::Matrix<*T*, *Eigen*::Dynamic, 1>> **eigen**(const DynCcoord<*MaxDim*, *T*> &coord)

return a const Eigen representation of the data stored in a std::array (e.g., for doing vector operations on a coordinate)

template<size_t **MaxDim**, typename **T** = *Dim_t*>

class **DynCcoord**

#include <grid_common.hh> Class to represent integer (cell-) coordinates or real-valued coordinates. This class can dynamically accept any spatial-dimension between 1 and MaxDim, and *DynCcoord* references can be cast to `muGrid::Ccoord_t &` or `muGrid::Rcoord_t &` references. These are used when templating with the spatial dimension of the problem is undesireable/impossible.

Public Types

using **iterator** = typename std::array<*T*, *MaxDim*>::iterator

iterator type

using **const_iterator** = typename std::array<*T*, *MaxDim*>::const_iterator

constant iterator type

Public Functions

inline **DynCcoord**()

default constructor

inline **DynCcoord**(std::initializer_list<*T*> init_list)

constructor from an initialiser list for compound initialisation.

Parameters

init_list – The length of the initialiser list becomes the spatial dimension of the coordinate, therefore the list must have a length between 1 and MaxDim

inline explicit **DynCcoord**(*Dim_t* dim)

Constructor only setting the dimension. WARNING: This constructor *needs* regular (round) braces '()', using curly braces '{}' results in the initialiser list constructor being called and creating a *DynCcoord* with spatial dimension 1

Parameters

dim – spatial dimension. Needs to be between 1 and MaxDim

template<size_t **Dim**>
inline explicit **DynCcoord**(const std::array<*T*, *Dim*> &ccoord)

Constructor from a statically sized coord.

DynCcoord(const *DynCcoord* &other) = default

Copy constructor.

DynCcoord(*DynCcoord* &&other) = default

Move constructor.

~DynCcoord() = default

nonvirtual Destructor

template<size_t **Dim**>
inline *DynCcoord* &**operator=**(const std::array<*T*, *Dim*> &ccoord)

Assign arrays.

DynCcoord &**operator=**(const *DynCcoord* &other) = default

Copy assignment operator.

DynCcoord &**operator=**(*DynCcoord* &&other) = default

Move assignment operator.

template<size_t **Dim2**>
inline bool **operator==**(const std::array<*T*, *Dim2*> &other) const

comparison operator

inline bool **operator==**(const *DynCcoord* &other) const

comparison operator

template<typename **T2**>
inline *DynCcoord*<*MaxDim*, decltype(*T*{} / *T2*{})> **operator/**(const *DynCcoord*<*MaxDim*, *T2*> &other) const

element-wise division

inline *T* &**operator[]**(const size_t &index)

access operator

inline const *T* &**operator[]**(const size_t &index) const

access operator

template<size_t **Dim**>
inline **operator std::array<*T*, *Dim*>**() const

conversion operator

template<*Dim_t* **Dim**>
inline std::array<*T*, *Dim*> &**get**()

cast to a reference to a statically sized array

template<*Dim_t* **Dim**>
inline const std::array<*T*, *Dim*> &**get**() const

cast to a const reference to a statically sized array

inline const *Dim_t* &**get_dim**() const

return the spatial dimension of this coordinate

inline *iterator* **begin**()

iterator to the first entry for iterating over only the valid entries

inline *iterator* **end**()

iterator past the dim-th entry for iterating over only the valid entries

inline *const_iterator* **begin**() const

const iterator to the first entry for iterating over only the valid entries

inline *const_iterator* **end**() const

const iterator past the dim-th entry for iterating over only the valid entries

inline *T* ***data**()

return the underlying data pointer

inline const *T* ***data**() const

return the underlying data pointer

inline *T* &**back**()

return a reference to the last valid entry

inline const *T* &**back**() const

return a const reference to the last valid entry

These types are also used to define nb_grid_pts or spatial lengths for computational domains.

4.1.2 Field Data Types

The most important part of μGrid to understand is how it handles field data and access to it. By field we mean the discretisation of a mathematical field on the grid points, i.e., numerical data associated with all pixels/voxels of an FFT grid or a subset thereof. The numerical data can be **scalar**, **vectorial**, **matricial**, **tensorial** or a generic **array** of **integer**, **real** or **complex** values per pixel.

Fields that are defined on every pixel/voxel of a grid are called **global fields** while fields defined on a subset of pixels/voxels **local fields**. As an example, the strain field is a global field for any calculation (it exists in the entire domain), while for instance the field of an internal (state) variable of a material in a composite is only defined for the pixels that belong to that material.

There are several ways in which we interact with fields, and the same field might be interacted with in different ways by different parts of a problem. Let's take the (global) strain field in a three-dimensional finite strain problem with $255 \times 255 \times 255$ voxels as an example: the solver treats it as a long vector (of length $3^2 \cdot 255^3$), the FFT sees it as a four-dimensional array of shape $255 \times 255 \times 255 \times 3^2$, and from the constitutive laws' perspective, it is just a sequence of second-rank tensors (i.e., shape $255^3 \times 3 \times 3$).

Basic μGrid Field Concepts

In order to reconcile these different interpretations without copying data around, μGrid splits the concept of a field into three components:

- storage

  This refers managing the actual memory in which field data is held. For this, the storage abstraction needs to know the scalar type of data (`Int`, `Real`, `Complex`, e.t.c.), the number of pixels/voxels for which the field is defined, and the number of scalar components per pixel/voxel (e.g., 9 for a second-rank asymmetric tensor in a three-dimensional problem).

  μGrid's abstraction for field data storage is the **field** represented by a child class of `FieldBase<FieldCollection_t>`, see fields.

- representation

  Meaning how to interpret the data at a given pixel/voxel (i.e., is it a vector, a matrix, ...). This will also determine which mathematical operations can be performed on per-pixel/voxel data. The representation allows also to iterate over a field pixel/voxel by pixel/voxel.

  μGrid's abstraction for field representations is the **field map** represented by a child class of `FieldMap<FieldCollection_t, Scalar_t, NbComponents[, IsConst]>`, see field_map.

- per-pixel/voxel access/iteration

  Given a pixel/voxel coordinate or index, the position of the associated pixel/voxel data is a function of the type of field (global or local). Since the determination procedure is identical for every field defined on the same domain, this ability (and the associated overhead) can be centralised into a manager of field collections.

  μGrid's abstraction for field access and management is the **field collection** represented by the two classes *`LocalFieldCollection<Dim>`* and *`GlobalFieldCollection<Dim>`*, see field_collection.

Fields

Fields are where the data is stored, so they are mainly distinguished by the scalar type they store (`Int`, `Real` or `Complex`), and the number of components (statically fixed size, or dynamic size).

The most commonly used fields are the statically sized ones, `TensorField`, `MatrixField`, and the `ScalarField` (which is really just a 1×1 matrix field).

Less commonly, we use the dynamically sized *`TypedField`*, but more on this later.

Fields have a protected constructor, which means that you cannot directly build a field object, instead you have to go through the factory function `make_field<Field_t>(name, collection)` (or `make_statefield<Field_t>(name, collection)` if you're building a statefield, see state_field) to create them and you only receive a reference to the built field. The field itself is stored in a `std::unique_ptr` which is registered in and managed by a field collection. This mechanism is meant to ensure that fields are not copied around or free'd so that field maps always remain valid and unambiguously linked to a field.

Fields give access to their bulk memory in form of an `Eigen:Map`. This is useful for instance for accessing the global strain, stress, and tangent moduli fields in the solver.

Example: Using fields as global arrays:

The following is a code example from the standard Cell:

```
template <Dim_t DimS, Dim_t DimM>
auto CellBase<DimS, DimM>::get_strain_vector() -> Vector_ref {
  return this->get_strain().eigenvec();
}
```

The return value of **:cpp:function:`Cell::get_strain_vector()<muSpectre::CellBase::get_strain_vector>`** is an `Eigen::Map` onto a matrix of shape 1×N.

If you wish to handle field data on a per-pixel/voxel basis, the mechanism for that is the field map and is described in field_map.

Field Maps

Field maps are light-weight resource handles (meaning they can be created and destroyed cheaply) that are iterable and provide direct per-pixel/voxel access to the data stored in the mapped field.

The choice of field map defines the type of reference you obtain when dereferencing an iterator or using the direct random acccess operator `[]`.

Typically used field maps include:

- *`ScalarFieldMap`*,
- *`ArrayFieldMap`*,
- *`MatrixFieldMap`*, and the
- `T4MatrixFieldMap`.

All of these are fixed size (meaning their size is set at compile time) and therefore support fast linear algebra on the iterates. There is also a dynamically sized field map type, the `TypedFieldMap` which is useful for debugging and python bindings. It supports all the features of the fixed-size maps, but linear algebra on the iterates will be slow because it cannot be effectively vectorised.

Example 1: Iterating over fields and do math on the iterates:

The following is a code example from the tests of the finite-strain projection operator defined by T.W.J. de Geus, J. Vondřejc, J. Zeman, R.H.J. Peerlings, M.G.D. Geers.

```
for (auto && tup :
     akantu::zip(fields.get_pixels().template get_dimensioned_pixels<dim>(),
                 grad, var)) {
  auto & ccoord = std::get<0>(tup);  // iterate from fields
  auto & g = std::get<1>(tup);       // iterate from grad
  auto & v = std::get<2>(tup);       // iterate from var

  // use iterate in arbitrary expressions
  Vector vec = muGrid::CcoordOps::get_vector(
      ccoord, (fix::projector.get_domain_lengths() /
               fix::projector.get_nb_domain_grid_pts())
                  .template get<dim>());
  // do efficient linear algebra on iterates
  g.row(0) = k.transpose() *
             cos(k.dot(vec));  // This is a plane wave with wave vector k in
                               // real space. A valid gradient field.
  v.row(0) = g.row(0);
}
```

Field Collections

Field collections come in two flavours; *LocalFieldCollection<Dim>* and *GlobalFieldCollection<Dim>* and are templated by the spatial dimension of the problem. They adhere to the interface defined by their common base class, `FieldCollectionBase`. Both types are iterable (the iterates are the coordinates of the pixels/voxels for which the fields of the collection are defiened.

Global field collections need to be given the problem nb_grid_pts (i.e. the size of the grid) at initialisation, while local collections need to be filled with pixels/voxels through repeated calls to *add_pixel(pixel)*. At initialisation, they derive their size from the number of pixels that have been added.

Fields (State Fields) are identified by their unique name (prefix) and can be retrieved in multiple ways:

Warning: doxygenfunction: Cannot find function “muGrid::FieldCollectionBase::operator[]” in doxygen xml output for project “µSpectre” from directory: ./doxygenxml

Warning: doxygenfunction: Cannot find function “muGrid::FieldCollectionBase::at” in doxygen xml output for project “µSpectre” from directory: ./doxygenxml

Warning: doxygenfunction: Cannot find function “muGrid::FieldCollectionBase::get_typed_field” in doxygen xml output for project “µSpectre” from directory: ./doxygenxml

Warning: doxygenfunction: Cannot find function "muGrid::FieldCollectionBase::get_statefield" in doxygen xml output for project "µSpectre" from directory: ./doxygenxml

Warning: doxygenfunction: Cannot find function "muGrid::FieldCollectionBase::get_typed_statefield" in doxygen xml output for project "µSpectre" from directory: ./doxygenxml

- per-pixel/voxel access/iteration

  Given a pixel/voxel coordinate or index, the position of the associated pixel/voxel data is a function of the type of field (global or local). Since the determination procedure is identical for every field defined on the same domain, this ability (and the associated overhead) can be centralised into a manager of field collections.

  μGrid's abstraction for field access and management is the **field collection** represented by the two classes *LocalFieldCollection<Dim>* and *LocalFieldCollection<Dim>*, see field_collection.

Fields

Fields are where the data is stored, so they are mainly distinguished by the scalar type they store (`Int`, `Real` or `Complex`), and the number of components (statically fixed size, or dynamic size).

The most commonly used fields are the statically sized ones, `TensorField`, `MatrixField`, and the `ScalarField` (which is really just a 1×1 matrix field).

Less commonly, we use the dynamically sized *TypedField*, but more on this later.

Fields have a protected constructor, which means that you cannot directly build a field object, instead you have to go through the factory function `make_field<Field_t>(name, collection)` (or `make_statefield<Field_t>(name, collection)` if you're building a statefield, see state_field) to create them and you only receive a reference to the built field. The field itself is stored in a `std::unique_ptr` which is registered in and managed by a field collection. This mechanism is meant to ensure that fields are not copied around or free'd so that field maps always remain valid and unambiguously linked to a field.

Fields give access to their bulk memory in form of an `Eigen:Map`. This is useful for instance for accessing the global strain, stress, and tangent moduli fields in the solver.

If you wish to handle field data on a per-pixel/voxel basis, the mechanism for that is the field map and is described in field_map.

Example: Using fields as global arrays:

The following is a code example from the standard Cell:

```
template <Dim_t DimS, Dim_t DimM>
auto CellBase<DimS, DimM>::get_strain_vector() -> Vector_ref {
  return this->get_strain().eigenvec();
}
```

The return value of **:cpp:function:`Cell::get_strain_vector()<muSpectre::CellBase::get_strain_vector>`** is an `Eigen::Map` onto a matrix of shape 1×N.

If you wish to handle field data on a per-pixel/voxel basis, the mechanism for that is the field map and is described in field_map.

Field Maps

Field maps are light-weight resource handles (meaning they can be created and destroyed cheaply) that are iterable and provide direct per-pixel/voxel access to the data stored in the mapped field.

The choice of field map defines the type of reference you obtain when dereferencing an iterator or using the direct random acccess operator `[]`.

Typically used field maps include:

- *ScalarFieldMap*,
- *ArrayFieldMap*,
- *MatrixFieldMap*, and the
- `T4MatrixFieldMap`.

All of these are fixed size (meaning their size is set at compile time) and therefore support fast linear algebra on the iterates. There is also a dynamically sized field map type, the `TypedFieldMap` which is useful for debugging and python bindings. It supports all the features of the fixed-size maps, but linear algebra on the iterates will be slow because it cannot be effectively vectorised.

Example 1: Iterating over fields and do math on the iterates:

The following is a code example from the tests of the finite-strain projection operator defined by T.W.J. de Geus, J. Vondřejc, J. Zeman, R.H.J. Peerlings, M.G.D. Geers.

```
for (auto && tup :
     akantu::zip(fields.get_pixels().template get_dimensioned_pixels<dim>(),
                 grad, var)) {
  auto & ccoord = std::get<0>(tup);  // iterate from fields
  auto & g = std::get<1>(tup);       // iterate from grad
  auto & v = std::get<2>(tup);       // iterate from var

  // use iterate in arbitrary expressions
  Vector vec = muGrid::CcoordOps::get_vector(
      ccoord, (fix::projector.get_domain_lengths() /
               fix::projector.get_nb_domain_grid_pts())
                  .template get<dim>());
  // do efficient linear algebra on iterates
  g.row(0) = k.transpose() *
             cos(k.dot(vec));  // This is a plane wave with wave vector k in
                               // real space. A valid gradient field.
  v.row(0) = g.row(0);
}
```

Field Collections

Field collections come in two flavours; *LocalFieldCollection<Dim>* and *GlobalFieldCollection<Dim>* and are templated by the spatial dimension of the problem. They adhere to the interface defined by their common base class, `FieldCollectionBase`. Both types are iterable (the iterates are the coordinates of the pixels/voxels for which the fields of the collection are defiened.

Global field collections need to be given the problem nb_grid_pts (i.e. the size of the grid) at initialisation, while local collections need to be filled with pixels/voxels through repeated calls to *add_pixel(pixel)*. At initialisation, they derive their size from the number of pixels that have been added.

Fields (State Fields) are identified by their unique name (prefix) and can be retrieved in multiple ways:

Warning: doxygenfunction: Cannot find function "muGrid::FieldCollectionBase::operator[]" in doxygen xml output for project "µSpectre" from directory: ./doxygenxml

Warning: doxygenfunction: Cannot find function "muGrid::FieldCollectionBase::at" in doxygen xml output for project "µSpectre" from directory: ./doxygenxml

Warning: doxygenfunction: Cannot find function "muGrid::FieldCollectionBase::get_typed_field" in doxygen xml output for project "µSpectre" from directory: ./doxygenxml

Warning: doxygenfunction: Cannot find function "muGrid::FieldCollectionBase::get_statefield" in doxygen xml output for project "µSpectre" from directory: ./doxygenxml

Warning: doxygenfunction: Cannot find function "muGrid::FieldCollectionBase::get_typed_statefield" in doxygen xml output for project "µSpectre" from directory: ./doxygenxml

State or History Variables

Some fields hold state or history variables, i.e., such fields have a current value and one or more old values. This is particularly common for internal variables of inelastic materials (e.g., damage variables, plastic flow, e.t.c.). The straight-forward way of handling this situation is to define a current field, and one or more fields of the same type to hold old values. This approach has the disadvantages that it leads to a multitude of variables to keep track of, and that the values need to by cycled between the fields using a copy; this approach is both inefficient and error-prone.

µGrid addresses this situation with the state field abstraction. A state
field is an encapsulated container of fields in a single variable. It allows to access the current field values globally, and gives read-only access to old field values globally. Iterative per-pixel access is handled through state field maps which, similarly to the field map, allow to iterate though all pixels/voxels on which the field is defined, and the iterates give access to the current value at the pixel/voxel or read-only access to the old values.

Mapped Fields

Some fields are only ever going to be used by one entity (e.g., internal variables of a material). For these fields, the flexibility of the field/field collection/field map paradigm can be a burden. Mapped fields are an encapsulation of a field and a corresponding map into a single object, drastically reducing boilerplate code.

CHAPTER

FIVE

CONSTITUTIVE LAWS

5.1 Generic Linear Elastic Material

The generic linear elastic material is implemented in the classes *MaterialLinearElasticGeneric1* and *MaterialLinearElasticGeneric2*, and is defined solely by the elastic stiffness tensor $\mathbb{C}$, which has to be specified in Voigt notation. In the case of *MaterialLinearElasticGeneric2*, additionally, a per-pixel eigenstrain $\bar{\varepsilon}$ can be supplied. The constitutive relation between the Cauchy stress $\boldsymbol{\sigma}$ and the small strain tensor $\boldsymbol{\varepsilon}$ is given by

$$\boldsymbol{\sigma} = \mathbb{C} : \boldsymbol{\varepsilon} \tag{5.1}$$

$$\sigma_{ij} = C_{ijkl}\,\varepsilon_{kl}, \quad \text{for the simple version, and} \tag{5.2}$$

$$\boldsymbol{\sigma} = \mathbb{C} : (\boldsymbol{\varepsilon} - \bar{\boldsymbol{\varepsilon}}) \tag{5.3}$$

$$\sigma_{ij} = C_{ijkl}\,(\varepsilon_{kl} - \bar{\varepsilon}_{kl}) \quad \text{for the version with eigenstrain} \tag{5.4}$$

This implementation is convenient, as it covers all possible linear elastic behaviours, but it is by far not as efficient as *MaterialLinearElastic1* for isotropic linear elasticity.

This law can be used in both small strain and finite strain calculations.

The following snippet shows how to use this law in python to implement isotropic linear elasticity:

5.1.1 Python Usage Example

```
C = np.array([[2 * mu + lam,          lam,          lam,  0,  0,  0],
              [         lam, 2 * mu + lam,          lam,  0,  0,  0],
              [         lam,          lam, 2 * mu + lam,  0,  0,  0],
              [           0,            0,            0, mu,  0,  0],
              [           0,            0,            0,  0, mu,  0],
              [           0,            0,            0,  0,  0, mu]])

eigenstrain = np.array([[  0, .01],
                        [.01,   0]])

mat1 = muSpectre.material.MaterialLinearElasticGeneric1_3d.make(
    cell, "material", C)
mat1.add_pixel(pixel)
mat2 = muSpectre.material.MaterialLinearElasticGeneric2_3d.make(
    cell, "material", C)
mat2.add_pixel(pixel, eigenstrain)
```

5.2 CellSplit

The implementation of split cell class is *CellSplit*. In order to compile this class one should set the cmake variable SPLIT_CELL **ON** in the configuration.

Simulating multi-phase structures with muSpectre involves pixels which share material as they may lie in the interface of phases. Different homogenisation schemes can be used for substituting pixels if the effective media consists of two or more phases. One of these approximations assuming iso-strain pixels is the Voigt method. In the *CellSplit* pixels' effective constitutive behavior is approximated as the weighted average of the constituent materials w.r.t their volume fractions (α)

$$\boldsymbol{P}^l = f\left(\boldsymbol{F}^l\right) \qquad , \boldsymbol{P}^r = f\left(\boldsymbol{F}^r\right) \tag{1}$$

$$\boldsymbol{F} = \boldsymbol{F}^r \qquad , \boldsymbol{F} = \boldsymbol{F}^l \tag{2}$$

$$\overline{\boldsymbol{P}} = \langle \boldsymbol{P} \rangle \tag{3}$$

$$\tag{5.5}$$

where

$$\langle \boldsymbol{P} \rangle = \alpha^l \boldsymbol{P}^l + \alpha^r \boldsymbol{P}^r. \tag{4}$$

The superscripts (l) and (r) show the two constituent materials of the pixel and $(\boldsymbol{P})$, $(\boldsymbol{F})$ are, respectively, first Piola-Kirchhoff and deforamtion gradient tensors. (α) is the volume fraction of the phases.The *CellSplit* inherits from CellBase and can be used in its stead. Currently, all materials inheriting from *MaterialMuSpectre* can be added to an instance of *CellSplit*. However, it should be noted that for adding pixel to the materials contained in this type of cell, *add_pixel_split()* sould be employed instead of plain add_pixel_split(). This function takes the ratio of the materials in the pixel that is being assigned to it as an input parameter. It is notable that the summation of ratio of materials should add up to unity for all the pixels in the cell.

Specialised function *make_automatic_precipitate_split_pixels()* exists in *CellSplit* which enables user to assign materials based on the material and geometry of precipitates (as a set of coordinates composing a polyheron/polygon in 3D/2D). Moreover, one can use the function *complete_material_assignment()* in order to assign the pixels whose assignments are not completed to a specific material. The following snippet shows how one can use the machinery to employ this specific kind of Cell in µSpectre.

5.2.1 Python Usage Example

```
rve = msp.Cell(res, lengths,
               formulation, None, 'fftw', None,
               msp.SplitCell.split)

mat1 = msp.material.MaterialLinearElastic1_2d.make(
   rve, "mat1", E1, .noo1)

mat2 = msp.material.MaterialLinearElastic1_2d.make(
   rve, "mat2",  E2, .noo2)

points = np.ndarray(shape=(num, 2))
for j, tetha in enumerate(np.linspace(0, 2*np.pi, num, endpoint=false)):
    points[j, 0] = center[0] + radius*np.cos(tetha)
    points[j, 1] = center[1] + radius*np.sin(tetha)

points_list = [points.tolist()]
```

(continues on next page)

(continued from previous page)

```
rve.make_precipitate(mat1, points_list)
rve.complete_material_assignemnt_simple(mat2)
```

5.3 Laminate Material

The generic laminate material is implemented in *MaterialLaminate* inheriting directly from *MaterialBase*, and it contains two arbitrary underlting materials(with either linear or nonlinear constitutive laws). In order to compile this class one should set the cmake variable SPLIT_CELL **ON** in the configuration.

The underlying materials behave as they consist a laminate. The resultant constitutive behavior depends on both of their constitutive laws, the volume fraction of phases ((α^l), (α^r)), and the normal vector of their interface. The formulation governing the stress and the deformation gradient of the underlying phases is given by:

$$\boldsymbol{F} = \alpha^l \boldsymbol{F}^l + \alpha^r \boldsymbol{F}^r \tag{1}$$

$$\boldsymbol{P} = \alpha^l \boldsymbol{P}^l + \alpha^r \boldsymbol{P}^r \tag{2}$$

$$\boldsymbol{P}^l \cdot \boldsymbol{n} = \boldsymbol{P}^r \cdot \boldsymbol{n} \tag{3}$$

$$\boldsymbol{F}^l \cdot \left(\mathbb{I} - \boldsymbol{n} \otimes \boldsymbol{n}\right) = \boldsymbol{F}^r \cdot \left(\mathbb{I} - \boldsymbol{n} \otimes \boldsymbol{n}\right) \tag{4}$$

where, The superscripts (l) and (r) show the two constituent materials of the pixel and $(\boldsymbol{P})$, $(\boldsymbol{F})$ are, respectively, first Piola-Kirchhoff and deforamtion gradient tensors. (α) is the normal vector of phases' interface and $(\mathbb{I})$ is the fourth order identity matrix. Equations (3) and (4) are the equilibrium and the compatibility equations on the pahses' interface in the laminate structure. By having deformations as the input (µSpectre) and from equation (4) some components of deforamtion of both phases are easily derived. For calculating the remaining components, it is necessary to solve equation (3), which in the most general case is a nonlinear equation depending on both materials' constiturive laws. Accordingly this material's *evaluate_stress()* calls an internal solver implemented in *laminate_solver()* where equation (3) is solved, per-pixel, employing both underlying materials' constitutive laws. Accordingly, this material is not expected to be as efficient as materials inheriting from *MaterialMuSpectre*.

MaterialLaminate at creation only needs a name. However, its *add_pixel()* takes a pixel and pointers two the underlying materials for each pixel as well as volume fraction and interface normal vector for each pixel. For convinience, function `make_pixels_precipitate_for_laminate_material()` has been added to `CellBase` using which user can add pixels to a *MaterialLaminate* object by introducing the shape of a precipitate, it's material an the base material of the matrix media in which the precipitate lies. In addition, `complete_material_assignment_simple()` enables to assign the remaining of the pixles (unassigned) pixels to a material(the base material of the matrix media).The following snippet shows how one can use the machinery to employ this specific Material in µSpectre.

5.3.1 Python Usage Example

```
rve = msp.Cell(res,
               lengths,
               formulation)

mat1_laminate = msp.material.MaterialLinearElastic1_2d.make_free(
   "mat1_free", E1, noo)

mat2_laminate = msp.material.MaterialLinearElastic1_2d.make_free(
   "mat2_free", E2, noo)
```

(continues on next page)

(continued from previous page)

```
mat1 = msp.material.MaterialLinearElastic1_2d.make(
    rve, "mat1", E1, noo)

mat2 = msp.material.MaterialLinearElastic1_2d.make(
    rve, "mat2",  E2, noo)

mat_lam = msp.material.MaterialLaminate_2d.make(rve, "laminate")

points = np.ndarray(shape=(num, 2))
for j, tetha in enumerate(np.linspace(0, 2*np.pi, num, endpoint=false)):
    points[j, 0] = center[0] + radius*np.cos(tetha)
    points[j, 1] = center[1] + radius*np.sin(tetha)

points_list = [points.tolist()]

rve.make_precipitate_laminate(mat_lam, mat1,
                              mat1_laminate,
                              mat2_laminate,
                              points_list)

rve.complete_material_assignemnt_simple(mat2)
```

CHAPTER

SIX

TESTING CONSTITUTIVE LAWS

When writing new constitutive laws, the ability to evaluate the stress-strain behaviour and the tangent moduli is convenient, but μSpectre's material model makes it cumbersome to isolate and execute the `evaluate_stress()` and `evaluate_stress_tangent()` methods than any daughter class of *MaterialMuSpectre* must implement (e.g., *evaluate_stress()*). As a helper object, μSpectre offers the class *MaterialEvaluator* to facilitate precisely this:

A *MaterialEvaluator* object can be constructed with a shared pointer to a *MaterialBase* and exposes functions to evaluate just the stress, both the stress and tangent moduli, or a numerical approximation to the tangent moduli. For materials with internal history variables, *MaterialEvaluator* also exposes the *MaterialBase::save_history_variables()* method. As a convenience function, all daughter classes of *MaterialMuSpectre* have the static factory function *make_evaluator()* to create a material and its evaluator at once. See the *Reference* for the full class description.

6.1 Python Usage Example

```
import numpy as np
from muSpectre import material
from muSpectre import Formulation

# MaterialLinearElastic1 is standard linear elasticity while
# MaterialLinearElastic2 has a per pixel eigenstrain which needs to be set
LinMat1, LinMat2 = (material.MaterialLinearElastic1_2d,
                    material.MaterialLinearElastic2_2d)

young, poisson = 210e9, .33

# the factory returns a material and it's corresponding evaluator
material1, evaluator1 = LinMat1.make_evaluator(young, poisson)

# the material is empty (i.e., does not have any pixel/voxel), so a pixel
# needs to be added. The coordinates are irrelevant, there just needs to
# be one pixel.
material1.add_pixel([0,0])

# the stress and tangent can be evaluated for finite strain
F = np.array([[1., .01],[0, 1.0]])
P, K = evaluator1.evaluate_stress_tangent(F, Formulation.finite_strain)
# or small strain
eps = .5 * ((F-np.eye(2)) + (F-np.eye(2)).T)
sigma, C = evaluator1.evaluate_stress_tangent(eps, Formulation.small_strain)
```

(continues on next page)

(continued from previous page)

```
# and the tangent can be checked against a numerical approximation
Delta_x = 1e-6
num_C = evaluator1.estimate_tangent(eps, Formulation.small_strain, Delta_x)


# Materials with per-pixel data behave similarly: the factory returns a
# material and it's corresponding evaluator like before
material2, evaluator2 = LinMat2.make_evaluator(young, poisson)

# when adding the pixel, we now need to specify also the per-pixel data:
eigenstrain = np.array([[.01, .002], [.002, 0.]])
material2.add_pixel([0,0], eigenstrain)
```

6.2 C++ Usage Example

```
#include"materials/material_linear_elastic2.hh"
#include "materials/material_evaluator.hh"
#include <libmugrid/T4_map_proxy.hh>

#include "Eigen/Dense"

using Mat_t = MaterialLinearElastic2<twoD, twoD>;

constexpr Real Young{210e9};
constexpr Real Poisson{.33};

auto mat_eval{Mat_t::make_evaluator(Young, Poisson)};
auto & mat{*std::get<0>(mat_eval)};
auto & evaluator{std::get<1>(mat_eval)};


using T2_t = Eigen::Matrix<Real, twoD, twoD>;
using T4_t = T4Mat<Real, twoD>;
const T2_t F{(T2_t::Random() - (T2_t::Ones() * .5)) * 1e-4 +
             T2_t::Identity()};

T2_t eigen_strain{[](auto x) {
  return 1e-4 * (x + x.transpose());
}(T2_t::Random() - T2_t::Ones() * .5)};

mat.add_pixel({}, eigen_strain);

T2_t P{};
T4_t K{};

std::tie(P, K) =
  evaluator.evaluate_stress_tangent(F, Formulation::finite_strain);
```

CHAPTER

SEVEN

REFERENCE

template<class **T**>

class **ArangeContainer**

#include <iterators.hh> helper class to generate range iterators

Public Types

using **iterator** = *iterators*::*ArangeIterator*<*T*>

undocumented

Public Functions

inline constexpr **ArangeContainer**(*T* start, *T* stop, *T* step = 1)

undocumented

inline explicit constexpr **ArangeContainer**(*T* stop)

undocumented

inline constexpr *T* **operator[]**(size_t i)

undocumented

inline constexpr *T* **size**()

undocumented

inline constexpr *iterator* **begin**()

undocumented

inline constexpr *iterator* **end**()

undocumented

Private Members

const *T* **start** = {0}

const *T* **stop** = {0}

const *T* **step** = {1}

template<class **T**>

class **ArangeIterator**

#include <iterators.hh> emulates python's range iterator

Public Types

using **value_type** = *T*

undocumented

using **pointer** = *T**

undocumented

using **reference** = *T*&

undocumented

using **iterator_category** = std::input_iterator_tag

undocumented

Public Functions

inline constexpr **ArangeIterator**(*T* value, *T* step)

undocumented

constexpr **ArangeIterator**(const *ArangeIterator*&) = default

undocumented

inline constexpr *ArangeIterator* &**operator++**()

undocumented

inline constexpr const *T* &**operator***() const

undocumented

inline constexpr bool **operator==**(const *ArangeIterator* &other) const

undocumented

inline constexpr bool **operator!=**(const *ArangeIterator* &other) const

undocumented

Private Members

T **value** = {0}

const *T* **step** = {1}

template<*Dim_t* **order**, typename **Fun_t**, *Dim_t* **dim**, *Dim_t*... **args**>

struct **CallSizesHelper**

#include <eigen_tools.hh> Call a passed lambda with the unpacked sizes as arguments.

Public Static Functions

static inline decltype(auto) **call**(*Fun_t* &&fun)

applies the call

template<typename **Fun_t**, *Dim_t* **dim**, *Dim_t*... **args**>

struct **CallSizesHelper**<0, *Fun_t*, *dim*, *args*...>

#include <eigen_tools.hh> Call a passed lambda with the unpacked sizes as arguments.

Public Static Functions

static inline decltype(auto) **call**(*Fun_t* &&fun)

applies the call

class **Cell**

#include <cell.hh> Base class for the representation of a homogenisatonion problem in µSpectre. The *muSpectre::Cell* holds the global strain, stress and (optionally) tangent moduli fields of the problem, maintains the list of materials present, as well as the projection operator.

Subclassed by *muSpectre::CellSplit*

Public Types

using **Material_ptr** = std::unique_ptr<*MaterialBase*>

materials handled through `std::unique_ptr`s

using **Material_sptr** = std::shared_ptr<*MaterialBase*>

using **Projection_ptr** = std::unique_ptr<*ProjectionBase*>

projections handled through `std::unique_ptr`s

using **Matrix_t** = *Eigen*::Matrix<Real, *Eigen*::Dynamic, *Eigen*::Dynamic>

short-hand for matrices

using **Eigen_cmap** = *muGrid*::*RealField*::Eigen_cmap

ref to constant vector

using **Eigen_map** = *muGrid*::*RealField*::Eigen_map

ref to vector

using **EigenVec_t** = *Eigen*::Ref<*Eigen*::Matrix<Real, *Eigen*::Dynamic, 1>>

Ref to input/output vector.

using **EigenCVec_t** = *Eigen*::Ref<const *Eigen*::Matrix<Real, *Eigen*::Dynamic, 1>>

Ref to input vector.

using **Adaptor** = *CellAdaptor*<*Cell*>

adaptor to represent the cell as an Eigen sparse matrix

Public Functions

Cell() = delete

Deleted default constructor.

explicit **Cell**(*Projection_ptr* projection, *SplitCell* is_cell_split = *SplitCell*::*no*)

Constructor from a projection operator.

Cell(const *Cell* &other) = delete

Copy constructor.

Cell(*Cell* &&other) = default

Move constructor.

virtual **~Cell**() = default

Destructor.

Cell &**operator=**(const *Cell* &other) = delete

Copy assignment operator.

Cell &**operator=**(*Cell* &&other) = delete

Move assignment operator.

bool **is_initialised**() const

for handling double initialisations right

Dim_t **get_nb_dof**() const

returns the number of degrees of freedom in the cell

size_t **get_nb_pixels**() const

number of pixels on this processor

const *muFFT*::*Communicator* &**get_communicator**() const

return the communicator object

const *Formulation* &**get_formulation**() const

formulation is hard set by the choice of the projection class

Dim_t **get_material_dim**() const

returns the material dimension of the problem

void **set_uniform_strain**(const *Eigen*::Ref<const *Matrix_t*>&)

set uniform strain (typically used to initialise problems

virtual *MaterialBase* &**add_material**(*Material_ptr* mat)

add a new material to the cell

void **complete_material_assignment_simple**(*MaterialBase* &material)

By taking a material as input this function assigns all the untouched(not-assigned) pixels to that material

void **make_pixels_precipitate_for_laminate_material**(const std::vector<*DynRcoord_t*> &precipitate_vertices, *MaterialBase* &mat_laminate, *MaterialBase* &mat_precipitate_cell, *Material_sptr* mat_precipitate, *Material_sptr* mat_matrix)

Given the vertices of polygonal/Polyhedral precipitate, this function assign pixels 1. inside precipitate->mat_precipitate_cell, material at the interface of precipitae-> to mat_precipitate & mat_matrix according to the intersection of pixels with the precipitate

template<Dim_t **Dim**>
void **make_pixels_precipitate_for_laminate_material_helper**(const std::vector<*DynRcoord_t*> &precipitate_vertices, *MaterialBase* &mat_laminate, *MaterialBase* &mat_precipitate_cell, *Material_sptr* mat_precipitate, *Material_sptr* mat_matrix)

Adaptor **get_adaptor**()

get a sparse matrix view on the cell

void **save_history_variables**()

freezes all the history variables of the materials

std::array<Dim_t, 2> **get_strain_shape**() const

returns the number of rows and cols for the strain matrix type (for full storage, the strain is stored in material_dim × material_dim matrices, but in symmetric storage, it is a column vector)

Dim_t **get_strain_size**() const

returns the number of components for the strain matrix type (for full storage, the strain is stored in material_dim × material_dim matrices, but in symmetric storage, it is a column vector)

const Dim_t &**get_spatial_dim**() const

return the spatial dimension of the discretisation grid

const Dim_t &**get_nb_quad**() const

return the number of quadrature points stored per pixel

virtual void **check_material_coverage**() const

makes sure every pixel has been assigned to exactly one material

void **initialise**(*muFFT*::*FFT_PlanFlags* flags = *muFFT*::*FFT_PlanFlags*::*estimate*)

initialise the projection, the materials and the global fields

const *muGrid*::*CcoordOps*::*DynamicPixels* &**get_pixels**() const

return a const reference to the grids pixels iterator

muGrid::*FieldCollection*::*IndexIterable* **get_quad_pt_indices**() const

return an iterable proxy to this cell's field collection, iterable by quadrature point

muGrid::*FieldCollection*::*PixelIndexIterable* **get_pixel_indices**() const

return an iterable proxy to this cell's field collection, iterable by pixel

muGrid::*RealField* &**get_strain**()

return a reference to the cell's strain field

const *muGrid*::*RealField* &**get_stress**() const

return a const reference to the cell's stress field

const *muGrid*::*RealField* &**get_tangent**(bool do_create = false)

return a const reference to the cell's field of tangent moduli

virtual const *muGrid*::*RealField* &**evaluate_stress**()

evaluates and returns the stress for the currently set strain

Eigen_cmap **evaluate_stress_eigen**()

evaluates and returns the stress for the currently set strain

virtual std::tuple<const *muGrid*::*RealField*&, const *muGrid*::*RealField*&> **evaluate_stress_tangent**()

evaluates and returns the stress and tangent moduli for the currently set strain

std::tuple<const *Eigen_cmap*, const *Eigen_cmap*> **evaluate_stress_tangent_eigen**()

evaluates and returns the stress and tangent moduli for the currently set strain

muGrid::*RealField* &**globalise_real_internal_field**(const std::string &unique_name)

collect the real-valued fields of name `unique_name` of each material in the cell and write their values into a global field of same type and name

muGrid::*IntField* &**globalise_int_internal_field**(const std::string &unique_name)

collect the integer-valued fields of name `unique_name` of each material in the cell and write their values into a global field of same type and name

muGrid::*UintField* &**globalise_uint_internal_field**(const std::string &unique_name)

collect the unsigned integer-valued fields of name `unique_name` of each material in the cell and write their values into a global field of same type and name

muGrid::*ComplexField* &**globalise_complex_internal_field**(const std::string &unique_name)

collect the complex-valued fields of name `unique_name` of each material in the cell and write their values into a global field of same type and name

muGrid::*GlobalFieldCollection* &**get_fields**()

return a reference to the cell's global fields

void **apply_projection**(*muGrid*::TypedFieldBase<Real> &field)

apply the cell's projection operator to field `field` (i.e., return G:f)

void **evaluate_projected_directional_stiffness**(const *muGrid*::TypedFieldBase<Real> &delta_strain, *muGrid*::TypedFieldBase<Real> &del_stress)

evaluates the directional and projected stiffness (this corresponds to G:K:F (note the negative sign in de Geus 2017, http://dx.doi.org/10.1016/j.cma.2016.12.032).

void **add_projected_directional_stiffness**(*EigenCVec_t* delta_strain, const Real &alpha, *EigenVec_t* del_stress)

evaluates the directional and projected stiffness (this corresponds to G:K:F (note the negative sign in de Geus 2017, http://dx.doi.org/10.1016/j.cma.2016.12.032). and then adds it do the values already in del_stress, scaled by alpha (i.e., del_stress += alpha*Q:K:Strain. This function should not be used directly,

as it does absolutely no input checking. Rather, it is meant to be called by the scaleAndAddTo function in the *CellAdaptor*

inline *SplitCell* **get_splitness**() const

transitional function, use discouraged

const *ProjectionBase* &**get_projection**() const

return a const ref to the projection implementation

bool **is_point_inside**(const *DynRcoord_t* &point) const

check if the pixel is inside of the cell

bool **is_pixel_inside**(const *DynCcoord_t* &pixel) const

check if the point is inside of the cell

Protected Functions

template<typename **T**>
muGrid::*TypedField*<*T*> &**globalise_internal_field**(const std::string &unique_name)

helper function for the globalise_<T>_internal_field() functions

Protected Attributes

bool **initialised** = {false}

to handle double initialisations right

std::vector<*Material_ptr*> **materials** = {}

container of the materials present in the cell

Projection_ptr **projection**

handle for the projection operator

std::unique_ptr<*muGrid*::*GlobalFieldCollection*> **fields**

handle for the global fields associated with this cell

muGrid::*RealField* &**strain**

ref to strain field

muGrid::*RealField* &**stress**

ref to stress field

optional<std::reference_wrapper<*muGrid*::*RealField*>> **tangent** = {}

Tangent field might not even be required; so this is an optional ref_wrapper instead of a ref

SplitCell **is_cell_split** = {*SplitCell*::*no*}

Protected Static Functions

template<Dim_t **DimM**>
static void **apply_directional_stiffness**(const *muGrid*::TypedFieldBase<Real> &delta_strain, const *muGrid*::TypedFieldBase<Real> &tangent, *muGrid*::TypedFieldBase<Real> &delta_stress)

statically dimensioned worker for evaluating the tangent operator

template<Dim_t **DimM**>
static void **add_projected_directional_stiffness_helper**(const *muGrid*::TypedFieldBase<Real> &delta_strain, const *muGrid*::TypedFieldBase<Real> &tangent, const Real &alpha, *muGrid*::TypedFieldBase<Real> &delta_stress)

statically dimensioned worker for evaluating the incremental tangent operator

template<class **Cell**>

class **CellAdaptor** : public *Eigen*::EigenBase<*CellAdaptor*<*Cell*>>

#include <cell.hh> *Cell* adaptors implement the matrix-vector multiplication and allow the system to be used like a sparse matrix in conjugate-gradient-type solvers

lightweight resource handle wrapping a *muSpectre::Cell* or a subclass thereof into `Eigen::EigenBase`, so it can be interpreted as a sparse matrix by Eigen solvers

Public Types

enum **[anonymous]**

Values:

enumerator **ColsAtCompileTime**

enumerator **MaxColsAtCompileTime**

enumerator **RowsAtCompileTime**

enumerator **MaxRowsAtCompileTime**

enumerator **IsRowMajor**

using **Scalar** = double

sparse matrix traits

using **RealScalar** = double

sparse matrix traits

using **StorageIndex** = int

sparse matrix traits

Public Functions

inline explicit **CellAdaptor**(*Cell* &cell)

constructor

inline *Eigen*::Index **rows**() const

returns the number of logical rows

inline *Eigen*::Index **cols**() const

returns the number of logical columns

template<typename **Rhs**>
inline *Eigen*::Product<*CellAdaptor*, *Rhs*, *Eigen*::AliasFreeProduct> **operator***(const *Eigen*::MatrixBase<*Rhs*> &x) const

implementation of the evaluation

Public Members

Cell &**cell**

ref to the cell

class **CellSplit** : public *muSpectre*::*Cell*

#include <cell_split.hh> DimS spatial dimension (dimension of problem DimM material_dimension (dimension of constitutive law)

Public Types

using **Parent** = *Cell*

base class

using **Projection_ptr** = std::unique_ptr<*ProjectionBase*>

projections handled through `std::unique_ptr`s

using **FullResponse_t** = std::tuple<const *muGrid*::*RealField*&, const *muGrid*::*RealField*&>

combined stress and tangent field

Public Functions

CellSplit() = delete

Default constructor.

explicit **CellSplit**(*Projection_ptr* projection)

constructor using sizes and resolution

CellSplit(const *CellSplit* &other) = delete

Copy constructor.

CellSplit(*CellSplit* &&other) = default

Move constructor.

virtual **~CellSplit**() = default

Destructor.

CellSplit &**operator=**(const *CellSplit* &other) = delete

Copy assignment operator.

CellSplit &**operator=**(*CellSplit* &&other) = delete

Move assignment operator.

virtual *MaterialBase* &**add_material**(Material_ptr mat) final

add a new material to the cell

void **complete_material_assignment**(*MaterialBase* &material)

completes the assignmnet of material with a specific material so all the under-assigned pixels would be assigned to a material.

std::vector<Real> **get_assigned_ratios**()

void **make_automatic_precipitate_split_pixels**(const std::vector<*DynRcoord_t*> &preciptiate_vertices, *MaterialBase* &material)

std::vector<Real> **get_unassigned_ratios_incomplete_pixels**() const

std::vector<int> **get_index_incomplete_pixels**() const

std::vector<*DynCcoord_t*> **get_unassigned_pixels**()

IncompletePixels **make_incomplete_pixels**()

virtual void **check_material_coverage**() const final

makes sure every pixel has been assigned to materials whose ratios add up to 1.0

virtual const *muGrid*::*RealField* &**evaluate_stress**() final

evaluates and returns the stress for the currently set strain

virtual std::tuple<const *muGrid*::*RealField*&, const *muGrid*::*RealField*&> **evaluate_stress_tangent**() final

evaluates and returns the stress and tangent moduli for the currently set strain

Protected Functions

void **set_p_k_zero**()

Friends

friend class Cell

class **Communicator**

#include <communicator.hh> stub communicator object that doesn't communicate anything

Public Functions

inline **Communicator**()

inline **~Communicator**()

inline int **rank**() const

get rank of present process

inline int **size**() const

get total number of processes

template<typename **T**>
inline *T* **sum**(const *T* &arg) const

sum reduction on scalar types

template<typename **T**>
inline *Matrix_t*<*T*> **sum_mat**(const *Eigen*::Ref<*Matrix_t*<*T*>> &arg) const

sum reduction on EigenMatrix types

template<typename **T**>
inline *Matrix_t*<*T*> **gather**(const *Eigen*::Ref<*Matrix_t*<*T*>> &arg) const

gather on EigenMatrix types

template<typename **T**>
auto **sum_mat**(const *Eigen*::Ref<*Matrix_t*<*T*>> &arg) const -> *Matrix_t*<*T*>

sum reduction on EigenMatrix types

template<typename **T**>
auto **gather**(const *Eigen*::Ref<*Matrix_t*<*T*>> &arg) const -> *Matrix_t*<*T*>

gather on EigenMatrix types

Public Static Functions

static inline bool **has_mpi**()

find whether the underlying communicator is mpi

class **ConvergenceError** : public *muSpectre*::*SolverError*

template<*ElasticModulus* **Out**, *ElasticModulus* **In1**, *ElasticModulus* **In2**>

struct **Converter**

#include <materials_toolbox.hh> Base template for elastic modulus conversion.

Public Static Functions

static inline constexpr Real **compute**(const Real&, const Real&)

wrapped function (raison d'être)

template<>

struct **Converter**<*ElasticModulus*::*Bulk*, *ElasticModulus*::*lambda*, *ElasticModulus*::*Shear*>

#include <materials_toolbox.hh> Specialisation K(, µ)

Public Static Functions

static inline constexpr Real **compute**(const Real &lambda, const Real &G)
wrapped function (raison d'être)

template<>

struct **Converter**<*ElasticModulus*::*Bulk*, *ElasticModulus*::*Young*, *ElasticModulus*::*Poisson*>
#include <materials_toolbox.hh> Specialisation K(E,)

Public Static Functions

static inline constexpr Real **compute**(const Real &E, const Real &nu)
wrapped function (raison d'être)

template<>

struct **Converter**<*ElasticModulus*::*lambda*, *ElasticModulus*::*Bulk*, *ElasticModulus*::*Shear*>
#include <materials_toolbox.hh> Specialisation (K, µ)

Public Static Functions

static inline constexpr Real **compute**(const Real &K, const Real &mu)
wrapped function (raison d'être)

template<>

struct **Converter**<*ElasticModulus*::*lambda*, *ElasticModulus*::*Young*, *ElasticModulus*::*Poisson*>
#include <materials_toolbox.hh> Specialisation (E,)

Public Static Functions

static inline constexpr Real **compute**(const Real &E, const Real &nu)
wrapped function (raison d'être)

template<>

struct **Converter**<*ElasticModulus*::*Poisson*, *ElasticModulus*::*Bulk*, *ElasticModulus*::*Shear*>
#include <materials_toolbox.hh> Specialisation (K, µ)

Public Static Functions

static inline constexpr Real **compute**(const Real &K, const Real &G)
wrapped function (raison d'être)

template<>

struct **Converter**<*ElasticModulus*::*Shear*, *ElasticModulus*::*Young*, *ElasticModulus*::*Poisson*>
#include <materials_toolbox.hh> Specialisation (E,)

Public Static Functions

static inline constexpr Real **compute**(const Real &E, const Real &nu)
wrapped function (raison d'être)

template<>

struct **Converter**<*ElasticModulus*::*Young*, *ElasticModulus*::*Bulk*, *ElasticModulus*::*Shear*>
#include <materials_toolbox.hh> Specialisation E(K, µ)

Public Static Functions

static inline constexpr Real **compute**(const Real &K, const Real &G)
wrapped function (raison d'être)

template<>

struct **Converter**<*ElasticModulus*::*Young*, *ElasticModulus*::*lambda*, *ElasticModulus*::*Shear*>
#include <materials_toolbox.hh> Specialisation E(, µ)

Public Static Functions

static inline constexpr Real **compute**(const Real &lambda, const Real &G)
wrapped function (raison d'être)

template<*ElasticModulus* **Out**, *ElasticModulus* **In**>

struct **Converter**<*Out*, *In*, *Out*>
#include <materials_toolbox.hh> Spectialisation for when the output is the second input

Public Static Functions

static inline constexpr Real **compute**(const Real&, const Real &B)
wrapped function (raison d'être)

template<*ElasticModulus* **Out**, *ElasticModulus* **In**>

struct **Converter**<*Out*, *Out*, *In*>
#include <materials_toolbox.hh> Spectialisation for when the output is the first input

Public Static Functions

static inline constexpr Real **compute**(const Real &A, const Real&)
wrapped function (raison d'être)

template<*StrainMeasure* **In**, *StrainMeasure* **Out** = *In*>

struct **ConvertStrain**
#include <materials_toolbox.hh> Structure for functions returning one strain measure as a function of another

Public Static Functions

template<class **Strain_t**>
static inline decltype(auto) **compute**(*Strain_t* &&input)

returns the converted strain

template<>

struct **ConvertStrain**<*StrainMeasure*::*Gradient*, *StrainMeasure*::*GreenLagrange*>

#include <materials_toolbox.hh> Specialisation for getting Green-Lagrange strain from the transformation gradient E = $^1/_2$ (C - I) = $^1/_2$ (F·F - I)

Public Static Functions

template<class **Strain_t**>
static inline decltype(auto) **compute**(*Strain_t* &&F)

returns the converted strain

template<>

struct **ConvertStrain**<*StrainMeasure*::*Gradient*, *StrainMeasure*::*LCauchyGreen*>

#include <materials_toolbox.hh> Specialisation for getting Left Cauchy-Green strain from the transformation gradient B = F·F = V^2

Public Static Functions

template<class **Strain_t**>
static inline decltype(auto) **compute**(*Strain_t* &&F)

returns the converted strain

template<>

struct **ConvertStrain**<*StrainMeasure*::*Gradient*, *StrainMeasure*::*Log*>

#include <materials_toolbox.hh> Specialisation for getting logarithmic (Hencky) strain from the transformation gradient E_0 = $^1/_2$ ln C = $^1/_2$ ln (F·F)

Public Static Functions

template<class **Strain_t**>
static inline decltype(auto) **compute**(*Strain_t* &&F)

returns the converted strain

template<>

struct **ConvertStrain**<*StrainMeasure*::*Gradient*, *StrainMeasure*::*RCauchyGreen*>

#include <materials_toolbox.hh> Specialisation for getting Right Cauchy-Green strain from the transformation gradient C = F·F = U^2

Public Static Functions

template<class **Strain_t**>
static inline decltype(auto) **compute**(*Strain_t* &&F)

returns the converted strain

template<Dim_t **DimS**>

class **Correction**

Public Static Functions

static *Rcoord_t*<3> **correct_origin**(const *Rcoord_t*<*DimS*> &array)

static *Rcoord_t*<3> **correct_length**(const *Rcoord_t*<*DimS*> &array)

static std::vector<*Rcoord_t*<3>> **correct_vector**(const std::vector<*Rcoord_t*<*DimS*>> &vector)

template<>

class **Correction**<2>

Public Static Functions

static inline std::vector<*Rcoord_t*<3>> **correct_vector**(const std::vector<*Rcoord_t*<2>> &vertices)

static inline *Rcoord_t*<3> **correct_origin**(const *Rcoord_t*<2> &array)

static inline *Rcoord_t*<3> **correct_length**(const *Rcoord_t*<2> &array)

template<>

class **Correction**<3>

Public Static Functions

static inline *Rcoord_t*<3> **correct_origin**(const *Rcoord_t*<3> &array)

static inline *Rcoord_t*<3> **correct_length**(const *Rcoord_t*<3> &array)

static inline std::vector<*Rcoord_t*<3>> **correct_vector**(const std::vector<*Rcoord_t*<3>> &vertices)

template<Dim_t **Dim**>

struct **DefaultOrder**

#include <geometry.hh> convenience structure providing the default order of rotations around (in order) the z, x, and y axis

Public Static Attributes

static constexpr *RotationOrder* **value** = {*RotationOrder*::*ZXYTaitBryan*}

holds the value of the rotation order

template<>

struct **DefaultOrder**<twoD>

#include <geometry.hh> specialisation for two-dimensional problems

Public Static Attributes

static constexpr *RotationOrder* **value** = {*RotationOrder*::*Z*}

holds the value of the rotation order

class **DerivativeBase**

#include <derivative.hh> Representation of a derivative

Subclassed by *muFFT::DiscreteDerivative*, *muFFT::FourierDerivative*

Public Types

using **Vector** = *Eigen*::Matrix<Real, *Eigen*::Dynamic, 1>

convenience alias

Public Functions

DerivativeBase() = delete

Deleted default constructor.

explicit **DerivativeBase**(Dim_t spatial_dimension)

constructor with spatial dimension

DerivativeBase(const *DerivativeBase* &other) = default

Copy constructor.

DerivativeBase(*DerivativeBase* &&other) = default

Move constructor.

virtual **~DerivativeBase**() = default

Destructor.

DerivativeBase &**operator=**(const *DerivativeBase* &other) = delete

Copy assignment operator.

DerivativeBase &**operator=**(*DerivativeBase* &&other) = delete

Move assignment operator.

virtual Complex **fourier**(const *Vector* &phase) const = 0

Return Fourier representation of the derivative as a function of the phase. The phase is the wavevector times cell dimension, but lacking a factor of 2 .

Protected Attributes

Dim_t **spatial_dimension**

spatial dimension of the problem

class **DerivativeError** : public runtime_error

#include <derivative.hh> base class for projection related exceptions

Public Functions

inline explicit **DerivativeError**(const std::string &what)

constructor

inline explicit **DerivativeError**(const char *what)

constructor

template<class **Derived**>

struct **DimCounter**

template<class **Derived**>

struct **DimCounter**<*Eigen*::MatrixBase<*Derived*>>

#include <T4_map_proxy.hh> Convenience structure to determine the spatial dimension of a tensor represented by a fixed-size `Eigen::Matrix`. used to derive spatial dimension from input arguments of template functions thus avoiding the need for redundant explicit specification.

Public Static Attributes

static constexpr *Dim_t* **value** = {*ct_sqrt*(*Rows*)}

storage for the dimension

Private Types

using **Type** = *Eigen*::MatrixBase<*Derived*>

Private Static Attributes

static constexpr *Dim_t* **Rows** = {*Type*::RowsAtCompileTime}

class **DiscreteDerivative** : public *muFFT*::*DerivativeBase*

#include <derivative.hh> Representation of a finite-differences stencil

Public Types

using **Parent** = *DerivativeBase*

base class

using **Vector** = typename *Parent*::Vector

convenience alias

Public Functions

DiscreteDerivative() = delete

Default constructor.

DiscreteDerivative(*DynCcoord_t* nb_pts, *DynCcoord_t* lbounds, const std::vector<Real> &stencil)

Constructor with raw stencil information

Parameters

- **nb_pts** – stencil size
- **lbounds** – relative starting point of stencil
- **stencil** – stencil coefficients

DiscreteDerivative(*DynCcoord_t* nb_pts, *DynCcoord_t* lbounds, const *Eigen*::ArrayXd &stencil)

Constructor with raw stencil information.

DiscreteDerivative(const *DiscreteDerivative* &other) = default

Copy constructor.

DiscreteDerivative(*DiscreteDerivative* &&other) = default

Move constructor.

virtual **~DiscreteDerivative**() = default

Destructor.

DiscreteDerivative &**operator=**(const *DiscreteDerivative* &other) = delete

Copy assignment operator.

DiscreteDerivative &**operator=**(*DiscreteDerivative* &&other) = delete

Move assignment operator.

inline Real **operator()**(const *DynCcoord_t* &dcoord) const

Return stencil value.

inline const *DynCcoord_t* &**get_nb_pts**() const

Return number of grid points in stencil.

inline const *DynCcoord_t* &**get_lbounds**() const

Return lower stencil bound.

inline virtual Complex **fourier**(const *Vector* &phase) const

Any translationally invariant linear combination of grid values (as expressed through a "stencil") becomes a multiplication with a number in Fourier space. This method returns the Fourier representation of this stencil.

DiscreteDerivative **rollaxes**(int distance = 1) const

Return a new stencil rolled axes. Given a stencil on a three-dimensional grid with axes (x, y, z), the stencil that has been "rolled" by distance one has axes (z, x, y). This is a simple implementation of a rotation operation. For example, given a stencil that described the derivative in the x-direction, rollaxes(1) gives the derivative in the y-direction and rollaxes(2) gives the derivative in the z-direction.

Protected Attributes

const *DynCcoord_t* **nb_pts**

Number of stencil points.

const *DynCcoord_t* **lbounds**

Lower bound of the finite-differences stencil.

const *Eigen*::ArrayXd **stencil**

Finite-differences stencil.

template<*Dim_t* **Dim**, *Dim_t* **Rank1**, *Dim_t* **Rank2**>

struct **Dotter**

template<*Dim_t* **Dim**>

struct **Dotter**<*Dim*, *fourthOrder*, *fourthOrder*>

#include <tensor_algebra.hh> Double contraction between two fourth-rank tensors A and B returns a fourth-rank tensor C = A·B

Public Static Functions

template<class **T1**, class **T2**>
static inline decltype(auto) constexpr **ddot**(*T1* &&t1, *T2* &&t2)

raison d'être

template<*Dim_t* **Dim**>

struct **Dotter**<*Dim*, *fourthOrder*, *secondOrder*>

#include <tensor_algebra.hh> Tensor-product between a fourth-rank tensor A and a second-rank tensor B. Returns a fourth-rank C = A·B

Public Static Functions

template<class **T4**, class **T2**>
static inline decltype(auto) constexpr **dot**(*T4* &&t4, *T2* &&t2)

raison d'être

template<*Dim_t* **Dim**>

struct **Dotter**<*Dim*, *secondOrder*, *fourthOrder*>

#include <tensor_algebra.hh> Tensor-product between a second-rank tensor A and a fourth-rank tensor B. Returns a fourth-rank C = A·B

Public Static Functions

template<class **T1**, class **T2**>
static inline decltype(auto) constexpr **dot**(*T1* &&t1, *T2* &&t2)

raison d'être

template<*Dim_t* **Dim**>

struct **Dotter**<*Dim*, *secondOrder*, *secondOrder*>

#include <tensor_algebra.hh> Double contraction between two second-rank tensors A and B returns a scalar c = AB

Public Static Functions

template<class **T1**, class **T2**>
static inline decltype(auto) constexpr **ddot**(*T1* &&t1, *T2* &&t2)

raison d'être

class **DynamicPixels**

#include <ccoord_operations.hh> Iteration over square (or cubic) discretisation grids. Duplicates capabilities of `muGrid::Ccoordops::Pixels` without needing to be templated with the spatial dimension. Iteration is slower, though.

Subclassed by *muGrid::CcoordOps::Pixels< Dim >*

Public Functions

DynamicPixels()

explicit **DynamicPixels**(const *DynCcoord_t* &nb_grid_pts, const *DynCcoord_t* &locations = *DynCcoord_t*{})

Constructor with default strides (column-major pixel storage order)

DynamicPixels(const *DynCcoord_t* &nb_grid_pts, const *DynCcoord_t* &locations, const *DynCcoord_t* &strides)

Constructor with custom strides (any, including partially transposed pixel storage order)

template<size_t **Dim**>
explicit **DynamicPixels**(const *Ccoord_t*<*Dim*> &nb_grid_pts, const *Ccoord_t*<*Dim*> &locations = *Ccoord_t*<*Dim*>{})

Constructor with default strides from statically sized coords.

template<size_t **Dim**>
DynamicPixels(const *Ccoord_t*<*Dim*> &nb_grid_pts, const *Ccoord_t*<*Dim*> &locations, const *Ccoord_t*<*Dim*> &strides)

Constructor with custom strides from statically sized coords.

DynamicPixels(const *DynamicPixels* &other) = default

Copy constructor.

DynamicPixels(*DynamicPixels* &&other) = default

Move constructor.

virtual **~DynamicPixels**() = default

Destructor.

DynamicPixels &**operator=**(const *DynamicPixels* &other) = default

Copy assignment operator.

DynamicPixels &**operator=**(*DynamicPixels* &&other) = default

Move assignment operator.

inline *Dim_t* **get_index**(const *DynCcoord_t* &ccoord) const

evaluate and return the linear index corresponding to dynamic `ccoord`

template<size_t **Dim**>
inline *Dim_t* **get_index**(const *Ccoord_t*<*Dim*> &ccoord) const

evaluate and return the linear index corresponding to `ccoord`

template<size_t **Dim**>
const Pixels<*Dim*> &**get_dimensioned_pixels**() const

return a reference to the *Pixels* object cast into a statically dimensioned grid. the statically dimensioned version duplicates `muGrid::Ccoordops::DynamicPixels`'s capabilities, but iterates much more efficiently.

iterator **begin**() const

stl conformance

iterator **end**() const

stl conformance

size_t **size**() const

stl conformance

inline const *Dim_t* &**get_dim**() const

return spatial dimension

inline const *DynCcoord_t* &**get_nb_grid_pts**() const

return the resolution of the discretisation grid in each spatial dim

inline const *DynCcoord_t* &**get_locations**() const

return the ccoordinates of the bottom, left, (front) pixel/voxel of this processors partition of the discretisation grid. For sequential calculations, this is always the origin

inline const *DynCcoord_t* &**get_strides**() const

return the strides used for iterating over the pixels

Enumerator **enumerate**() const

iterates in tuples of pixel index ond coordinate. Useful in parallel problems, where simple enumeration of the pixels would be incorrect

Protected Attributes

Dim_t **dim**

spatial dimension

DynCcoord_t **nb_grid_pts**

nb_grid_pts of this domain

DynCcoord_t **locations**

locations of this domain

DynCcoord_t **strides**

strides of memory layout

template<size_t **MaxDim**, typename **T** = *Dim_t*>

class **DynCcoord**

#include <grid_common.hh> Class to represent integer (cell-) coordinates or real-valued coordinates. This class can dynamically accept any spatial-dimension between 1 and MaxDim, and *DynCcoord* references can be cast to `muGrid::Ccoord_t &` or `muGrid::Rcoord_t &` references. These are used when templating with the spatial dimension of the problem is undesireable/impossible.

Public Types

using **iterator** = typename std::array<*T*, *MaxDim*>::iterator

iterator type

using **const_iterator** = typename std::array<*T*, *MaxDim*>::const_iterator

constant iterator type

Public Functions

inline **DynCcoord**()

default constructor

inline **DynCcoord**(std::initializer_list<*T*> init_list)

constructor from an initialiser list for compound initialisation.

Parameters

init_list – The length of the initialiser list becomes the spatial dimension of the coordinate, therefore the list must have a length between 1 and MaxDim

inline explicit **DynCcoord**(*Dim_t* dim)

Constructor only setting the dimension. WARNING: This constructor *needs* regular (round) braces '()', using curly braces '{}' results in the initialiser list constructor being called and creating a *DynCcoord* with spatial dimension 1

Parameters

dim – spatial dimension. Needs to be between 1 and MaxDim

template<size_t **Dim**>
inline explicit **DynCcoord**(const std::array<*T*, *Dim*> &ccoord)

Constructor from a statically sized coord.

inline explicit **DynCcoord**(const std::vector<*T*> &ccoord)

DynCcoord(const *DynCcoord* &other) = default

Copy constructor.

DynCcoord(*DynCcoord* &&other) = default

Move constructor.

~DynCcoord() = default

nonvirtual Destructor

template<size_t **Dim**>
inline *DynCcoord* &**operator=**(const std::array<*T*, *Dim*> &ccoord)

Assign arrays.

DynCcoord &**operator=**(const *DynCcoord* &other) = default

Copy assignment operator.

DynCcoord &**operator=**(*DynCcoord* &&other) = default

Move assignment operator.

template<size_t **Dim2**>
inline bool **operator==**(const std::array<*T*, *Dim2*> &other) const

comparison operator

inline bool **operator==**(const *DynCcoord* &other) const

comparison operator

template<typename **T2**>
inline *DynCcoord*<*MaxDim*, decltype(*T*{} / *T2*{})> **operator/**(const *DynCcoord*<*MaxDim*, *T2*> &other) const

element-wise division

inline *T* &**operator[]**(const size_t &index)

access operator

inline const *T* &**operator[]**(const size_t &index) const

access operator

template<size_t **Dim**>
inline **operator std::array<*T*, *Dim*>**() const

conversion operator

template<*Dim_t* **Dim**>
inline std::array<*T*, *Dim*> &**get**()

cast to a reference to a statically sized array

template<*Dim_t* **Dim**>
inline const std::array<*T*, *Dim*> &**get**() const

cast to a const reference to a statically sized array

inline const *Dim_t* &**get_dim**() const

return the spatial dimension of this coordinate

inline *iterator* **begin**()

iterator to the first entry for iterating over only the valid entries

inline *iterator* **end**()

iterator past the dim-th entry for iterating over only the valid entries

inline *const_iterator* **begin**() const

const iterator to the first entry for iterating over only the valid entries

inline *const_iterator* **end**() const

const iterator past the dim-th entry for iterating over only the valid entries

inline *T* ***data**()

return the underlying data pointer

inline const *T* ***data**() const

return the underlying data pointer

inline *T* &**back**()

return a reference to the last valid entry

inline const *T* &**back**() const

return a const reference to the last valid entry

Protected Attributes

Dim_t **dim**

spatial dimension of the coordinate

std::array<*T*, *MaxDim*> **long_array**

storage for coordinate components

Private Functions

template<size_t **Dim**>
inline constexpr std::array<*T*, *MaxDim*> **fill_front**(const std::array<*T*, *Dim*> &ccoord)

Private Static Functions

template<size_t **Dim**, size_t... **Indices**>
static inline constexpr std::array<*T*, *MaxDim*> **fill_front_helper**(const std::array<*T*, *Dim*> &ccoord, std::index_sequence<*Indices*...>)

template<typename **T**, class **EigenPlain**>

struct **EigenMap**

#include <field_map_static.hh> Internal struct for handling the matrix-shaped iterates of *muGrid::FieldMap*

Public Types

using **PlainType** = *EigenPlain*

Eigen type of the iterate.

using **value_type** = std::conditional_t<MutIter == *Mapping*::*Const*, *Eigen*::Map<const *PlainType*>, *Eigen*::Map<*PlainType*>>

stl (const-correct)

using **ref_type** = *value_type*<MutIter>

stl (const-correct)

using **Return_t** = *value_type*<MutIter>

for direct access through operator[]

using **storage_type** = *value_type*<MutIter>

stored type (cannot always be same as ref_type)

Public Static Functions

static inline constexpr bool **IsValidStaticMapType**()

check at compile time whether the type is meant to be a map with statically sized iterates.

static inline constexpr bool **IsScalarMapType**()

check at compiler time whether this map is scalar

template<*Mapping* **MutIter**>
static inline constexpr *value_type*<*MutIter*> &**provide_ref**(*storage_type*<*MutIter*> &storage)

return the return_type version of the iterate from storage_type

template<*Mapping* **MutIter**>
static inline constexpr const *value_type*<*MutIter*> &**provide_const_ref**(const *storage_type*<*MutIter*> &storage)

return the const return_type version of the iterate from storage_type

template<*Mapping* **MutIter**>
static inline constexpr *value_type*<*MutIter*> ***provide_ptr**(*storage_type*<*MutIter*> &storage)

return a pointer to the iterate from storage_type

template<*Mapping* **MutIter**>
static inline constexpr *Return_t*<*MutIter*> **from_data_ptr**(std::conditional_t<*MutIter* == *Mapping*::*Const*, const *T**, *T**> data)

return a return_type version of the iterate from its pointer

template<*Mapping* **MutIter**>
static inline constexpr *storage_type*<*MutIter*> **to_storage**(*value_type*<*MutIter*> &&value)

return a storage_type version of the iterate from its value

static inline constexpr *Dim_t* **stride**()

return the nb of components of the iterate (known at compile time)

static inline std::string **shape**()

return the iterate's shape as text, mostly for error messages

static inline constexpr *Dim_t* **NbRow**()

class **Enumerator**

#include <ccoord_operations.hh> enumerator class for `muSpectre::DynamicPixels`

Public Functions

Enumerator() = delete

Default constructor.

explicit **Enumerator**(const *DynamicPixels* &pixels)

Constructor.

Enumerator(const *Enumerator* &other) = default

Copy constructor.

Enumerator(*Enumerator* &&other) = default

Move constructor.

virtual **~Enumerator**() = default

Destructor.

Enumerator &**operator=**(const *Enumerator* &other) = delete

Copy assignment operator.

Enumerator &**operator=**(*Enumerator* &&other) = delete

Move assignment operator.

iterator **begin**() const

stl conformance

iterator **end**() const

stl conformance

size_t **size**() const

stl conformance

Protected Attributes

const *DynamicPixels* &**pixels**

template<Dim_t **dim**>

class **FFT_freqs**

#include <fft_utils.hh> simple class encapsulating the creation, and retrieval of wave vectors

Public Types

using **CcoordVector** = *Eigen*::Matrix<Dim_t, *dim*, 1>

Eigen variant equivalent to Ccoord_t.

using **Vector** = *Eigen*::Matrix<Real, *dim*, 1>

return type for wave vectors

using **VectorComplex** = *Eigen*::Matrix<Complex, *dim*, 1>

return type for complex wave vectors

Public Functions

FFT_freqs() = delete
Default constructor.

inline explicit **FFT_freqs**(*Ccoord_t*<*dim*> nb_grid_pts)
constructor with just number of grid points

inline **FFT_freqs**(*Ccoord_t*<*dim*> nb_grid_pts, std::array<Real, *dim*> lengths)
constructor with domain length

FFT_freqs(const *FFT_freqs* &other) = delete
Copy constructor.

FFT_freqs(*FFT_freqs* &&other) = default
Move constructor.

virtual **~FFT_freqs**() = default
Destructor.

FFT_freqs &**operator=**(const *FFT_freqs* &other) = delete
Copy assignment operator.

FFT_freqs &**operator=**(*FFT_freqs* &&other) = default
Move assignment operator.

inline *Vector* **get_xi**(const *Ccoord_t*<*dim*> ccoord) const
get unnormalised wave vector (in sampling units)

inline *VectorComplex* **get_complex_xi**(const *Ccoord_t*<*dim*> ccoord) const
get unnormalised complex wave vector (in sampling units)

inline *Vector* **get_unit_xi**(const *Ccoord_t*<*dim*> ccoord) const
get normalised wave vector

inline Dim_t **get_nb_grid_pts**(Dim_t i) const

Protected Attributes

const std::array<std::valarray<Real>, *dim*> **freqs**
container for frequencies ordered by spatial dimension

class **FFTEngineBase**

#include <fft_engine_base.hh> Virtual base class for FFT engines. To be implemented by all FFT_engine implementations.

Subclassed by *muFFT::FFTWEngine*, *muFFT::FFTWMPIEngine*

Public Types

using **GFieldCollection_t** = *muGrid*::*GlobalFieldCollection*

global FieldCollection

using **Pixels** = typename *GFieldCollection_t*::DynamicPixels

pixel iterator

using **Field_t** = *muGrid*::TypedFieldBase<Real>

Field type on which to apply the projection. This is a TypedFieldBase because it need to be able to hold either TypedField or a WrappedField.

using **Workspace_t** = *muGrid*::*ComplexField*

Field type holding a Fourier-space representation of a real-valued second-order tensor field

using **iterator** = typename *GFieldCollection_t*::DynamicPixels::iterator

iterator over Fourier-space discretisation point

Public Functions

FFTEngineBase() = delete

Default constructor.

FFTEngineBase(*DynCcoord_t* nb_grid_pts, Dim_t nb_dof_per_pixel, *Communicator* comm = *Communicator*())

Constructor with the domain's number of grid points in each direciton, the number of components to transform, and the communicator

FFTEngineBase(const *FFTEngineBase* &other) = delete

Copy constructor.

FFTEngineBase(*FFTEngineBase* &&other) = delete

Move constructor.

virtual **~FFTEngineBase**() = default

Destructor.

FFTEngineBase &**operator=**(const *FFTEngineBase* &other) = delete

Copy assignment operator.

FFTEngineBase &**operator=**(*FFTEngineBase* &&other) = delete

Move assignment operator.

virtual void **initialise**(*FFT_PlanFlags*)

compute the plan, etc

virtual *Workspace_t* &**fft**(*Field_t*&) = 0

forward transform (dummy for interface)

virtual void **ifft**(*Field_t*&) const = 0

inverse transform (dummy for interface)

inline virtual bool **is_active**() const

return whether this engine is active

const *Pixels* &**get_pixels**() const

iterators over only those pixels that exist in frequency space (i.e. about half of all pixels, see rfft)

size_t **size**() const

nb of pixels (mostly for debugging)

size_t **fourier_size**() const

nb of pixels in Fourier space

size_t **workspace_size**() const

nb of pixels in the work space (may contain a padding region)

inline const *Communicator* &**get_communicator**() const

return the communicator object

inline const *DynCcoord_t* &**get_nb_subdomain_grid_pts**() const

returns the process-local number of grid points in each direction of the cell

inline const *DynCcoord_t* &**get_nb_domain_grid_pts**() const

returns the process-local number of grid points in each direction of the cell

inline const *DynCcoord_t* &**get_subdomain_locations**() const

returns the process-local locations of the cell

inline const *DynCcoord_t* &**get_nb_fourier_grid_pts**() const

returns the process-local number of grid points in each direction of the cell in Fourier space

inline const *DynCcoord_t* &**get_fourier_locations**() const

returns the process-local locations of the cell in Fourier space

inline *GFieldCollection_t* &**get_field_collection**()

only required for testing and debugging

inline *Workspace_t* &**get_work_space**()

only required for testing and debugging

inline Real **normalisation**() const

factor by which to multiply projection before inverse transform (this is typically 1/nb_pixels for so-called unnormalized transforms (see, e.g. http://www.fftw.org/fftw3_doc/Multi_002dDimensional-DFTs-of-Real-Data.html#Multi_002dDimensional-DFTs-of-Real-Data or https://docs.scipy.org/doc/numpy-1.13.0/reference/routines.fft.html . Rather than scaling the inverse transform (which would cost one more loop), FFT engines provide this value so it can be used in the projection operator (where no additional loop is required)

const Dim_t &**get_nb_dof_per_pixel**() const

return the number of components per pixel

const Dim_t &**get_dim**() const

return the number of spatial dimensions

const Dim_t &**get_nb_quad**() const

returns the number of quadrature points

inline bool **is_initialised**() const

has this engine been initialised?

Protected Attributes

Dim_t **spatial_dimension**

spatial dimension of the grid

Communicator **comm**

Field collection in which to store fields associated with Fourier-space pointscommunicator

GFieldCollection_t **work_space_container**

Field collection to store the fft workspace.

DynCcoord_t **nb_subdomain_grid_pts**

nb_grid_pts of the process-local (subdomain) portion of the cell

DynCcoord_t **subdomain_locations**

location of the process-local (subdomain) portion of the cell

DynCcoord_t **nb_fourier_grid_pts**

nb_grid_pts of the process-local (subdomain) portion of the Fourier transformed data

DynCcoord_t **fourier_locations**

location of the process-local (subdomain) portion of the Fourier transformed data

const *DynCcoord_t* **nb_domain_grid_pts**

nb_grid_pts of the full domain of the cell

Workspace_t &**work**

field to store the Fourier transform of P

const Real **norm_factor**

normalisation coefficient of fourier transform

Dim_t **nb_dof_per_pixel**

number of degrees of freedom per pixel. Corresponds to the number of quadrature points per pixel multiplied by the number of components per quadrature point

bool **initialised** = {false}

to prevent double initialisation

class **FFTWEngine** : public *muFFT*::*FFTEngineBase*

#include <fftw_engine.hh> implements the `muFFT::FftEngine_Base` interface using the FFTW library

Public Types

using **Parent** = *FFTEngineBase*

base class

using **Workspace_t** = typename *Parent*::Workspace_t

field for Fourier transform of second-order tensor

using **Field_t** = typename *Parent*::Field_t

real-valued second-order tensor

Public Functions

FFTWEngine() = delete

Default constructor.

FFTWEngine(const *DynCcoord_t* &nb_grid_pts, Dim_t nb_dof_per_pixel, *Communicator* comm = *Communicator*())

Constructor with the domain's number of grid points in each direciton, the number of components to transform, and the communicator

FFTWEngine(const *FFTWEngine* &other) = delete

Copy constructor.

FFTWEngine(*FFTWEngine* &&other) = delete

Move constructor.

virtual **~FFTWEngine**() noexcept

Destructor.

FFTWEngine &**operator=**(const *FFTWEngine* &other) = delete

Copy assignment operator.

FFTWEngine &**operator=**(*FFTWEngine* &&other) = delete

Move assignment operator.

virtual void **initialise**(*FFT_PlanFlags* plan_flags) override

compute the plan, etc

virtual *Workspace_t* &**fft**(*Field_t* &field) override

forward transform

virtual void **ifft**(*Field_t* &field) const override

inverse transform

Protected Attributes

fftw_plan **plan_fft** = {}

holds the plan for forward fourier transform

fftw_plan **plan_ifft** = {}

holds the plan for inverse fourier transform

class **FFTWMPIEngine** : public *muFFT*::*FFTEngineBase*

#include <fftwmpi_engine.hh> implements the *muFFT::FFTEngineBase* interface using the FFTW library

Public Types

using **Parent** = *FFTEngineBase*

base class

using **Workspace_t** = typename *Parent*::Workspace_t

field for Fourier transform of second-order tensor

using **Field_t** = typename *Parent*::Field_t

real-valued second-order tensor

Public Functions

FFTWMPIEngine() = delete

Default constructor.

FFTWMPIEngine(*DynCcoord_t* nb_grid_pts, Dim_t nb_dof_per_pixel, *Communicator* comm = *Communicator*())

Constructor with the domain's number of grid points in each direciton, the number of components to transform, and the communicator

FFTWMPIEngine(const *FFTWMPIEngine* &other) = delete

Copy constructor.

FFTWMPIEngine(*FFTWMPIEngine* &&other) = delete

Move constructor.

virtual **~FFTWMPIEngine**() noexcept

Destructor.

FFTWMPIEngine &**operator=**(const *FFTWMPIEngine* &other) = delete

Copy assignment operator.

FFTWMPIEngine &**operator=**(*FFTWMPIEngine* &&other) = delete

Move assignment operator.

virtual void **initialise**(*FFT_PlanFlags* plan_flags) override

compute the plan, etc

virtual *Workspace_t* &**fft**(*Field_t* &field) override

forward transform

virtual void **ifft**(*Field_t* &field) const override

inverse transform

inline virtual bool **is_active**() const override

return whether this engine is active

Protected Attributes

fftw_plan **plan_fft** = {}

holds the plan for forward fourier transform

fftw_plan **plan_ifft** = {}

holds the plan for inverse fourier transform

ptrdiff_t **workspace_size** = {}

size of workspace buffer returned by planner

Real ***real_workspace** = {}

temporary real workspace that is correctly padded

bool **active** = {true}

FFTWMPI sometimes assigns zero grid points.

Protected Static Attributes

static int **nb_engines** = {0}

number of times this engine has been instatiated

class **Field**

#include <field.hh> Abstract base class for all fields. A field provides storage discretising a mathematical (scalar, vectorial, tensorial) (real-valued, integer-valued, complex-valued) field on a fixed number of quadrature points per pixel/voxel of a regular grid. Fields defined on the same domains are grouped within *muGrid::FieldCollection*s.

Subclassed by *muGrid::TypedFieldBase< T >*

Public Functions

Field() = delete

Default constructor.

Field(const *Field* &other) = delete

Copy constructor.

Field(*Field* &&other) = default

Move constructor.

virtual **~Field**() = default

Destructor.

Field &**operator=**(const *Field* &other) = delete

Copy assignment operator.

Field &**operator=**(*Field* &&other) = delete

Move assignment operator.

const std::string &**get_name**() const

return the field's unique name

FieldCollection &**get_collection**() const

return a const reference to the field's collection

const *Dim_t* &**get_nb_components**() const

return the number of components stored per quadrature point

std::vector<*Dim_t*> **get_shape**(*Iteration* iter_type) const

evaluate and return the overall shape of the field (for passing the field to generic multidimensional array objects such as numpy.ndarray)

std::vector<*Dim_t*> **get_pixels_shape**() const

evaluate and return the overall shape of the pixels portion of the field (for passing the field to generic multidimensional array objects such as numpy.ndarray)

virtual std::vector<*Dim_t*> **get_components_shape**(*Iteration* iter_type) const

evaluate and return the shape of the data contained in a single pixel or quadrature point (for passing the field to generic multidimensional array objects such as numpy.ndarray)

Dim_t **get_stride**(*Iteration* iter_type) const

evaluate and return the number of components in an iterate when iterating over this field

virtual const std::type_info &**get_stored_typeid**() const = 0

return the type information of the stored scalar (for compatibility checking)

size_t **size**() const

number of entries in the field (= nb_pixel × nb_quad)

virtual size_t **buffer_size**() const = 0

size of the internal buffer including the pad region (in scalars)

virtual void **set_pad_size**(size_t pad_size_) = 0

add a pad region to the end of the field buffer; required for using this as e.g. an FFT workspace

const size_t &**get_pad_size**() const

pad region size

virtual void **set_zero**() = 0

initialise field to zero (do more complicated initialisations through fully typed maps)

bool **is_global**() const

checks whether this field is registered in a global *FieldCollection*

Protected Functions

Field(const std::string &unique_name, *FieldCollection* &collection, *Dim_t* nb_components)

*Field*s are supposed to only exist in the form of `std::unique_ptr`s held by a *FieldCollection*. The *Field* constructor is protected to ensure this.

Parameters

- **unique_name** – unique field name (unique within a collection)
- **nb_components** – number of components to store per quadrature point
- **collection** – reference to the holding field collection.

virtual void **resize**(size_t size) = 0

resizes the field to the given size

Protected Attributes

friend FieldCollection

gives field collections the ability to *resize()* fields

size_t **current_size** = {}

maintains a tally of the current size, as it cannot be reliably determined from either `values` or `alt_values` alone.

const std::string **name**

the field's unique name

FieldCollection &**collection**

reference to the collection this field belongs to

const *Dim_t* **nb_components**

number of components stored per quadrature point (e.g., 3 for a three-dimensional vector, or 9 for a three-dimensional second-rank tensor)

size_t **pad_size** = {}

size of padding region at end of buffer

class **FieldCollection**

#include <field_collection.hh> Base class for both *muGrid::GlobalFieldCollection* and *muGrid::LocalFieldCollection*. Manages the a group of fields with the same domain of validitiy (i.e., global fields, or local fields defined on the same pixels).

Subclassed by *muGrid::GlobalFieldCollection*, *muGrid::LocalFieldCollection*

Public Types

enum **ValidityDomain**

domain of validity of the managed fields

Values:

enumerator **Global**

enumerator **Local**

using **Field_ptr** = std::unique_ptr<*Field*, *FieldDestructor*<*Field*>>

unique_ptr for holding fields

using **StateField_ptr** = std::unique_ptr<*StateField*, *FieldDestructor*<*StateField*>>

unique_ptr for holding state fields

using **QuadPtIndexIterable** = *IndexIterable*

convenience alias

Public Functions

FieldCollection() = delete

Default constructor.

FieldCollection(const *FieldCollection* &other) = delete

Copy constructor.

FieldCollection(*FieldCollection* &&other) = default

Move constructor.

virtual **~FieldCollection**() = default

Destructor.

FieldCollection &**operator=**(const *FieldCollection* &other) = delete

Copy assignment operator.

FieldCollection &**operator=**(*FieldCollection* &&other) = default

Move assignment operator.

template<typename **T**>
inline *TypedField*<*T*> &**register_field**(const std::string &unique_name, const *Dim_t* &nb_components)

place a new field in the responsibility of this collection (Note, because fields have protected constructors, users can't create them

Technically, these explicit instantiations are not necessary, as they are implicitly instantiated when the register_<T>field(...) member functions are compiled.

Parameters

- **unique_name** – unique identifier for this field

- **nb_components** – number of components to be stored per quadrature point (e.g., 4 for a two-dimensional second-rank tensor, or 1 for a scalar field)

TypedField<*Real*> &**register_real_field**(const std::string &unique_name, const *Dim_t* &nb_components)

place a new real-valued field in the responsibility of this collection (Note, because fields have protected constructors, users can't create them

Parameters

- **unique_name** – unique identifier for this field
- **nb_components** – number of components to be stored per quadrature point (e.g., 4 for a two-dimensional second-rank tensor, or 1 for a scalar field)

TypedField<*Complex*> &**register_complex_field**(const std::string &unique_name, const *Dim_t* &nb_components)

place a new complex-valued field in the responsibility of this collection (Note, because fields have protected constructors, users can't create them

Parameters

- **unique_name** – unique identifier for this field
- **nb_components** – number of components to be stored per quadrature point (e.g., 4 for a two-dimensional second-rank tensor, or 1 for a scalar field)

TypedField<*Int*> &**register_int_field**(const std::string &unique_name, const *Dim_t* &nb_components)

place a new integer-valued field in the responsibility of this collection (Note, because fields have protected constructors, users can't create them

Parameters

- **unique_name** – unique identifier for this field
- **nb_components** – number of components to be stored per quadrature point (e.g., 4 for a two-dimensional second-rank tensor, or 1 for a scalar field)

TypedField<*Uint*> &**register_uint_field**(const std::string &unique_name, const *Dim_t* &nb_components)

place a new unsigned integer-valued field in the responsibility of this collection (Note, because fields have protected constructors, users can't create them

Parameters

- **unique_name** – unique identifier for this field
- **nb_components** – number of components to be stored per quadrature point (e.g., 4 for a two-dimensional second-rank tensor, or 1 for a scalar field)

template<typename **T**>
inline *TypedStateField*<*T*> &**register_state_field**(const std::string &unique_prefix, const *Dim_t* &nb_memory, const *Dim_t* &nb_components)

place a new state field in the responsibility of this collection (Note, because state fields have protected constructors, users can't create them

TypedStateField<*Real*> &**register_real_state_field**(const std::string &unique_prefix, const *Dim_t* &nb_memory, const *Dim_t* &nb_components)

place a new real-valued state field in the responsibility of this collection (Note, because state fields have protected constructors, users can't create them

Parameters

- **unique_prefix** – unique idendifier for this state field
- **nb_memory** – number of previous values of this field to store
- **nb_components** – number of scalar components to store per quadrature point

TypedStateField<*Complex*> &**register_complex_state_field**(const std::string &unique_prefix, const *Dim_t* &nb_memory, const *Dim_t* &nb_components)

place a new complex-valued state field in the responsibility of this collection (Note, because state fields have protected constructors, users can't create them

Parameters

- **unique_prefix** – unique idendifier for this state field
- **nb_memory** – number of previous values of this field to store
- **nb_components** – number of scalar components to store per quadrature point

TypedStateField<*Int*> &**register_int_state_field**(const std::string &unique_prefix, const *Dim_t* &nb_memory, const *Dim_t* &nb_components)

place a new integer-valued state field in the responsibility of this collection (Note, because state fields have protected constructors, users can't create them

Parameters

- **unique_prefix** – unique idendifier for this state field
- **nb_memory** – number of previous values of this field to store
- **nb_components** – number of scalar components to store per quadrature point

TypedStateField<*Uint*> &**register_uint_state_field**(const std::string &unique_prefix, const *Dim_t* &nb_memory, const *Dim_t* &nb_components)

place a new unsigned integer-valued state field in the responsibility of this collection (Note, because state fields have protected constructors, users can't create them

Parameters

- **unique_prefix** – unique idendifier for this state field
- **nb_memory** – number of previous values of this field to store
- **nb_components** – number of scalar components to store per quadrature point

bool **field_exists**(const std::string &unique_name) const

check whether a field of name 'unique_name' has already been registered

bool **state_field_exists**(const std::string &unique_prefix) const

check whether a field of name 'unique_name' has already been registered

const *Dim_t* &**get_nb_entries**() const

returns the number of entries held by any given field in this collection. This corresponds to nb_pixels × nb_quad_pts, (I.e., a scalar field field and a vector field sharing the the same collection have the same number of entries, even though the vector field has more scalar values.)

size_t **get_nb_pixels**() const

returns the number of pixels present in the collection

bool **has_nb_quad**() const

check whether the number of quadrature points per pixel/voxel has ben set

void **set_nb_quad**(*Dim_t* nb_quad_pts_per_pixel)

set the number of quadrature points per pixel/voxel. Can only be done once.

const *Dim_t* &**get_nb_quad**() const

return the number of quadrature points per pixel

const *Dim_t* &**get_spatial_dim**() const

return the spatial dimension of the underlying discretisation grid

const *ValidityDomain* &**get_domain**() const

return the domain of validity (i.e., wher the fields are defined globally (`muGrid::FieldCollection::ValidityDomain::Global`) or locally (`muGrid::FieldCollection::ValidityDomain::Local`)

bool **is_initialised**() const

whether the collection has been properly initialised (i.e., it knows the number of quadrature points and all its pixels/voxels

PixelIndexIterable **get_pixel_indices_fast**() const

return an iterable proxy to the collection which allows to efficiently iterate over the indices fo the collection's pixels

IndexIterable **get_pixel_indices**() const

return an iterable proxy to the collection which allows to iterate over the indices fo the collection's pixels

IndexIterable **get_quad_pt_indices**() const

return an iterable proxy to the collection which allows to iterate over the indices fo the collection's quadrature points

inline std::vector<size_t> **get_pixel_ids**()

Field &**get_field**(const std::string &unique_name)

returns a (base-type) reference to the field identified by `unique_name`. Throws a *muGrid::FieldCollectionError* if the field does not exist.

StateField &**get_state_field**(const std::string &unique_prefix)

returns a (base-type) reference to the state field identified by `unique_prefix`. Throws a *muGrid::FieldCollectionError* if the state field does not exist.

std::vector<std::string> **list_fields**() const

returns a vector of all field names

void **preregister_map**(std::shared_ptr<std::function<void()>> &call_back)

preregister a map for latent initialisation

Protected Functions

FieldCollection(*ValidityDomain* domain, const *Dim_t* &spatial_dimension, const *Dim_t* &nb_quad_pts)

Constructor (not called by user, who constructs either a *LocalFieldCollection* or a *GlobalFieldCollection*

Parameters

- **domain** – Domain of validity, can be global or local
- **spatial_dimension** – spatial dimension of the field (can be muGrid::Unknown, e.g., in the case of the local fields for storing internal material variables)
- **nb_quad_pts** – number of quadrature points per pixel/voxel

template<typename **T**>
TypedField<*T*> &**register_field_helper**(const std::string &unique_name, const *Dim_t* &nb_components)

internal worker function called by register_<T>_field

template<typename **T**>
TypedStateField<*T*> &**register_state_field_helper**(const std::string &unique_prefix, const *Dim_t* &nb_memory, const *Dim_t* &nb_components)

internal worker function called by register_<T>_state_field

void **allocate_fields**()

loop through all fields and allocate their memory. Is exclusively called by the daughter classes' `initialise` member function.

void **initialise_maps**()

initialise all preregistered maps

Protected Attributes

std::map<std::string, *Field_ptr*> **fields** = {}

storage container for fields

std::map<std::string, *StateField_ptr*> **state_fields** = {}

storage container for state fields

std::vector<std::weak_ptr<std::function<void()>>> **init_callbacks** = {}

Maps registered before initialisation which will need their data_ptr set.

ValidityDomain **domain**

domain of validity

Dim_t **spatial_dim**

spatial dimension

Dim_t **nb_quad_pts**

number of quadrature points per pixel/voxel

Dim_t **nb_entries** = {*Unknown*}

total number of entries

bool **initialised** = {false}

keeps track of whether the collection has already been initialised

std::vector<size_t> **pixel_indices** = {}

Storage for indices of the stored quadrature points in the global field collection. Note that these are not truly global indices, but rather absolute indices within the domain of the local processor. I.e., they are universally valid to address any quadrature point on the local processor, and not for any quadrature point located on anothe processor.

class **FieldCollectionError** : public runtime_error

#include <field_collection.hh> base class for field collection-related exceptions

Public Functions

inline explicit **FieldCollectionError**(const std::string &what)

constructor

inline explicit **FieldCollectionError**(const char *what)

constructor

template<class **DefaultDestroyable**>

struct **FieldDestructor**

#include <field_collection.hh> forward declacation of the field's destructor-functor

Public Functions

void **operator()**(*DefaultDestroyable* *field)

deletes the held field

class **FieldError** : public runtime_error

#include <field.hh> base class for field-related exceptions

Public Functions

inline explicit **FieldError**(const std::string &what)

constructor

inline explicit **FieldError**(const char *what)

constructor

template<typename **T**, *Mapping* **Mutability**>

class **FieldMap**

#include <field_map.hh> forward declaration

Dynamically sized field map. *Field* maps allow iterating over the pixels or quadrature points of a field and to select the shape (in a matrix sense) of the iterate. For example, it allows to iterate in 2×2 matrices over the quadrature points of a strain field for a two-dimensional problem.

Subclassed by *muGrid::StaticFieldMap< T, Mutability, MapType, IterationType >*

Public Types

using **Scalar** = *T*

stored scalar type

using **Field_t** = std::conditional_t<*Mutability* == *Mapping*::*Const*, const TypedFieldBase<*T*>, TypedFieldBase<*T*>>

const-correct field depending on mapping mutability

using **PlainType** = *Eigen*::Matrix<*T*, *Eigen*::Dynamic, *Eigen*::Dynamic>

dynamically mapped eigen type

using **Return_t** = std::conditional_t<MutVal == *Mapping*::*Const*, *Eigen*::Map<const *PlainType*>, *Eigen*::Map<*PlainType*>>

return type for iterators over this- map

using **EigenRef** = *Eigen*::Ref<const *PlainType*>

Input type for matrix-like values (used for setting uniform values)

using **PixelEnumeration_t** = *akantu*::*containers*::*ZipContainer*<*FieldCollection*::*PixelIndexIterable*, *FieldMap*&>

zip-container for iterating over pixel index and stored value simultaneously

using **Enumeration_t** = *akantu*::*containers*::*ZipContainer*<*FieldCollection*::*IndexIterable*, *FieldMap*&>

zip-container for iterating over pixel or quadrature point index and stored value simultaneously

using **iterator** = Iterator<(*Mutability* == *Mapping*::*Mut*) ? *Mapping*::*Mut* : *Mapping*::*Const*>

stl

using **const_iterator** = Iterator<*Mapping*::*Const*>

stl

Public Functions

FieldMap() = delete

Default constructor.

explicit **FieldMap**(*Field_t* &field, *Iteration* iter_type = *Iteration*::*QuadPt*)

Constructor from a field. The default case is a map iterating over quadrature points with a matrix of shape (nb_components × 1) per field entry

FieldMap(*Field_t* &field, *Dim_t* nb_rows, *Iteration* iter_type = *Iteration*::*QuadPt*)

Constructor from a field with explicitly chosen shape of iterate. (the number of columns is inferred).

FieldMap(const *FieldMap* &other) = delete

Copy constructor.

FieldMap(*FieldMap* &&other)

Move constructor.

virtual **~FieldMap**() = default

Destructor.

FieldMap &**operator=**(const *FieldMap* &other) = delete

Copy assignment operator (delete because of reference member)

FieldMap &**operator=**(*FieldMap* &&other) = delete

Move assignment operator (delete because of reference member)

template<bool **IsMutableField** = *Mutability* == *Mapping*::*Mut*>
inline std::enable_if_t<*IsMutableField*, *FieldMap*> &**operator=**(const *EigenRef* &val)

Assign a matrix-like value to every entry.

template<bool **IsMutableField** = *Mutability* == *Mapping*::*Mut*>
inline std::enable_if_t<*IsMutableField*, *FieldMap*> &**operator=**(const *Scalar* &val)

Assign a scalar value to every entry.

iterator **begin**()

stl

iterator **end**()

stl

const_iterator **cbegin**()

stl

const_iterator **cend**()

stl

const_iterator **begin**() const

stl

const_iterator **end**() const

stl

size_t **size**() const

returns the number of iterates produced by this map (corresponds to the number of field entries if Iteration::Quadpt, or the number of pixels/voxels if Iteration::Pixel);

inline *Return_t*<*Mutability*> **operator[]**(size_t index)

random acces operator

inline *Return_t*<*Mapping*::*Const*> **operator[]**(size_t index) const

random const acces operator

void **set_data_ptr**()

query the size from the field's collection and set data_ptr

PixelEnumeration_t **enumerate_pixel_indices_fast**()

return an iterable proxy over pixel indices and stored values simultaneously. Throws a *muGrid::FieldMapError* if the iteration type is over quadrature points

Enumeration_t **enumerate_indices**()

return an iterable proxy over pixel/quadrature indices and stored values simultaneously

PlainType **mean**() const

evaluate and return the mean value of the map

Public Static Functions

static inline constexpr *Mapping* **FieldMutability**()

determine whether a field is mutably mapped at compile time

static inline constexpr bool **IsStatic**()

determine whether a field map is statically sized at compile time

Protected Attributes

const *Field_t* &**field**

mapped field. Needed for query at initialisations

const *Iteration* **iteration**

type of map iteration

const *Dim_t* **stride**

precomputed stride

const *Dim_t* **nb_rows**

number of rows of the iterate

const *Dim_t* **nb_cols**

number of columns fo the iterate

T ***data_ptr** = {nullptr}

Pointer to mapped data; is also unknown at construction and set in the map's begin function

bool **is_initialised** = {false}

keeps track of whether the map has been initialised.

std::shared_ptr<std::function<void()>> **callback** = {nullptr}

shared_ptr used for latent initialisation

class **FieldMapError** : public runtime_error

#include <field_map.hh> base class for field map-related exceptions

Public Functions

inline explicit **FieldMapError**(const std::string &what)

constructor

inline explicit **FieldMapError**(const char *what)

constructor

template<size_t **N**>

struct **Foreach**

#include <iterators.hh> static for loop

Public Static Functions

template<class **Tuple**>
static inline bool **not_equal**(*Tuple* &&a, *Tuple* &&b)

undocumented

template<>

struct **Foreach**<0>

#include <iterators.hh> static comparison

Public Static Functions

template<class **Tuple**>
static inline bool **not_equal**(*Tuple* &&a, *Tuple* &&b)

undocumented

class **FourierDerivative** : public *muFFT*::*DerivativeBase*

#include <derivative.hh> Representation of a derivative computed by Fourier interpolation

Public Types

using **Parent** = *DerivativeBase*

base class

using **Vector** = typename *Parent*::Vector

convenience alias

Public Functions

FourierDerivative() = delete

Default constructor.

explicit **FourierDerivative**(Dim_t spatial_dimension, Dim_t direction)

Constructor with raw *FourierDerivative* information.

FourierDerivative(const *FourierDerivative* &other) = default

Copy constructor.

FourierDerivative(*FourierDerivative* &&other) = default

Move constructor.

virtual **~FourierDerivative**() = default

Destructor.

FourierDerivative &**operator=**(const *FourierDerivative* &other) = delete

Copy assignment operator.

FourierDerivative &**operator=**(*FourierDerivative* &&other) = delete

Move assignment operator.

inline virtual Complex **fourier**(const *Vector* &phase) const

Return Fourier representation of the Fourier interpolated derivative. This here simply returns I*2*pi*phase. (I*2*pi*wavevector is the Fourier representation of the derivative.)

Protected Attributes

Dim_t **direction**

spatial direction in which to perform differentiation

template<typename **Rhs**, class **CellAdaptor**>

struct **generic_product_impl**<*CellAdaptor*, *Rhs*, SparseShape, DenseShape, GemvProduct> : public generic_product_impl_base<*CellAdaptor*, *Rhs*, generic_product_impl<*CellAdaptor*, *Rhs*>>

#include <cell_adaptor.hh> Implementation of *muSpectre::CellAdaptor* * `Eigen::DenseVector` through a specialization of `Eigen::internal::generic_product_impl`:

Public Types

typedef Product<*CellAdaptor*, *Rhs*>::Scalar **Scalar**

undocumented

Public Static Functions

template<typename **Dest**>
static inline void **scaleAndAddTo**(*Dest* &dst, const *CellAdaptor* &lhs, const *Rhs* &rhs, const *Scalar* &alpha)

undocumented

class **GlobalFieldCollection** : public *muGrid*::*FieldCollection*

#include <field_collection_global.hh> `muGrid::GlobalFieldCollection` derives from `muGrid::FieldCollection` and stores global fields that live throughout the whole computational domain, i.e. are defined for every pixel/voxel.

Public Types

using **Parent** = *FieldCollection*

alias of base class

using **DynamicPixels** = *CcoordOps*::*DynamicPixels*

pixel iterator

Public Functions

GlobalFieldCollection() = delete

Default constructor.

GlobalFieldCollection(*Dim_t* spatial_dimension, *Dim_t* nb_quad_pts)

Constructor

Parameters

- **spatial_dimension** – number of spatial dimensions, must be 1, 2, 3, or Unknown
- **nb_quad_pts** – number of quadrature points per pixel/voxel

GlobalFieldCollection(*Dim_t* spatial_dimension, *Dim_t* nb_quad_pts, const *DynCcoord_t* &nb_grid_pts, const *DynCcoord_t* &locations = {})

Constructor with initialization

Parameters

- **spatial_dimension** – number of spatial dimensions, must be 1, 2, 3, or Unknown
- **nb_quad_pts** – number of quadrature points per pixel/voxel

GlobalFieldCollection(const *GlobalFieldCollection* &other) = delete

Copy constructor.

GlobalFieldCollection(*GlobalFieldCollection* &&other) = default

Move constructor.

virtual **~GlobalFieldCollection**() = default

Destructor.

GlobalFieldCollection &**operator=**(const *GlobalFieldCollection* &other) = delete

Copy assignment operator.

GlobalFieldCollection &**operator=**(*GlobalFieldCollection* &&other) = delete

Move assignment operator.

const *DynamicPixels* &**get_pixels**() const

Return the pixels class that allows to iterator over pixels.

template<size_t **Dim**>
inline *Dim_t* **get_index**(const *Ccoord_t*<*Dim*> &ccoord) const

Return index for a ccoord.

inline *DynCcoord_t* **get_ccoord**(const *Dim_t* &index) const

return coordinates of the i-th pixel

void **initialise**(const *DynCcoord_t* &nb_grid_pts, const *DynCcoord_t* &locations = {})

freeze the problem size and allocate memory for all fields of the collection. Fields added later on will have their memory allocated upon construction.

template<size_t **Dim**>
inline void **initialise**(const *Ccoord_t*<*Dim*> &nb_grid_pts, const *Ccoord_t*<*Dim*> &locations = {})

freeze the problem size and allocate memory for all fields of the collection. Fields added later on will have their memory allocated upon construction.

void **initialise**(const *DynCcoord_t* &nb_grid_pts, const *DynCcoord_t* &locations, const *DynCcoord_t* &strides)

freeze the problem size and allocate memory for all fields of the collection. Fields added later on will have their memory allocated upon construction.

template<size_t **Dim**>
inline void **initialise**(const *Ccoord_t*<*Dim*> &nb_grid_pts, const *Ccoord_t*<*Dim*> &locations, const *Ccoord_t*<*Dim*> &strides)

freeze the problem size and allocate memory for all fields of the collection. Fields added later on will have their memory allocated upon construction.

GlobalFieldCollection **get_empty_clone**() const

obtain a new field collection with the same domain and pixels

Protected Attributes

DynamicPixels **pixels** = {}

helper to iterate over the grid

template<Dim_t **Dim**, class **Strain_t**, class **Tangent_t**>

struct **Hooke**

#include <materials_toolbox.hh> static inline implementation of *Hooke*'s law

Public Static Functions

static inline constexpr Real **compute_lambda**(const Real &young, const Real &poisson)

compute Lamé's first constant

Parameters

- **young** – Young's modulus
- **poisson** – Poisson's ratio

static inline constexpr Real **compute_mu**(const Real &young, const Real &poisson)

compute Lamé's second constant (i.e., shear modulus)

Parameters

- **young** – Young's modulus
- **poisson** – Poisson's ratio

static inline constexpr Real **compute_K**(const Real &young, const Real &poisson)

compute the bulk modulus

Parameters

- **young** – Young's modulus
- **poisson** – Poisson's ratio

static inline *Eigen*::TensorFixedSize<Real, *Eigen*::Sizes<*Dim*, *Dim*, *Dim*, *Dim*>> **compute_C**(const Real &lambda, const Real &mu)

compute the stiffness tensor

Parameters

- **lambda** – Lamé's first constant
- **mu** – Lamé's second constant (i.e., shear modulus)

static inline T4Mat<Real, *Dim*> **compute_C_T4**(const Real &lambda, const Real &mu)

compute the stiffness tensor

Parameters

- **lambda** – Lamé's first constant
- **mu** – Lamé's second constant (i.e., shear modulus)

template<class **s_t**>
static inline auto **evaluate_stress**(const Real &lambda, const Real &mu, *s_t* &&E) -> decltype(auto)

return stress

Parameters

- **lambda** – First Lamé's constant
- **mu** – Second Lamé's constant (i.e. shear modulus)
- **E** – Green-Lagrange or small strain tensor

template<class **T_t**, class **s_t**>

static inline auto **evaluate_stress**(const *T_t* C, *s_t* &&E) -> decltype(auto)

return stress

Parameters

- **C** – stiffness tensor (Piola-Kirchhoff 2 (or) w.r.t to E)
- **E** – Green-Lagrange or small strain tensor

template<class **s_t**>
static inline auto **evaluate_stress**(const Real &lambda, const Real &mu, *Tangent_t* &&C, *s_t* &&E) -> decltype(auto)

return stress and tangent stiffness

Parameters

- **lambda** – First Lamé's constant
- **mu** – Second Lamé's constant (i.e. shear modulus)
- **E** – Green-Lagrange or small strain tensor
- **C** – stiffness tensor (Piola-Kirchhoff 2 (or) w.r.t to E)

class **IncompletePixels**

Public Functions

explicit **IncompletePixels**(const *CellSplit* &cell)

constructor

IncompletePixels(const *IncompletePixels* &other) = default

copy constructor

IncompletePixels(*IncompletePixels* &other) = default

move constructor

virtual **~IncompletePixels**() = default

inline *iterator* **begin**() const

stl conformance

inline *iterator* **end**() const

stl conformance

inline size_t **size**() const

stl conformance

Protected Attributes

const *CellSplit* &**cell**

std::vector<Real> **incomplete_assigned_ratios**

std::vector<Dim_t> **index_incomplete_pixels**

class **IndexIterable**

#include <field_collection.hh> Iterate class for iterating over quadrature point indices of a field collection (i.e. the iterate you get when iterating over the result of *muGrid::FieldCollection::get_quad_pt_indices*).

Public Functions

IndexIterable() = delete

Default constructor.

IndexIterable(const *IndexIterable* &other) = delete

Copy constructor.

IndexIterable(*IndexIterable* &&other) = default

Move constructor.

virtual **~IndexIterable**() = default

Destructor.

IndexIterable &**operator=**(const *IndexIterable* &other) = delete

Copy assignment operator.

IndexIterable &**operator=**(*IndexIterable* &&other) = delete

Move assignment operator.

iterator **begin**() const

stl

iterator **end**() const

stl

size_t **size**() const

stl

Protected Functions

inline *Dim_t* **get_stride**() const

evaluate and return the stride with with the fast index of the iterators over the indices of this collection rotate

IndexIterable(const *FieldCollection* &collection, const *Iteration* &iteration_type)

Constructor is protected, because no one ever need to construct this except the fieldcollection

Protected Attributes

friend FieldCollection

allow the field collection to create *muGrid::FieldCollection::IndexIterable*s

const *FieldCollection* &**collection**

reference back to the proxied collection

const *Iteration* **iteration_type**

whether to iterate over pixels or quadrature points

template<class **Derived**>

struct **is_fixed**

#include <eigen_tools.hh> Helper class to check whether an `Eigen::Array` or `Eigen::Matrix` is statically sized

Public Types

using **T** = std::remove_cv_t<std::remove_reference_t<*Derived*>>

raw type for testing

Public Static Attributes

static constexpr bool **value** = {*T*::SizeAtCompileTime != *Eigen*::Dynamic}

evaluated test

template<class **TestClass**>

struct **is_matrix**

#include <eigen_tools.hh> Structure to determine whether an expression can be evaluated into a `Eigen::Matrix`, `Eigen::Array`, etc. and which helps determine compile-time size

Public Types

using **T** = std::remove_cv_t<std::remove_reference_t<*TestClass*>>

Public Static Attributes

static constexpr bool **value**{std::is_base_of<*Eigen*::MatrixBase<*T*>, *T*>::value}

template<class **Derived**>

struct **is_matrix**<*Eigen*::Map<*Derived*>>

Public Static Attributes

static constexpr bool **value** = {is_matrix<*Derived*>::value}

template<class **Derived**>

struct **is_matrix**<*Eigen*::Ref<*Derived*>>

Public Static Attributes

static constexpr bool **value** = {is_matrix<*Derived*>::value}

template<class **T**>

struct **is_reference_wrapper** : public false_type

template<class **U**>

struct **is_reference_wrapper**<std::reference_wrapper<*U*>> : public true_type

template<class **Derived**>

struct **is_square**

#include <eigen_tools.hh> Helper class to check whether an `Eigen::Array` or `Eigen::Matrix` is a static-size and square.

Public Types

using **T** = std::remove_cv_t<std::remove_reference_t<*Derived*>>

raw type for testing

Public Static Attributes

static constexpr bool **value**{(*T*::RowsAtCompileTime == *T*::ColsAtCompileTime) && *is_fixed*<*T*>::value}

true if the object is square and statically sized

template<class **T**, *Dim_t* **order**>

struct **is_tensor**

#include <tensor_algebra.hh> Check whether a given expression represents a Tensor specified order.

Public Static Attributes

static constexpr bool **value** = (std::is_convertible<*T*, *Eigen*::Tensor<*Real*, *order*>>::value || std::is_convertible<*T*, *Eigen*::Tensor<*Int*, *order*>>::value || std::is_convertible<*T*, *Eigen*::Tensor<*Complex*, *order*>>::value)

evaluated test

template<class **Strains_t**, class **Stresses_t**, *SplitCell* **is_cell_split** = *SplitCell*::*no*>

class **iterable_proxy**

#include <iterable_proxy.hh> this iterator class is a default for simple laws that just take a strain

Public Types

using **Strain_t** = typename *internal*::*StrainsTComputer*<*Strains_t*>::type

expected type for strain values

using **Stress_t** = typename *internal*::*StressesTComputer*<*Stresses_t*>::type

expected type for stress values

using **StrainFieldTup** = std::conditional_t<(std::tuple_size<*Strains_t*>::value == 2), std::tuple<const *muGrid*::*RealField*&, const *muGrid*::*RealField*&>, std::tuple<const *muGrid*::*RealField*&>>

tuple containing a strain and possibly a strain-rate field

using **StressFieldTup** = std::conditional_t<(std::tuple_size<*Stresses_t*>::value == 2), std::tuple<*muGrid*::*RealField*&, *muGrid*::*RealField*&>, std::tuple<*muGrid*::*RealField*&>>

tuple containing a stress and possibly a tangent stiffness field

Public Functions

iterable_proxy() = delete

Default constructor.

template<bool **DoNeedTgt** = std::tuple_size<*Stresses_t*>::value == 2, bool **DoNeedRate** = std::tuple_size<*Strain_t*>::value == 2>
inline **iterable_proxy**(*MaterialBase* &mat, const *muGrid*::*RealField* &F, std::enable_if_t<*DoNeedRate*, const *muGrid*::*RealField*> &F_rate, *muGrid*::*RealField* &P, std::enable_if_t<*DoNeedTgt*, *muGrid*::*RealField*> &K)

Iterator uses the material's internal variables field collection to iterate selectively over the global fields (such as the transformation gradient F and first Piola-Kirchhoff stress P.

template<bool **DontNeedTgt** = std::tuple_size<*Stresses_t*>::value == 1, bool **DoNeedRate** = std::tuple_size<*Strain_t*>::value == 2>
inline **iterable_proxy**(*MaterialBase* &mat, const *muGrid*::*RealField* &F, std::enable_if_t<*DoNeedRate*, const *muGrid*::*RealField*> &F_rate, std::enable_if_t<*DontNeedTgt*, *muGrid*::*RealField*> &P)

template<bool **DoNeedTgt** = std::tuple_size<*Stresses_t*>::value == 2, bool **DontNeedRate** = std::tuple_size<*Strain_t*>::value == 1>
inline **iterable_proxy**(*MaterialBase* &mat, std::enable_if_t<*DontNeedRate*, const *muGrid*::*RealField*> &F, *muGrid*::*RealField* &P, std::enable_if_t<*DoNeedTgt*, *muGrid*::*RealField*> &K)

template<bool **DontNeedTgt** = std::tuple_size<*Stresses_t*>::value == 1, bool **DontNeedRate** = std::tuple_size<*Strain_t*>::value == 1>
inline **iterable_proxy**(*MaterialBase* &mat, std::enable_if_t<*DontNeedRate*, const *muGrid*::*RealField*> &F, std::enable_if_t<*DontNeedTgt*, *muGrid*::*RealField*> &P)

iterable_proxy(const *iterable_proxy* &other) = default

Copy constructor.

iterable_proxy(*iterable_proxy* &&other) = default

Move constructor.

virtual **~iterable_proxy**() = default

Destructor.

iterable_proxy &**operator=**(const *iterable_proxy* &other) = default

Copy assignment operator.

iterable_proxy &**operator=**(*iterable_proxy* &&other) = default

Move assignment operator.

inline iterator **begin**()

returns iterator to first pixel if this material

inline iterator **end**()

returns iterator past the last pixel in this material

Protected Attributes

MaterialBase &**material**

reference to the proxied material

StrainFieldTup **strain_field**

cell's global strain field

StressFieldTup **stress_tup**

references to the global stress field and perhaps tangent

class **iterator**

#include <iterable_proxy.hh> dereferences into a tuple containing strains, and internal variables, as well as maps to the stress and potentially stiffness maps where to write the response of a pixel

Public Types

using **value_type** = std::tuple<Strain_t, Stress_t, const size_t&, Real>

return type contains a tuple of strain and possibly strain rate, stress and possibly stiffness, and a refererence to the pixel index

using **iterator_category** = std::forward_iterator_tag

stl conformance

Public Functions

iterator() = delete

Default constructor.

inline explicit **iterator**(const iterable_proxy &proxy, bool begin = true)

Iterator uses the material's internal variables field collection to iterate selectively over the global fields (such as the transformation gradient F and first Piola-Kirchhoff stress P.

iterator(const *iterator* &other) = default

Copy constructor.

iterator(*iterator* &&other) = default

Move constructor.

virtual **~iterator**() = default

Destructor.

iterator &**operator=**(const *iterator* &other) = default

Copy assignment operator.

iterator &**operator=**(*iterator* &&other) = default

Move assignment operator.

inline *iterator* &**operator++**()

pre-increment

inline *value_type* **operator***()

dereference

inline bool **operator!=**(const *iterator* &other) const

inequality

Protected Attributes

const iterable_proxy &**proxy**

ref to the proxy

Strains_t **strain_map**

map onto the global strain field

Stresses_t **stress_map**

map onto the global stress field and possibly tangent stiffness

size_t **index**

counter of current iterate (quad point). This value is the look-up index for the local field collection

muGrid::*FieldCollection*::*IndexIterable*::*iterator* **quad_pt_iter**

iterator over quadrature point. This value is the look-up index for the global field collection

class **iterator**

#include <field_collection.hh> iterator class for iterating over quadrature point indices or pixel indices of a *muGrid::FieldCollection::IndexIterable*. Dereferences to an index.

Public Types

using **PixelIndexIterator_t** = typename std::vector<size_t>::const_iterator

convenience alias

Public Functions

iterator() = delete

Default constructor.

iterator(const *PixelIndexIterator_t* &pixel_index_iterator, const size_t &stride)

constructor

iterator(const *iterator* &other) = default

Copy constructor.

iterator(*iterator* &&other) = default

Move constructor.

~iterator() = default

Destructor.

iterator &**operator=**(const *iterator* &other) = default

Copy assignment operator.

iterator &**operator=**(*iterator* &&other) = default

Move assignment operator.

inline *iterator* &**operator++**()

pre-increment

inline bool **operator!=**(const *iterator* &other) const

comparison

inline bool **operator==**(const *iterator* &other) const

comparison (required by akantu::iterators)

inline size_t **operator***()

dereference

Protected Attributes

size_t **stride**

stride for the slow moving index

size_t **offset** = {}

fast-moving index

PixelIndexIterator_t **pixel_index_iterator**

iterator of slow moving index

template<*Mapping* **MutIter**>

class **Iterator**

#include <field_map.hh> forward-declaration for `mugrid::FieldMap`'s iterator

Public Types

using **FieldMap_t** = std::conditional_t<*MutIter* == *Mapping*::*Const*, const FieldMap, FieldMap>

convenience alias

using **value_type** = typename FieldMap<T, Mutability>::template Return_t<*MutIter*>

stl

using **cvalue_type** = typename FieldMap<T, Mutability>::template Return_t<*Mapping*::*Const*>

stl

Public Functions

Iterator() = delete

Default constructor.

inline **Iterator**(*FieldMap_t* &map, bool end)

Constructor to beginning, or to end.

Iterator(const *Iterator* &other) = delete

Copy constructor.

Iterator(*Iterator* &&other) = default

Move constructor.

virtual **~Iterator**() = default

Destructor.

Iterator &**operator=**(const *Iterator* &other) = default

Copy assignment operator.

Iterator &**operator=**(*Iterator* &&other) = default

Move assignment operator.

inline *Iterator* &**operator++**()

pre-increment

inline *value_type* **operator***()

dereference

inline *cvalue_type* **operator***() const

dereference

inline bool **operator==**(const *Iterator* &other) const

equality

inline bool **operator!=**(const *Iterator* &other) const

inequality

Protected Attributes

FieldMap_t &**map**
: *FieldMap* being iterated over.

size_t **index**
: current iteration index

template<*Mapping* **MutIter**>

class **Iterator**

#include <field_map_static.hh> *Iterator* class for *muGrid::StaticFieldMap*

Public Types

using **value_type** = typename MapType::template value_type<*MutIter*>
: type returned by iterator

using **storage_type** = typename MapType::template storage_type<*MutIter*>
: type stored

Public Functions

Iterator() = delete
: Default constructor.

inline **Iterator**(const StaticFieldMap &map, bool end)
: Constructor to beginning, or to end.

Iterator(const *Iterator* &other) = default
: Copy constructor.

Iterator(*Iterator* &&other) = default
: Move constructor.

virtual **~Iterator**() = default
: Destructor.

Iterator &**operator=**(const *Iterator* &other) = default
: Copy assignment operator.

Iterator &**operator=**(*Iterator* &&other) = default
: Move assignment operator.

inline *Iterator* &**operator++**()
: pre-increment

inline *value_type* &**operator***()
: dereference

inline *value_type* ***operator->**()
: pointer to member

inline bool **operator==**(const *Iterator* &other) const
equality

inline bool **operator!=**(const *Iterator* &other) const
inequality

Protected Attributes

const StaticFieldMap &**map**
FieldMap being iterated over.

size_t **index**
current iteration index

storage_type **iterate**
map which is being returned per iterate

class **iterator**
#include <ccoord_operations.hh> Iterator class for `muSpectre::DynamicPixels`

Subclassed by *muGrid::CcoordOps::DynamicPixels::Enumerator::iterator*

Public Types

using **value_type** = *DynCcoord*<*threeD*>
stl

using **const_value_type** = const *value_type*
stl conformance

using **pointer** = *value_type**
stl conformance

using **difference_type** = std::ptrdiff_t
stl conformance

using **iterator_category** = std::forward_iterator_tag
stl conformance

Public Functions

inline **iterator**(const *DynamicPixels* &pixels, size_t index)
constructor

iterator() = delete
Default constructor.

iterator(const *iterator* &other) = default
Copy constructor.

iterator(*iterator* &&other) = default
Move constructor.

~iterator() = default
Destructor.

iterator &**operator=**(const *iterator* &other) = delete
Copy assignment operator.

iterator &**operator=**(*iterator* &&other) = delete
Move assignment operator.

inline *value_type* **operator***() const
dereferencing

inline *iterator* &**operator++**()
pre-increment

inline bool **operator!=**(const *iterator* &other) const
inequality

inline bool **operator==**(const *iterator* &other) const
equality

Protected Attributes

const *DynamicPixels* &**pixels**
ref to pixels in cell

size_t **index**
index of currently pointed-to pixel

class **iterator** : public *muGrid*::*CcoordOps*::*DynamicPixels*::*iterator*

Public Types

using **Parent** = *DynamicPixels*::*iterator*

Public Functions

inline std::tuple<*Dim_t*, *Parent*::value_type> **operator*()** const

class **iterator**

#include <ccoord_operations.hh> iterators over *Pixels* dereferences to cell coordinates

Public Types

using **value_type** = Ccoord

stl conformance

using **const_value_type** = const *value_type*

stl conformance

using **pointer** = *value_type**

stl conformance

using **difference_type** = std::ptrdiff_t

stl conformance

using **iterator_category** = std::forward_iterator_tag

stl conformance

using **reference** = *value_type*

stl conformance

Public Functions

explicit **iterator**(const Pixels &pixels, bool begin = true)

constructor

virtual **~iterator**() = default

inline *value_type* **operator*()** const

dereferencing

inline *iterator* &**operator++**()

pre-increment

inline bool **operator!=**(const *iterator* &other) const

inequality

inline bool **operator==**(const *iterator* &other) const
equality

Protected Attributes

const Pixels &**pixels**
ref to pixels in cell

size_t **index**
index of currently pointed-to pixel

class **iterator** : public std::vector::iterator<T*>
#include <ref_vector.hh> iterator over *muGrid::RefVector*

Public Functions

inline **iterator**(*Parent* &iterator)
copy constructor

inline **iterator**(*Parent* &&iterator)
move constructor

inline T &**operator***()
dereference

Private Types

using **Parent** = typename std::vector<T*>::iterator

class **iterator**

Public Types

using **value_type** = *Eigen*::Map<*Vector_t*>

using **value_type_const** = *Eigen*::Map<const *Vector_t*>

Public Functions

inline explicit **iterator**(const *Vectors_t* &data, const Dim_t &dim, bool begin = true)

constructor

virtual **~iterator**() = default

inline *value_type_const* **operator***() const

dereferencing

inline *iterator* &**operator++**()

pre-increment

inline *iterator* &**operator--**()

inline bool **operator!=**(const *iterator* &other)

inequality

inline bool **operator==**(const *iterator* &other) const

equality

Protected Attributes

const *Vectors_t* &**vectors**

Dim_t **dim**

size_t **index**

template<*Mapping* **MutIter**>

class **Iterator**

#include <state_field_map.hh> iterator type

Iterator class for *muGrid::StateFieldMap*

Public Types

using **StateFieldMap_t** = std::conditional_t<*MutIter* == *Mapping*::*Const*, const StateFieldMap, StateFieldMap>

convenience alias

using **StateWrapper_t** = typename StateFieldMap::template *StateWrapper*<*MutIter*>

const-correct proxy for iterates

Public Functions

Iterator() = delete

Deleted default constructor.

Iterator(*StateFieldMap_t* &state_field_map, size_t index)

constructor (should never have to be called by the user)

Iterator(const *Iterator* &other) = delete

Copy constructor.

Iterator(*Iterator* &&other) = default

Move constructor.

virtual **~Iterator**() = default

destructor

Iterator &**operator=**(const *Iterator* &other) = delete

Copy assignment operator.

Iterator &**operator=**(*Iterator* &&other) = default

Move assignment operator.

inline bool **operator!=**(const *Iterator* &other)

comparison

inline *Iterator* &**operator++**()

pre-increment

inline *StateWrapper_t* **operator***()

dereference

Protected Attributes

StateFieldMap_t &**state_field_map**

reference back to the iterated map

size_t **index**

current iteration progress

template<*Mapping* **MutIter**>

class **Iterator**

#include <state_field_map_static.hh> froward declaration of iterator class

Public Types

using **StaticStateFieldMap_t** = std::conditional_t<*MutIter* == *Mapping*::*Const*, const StaticStateFieldMap, StaticStateFieldMap>

const correct iterated map

using **StateWrapper_t** = typename StaticStateFieldMap::template *StaticStateWrapper*<*MutIter*>

convenience alias to dererencing return type

Public Functions

Iterator() = delete

Default constructor.

Iterator(const *Iterator* &other) = delete

Copy constructor.

inline **Iterator**(*StaticStateFieldMap_t* &state_field_map, size_t index)

constructor with field map and index, not for user to call

Iterator(*Iterator* &&other) = default

Move constructor.

virtual **~Iterator**() = default

Destructor.

Iterator &**operator=**(const *Iterator* &other) = delete

Copy assignment operator.

Iterator &**operator=**(*Iterator* &&other) = default

Move assignment operator.

inline bool **operator!=**(const *Iterator* &other) const

comparison

inline bool **operator==**(const *Iterator* &other) const

comparison (needed by akantu::iterator

inline *Iterator* &**operator++**()

pre-increment

inline *StateWrapper_t* **operator***()

dereference

Protected Attributes

StaticStateFieldMap_t &**state_field_map**

reference bap to iterated map

size_t **index**

current progress in iteration

class **iterator**

#include <cell_split.hh> iterator type over all incompletetedly assigned pixel's

Public Types

using **value_type** = std::tuple<*DynCcoord_t*, Real>

stl conformance

Public Functions

iterator(const *IncompletePixels* &pixels, Dim_t dim, bool begin = true)

constructor

virtual **~iterator**() = default

value_type **operator***() const

dereferencing

template<Dim_t **DimS**>
value_type **deref_helper**() const

iterator &**operator++**()

pre-increment

bool **operator!=**(const *iterator* &other)

inequality

inline bool **operator==**(const *iterator* &other) const

equality

template<Dim_t **DimS**>
auto **deref_helper**() const -> *value_type*

Protected Attributes

const *IncompletePixels* &**incomplete_pixels**

Dim_t **dim**

size_t **index**

template<Dim_t **Dim**, *StressMeasure* **StressM**, *StrainMeasure* **StrainM**>

struct **Kirchhoff_stress**

#include <stress_transformations_default_case.hh> Structure for functions returning Kirchhoff stress from other stress measures

Public Static Functions

template<class **Strain_t**, class **Stress_t**>
static inline decltype(auto) **compute**(*Strain_t*&&, *Stress_t*&&)

returns the converted stress

template<class **Strain_t**, class **Stress_t**, class **Tangent_t**>
static inline decltype(auto) **compute**(*Strain_t*&&, *Stress_t*&&, *Tangent_t*&&)

returns the converted stress and stiffness

template<Dim_t **Dim**, *StrainMeasure* **StrainM**>

struct **Kirchhoff_stress**<*Dim*, *StressMeasure*::*PK2*, *StrainM*> : public *muSpectre*::*MatTB*::*internal*::*Kirchhoff_stress*<*Dim*, *StressMeasure*::*no_stress_*, *StrainMeasure*::*no_strain_*>

#include <stress_transformations_PK2_impl.hh> Specialisation for the case where we get material stress (Piola-Kirchhoff-2, PK2) and we need to have Kirchhoff stress ()

Public Static Functions

template<class **Strain_t**, class **Stress_t**>
static inline decltype(auto) **compute**(*Strain_t* &&F, *Stress_t* &&S)

returns the converted stress

template<Dim_t **Dim**>

class **LamCombination**

Public Types

using **Stiffness_t** = typename *LamHomogen*<*Dim*, *Formulation*::*small_strain*>::Stiffness_t

using **Stress_t** = typename *LamHomogen*<*Dim*, *Formulation*::*small_strain*>::Stress_t

Public Functions

template<>
auto **lam_C_combine**(const *Eigen*::MatrixBase<Derived1> &C_1, const *Eigen*::MatrixBase<Derived2> &C_2, const Real &ratio) -> *Stiffness_t*

template<>
auto **lam_C_combine**(const *Eigen*::MatrixBase<Derived1> &C_1, const *Eigen*::MatrixBase<Derived2> &C_2, const Real &ratio) -> *Stiffness_t*

template<class **Derived1**, class **Derived2**>
auto **lam_S_combine**(const *Eigen*::MatrixBase<*Derived1*> &S_1, const *Eigen*::MatrixBase<*Derived2*> &S_2, const Real &ratio) -> *Stress_t*

Public Static Functions

template<class **Derived1**, class **Derived2**>
static inline *Stress_t* **lam_S_combine**(const *Eigen*::MatrixBase<*Derived1*> &S_1, const *Eigen*::MatrixBase<*Derived2*> &S_2, const Real &ratio)

This functions calculate the resultant stress and tangent matrices according to the computed E_1 and E_2 from the solver.

template<class **Derived1**, class **Derived2**>
static *Stiffness_t* **lam_C_combine**(const *Eigen*::MatrixBase<*Derived1*> &C_1, const *Eigen*::MatrixBase<*Derived2*> &C_2, const Real &ratio)

template<Dim_t **Dim**, *Formulation* **Form**>

class **LamHomogen**

Public Types

using **Vec_t** = *Eigen*::Matrix<Real, *Dim*, 1>

typedefs for data handled by this interface

using **Stiffness_t** = *muGrid*::*T4Mat*<Real, *Dim*>

using **Strain_t** = *Eigen*::Matrix<Real, *Dim*, *Dim*>

using **Stress_t** = *Strain_t*

using **Equation_index_t** = std::array<std::array<Dim_t, 2>, *Dim*>

using **Equation_stiffness_t** = *Eigen*::Matrix<Real, *Dim*, *Dim*>

using **Equation_strain_t** = *Eigen*::Matrix<Real, *Dim*, 1>

using **Equation_stress_t** = *Equation_strain_t*

using **Parallel_index_t** = std::conditional_t<*Form* == *Formulation*::*finite_strain*, std::array<std::array<Dim_t, 2>, *Dim* * (*Dim* - 1)>, std::array<std::array<Dim_t, 2>, (*Dim* - 1) * (*Dim* - 1)>>

using **Parallel_strain_t** = std::conditional_t<*Form* == *Formulation*::*finite_strain*, *Eigen*::Matrix<Real, *Dim* * (*Dim* - 1), 1>, *Eigen*::Matrix<Real, (*Dim* - 1) * (*Dim* - 1), 1>>

using **Parallel_stress_t** = *Parallel_strain_t*

using **Function_t** = std::function<std::tuple<*Stress_t*, *Stiffness_t*>(const *Eigen*::Ref<const *Strain_t*>&)>

Public Functions

template<class **Derived1**, class **Derived2**>
auto **make_total_strain**(const *Eigen*::MatrixBase<*Derived1*> &E_eq, const *Eigen*::MatrixBase<*Derived2*> &E_para) -> *Strain_t*

template<class **Derived**>
auto **get_equation_stiffness**(const *Eigen*::MatrixBase<*Derived*> &C) -> *Equation_stiffness_t*

template<class **Derived1**, class **Derived2**>
auto **delta_equation_stress_stiffness_eval**(const *Function_t* &mat_1_stress_eval, const *Function_t* &mat_2_stress_eval, const *Eigen*::MatrixBase<*Derived1*> &strain_1, const *Eigen*::MatrixBase<*Derived2*> &strain_2, const *RotatorNormal*<*Dim*> &rotator, const Real &ratio) -> std::tuple<*Equation_stress_t*, *Equation_stiffness_t*, Real>

template<class **Derived1**, class **Derived2**>
auto **delta_equation_stress_stiffness_eval_strain_1**(const *Function_t* &mat_1_stress_eval, const *Function_t* &mat_2_stress_eval, const *Eigen*::MatrixBase<*Derived1*> &strain_0_rot, const *Eigen*::MatrixBase<*Derived2*> &strain_1_rot, const *RotatorNormal*<*Dim*> &rotator, const Real &ratio) -> std::tuple<*Equation_stress_t*, *Equation_stiffness_t*, Real>

template<class **Derived1**, class **Derived2**>
auto **lam_stress_combine**(const *Eigen*::MatrixBase<*Derived1*> &stress_1, const *Eigen*::MatrixBase<*Derived2*> &stress_2, const Real &ratio) -> *Stress_t*

These functions are used as intrface for combination functions, They are also used for carrying out the stress transformation necessary for combining stifness matrix in Finite-Strain formulation because the combining formula from the bokk “Theory of Composites” written by “Graeme

Miltonare” for symmetric stifness matrices such as C and we have to transform stress to PK2 in order to be able to use it

template<>
constexpr auto **get_equation_indices**() -> *Equation_index_t*

template<>
constexpr auto **get_equation_indices**() -> *Equation_index_t*

template<>
constexpr auto **get_equation_indices**() -> *Equation_index_t*

template<>
constexpr auto **get_equation_indices**() -> *Equation_index_t*

template<>
constexpr auto **get_parallel_indices**() -> *Parallel_index_t*

template<>
constexpr auto **get_parallel_indices**() -> *Parallel_index_t*

template<>

constexpr auto **get_parallel_indices**() -> *Parallel_index_t*

template<>
constexpr auto **get_parallel_indices**() -> *Parallel_index_t*

template<class **Derived**>
auto **get_equation_stress**(const *Eigen*::MatrixBase<*Derived*> &S_total) -> *Equation_stress_t*

template<class **Derived**>
auto **get_parallel_stress**(const *Eigen*::MatrixBase<*Derived*> &S_total) -> *Parallel_stress_t*

template<class **Derived**>
auto **get_parallel_strain**(const *Eigen*::MatrixBase<*Derived*> &E_total) -> *Parallel_strain_t*

template<class **Derived**>
auto **get_equation_strain**(const *Eigen*::MatrixBase<*Derived*> &E_total) -> *Equation_strain_t*

template<class **Derived1**, class **Derived2**>
auto **linear_eqs**(const Real &ratio, const *Eigen*::MatrixBase<*Derived1*> &E_0_eq, const *Eigen*::MatrixBase<*Derived2*> &E_1_eq) -> *Equation_strain_t*

template<class **Derived1**, class **Derived2**>
auto **make_total_stress**(const *Eigen*::MatrixBase<*Derived1*> &S_eq, const *Eigen*::MatrixBase<*Derived2*> &S_para) -> *Stress_t*

Public Static Functions

static inline constexpr *Parallel_index_t* **get_parallel_indices**()

static inline constexpr *Equation_index_t* **get_equation_indices**()

template<class **Derived**>
static inline *Equation_strain_t* **get_equation_strain**(const *Eigen*::MatrixBase<*Derived*> &E_total)

template<class **Derived**>
static inline *Equation_stress_t* **get_equation_stress**(const *Eigen*::MatrixBase<*Derived*> &S_total)

template<class **Derived**>
static *Equation_stiffness_t* **get_equation_stiffness**(const *Eigen*::MatrixBase<*Derived*> &C)

template<class **Derived1**>
static inline *Parallel_strain_t* **get_parallel_strain**(const *Eigen*::MatrixBase<*Derived1*> &E)

template<class **Derived1**>
static inline *Parallel_stress_t* **get_parallel_stress**(const *Eigen*::MatrixBase<*Derived1*> &S)

template<class **Derived1**, class **Derived2**>
static *Strain_t* **make_total_strain**(const *Eigen*::MatrixBase<*Derived1*> &E_eq, const *Eigen*::MatrixBase<*Derived2*> &E_para)

template<class **Derived1**, class **Derived2**>
static inline *Stress_t* **make_total_stress**(const *Eigen*::MatrixBase<*Derived1*> &S_eq, const *Eigen*::MatrixBase<*Derived2*> &S_para)

template<class **Derived1**, class **Derived2**>

static inline *Equation_strain_t* **linear_eqs**(const Real &ratio, const *Eigen*::MatrixBase<*Derived1*> &E_0, const *Eigen*::MatrixBase<*Derived2*> &E_1)

template<class **Derived1**, class **Derived2**>
static std::tuple<*Equation_stress_t*, *Equation_stiffness_t*, Real> **delta_equation_stress_stiffness_eval**(const *Function_t* &mat_1_stress_e, const *Function_t* &mat_2_stress_e, const *Eigen*::MatrixBa &E_1, const *Eigen*::MatrixBa &E_2, const *RotatorNormal*<*Dim*> &rotator, const Real &ratio)

the objective in homogenisation of a single laminate pixel is equating the stress in the serial directions so the difference of stress between their layers should tend to zero. this function return the stress difference and the difference of Stiffness matrices which is used as the Jacobian in the solution process

template<class **Derived1**, class **Derived2**>

static std::tuple<*Equation_stress_t*, *Equation_stiffness_t*, Real> **delta_equation_stress_stiffness_eval_strain_1**(const *Function_t* &mat_1_stress_eval, const *Function_t* &mat_1_stiffness_eval, const *Eigen*::Ref<*Strain_t*> &E_1, const *Eigen*::Ref<*Strain_t*> &E_2, const *RotatorNormal*<Dim_t> &rotator, const Real &ratio)

static inline Real **del_energy_eval**(const Real &del_E_norm, const Real &delta_S_norm)

the following functions claculate the energy computation error of the solution. it will be used in each step of the solution to determine the relevant difference that implementation of that step has had on convergence to the solution.

template<class **Derived1**, class **Derived2**>
static *Stress_t* **lam_stress_combine**(const *Eigen*::MatrixBase<*Derived1*> &stress_1, const *Eigen*::MatrixBase<*Derived2*> &stress_2, const Real &ratio)

These functions are used as intrface for combination functions, They are also used for carrying out the stress transformation necessary for combining stifness matrix in Finite-Strain formulation because the combining formula from the bokk "Theory of Composites" are for symmetric stifness matrices such as C and we have to transform stress to PK2 in order to be able to use it

static *Stiffness_t* **lam_stiffness_combine**(const *Eigen*::Ref<*Stiffness_t*> &stiffness_1, const *Eigen*::Ref<*Stiffness_t*> &stiffness_2, const Real &ratio, const *Eigen*::Ref<*Strain_t*> &F_1, const *Eigen*::Ref<*Stress_t*> &F_2, const *Eigen*::Ref<*Strain_t*> &P_1, const *Eigen*::Ref<*Stress_t*> &P_2, const *Eigen*::Ref<*Strain_t*> &F, const *Eigen*::Ref<*Stress_t*> &P)

static std::tuple<Dim_t, Real, *Strain_t*, *Strain_t*> **laminate_solver**(const *Eigen*::Ref<*Strain_t*> &strain_coord, const *Function_t* &mat_1_stress_eval, const *Function_t* &mat_2_stress_eval, const Real &ratio, const *Eigen*::Ref<*Vec_t*> &normal_vec, const Real tol = 1e-10, const Dim_t max_iter = 1000)

This is the main solver function that might be called staically from an external file. this will return the

resultant stress and stiffness tensor according to interanl "equilibrium" of the lamiante. The inputs are : 1- global Strain 2- stress calculation function of the layer 1 3- stress calculation function of the layer 2 4- the ratio of the first material in the laminate sturucture of the pixel 5- the normal vector of the interface of two layers 6- the tolerance error for the internal solution of the laminate pixel 7- the maximum iterations for the internal solution of the laminate pixel

static *Stress_t* **evaluate_stress**(const *Eigen*::Ref<*Strain_t*> &strain_coord, const *Function_t* &mat_1_stress_eval, const *Function_t* &mat_2_stress_eval, const Real &ratio, const *Eigen*::Ref<*Vec_t*> &normal_vec, const Real tol = 1e-10, const Dim_t max_iter = 1000)

static std::tuple<*Stress_t*, *Stiffness_t*> **evaluate_stress_tangent**(const *Eigen*::Ref<*Strain_t*> &strain_coord, const *Function_t* &mat_1_stress_eval, const *Function_t* &mat_2_stress_eval, const Real &ratio, const *Eigen*::Ref<*Vec_t*> &normal_vec, const Real tol = 1e-10, const Dim_t max_iter = 1000)

class **LocalFieldCollection** : public *muGrid*::*FieldCollection*

#include <field_collection_local.hh> *muGrid::LocalFieldCollection* derives from *muGrid::FieldCollection* and stores local fields, i.e. fields that are only defined for a subset of all pixels/voxels in the computational domain. The coordinates of these active pixels are explicitly stored by this field collection. *muGrid::LocalFieldCollection::add_pixel* allows to add individual pixels/voxels to the field collection.

Public Types

using **Parent** = *FieldCollection*

alias for base class

Public Functions

LocalFieldCollection() = delete

Default constructor.

LocalFieldCollection(*Dim_t* spatial_dimension, *Dim_t* nb_quad_pts)

Constructor

Parameters

- **spatial_dimension** – spatial dimension of the field (can be muGrid::Unknown, e.g., in the case of the local fields for storing internal material variables)
- **nb_quad_pts** – number of quadrature points per pixel/voxel

LocalFieldCollection(const *LocalFieldCollection* &other) = delete

Copy constructor.

LocalFieldCollection(*LocalFieldCollection* &&other) = default

Move constructor.

virtual **~LocalFieldCollection**() = default

Destructor.

LocalFieldCollection &**operator=**(const *LocalFieldCollection* &other) = delete

Copy assignment operator.

LocalFieldCollection &**operator=**(*LocalFieldCollection* &&other) = delete

Move assignment operator.

void **add_pixel**(const size_t &global_index)

Insert a new pixel/voxel into the collection.

Parameters

global_index – refers to the linear index this pixel has in the global field collection defining the problem space

void **initialise**()

Freeze the set of pixels this collection is responsible for and allocate memory for all fields of the collection. Fields added lateron will have their memory allocated upon construction

LocalFieldCollection **get_empty_clone**() const

obtain a new field collection with the same domain and pixels

inline std::map<size_t, size_t> &**get_global_to_local_index_map**()

Protected Attributes

std::map<size_t, size_t> **global_to_local_index_map** = {}

template<class **FieldMapType**>

class **MappedField**

#include <mapped_field.hh> MappedFields are a combination of a field and an associated map, and as such it does not introduce any new functionality that Fields and FieldMaps do not already possess. They provide a convenience structure for the default use case of internal variables, which are typically used only by a single material and always the same way.

Public Types

using **Scalar** = typename *FieldMapType*::Scalar

stored scalar type

using **Return_t** = typename *FieldMapType*::template Return_t<*FieldMapType*::FieldMutability()>

return type for iterators over this- map

using **iterator** = typename *FieldMapType*::iterator

iterator over this map

using **const_iterator** = typename *FieldMapType*::const_iterator

constant iterator over this map

Public Functions

MappedField() = delete

Default constructor.

template<bool **StaticConstructor** = *IsStatic*(), std::enable_if_t<*StaticConstructor*, int> = 0>
inline **MappedField**(const std::string &unique_name, *FieldCollection* &collection)

Constructor with name and collection for statically sized mapped fields

template<bool **StaticConstructor** = *IsStatic*(), std::enable_if_t<not *StaticConstructor*, int> = 0>
inline **MappedField**(const std::string &unique_name, const *Dim_t* &nb_rows, const *Dim_t* &nb_cols, const *Iteration* &iter_type, *FieldCollection* &collection)

Constructor for dynamically sized mapped field

Parameters

- **unique_name** – unique identifier for this field
- **nb_rows** – number of rows for the iterates
- **nb_cols** – number of columns for the iterates
- **iter_type** – whether to iterate over pixels or quadrature points
- **collection** – collection where the field is to be registered

MappedField(const *MappedField* &other) = delete

Copy constructor.

MappedField(*MappedField* &&other) = default

Move constructor.

virtual **~MappedField**() = default

Destructor.

MappedField &**operator=**(const *MappedField* &other) = delete

Copy assignment operator.

MappedField &**operator=**(*MappedField* &&other) = default

Move assignment operator.

inline *Return_t* **operator[]**(size_t index)

random access operator

inline *iterator* **begin**()

stl

inline *iterator* **end**()

stl

inline *const_iterator* **begin**() const

stl

inline *const_iterator* **end**() const

stl

inline *TypedField*<*Scalar*> &**get_field**()

return a reference to the mapped field

inline *FieldMapType* &**get_map**()

return a reference to the map

Public Static Functions

static inline constexpr bool **IsStatic**()

detemine at compile time whether the field map is statically sized

Protected Attributes

Dim_t **nb_components**

number of components stored per quadrature point

TypedField<*Scalar*> &**field**

reference to mapped field

FieldMapType **map**

associated field map

Protected Static Functions

template<bool **StaticConstructor** = *IsStatic*(), std::enable_if_t<not *StaticConstructor*, int> = 0>
static inline *Dim_t* **compute_nb_components_dynamic**(const *Dim_t* &nb_rows, const *Dim_t* &nb_cols, const *Iteration* &iter_type, const std::string &unique_name, *FieldCollection* &collection)

evaluate and return the number of components the dynamically mapped field needs to store per quadrature point

template<bool **StaticConstructor** = *IsStatic*(), std::enable_if_t<*StaticConstructor*, int> = 0>
static inline *Dim_t* **compute_nb_components_static**(const std::string &unique_name, *FieldCollection* &collection)

evaluate and return the number of components the statically mapped field needs to store per quadrature point

template<class **StateFieldMapType**>

class **MappedStateField**

#include <mapped_state_field.hh> MappedStateFields are a combination of a state field and an associated map, and as such it does not introduce any new functionality that StateFields and StateFieldMaps do not already possess. They provide a convenience structure for the default use case of internal variables, which are typically used only by a single material and always the same way.

Public Types

using **Scalar** = typename *StateFieldMapType*::Scalar

stored scalar type

using **Return_t** = typename *StateFieldMapType*::template StaticStateWrapper<*StateFieldMapType*::FieldMutability()>

return type for iterators over this- map

using **iterator** = typename *StateFieldMapType*::iterator

iterator over this map

using **const_iterator** = typename *StateFieldMapType*::const_iterator

constant iterator over this map

Public Functions

MappedStateField() = delete

Deleted default constructor.

inline **MappedStateField**(const std::string &unique_name, *FieldCollection* &collection)

Constructor with name and collection.

MappedStateField(const *MappedStateField* &other) = delete

Copy constructor.

MappedStateField(*MappedStateField* &&other) = default

Move constructor.

virtual ~**MappedStateField**() = default

Destructor.

MappedStateField &**operator=**(const *MappedStateField* &other) = delete

Copy assignment operator.

MappedStateField &**operator=**(*MappedStateField* &&other) = default

Move assignment operator.

inline *Return_t* **operator[]**(size_t index)

random access operator

inline *iterator* **begin**()

stl

inline *iterator* **end**()

stl

inline *const_iterator* **begin**() const

stl

inline *const_iterator* **end**() const

stl

inline *TypedStateField*<*Scalar*> &**get_state_field**()

return a reference to the mapped state field

inline *StateFieldMapType* &**get_map**()

return a reference to the map

Protected Attributes

Dim_t **nb_components**

number of components stored per quadrature point

TypedStateField<*Scalar*> &**state_field**

ref to mapped state field

StateFieldMapType **map**

associated field map

Protected Static Functions

static inline *Dim_t* **compute_nb_components**(const std::string &unique_prefix, *FieldCollection* &collection)

evaluate and return the number of components the statically mapped state field needs to store per quadrature point

class **MaterialBase**

#include <material_base.hh> base class for materials

Subclassed by *muSpectre::MaterialMuSpectre< MaterialHyperElastoPlastic1< DimM >, DimM >*, *muSpectre::MaterialMuSpectre< MaterialHyperElastoPlastic2< DimM >, DimM >*, *muSpectre::MaterialMuSpectre< MaterialLinearAnisotropic< DimM >, DimM >*, *muSpectre::MaterialMuSpectre< MaterialLinearElastic1< DimM >, DimM >*, *muSpectre::MaterialMuSpectre< MaterialLinearElastic2< DimM >, DimM >*, *muSpectre::MaterialMuSpectre< MaterialLinearElastic3< DimM >, DimM >*, *muSpectre::MaterialMuSpectre< MaterialLinearElastic4< DimM >, DimM >*, *muSpectre::MaterialMuSpectre< MaterialLinearElasticGeneric1< DimM >, DimM >*, *muSpectre::MaterialMuSpectre< MaterialLinearElasticGeneric2< DimM >, DimM >*, *muSpectre::MaterialMuSpectre< MaterialStochasticPlasticity< DimM >, DimM >*, *muSpectre::MaterialMuSpectre< STMaterialLinearElasticGeneric1< DimM, StrainM, StressM >, DimM >*, *muSpectre::MaterialLaminate< DimM >*, *muSpectre::MaterialMuSpectre< Material, DimM >*

Public Types

using **DynMatrix_t** = *Eigen*::Matrix<Real, *Eigen*::Dynamic, *Eigen*::Dynamic>

Public Functions

MaterialBase() = delete

Default constructor.

MaterialBase(const std::string &name, const Dim_t &spatial_dimension, const Dim_t &material_dimension, const Dim_t &nb_quad_pts)

Construct by name

Parameters

- **name** – of the material
- **spatial_dimension** – is the number of spatial dimension, i.e. the grid

- **material_dimension** – is the material dimension (i.e., the dimension of constitutive law; even for e.g. two-dimensional problems the constitutive law could live in three-dimensional space for e.g. plane strain or stress problems)
- **nb_quad_pts** – is the number of quadrature points per grid cell

MaterialBase(const *MaterialBase* &other) = delete

Copy constructor.

MaterialBase(*MaterialBase* &&other) = delete

Move constructor.

virtual **~MaterialBase**() = default

Destructor.

MaterialBase &**operator=**(const *MaterialBase* &other) = delete

Copy assignment operator.

MaterialBase &**operator=**(*MaterialBase* &&other) = delete

Move assignment operator.

virtual void **add_pixel**(const size_t &pixel_index)

take responsibility for a pixel identified by its cell coordinates WARNING: this won't work for materials with additional info per pixel (as, e.g. for eigenstrain), we need to pass more parameters. Materials of this type need to overload add_pixel

virtual void **add_pixel_split**(const size_t &pixel_index, const Real &ratio)

void **allocate_optional_fields**(*SplitCell* is_cell_split = *SplitCell*::*no*)

virtual void **initialise**()

allocate memory, etc, but also: wipe history variables!

inline virtual void **save_history_variables**()

for materials with state variables, these typically need to be saved/updated an the end of each load increment, the virtual base implementation does nothing, but materials with history variables need to implement this

const std::string &**get_name**() const

return the material's name

inline Dim_t **get_material_dimension**()

material dimension for inheritance

virtual void **compute_stresses**(const *muGrid*::*RealField* &F, *muGrid*::*RealField* &P, const *Formulation* &form, *SplitCell* is_cell_split = *SplitCell*::*no*) = 0

computes stress

void **compute_stresses**(const *muGrid*::*Field* &F, *muGrid*::*Field* &P, const *Formulation* &form, *SplitCell* is_cell_split = *SplitCell*::*no*)

Convenience function to compute stresses, mostly for debugging and testing. Has runtime-cost associated with compatibility-checking and conversion of the Field_t arguments that can be avoided by using the version with strongly typed field references

virtual void **compute_stresses_tangent**(const *muGrid*::*RealField* &F, *muGrid*::*RealField* &P, *muGrid*::*RealField* &K, const *Formulation* &form, *SplitCell* is_cell_split = *SplitCell*::*no*) = 0

computes stress and tangent moduli

void **compute_stresses_tangent**(const *muGrid*::*Field* &F, *muGrid*::*Field* &P, *muGrid*::*Field* &K, *Formulation* form, *SplitCell* is_cell_split = *SplitCell*::*no*)

Convenience function to compute stresses and tangent moduli, mostly for debugging and testing. Has runtime-cost associated with compatibility-checking and conversion of the Field_t arguments that can be avoided by using the version with strongly typed field references

Real **get_assigned_ratio**(const size_t &pixel_id)

void **get_assigned_ratios**(std::vector<Real> &pixel_assigned_ratios)

muGrid::*RealField* &**get_assigned_ratio_field**()

muGrid::*LocalFieldCollection*::PixelIndexIterable **get_pixel_indices**() const

return and iterable proxy over the indices of this material's pixels

muGrid::*LocalFieldCollection*::IndexIterable **get_quad_pt_indices**() const

return and iterable proxy over the indices of this material's quadrature points

inline Dim_t **size**() const

number of quadrature points assigned to this material

std::vector<std::string> **list_fields**() const

list the names of all internal fields

inline *muGrid*::*LocalFieldCollection* &**get_collection**()

gives access to internal fields

virtual std::tuple<*DynMatrix_t*, *DynMatrix_t*> **constitutive_law_dynamic**(const *Eigen*::Ref<const *DynMatrix_t*> &strain, const size_t &quad_pt_index, const *Formulation* &form) = 0

evaluates both second Piola-Kirchhoff stress and stiffness given the Green-Lagrange strain (or Cauchy stress and stiffness if called with a small strain tensor)

Protected Attributes

const std::string **name**

material's name (for output and debugging)

muGrid::*LocalFieldCollection* **internal_fields**

storage for internal variables

Dim_t **material_dimension**

spatial dimension of the material

field holding the assigned ratios of the material

std::unique_ptr<*muGrid*::*MappedScalarField*<Real, *muGrid*::*Mapping*::*Mut*>> **assigned_ratio** = {nullptr}

bool **is_initialised** = {false}

class **MaterialError** : public runtime_error

#include <material_base.hh> base class for material-related exceptions

Public Functions

inline explicit **MaterialError**(const std::string &what)

constructor

inline explicit **MaterialError**(const char *what)

constructor

template<Dim_t **DimM**>

class **MaterialEvaluator**

#include <material_evaluator.hh> Small convenience class providing a common interface to evaluate materials without the need to set up an entire homogenisation problem. Useful for debugging material laws.

Template Parameters

DimM – Dimensionality of the material

Public Types

using **T2_t** = *Eigen*::Matrix<Real, *DimM*, *DimM*>

shorthand for second-rank tensors

using **T4_t** = *muGrid*::*T4Mat*<Real, *DimM*>

shorthand for fourth-rank tensors

using **T2_map** = *Eigen*::Map<*T2_t*>

map of a second-rank tensor

using **T4_map** = *muGrid*::*T4MatMap*<Real, *DimM*>

map of a fourth-rank tensor

using **T2_const_map** = *Eigen*::Map<const *T2_t*>

const map of a second-rank tensor

using **T4_const_map** = *muGrid*::*T4MatMap*<Real, *DimM*, true>

const map of a fourth-rank tensor

using **FieldColl_t** = *muGrid*::*GlobalFieldCollection*

convenience alias

Public Functions

MaterialEvaluator() = delete

Default constructor.

inline explicit **MaterialEvaluator**(std::shared_ptr<*MaterialBase*> material)

constructor with a shared pointer to a Material

MaterialEvaluator(const *MaterialEvaluator* &other) = delete

Copy constructor.

MaterialEvaluator(*MaterialEvaluator* &&other) = default

Move constructor.

virtual **~MaterialEvaluator**() = default

Destructor.

MaterialEvaluator &**operator=**(const *MaterialEvaluator* &other) = delete

Copy assignment operator.

MaterialEvaluator &**operator=**(*MaterialEvaluator* &&other) = default

Move assignment operator.

inline void **save_history_variables**()

for materials with state variables. See *muSpectre::MaterialBase* for details

inline *T2_const_map* **evaluate_stress**(const *Eigen*::Ref<const *T2_t*> &grad, const *Formulation* &form)

Evaluates the underlying materials constitutive law and returns the stress P or as a function of the placement gradient F or small strain tensor depending on the formulation (`muSpectre::Formulation::small_strain` for (), `muSpectre::Formulation::finite_strain` for P(F))

inline std::tuple<*T2_const_map*, *T4_const_map*> **evaluate_stress_tangent**(const *Eigen*::Ref<const *T2_t*> &grad, const *Formulation* &form)

Evaluates the underlying materials constitutive law and returns the the stress P or and the tangent moduli K as a function of the placement gradient F or small strain tensor depending on the formulation (`muSpectre::Formulation::small_strain` for (), `muSpectre::Formulation::finite_strain` for P(F))

inline *T4_t* **estimate_tangent**(const *Eigen*::Ref<const *T2_t*> &grad, const *Formulation* &form, const Real step, const *FiniteDiff* diff_type = *FiniteDiff*::*centred*)

estimate the tangent using finite difference

inline void **initialise**()

initialise the material and the fields

Protected Functions

void **check_init**()

throws a runtime error if the material's per-pixel data has not been set.

Protected Attributes

std::shared_ptr<*MaterialBase*> **material**

storage of the material is managed through a shared pointer

std::unique_ptr<*FieldColl_t*> **collection**

storage of the strain, stress and tangent fields is managed through a unique pointer

muGrid::*MappedT2Field*<Real, Mapping::Mut, *DimM*> **strain**

strain field (independent variable)

muGrid::*MappedT2Field*<Real, Mapping::Mut, *DimM*> **stress**

stress field (result)

muGrid::*MappedT4Field*<Real, Mapping::Mut, *DimM*> **tangent**

field of tangent moduli (result)

bool **is_initialised** = {false}

whether the evaluator has been initialised

template<Dim_t **DimM**>

class **MaterialHyperElastoPlastic1** : public *muSpectre*::*MaterialMuSpectre*<*MaterialHyperElastoPlastic1*<*DimM*>, *DimM*>

#include <material_hyper_elasto_plastic1.hh> material implementation for hyper-elastoplastic constitutive law. Note for developpers: this law is tested against a reference python implementation in `py_comparison_test_material_hyper_elasto_plastic1.py`

Public Types

using **Parent** = *MaterialMuSpectre*<*MaterialHyperElastoPlastic1*<*DimM*>, *DimM*>

base class

using **T2_t** = *Eigen*::Matrix<Real, *DimM*, *DimM*>

short-hand for second-rank tensors

using **T4_t** = *muGrid*::*T4Mat*<Real, *DimM*>

short-hand for fourth-rank tensors

using **traits** = *MaterialMuSpectre_traits*<*MaterialHyperElastoPlastic1*>

shortcut to traits

using **Hooke** = typename *MatTB*::*Hooke*<*DimM*, typename *traits*::StrainMap_t::reference, typename *traits*::TangentMap_t::reference>

Hooke's law implementation.

using **T2StRef_t** = typename *muGrid*::*MappedT2StateField*<Real, Mapping::Mut, *DimM*>::Return_t

type in which the previous strain state is referenced

using **ScalarStRef_t** = typename *muGrid*::*MappedScalarStateField*<Real, Mapping::Mut>::Return_t

type in which the previous plastic flow is referenced

Public Functions

MaterialHyperElastoPlastic1() = delete

Default constructor.

MaterialHyperElastoPlastic1(const std::string &name, const Dim_t &spatial_dimension, const Dim_t &nb_quad_pts, const Real &young, const Real &poisson, const Real &tau_y0, const Real &H)

Constructor with name and material properties.

MaterialHyperElastoPlastic1(const *MaterialHyperElastoPlastic1* &other) = delete

Copy constructor.

MaterialHyperElastoPlastic1(*MaterialHyperElastoPlastic1* &&other) = delete

Move constructor.

virtual **~MaterialHyperElastoPlastic1**() = default

Destructor.

MaterialHyperElastoPlastic1 &**operator=**(const *MaterialHyperElastoPlastic1* &other) = delete

Copy assignment operator.

MaterialHyperElastoPlastic1 &**operator=**(*MaterialHyperElastoPlastic1* &&other) = delete

Move assignment operator.

T2_t **evaluate_stress**(const *T2_t* &F, *T2StRef_t* F_prev, *T2StRef_t* be_prev, *ScalarStRef_t* plast_flow)

evaluates Kirchhoff stress given the current placement gradient F, the previous Gradient F_1 and the cumulated plastic flow

inline *T2_t* **evaluate_stress**(const *T2_t* &F, const size_t &quad_pt_index)

evaluates Kirchhoff stress given the local placement gradient and pixel id.

std::tuple<*T2_t*, *T4_t*> **evaluate_stress_tangent**(const *T2_t* &F, *T2StRef_t* F_prev, *T2StRef_t* be_prev, *ScalarStRef_t* plast_flow)

evaluates Kirchhoff stress and tangent moduli given the current placement gradient F, the previous Gradient F_1 and the cumulated plastic flow

inline std::tuple<*T2_t*, *T4_t*> **evaluate_stress_tangent**(const *T2_t* &F, const size_t &quad_pt_index)

evaluates Kirchhoff stressstiffness and tangent moduli given the local placement gradient and pixel id.

virtual void **save_history_variables**() override

The statefields need to be cycled at the end of each load increment

virtual void **initialise**() final

set the previous gradients to identity

inline *muGrid*::*MappedScalarStateField*<Real, Mapping::Mut> &**get_plast_flow_field**()

getter for internal variable field

inline *muGrid*::*MappedT2StateField*<Real, Mapping::Mut, *DimM*> &**get_F_prev_field**()

getter for previous gradient field F

inline *muGrid*::*MappedT2StateField*<Real, Mapping::Mut, *DimM*> &**get_be_prev_field**()

getterfor elastic left Cauchy-Green deformation tensor b

Protected Types

using **Worker_t** = std::tuple<*T2_t*, Real, Real, *T2_t*, bool, *muGrid*::*Decomp_t*<*DimM*>>

result type of the stress calculation with intermediate results for tangent moduli calculation

Protected Functions

Worker_t **stress_n_internals_worker**(const *T2_t* &F, *T2StRef_t* &F_prev, *T2StRef_t* &be_prev, *ScalarStRef_t* &plast_flow)

worker function computing stresses and internal variables

Protected Attributes

muGrid::*MappedScalarStateField*<Real, Mapping::Mut> **plast_flow_field**

storage for cumulated plastic flow

muGrid::*MappedT2StateField*<Real, Mapping::Mut, *DimM*> **F_prev_field**

storage for previous gradient F

muGrid::*MappedT2StateField*<Real, Mapping::Mut, *DimM*> **be_prev_field**

storage for elastic left Cauchy-Green deformation tensor b

const Real **young**

Young's modulus.

const Real **poisson**

Poisson's ratio.

const Real **lambda**

first Lamé constant

const Real **mu**

second Lamé constant (shear modulus)

const Real **K**

Bulk modulus.

const Real **tau_y0**

initial yield stress

const Real **H**

hardening modulus

std::unique_ptr<const *muGrid*::*T4Mat*<Real, *DimM*>> **C_holder**

stiffness tensor

const *muGrid*::*T4Mat*<Real, *DimM*> &**C**

ref to elastic tensor

template<Dim_t **DimM**>

class **MaterialHyperElastoPlastic2** : public *muSpectre*::*MaterialMuSpectre*<*MaterialHyperElastoPlastic2*<*DimM*>, *DimM*>

#include <material_hyper_elasto_plastic2.hh> material implementation for hyper-elastoplastic constitutive law.

Public Types

using **Parent** = *MaterialMuSpectre*<*MaterialHyperElastoPlastic2*<*DimM*>, *DimM*>

base class

using **T2_t** = *Eigen*::Matrix<Real, *DimM*, *DimM*>

using **T4_t** = *muGrid*::*T4Mat*<Real, *DimM*>

using **traits** = *MaterialMuSpectre_traits*<*MaterialHyperElastoPlastic2*>

shortcut to traits

using **Field_t** = *muGrid*::*MappedScalarField*<Real, Mapping::Const>

storage type for scalar material constant fields

using **Hooke** = typename *MatTB*::*Hooke*<*DimM*, typename *traits*::StrainMap_t::reference, typename *traits*::TangentMap_t::reference>

Hooke's law implementation.

using **FlowField_t** = *muGrid*::*MappedScalarStateField*<Real, Mapping::Mut>

using **FlowField_ref** = typename *FlowField_t*::Return_t

using **PrevStrain_t** = *muGrid*::*MappedT2StateField*<Real, Mapping::Mut, *DimM*>

using **PrevStrain_ref** = typename *PrevStrain_t*::Return_t

Public Functions

MaterialHyperElastoPlastic2() = delete

Default constructor.

MaterialHyperElastoPlastic2(const std::string &name, const Dim_t &spatial_dimension, const Dim_t &nb_quad_pts)

Constructor with name.

MaterialHyperElastoPlastic2(const *MaterialHyperElastoPlastic2* &other) = delete

Copy constructor.

MaterialHyperElastoPlastic2(*MaterialHyperElastoPlastic2* &&other) = delete

Move constructor.

virtual **~MaterialHyperElastoPlastic2**() = default

Destructor.

MaterialHyperElastoPlastic2 &**operator=**(const *MaterialHyperElastoPlastic2* &other) = delete

Copy assignment operator.

MaterialHyperElastoPlastic2 &**operator=**(*MaterialHyperElastoPlastic2* &&other) = delete

Move assignment operator.

T2_t **evaluate_stress**(const *T2_t* &F, *PrevStrain_ref* F_prev, *PrevStrain_ref* be_prev, *FlowField_ref* plast_flow, const Real lambda, const Real mu, const Real tau_y0, const Real H)

evaluates Kirchhoff stress given the current placement gradient F, the previous Gradient F_1 and the cumulated plastic flow

inline *T2_t* **evaluate_stress**(const *T2_t* &F, const size_t &pixel_index)

evaluates Kirchhoff stress given the local placement gradient and pixel id.

std::tuple<*T2_t*, *T4_t*> **evaluate_stress_tangent**(const *T2_t* &F, *PrevStrain_ref* F_prev, *PrevStrain_ref* be_prev, *FlowField_ref* plast_flow, const Real lambda, const Real mu, const Real tau_y0, const Real H, const Real K)

evaluates Kirchhoff stress and tangent moduli given the current placement gradient F, the previous Gradient F_1 and the cumulated plastic flow

inline std::tuple<*T2_t*, *T4_t*> **evaluate_stress_tangent**(const *T2_t* &F, const size_t &pixel_index)

evaluates Kirchhoff stressstiffness and tangent moduli given the local placement gradient and pixel id.

virtual void **save_history_variables**() override

The statefields need to be cycled at the end of each load increment

virtual void **initialise**() final

set the previous gradients to identity

virtual void **add_pixel**(const size_t &pixel_id) final

overload add_pixel to write into loacal stiffness tensor

void **add_pixel**(const size_t &pixel_id, const Real &Youngs_modulus, const Real &Poisson_ratio, const Real &tau_y0, const Real &H)

overload add_pixel to write into local stiffness tensor

inline *muGrid*::*MappedScalarStateField*<Real, Mapping::Mut> &**get_plast_flow_field**()

getter for internal variable field

inline *muGrid*::*MappedT2StateField*<Real, Mapping::Mut, *DimM*> &**get_F_prev_field**()

getter for previous gradient field F

inline *muGrid*::*MappedT2StateField*<Real, Mapping::Mut, *DimM*> &**get_be_prev_field**()

getterfor elastic left Cauchy-Green deformation tensor b

Protected Types

using **Worker_t** = std::tuple<*T2_t*, Real, Real, *T2_t*, bool, *muGrid*::*Decomp_t*<*DimM*>>

worker function computing stresses and internal variables

Protected Functions

Worker_t **stress_n_internals_worker**(const *T2_t* &F, *PrevStrain_ref* &F_prev, *PrevStrain_ref* &be_prev, *FlowField_ref* &plast_flow, const Real lambda, const Real mu, const Real tau_y0, const Real H)

Protected Attributes

FlowField_t **plast_flow_field**

storage for cumulated plastic flow

PrevStrain_t **F_prev_field**

storage for previous gradient F

PrevStrain_t **be_prev_field**

storage for elastic left Cauchy-Green deformation tensor b

Field_t **lambda_field**

storage for first Lamé constant

Field_t **mu_field**

storage for second Lamé constant (shear modulus)

Field_t **tau_y0_field**

storage for initial yield stress

Field_t **H_field**

storage for hardening modulus

Field_t **K_field**

storage for Bulk modulus

template<Dim_t **DimM**>

class **MaterialLaminate** : public *muSpectre*::*MaterialBase*

Public Types

using **Parent** = *MaterialBase*

base class

using **RealField** = *muGrid*::*RealField*

using **DynMatrix_t** = *Parent*::DynMatrix_t

using **MatBase_t** = *MaterialBase*

using **MatPtr_t** = std::shared_ptr<*MatBase_t*>

using **T2_t** = *Eigen*::Matrix<Real, *DimM*, *DimM*>

using **T4_t** = *muGrid*::*T4Mat*<Real, *DimM*>

using **VectorField_t** = *muGrid*::*RealField*

using **MappedVectorField_t** = *muGrid*::*MappedT1Field*<Real, Mapping::Mut, *DimM*>

using **VectorFieldMap_t** = *muGrid*::*T1FieldMap*<Real, Mapping::Mut, *DimM*>

using **ScalarField_t** = *muGrid*::*RealField*

using **MappedScalarField_t** = *muGrid*::*MappedScalarField*<Real, Mapping::Mut>

using **ScalarFieldMap_t** = *muGrid*::*ScalarFieldMap*<Real, Mapping::Mut>

using **Strain_t** = *Eigen*::Matrix<Real, *DimM*, *DimM*>

using **Stress_t** = *Strain_t*

using **Stiffness_t** = *muGrid*::*T4Mat*<Real, *DimM*>

using **NeedTangent** = *MatTB*::*NeedTangent*

type used to determine whether the `muSpectre::MaterialMuSpectre::iterable_proxy` evaluate only stresses or also tangent stiffnesses

using **traits** = *MaterialMuSpectre_traits*<*MaterialLaminate*>

traits of this material

Public Functions

MaterialLaminate() = delete

Default constructor.

MaterialLaminate(const std::string &name, const Dim_t &spatial_dimension, const Dim_t &nb_quad_pts)

Constructor with name and material properties.

MaterialLaminate(const *MaterialLaminate* &other) = delete

Copy constructor.

MaterialLaminate(*MaterialLaminate* &&other) = delete

Move constructor.

virtual **~MaterialLaminate**() = default

Destructor.

template<typename **Derived**>
inline decltype(auto) **evaluate_stress**(const *Eigen*::MatrixBase<*Derived*> &E, const size_t &pixel_index, const *Formulation* &form)

evaluates second Piola-Kirchhoff stress given the Green-Lagrange strain (or Cauchy stress if called with a small strain tensor)

template<typename **Derived**>
inline decltype(auto) **evaluate_stress_tangent**(const *Eigen*::MatrixBase<*Derived*> &E, const size_t &pixel_index, const *Formulation* &form)

evaluates second Piola-Kirchhoff stress and its corresponding tangent given the Green-Lagrange strain (or Cauchy stress and its corresponding tangetn if called with a small strain tensor)

template<*Formulation* **Form**, class **Strains**, class **Stresses**>
void **constitutive_law**(const *Strains* &strains, *Stresses* &stress, const size_t &quad_pt_id)

template<*Formulation* **Form**, class **Strains**, class **Stresses**>
void **constitutive_law**(const *Strains* &strains, *Stresses* &stress, const size_t &quad_pt_id, const Real &ratio)

template<*Formulation* **Form**, class **Strains**, class **Stresses**>
void **constitutive_law_tangent**(const *Strains* &strains, *Stresses* &stresses, const size_t &quad_pt_id)

template<*Formulation* **Form**, class **Strains**, class **Stresses**>
void **constitutive_law_tangent**(const *Strains* &strains, *Stresses* &stresses, const size_t &quad_pt_id, const Real &ratio)

template<*Formulation* **Form**, class **Strains_t**>
decltype(auto) **constitutive_law**(const *Strains_t* &Strains, const size_t &quad_pt_id)

template<*Formulation* **Form**, class **Strains_t**>
decltype(auto) **constitutive_law_tangent**(const *Strains_t* &Strains, const size_t &quad_pt_id)

virtual void **compute_stresses**(const *RealField* &F, *RealField* &P, const *Formulation* &form, *SplitCell* is_cell_split) final

computes stress

virtual void **compute_stresses_tangent**(const *RealField* &F, *RealField* &P, *RealField* &K, const *Formulation* &form, *SplitCell* is_cell_split) final

stress and tangent modulus

virtual void **add_pixel**(const size_t &pixel_id) final

overload add_pixel to write into volume ratio and normal vectors and ...

void **add_pixel**(const size_t &pixel_id, *MatPtr_t* mat1, *MatPtr_t* mat2, const Real &ratio, const *Eigen*::Ref<const *Eigen*::Matrix<Real, *Eigen*::Dynamic, 1>> &normal_Vector)

overload add_pixel to add underlying materials and their ratio and interface direction to the material laminate

void **add_pixels_precipitate**(const std::vector<*Ccoord_t*<*DimM*>> &intersected_pixels, const std::vector<Dim_t> &intersected_pixels_id, const std::vector<Real> &intersection_ratios, const std::vector<*Eigen*::Matrix<Real, *DimM*, 1>> &intersection_normals, *MatPtr_t* mat1, *MatPtr_t* mat2)

This function adds pixels according to the precipitate intersected pixels and the materials incolved

virtual std::tuple<*DynMatrix_t*, *DynMatrix_t*> **constitutive_law_dynamic**(const *Eigen*::Ref<const *DynMatrix_t*> &strain, const size_t &pixel_index, const *Formulation* &form) final

evaluates both second Piola-Kirchhoff stress and stiffness given the Green-Lagrange strain (or Cauchy stress and stiffness if called with a small strain tensor)

Public Static Functions

static *MaterialLaminate*<*DimM*> &**make**(*Cell* &cell, const std::string &name)

Factory.

template<class ...**ConstructorArgs**>
static std::tuple<std::shared_ptr<*MaterialLaminate*<*DimM*>>, *MaterialEvaluator*<*DimM*>> **make_evaluator**(*ConstructorArgs* args)

Protected Functions

template<*Formulation* **Form**, *SplitCell* **IsCellSplit**>
inline void **compute_stresses_worker**(const *RealField* &F, *RealField* &P) __attribute__((visibility("default")))

computes stress with the formulation available at compile time **attribute** required by g++-6 and g++-7 because of this bug: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=80947

template<*Formulation* **Form**, *SplitCell* **IsCellSplit**>
inline void **compute_stresses_worker**(const *RealField* &F, *RealField* &P, *RealField* &K) __attribute__((visibility("default")))

computes stress with the formulation available at compile time **attribute** required by g++-6 and g++-7 because of this bug: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=80947

Protected Attributes

MappedVectorField_t **normal_vector_field**

field holding the normal vector of the interface of the layers

MappedScalarField_t **volume_ratio_field**

field holding the normal vector

std::vector<*MatPtr_t*> **material_left_vector** = { }

std::vector<*MatPtr_t*> **material_right_vector** = { }

template<Dim_t **DimM**>

class **MaterialLinearAnisotropic** : public *muSpectre*::*MaterialMuSpectre*<*MaterialLinearAnisotropic*<*DimM*>, *DimM*>

#include <material_linear_anisotropic.hh> Material implementation for anisotropic constitutive law

Subclassed by *muSpectre::MaterialLinearOrthotropic< DimM >*

Public Types

using **Parent** = *MaterialMuSpectre*<*MaterialLinearAnisotropic*, *DimM*>

base class

using **Stiffness_t** = *muGrid*::*T4Mat*<Real, *DimM*>

using **traits** = *MaterialMuSpectre_traits*<*MaterialLinearAnisotropic*>

traits of this material

using **Hooke** = typename *MatTB*::*Hooke*<*DimM*, typename *traits*::StrainMap_t::reference, typename *traits*::TangentMap_t::reference>

Hooke's law implementation.

Public Functions

MaterialLinearAnisotropic() = delete

Default constructor.

MaterialLinearAnisotropic(const std::string &name, const Dim_t &spatial_dimension, const Dim_t &nb_quad_pts, const std::vector<Real> &input_c)

MaterialLinearAnisotropic(const *MaterialLinearAnisotropic* &other) = delete

Copy constructor.

MaterialLinearAnisotropic(*MaterialLinearAnisotropic* &&other) = delete

Move constructor.

virtual **~MaterialLinearAnisotropic**() = default

Destructor.

template<class **s_t**>
inline auto **evaluate_stress**(*s_t* &&E) -> decltype(auto)

template<class **s_t**>
inline auto **evaluate_stress**(*s_t* &&E, const size_t&) -> decltype(auto)

template<class **s_t**>
inline auto **evaluate_stress_tangent**(*s_t* &&E) -> decltype(auto)

evaluates both second Piola-Kirchhoff stress and stiffness given the Green-Lagrange strain (or Cauchy stress and stiffness if called with a small strain tensor) and the local stiffness tensor.

template<class **s_t**>
inline auto **evaluate_stress_tangent**(*s_t* &&E, const size_t&) -> decltype(auto)

Public Static Functions

static auto **c_maker**(std::vector<Real> input) -> *Stiffness_t*

Protected Attributes

std::unique_ptr<*Stiffness_t*> **C_holder**

Stiffness_t &**C**

memory for stiffness tensor

stiffness tensor

template<Dim_t **DimM**>

class **MaterialLinearElastic1** : public *muSpectre*::*MaterialMuSpectre*<*MaterialLinearElastic1*<*DimM*>, *DimM*>

#include <material_linear_elastic1.hh> DimM material_dimension (dimension of constitutive law)

implements objective linear elasticity

Public Types

using **Parent** = *MaterialMuSpectre*<*MaterialLinearElastic1*, *DimM*>

base class

using **Stiffness_t** = T4Mat<Real, *DimM*>

short hand for the type of the elastic tensor

using **traits** = *MaterialMuSpectre_traits*<*MaterialLinearElastic1*>

traits of this material

using **Hooke** = typename *MatTB*::*Hooke*<*DimM*, typename *traits*::StrainMap_t::reference, typename *traits*::TangentMap_t::reference>

Hooke's law implementation.

Public Functions

MaterialLinearElastic1() = delete

Default constructor.

MaterialLinearElastic1(const *MaterialLinearElastic1* &other) = delete

Copy constructor.

MaterialLinearElastic1(const std::string &name, const Dim_t &spatial_dimension, const Dim_t &nb_quad_pts, const Real &young, const Real &poisson)

Construct by name, Young's modulus and Poisson's ratio.

MaterialLinearElastic1(*MaterialLinearElastic1* &&other) = delete

Move constructor.

virtual **~MaterialLinearElastic1**() = default

Destructor.

MaterialLinearElastic1 &**operator=**(const *MaterialLinearElastic1* &other) = delete

Copy assignment operator.

MaterialLinearElastic1 &**operator=**(*MaterialLinearElastic1* &&other) = delete

Move assignment operator.

template<class **Derived**>
inline decltype(auto) **evaluate_stress**(const *Eigen*::MatrixBase<*Derived*> &E, const size_t&)

evaluates second Piola-Kirchhoff stress given the Green-Lagrange strain (or Cauchy stress if called with a small strain tensor)

template<class **Derived**>
inline decltype(auto) **evaluate_stress_tangent**(const *Eigen*::MatrixBase<*Derived*> &E, const size_t&)

evaluates both second Piola-Kirchhoff stress and stiffness given the Green-Lagrange strain (or Cauchy stress and stiffness if called with a small strain tensor)

template<class **Derived**>
auto **evaluate_stress**(const *Eigen*::MatrixBase<*Derived*> &E, const size_t&) -> decltype(auto)

template<class **Derived**>
auto **evaluate_stress_tangent**(const *Eigen*::MatrixBase<*Derived*> &E, const size_t&) -> decltype(auto)

Protected Attributes

const Real **young**

Young's modulus.

const Real **poisson**

Poisson's ratio.

const Real **lambda**

first Lamé constant

const Real **mu**

second Lamé constant (shear modulus)

std::unique_ptr<const *Stiffness_t*> **C_holder**

stiffness tensor

const *Stiffness_t* &**C**

ref to stiffness tensor

template<Dim_t **DimM**>

class **MaterialLinearElastic2** : public *muSpectre*::*MaterialMuSpectre*<*MaterialLinearElastic2*<*DimM*>, *DimM*>

#include <material_linear_elastic2.hh> implements objective linear elasticity with an eigenstrain per pixel

Public Types

using **Parent** = *MaterialMuSpectre*<*MaterialLinearElastic2*, *DimM*>

base class

using **traits** = *MaterialMuSpectre_traits*<*MaterialLinearElastic2*>

traits of this material

using **StrainTensor** = *Eigen*::Ref<const *Eigen*::Matrix<Real, *DimM*, *DimM*>>

reference to any type that casts to a matrix

Public Functions

MaterialLinearElastic2() = delete

Default constructor.

MaterialLinearElastic2(const std::string &name, const Dim_t &spatial_dimension, const Dim_t &nb_quad_pts, Real young, Real poisson)

Construct by name, Young's modulus and Poisson's ratio.

MaterialLinearElastic2(const *MaterialLinearElastic2* &other) = delete

Copy constructor.

MaterialLinearElastic2(*MaterialLinearElastic2* &&other) = delete

Move constructor.

virtual **~MaterialLinearElastic2**() = default

Destructor.

MaterialLinearElastic2 &**operator=**(const *MaterialLinearElastic2* &other) = delete

Copy assignment operator.

MaterialLinearElastic2 &**operator=**(*MaterialLinearElastic2* &&other) = delete

Move assignment operator.

template<class **s_t**>
inline decltype(auto) **evaluate_stress**(*s_t* &&E, const size_t &quad_pt_index)

evaluates second Piola-Kirchhoff stress given the Green-Lagrange strain (or Cauchy stress if called with a small strain tensor)

template<class **s_t**>
inline decltype(auto) **evaluate_stress_tangent**(*s_t* &&E, const size_t &quad_pt_index)

evaluates both second Piola-Kirchhoff stress and stiffness given the Green-Lagrange strain (or Cauchy stress and stiffness if called with a small strain tensor)

virtual void **add_pixel**(const size_t &pixel_index) final

overload add_pixel to write into eigenstrain

void **add_pixel**(const size_t &pixel_index, const *StrainTensor* &E_eig)

overload add_pixel to write into eigenstrain

template<class **s_t**>
auto **evaluate_stress**(*s_t* &&E, const size_t &quad_pt_index) -> decltype(auto)

template<class **s_t**>
auto **evaluate_stress_tangent**(*s_t* &&E, const size_t &quad_pt_index) -> decltype(auto)

Protected Attributes

MaterialLinearElastic1<*DimM*> **material**

linear material without eigenstrain used to compute response

muGrid::*MappedT2Field*<Real, Mapping::Const, *DimM*> **eigen_strains**

storage for eigenstrain

template<Dim_t **DimM**>

class **MaterialLinearElastic3** : public *muSpectre*::*MaterialMuSpectre*<*MaterialLinearElastic3*<*DimM*>, *DimM*>

#include <material_linear_elastic3.hh> implements objective linear elasticity with an eigenstrain per pixel

Public Types

using **Parent** = *MaterialMuSpectre*<*MaterialLinearElastic3*, *DimM*>

base class

using **NeedTangent** = typename *Parent*::NeedTangent

type used to determine whether the `muSpectre::MaterialMuSpectre::iterable_proxy` evaluate only stresses or also tangent stiffnesses

using **traits** = *MaterialMuSpectre_traits*<*MaterialLinearElastic3*>

global field collection

traits of this material

using **Hooke** = typename *MatTB*::*Hooke*<*DimM*, typename *traits*::StrainMap_t::reference, typename *traits*::TangentMap_t::reference>

Hooke's law implementation.

using **StiffnessField_t** = *muGrid*::*MappedT4Field*<Real, Mapping::Const, *DimM*>

short hand for storage type of elastic tensors

Public Functions

MaterialLinearElastic3() = delete

Default constructor.

MaterialLinearElastic3(const std::string &name, const Dim_t &spatial_dimension, const Dim_t &nb_quad_pts)

Construct by name.

MaterialLinearElastic3(const *MaterialLinearElastic3* &other) = delete

Copy constructor.

MaterialLinearElastic3(*MaterialLinearElastic3* &&other) = delete

Move constructor.

virtual **~MaterialLinearElastic3**() = default

Destructor.

MaterialLinearElastic3 &**operator=**(const *MaterialLinearElastic3* &other) = delete

Copy assignment operator.

MaterialLinearElastic3 &**operator=**(*MaterialLinearElastic3* &&other) = delete

Move assignment operator.

template<class **Derived**>
inline decltype(auto) **evaluate_stress**(const *Eigen*::MatrixBase<*Derived*> &E, const typename *StiffnessField_t*::Return_t &C)

evaluates second Piola-Kirchhoff stress given the Green-Lagrange strain (or Cauchy stress if called with a small strain tensor) and the local stiffness tensor.

template<class **Derived**>
inline decltype(auto) **evaluate_stress**(const *Eigen*::MatrixBase<*Derived*> &E, const size_t &quad_pt_index)

evaluates second Piola-Kirchhoff stress given the Green-Lagrange strain (or Cauchy stress if called with a small strain tensor) and the local pixel id.

template<class **Derived**>
inline decltype(auto) **evaluate_stress_tangent**(const *Eigen*::MatrixBase<*Derived*> &E, const typename *StiffnessField_t*::Return_t &C)

evaluates both second Piola-Kirchhoff stress and tangent moduli given the Green-Lagrange strain (or Cauchy stress and tangent moduli if called with a small strain tensor) and the local tangent moduli tensor.

template<class **Derived**>

inline decltype(auto) **evaluate_stress_tangent**(const *Eigen*::MatrixBase<*Derived*> &E, const size_t &quad_pt_index)

evaluates both second Piola-Kirchhoff stress and tangent moduli given the Green-Lagrange strain (or Cauchy stress and tangent moduli if called with a small strain tensor) and the local pixel id.

virtual void **add_pixel**(const size_t &pixel_index) final

overload add_pixel to write into loacal stiffness tensor

void **add_pixel**(const size_t &pixel_index, const Real &Young, const Real &PoissonRatio)

overload add_pixel to write into local stiffness tensor

template<class **Derived**>
auto **evaluate_stress**(const *Eigen*::MatrixBase<*Derived*> &E, const typename *StiffnessField_t*::Return_t &C) -> decltype(auto)

template<class **Derived**>
auto **evaluate_stress_tangent**(const *Eigen*::MatrixBase<*Derived*> &E, const typename *StiffnessField_t*::Return_t &C) -> decltype(auto)

Protected Attributes

StiffnessField_t **C_field**

storage for stiffness tensor

template<Dim_t **DimM**>

class **MaterialLinearElastic4** : public *muSpectre*::*MaterialMuSpectre*<*MaterialLinearElastic4*<*DimM*>, *DimM*>

#include <material_linear_elastic4.hh> implements objective linear elasticity with an eigenstrain per pixel

Public Types

using **Parent** = *MaterialMuSpectre*<*MaterialLinearElastic4*, *DimM*>

base class

using **NeedTangent** = typename *Parent*::NeedTangent

type used to determine whether the `muSpectre::MaterialMuSpectre::iterable_proxy` evaluate only stresses or also tangent stiffnesses

using **Stiffness_t** = *Eigen*::TensorFixedSize<Real, *Eigen*::Sizes<*DimM*, *DimM*, *DimM*, *DimM*>>

global field collection

using **traits** = *MaterialMuSpectre_traits*<*MaterialLinearElastic4*>

traits of this material

using **Field_t** = *muGrid*::*MappedScalarField*<Real, Mapping::Const>

storage type for Lamé constants

using **Hooke** = typename *MatTB*::*Hooke*<*DimM*, typename *traits*::StrainMap_t::reference, typename *traits*::TangentMap_t::reference>

Hooke's law implementation.

Public Functions

MaterialLinearElastic4() = delete

Default constructor.

explicit **MaterialLinearElastic4**(const std::string &name, const Dim_t &spatial_dimension, const Dim_t &nb_quad_pts)

Construct by name.

MaterialLinearElastic4(const *MaterialLinearElastic4* &other) = delete

Copy constructor.

MaterialLinearElastic4(*MaterialLinearElastic4* &&other) = delete

Move constructor.

virtual **~MaterialLinearElastic4**() = default

Destructor.

MaterialLinearElastic4 &**operator=**(const *MaterialLinearElastic4* &other) = delete

Copy assignment operator.

MaterialLinearElastic4 &**operator=**(*MaterialLinearElastic4* &&other) = delete

Move assignment operator.

template<class **Derived**>
inline decltype(auto) **evaluate_stress**(const *Eigen*::MatrixBase<*Derived*> &E, const Real &lambda, const Real &mu)

evaluates second Piola-Kirchhoff stress given the Green-Lagrange strain (or Cauchy stress if called with a small strain tensor), the first Lame constant (lambda) and the second Lame constant (shear modulus/mu).

template<class **Derived**>
inline decltype(auto) **evaluate_stress**(const *Eigen*::MatrixBase<*Derived*> &E, const size_t &quad_pt_index)

evaluates second Piola-Kirchhoff stress given the Green-Lagrange strain (or Cauchy stress if called with a small strain tensor), and the local pixel id.

template<class **Derived**>
inline decltype(auto) **evaluate_stress_tangent**(const *Eigen*::MatrixBase<*Derived*> &E, const Real &lambda, const Real &mu)

evaluates both second Piola-Kirchhoff stress and stiffness given the Green-Lagrange strain (or Cauchy stress and stiffness if called with a small strain tensor), the first Lame constant (lambda) and the second Lame constant (shear modulus/mu).

template<class **Derived**>
inline decltype(auto) **evaluate_stress_tangent**(const *Eigen*::MatrixBase<*Derived*> &E, const size_t &quad_pt_index)

evaluates both second Piola-Kirchhoff stress and stiffness given the Green-Lagrange strain (or Cauchy stress and tangent moduli if called with a small strain tensor), and the local pixel id.

virtual void **add_pixel**(const size_t &pixel_index) final

overload add_pixel to write into loacal stiffness tensor

void **add_pixel**(const size_t &pixel_index, const Real &Youngs_modulus, const Real &Poisson_ratio)

overload add_pixel to write into local stiffness tensor

template<class **Derived**>
auto **evaluate_stress**(const *Eigen*::MatrixBase<*Derived*> &E, const Real &lambda, const Real &mu) -> decltype(auto)

template<class **Derived**>
auto **evaluate_stress_tangent**(const *Eigen*::MatrixBase<*Derived*> &E, const Real &lambda, const Real &mu) -> decltype(auto)

Protected Attributes

Field_t **lambda_field**

storage for first Lamé constant

Field_t **mu_field**

storage for second Lamé constant (shear modulus)

template<Dim_t **DimM**>

class **MaterialLinearElasticGeneric1** : public *muSpectre*::*MaterialMuSpectre*<*MaterialLinearElasticGeneric1*<*DimM*>, *DimM*>

#include <material_linear_elastic_generic1.hh> forward declaration

Linear elastic law defined by a full stiffness tensor. Very generic, but not most efficient. Note: it is template by ImpMaterial to make other materials to inherit form this class without any malfunctioning. i.e. the typeof classes inherits from this class will be passed to *MaterialMuSpectre* and MAterialMuSpectre will be able to access their types and methods directly without any interference of *MaterialLinearElasticGeneric1*.

Public Types

using **Parent** = *MaterialMuSpectre*<*MaterialLinearElasticGeneric1*<*DimM*>, *DimM*>

parent type

using **CInput_t** = *Eigen*::Ref<*Eigen*::Matrix<Real, *Eigen*::Dynamic, *Eigen*::Dynamic>, 0, *Eigen*::Stride<*Eigen*::Dynamic, *Eigen*::Dynamic>>

generic input tolerant to python input

Public Functions

MaterialLinearElasticGeneric1() = delete

Default constructor.

MaterialLinearElasticGeneric1(const std::string &name, const Dim_t &spatial_dimension, const Dim_t &nb_quad_pts, const *CInput_t* &C_voigt)

Constructor by name and stiffness tensor.

Parameters

- **name** – unique material name
- **spatial_dimension** – spatial dimension of the problem. This corresponds to the dimensionality of the *Cell*
- **nb_quad_pts** – number of quadrature points per pixel
- **C_voigt** – elastic tensor in Voigt notation

MaterialLinearElasticGeneric1(const *MaterialLinearElasticGeneric1* &other) = delete

Copy constructor.

MaterialLinearElasticGeneric1(*MaterialLinearElasticGeneric1* &&other) = delete

Move constructor.

virtual **~MaterialLinearElasticGeneric1**() = default

Destructor.

MaterialLinearElasticGeneric1 &**operator=**(const *MaterialLinearElasticGeneric1* &other) = delete

Copy assignment operator.

MaterialLinearElasticGeneric1 &**operator=**(*MaterialLinearElasticGeneric1* &&other) = delete

Move assignment operator.

template<class **Derived**>
inline decltype(auto) **evaluate_stress**(const *Eigen*::MatrixBase<*Derived*> &E, const size_t &quad_pt_index = 0)

evaluates second Piola-Kirchhoff stress given the Green-Lagrange strain (or Cauchy stress if called with a small strain tensor). Note: the pixel index is ignored.

template<class **Derived**>
inline decltype(auto) **evaluate_stress_tangent**(const *Eigen*::MatrixBase<*Derived*> &E, const size_t &quad_pt_index = 0)

evaluates both second Piola-Kirchhoff stress and stiffness given the Green-Lagrange strain (or Cauchy stress and stiffness if called with a small strain tensor). Note: the pixel index is ignored.

inline const *muGrid*::*T4Mat*<Real, *DimM*> &**get_C**() const

return a reference to the stiffness tensor

template<class **Derived1**, class **Derived2**>
void **make_C_from_C_voigt**(const *Eigen*::MatrixBase<*Derived1*> &C_voigt, *Eigen*::MatrixBase<*Derived2*> &C_holder)

template<class **Derived**>
auto **evaluate_stress**(const *Eigen*::MatrixBase<*Derived*> &E, const size_t&) -> decltype(auto)

template<class **Derived**>
auto **evaluate_stress_tangent**(const *Eigen*::MatrixBase<*Derived*> &E, const size_t&) -> decltype(auto)

Protected Attributes

std::unique_ptr<*muGrid*::*T4Mat*<Real, *DimM*>> **C_holder**

const *muGrid*::*T4Mat*<Real, *DimM*> &**C**

stiffness tensor

template<Dim_t **DimM**>

class **MaterialLinearElasticGeneric2** : public *muSpectre*::*MaterialMuSpectre*<*MaterialLinearElasticGeneric2*<*DimM*>, *DimM*>

#include <material_linear_elastic_generic2.hh> forward declaration

Implementation proper of the class

Public Functions

MaterialLinearElasticGeneric2() = delete

Default constructor.

MaterialLinearElasticGeneric2(const std::string &name, const Dim_t &spatial_dimension, const Dim_t &nb_quad_pts, const *CInput_t* &C_voigt)

Construct by name and elastic stiffness tensor.

MaterialLinearElasticGeneric2(const *MaterialLinearElasticGeneric2* &other) = delete

Copy constructor.

MaterialLinearElasticGeneric2(*MaterialLinearElasticGeneric2* &&other) = default

Move constructor.

virtual **~MaterialLinearElasticGeneric2**() = default

Destructor.

MaterialLinearElasticGeneric2 &**operator=**(const *MaterialLinearElasticGeneric2* &other) = delete

Copy assignment operator.

MaterialLinearElasticGeneric2 &**operator=**(*MaterialLinearElasticGeneric2* &&other) = default

Move assignment operator.

template<class **Derived**>
inline decltype(auto) **evaluate_stress**(const *Eigen*::MatrixBase<*Derived*> &E, const *Eigen*::Map<const *Eigen*::Matrix<Real, *DimM*, *DimM*>> &E_eig)

evaluates second Piola-Kirchhoff stress given the Green-Lagrange strain (or Cauchy stress if called with a small strain tensor)

template<class **Derived**>
inline decltype(auto) **evaluate_stress**(const *Eigen*::MatrixBase<*Derived*> &E, const size_t &quad_pt_index)

evaluates second Piola-Kirchhoff stress given the Green-Lagrange strain (or Cauchy stress if called with a small strain tensor) and the local pixel id

template<class **Derived**>

inline decltype(auto) **evaluate_stress_tangent**(const *Eigen*::MatrixBase<*Derived*> &E, const *Eigen*::Map<const *Eigen*::Matrix<Real, *DimM*, *DimM*>> &E_eig)

evaluates both second Piola-Kirchhoff stress and stiffness given the Green-Lagrange strain (or Cauchy stress and stiffness if called with a small strain tensor)

template<class **Derived**>
inline decltype(auto) **evaluate_stress_tangent**(const *Eigen*::MatrixBase<*Derived*> &E, const size_t &quad_pt_index)

evaluates both second Piola-Kirchhoff stress and tangent moduli given the Green-Lagrange strain (or Cauchy stress and stiffness if called with a small strain tensor) and the local pixel id

inline const *muGrid*::*T4Mat*<Real, *DimM*> &**get_C**() const

return a reference to the stiffness tensor

virtual void **add_pixel**(const size_t &pixel_index) final

overload add_pixel to write into eigenstrain

void **add_pixel**(const size_t &pixel_index, const *StrainTensor* &E_eig)

overload add_pixel to write into eigenstrain

template<class **Derived**>
auto **evaluate_stress**(const *Eigen*::MatrixBase<*Derived*> &E, const *Eigen*::Map<const *Eigen*::Matrix<Real, *DimM*, *DimM*>> &E_eig) -> decltype(auto)

template<class **Derived**>
auto **evaluate_stress_tangent**(const *Eigen*::MatrixBase<*Derived*> &E, const *Eigen*::Map<const *Eigen*::Matrix<Real, *DimM*, *DimM*>> &E_eig) -> decltype(auto)

Protected Attributes

Law_t **worker**

elastic law without eigenstrain used as worker

muGrid::*MappedT2Field*<Real, Mapping::Const, *DimM*> **eigen_field**

storage for eigenstrain

underlying law to be evaluated

Private Types

using **Parent** = *MaterialMuSpectre*<*MaterialLinearElasticGeneric2*<*DimM*>, *DimM*>

parent type

using **Law_t** = *MaterialLinearElasticGeneric1*<*DimM*>

underlying worker class

using **CInput_t** = typename *Law_t*::CInput_t

generic input tolerant to python input

using **StrainTensor** = *Eigen*::Ref<*Eigen*::Matrix<Real, *DimM*, *DimM*>>

reference to any type that casts to a matrix

using **traits** = *MaterialMuSpectre_traits*<*MaterialLinearElasticGeneric2*>

traits of this material

template<Dim_t **DimM**>

class **MaterialLinearOrthotropic** : public *muSpectre*::*MaterialLinearAnisotropic*<*DimM*>

#include <material_linear_orthotropic.hh> Material implementation for orthotropic constitutive law

Public Types

using **Parent** = *MaterialLinearAnisotropic*<*DimM*>

base class

using **Stiffness_t** = *muGrid*::*T4Mat*<Real, *DimM*>

using **traits** = *MaterialMuSpectre_traits*<*MaterialLinearOrthotropic*>

traits of this material

Public Functions

MaterialLinearOrthotropic() = delete

Default constructor.

MaterialLinearOrthotropic(const std::string &name, const Dim_t &spatial_dimension, const Dim_t &nb_quad_pts, const std::vector<Real> &input)

MaterialLinearOrthotropic(const *MaterialLinearOrthotropic* &other) = delete

Copy constructor.

MaterialLinearOrthotropic(*MaterialLinearOrthotropic* &&other) = delete

Move constructor.

virtual **~MaterialLinearOrthotropic**() = default

Destructor.

Public Static Functions

static *MaterialLinearOrthotropic*<*DimM*> &**make**(*Cell* &cell, const std::string &name, const std::vector<Real> &input)

make function needs to be overloaded, because this class does not directly inherit from *MaterialMuSpectre*. If this overload is not made, calls to make for *MaterialLinearOrthotropic* would call the constructor for *MaterialLinearAnisotropic*

Protected Functions

std::vector<Real> **input_c_maker**(const std::vector<Real> &input)

template<> std::array< bool, 6 > ret_flag

template<> std::array< bool, 21 > ret_flag

Protected Static Attributes

static constexpr std::array<std::size_t, 2> **output_size** = {6, 21}

these variable are used to determine which elements of the stiffness matrix should be replaced with the inpts for the orthotropic material

static std::array<bool, *output_size*[*DimM* - 2]> **ret_flag**

template<class **Material**, Dim_t **DimM**>

class **MaterialMuSpectre** : public *muSpectre*::*MaterialBase*

#include <material_muSpectre_base.hh> material traits are used by *muSpectre::MaterialMuSpectre* to break the circular dependence created by the curiously recurring template parameter. These traits must define

- these `muSpectre::FieldMaps`:
  - `StrainMap_t`: typically a `muSpectre::MatrixFieldMap` for a constant second-order `muSpectre::TensorField`
  - `StressMap_t`: typically a `muSpectre::MatrixFieldMap` for a writable secord-order `muSpectre::TensorField`
  - `TangentMap_t`: typically a `muSpectre::T4MatrixFieldMap` for a writable fourth-order `muSpectre::TensorField`
- `strain_measure`: the expected strain type (will be replaced by the small-strain tensor `muspectre::StrainMeasure::Infinitesimal` in small strain computations)
- `stress_measure`: the measure of the returned stress. Is used by `muspectre::MaterialMuSpectre` to transform it into Cauchy stress (`muspectre::StressMeasure::Cauchy`) in small-strain computations and into first Piola-Kirchhoff stress `muspectre::StressMeasure::PK1` in finite-strain computations

Base class for most convenient implementation of materials

Public Types

using **NeedTangent** = *MatTB*::*NeedTangent*

type used to determine whether the `muSpectre::MaterialBase::iterable_proxy` evaluate only stresses or also tangent stiffnesses

using **Parent** = *MaterialBase*

base class

using **traits** = *MaterialMuSpectre_traits*<*Material*>

traits for the CRTP subclass

using **DynMatrix_t** = *Parent*::DynMatrix_t

using **Strain_t** = *Eigen*::Matrix<Real, *DimM*, *DimM*>

using **Stress_t** = *Strain_t*

using **Stiffness_t** = *muGrid*::*T4Mat*<Real, *DimM*>

Public Functions

MaterialMuSpectre() = delete

Default constructor.

explicit **MaterialMuSpectre**(const std::string &name, const Dim_t &spatial_dimension, const Dim_t &nb_quad_pts)

Construct by name.

MaterialMuSpectre(const *MaterialMuSpectre* &other) = delete

Copy constructor.

MaterialMuSpectre(*MaterialMuSpectre* &&other) = delete

Move constructor.

virtual **~MaterialMuSpectre**() = default

Destructor.

MaterialMuSpectre &**operator=**(const *MaterialMuSpectre* &other) = delete

Copy assignment operator.

MaterialMuSpectre &**operator=**(*MaterialMuSpectre* &&other) = delete

Move assignment operator.

template<class ...**InternalArgs**>
void **add_pixel_split**(const size_t &pixel_id, Real ratio, *InternalArgs*... args)

void **add_split_pixels_precipitate**(const std::vector<size_t> &intersected_pixel_ids, const std::vector<Real> &intersection_ratios)

virtual void **compute_stresses**(const *muGrid*::*RealField* &F, *muGrid*::*RealField* &P, const *Formulation* &form, *SplitCell* is_cell_split = *SplitCell*::*no*) final

computes stress

virtual void **compute_stresses_tangent**(const *muGrid*::*RealField* &F, *muGrid*::*RealField* &P, *muGrid*::*RealField* &K, const *Formulation* &form, *SplitCell* is_cell_split = *SplitCell*::*no*) final

computes stress and tangent modulus

virtual std::tuple<*DynMatrix_t*, *DynMatrix_t*> **constitutive_law_dynamic**(const *Eigen*::Ref<const *DynMatrix_t*> &strain, const size_t &pixel_index, const *Formulation* &form) final

evaluates both second Piola-Kirchhoff stress and stiffness given the Green-Lagrange strain (or Cauchy stress and stiffness if called with a small strain tensor)

Public Static Functions

template<class ...**ConstructorArgs**>
static *Material* &**make**(*Cell* &cell, const std::string &name, *ConstructorArgs*&&... args)

Factory. The ConstructorArgs refer the arguments after `name`

template<class ...**ConstructorArgs**>
static std::tuple<std::shared_ptr<*Material*>, *MaterialEvaluator*<*DimM*>> **make_evaluator**(*ConstructorArgs*&&... args)

Factory takes all arguments after the name of the underlying Material's constructor. E.g., if the underlying material is a *muSpectre::MaterialLinearElastic1*<threeD>, these would be Young's modulus and Poisson's ratio.

static inline constexpr Dim_t **MaterialDimension**()

return the material dimension at compile time

Protected Functions

template<*Formulation* **Form**, *SplitCell* **is_cell_split** = *SplitCell*::*no*>
inline void **compute_stresses_worker**(const *muGrid*::*RealField* &F, *muGrid*::*RealField* &P) __attribute__((visibility("default")))

computes stress with the formulation available at compile time **attribute** required by g++-6 and g++-7 because of this bug: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=80947

template<*Formulation* **Form**, *SplitCell* **is_cell_split** = *SplitCell*::*no*>
inline void **compute_stresses_worker**(const *muGrid*::*RealField* &F, *muGrid*::*RealField* &P, *muGrid*::*RealField* &K) __attribute__((visibility("default")))

computes stress with the formulation available at compile time **attribute** required by g++-6 and g++-7 because of this bug: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=80947

template<class **Material**>

struct **MaterialMuSpectre_traits**

template<Dim_t **DimM**>

struct **MaterialMuSpectre_traits**<*MaterialHyperElastoPlastic1*<*DimM*>>

#include <material_hyper_elasto_plastic1.hh> traits for hyper-elastoplastic material

Public Types

using **StrainMap_t** = *muGrid*::*T2FieldMap*<Real, Mapping::Const, *DimM*>
expected map type for strain fields

using **StressMap_t** = *muGrid*::*T2FieldMap*<Real, Mapping::Mut, *DimM*>
expected map type for stress fields

using **TangentMap_t** = *muGrid*::*T4FieldMap*<Real, Mapping::Mut, *DimM*>
expected map type for tangent stiffness fields

Public Static Attributes

static constexpr auto **strain_measure** = {*StrainMeasure*::*Gradient*}
declare what type of strain measure your law takes as input

static constexpr auto **stress_measure** = {*StressMeasure*::*Kirchhoff*}
declare what type of stress measure your law yields as output

template<Dim_t **DimM**>

struct **MaterialMuSpectre_traits**<*MaterialHyperElastoPlastic2*<*DimM*>>
#include <material_hyper_elasto_plastic2.hh> traits for hyper-elastoplastic material

Public Types

using **StrainMap_t** = *muGrid*::*T2FieldMap*<Real, Mapping::Const, *DimM*>
expected map type for strain fields

using **StressMap_t** = *muGrid*::*T2FieldMap*<Real, Mapping::Mut, *DimM*>
expected map type for stress fields

using **TangentMap_t** = *muGrid*::*T4FieldMap*<Real, Mapping::Mut, *DimM*>
expected map type for tangent stiffness fields

Public Static Attributes

static constexpr auto **strain_measure** = {*StrainMeasure*::*Gradient*}
declare what type of strain measure your law takes as input

static constexpr auto **stress_measure** = {*StressMeasure*::*Kirchhoff*}
declare what type of stress measure your law yields as output

template<Dim_t **DimM**>

struct **MaterialMuSpectre_traits**<*MaterialLaminate*<*DimM*>>

Public Types

using **StrainMap_t** = *muGrid*::*T2FieldMap*<Real, Mapping::Const, *DimM*>
expected map type for strain fields

using **StressMap_t** = *muGrid*::*T2FieldMap*<Real, Mapping::Mut, *DimM*>
expected map type for stress fields

using **TangentMap_t** = *muGrid*::*T4FieldMap*<Real, Mapping::Mut, *DimM*>
expected map type for tangent stiffness fields

Public Static Attributes

static constexpr auto **strain_measure** = {*StrainMeasure*::*Gradient*}

static constexpr auto **stress_measure** = {*StressMeasure*::*PK1*}
declare what type of stress measure your law yields as output

template<Dim_t **DimM**>

struct **MaterialMuSpectre_traits**<*MaterialLinearAnisotropic*<*DimM*>>

Public Types

using **StrainMap_t** = *muGrid*::*T2FieldMap*<Real, Mapping::Const, *DimM*>
expected map type for strain fields

using **StressMap_t** = *muGrid*::*T2FieldMap*<Real, Mapping::Mut, *DimM*>
expected map type for stress fields

using **TangentMap_t** = *muGrid*::*T4FieldMap*<Real, Mapping::Mut, *DimM*>
expected map type for tangent stiffness fields

Public Static Attributes

static constexpr auto **strain_measure** = {*StrainMeasure*::*GreenLagrange*}
declare what type of strain measure your law takes as input

static constexpr auto **stress_measure** = {*StressMeasure*::*PK2*}
declare what type of stress measure your law yields as output

template<Dim_t **DimM**>

struct **MaterialMuSpectre_traits**<*MaterialLinearElastic1*<*DimM*>>
#include <material_linear_elastic1.hh> traits for objective linear elasticity

Public Types

using **StrainMap_t** = *muGrid*::*T2FieldMap*<Real, Mapping::Const, *DimM*>
expected map type for strain fields

using **StressMap_t** = *muGrid*::*T2FieldMap*<Real, Mapping::Mut, *DimM*>
expected map type for stress fields

using **TangentMap_t** = *muGrid*::*T4FieldMap*<Real, Mapping::Mut, *DimM*>
expected map type for tangent stiffness fields

Public Static Attributes

static constexpr auto **strain_measure** = {*StrainMeasure*::*GreenLagrange*}
declare what type of strain measure your law takes as input

static constexpr auto **stress_measure** = {*StressMeasure*::*PK2*}
declare what type of stress measure your law yields as output

template<Dim_t **DimM**>

struct **MaterialMuSpectre_traits**<*MaterialLinearElastic2*<*DimM*>>
#include <material_linear_elastic2.hh> traits for objective linear elasticity with eigenstrain

Public Types

using **StrainMap_t** = *muGrid*::*T2FieldMap*<double, Mapping::Const, *DimM*>
expected map type for strain fields

using **StressMap_t** = *muGrid*::*T2FieldMap*<double, Mapping::Mut, *DimM*>
expected map type for stress fields

using **TangentMap_t** = *muGrid*::*T4FieldMap*<double, Mapping::Mut, *DimM*>
expected map type for tangent stiffness fields

Public Static Attributes

static constexpr auto **strain_measure** = {*StrainMeasure*::*GreenLagrange*}
declare what type of strain measure your law takes as input

static constexpr auto **stress_measure** = {*StressMeasure*::*PK2*}
declare what type of stress measure your law yields as output

template<Dim_t **DimM**>

struct **MaterialMuSpectre_traits**<*MaterialLinearElastic3*<*DimM*>>
#include <material_linear_elastic3.hh> traits for objective linear elasticity with eigenstrain

Public Types

using **StrainMap_t** = *muGrid*::*T2FieldMap*<Real, Mapping::Const, *DimM*>
expected map type for strain fields

using **StressMap_t** = *muGrid*::*T2FieldMap*<Real, Mapping::Mut, *DimM*>
expected map type for stress fields

using **TangentMap_t** = *muGrid*::*T4FieldMap*<Real, Mapping::Mut, *DimM*>
expected map type for tangent stiffness fields

Public Static Attributes

static constexpr auto **strain_measure** = {*StrainMeasure*::*GreenLagrange*}
declare what type of strain measure your law takes as input

static constexpr auto **stress_measure** = {*StressMeasure*::*PK2*}
declare what type of stress measure your law yields as output

template<Dim_t **DimM**>

struct **MaterialMuSpectre_traits**<*MaterialLinearElastic4*<*DimM*>>
#include <material_linear_elastic4.hh> traits for objective linear elasticity with eigenstrain

Public Types

using **StrainMap_t** = *muGrid*::*T2FieldMap*<Real, Mapping::Const, *DimM*>
expected map type for strain fields

using **StressMap_t** = *muGrid*::*T2FieldMap*<Real, Mapping::Mut, *DimM*>
expected map type for stress fields

using **TangentMap_t** = *muGrid*::*T4FieldMap*<Real, Mapping::Mut, *DimM*>
expected map type for tangent stiffness fields

Public Static Attributes

static constexpr auto **strain_measure** = {*StrainMeasure*::*GreenLagrange*}
declare what type of strain measure your law takes as input

static constexpr auto **stress_measure** = {*StressMeasure*::*PK2*}
declare what type of stress measure your law yields as output

template<Dim_t **DimM**>

struct **MaterialMuSpectre_traits**<*MaterialLinearElasticGeneric1*<*DimM*>>
#include <material_linear_elastic_generic1.hh> traits for use by *MaterialMuSpectre* for crtp

Public Types

using **StrainMap_t** = *muGrid*::*T2FieldMap*<Real, Mapping::Const, *DimM*>

global field collection

expected map type for strain fields

using **StressMap_t** = *muGrid*::*T2FieldMap*<Real, Mapping::Mut, *DimM*>

expected map type for stress fields

using **TangentMap_t** = *muGrid*::*T4FieldMap*<Real, Mapping::Mut, *DimM*>

expected map type for tangent stiffness fields

Public Static Attributes

static constexpr auto **strain_measure** = {*StrainMeasure*::*GreenLagrange*}

declare what type of strain measure your law takes as input

static constexpr auto **stress_measure** = {*StressMeasure*::*PK2*}

declare what type of stress measure your law yields as output

template<Dim_t **DimM**>

struct **MaterialMuSpectre_traits**<*MaterialLinearElasticGeneric2*<*DimM*>>

#include <material_linear_elastic_generic2.hh> traits for use by *MaterialMuSpectre* for crtp

Public Types

using **StrainMap_t** = *muGrid*::*T2FieldMap*<Real, Mapping::Const, *DimM*>

expected map type for strain fields

using **StressMap_t** = *muGrid*::*T2FieldMap*<Real, Mapping::Mut, *DimM*>

expected map type for stress fields

using **TangentMap_t** = *muGrid*::*T4FieldMap*<Real, Mapping::Mut, *DimM*>

expected map type for tangent stiffness fields

Public Static Attributes

static constexpr auto **strain_measure** = {*StrainMeasure*::*GreenLagrange*}

declare what type of strain measure your law takes as input

static constexpr auto **stress_measure** = {*StressMeasure*::*PK2*}

declare what type of stress measure your law yields as output

template<Dim_t **DimM**>

struct **MaterialMuSpectre_traits**<*MaterialLinearOrthotropic*<*DimM*>>

Public Types

using **StrainMap_t** = *muGrid*::*T2FieldMap*<Real, Mapping::Const, *DimM*>
expected map type for strain fields

using **StressMap_t** = *muGrid*::*T2FieldMap*<Real, Mapping::Mut, *DimM*>
expected map type for stress fields

using **TangentMap_t** = *muGrid*::*T4FieldMap*<Real, Mapping::Mut, *DimM*>
expected map type for tangent stiffness fields

Public Static Attributes

static constexpr auto **strain_measure** = {*StrainMeasure*::*GreenLagrange*}
declare what type of strain measure your law takes as input

static constexpr auto **stress_measure** = {*StressMeasure*::*PK2*}
declare what type of stress measure your law yields as output

template<Dim_t **DimM**>

struct **MaterialMuSpectre_traits**<*MaterialStochasticPlasticity*<*DimM*>>
#include <material_stochastic_plasticity.hh> traits for stochastic plasticity with eigenstrain

Public Types

using **StrainMap_t** = *muGrid*::*T2FieldMap*<Real, Mapping::Const, *DimM*>
expected map type for strain fields

using **StressMap_t** = *muGrid*::*T2FieldMap*<Real, Mapping::Mut, *DimM*>
expected map type for stress fields

using **TangentMap_t** = *muGrid*::*T4FieldMap*<Real, Mapping::Mut, *DimM*>
expected map type for tangent stiffness fields

Public Static Attributes

static constexpr auto **strain_measure** = {*StrainMeasure*::*GreenLagrange*}

declare what type of strain measure your law takes as input

static constexpr auto **stress_measure** = {*StressMeasure*::*PK2*}

declare what type of stress measure your law yields as output

template<Dim_t **DimM**, *StrainMeasure* **StrainMIn**, *StressMeasure* **StressMOut**>

struct **MaterialMuSpectre_traits**<*STMaterialLinearElasticGeneric1*<*DimM*, *StrainMIn*, *StressMOut*>>

#include <s_t_material_linear_elastic_generic1.hh> traits for use by *MaterialMuSpectre* for crtp

Public Types

using **StrainMap_t** = *muGrid*::*T2FieldMap*<Real, Mapping::Const, *DimM*>

expected map type for strain fields

using **StressMap_t** = *muGrid*::*T2FieldMap*<Real, Mapping::Mut, *DimM*>

expected map type for stress fields

using **TangentMap_t** = *muGrid*::*T4FieldMap*<Real, Mapping::Mut, *DimM*>

expected map type for tangent stiffness fields

Public Static Attributes

static constexpr auto **strain_measure** = {*StrainMIn*}

declare what type of strain measure your law takes as input

static constexpr auto **stress_measure** = {*StressMOut*}

declare what type of stress measure your law yields as output

template<Dim_t **DimM**>

class **MaterialStochasticPlasticity** : public *muSpectre*::*MaterialMuSpectre*<*MaterialStochasticPlasticity*<*DimM*>, *DimM*>

#include <material_stochastic_plasticity.hh> implements stochastic plasticity with an eigenstrain, Lame constants and plastic flow per pixel.

Public Types

using **Parent** = *MaterialMuSpectre*<*MaterialStochasticPlasticity*, *DimM*>

base class

using **Vector_t** = *Eigen*::Matrix<Real, *Eigen*::Dynamic, 1>

dynamic vector type for interactions with numpy/scipy/solvers etc.

using **EigenStrainArg_t** = *Eigen*::Map<*Eigen*::Matrix<Real, *DimM*, *DimM*>>

using **traits** = *MaterialMuSpectre_traits*<*MaterialStochasticPlasticity*>

traits of this material

using **Hooke** = typename *MatTB*::*Hooke*<*DimM*, typename *traits*::StrainMap_t::reference, typename *traits*::TangentMap_t::reference>

Hooke's law implementation.

Public Functions

MaterialStochasticPlasticity() = delete

Default constructor.

explicit **MaterialStochasticPlasticity**(const std::string &name, const Dim_t &spatial_dimension, const Dim_t &nb_quad_pts)

Construct by name.

MaterialStochasticPlasticity(const *MaterialStochasticPlasticity* &other) = delete

Copy constructor.

MaterialStochasticPlasticity(*MaterialStochasticPlasticity* &&other) = delete

Move constructor.

virtual **~MaterialStochasticPlasticity**() = default

Destructor.

MaterialStochasticPlasticity &**operator=**(const *MaterialStochasticPlasticity* &other) = delete

Copy assignment operator.

MaterialStochasticPlasticity &**operator=**(*MaterialStochasticPlasticity* &&other) = delete

Move assignment operator.

template<class **s_t**>
inline decltype(auto) **evaluate_stress**(*s_t* &&E, const size_t &pixel_index)

evaluates second Piola-Kirchhoff stress given the Green-Lagrange strain (or Cauchy stress if called with a small strain tensor), and the local pixel id.

template<class **s_t**>
inline decltype(auto) **evaluate_stress**(*s_t* &&E, const Real &lambda, const Real &mu, const *EigenStrainArg_t* &eigen_strain)

evaluates second Piola-Kirchhoff stress given the Green-Lagrange strain (or Cauchy stress if called with a small strain tensor), the first Lame constant (lambda) and the second Lame constant (shear modulus/mu).

template<class **s_t**>

inline decltype(auto) **evaluate_stress_tangent**(*s_t* &&E, const size_t &pixel_index)

evaluates both second Piola-Kirchhoff stress and stiffness given the Green-Lagrange strain (or Cauchy stress and stiffness if called with a small strain tensor), and the local pixel id.

template<class **s_t**>
inline decltype(auto) **evaluate_stress_tangent**(*s_t* &&E, const Real &lambda, const Real &mu, const *EigenStrainArg_t* &eigen_strain)

evaluates both second Piola-Kirchhoff stress and stiffness given the Green-Lagrange strain (or Cauchy stress and stiffness if called with a small strain tensor), the first Lame constant (lambda) and the second Lame constant (shear modulus/mu).

void **set_plastic_increment**(const size_t &quad_pt_id, const Real &increment)

set the plastic_increment on a single quadrature point

void **set_stress_threshold**(const size_t &quad_pt_id, const Real &threshold)

set the stress_threshold on a single quadrature point

void **set_eigen_strain**(const size_t &quad_pt_id, *Eigen*::Ref<*Eigen*::Matrix<Real, *DimM*, *DimM*>> &eigen_strain)

set the eigen_strain on a single quadrature point

const Real &**get_plastic_increment**(const size_t &quad_pt_id)

get the plastic_increment on a single quadrature point

const Real &**get_stress_threshold**(const size_t &quad_pt_id)

get the stress_threshold on a single quadrature point

const *Eigen*::Ref<*Eigen*::Matrix<Real, *DimM*, *DimM*>> **get_eigen_strain**(const size_t &quad_pt_id)

get the eigen_strain on a single quadrature point

void **reset_overloaded_quad_pts**()

reset_overloaded_quadrature points, reset the internal variable overloaded_quad_pts by clear the std::vector

virtual void **add_pixel**(const size_t &pixel_id) final

overload add_pixel to write into loacal stiffness tensor

void **add_pixel**(const size_t &pixel_id, const Real &Youngs_modulus, const Real &Poisson_ratio, const Real &plastic_increment, const Real &stress_threshold, const *Eigen*::Ref<const *Eigen*::Matrix<Real, *Eigen*::Dynamic, *Eigen*::Dynamic>> &eigen_strain)

overload add_pixel to write into local stiffness tensor

inline decltype(auto) **identify_overloaded_quad_pts**(*Cell* &cell, *Eigen*::Ref<*Vector_t*> &stress_numpy_array)

evaluate how many pixels have a higher stress than their stress threshold

inline std::vector<size_t> &**identify_overloaded_quad_pts**(const *muGrid*::TypedFieldBase<Real> &stress_field)

inline decltype(auto) **update_eigen_strain_field**(*Cell* &cell, *Eigen*::Ref<*Vector_t*> &stress_numpy_array)

Update the eigen_strain_field of overloaded pixels by a discrete plastic step from the plastic_increment_field in the direction of the deviatoric stress tensor

inline void **update_eigen_strain_field**(const *muGrid*::TypedFieldBase<Real> &stress_field)

inline void **archive_overloaded_quad_pts**(std::list<std::vector<size_t>> &avalanche_history)

Archive the overloaded pixels into an avalanche history

archive_overloaded_quad_pts(), archives the overloaded pixels saved in this->overloaded_quad_pts to the input vector avalanche_history and empties overloaded_quad_pts.

inline decltype(auto) **relax_overloaded_quad_pts**(*Cell* &cell, *Eigen*::Ref<*Vector_t*> &stress_numpy_array)

relax all overloaded pixels, return the new stress field and the avalance history

template<class **s_t**>
auto **evaluate_stress**(*s_t* &&E, const Real &lambda, const Real &mu, const *EigenStrainArg_t* &eigen_strain) -> decltype(auto)

template<class **s_t**>
auto **evaluate_stress_tangent**(*s_t* &&E, const Real &lambda, const Real &mu, const *EigenStrainArg_t* &eigen_strain) -> decltype(auto)

Protected Types

using **Field_t** = *muGrid*::*MappedScalarField*<Real, Mapping::Mut>

storage for first Lame constant 'lambda', second Lame constant(shear modulus) 'mu', plastic strain epsilon_p, and a vector of overloaded (stress>stress_threshold) pixel coordinates

using **LTensor_Field_t** = *muGrid*::*MappedT2Field*<Real, Mapping::Mut, *DimM*>

Protected Attributes

Field_t **lambda_field**

Field_t **mu_field**

Field_t **plastic_increment_field**

Field_t **stress_threshold_field**

LTensor_Field_t **eigen_strain_field**

std::vector<size_t> **overloaded_quad_pts** = { }

class **MaterialsToolboxError** : public runtime_error

#include <materials_toolbox.hh> thrown when generic materials-related runtime errors occur (mostly continuum mechanics problems)

Public Functions

inline explicit **MaterialsToolboxError**(const std::string &what)

constructor

inline explicit **MaterialsToolboxError**(const char *what)

constructor

template<*Formulation* **Form**>

struct **MaterialStressEvaluator**

Public Static Functions

template<class **Material**, class **Strain**, class **Stress**, class **Op**>
static inline decltype(auto) static **compute**(*Material* &mat, const *Strain* &strain, *Stress* &stress, const size_t &quad_pt_id, const *Op* &operation)

template<*Formulation* **Form**>

struct **MaterialStressEvaluator**

Public Static Functions

template<class **Material**, class **Strain**, class **Stress**, class **Op**>
static inline void **compute**(*Material* &mat, *Strain* &&strain, *Stress* &stress, const size_t &quad_pt_id, const *Op* &operation)

template<>

struct **MaterialStressEvaluator**<*Formulation*::*finite_strain*>

Public Static Functions

template<class **Material**, class **Strain**, class **Stress**, class **Op**>
static inline decltype(auto) static **compute**(*Material* &mat, const *Strain* &strain, *Stress* &stress, const size_t &quad_pt_id, const *Op* &operation)

template<>

struct **MaterialStressEvaluator**<*Formulation*::*finite_strain*>

Public Static Functions

template<class **Material**, class **Strain**, class **Stress**, class **Op**>
static inline void **compute**(*Material* &mat, *Strain* &&strain, *Stress* &stress, const size_t &quad_pt_id, const *Op* &operation)

template<*Formulation* **Form**>

struct **MaterialStressTangentEvaluator**

Public Static Functions

template<class **Material**, class **Strain**, class **Stress**, class **Stiffness**, class **Op**>
static inline decltype(auto) static **compute**(*Material* &mat, const *Strain* &strain, std::tuple<*Stress*, *Stiffness*> &stress_stiffness, const size_t &quad_pt_id, const *Op* &operation)

template<*Formulation* **Form**>

struct **MaterialStressTangentEvaluator**

Public Static Functions

template<class **Material**, class **Strain**, class **Stress**, class **Stiffness**, class **Op**>
static inline void **compute**(*Material* &mat, *Strain* &&strain, std::tuple<*Stress*, *Stiffness*> &stress_stiffness, const size_t &quad_pt_id, const *Op* &operation)

template<>

struct **MaterialStressTangentEvaluator**<*Formulation*::*finite_strain*>

Public Static Functions

template<class **Material**, class **Strain**, class **Stress**, class **Stiffness**, class **Op**>
static inline decltype(auto) static **compute**(*Material* &mat, const *Strain* &strain, std::tuple<*Stress*, *Stiffness*> &stress_stiffness, const size_t &quad_pt_id, const *Op* &operation)

template<>

struct **MaterialStressTangentEvaluator**<*Formulation*::*finite_strain*>

Public Static Functions

template<class **Material**, class **Strain**, class **Stress**, class **Stiffness**, class **Op**>
static inline void **compute**(*Material* &mat, *Strain* &&strain, std::tuple<*Stress*, *Stiffness*> &stress_stiffness, const size_t &quad_pt_id, const *Op* &operation)

struct **Negative**

#include <field_typed.hh> Simple structure used to allow for lazy evaluation of the unary '-' sign. When assiging the the negative of a field to another, as in field_a = -field_b, this structure allows to implement this operation without needing a temporary object holding the negative value of field_b.

Public Members

const TypedFieldBase &**field**

field on which the unary '-' was applied

template<*SplitCell* **IsSplit**>

class **Node**

Subclassed by *muSpectre::RootNode< IsSplit >*

Public Types

using **RootNode_t** = *RootNode*<*IsSplit*>

using **Vector_t** = *Eigen*::Matrix<Real, *Eigen*::Dynamic, 1>

Public Functions

Node() = delete

Default constructor.

Node(const Dim_t &dim, const *DynRcoord_t* &new_origin, const *DynCcoord_t* &new_lenghts, const Dim_t &depth, const Dim_t &max_depth, *RootNode_t* &root, const bool &is_root)

Node(const *Node* &other) = delete

Copy constructor.

Node(*Node* &&other) = default

Move constructor.

virtual **~Node**() = default

Destructor.

template<Dim_t **DimS**>
void **check_node_helper**()

void **check_node**()

template<Dim_t **DimS**>
void **split_node_helper**(const Real &ratio, const corkpp::IntersectionState &state)

template<Dim_t **DimS**>
void **split_node_helper**(const Real &intersection_ratio, const corkpp::vector_t &normal_vector, const corkpp::IntersectionState &state)

void **split_node**(const Real &ratio, const corkpp::IntersectionState &state)

void **split_node**(const Real &intersection_ratio, const corkpp::vector_t &normal_vector, const corkpp::IntersectionState &state)

template<Dim_t **DimS**>
void **divide_node_helper**()

void **divide_node**()

Protected Attributes

Dim_t **dim**

RootNode_t &**root_node**

DynRcoord_t **origin**

DynRcoord_t **Rlengths** = { }

DynCcoord_t **Clengths** = { }

int **depth**

bool **is_pixel**

int **children_no**

std::vector<*Node*> **children** = { }

template<Dim_t **Dim**, *FiniteDiff* **FinDif**>

struct **NumericalTangentHelper**

#include <materials_toolbox.hh> implementation-structure for computing numerical tangents. For internal use only.

Template Parameters

- **Dim** – dimensionality of the material
- **FinDif** – specificaition of the type of finite differences

Public Types

using **T4_t** = *muGrid*::*T4Mat*<Real, *Dim*>

short-hand for fourth-rank tensors

using **T2_t** = *Eigen*::Matrix<Real, *Dim*, *Dim*>

short-hand for second-rank tensors

using **T2_vec** = *Eigen*::Map<*Eigen*::Matrix<Real, *Dim* * *Dim*, 1>>

short-hand for second-rank tensor reshaped to a vector

Public Static Functions

template<class **FunType**, class **Derived**>
static inline *T4_t* **compute**(*FunType* &&fun, const *Eigen*::MatrixBase<*Derived*> &strain, Real delta)

compute and return the approximate tangent moduli at strain `strain`

template<Dim_t **Dim**>

struct **NumericalTangentHelper**<*Dim*, *FiniteDiff*::*centred*>

#include <materials_toolbox.hh> specialisation for centred differences

Public Types

using **T4_t** = *muGrid*::*T4Mat*<Real, *Dim*>

short-hand for fourth-rank tensors

using **T2_t** = *Eigen*::Matrix<Real, *Dim*, *Dim*>

short-hand for second-rank tensors

using **T2_vec** = *Eigen*::Map<*Eigen*::Matrix<Real, *Dim* * *Dim*, 1>>

short-hand for second-rank tensor reshaped to a vector

Public Static Functions

template<class **FunType**, class **Derived**>
static inline *T4_t* **compute**(*FunType* &&fun, const *Eigen*::MatrixBase<*Derived*> &strain, Real delta)

compute and return the approximate tangent moduli at strain `strain`

class **NumpyError** : public runtime_error

#include <numpy_tools.hh> base class for numpy related exceptions

Public Functions

inline explicit **NumpyError**(const std::string &what)

constructor

inline explicit **NumpyError**(const char *what)

constructor

template<typename **T**, class **Collection_t** = *GlobalFieldCollection*>

class **NumpyProxy**

#include <numpy_tools.hh> Wrap a pybind11::array into a *WrappedField* and check the shape of the array

Public Functions

inline **NumpyProxy**(*DynCcoord_t* nb_subdomain_grid_pts, *DynCcoord_t* subdomain_locations, *Dim_t* nb_components, pybind11::array_t<*T*, pybind11::array::f_style> array)

Construct a *NumpyProxy* given that we only know the number of components of the field. The constructor will complain if the grid dimension differs but will wrap any field whose number of components match. For example, a 3x3 grid with 8 components could look like this:

i. (8, 3, 3)

ii. (2, 4, 3, 3)

iii. (2, 2, 2, 3, 3) The method `get_components_shape` return the shape of the component part of the field in this case. For the above examples, it would return:

i. (8,)

ii. (2, 4)

iii. (2, 2, 2) Note that a field with a single component can be passed either with a shaping having leading dimension of one or without any leading dimension. In the latter case, `get_component_shape` will return a vector of size 0. The same applies for fields with a single quadrature point, whose dimension can be omitted. In general, the shape of the field needs to look like this: (component_1, component:2, quad_pt, grid_x, grid_y, grid_z) where the number of components and grid indices can be arbitrary.

inline **NumpyProxy**(*DynCcoord_t* nb_subdomain_grid_pts, *DynCcoord_t* subdomain_locations, *Dim_t* nb_quad_pts, std::vector<*Dim_t*> components_shape, pybind11::array_t<*T*, pybind11::array::f_style> array)

Construct a *NumpyProxy* given that we know the shape of the leading component indices. The constructor will complain if both the grid dimensions and the component dimensions differ. `get_component_shape` returns exactly the shape passed to this constructor.

In general, the shape of the field needs to look like this: (component_1, component:2, quad_pt, grid_x, grid_y, grid_z) where the number of components and grid indices can be arbitrary. The quad_pt dimension can be omitted if there is only a single quad_pt.

NumpyProxy(*NumpyProxy* &&other) = default

move constructor

inline *WrappedField*<*T*> &**get_field**()

inline const std::vector<*Dim_t*> &**get_components_shape**() const

inline std::vector<*Dim_t*> **get_components_and_quad_pt_shape**() const

Protected Attributes

Collection_t **collection**

WrappedField<*T*> **field**

Dim_t **quad_pt_shape**

std::vector<*Dim_t*> **components_shape**

number of quad pts, omit if zero

struct **OperationAddition**

Public Functions

inline explicit **OperationAddition**(const Real &ratio)

template<typename **Derived1**, typename **Derived2**>
inline void **operator()**(const *Eigen*::MatrixBase<*Derived1*> &material_stress,
Eigen::MatrixBase<*Derived2*> &stored_stress) const

Public Members

const Real &**ratio**

struct **OperationAssignment**

Public Functions

template<typename **Derived1**, typename **Derived2**>
inline void **operator()**(const *Eigen*::MatrixBase<*Derived1*> &material_stress,
Eigen::MatrixBase<*Derived2*> &stored_stress) const

struct **OptimizeResult**

#include <solver_common.hh> emulates scipy.optimize.OptimizeResult

Public Members

Eigen::ArrayXXd **grad**

Strain or Gradient F at solution.

Eigen::ArrayXXd **stress**

Cauchy stress or first Piola-Kirchhoff stress P at solution.

bool **success**

whether or not the solver exited successfully

Int **status**

Termination status of the optimizer. Its value depends on the underlying solver. Refer to message for details.

std::string **message**

Description of the cause of the termination.

Uint **nb_it**

number of iterations

Uint **nb_fev**

number of cell evaluations

Formulation **formulation**

continuum mechanic flag

template<Dim_t **DimS**>

class **PFFTEngine** : public *muFFT*::*FFTEngineBase*<*DimS*>

#include <pfft_engine.hh> implements the `muFFT::FFTEngineBase` interface using the FFTW library

Public Types

using **Parent** = *FFTEngineBase*<*DimS*>

base class

using **Ccoord** = typename *Parent*::Ccoord

cell coordinates type

using **Workspace_t** = typename *Parent*::Workspace_t

field for Fourier transform of second-order tensor

using **Field_t** = typename *Parent*::Field_t

real-valued second-order tensor

Public Functions

PFFTEngine() = delete

Default constructor.

PFFTEngine(*Ccoord* nb_grid_pts, Dim_t nb_components, *Communicator* comm = *Communicator*())

Constructor with the domain's number of grid points in each direciton, the number of components to transform, and the communicator

PFFTEngine(const *PFFTEngine* &other) = delete

Copy constructor.

PFFTEngine(*PFFTEngine* &&other) = default

Move constructor.

virtual **~PFFTEngine**() noexcept

Destructor.

PFFTEngine &**operator=**(const *PFFTEngine* &other) = delete

Copy assignment operator.

PFFTEngine &**operator=**(*PFFTEngine* &&other) = default

Move assignment operator.

virtual void **initialise**(*FFT_PlanFlags* plan_flags) override

compute the plan, etc

Workspace_t &**fft**(*Field_t* &field) override

forward transform

void **ifft**(*Field_t* &field) const override

inverse transform

Protected Attributes

MPI_Comm **mpi_comm**

MPI communicator.

pfft_plan **plan_fft** = {}

holds the plan for forward fourier transform

pfft_plan **plan_ifft** = {}

holds the plan for inverse fourier transform

ptrdiff_t **workspace_size** = {}

size of workspace buffer returned by planner

Real ***real_workspace** = {}

temporary real workspace that is correctly padded

Protected Static Attributes

static int **nb_engines** = {0}

number of times this engine has been instatiated

class **PixelIndexIterable**

#include <field_collection.hh> Lightweight proxy class providing iteration over the pixel indices of a *muGrid::FieldCollection*

Public Types

using **iterator** = typename std::vector<size_t>::const_iterator

stl

Public Functions

PixelIndexIterable() = delete

Default constructor.

PixelIndexIterable(const *PixelIndexIterable* &other) = delete

Copy constructor.

PixelIndexIterable(*PixelIndexIterable* &&other) = default

Move constructor.

virtual **~PixelIndexIterable**() = default

Destructor.

PixelIndexIterable &**operator=**(const *PixelIndexIterable* &other) = delete

Copy assignment operator.

PixelIndexIterable &**operator=**(*PixelIndexIterable* &&other) = delete

Move assignment operator.

iterator **begin**() const

stl

iterator **end**() const

stl

size_t **size**() const

stl

Protected Functions

explicit **PixelIndexIterable**(const *FieldCollection* &collection)

Constructor is protected, because no one ever need to construct this except the fieldcollection

Protected Attributes

friend FieldCollection

allow field collections to call the procted constructor of this iterable

const *FieldCollection* &**collection**

reference back to the proxied collection

template<size_t **Dim**>

class **Pixels** : public *muGrid*::*CcoordOps*::*DynamicPixels*

#include <ccoord_operations.hh> forward declaration

Centralised iteration over square (or cubic) discretisation grids.

Public Types

using **Parent** = *DynamicPixels*

base class

using **Ccoord** = *Ccoord_t*<*Dim*>

cell coordinates

Public Functions

inline **Pixels**(const *Ccoord* &nb_grid_pts = *Ccoord*{}, const *Ccoord* &locations = *Ccoord*{})

constructor

inline **Pixels**(const *Ccoord* &nb_grid_pts, const *Ccoord* &locations, const *Ccoord* &strides)

constructor with strides

Pixels(const *Pixels* &other) = default

copy constructor

Pixels &**operator=**(const *Pixels* &other) = default

assignment operator

virtual **~Pixels**() = default

inline *Dim_t* **get_index**(const *Ccoord* &ccoord) const

return index for a ccoord

inline iterator **begin**() const

stl conformance

inline iterator **end**() const

stl conformance

inline size_t **size**() const

stl conformance

Protected Functions

inline const *Ccoord* &**get_nb_grid_pts**() const

inline const *Ccoord* &**get_location**() const

inline const *Ccoord* &**get_strides**() const

template<Dim_t **Dim**, *StressMeasure* **StressM**, *StrainMeasure* **StrainM**>

struct **PK1_stress**

#include <stress_transformations_default_case.hh> Structure for functions returning PK1 stress from other stress measures

Public Static Functions

template<class **Strain_t**, class **Stress_t**>
static inline decltype(auto) **compute**(*Strain_t*&&, *Stress_t*&&)

returns the converted stress

template<class **Strain_t**, class **Stress_t**, class **Tangent_t**>
static inline decltype(auto) **compute**(*Strain_t*&&, *Stress_t*&&, *Tangent_t*&&)

returns the converted stress and stiffness

template<Dim_t **Dim**, *StrainMeasure* **StrainM**>

struct **PK1_stress**<*Dim*, *StressMeasure*::*Kirchhoff*, *StrainM*> : public *muSpectre*::*MatTB*::*internal*::*PK1_stress*<*Dim*, *StressMeasure*::*no_stress_*, *StrainMeasure*::*no_strain_*>

#include <stress_transformations_Kirchhoff_impl.hh> Specialisation for the case where we get Kirchhoff stress ()

Public Static Functions

template<class **Strain_t**, class **Stress_t**>
static inline decltype(auto) **compute**(*Strain_t* &&F, *Stress_t* &&tau)

returns the converted stress

template<Dim_t **Dim**>

struct **PK1_stress**<*Dim*, *StressMeasure*::*Kirchhoff*, *StrainMeasure*::*Gradient*> : public *muSpectre*::*MatTB*::*internal*::*PK1_stress*<*Dim*, *StressMeasure*::*Kirchhoff*, *StrainMeasure*::*no_strain_*>

#include <stress_transformations_Kirchhoff_impl.hh> Specialisation for the case where we get Kirchhoff stress () derived with respect to Gradient

Public Types

using **Parent** = PK1_stress<*Dim*, *StressMeasure*::*Kirchhoff*, *StrainMeasure*::*no_strain_*>

short-hand

Public Static Functions

template<class **Strain_t**, class **Stress_t**, class **Tangent_t**>
static inline decltype(auto) **compute**(*Strain_t* &&F, *Stress_t* &&tau, *Tangent_t* &&C)

returns the converted stress and stiffness

template<Dim_t **Dim**>

struct **PK1_stress**<*Dim*, *StressMeasure*::*Kirchhoff*, *StrainMeasure*::*GreenLagrange*> : public *muSpectre*::*MatTB*::*internal*::*PK1_stress*<*Dim*, *StressMeasure*::*Kirchhoff*, *StrainMeasure*::*no_strain_*>

#include <stress_transformations_Kirchhoff_impl.hh> Specialisation for the case where we get Kirchhoff stress () derived with respect to GreenLagrange

Public Types

using **Parent** = PK1_stress<*Dim*, *StressMeasure*::*Kirchhoff*, *StrainMeasure*::*no_strain_*>

short-hand

Public Static Functions

template<class **Strain_t**, class **Stress_t**, class **Tangent_t**>
static inline decltype(auto) **compute**(*Strain_t* &&F, *Stress_t* &&tau, *Tangent_t* &&C)

returns the converted stress and stiffness

template<Dim_t **Dim**, *StrainMeasure* **StrainM**>

struct **PK1_stress**<*Dim*, *StressMeasure*::*PK1*, *StrainM*> : public *muSpectre*::*MatTB*::*internal*::*PK1_stress*<*Dim*, *StressMeasure*::*no_stress_*, *StrainMeasure*::*no_strain_*>

#include <stress_transformations_PK1_impl.hh> Specialisation for the transparent case, where we already have Piola-Kirchhoff-1, PK1

Public Static Functions

template<class **Strain_t**, class **Stress_t**>
static inline decltype(auto) **compute**(*Strain_t*&&, *Stress_t* &&P)

returns the converted stress

template<Dim_t **Dim**>

struct **PK1_stress**<*Dim*, *StressMeasure*::*PK1*, *StrainMeasure*::*Gradient*> : public *muSpectre*::*MatTB*::*internal*::*PK1_stress*<*Dim*, *StressMeasure*::*PK1*, *StrainMeasure*::*no_strain_*>

#include <stress_transformations_PK1_impl.hh> Specialisation for the transparent case, where we already have PK1 stress *and* stiffness is given with respect to the transformation gradient

Public Types

using **Parent** = PK1_stress<*Dim*, *StressMeasure*::*PK1*, *StrainMeasure*::*no_strain_*>

base class

Public Static Functions

template<class **Strain_t**, class **Stress_t**, class **Tangent_t**>
static inline decltype(auto) **compute**(*Strain_t*&&, *Stress_t* &&P, *Tangent_t* &&K)

returns the converted stress and stiffness

template<Dim_t **Dim**, *StrainMeasure* **StrainM**>

struct **PK1_stress**<*Dim*, *StressMeasure*::*PK2*, *StrainM*> : public *muSpectre*::*MatTB*::*internal*::*PK1_stress*<*Dim*, *StressMeasure*::*no_stress_*, *StrainMeasure*::*no_strain_*>

#include <stress_transformations_PK2_impl.hh> Specialisation for the case where we get material stress (Piola-Kirchhoff-2, PK2)

Public Static Functions

template<class **Strain_t**, class **Stress_t**>
static inline decltype(auto) **compute**(*Strain_t* &&F, *Stress_t* &&S)

returns the converted stress

template<Dim_t **Dim**>

struct **PK1_stress**<*Dim*, *StressMeasure*::*PK2*, *StrainMeasure*::*Gradient*> : public *muSpectre*::*MatTB*::*internal*::*PK1_stress*<*Dim*, *StressMeasure*::*PK2*, *StrainMeasure*::*no_strain_*>

#include <stress_transformations_PK2_impl.hh> Specialisation for the case where we get material stress (Piola-Kirchhoff-2, PK2) derived with respect to the placement Gradient (F)

Public Types

using **Parent** = PK1_stress<*Dim*, *StressMeasure*::*PK2*, *StrainMeasure*::*no_strain_*>

base class

Public Static Functions

template<class **Strain_t**, class **Stress_t**, class **Tangent_t**>
static inline decltype(auto) **compute**(*Strain_t* &&F, *Stress_t* &&S, *Tangent_t* &&C)

returns the converted stress and stiffness

template<Dim_t **Dim**>

struct **PK1_stress**<*Dim*, *StressMeasure*::*PK2*, *StrainMeasure*::*GreenLagrange*> : public *muSpectre*::*MatTB*::*internal*::*PK1_stress*<*Dim*, *StressMeasure*::*PK2*, *StrainMeasure*::*no_strain_*>

#include <stress_transformations_PK2_impl.hh> Specialisation for the case where we get material stress (Piola-Kirchhoff-2, PK2) derived with respect to Green-Lagrange strain

Public Types

using **Parent** = PK1_stress<*Dim*, *StressMeasure*::*PK2*, *StrainMeasure*::*no_strain_*>

base class

Public Static Functions

template<class **Strain_t**, class **Stress_t**, class **Tangent_t**>
static inline decltype(auto) **compute**(*Strain_t* &&F, *Stress_t* &&S, *Tangent_t* &&C)

returns the converted stress and stiffness

template<Dim_t **Dim**, *StressMeasure* **StressM**, *StrainMeasure* **StrainM**>

struct **PK2_stress**

#include <stress_transformations_default_case.hh> Structure for functions returning PK2 stress from other stress measures

Public Static Functions

template<class **Strain_t**, class **Stress_t**>
static inline decltype(auto) **compute**(*Strain_t*&&, *Stress_t*&&)

returns the converted stress

template<class **Strain_t**, class **Stress_t**, class **Tangent_t**>
static inline decltype(auto) **compute**(*Strain_t*&&, *Stress_t*&&, *Tangent_t*&&)

returns the converted stress and stiffness

template<Dim_t **Dim**, *StrainMeasure* **StrainM**>

struct **PK2_stress**<*Dim*, *StressMeasure*::*Kirchhoff*, *StrainM*> : public *muSpectre*::*MatTB*::*internal*::*PK2_stress*<*Dim*, *StressMeasure*::*no_stress_*, *StrainMeasure*::*no_strain_*>

#include <stress_transformations_Kirchhoff_impl.hh> Specialisation for the case where we get Kirchhoff stress () and we need PK2(S)

Public Static Functions

template<class **Strain_t**, class **Stress_t**>
static inline decltype(auto) **compute**(*Strain_t* &&F, *Stress_t* &&tau)

returns the converted stress

template<Dim_t **Dim**, *StrainMeasure* **StrainM**>

struct **PK2_stress**<*Dim*, *StressMeasure*::*PK1*, *StrainM*> : public *muSpectre*::*MatTB*::*internal*::*PK2_stress*<*Dim*, *StressMeasure*::*no_stress_*, *StrainMeasure*::*no_strain_*>

#include <stress_transformations_PK1_impl.hh> Specialisation for the case where we get material stress (Piola-Kirchhoff-1, PK1)

Public Static Functions

template<class **Strain_t**, class **Stress_t**>
static inline decltype(auto) **compute**(*Strain_t* &&F, *Stress_t* &&P)

returns the converted stress

template<Dim_t **Dim**>

struct **PK2_stress**<*Dim*, *StressMeasure*::*PK1*, *StrainMeasure*::*Gradient*> : public *muSpectre*::*MatTB*::*internal*::*PK2_stress*<*Dim*, *StressMeasure*::*PK1*, *StrainMeasure*::*no_strain_*>

#include <stress_transformations_PK1_impl.hh> Specialisation for the case where we get material stress (Piola-Kirchhoff-1, PK1) derived with respect to the placement Gradient (F)

Public Types

using **Parent** = PK2_stress<*Dim*, *StressMeasure*::*PK1*, *StrainMeasure*::*no_strain_*>

base class

Public Static Functions

template<class **Strain_t**, class **Stress_t**, class **Tangent_t**>
static inline decltype(auto) **compute**(*Strain_t* &&F, *Stress_t* &&P, *Tangent_t* &&K)

returns the converted stress and stiffness

template<Dim_t **Dim**, *StrainMeasure* **StrainM**>

struct **PK2_stress**<*Dim*, *StressMeasure*::*PK2*, *StrainM*> : public *muSpectre*::*MatTB*::*internal*::*PK2_stress*<*Dim*, *StressMeasure*::*no_stress_*, *StrainMeasure*::*no_strain_*>

#include <stress_transformations_PK2_impl.hh> Specialisation for the transparent case, where we already have PK2 stress

Public Static Functions

template<class **Strain_t**, class **Stress_t**>
static inline decltype(auto) **compute**(*Strain_t*&&, *Stress_t* &&S)

returns the converted stress

template<Dim_t **Dim**>

struct **PK2_stress**<*Dim*, *StressMeasure*::*PK2*, *StrainMeasure*::*GreenLagrange*> : public *muSpectre*::*MatTB*::*internal*::*PK2_stress*<*Dim*, *StressMeasure*::*PK2*, *StrainMeasure*::*no_strain_*>

#include <stress_transformations_PK2_impl.hh> Specialisation for the transparent case, where we already have PK2 stress *and* stiffness is given with respect to the transformation Green-Lagrange

Public Types

using **Parent** = PK2_stress<*Dim*, *StressMeasure*::*PK2*, *StrainMeasure*::*no_strain_*>

base class

Public Static Functions

template<class **Strain_t**, class **Stress_t**, class **Tangent_t**>
static inline decltype(auto) **compute**(*Strain_t*&&, *Stress_t* &&S, *Tangent_t* &&C)

returns the converted stress and stiffness

template<Dim_t **DimS**>

class **PrecipitateIntersectBase**

Public Static Functions

static std::tuple<std::vector<corkpp::point_t>, std::vector<corkpp::point_t>> **correct_dimension**(const std::vector<*Rcoord_t*<*DimS*> &convex_poly_vertices, const *Rcoord_t*<*DimS*> &origin, const *Rcoord_t*<*DimS*> &lengths)

static corkpp::VolNormStateIntersection **intersect_precipitate**(const std::vector<*DynRcoord_t*> &convex_poly_vertices, const *Rcoord_t*<*DimS*> &origin, const *Rcoord_t*<*DimS*> &lengths)

this function is the palce that CORK is called to analyze the geometry and make the intersection of the precipitate with a grid

template<*Dim_t* **dim**, *Dim_t* **i**, *Dim_t* **j** = *dim* - 1>

struct **Proj**

#include <eigen_tools.hh> This is a static implementation of the explicit determination of log(Tensor) following Jog, C.S. J Elasticity (2008) 93:

a. https://doi.org/10.1007/s10659-008-9169-x

Public Static Functions

static inline decltype(auto) **compute**(const *Vec_t*<*dim*> &eigs, const *Mat_t*<*dim*> &T)

wrapped function (raison d'être)

template<>

struct **Proj**<1, 0, 0>

#include <eigen_tools.hh> catch the general tail case

Public Static Functions

static inline decltype(auto) **compute**(const *Vec_t*<*dim*>&, const *Mat_t*<*dim*>&)

wrapped function (raison d'être)

Public Static Attributes

static constexpr *Dim_t* **dim** = {1}

short-hand

static constexpr *Dim_t* **i** = {0}

short-hand

static constexpr *Dim_t* **j** = {0}

short-hand

template<*Dim_t* **dim**>

struct **Proj**<*dim*, 0, 1>

#include <eigen_tools.hh> catch the tail case when the last dimension is i

Public Static Functions

static inline decltype(auto) **compute**(const *Vec_t*<*dim*> &eigs, const *Mat_t*<*dim*> &T)

wrapped function (raison d'être)

Public Static Attributes

static constexpr *Dim_t* **i** = {0}

short-hand

static constexpr *Dim_t* **j** = {1}

short-hand

template<*Dim_t* **dim**, *Dim_t* **i**>

struct **Proj**<*dim*, *i*, 0>

#include <eigen_tools.hh> catch the normal tail case

Public Static Functions

static inline decltype(auto) **compute**(const *Vec_t*<*dim*> &eigs, const *Mat_t*<*dim*> &T)

wrapped function (raison d'être)

Public Static Attributes

static constexpr *Dim_t* **j** = {0}

short-hand

template<*Dim_t* **dim**, *Dim_t* **other**>

struct **Proj**<*dim*, *other*, *other*>

#include <eigen_tools.hh> catch the case when there's nothing to do

Public Static Functions

static inline decltype(auto) **compute**(const *Vec_t*<*dim*> &eigs, const *Mat_t*<*dim*> &T)

wrapped function (raison d'être)

template<class **Projection**>

struct **Projection_traits**

class **ProjectionBase**

#include <projection_base.hh> defines the interface which must be implemented by projection operators

Subclassed by *muSpectre::ProjectionDefault< DimS >*, *muSpectre::ProjectionFiniteStrainFast< DimS >*

Public Types

using **Vector_t** = *Eigen*::Matrix<Real, *Eigen*::Dynamic, 1>

Eigen type to replace fields.

using **GFieldCollection_t** = typename *muFFT*::*FFTEngineBase*::*GFieldCollection_t*

global FieldCollection

using **Field_t** = *muGrid*::TypedFieldBase<Real>

Field type on which to apply the projection.

using **iterator** = typename *muFFT*::*FFTEngineBase*::*iterator*

iterator over all pixels. This is taken from the FFT engine, because depending on the real-to-complex FFT employed, only roughly half of the pixels are present in Fourier space (because of the hermitian nature of the transform)

Public Functions

ProjectionBase() = delete

Default constructor.

ProjectionBase(*muFFT*::*FFTEngine_ptr* engine, *DynRcoord_t* domain_lengths, *Formulation* form)

Constructor with cell sizes.

ProjectionBase(const *ProjectionBase* &other) = delete

Copy constructor.

ProjectionBase(*ProjectionBase* &&other) = default

Move constructor.

virtual **~ProjectionBase**() = default

Destructor.

ProjectionBase &**operator=**(const *ProjectionBase* &other) = delete

Copy assignment operator.

ProjectionBase &**operator=**(*ProjectionBase* &&other) = delete

Move assignment operator.

virtual void **initialise**(*muFFT*::*FFT_PlanFlags* flags = *muFFT*::*FFT_PlanFlags*::*estimate*)

initialises the fft engine (plan the transform)

virtual void **apply_projection**(*Field_t* &field) = 0

apply the projection operator to a field

const *DynCcoord_t* &**get_nb_subdomain_grid_pts**() const

returns the process-local number of grid points in each direction of the cell

inline const *DynCcoord_t* &**get_subdomain_locations**() const

returns the process-local locations of the cell

const *DynCcoord_t* &**get_nb_domain_grid_pts**() const

returns the global number of grid points in each direction of the cell

inline const *DynRcoord_t* &**get_domain_lengths**() const

returns the physical sizes of the cell

const *DynRcoord_t* **get_pixel_lengths**() const

returns the physical sizes of the pixles of the cell

inline const *Formulation* &**get_formulation**() const

return the `muSpectre::Formulation` that is used in solving this cell. This allows tho check whether a projection is compatible with the chosen formulation

inline const auto &**get_communicator**() const

return the communicator object

return the raw projection operator. This is mainly intended for maintenance and debugging and should never be required in regular use

virtual std::array<Dim_t, 2> **get_strain_shape**() const = 0

returns the number of rows and cols for the strain matrix type (for full storage, the strain is stored in material_dim × material_dim matrices, but in symmetric storage, it is a column vector)

virtual Dim_t **get_nb_components**() const = 0

get number of components to project per pixel

const Dim_t &**get_dim**() const

return the number of spatial dimensions

const Dim_t &**get_nb_quad**() const

returns the number of quadrature points

muFFT::*FFTEngineBase* &**get_fft_engine**()

return a reference to the fft_engine

Protected Attributes

muFFT::*FFTEngine_ptr* **fft_engine**

handle on the fft_engine used

DynRcoord_t **domain_lengths**

physical sizes of the cell

Formulation **form**

formulation this projection can be applied to (determines whether the projection enforces gradients, small strain tensor or symmetric smal strain tensor

GFieldCollection_t &**projection_container**

A local `muSpectre::FieldCollection` to store the projection operator per k-space point. This is a local rather than a global collection, since the pixels considered depend on the FFT implementation. See http://www.fftw.org/fftw3_doc/Multi_002dDimensional-DFTs-of-Real-Data.html#Multi_002dDimensional-DFTs-of-Real-Data for an example

template<Dim_t **DimS**>

class **ProjectionDefault** : public *muSpectre*::*ProjectionBase*

#include <projection_default.hh> base class to inherit from if one implements a projection operator that is stored in form of a fourth-order tensor of real values per k-grid point

Subclassed by *muSpectre::ProjectionFiniteStrain< DimS >*, *muSpectre::ProjectionSmallStrain< DimS >*

Public Types

using **Parent** = *ProjectionBase*

base class

using **Vector_t** = typename *Parent*::Vector_t

to represent fields

using **Gradient_t** = *muFFT*::*Gradient_t*

gradient, i.e. derivatives in each Cartesian direction

using **Ccoord** = *Ccoord_t*<*DimS*>

cell coordinates type

using **Rcoord** = *Rcoord_t*<*DimS*>

spatial coordinates type

using **GFieldCollection_t** = *muGrid*::*GlobalFieldCollection*

global field collection

using **Field_t** = *muGrid*::TypedFieldBase<Real>

Real space second order tensor fields (to be projected)

using **Proj_t** = *muGrid*::*ComplexField*

fourier-space field containing the projection operator itself

using **Proj_map** = *muGrid*::*T4FieldMap*<Complex, Mapping::Mut, *DimS*>

iterable form of the operator

using **Vector_map** = *muGrid*::*MatrixFieldMap*<Complex, Mapping::Mut, *DimS* * *DimS*, 1>

vectorized version of the Fourier-space second-order tensor field

Public Functions

ProjectionDefault() = delete

Default constructor.

ProjectionDefault(*muFFT*::*FFTEngine_ptr* engine, *DynRcoord_t* lengths, *Gradient_t* gradient, *Formulation* form)

Constructor with cell sizes and formulation.

ProjectionDefault(const *ProjectionDefault* &other) = delete

Copy constructor.

ProjectionDefault(*ProjectionDefault* &&other) = default

Move constructor.

virtual **~ProjectionDefault**() = default

Destructor.

ProjectionDefault &**operator=**(const *ProjectionDefault* &other) = delete

Copy assignment operator.

ProjectionDefault &**operator=**(*ProjectionDefault* &&other) = delete

Move assignment operator.

virtual void **apply_projection**(*Field_t* &field) final

apply the projection operator to a field

Eigen::Map<*MatrixXXc*> **get_operator**()

virtual std::array<Dim_t, 2> **get_strain_shape**() const final

returns the number of rows and cols for the strain matrix type (for full storage, the strain is stored in material_dim × material_dim matrices, but in symmetriy storage, it is a column vector)

inline virtual Dim_t **get_nb_components**() const

get number of components to project per pixel

Public Static Functions

static inline constexpr Dim_t **NbComponents**()
get number of components to project per pixel

Protected Attributes

Proj_t &**Gfield**
field holding the operator

Proj_map **Ghat**
iterable version of operator

Gradient_t **gradient**
gradient (nabla) operator, can be computed using Fourier interpolation or through a weighted residual

class **ProjectionError** : public runtime_error
#include <projection_base.hh> base class for projection related exceptions

Public Functions

inline explicit **ProjectionError**(const std::string &what)
constructor

inline explicit **ProjectionError**(const char *what)
constructor

template<Dim_t **DimS**>

class **ProjectionFiniteStrain** : public *muSpectre*::*ProjectionDefault*<*DimS*>
#include <projection_finite_strain.hh> Implements the discrete finite strain gradient projection operator

Public Types

using **Parent** = *ProjectionDefault*<*DimS*>
base class

using **Ccoord** = typename *Parent*::Ccoord
cell coordinates type

using **Rcoord** = typename *Parent*::Rcoord
spatial coordinates type

using **Gradient_t** = typename *Parent*::Gradient_t
gradient, i.e. derivatives in each Cartesian direction

using **Proj_map** = *muGrid*::*T4FieldMap*<Real, Mapping::Mut, *DimS*>

Field type on which to apply the projection.

using **Vector_map** = *muGrid*::*MatrixFieldMap*<Complex, Mapping::Mut, *DimS* * *DimS*, 1>

iterable vectorised version of the Fourier-space tensor field

Public Functions

ProjectionFiniteStrain() = delete

Default constructor.

ProjectionFiniteStrain(*muFFT*::*FFTEngine_ptr* engine, const *DynRcoord_t* &lengths, *Gradient_t* gradient)

Constructor with fft_engine and stencil.

ProjectionFiniteStrain(*muFFT*::*FFTEngine_ptr* engine, const *DynRcoord_t* &lengths)

Constructor with fft_engine and default (Fourier) gradient.

ProjectionFiniteStrain(const *ProjectionFiniteStrain* &other) = delete

Copy constructor.

ProjectionFiniteStrain(*ProjectionFiniteStrain* &&other) = default

Move constructor.

virtual **~ProjectionFiniteStrain**() = default

Destructor.

ProjectionFiniteStrain &**operator=**(const *ProjectionFiniteStrain* &other) = delete

Copy assignment operator.

ProjectionFiniteStrain &**operator=**(*ProjectionFiniteStrain* &&other) = default

Move assignment operator.

virtual void **initialise**(*muFFT*::*FFT_PlanFlags* flags = *muFFT*::*FFT_PlanFlags*::*estimate*) final

initialises the fft engine (plan the transform)

template<Dim_t **DimS**>

class **ProjectionFiniteStrainFast** : public *muSpectre*::*ProjectionBase*

#include <projection_finite_strain_fast.hh> replaces *muSpectre::ProjectionFiniteStrain* with a faster and less memory-hungry alternative formulation. Use this if you don't have a very good reason not to (and tell me (author) about it, I'd be interested to hear it).

Public Types

using **Parent** = *ProjectionBase*

base class

using **Gradient_t** = *muFFT*::*Gradient_t*

gradient, i.e. derivatives in each Cartesian direction

using **Ccoord** = *Ccoord_t*<*DimS*>

cell coordinates type

using **Rcoord** = *Rcoord_t*<*DimS*>

spatial coordinates type

using **Field_t** = *muGrid*::TypedFieldBase<Real>

Real space second order tensor fields (to be projected)

using **Proj_t** = *muGrid*::*ComplexField*

Fourier-space field containing the projection operator itself.

using **Proj_map** = *muGrid*::*MatrixFieldMap*<Complex, Mapping::Mut, *DimS*, 1, *muGrid*::*Iteration*::*Pixel*>

iterable form of the operator

using **Grad_map** = *muGrid*::*MatrixFieldMap*<Complex, Mapping::Mut, *DimS*, *DimS*, *muGrid*::*Iteration*::*Pixel*>

iterable Fourier-space second-order tensor field

Public Functions

ProjectionFiniteStrainFast() = delete

Default constructor.

ProjectionFiniteStrainFast(*muFFT*::*FFTEngine_ptr* engine, const *DynRcoord_t* &lengths, *Gradient_t* gradient)

Constructor with FFT engine.

ProjectionFiniteStrainFast(*muFFT*::*FFTEngine_ptr* engine, const *DynRcoord_t* &lengths)

Constructor with FFT engine and default (Fourier) gradient.

ProjectionFiniteStrainFast(const *ProjectionFiniteStrainFast* &other) = delete

Copy constructor.

ProjectionFiniteStrainFast(*ProjectionFiniteStrainFast* &&other) = default

Move constructor.

virtual **~ProjectionFiniteStrainFast**() = default

Destructor.

ProjectionFiniteStrainFast &**operator=**(const *ProjectionFiniteStrainFast* &other) = delete

Copy assignment operator.

ProjectionFiniteStrainFast &**operator=**(*ProjectionFiniteStrainFast* &&other) = default

Move assignment operator.

virtual void **initialise**(*muFFT*::*FFT_PlanFlags* flags = *muFFT*::*FFT_PlanFlags*::*estimate*) final

initialises the fft engine (plan the transform)

virtual void **apply_projection**(*Field_t* &field) final

apply the projection operator to a field

Eigen::Map<*MatrixXXc*> **get_operator**()

virtual std::array<Dim_t, 2> **get_strain_shape**() const final

returns the number of rows and cols for the strain matrix type (for full storage, the strain is stored in material_dim × material_dim matrices, but in symmetriy storage, it is a column vector)

inline virtual Dim_t **get_nb_components**() const

get number of components to project per pixel

Public Static Functions

static inline constexpr Dim_t **NbComponents**()

get number of components to project per pixel

Protected Attributes

Proj_t &**xi_field**

field of normalised wave vectors

Proj_map **xis**

iterable normalised wave vectors

Gradient_t **gradient**

gradient (nabla) operator, can be computed using Fourier interpolation or through a weighted residual

template<Dim_t **DimS**>

class **ProjectionSmallStrain** : public *muSpectre*::*ProjectionDefault*<*DimS*>

#include <projection_small_strain.hh> Implements the small strain projection operator as defined in Appendix A1 of DOI: 10.1002/nme.5481 ("A finite element perspective on nonlinear FFT-based micromechanical simulations", Int. J. Numer. Meth. Engng 2017; 111 :903–926)

Public Types

using **Parent** = *ProjectionDefault*<*DimS*>

base class

using **Gradient_t** = typename *Parent*::Gradient_t

gradient, i.e. derivatives in each Cartesian direction

using **Ccoord** = typename *Parent*::Ccoord

cell coordinates type

using **Rcoord** = typename *Parent*::Rcoord

spatial coordinates type

using **Proj_t** = *muGrid*::*RealField*

Fourier-space field containing the projection operator itself.

using **Proj_map** = *muGrid*::*T4FieldMap*<Real, Mapping::Mut, *DimS*>

iterable operator

using **Vector_map** = *muGrid*::*MatrixFieldMap*<Complex, Mapping::Mut, *DimS* * *DimS*, 1>

iterable vectorised version of the Fourier-space tensor field

Public Functions

ProjectionSmallStrain() = delete

Default constructor.

ProjectionSmallStrain(*muFFT*::*FFTEngine_ptr* engine, const *DynRcoord_t* &lengths, *Gradient_t* gradient)

Constructor with fft_engine.

ProjectionSmallStrain(*muFFT*::*FFTEngine_ptr* engine, const *DynRcoord_t* &lengths)

Constructor with fft_engine and default (Fourier) gradient.

ProjectionSmallStrain(const *ProjectionSmallStrain* &other) = delete

Copy constructor.

ProjectionSmallStrain(*ProjectionSmallStrain* &&other) = default

Move constructor.

virtual **~ProjectionSmallStrain**() = default

Destructor.

ProjectionSmallStrain &**operator=**(const *ProjectionSmallStrain* &other) = delete

Copy assignment operator.

ProjectionSmallStrain &**operator=**(*ProjectionSmallStrain* &&other) = delete

Move assignment operator.

virtual void **initialise**(*muFFT*::*FFT_PlanFlags* flags = *muFFT*::*FFT_PlanFlags*::*estimate*) final

initialises the fft engine (plan the transform)

template<typename **T**, size_t **N**>

class **RefArray**

#include <ref_array.hh> work-around to allow making a statically sized array of references (which are forbidden by the C++ language

Public Functions

RefArray() = delete

Deleted default constructor.

template<typename ...**Vals**>
inline explicit **RefArray**(*Vals*&... vals)

bulk initialisation constructor

RefArray(const *RefArray* &other) = default

Copy constructor.

RefArray(*RefArray* &&other) = default

Move constructor.

virtual ~**RefArray**() = default

Destructor.

RefArray &**operator=**(const *RefArray* &other) = default

Copy assignment operator.

RefArray &**operator=**(*RefArray* &&other) = delete

Move assignment operator.

inline *T* &**operator[]**(size_t index)

random access operator

inline constexpr *T* &**operator[]**(size_t index) const

random constant access operator

Protected Attributes

std::array<*T**, *N*> **values** = { }

storage

template<typename **T**>

class **RefVector** : protected std::vector<*T**>

#include <ref_vector.hh> work-around to allow using vectors of references (which are forbidden by the C++ stl

Public Functions

RefVector() = default

Default constructor.

RefVector(const *RefVector* &other) = default

Copy constructor.

RefVector(*RefVector* &&other) = default

Move constructor.

virtual ~**RefVector**() = default

Destructor.

RefVector &**operator=**(const *RefVector* &other) = default
Copy assignment operator.

RefVector &**operator=**(*RefVector* &&other) = default
Move assignment operator.

inline void **push_back**(*T* &value)
stl

inline *T* &**at**(size_t index)
stl

inline const *T* &**at**(size_t index) const
stl

inline *T* &**operator[]**(size_t index)
random access operator

inline const *T* &**operator[]**(size_t index) const
random const access operator

inline iterator **begin**()
stl

inline iterator **end**()
stl

Private Types

using **Parent** = std::vector<*T**>

template<*SplitCell* **IsSplit**>

class **RootNode** : public *muSpectre*::*Node*<*IsSplit*>

Public Types

using **Parent** = *Node*<*IsSplit*>
base class

using **Vector_t** = typename *Parent*::Vector_t

Public Functions

RootNode() = delete
Default Constructor.

RootNode(const *Cell* &cell, const std::vector<*DynRcoord_t*> &vert_precipitate)
Constructing a root node for a cell and a preticipate inside that cell.

RootNode(const *RootNode* &other) = delete
Copy constructor.

RootNode(*RootNode* &&other) = default

Move constructor.

~RootNode() = default

Destructor.

inline std::vector<*DynCcoord_t*> **get_intersected_pixels**()

inline std::vector<size_t> **get_intersected_pixels_id**()

inline std::vector<Real> **get_intersection_ratios**()

inline *Vectors_t* **get_intersection_normals**()

inline std::vector<corkpp::IntersectionState> **get_intersection_status**()

Dim_t **make_max_resolution**(const *Cell* &cell) const

Dim_t **make_max_depth**(const *Cell* &cell) const

void **check_root_node**()

int **compute_squared_circum_square**(const *Cell* &cell) const

DynRcoord_t **make_root_origin**(const *Cell* &cell) const

Protected Attributes

const *Cell* &**cell**

DynRcoord_t **cell_length**

the cell to be intersected

DynRcoord_t **pixel_lengths**

The Real size of the cell.

DynCcoord_t **cell_resolution**

The Real size of each pixel.

int **max_resolution**

The nb_grid_pts for the.

int **max_depth**

The maximum of the nb_grid_pts in all directions.

std::vector<*DynRcoord_t*> **precipitate_vertices** = {}

The maximum depth of the branches in the OctTree.

std::vector<*DynCcoord_t*> **intersected_pixels** = {}

The coordinates of the vertices of the perticpiate

std::vector<size_t> **intersected_pixels_id** = { }

The pixels of the cell which intersect with the percipitate

std::vector<Real> **intersection_ratios** = { }

The index of the intersecting pixels.

Vectors_t **intersection_normals**

The intesrction ratio of intersecting pixels.

std::vector<corkpp::IntersectionState> **intersection_state** = { }

The normal vectors of the interface in the intersecting pixels

Friends

friend class Node< IsSplit >

template<Dim_t **Rank**>

struct **RotationHelper**

template<>

struct **RotationHelper**<firstOrder>

#include <geometry.hh> Specialisation for first-rank tensors (vectors)

Public Static Functions

template<class **Derived1**, class **Derived2**>
static inline decltype(auto) **rotate**(const *Eigen*::MatrixBase<*Derived1*> &input, const *Eigen*::MatrixBase<*Derived2*> &R)

template<>

struct **RotationHelper**<fourthOrder>

#include <geometry.hh> Specialisation for fourth-rank tensors

Public Static Functions

template<class **Derived1**, class **Derived2**>
static inline decltype(auto) **rotate**(const *Eigen*::MatrixBase<*Derived1*> &input, const *Eigen*::MatrixBase<*Derived2*> &R)

template<>

struct **RotationHelper**<secondOrder>

#include <geometry.hh> Specialisation for second-rank tensors

Public Static Functions

template<class **Derived1**, class **Derived2**>
static inline decltype(auto) **rotate**(const *Eigen*::MatrixBase<*Derived1*> &input, const *Eigen*::MatrixBase<*Derived2*> &R)

raison d'être

template<*RotationOrder* **Order**, Dim_t **Dim**>

struct **RotationMatrixComputerAngle**

#include <geometry.hh> internal structure for computing rotation matrices

template<*RotationOrder* **Order**>

struct **RotationMatrixComputerAngle**<*Order*, threeD>

#include <geometry.hh> specialisation for three-dimensional problems

Public Types

using **RotMat_t** = typename *RotatorAngle*<*Dim*, *Order*>::RotMat_t

using **Angles_t** = typename *RotatorAngle*<*Dim*, *Order*>::Angles_t

Public Static Functions

template<typename **Derived**>
static inline *RotMat_t* **compute**(const *Eigen*::MatrixBase<*Derived*> &angles)

compute and return the rotation matrixtemplate <typename derived>="">

Public Static Attributes

static constexpr Dim_t **Dim** = {threeD}

template<*RotationOrder* **Order**>

struct **RotationMatrixComputerAngle**<*Order*, twoD>

#include <geometry.hh> specialisation for two-dimensional problems

Public Types

using **RotMat_t** = typename *RotatorAngle*<*Dim*, *Order*>::RotMat_t

using **Angles_t** = typename *RotatorAngle*<*Dim*, *Order*>::Angles_t

Public Static Functions

template<typename **Derived**>
static inline *RotMat_t* **compute**(const *Eigen*::MatrixBase<*Derived*> &angles)

compute and return the rotation matrix

Public Static Attributes

static constexpr Dim_t **Dim** = {twoD}

template<Dim_t **Dim**>

struct **RotationMatrixComputerNormal**

template<>

struct **RotationMatrixComputerNormal**<threeD>

Public Types

using **RotMat_t** = typename *RotatorTwoVec*<*Dim*>::RotMat_t

using **Vec_t** = typename *RotatorTwoVec*<*Dim*>::Vec_t

Public Static Functions

template<typename **Derived**>
static inline *RotMat_t* **compute**(const *Eigen*::MatrixBase<*Derived*> &vec)

Public Static Attributes

static constexpr Dim_t **Dim** = {threeD}

template<>

struct **RotationMatrixComputerNormal**<twoD>

Public Types

using **RotMat_t** = typename *RotatorTwoVec*<*Dim*>::RotMat_t

using **Vec_t** = typename *RotatorTwoVec*<*Dim*>::Vec_t

Public Static Functions

template<typename **Derived**>
static inline *RotMat_t* **compute**(const *Eigen*::MatrixBase<*Derived*> &vec)

Public Static Attributes

static constexpr Dim_t **Dim** = {twoD}

template<Dim_t **Dim**>

struct **RotationMatrixComputerTwoVec**

template<>

struct **RotationMatrixComputerTwoVec**<threeD>

Public Types

using **RotMat_t** = typename *RotatorTwoVec*<*Dim*>::RotMat_t

using **Vec_t** = typename *RotatorTwoVec*<*Dim*>::Vec_t

Public Static Functions

template<typename **DerivedA**, typename **DerivedB**>
static inline *RotMat_t* **compute**(const *Eigen*::MatrixBase<*DerivedA*> &vec_ref, const *Eigen*::MatrixBase<*DerivedB*> &vec_des)

Public Static Attributes

static constexpr Dim_t **Dim** = {threeD}

template<>

struct **RotationMatrixComputerTwoVec**<twoD>

Public Types

using **RotMat_t** = typename *RotatorTwoVec*<*Dim*>::RotMat_t

using **Vec_t** = typename *RotatorTwoVec*<*Dim*>::Vec_t

Public Static Functions

template<typename **DerivedA**, typename **DerivedB**>
static inline *RotMat_t* **compute**(const *Eigen*::MatrixBase<*DerivedA*> &vec_ref, const *Eigen*::MatrixBase<*DerivedB*> &vec_des)

Public Static Attributes

static constexpr Dim_t **Dim** = {twoD}

template<Dim_t **Dim**, *RotationOrder* **Order** = *internal*::*DefaultOrder*<*Dim*>::value>

class **RotatorAngle** : public *muSpectre*::*RotatorBase*<*Dim*>

Public Types

using **Parent** = *RotatorBase*<*Dim*>

using **Angles_t** = *Eigen*::Matrix<Real, (*Dim* == twoD) ? 1 : 3, 1>

using **RotMat_t** = *Eigen*::Matrix<Real, *Dim*, *Dim*>

Public Functions

RotatorAngle() = delete

Default constructor.

template<class **Derived**>
inline explicit **RotatorAngle**(const *Eigen*::MatrixBase<*Derived*> &angles_inp)

constructor given the euler angles:

RotatorAngle(const *RotatorAngle* &other) = default

Copy constructor.

RotatorAngle(*RotatorAngle* &&other) = default

Move constructor.

virtual **~RotatorAngle**() = default

Destructor.

RotatorAngle &**operator=**(const *RotatorAngle* &other) = default

Copy assignment operator.

RotatorAngle &**operator=**(*RotatorAngle* &&other) = default

Move assignment operator.

template<typename **Derived**>
auto **compute_rotation_matrix_angle**(const *Eigen*::MatrixBase<*Derived*> &angles) -> *RotMat_t*

Protected Functions

template<class **Derived**>
inline *RotMat_t* **compute_rotation_matrix_angle**(const *Eigen*::MatrixBase<*Derived*> &angles)

template<Dim_t **Dim**>

class **RotatorBase**

Subclassed by *muSpectre::RotatorAngle< Dim, Order >*, *muSpectre::RotatorNormal< Dim >*, *muSpectre::RotatorTwoVec< Dim >*

Public Types

using **RotMat_t** = *Eigen*::Matrix<Real, *Dim*, *Dim*>

using **RotMat_ptr** = std::unique_ptr<*RotMat_t*>

Public Functions

RotatorBase() = delete

Default constructor.

inline explicit **RotatorBase**(*RotMat_t* rotation_matrix_input)

constructor with given rotation matrix

RotatorBase(const *RotatorBase* &other) = default

Copy constructor.

RotatorBase(*RotatorBase* &&other) = default

Move constructor.

virtual **~RotatorBase**() = default

Destructor.

RotatorBase &**operator=**(const *RotatorBase* &other) = default

Copy assignment operator.

RotatorBase &**operator=**(*RotatorBase* &&other) = default

Move assignment operator.

template<class **Derived**>
inline decltype(auto) **rotate**(const *Eigen*::MatrixBase<*Derived*> &input) const

Applies the rotation into the frame define my the rotation

matrix

Parameters
input – is a first-, second-, or fourth-rank tensor (column vector, square matrix, or T4Matrix, or a Eigen::Map of either of these, or an expression that evaluates into any of these)

template<class **Derived**>

inline decltype(auto) **rotate_back**(const *Eigen*::MatrixBase<*Derived*> &input) const

Applies the rotation back out from the frame define my the rotation matrix

Parameters

input – is a first-, second-, or fourth-rank tensor (column vector, square matrix, or T4Matrix, or a Eigen::Map of either of these, or an expression that evaluates into any of these)

inline const *RotMat_t* &**get_rot_mat**() const

template<class **Derived**>
inline void **set_rot_mat**(const *Eigen*::MatrixBase<*Derived*> &mat_inp)

Protected Attributes

RotMat_ptr **rot_mat_holder**

const *RotMat_t* &**rot_mat**

template<Dim_t **Dim**>

class **RotatorNormal** : public *muSpectre*::*RotatorBase*<*Dim*>

#include <geometry.hh> this class is used to make a vector aligned to x-axis of the coordinate system, the input for the constructor is the vector itself and the functions rotate and rotate back would be available as they exist in the parent class (*RotatorBase*) nad can be used in order to do the functionality of the class

Public Types

using **Parent** = *RotatorBase*<*Dim*>

using **Vec_t** = *Eigen*::Matrix<Real, *Dim*, 1>

using **RotMat_t** = *Eigen*::Matrix<Real, *Dim*, *Dim*>

Public Functions

RotatorNormal() = delete

Default constructor.

template<typename **Derived**>
inline explicit **RotatorNormal**(const *Eigen*::MatrixBase<*Derived*> &vec)

constructor

RotatorNormal(const *RotatorNormal* &other) = default

Copy constructor.

RotatorNormal(*RotatorNormal* &&other) = default

Move constructor.

virtual ~**RotatorNormal**() = default

Destructor.

RotatorNormal &**operator=**(const *RotatorNormal* &other) = default

Copy assignment operator.

RotatorNormal &**operator=**(*RotatorNormal* &&other) = default

Move assignment operator.

template<typename **Derived**>
auto **compute_rotation_matrix_normal**(const *Eigen*::MatrixBase<*Derived*> &vec) -> *RotMat_t*

Protected Functions

template<typename **Derived**>
inline *RotMat_t* **compute_rotation_matrix_normal**(const *Eigen*::MatrixBase<*Derived*> &vec)

template<Dim_t **Dim**>

class **RotatorTwoVec** : public *muSpectre*::*RotatorBase*<*Dim*>

#include <geometry.hh> this class is used to make the vector a aligned to the vec b by means of a rotation system, the input for the constructor is the vector itself and the functions rotate and rotate back would be available as they exist in the parent class (*RotatorBase*) nad can be used in order to do the functionality of the class

Public Types

using **Parent** = *RotatorBase*<*Dim*>

using **Vec_t** = *Eigen*::Matrix<Real, (*Dim* == twoD) ? 2 : 3, 1>

using **Vec_ptr** = std::unique_ptr<*Vec_t*>

using **RotMat_t** = *Eigen*::Matrix<Real, *Dim*, *Dim*>

Public Functions

RotatorTwoVec() = delete

Default constructor.

template<typename **DerivedA**, typename **DerivedB**>
inline **RotatorTwoVec**(const *Eigen*::MatrixBase<*DerivedA*> &vec_a_inp, const *Eigen*::MatrixBase<*DerivedB*> &vec_b_inp)

Constructor given the two vectors.

RotatorTwoVec(const *RotatorTwoVec* &other) = default

Copy constructor.

RotatorTwoVec(*RotatorTwoVec* &&other) = default

Move constructor.

virtual ~**RotatorTwoVec**() = default

Destructor.

RotatorTwoVec &**operator=**(const *RotatorTwoVec* &other) = default

Copy assignment operator.

RotatorTwoVec &**operator=**(*RotatorTwoVec* &&other) = default

Move assignment operator.

template<typename **DerivedA**, typename **DerivedB**>
auto **compute_rotation_matrix_TwoVec**(const *Eigen*::MatrixBase<*DerivedA*> &vec_ref, const *Eigen*::MatrixBase<*DerivedB*> &vec_des) -> *RotMat_t*

Protected Functions

template<typename **DerivedA**, typename **DerivedB**>
inline *RotMat_t* **compute_rotation_matrix_TwoVec**(const *Eigen*::MatrixBase<*DerivedA*> &vec_ref, const *Eigen*::MatrixBase<*DerivedB*> &vec_des)

template<typename **T**>

struct **ScalarMap**

#include <field_map_static.hh> Internal struct for handling the scalar iterates of *muGrid::FieldMap*

Public Types

using **PlainType** = *T*

Scalar maps don't have an eigen type representing the iterate, just the raw stored type itsef

using **value_type** = std::conditional_t<MutIter == *Mapping*::*Const*, const *T*, *T*>

return type for iterates

using **ref_type** = *value_type*<MutIter>&

reference type for iterates

using **Return_t** = *value_type*<MutIter>&

for direct access through operator[]

using **storage_type** = std::conditional_t<MutIter == *Mapping*::*Const*, const *T**, *T**>

need to encapsulate

Public Static Functions

static inline constexpr bool **IsValidStaticMapType**()

check at compile time whether this map is suitable for statically sized iterates

static inline constexpr bool **IsScalarMapType**()

check at compiler time whether this map is scalar

template<*Mapping* **MutIter**>

static inline constexpr *value_type*<*MutIter*> &**provide_ref**(*storage_type*<*MutIter*> storage)

return the return_type version of the iterate from storage_type

template<*Mapping* **MutIter**>
static inline constexpr const *value_type*<*MutIter*> &**provide_const_ref**(const *storage_type*<*MutIter*> storage)

return the const return_type version of the iterate from storage_type

template<*Mapping* **MutIter**>
static inline constexpr *storage_type*<*MutIter*> **provide_ptr**(*storage_type*<*MutIter*> storage)

return a pointer to the iterate from storage_type

template<*Mapping* **MutIter**>
static inline constexpr *Return_t*<*MutIter*> **from_data_ptr**(std::conditional_t<*MutIter* == *Mapping*::*Const*, const *T**, *T**> data)

return a return_type version of the iterate from its pointer

template<*Mapping* **MutIter**>
static inline constexpr *storage_type*<*MutIter*> **to_storage**(*ref_type*<*MutIter*> ref)

return a storage_type version of the iterate from its value

static inline constexpr *Dim_t* **stride**()

return the nb of components of the iterate (known at compile time)

static inline std::string **shape**()

return the iterate's shape as text, mostly for error messages

static inline constexpr *Dim_t* **NbRow**()

template<*Dim_t* **order**, *Dim_t* **dim**>

struct **SizesByOrder**

#include <eigen_tools.hh> Creates a Eigen::Sizes type for a Tensor defined by an order and dim.

Public Types

using **Sizes** = typename *internal*::*SizesByOrderHelper*<*order* - 1, *dim*, *dim*>::Sizes

`Eigen::Sizes`

template<*Dim_t* **order**, *Dim_t* **dim**, *Dim_t*... **dims**>

struct **SizesByOrderHelper**

#include <eigen_tools.hh> Creates a Eigen::Sizes type for a Tensor defined by an order and dim.

Public Types

using **Sizes** = typename *SizesByOrderHelper*<*order* - 1, *dim*, *dim*, *dims*...>::Sizes

type to use

template<*Dim_t* **dim**, *Dim_t*... **dims**>

struct **SizesByOrderHelper**<0, *dim*, *dims*...>

#include <eigen_tools.hh> Creates a Eigen::Sizes type for a Tensor defined by an order and dim.

Public Types

using **Sizes** = *Eigen*::Sizes<*dims*...>

type to use

template<class **Solver**>

struct **Solver_traits**

template<>

struct **Solver_traits**<*SolverBiCGSTABEigen*>

#include <solver_eigen.hh> traits for the Eigen BiCGSTAB solver

Public Types

using **Solver** = *Eigen*::BiCGSTAB<typename *Cell*::*Adaptor*, *Eigen*::IdentityPreconditioner>

Eigen Iterative Solver.

template<>

struct **Solver_traits**<*SolverCGEigen*>

#include <solver_eigen.hh> traits for the Eigen conjugate gradient solver

Public Types

using **Solver** = *Eigen*::ConjugateGradient<typename *Cell*::*Adaptor*, *Eigen*::Lower | *Eigen*::Upper, *Eigen*::IdentityPreconditioner>

Eigen Iterative Solver.

template<>

struct **Solver_traits**<*SolverDGMRESEigen*>

#include <solver_eigen.hh> traits for the Eigen DGMRES solver

Public Types

using **Solver** = *Eigen*::DGMRES<typename *Cell*::*Adaptor*, *Eigen*::IdentityPreconditioner>

Eigen Iterative Solver.

template<>

struct **Solver_traits**<*SolverGMRESEigen*>

#include <solver_eigen.hh> traits for the Eigen GMRES solver

Public Types

using **Solver** = *Eigen*::GMRES<typename *Cell*::*Adaptor*, *Eigen*::IdentityPreconditioner>

Eigen Iterative Solver.

template<>

struct **Solver_traits**<*SolverMINRESEigen*>

#include <solver_eigen.hh> traits for the Eigen MINRES solver

Public Types

using **Solver** = *Eigen*::MINRES<typename *Cell*::*Adaptor*, *Eigen*::Lower | *Eigen*::Upper, *Eigen*::IdentityPreconditioner>

Eigen Iterative Solver.

class **SolverBase**

#include <solver_base.hh> Virtual base class for solvers. An implementation of this interface can be used with the solution strategies in *solvers.hh*

Subclassed by *muSpectre::SolverCG*, *muSpectre::SolverEigen< SolverType >*, *muSpectre::SolverEigen< SolverBiCGSTABEigen >*, *muSpectre::SolverEigen< SolverCGEigen >*, *muSpectre::SolverEigen< SolverDGMRESEigen >*, *muSpectre::SolverEigen< SolverGMRESEigen >*, *muSpectre::SolverEigen< SolverMINRESEigen >*

Public Types

using **Vector_t** = *Eigen*::Matrix<Real, *Eigen*::Dynamic, 1>

underlying vector type

using **Vector_ref** = *Eigen*::Ref<*Vector_t*>

Input vector for solvers.

using **ConstVector_ref** = *Eigen*::Ref<const *Vector_t*>

Input vector for solvers.

using **Vector_map** = *Eigen*::Map<*Vector_t*>

Output vector for solvers.

Public Functions

SolverBase() = delete

Default constructor.

SolverBase(*Cell* &cell, Real tol, Uint maxiter, bool verbose = false)

Constructor takes a *Cell*, tolerance, max number of iterations and verbosity flag as input

SolverBase(const *SolverBase* &other) = delete

Copy constructor.

SolverBase(*SolverBase* &&other) = default

Move constructor.

virtual **~SolverBase**() = default

Destructor.

SolverBase &**operator=**(const *SolverBase* &other) = delete

Copy assignment operator.

SolverBase &**operator=**(*SolverBase* &&other) = delete

Move assignment operator.

virtual void **initialise**() = 0

Allocate fields used during the solution.

bool **has_converged**() const

returns whether the solver has converged

void **reset_counter**()

reset the iteration counter to zero

Uint **get_counter**() const

get the count of how many solve steps have been executed since construction of most recent counter reset

Uint **get_maxiter**() const

returns the max number of iterations

Real **get_tol**() const

returns the solving tolerance

virtual std::string **get_name**() const = 0

returns the solver's name (i.e. 'CG', 'GMRES', etc)

virtual *Vector_map* **solve**(const *ConstVector_ref* rhs) = 0

run the solve operation

Protected Attributes

Cell &**cell**

reference to the problem's cell

Real **tol**

convergence tolerance

Uint **maxiter**

maximum allowed number of iterations

bool **verbose**

whether to write information to the stdout

Uint **counter** = {0}

iteration counter

bool **converged** = {false}

whether the solver has converged

class **SolverBiCGSTABEigen** : public *muSpectre*::*SolverEigen*<*SolverBiCGSTABEigen*>

#include <solver_eigen.hh> Binding to Eigen's BiCGSTAB solver

Public Functions

inline virtual std::string **get_name**() const final

Solver's name.

class **SolverCG** : public *muSpectre*::*SolverBase*

#include <solver_cg.hh> implements the *muSpectre::SolverBase* interface using a conjugate gradient solver. This particular class is useful for trouble shooting, as it can be made very verbose, but for production runs, it is probably better to use *muSpectre::SolverCGEigen*.

Public Types

using **Parent** = *SolverBase*

standard short-hand for base class

using **Vector_t** = *Parent*::Vector_t

for storage of fields

using **Vector_ref** = *Parent*::Vector_ref

Input vector for solvers.

using **ConstVector_ref** = *Parent*::ConstVector_ref

Input vector for solvers.

using **Vector_map** = *Parent*::Vector_map

Output vector for solvers.

Public Functions

SolverCG() = delete

Default constructor.

SolverCG(const *SolverCG* &other) = delete

Copy constructor.

SolverCG(*Cell* &cell, Real tol, Uint maxiter, bool verbose = false)

Constructor takes a *Cell*, tolerance, max number of iterations and verbosity flag as input

SolverCG(*SolverCG* &&other) = default

Move constructor.

virtual **~SolverCG**() = default
Destructor.

SolverCG &**operator=**(const *SolverCG* &other) = delete
Copy assignment operator.

SolverCG &**operator=**(*SolverCG* &&other) = delete
Move assignment operator.

inline virtual void **initialise**() final
initialisation does not need to do anything in this case

inline virtual std::string **get_name**() const final
returns the solver's name

virtual *Vector_map* **solve**(const *ConstVector_ref* rhs) final
the actual solver

Protected Attributes

Vector_t **r_k**
residual

Vector_t **p_k**
search direction

Vector_t **Ap_k**
directional stiffness

Vector_t **x_k**
current solution

class **SolverCGEigen** : public *muSpectre*::*SolverEigen*<*SolverCGEigen*>
#include <solver_eigen.hh> Binding to Eigen's conjugate gradient solver

Public Functions

inline virtual std::string **get_name**() const final
returns the solver's name (i.e. 'CG', 'GMRES', etc)

class **SolverDGMRESEigen** : public *muSpectre*::*SolverEigen*<*SolverDGMRESEigen*>
#include <solver_eigen.hh> Binding to Eigen's DGMRES solver

Public Functions

inline virtual std::string **get_name**() const final

Solver's name.

template<class **SolverType**>

class **SolverEigen** : public *muSpectre*::*SolverBase*

#include <solver_eigen.hh> base class for iterative solvers from Eigen

Public Types

using **Parent** = *SolverBase*

base class

using **Solver** = typename *internal*::*Solver_traits*<*SolverType*>::Solver

traits obtained from CRTP

using **ConstVector_ref** = *Parent*::ConstVector_ref

Input vectors for solver.

using **Vector_map** = *Parent*::Vector_map

Output vector for solver.

using **Vector_t** = *Parent*::Vector_t

storage for output vector

Public Functions

SolverEigen() = delete

Default constructor.

SolverEigen(*Cell* &cell, Real tol, Uint maxiter = 0, bool verbose = false)

Constructor with cell and solver parameters.

SolverEigen(const *SolverEigen* &other) = delete

Copy constructor.

SolverEigen(*SolverEigen* &&other) = default

Move constructor.

virtual **~SolverEigen**() = default

Destructor.

SolverEigen &**operator=**(const *SolverEigen* &other) = delete

Copy assignment operator.

SolverEigen &**operator=**(*SolverEigen* &&other) = default

Move assignment operator.

virtual void **initialise**() final

Allocate fields used during the solution.

virtual *Vector_map* **solve**(const *ConstVector_ref* rhs) final

executes the solver

Protected Attributes

Cell::*Adaptor* **adaptor**

cell handle

Solver **solver**

Eigen's Iterative solver.

Vector_t **result**

storage for result

class **SolverError** : public runtime_error

Subclassed by *muSpectre::ConvergenceError*

class **SolverGMRESEigen** : public *muSpectre*::*SolverEigen*<*SolverGMRESEigen*>

#include <solver_eigen.hh> Binding to Eigen's GMRES solver

Public Functions

inline virtual std::string **get_name**() const final

returns the solver's name (i.e. 'CG', 'GMRES', etc)

class **SolverMINRESEigen** : public *muSpectre*::*SolverEigen*<*SolverMINRESEigen*>

#include <solver_eigen.hh> Binding to Eigen's MINRES solver

Public Functions

inline virtual std::string **get_name**() const final

Solver's name.

class **StateField**

#include <state_field.hh> Base class for state fields, useful for storing polymorphic references

Subclassed by *muGrid::TypedStateField< T >*, *muGrid::TypedStateField< Scalar >*

Public Functions

StateField() = delete

Default constructor.

StateField(const *StateField* &other) = delete

Copy constructor.

StateField(*StateField* &&other) = delete

Move constructor.

virtual **~StateField**() = default

Destructor.

StateField &**operator=**(const *StateField* &other) = delete

Copy assignment operator.

StateField &**operator=**(*StateField* &&other) = delete

Move assignment operator.

const *Dim_t* &**get_nb_memory**() const

returns number of old states that are stored

virtual const std::type_info &**get_stored_typeid**() const = 0

return type_id of stored type

void **cycle**()

cycle the fields (current becomes old, old becomes older, oldest becomes current)

Field &**current**()

return a reference to the field holding the current values

const *Field* &**current**() const

return a const reference to the field holding the current values

const *Field* &**old**(size_t nb_steps_ago = 1) const

return a reference to the field holding the values which were current `nb_steps_ago` ago

inline const std::vector<size_t> &**get_indices**() const

get the current ordering of the fields (inlineable because called in hot loop)

Protected Functions

StateField(const std::string &unique_prefix, *FieldCollection* &collection, *Dim_t* nb_memory = 1)

Protected constructor

Protected Attributes

std::string **prefix**

the unique prefix is used as the first part of the unique name of the subfields belonging to this state field

FieldCollection &**collection**

reference to the collection this statefield belongs to

const *Dim_t* **nb_memory**

number of old states to store, defaults to 1

std::vector<size_t> **indices** = {}

the current (historically accurate) ordering of the fields

RefVector<*Field*> **fields** = {}

storage of references to the diverse fields

template<typename **T**, *Mapping* **Mutability**>

class **StateFieldMap**

#include <state_field.hh> forward-declaration for friending

Dynamically sized map for iterating over *muGrid::StateField*s

Subclassed by *muGrid::StaticStateFieldMap< T, Mutability, MapType, NbMemory, IterationType >*

Public Types

using **FieldMap_t** = *FieldMap*<*T*, *Mutability*>

type for the current-values map (may be mutable, if the underlying field was)

using **CFieldMap_t** = *FieldMap*<*T*, *Mapping*::*Const*>

type for the old-values map, non-mutable

using **iterator** = Iterator<(*Mutability* == *Mapping*::*Mut*) ? *Mapping*::*Mut* : *Mapping*::*Const*>

stl

using **const_iterator** = Iterator<*Mapping*::*Const*>

stl

Public Functions

StateFieldMap() = delete

Default constructor.

StateFieldMap(*TypedStateField*<*T*> &state_field, *Iteration* iter_type = *Iteration*::*QuadPt*)

constructor from a state field. The default case is a map iterating over quadrature points with a matrix of shape (nb_components × 1) per field entry

StateFieldMap(*TypedStateField*<*T*> &state_field, *Dim_t* nb_rows, *Iteration* iter_type = *Iteration*::*QuadPt*)

Constructor from a state field with explicitly chosen shape of iterate. (the number of columns is inferred).

StateFieldMap(const *StateFieldMap* &other) = delete

StateFieldMap(*StateFieldMap* &&other) = delete

Move constructor.

virtual **~StateFieldMap**() = default

Destructor.

StateFieldMap &**operator=**(const *StateFieldMap* &other) = delete

Copy assignment operator.

StateFieldMap &**operator=**(*StateFieldMap* &&other) = delete

Move assignment operator.

iterator **begin**()

stl

iterator **end**()

stl

const *TypedStateField*<*T*> &**get_state_field**() const

return a const reference to the mapped state field

const *Dim_t* &**get_nb_rows**() const

return the number of rows the iterates have

size_t **size**() const

returns the number of iterates produced by this map (corresponds to the number of field entries if Iteration::Quadpt, or the number of pixels/voxels if Iteration::Pixel);

inline StateWrapper<*Mutability*> **operator[]**(size_t index)

random access operator

inline StateWrapper<*Mapping*::*Const*> **operator[]**(size_t index) const

random constaccess operator

FieldMap_t &**get_current**()

returns a reference to the map over the current data

const *FieldMap_t* &**get_current**() const

returns a const reference to the map over the current data

const *CFieldMap_t* &**get_old**(size_t nb_steps_ago) const

returns a const reference to the map over the data which was current `nb_steps_ago` ago

Protected Functions

RefVector<*Field*> &**get_fields**()

protected access to the constituent fields

std::vector<*FieldMap_t*> **make_maps**(RefVector<*Field*> &fields)

helper function creating the list of maps to store for current values

std::vector<*CFieldMap_t*> **make_cmaps**(RefVector<*Field*> &fields)

helper function creating the list of maps to store for old values

Protected Attributes

TypedStateField<*T*> &**state_field**

mapped state field. Needed for query at initialisations

const *Iteration* **iteration**

type of map iteration

const *Dim_t* **nb_rows**

number of rows of the iterate

std::vector<*FieldMap_t*> **maps**

maps over nb_memory + 1 possibly mutable maps. current points to one of these

std::vector<*CFieldMap_t*> **cmaps**

maps over nb_memory + 1 const maps. old(nb_steps_ago) points to one of these

template<*Mapping* **MutWrapper**>

class **StateWrapper**

#include <state_field_map.hh> The iterate needs to give access to current or previous values. This is handled by the *muGrid::StateFieldMap::StateWrapper*, a light-weight wrapper around the iterate's data.

Public Types

using **StateFieldMap_t** = std::conditional_t<*MutWrapper* == *Mapping*::*Const*, const StateFieldMap, StateFieldMap>

convenience alias

using **CurrentVal_t** = typename FieldMap_t::template Return_t<*MutWrapper*>

return value when getting current value from iterate

using **OldVal_t** = typename FieldMap_t::template Return_t<*Mapping*::*Const*>

return value when getting old value from iterate

Public Functions

inline **StateWrapper**(*StateFieldMap_t* &state_field_map, size_t index)

constructor (should never have to be called by user)

~StateWrapper() = default

inline *CurrentVal_t* &**current**()

return the current value at this iterate

inline const *OldVal_t* &**old**(size_t nb_steps_ago) const

return the value at this iterate which was current `nb_steps_ago` ago

Protected Attributes

CurrentVal_t **current_val**

current value at this iterate

std::vector<*OldVal_t*> **old_vals** = { }

all old values at this iterate

template<typename **T**, *Mapping* **Mutability**, class **MapType**, *Iteration* **IterationType** = *Iteration*::*QuadPt*>

class **StaticFieldMap** : public *muGrid*::*FieldMap*<*T*, *Mutability*>

#include <field_map_static.hh> Statically sized field map. Static field maps reproduce the capabilities of the (dynamically sized) *muGrid::FieldMap*, but iterate much more efficiently.

Public Types

using **Scalar** = *T*

stored scalar type

using **Parent** = *FieldMap*<*T*, *Mutability*>

base class

using **Field_t** = typename *Parent*::Field_t

convenience alias

using **Return_t** = typename *MapType*::template Return_t<MutType>

return type when dereferencing iterators over this map

using **reference** = *Return_t*<*Mutability*>

stl

using **PlainType** = typename *MapType*::PlainType

Eigen type representing iterates of this map.

using **Enumeration_t** = *akantu*::*containers*::*ZipContainer*<std::conditional_t<(*IterationType* == *Iteration*::*QuadPt*), *FieldCollection*::*IndexIterable*, *FieldCollection*::*PixelIndexIterable*>, *StaticFieldMap*&>

iterable proxy type to iterate over the quad point/pixel indices and stored values simultaneously

using **iterator** = Iterator<(*Mutability* == *Mapping*::*Mut*) ? *Mapping*::*Mut* : *Mapping*::*Const*>

stl

using **const_iterator** = Iterator<*Mapping*::*Const*>

stl

Public Functions

StaticFieldMap() = delete

Default constructor.

inline explicit **StaticFieldMap**(*Field* &field)

Constructor from a non-typed field ref (has more runtime cost than the next constructor

inline explicit **StaticFieldMap**(*Field_t* &field)

Constructor from typed field ref.

StaticFieldMap(const *StaticFieldMap* &other) = delete

Copy constructor.

StaticFieldMap(*StaticFieldMap* &&other) = default

Move constructor.

virtual **~StaticFieldMap**() = default

Destructor.

StaticFieldMap &**operator=**(const *StaticFieldMap* &other) = delete

Copy assignment operator.

StaticFieldMap &**operator=**(*StaticFieldMap* &&other) = delete

Move assignment operator.

template<bool **IsMutableField** = *Mutability* == *Mapping*::*Mut*>
inline std::enable_if_t<*IsMutableField*, *StaticFieldMap*> &**operator=**(const typename *Parent*::EigenRef &val)

Assign a matrix-like value with dynamic size to every entry.

template<bool **IsMutableField** = *Mutability* == *Mapping*::*Mut*>
inline std::enable_if_t<*IsMutableField* && !*MapType*::IsScalarMapType(), *StaticFieldMap*<*T*, *Mutability*, *MapType*, *IterationTyp*

Assign a matrix-like value with static size to every entry.

template<bool **IsMutableField** = *Mutability* == *Mapping*::*Mut*>
inline std::enable_if_t<*IsMutableField* && *MapType*::IsScalarMapType(), *StaticFieldMap*<*T*, *Mutability*, *MapType*, *IterationTyp*

Assign a scalar value to every entry.

inline *Return_t*<*Mutability*> **operator[]**(size_t index)

random access operator

inline *Return_t*<*Mapping*::*Const*> **operator[]**(size_t index) const

random const access operator

inline *PlainType* **mean**() const

evaluate the average of the field

inline *iterator* **begin**()

stl

inline *iterator* **end**()

stl

inline *const_iterator* **begin**() const

stl

inline *const_iterator* **end**() const

stl

template<bool **IsPixelIterable** = (*IterationType* == *Iteration*::*Pixel*)>
inline std::enable_if_t<*IsPixelIterable*, *Enumeration_t*> **enumerate_indices**()

iterate over pixel/quad point indices and stored values simultaneously

template<*Iteration* **Iter** = *Iteration*::*QuadPt*, class **Dummy** = std::enable_if_t<*IterationType* == *Iter*, bool>>
inline *Enumeration_t* **enumerate_indices**()

iterate over pixel/quad point indices and stored values simultaneously

Public Static Functions

static inline constexpr *Iteration* **GetIterationType**()

determine at compile time whether pixels or quadrature points are iterater over

static inline constexpr size_t **Stride**()

determine the number of components in the iterate at compile time

static inline constexpr bool **IsStatic**()

determine whether this map has statically sized iterates at compile time

template<typename **T**, *Mapping* **Mutability**, class **MapType**, size_t **NbMemory**, *Iteration* **IterationType** = *Iteration*::*QuadPt*>
class **StaticStateFieldMap** : public *muGrid*::StateFieldMap<*T*, *Mutability*>

#include <state_field_map_static.hh> statically sized version of *muGrid::TypedStateField*. Duplicates its capabilities, with much more efficient statically sized iterates.

Public Types

using **Scalar** = *T*

stored scalar type

using **Parent** = StateFieldMap<*T*, *Mutability*>

base class

using **StaticFieldMap_t** = StaticFieldMap<*T*, *Mutability*, *MapType*, *IterationType*>

convenience alias for current map

using **CStaticFieldMap_t** = StaticFieldMap<*T*, *Mapping*::*Const*, *MapType*, *IterationType*>

convenience alias for old map

using **MapArray_t** = std::array<*StaticFieldMap_t*, *NbMemory* + 1>

storage type for current maps

using **CMapArray_t** = std::array<*CStaticFieldMap_t*, *NbMemory* + 1>

storage type for old maps

using **iterator** = Iterator<(*Mutability* == *Mapping*::*Mut*) ? *Mapping*::*Mut* : *Mapping*::*Const*>

stl

using **const_iterator** = Iterator<*Mapping*::*Const*>

stl

Public Functions

StaticStateFieldMap() = delete

Deleted default constructor.

inline explicit **StaticStateFieldMap**(*TypedStateField*<*T*> &state_field)

constructor from a state field. The default case is a map iterating over quadrature points with a matrix of shape (nb_components × 1) per field entry

StaticStateFieldMap(const *StaticStateFieldMap* &other) = delete

Deleted copy constructor.

StaticStateFieldMap(*StaticStateFieldMap* &&other) = default

Move constructor.

virtual **~StaticStateFieldMap**() = default

Destructor.

StaticStateFieldMap &**operator=**(const *StaticStateFieldMap* &other) = delete

Copy assignment operator.

StaticStateFieldMap &**operator=**(*StaticStateFieldMap* &&other) = default

Move assignment operator.

inline *iterator* **begin**()

stl

inline *iterator* **end**()

stl

inline const *CStaticFieldMap_t* &**get_old_static**(size_t nb_steps_ago) const

return a const ref to an old value map

inline *StaticFieldMap_t* &**get_current_static**()

return a ref to an the current map

inline *StaticFieldMap_t* &**get_current**()

inline *StaticFieldMap_t* &**get_current_static**() const

return a const ref to an the current map

inline *StaticFieldMap_t* &**get_current**() const

inline StaticStateWrapper<*Mutability*> **operator[]**(size_t index)

random access operator

inline StaticStateWrapper<*Mapping*::*Const*> **operator[]**(size_t index) const

random const access operator

Public Static Functions

static inline constexpr size_t **GetNbMemory**()

determine at compile time the number of old values stored

static inline constexpr *Mapping* **FieldMutability**()

determine the map's mutability at compile time

static inline constexpr *Iteration* **GetIterationType**()

determine the map's iteration type (pixels vs quad pts) at compile time

Protected Types

using **HelperRet_t** = std::conditional_t<MutIter == *Mapping*::*Const*, *CMapArray_t*, *MapArray_t*>

internal convenience alias

Protected Functions

template<*Mapping* **MutIter**, size_t... **I**>
inline auto **map_helper**(std::index_sequence<*I*...>) -> *HelperRet_t*<*MutIter*>

helper for building the maps

inline *MapArray_t* **make_maps**()

build the current value maps

inline *CMapArray_t* **make_cmaps**()

build the old value maps

Protected Attributes

MapArray_t **static_maps**

container for current maps

CMapArray_t **static_cmaps**

container for old maps

template<*Mapping* **MutWrapper**>

class **StaticStateWrapper**

#include <state_field_map_static.hh> The iterate needs to give access to current or previous values. This is handled by the `muGrid::StaticStateFieldMap::StateWrapper`, a light-weight wrapper around the iterate's data.

Template Parameters

MutWrapper – mutability of the mapped field. It should never be necessary to set this manually, rather the iterators dereference operator*() should return the correct type.

Public Types

using **StaticStateFieldMap_t** = std::conditional_t<*MutWrapper* == *Mapping*::*Const*, const StaticStateFieldMap, StaticStateFieldMap>

const-correct map

using **CurrentVal_t** = typename MapType::template ref_type<*MutWrapper*>

return type handle for current value

using **CurrentStorage_t** = typename MapType::template storage_type<*MutWrapper*>

storage type for current value handle

using **OldVal_t** = typename MapType::template ref_type<*Mapping*::*Const*>

return type handle for old value

using **OldStorage_t** = typename MapType::template storage_type<*Mapping*::*Const*>

storage type for old value handle

Public Functions

inline **StaticStateWrapper**(*StaticStateFieldMap_t* &state_field_map, size_t index)

constructor with map and index, not for user to call

~StaticStateWrapper() = default

inline *CurrentVal_t* &**current**()

return the current value of the iterate

inline const *OldVal_t* &**old**(size_t nb_steps_ago) const

return the value of the iterate which was current `nb_steps_ago` steps ago. Possibly has excess runtime cost compared to the next function, and has no bounds checking, unlike the next function

template<size_t **NbStepsAgo** = 1>
inline const *OldVal_t* &**old**() const

return the value of the iterate which was current `NbStepsAgo` steps ago

Protected Functions

inline std::array<*OldStorage_t*, NbMemory> **make_old_vals_static**(*StaticStateFieldMap_t* &state_field_map, size_t index)

helper function to build the list of old values

template<size_t... **NbStepsAgo**>
inline std::array<*OldStorage_t*, NbMemory> **old_vals_helper_static**(*StaticStateFieldMap_t* &state_field_map, size_t index, std::index_sequence<*NbStepsAgo*...>)

helper function to build the list of old values

Protected Attributes

CurrentStorage_t **current_val**

handle to current value

std::array<*OldStorage_t*, NbMemory> **old_vals** = {}

storage for handles to old values

template<Dim_t **DimM**, *StrainMeasure* **StrainM**, *StressMeasure* **StressM**>

class **STMaterialLinearElasticGeneric1** : public *muSpectre*::*MaterialMuSpectre*<*STMaterialLinearElasticGeneric1*<*DimM*, *StrainM*, *StressM*>, *DimM*>

#include <s_t_material_linear_elastic_generic1.hh> forward declaration

Linear elastic law defined by a full stiffness tensor with the ability to compile and work for different strain/stress measures

Public Types

using **Parent** = *MaterialMuSpectre*<*STMaterialLinearElasticGeneric1*<*DimM*, *StrainM*, *StressM*>, *DimM*>

base class:

using **CInput_t** = *Eigen*::Ref<*Eigen*::MatrixXd>

using **Strain_t** = *Eigen*::Matrix<Real, *DimM*, *DimM*>

using **Stress_t** = *Eigen*::Matrix<Real, *DimM*, *DimM*>

using **Stiffness_t** = *muGrid*::*T4Mat*<Real, *DimM*>

using **traits** = *MaterialMuSpectre_traits*<*STMaterialLinearElasticGeneric1*<*DimM*, *StrainM*, *StressM*>>

traits of this material

using **Material_sptr** = std::shared_ptr<*STMaterialLinearElasticGeneric1*>

Public Functions

STMaterialLinearElasticGeneric1() = delete

Default constructor.

STMaterialLinearElasticGeneric1(const std::string &name, const Dim_t &spatial_dimension, const Dim_t &nb_quad_pts, const *CInput_t* &C_voigt)

Constructor by name and stiffness tensor.

Parameters

- **name** – unique material name
- **spatial_dimension** – spatial dimension of the problem. This corresponds to the dimensionality of the *Cell*

- **nb_quad_pts** – number of quadrature points per pixel
- **C_voigt** – elastic tensor in Voigt notation

STMaterialLinearElasticGeneric1(const *STMaterialLinearElasticGeneric1* &other) = delete

Copy constructor.

STMaterialLinearElasticGeneric1(*STMaterialLinearElasticGeneric1* &&other) = default

Move constructor.

virtual **~STMaterialLinearElasticGeneric1**() = default

Destructor.

STMaterialLinearElasticGeneric1 &**operator=**(const *STMaterialLinearElasticGeneric1* &other) = delete

Copy assignment operator.

STMaterialLinearElasticGeneric1 &**operator=**(*STMaterialLinearElasticGeneric1* &&other) = delete

Move assignment operator.

template<class **Derived**>
inline *Stress_t* **evaluate_stress**(const *Eigen*::MatrixBase<*Derived*> &E, const size_t &quad_pt_index = 0)

evaluates stress given the strain

template<class **Derived**>
inline std::tuple<*Stress_t*, *Stiffness_t*> **evaluate_stress_tangent**(const *Eigen*::MatrixBase<*Derived*> &strain, const size_t &quad_pt_index = 0)

evaluates both stress and stiffness given the strain

inline void **set_F**(const *Strain_t* &Finp)

inline *Stiffness_t* **get_C**()

template<class **Derived**>
auto **evaluate_stress**(const *Eigen*::MatrixBase<*Derived*> &strain, const size_t&) -> *Stress_t*

template<class **Derived**>
auto **evaluate_stress_tangent**(const *Eigen*::MatrixBase<*Derived*> &strain, const size_t&) -> std::tuple<*Stress_t*, *Stiffness_t*>

Public Static Functions

static std::tuple<*Material_sptr*, *MaterialEvaluator*<*DimM*>> **make_evaluator**(const *CInput_t* &C_voigt)

Factory.

Protected Attributes

std::unique_ptr<*Stiffness_t*> **C_holder**

const *Stiffness_t* &**C**

stiffness tensor

std::unique_ptr<*Strain_t*> **F_holder**

Strain_t &**F**

bool **F_is_set**

template<class **Dummy**>

struct **StrainsTComputer**

template<class **StrainMap_t**>

struct **StrainsTComputer**<std::tuple<*StrainMap_t*>>

Public Types

using **type** = std::tuple<typename *StrainMap_t*::reference>

template<class **StrainMap_t**>

struct **StrainsTComputer**<std::tuple<*StrainMap_t*, *StrainMap_t*>>

Public Types

using **type** = std::tuple<typename *StrainMap_t*::reference, typename *StrainMap_t*::reference>

template<class **Dummy**>

struct **StressesTComputer**

template<class **StressMap_t**>

struct **StressesTComputer**<std::tuple<*StressMap_t*>>

Public Types

using **type** = std::tuple<typename *StressMap_t*::reference>

template<class **StressMap_t**, class **TangentMap_t**>

struct **StressesTComputer**<std::tuple<*StressMap_t*, *TangentMap_t*>>

Public Types

using **type** = std::tuple<typename *StressMap_t*::reference, typename *TangentMap_t*::reference>

template<*Dim_t* **dim**, *Dim_t* **i** = *dim* - 1>

struct **Summand**

#include <eigen_tools.hh> sum term

Public Static Functions

static inline decltype(auto) **compute**(const *Vec_t*<*dim*> &eigs, const *Mat_t*<*dim*> &T)
wrapped function (raison d'être)

template<*Dim_t* **dim**>

struct **Summand**<*dim*, 0>

#include <eigen_tools.hh> sum term

Public Static Functions

static inline decltype(auto) **compute**(const *Vec_t*<*dim*> &eigs, const *Mat_t*<*dim*> &T)
wrapped function (raison d'être)

Public Static Attributes

static constexpr *Dim_t* **i** = {0}
short-hand

template<class **Derived**>

struct **tensor_4_dim**

#include <eigen_tools.hh> computes the dimension from a fourth order tensor represented by a square matrix

Public Types

using **T** = std::remove_reference_t<*Derived*>
raw type for testing

Public Static Attributes

static constexpr *Dim_t* **value** = {*ct_sqrt*(*T*::RowsAtCompileTime)}
evaluated dimension

template<class **Derived**>

struct **tensor_dim**

#include <eigen_tools.hh> computes the dimension from a second order tensor represented square matrix or array

Public Types

using **T** = std::remove_reference_t<*Derived*>

raw type for testing

Public Static Attributes

static constexpr *Dim_t* **value** = {*T*::RowsAtCompileTime}

evaluated dimension

template<class **Derived**, *Dim_t* **Dim**>

struct **tensor_rank**

#include <eigen_tools.hh> computes the rank of a tensor given the spatial dimension

Public Types

using **T** = std::remove_reference_t<*Derived*>

Public Static Attributes

static constexpr *Dim_t* **value**{*internal*::*get_rank*<*Dim*, *T*::RowsAtCompileTime, *T*::ColsAtCompileTime>()}

template<class **Cell**>

struct **traits**<*muSpectre*::*CellAdaptor*<*Cell*>> : public *Eigen*::*internal*::traits<*Eigen*::SparseMatrix<*Real*>>, public *Eigen*::*internal*::traits<*Eigen*::SparseMatrix<*Real*>>

template<class **OutType**>

struct **TupleBuilder**

Public Static Functions

template<class ...**InTypes**, size_t... **I**>
static inline *OutType* **helper**(std::tuple<*InTypes*...> const &arg, std::index_sequence<*I*...>)

template<class ...**InTypes**>
static inline *OutType* **build**(std::tuple<*InTypes*...> const &arg)

template<typename **T**, typename **FirstVal**, typename ...**RestVals**>

struct **TypeChecker**

#include <ref_array.hh> Struct user for checking that every member of a parameter pack has type T

Public Static Attributes

static constexpr bool value {std::is_same<T, std::remove_reference_t<FirstVal>>::value andTypeChecker<T, RestVals...>::value}

whether the check passed

template<typename **T**, typename **OnlyVal**>

struct **TypeChecker**<*T*, *OnlyVal*>

#include <ref_array.hh> Specialisation for recursion tail

Public Static Attributes

static constexpr bool **value**{std::is_same<*T*, std::remove_reference_t<*OnlyVal*>>::value}

whether the check passed

template<typename **T**>

class **TypedField** : public *muGrid*::TypedFieldBase<*T*>

#include <field_collection.hh> forward declaration of the `muSpectre::TypedField`

forward declaration of the *`muGrid::TypedField`*

A *`muGrid::TypedField`* holds a certain number of components (scalars of type `T` per quadrature point of a *`muGrid::FieldCollection`*'s domain.

Template Parameters

T – type of scalar to hold. Must be one of `muGrid::Real`, `muGrid::Int`, `muGrid::Uint`, `muGrid::Complex`.

Public Types

using **Parent** = TypedFieldBase<*T*>

base class

using **EigenRep_t** = typename *Parent*::EigenRep_t

Eigen type to represent the field's data.

using **Negative** = typename *Parent*::Negative

convenience alias

Public Functions

TypedField() = delete

Default constructor.

TypedField(*TypedField* &&other) = delete

Copy constructor.

Move constructor

virtual **~TypedField**() = default

Destructor.

TypedField &**operator=**(*TypedField* &&other) = delete

Move assignment operator.

TypedField &**operator=**(const *Parent* &other)

Copy assignment operator.

TypedField &**operator=**(const *Negative* &other)

Copy assignment operator.

TypedField &**operator=**(const *EigenRep_t* &other)

Copy assignment operator.

virtual void **set_zero**() final

initialise field to zero (do more complicated initialisations through fully typed maps)

virtual void **set_pad_size**(size_t pad_size) final

add a pad region to the end of the field buffer; required for using this as e.g. an FFT workspace

virtual size_t **buffer_size**() const final

size of the internal buffer including the pad region (in scalars)

void **push_back**(const *T* &value)

add a new scalar value at the end of the field (incurs runtime cost, do not use this in any hot loop)

void **push_back**(const *Eigen*::Ref<const *Eigen*::Array<*T*, *Eigen*::Dynamic, *Eigen*::Dynamic>> &value)

add a new non-scalar value at the end of the field (incurs runtime cost, do not use this in any hot loop)

Public Members

friend FieldCollection

give access to collections

Public Static Functions

static *TypedField* &**safe_cast**(*Field* &other)

cast a reference to a base type to this type, with full checks

static const *TypedField* &**safe_cast**(const *Field* &other)

cast a const reference to a base type to this type, with full checks

static *TypedField* &**safe_cast**(*Field* &other, const *Dim_t* &nb_components)

cast a reference to a base type to this type safely, plus check whether it has the right number of components

static const *TypedField* &**safe_cast**(const *Field* &other, const *Dim_t* &nb_components)

cast a const reference to a base type to this type safely, plus check whether it has the right number of components

Protected Functions

inline **TypedField**(const std::string &unique_name, *FieldCollection* &collection, *Dim_t* nb_components)

*Field*s are supposed to only exist in the form of `std::unique_ptr`s held by a *FieldCollection*. The *Field* constructor is protected to ensure this.

Parameters

- **unique_name** – unique field name (unique within a collection)
- **nb_components** – number of components to store per quadrature point
- **collection** – reference to the holding field collection.

virtual void **resize**(size_t size) final

resizes the field to the given size

Protected Attributes

std::vector<*T*> **values** = {}

storage of the raw field data

template<typename **T**>

class **TypedFieldBase** : public *muGrid*::*Field*

#include <field_typed.hh> forward declaration

Subclassed by *muGrid::TypedField< T >*, *muGrid::WrappedField< T >*, *muGrid::TypedField< Scalar >*

Public Types

using **Scalar** = *T*

stored scalar type

using **EigenRep_t** = *Eigen*::Matrix<*T*, *Eigen*::Dynamic, *Eigen*::Dynamic>

Eigen type used to represent the field's data.

using **Eigen_map** = *Eigen*::Map<*EigenRep_t*>

eigen map (handle for EigenRep_t)

using **Eigen_cmap** = *Eigen*::Map<const *EigenRep_t*>

eigen const map (handle for EigenRep_t)

using **Parent** = *Field*

base class

Public Functions

TypedFieldBase() = delete

Default constructor.

TypedFieldBase(const *TypedFieldBase* &other) = delete

Copy constructor.

TypedFieldBase(*TypedFieldBase* &&other) = default

Move constructor.

virtual **~TypedFieldBase**() = default

Destructor.

TypedFieldBase &**operator=**(*TypedFieldBase* &&other) = delete

Move assignment operator.

TypedFieldBase &**operator=**(const *TypedFieldBase* &other)

Copy assignment operator.

TypedFieldBase &**operator=**(const Negative &other)

Copy assignment operator.

TypedFieldBase &**operator=**(const *EigenRep_t* &other)

Copy assignment operators.

Negative **operator-**() const

Unary negative.

TypedFieldBase &**operator+=**(const *TypedFieldBase* &other)

addition assignment

TypedFieldBase &**operator-=**(const *TypedFieldBase* &other)

subtraction assignment

inline virtual const std::type_info &**get_stored_typeid**() const final

return the type information of the stored scalar (for compatibility checking)

Eigen_map **eigen_vec**()

return a vector map onto the underlying data

Eigen_cmap **eigen_vec**() const

return a const vector map onto the underlying data

Eigen_map **eigen_quad_pt**()

return a matrix map onto the underlying data with one column per quadrature point

Eigen_cmap **eigen_quad_pt**() const

return a const matrix map onto the underlying data with one column per quadrature point

Eigen_map **eigen_pixel**()

return a matrix map onto the underlying data with one column per pixel

Eigen_cmap **eigen_pixel**() const

return a const matrix map onto the underlying data with one column per pixel

FieldMap<*T*, *Mapping*::*Mut*> **get_pixel_map**(const *Dim_t* &nb_rows = *Unknown*)

convenience function returns a map of this field, iterable per pixel.

Parameters
nb_rows – optional specification of the number of rows for the iterate. If left to default value, a matrix of shape `nb_components` × `nb_quad_pts` is used

FieldMap<*T*, *Mapping*::*Const*> **get_pixel_map**(const *Dim_t* &nb_rows = *Unknown*) const

convenience function returns a const map of this field, iterable per pixel.

Parameters
nb_rows – optional specification of the number of rows for the iterate. If left to default value, a matrix of shape `nb_components` × `nb_quad_pts` is used

FieldMap<*T*, *Mapping*::*Mut*> **get_quad_pt_map**(const *Dim_t* &nb_rows = *Unknown*)

convenience function returns a map of this field, iterable per quadrature point.

Parameters
nb_rows – optional specification of the number of rows for the iterate. If left to default value, a column vector is used

FieldMap<*T*, *Mapping*::*Const*> **get_quad_pt_map**(const *Dim_t* &nb_rows = *Unknown*) const

convenience function returns a const map of this field, iterable per quadrature point.

Parameters
nb_rows – optional specification of the number of rows for the iterate. If left to default value, a column vector is used

T ***data**() const

get the raw data ptr. don't use unless interfacing with external libs

Protected Functions

inline **TypedFieldBase**(const std::string &unique_name, *FieldCollection* &collection, *Dim_t* nb_components)

*Field*s are supposed to only exist in the form of `std::unique_ptr`s held by a *FieldCollection*. The*Field*`constructor is protected to ensure this. Fields are instantiated through the`register_field` methods *FieldCollection*.

Parameters

- **unique_name** – unique field name (unique within a collection)
- **nb_components** – number of components to store per quadrature point
- **collection** – reference to the holding field collection.

Eigen_map **eigen_map**(const *Dim_t* &nb_rows, const *Dim_t* &nb_cols)

back-end for the public non-const eigen_XXX functions

Eigen_cmap **eigen_map**(const *Dim_t* &nb_rows, const *Dim_t* &nb_cols) const

back-end for the public const eigen_XXX functions

void **set_data_ptr**(*T* *ptr)

set the data_ptr

Protected Attributes

T ***data_ptr** = {}

in order to accomodate both registered fields (who own and manage their data) and unregistered temporary field proxies (piggy-backing on a chunk of existing memory as e.g., a numpy array) *efficiently*, the `get_ptr_to_entry` methods need to be branchless. this means that we cannot decide on the fly whether to return pointers pointing into values or into alt_values, we need to maintain an (shudder) raw data pointer that is set either at construction (for unregistered fields) or at any resize event (which may invalidate existing pointers). For the coder, this means that they need to be absolutely vigilant that *any* operation on the values vector that invalidates iterators needs to be followed by an update of data_ptr, or we will get super annoying memory bugs.

Friends

friend class FieldMap

template<typename **T**>

class **TypedStateField** : public *muGrid*::*StateField*

#include <field_collection.hh> forward declaration of the state field

forward declaration

The *TypedStateField* class is a byte compatible daughter class of the *StateField* class, and it can return fully typed *Field* references.

Public Types

using **Parent** = *StateField*

base class

Public Functions

TypedStateField() = delete

Deleted default constructor.

TypedStateField(const *TypedStateField* &other) = delete

Copy constructor.

TypedStateField(*TypedStateField* &&other) = delete

Move constructor.

virtual **~TypedStateField**() = default

Destructor.

TypedStateField &**operator=**(const *TypedStateField* &other) = delete

Copy assignment operator.

TypedStateField &**operator=**(*TypedStateField* &&other) = delete

Move assignment operator.

virtual const std::type_info &**get_stored_typeid**() const final

return type_id of stored type

TypedField<*T*> &**current**()

return a reference to the current field

const *TypedField*<*T*> &**current**() const

return a const reference to the current field

const *TypedField*<*T*> &**old**(size_t nb_steps_ago = 1) const

return a const reference to the field which was current `nb_steps_ago` steps ago

Protected Functions

TypedStateField(const std::string &unique_prefix, *FieldCollection* &collection, *Dim_t* nb_memory, *Dim_t* nb_components)

protected constructor, to avoid the creation of unregistered fields. Users should create fields through the *muGrid::FieldCollection::register_real_field()* (or `int`, `uint`, `compplex`) factory functions.

RefVector<*Field*> &**get_fields**()

return a reference to the storage of the constituent fields

Protected Attributes

friend FieldCollection

give access to the protected state field constructor

Friends

friend class StateFieldMap< T, Mapping::Const >

friend class StateFieldMap< T, Mapping::Mut >

class **Vectors_t**

Public Functions

inline explicit **Vectors_t**(const Dim_t &dim)

constructor

inline **Vectors_t**(const std::vector<Real> &data, const Dim_t &dim)

constructor

inline *Eigen*::Map<const *Vector_t*> **operator[]**(const Dim_t &id) const

access operator:

inline *Eigen*::Map<*Vector_t*> **operator[]**(const Dim_t &id)

access operator:

template<Dim_t **DimS**>

inline *Eigen*::Map<*Eigen*::Matrix<Real, *DimS*, 1>> **at**(const Dim_t &id)
access to staic sized map of the vectors:

inline void **push_back**(const *Vector_t* &vector)
push back for adding new vector to the data of the class

inline void **push_back**(const *Eigen*::Map<*Vector_t*, 0> &vector)
push back for adding new vector to the data of the class

inline void **push_back**(const *Eigen*::Map<const *Vector_t*, 0> &vector)
push back for adding new vector to the data of the class

inline void **push_back**(const *DynRcoord_t* &vector)
push back for adding new vector from DynRcoord

inline std::vector<Real> **get_a_vector**(const Dim_t &id)

inline const Dim_t &**get_dim**()

inline *iterator* **begin**()

inline *iterator* **end**()

inline size_t **size**() const

Protected Attributes

std::vector<Real> **data** = { }

Dim_t **dim**

Private Types

using **Vector_t** = *Eigen*::Matrix<Real, *Eigen*::Dynamic, 1>

template<Dim_t **dim**>

class **VoigtConversion**

#include <voigt_conversion.hh> implements a bunch of static functions to convert between full and Voigt notation of tensors

Public Functions

VoigtConversion()

template<>
inline auto **get_sym_mat**() -> decltype(auto)
voigt vector indices for symmetric tensors

template<>
inline auto **get_sym_mat**() -> decltype(auto)

template<>

inline auto **get_sym_mat**() -> decltype(auto)

template<>
inline auto **get_mat**() -> decltype(auto)

voigt vector indices for non_symmetric tensors

template<>
inline auto **get_mat**() -> decltype(auto)

template<>
inline auto **get_mat**() -> decltype(auto)

template<>
inline auto **get_vec**() -> decltype(auto)

matrix indices from voigt vectors

template<>
inline auto **get_vec**() -> decltype(auto)

template<>
inline auto **get_vec**() -> decltype(auto)

template<>
inline auto **get_factors**() -> decltype(auto)

template<>
inline auto **get_factors**() -> decltype(auto)

template<>
inline auto **get_factors**() -> decltype(auto)

template<>
inline auto **get_vec_vec**() -> decltype(auto)

reordering between a row/column in voigt vs col-major matrix (e.g., stiffness tensor)

template<>
inline auto **get_vec_vec**() -> decltype(auto)

template<>
inline auto **get_vec_vec**() -> decltype(auto)

Public Static Functions

template<class **Tens4**, class **Voigt**, bool **sym** = true>
static inline void **fourth_to_voigt**(const *Tens4* &t, *Voigt* &v)

obtain a fourth order voigt matrix from a tensor

template<class Tens4, bool sym = true> static inline Eigen::Matrix< Real, vsize< sym >dim), vsize< sym >dim)> fourth_to_voigt (const Tens4 &t)

return a fourth order voigt matrix from a tensor

template<class Tens4> static inline Eigen::Matrix< Real, vsize< false >dim), vsize< false >dim)> fourth_to_2d (const Tens4 &t)

return a fourth order non-symmetric voigt matrix from a tensor

template<class **Tens2**, class **Voigt**, bool **sym** = true>

static inline void **second_to_voigt**(const *Tens2* &t, *Voigt* &v)

probably obsolete

template<class **Tens2**, class **Voigt**>
static inline void **gradient_to_voigt_strain**(const *Tens2* &F, *Voigt* &v)

probably obsolete

template<class **Tens2**, class **Voigt**>
static inline void **gradient_to_voigt_GreenLagrange_strain**(const *Tens2* &F, *Voigt* &v)

probably obsolete

template<class **Tens2**, class **Voigt**, bool **sym** = true>
static inline void **stress_from_voigt**(const *Voigt* &v, *Tens2* &sigma)

probably obsolete

static inline auto **get_mat**() -> decltype(auto)

static inline auto **get_sym_mat**() -> decltype(auto)

static inline auto **get_vec**() -> decltype(auto)

static inline auto **get_factors**() -> decltype(auto)

static inline auto **get_vec_vec**() -> decltype(auto)

Private Functions

template<> const Eigen::Matrix< Dim_t, 1, 1 > mat

voigt vector indices for non-symmetric tensors

template<> const Eigen::Matrix< Dim_t, 2, 2 > mat

voigt vector indices for non-symmetric tensors

template<> const Eigen::Matrix< Dim_t, 3, 3 > mat

voigt vector indices for non-symmetric tensors

template<> const Eigen::Matrix< Dim_t, 1, 1 > sym_mat

voigt vector indices

template<> const Eigen::Matrix< Dim_t, 2, 2 > sym_mat

voigt vector indices

template<> const Eigen::Matrix< Dim_t, 3, 3 > sym_mat

voigt vector indices

template<> const Eigen::Matrix< Dim_t, 1 *1, 2 > vec

matrix indices from voigt vectors

template<> const Eigen::Matrix< Dim_t, 2 *2, 2 > vec

matrix indices from voigt vectors

template<> const Eigen::Matrix< Dim_t, 3 *3, 2 > vec

matrix indices from voigt vectors

template<> const Eigen::Matrix< Real, vsize(1), 1 > factors

factors for shear components in voigt notation

template<> const Eigen::Matrix< Real, vsize(2), 1 > factors

factors for shear components in voigt notation

template<> const Eigen::Matrix< Real, vsize(3), 1 > factors

factors for shear components in voigt notation

template<> const Eigen::Matrix< Dim_t, 1 *1, 1 > vec_vec

reordering between a row/column in voigt vs col-major matrix (e.g., stiffness tensor)

template<> const Eigen::Matrix< Dim_t, 2 *2, 1 > vec_vec

template<> const Eigen::Matrix< Dim_t, 3 *3, 1 > vec_vec

Private Static Attributes

static const *Eigen*::Matrix<Dim_t, *dim*, *dim*> **mat**

matrix of vector index I as function of tensor indices i,j

static const *Eigen*::Matrix<Dim_t, *dim*, *dim*> **sym_mat**

matrix of vector index I as function of tensor indices i,j

static const *Eigen*::Matrix<Dim_t, *dim* * *dim*, 2> **vec**

array of matrix indices ij as function of vector index I

static const *Eigen*::Matrix<Real, *vsize*(*dim*), 1> **factors**

factors to multiply the strain by for voigt notation

static const *Eigen*::Matrix<Dim_t, *dim* * *dim*, 1> **vec_vec**

reordering between a row/column in voigt vs col-major matrix (e.g., stiffness tensor)

template<typename **T**>

class **WrappedField** : public *muGrid*::TypedFieldBase<*T*>

#include <field_typed.hh> Wrapper class providing a field view of existing memory. This is particularly useful when dealing with input from external libraries (e.g., numpy arrays)

Public Types

using **Parent** = TypedFieldBase<*T*>

base class

using **EigenRep_t** = typename *Parent*::EigenRep_t

convenience alias to the Eigen representation of this field's data

Public Functions

WrappedField(const std::string &unique_name, *FieldCollection* &collection, *Dim_t* nb_components, size_t size, *T* *ptr)

constructor from a raw pointer. Typically, this would be a reference to a numpy array from the python bindings.

WrappedField(const std::string &unique_name, *FieldCollection* &collection, *Dim_t* nb_components, *Eigen*::Ref<*EigenRep_t*> values)

constructor from an eigen array ref.

WrappedField() = delete

Default constructor.

WrappedField(const *WrappedField* &other) = delete

Copy constructor.

WrappedField(*WrappedField* &&other) = default

Move constructor.

virtual **~WrappedField**() = default

Destructor.

WrappedField &**operator=**(const *WrappedField* &other) = delete

Copy assignment operator.

WrappedField &**operator=**(*WrappedField* &&other) = delete

Move assignment operator.

virtual void **set_zero**() final

initialise field to zero (do more complicated initialisations through fully typed maps)

virtual void **set_pad_size**(size_t pad_size) final

add a pad region to the end of the field buffer; required for using this as e.g. an FFT workspace

virtual size_t **buffer_size**() const final

size of the internal buffer including the pad region (in scalars)

Public Members

friend FieldCollection

give access to collections

Public Static Functions

static std::unique_ptr<const *WrappedField*> **make_const**(const std::string &unique_name, *FieldCollection* &collection, *Dim_t* nb_components, const *Eigen*::Ref<const *EigenRep_t*> values)

Emulation of a const constructor.

Protected Functions

virtual void **resize**(size_t size) final

resizes the field to the given size

Protected Attributes

size_t **size**

size of the wrapped buffer

template<class ...**Containers**>

class **ZipContainer**

#include <iterators.hh> helper for the emulation of python zip

Public Functions

inline explicit **ZipContainer**(*Containers*&&... containers)

undocumented

inline decltype(auto) **begin**() const

undocumented

inline decltype(auto) **end**() const

undocumented

inline decltype(auto) **begin**()

undocumented

inline decltype(auto) **end**()

undocumented

Private Types

using **containers_t** = std::tuple<*Containers*...>

Private Members

containers_t **containers**

template<class ...**Iterators**>

class **ZipIterator**

#include <iterators.hh> iterator for emulation of python zip

Public Functions

inline explicit **ZipIterator**(*tuple_t* iterators)

undocumented

inline decltype(auto) **operator***()

undocumented

inline *ZipIterator* &**operator++**()

undocumented

inline bool **operator==**(const *ZipIterator* &other) const

undocumented

inline bool **operator!=**(const *ZipIterator* &other) const

undocumented

Private Types

using **tuple_t** = std::tuple<*Iterators*...>

Private Members

tuple_t **iterators**

namespace **akantu**

Functions

template<class ...**Iterators**>
decltype(auto) **zip_iterator**(std::tuple<*Iterators*...> &&iterators_tuple)

emulates python *zip()*

template<class ...**Containers**>
decltype(auto) **zip**(*Containers*&&... conts)

emulates python's *zip()*

template<class **T**, typename = std::enable_if_t<std::is_integral<std::decay_t<*T*>>::value>>
inline decltype(auto) **arange**(const *T* &stop)

emulates python's range()

template<class **T1**, class **T2**, typename = std::enable_if_t<std::is_integral<std::common_type_t<*T1*, *T2*>>::value>>
inline decltype(auto) constexpr **arange**(const *T1* &start, const *T2* &stop)

emulates python's range()

template<class **T1**, class **T2**, class **T3**, typename = std::enable_if_t<std::is_integral<std::common_type_t<*T1*, *T2*, *T3*>>::value>>
inline decltype(auto) constexpr **arange**(const *T1* &start, const *T2* &stop, const *T3* &step)

emulates python's range()

template<class **Container**>
inline decltype(auto) constexpr **enumerate**(*Container* &&container, size_t start_ = 0)

emulates python's enumerate

namespace **containers**

namespace **iterators**

namespace **tuple**

Functions

template<class **Tuple**>
bool **are_not_equal**(*Tuple* &&a, *Tuple* &&b)

detail

template<class **F**, class **Tuple**>
void **foreach_**(*F* &&func, *Tuple* &&tuple)

detail

template<class **F**, class **Tuple**>
decltype(auto) **transform**(*F* &&func, *Tuple* &&tuple)

detail

namespace **details**

Functions

template<typename ...**Ts**>
decltype(auto) **make_tuple_no_decay**(*Ts*&&... args)

eats up a bunch of arguments and returns them packed in a tuple

template<class **F**, class **Tuple**, size_t... **Is**>
void **foreach_impl**(*F* &&func, *Tuple* &&tuple, std::index_sequence<*Is*...>&&)

helper for static for loop

template<class **F**, class **Tuple**, size_t... **Is**>
decltype(auto) **transform_impl**(*F* &&func, *Tuple* &&tuple, std::index_sequence<*Is*...>&&)

detail

namespace **Eigen**

namespace **internal**

Typedefs

typedef *muSpectre*::Dim_t **Dim_t**

universal index type

typedef *muSpectre*::Real **Real**

universal real value type

namespace **muFFT**

Typedefs

using **Matrix_t** = *Eigen*::Matrix<T, *Eigen*::Dynamic, *Eigen*::Dynamic>

using **Derivative_ptr** = std::shared_ptr<*DerivativeBase*>

convenience alias

using **Gradient_t** = std::vector<*Derivative_ptr*>

convenience alias

using **FFTEngine_ptr** = std::shared_ptr<*FFTEngineBase*>

reference to fft engine is safely managed through a `std::shared_ptr`

Enums

enum **FFT_PlanFlags**

Planner flags for FFT (follows FFTW, hopefully this choice will be compatible with alternative FFT implementations)

Values:

enumerator **estimate**

cheapest plan for slowest execution

enumerator **measure**

more expensive plan for fast execution

enumerator **patient**

very expensive plan for fastest execution

Functions

template<typename **T**>
T **modulo**(*T* a, *T* b)

module operator that can handle negative values

std::ostream &**operator<<**(std::ostream &os, const *DiscreteDerivative* &derivative)

Allows inserting *muFFT::DiscreteDerivative*s into `std::ostream`s

Gradient_t **make_fourier_gradient**(const Dim_t &spatial_dimension)

convenience function to build a spatial_dimension-al gradient operator using exact Fourier differentiation

Parameters
spatial_dimension – number of spatial dimensions

std::valarray<Real> **fft_freqs**(size_t nb_samples)

compute fft frequencies (in time (or length) units of of sampling periods), see numpy's fftfreq function for reference

std::valarray<Real> **fft_freqs**(size_t nb_samples, Real length)

compute fft frequencies in correct length or time units. Here, length refers to the total size of the domain over which the fft is taken (for instance the length of an edge of an RVE)

template<size_t **dim**>
constexpr *Ccoord_t*<*dim*> **get_nb_hermitian_grid_pts**(*Ccoord_t*<*dim*> full_nb_grid_pts)

returns the hermition grid to correcsponding to a full grid, assuming that the last dimension is not fully represented in reciprocal space

template<size_t **MaxDim**>
inline *muGrid*::*DynCcoord*<*MaxDim*> **get_nb_hermitian_grid_pts**(*muGrid*::*DynCcoord*<*MaxDim*> full_nb_grid_pts)

returns the hermition grid to correcsponding to a full grid, assuming that the last dimension is not fully represented in reciprocal space

inline Int **fft_freq**(Int i, size_t nb_samples)

compute fft frequency (in time (or length) units of of sampling periods), see numpy's fftfreq function for reference

inline Real **fft_freq**(Int i, size_t nb_samples, Real length)

compute fft frequency in correct length or time units. Here, length refers to the total size of the domain over which the fft is taken (for instance the length of an edge of an RVE)

template<size_t **dim**>
inline std::array<std::valarray<Real>, *dim*> **fft_freqs**(*Ccoord_t*<*dim*> nb_grid_pts)

Get fft_freqs for a grid

template<size_t **dim**>
inline std::array<std::valarray<Real>, *dim*> **fft_freqs**(*Ccoord_t*<*dim*> nb_grid_pts, std::array<Real, *dim*> lengths)

Get fft_freqs for a grid in correct length or time units.

namespace **internal**

Functions

template<Dim_t **Dim**, size_t... **I**>
constexpr *Ccoord_t*<*Dim*> **herm**(const *Ccoord_t*<*Dim*> &nb_grid_pts, std::index_sequence<*I*...>)

computes hermitian size according to FFTW

namespace **muGrid**

Typedefs

using **optional** = typename std::experimental::optional<T>

emulation `std::optional` (a C++17 feature)

using **Decomp_t** = *Eigen*::SelfAdjointEigenSolver<*Eigen*::Matrix<*Real*, dim, dim>>

It seems we only need to take logs of self-adjoint matrices

using **Matrix_t** = *Eigen*::Matrix<*Real*, Dim, Dim>

using **MatrixFieldMap** = StaticFieldMap<T, Mutability, *internal*::*MatrixMap*<T, NbRow, NbCol>, IterationType>

Alias of *muGrid::StaticFieldMap* you wish to iterate over pixel by pixel or quadrature point by quadrature point with statically sized `Eigen::Matrix` iterates

Template Parameters

- **T** – scalar type stored in the field, must be one of `muGrid::Real`, `muGrid::Int`, `muGrid::Uint`, `muGrid::Complex`
- **Mutability** – whether or not the map allows to modify the content of the field
- **NbRow** – number of rows of the iterate
- **NbCol** – number of columns of the iterate

- **IterationType** – whether to iterate over pixels or quadrature points

using **ArrayFieldMap** = StaticFieldMap<T, Mutability, *internal*::*ArrayMap*<T, NbRow, NbCol>, IterationType>

Alias of *muGrid::StaticFieldMap* you wish to iterate over pixel by pixel or quadrature point by quadrature point with* statically sized `Eigen::Array` iterates

Template Parameters

- **T** – scalar type stored in the field, must be one of `muGrid::Real`, `muGrid::Int`, `muGrid::Uint`, `muGrid::Complex`
- **Mutability** – whether or not the map allows to modify the content of the field
- **NbRow** – number of rows of the iterate
- **NbCol** – number of columns of the iterate
- **IterationType** – whether to iterate over pixels or quadrature points

using **ScalarFieldMap** = StaticFieldMap<T, Mutability, *internal*::*ScalarMap*<T>, *Iteration*::*QuadPt*>

Alias of *muGrid::StaticFieldMap* over a scalar field you wish to iterate over quadrature point by quadrature point.

Template Parameters

- **T** – scalar type stored in the field, must be one of `muGrid::Real`, `muGrid::Int`, `muGrid::Uint`, `muGrid::Complex`
- **Mutability** – whether or not the map allows to modify the content of the field

using **T1NFieldMap** = StaticFieldMap<T, Mutability, *internal*::*MatrixMap*<T, Dim, 1>, *Iteration*::*QuadPt*>

Alias of `muGrid::StaticNFieldMap` over a first-rank tensor field you wish to iterate over quadrature point by quadrature point.

Template Parameters

- **T** – scalar type stored in the field, must be one of `muGrid::Real`, `muGrid::Int`, `muGrid::Uint`, `muGrid::Complex`
- **Mutability** – whether or not the map allows to modify the content of the field
- **Dim** – spatial dimension of the tensor

using **T1FieldMap** = StaticFieldMap<T, Mutability, *internal*::*MatrixMap*<T, Dim, 1>, *Iteration*::*QuadPt*>

Alias of *muGrid::StaticFieldMap* over a second-rank tensor field you wish to iterate over quadrature point by quadrature point.

Template Parameters

- **T** – scalar type stored in the field, must be one of `muGrid::Real`, `muGrid::Int`, `muGrid::Uint`, `muGrid::Complex`
- **Mutability** – whether or not the map allows to modify the content of the field
- **Dim** – spatial dimension of the tensor

using **T2FieldMap** = StaticFieldMap<T, Mutability, *internal*::*MatrixMap*<T, Dim, Dim>, *Iteration*::*QuadPt*>

Alias of *muGrid::StaticFieldMap* over a second-rank tensor field you wish to iterate over quadrature point by quadrature point.

Template Parameters

- **T** – scalar type stored in the field, must be one of `muGrid::Real`, `muGrid::Int`, `muGrid::Uint`, `muGrid::Complex`
- **Mutability** – whether or not the map allows to modify the content of the field
- **Dim** – spatial dimension of the tensor

using **T4FieldMap** = StaticFieldMap<T, Mutability, *internal*::*MatrixMap*<T, Dim * Dim, Dim * Dim>, *Iteration*::*QuadPt*>

Alias of *muGrid::StaticFieldMap* over a fourth-rank tensor field you wish to iterate over quadrature point by quadrature point.

Template Parameters

- **T** – scalar type stored in the field, must be one of `muGrid::Real`, `muGrid::Int`, `muGrid::Uint`, `muGrid::Complex`
- **Mutability** – whether or not the map allows to modify the content of the field
- **Dim** – spatial dimension of the tensor

using **RealField** = *TypedField*<*Real*>

Alias for real-valued fields.

using **ComplexField** = *TypedField*<*Complex*>

Alias for complex-valued fields.

using **IntField** = *TypedField*<*Int*>

Alias for integer-valued fields.

using **UintField** = *TypedField*<*Uint*>

Alias for unsigned integer-valued fields.

using **Dim_t** = int

Eigen uses signed integers for dimensions. For consistency, µGrid uses them througout the code. Needs to represent -1 for Eigen

using **Uint** = unsigned int

type to use in math for unsigned integers

using **Int** = int

type to use in math for signed integers

using **Real** = double

type to use in math for real numbers

using **Complex** = std::complex<*Real*>

type to use in math for complex numbers

using **Ccoord_t** = std::array<*Dim_t*, Dim>

Ccoord_t are cell coordinates, i.e. integer coordinates.

using **Rcoord_t** = std::array<*Real*, Dim>

Real space coordinates.

using **DynCcoord_t** = *DynCcoord*<*threeD*>

usually, we should not need omre than three dimensions

using **DynRcoord_t** = *DynCcoord*<*threeD*, *Real*>

usually, we should not need omre than three dimensions

using **MappedMatrixField** = *MappedField*<*MatrixFieldMap*<T, Mutability, NbRow, NbCol, IterationType>>

Alias of *muGrid::MappedField* for a map with corresponding `muSpectre::Field` you wish to iterate over pixel by pixel or quadrature point by quadrature point with statically sized `Eigen::Matrix` iterates

Template Parameters

- **T** – scalar type stored in the field, must be one of `muGrid::Real`, `muGrid::Int`, `muGrid::Uint`, `muGrid::Complex`
- **Mutability** – whether or not the map allows to modify the content of the field
- **NbRow** – number of rows of the iterate
- **NbCol** – number of columns of the iterate
- **IterationType** – whether to iterate over pixels or quadrature points

using **MappedArrayField** = *MappedField*<*ArrayFieldMap*<T, Mutability, NbRow, NbCol, IterationType>>

Alias of *muGrid::MappedField* for a map with corresponding `muSpectre::Field` you wish to iterate over pixel by pixel or quadrature point by quadrature point with statically sized `Eigen::Array` iterates

Template Parameters

- **T** – scalar type stored in the field, must be one of `muGrid::Real`, `muGrid::Int`, `muGrid::Uint`, `muGrid::Complex`
- **Mutability** – whether or not the map allows to modify the content of the field
- **NbRow** – number of rows of the iterate
- **NbCol** – number of columns of the iterate
- **IterationType** – whether to iterate over pixels or quadrature points

using **MappedScalarField** = *MappedField*<*ScalarFieldMap*<T, Mutability>>

Alias of *muGrid::MappedField* for a map of scalars with corresponding `muSpectre::Field` you wish to iterate over quadrature point by quadrature point.

Template Parameters

- **T** – scalar type stored in the field, must be one of `muGrid::Real`, `muGrid::Int`, `muGrid::Uint`, `muGrid::Complex`
- **Mutability** – whether or not the map allows to modify the content of the field

using **MappedT1Field** = *MappedField*<*T1FieldMap*<T, Mutability, Dim>>

Alias of *muGrid::MappedField* for a map of second-rank with corresponding `muSpectre::Field` you wish to iterate over quadrature point by quadrature point.

Template Parameters

- **T** – scalar type stored in the field, must be one of `muGrid::Real`, `muGrid::Int`, `muGrid::Uint`, `muGrid::Complex`
- **Mutability** – whether or not the map allows to modify the content of the field
- **Dim** – spatial dimension of the tensors

using **MappedT2Field** = *MappedField*<*T2FieldMap*<T, Mutability, Dim>>

Alias of *muGrid::MappedField* for a map of first-rank with corresponding `muSpectre::Field` you wish to iterate over quadrature point by quadrature point.

Template Parameters

- **T** – scalar type stored in the field, must be one of `muGrid::Real`, `muGrid::Int`, `muGrid::Uint`, `muGrid::Complex`
- **Mutability** – whether or not the map allows to modify the content of the field
- **Dim** – spatial dimension of the tensors

using **MappedT4Field** = *MappedField*<*T4FieldMap*<T, Mutability, Dim>>

Alias of *muGrid::MappedField* for a map of fourth-rank with corresponding `muSpectre::Field` you wish to iterate over quadrature point by quadrature point.

Template Parameters

- **T** – scalar type stored in the field, must be one of `muGrid::Real`, `muGrid::Int`, `muGrid::Uint`, `muGrid::Complex`
- **Mutability** – whether or not the map allows to modify the content of the field
- **Dim** – spatial dimension of the tensors

using **MappedMatrixStateField** = *MappedStateField*<*MatrixStateFieldMap*<T, Mutability, NbRow, NbCol, NbMemory, IterationType>>

Alias of *muGrid::MappedStateField* for a map with corresponding `muSpectre::StateField` you wish to iterate over pixel by pixel or quadrature point by quadrature point with statically sized `Eigen::Matrix` iterates

Template Parameters

- **T** – scalar type stored in the field, must be one of `muGrid::Real`, `muGrid::Int`, `muGrid::Uint`, `muGrid::Complex`
- **Mutability** – whether or not the map allows to modify the content of the field
- **NbRow** – number of rows of the iterate
- **NbCol** – number of columns of the iterate
- **NbMemory** – number of previous values to store
- **IterationType** – whether to iterate over pixels or quadrature points

using **MappedArrayStateField** = *MappedStateField*<*ArrayStateFieldMap*<T, Mutability, NbRow, NbCol, NbMemory, IterationType>>

Alias of *muGrid::MappedStateField* for a map with corresponding `muSpectre::StateField` you wish to iterate over pixel by pixel or quadrature point by quadrature point with statically sized `Eigen::Array` iterates

Template Parameters

- **T** – scalar type stored in the field, must be one of `muGrid::Real`, `muGrid::Int`, `muGrid::Uint`, `muGrid::Complex`
- **Mutability** – whether or not the map allows to modify the content of the field
- **NbRow** – number of rows of the iterate
- **NbCol** – number of columns of the iterate
- **NbMemory** – number of previous values to store
- **IterationType** – whether to iterate over pixels or quadrature points

using **MappedScalarStateField** = *MappedStateField*<*ScalarStateFieldMap*<T, Mutability, NbMemory>>

Alias of *muGrid::MappedStateField* for a map of scalars with corresponding `muSpectre::StateField` you wish to iterate over quadrature point by quadrature point.

Template Parameters

- **T** – scalar type stored in the field, must be one of `muGrid::Real`, `muGrid::Int`, `muGrid::Uint`, `muGrid::Complex`
- **Mutability** – whether or not the map allows to modify the content of the field
- **NbMemory** – number of previous values to store

using **MappedT1StateNField** = *MappedStateField*<*T1StateNFieldMap*<T, Mutability, Dim, NbMemory>>

Alias of *muGrid::MappedStateField* for a map of first-rank with corresponding `muSpectre::StateNField` you wish to iterate over quadrature point by quadrature point.

Template Parameters

- **T** – scalar type stored in the field, must be one of `muGrid::Real`, `muGrid::Int`, `muGrid::Uint`, `muGrid::Complex`
- **Mutability** – whether or not the map allows to modify the content of the field
- **Dim** – spatial dimension of the tensors
- **NbMemory** – number of previous values to store

using **MappedT2StateField** = *MappedStateField*<*T2StateFieldMap*<T, Mutability, Dim, NbMemory>>

Alias of *muGrid::MappedStateField* for a map of second-rank with corresponding `muSpectre::StateField` you wish to iterate over quadrature point by quadrature point.

Template Parameters

- **T** – scalar type stored in the field, must be one of `muGrid::Real`, `muGrid::Int`, `muGrid::Uint`, `muGrid::Complex`
- **Mutability** – whether or not the map allows to modify the content of the field
- **Dim** – spatial dimension of the tensors

- **NbMemory** – number of previous values to store

using **MappedT4StateField** = *MappedStateField*<*T4StateFieldMap*<T, Mutability, Dim, NbMemory>>

Alias of *muGrid::MappedStateField* for a map of fourth-rank with corresponding `muSpectre::StateField` you wish to iterate over quadrature point by quadrature point.

Template Parameters

- **T** – scalar type stored in the field, must be one of `muGrid::Real`, `muGrid::Int`, `muGrid::Uint`, `muGrid::Complex`
- **Mutability** – whether or not the map allows to modify the content of the field
- **Dim** – spatial dimension of the tensors
- **NbMemory** – number of previous values to store

using **RealStateField** = *TypedStateField*<*Real*>

Alias for real-valued state fields.

using **ComplexStateField** = *TypedStateField*<*Complex*>

Alias for complex-valued state fields.

using **IntStateField** = *TypedStateField*<*Int*>

Alias for integer-valued state fields.

using **Uintfield** = *TypedStateField*<*Uint*>

Alias for unsigned integer-valued state fields.

using **MatrixStateFieldMap** = StaticStateFieldMap<T, Mutability, *internal*::*MatrixMap*<T, NbRow, NbCol>, NbMemory, IterationType>

Alias of *muGrid::StaticStateFieldMap* you wish to iterate over pixel by pixel or quadrature point by quadrature point with statically sized `Eigen::Matrix` iterates

Template Parameters

- **T** – scalar type stored in the field, must be one of `muGrid::Real`, `muGrid::Int`, `muGrid::Uint`, `muGrid::Complex`
- **Mutability** – whether or not the map allows to modify the content of the field
- **NbRow** – number of rows of the iterate
- **NbCol** – number of columns of the iterate
- **NbMemory** – number of previous values to store
- **IterationType** – whether to iterate over pixels or quadrature points

using **ArrayStateFieldMap** = StaticStateFieldMap<T, Mutability, *internal*::*ArrayMap*<T, NbRow, NbCol>, NbMemory, IterationType>

Alias of *muGrid::StaticStateFieldMap* you wish to iterate over pixel by pixel or quadrature point by quadrature point with* statically sized `Eigen::Array` iterates

Template Parameters

- **T** – scalar type stored in the field, must be one of `muGrid::Real`, `muGrid::Int`, `muGrid::Uint`, `muGrid::Complex`

- **Mutability** – whether or not the map allows to modify the content of the field
- **NbRow** – number of rows of the iterate
- **NbCol** – number of columns of the iterate
- **NbMemory** – number of previous values to store
- **IterationType** – whether to iterate over pixels or quadrature points

using **ScalarStateFieldMap** = StaticStateFieldMap<T, Mutability, *internal*::*ScalarMap*<T>, NbMemory, *Iteration*::*QuadPt*>

Alias of *muGrid::StaticStateFieldMap* over a scalar field you wish to iterate over quadrature point by quadrature point.

Template Parameters

- **T** – scalar type stored in the field, must be one of `muGrid::Real`, `muGrid::Int`, `muGrid::Uint`, `muGrid::Complex`
- **Mutability** – whether or not the map allows to modify the content of the field
- **NbMemory** – number of previous values to store

using **T1StateNFieldMap** = StaticStateFieldMap<T, Mutability, *internal*::*MatrixMap*<T, Dim, 1>, NbMemory, *Iteration*::*QuadPt*>

Alias of `muGrid::StaticStateNFieldMap` over a first-rank tensor field you wish to iterate over quadrature point by quadrature point.

Template Parameters

- **T** – scalar type stored in the field, must be one of `muGrid::Real`, `muGrid::Int`, `muGrid::Uint`, `muGrid::Complex`
- **Mutability** – whether or not the map allows to modify the content of the field
- **Dim** – spatial dimension of the tensor
- **NbMemory** – number of previous values to store

using **T2StateFieldMap** = StaticStateFieldMap<T, Mutability, *internal*::*MatrixMap*<T, Dim, Dim>, NbMemory, *Iteration*::*QuadPt*>

Alias of `muGrid::StaticStateNFieldMap` over a second-rank tensor field you wish to iterate over quadrature point by quadrature point.

Template Parameters

- **T** – scalar type stored in the field, must be one of `muGrid::Real`, `muGrid::Int`, `muGrid::Uint`, `muGrid::Complex`
- **Mutability** – whether or not the map allows to modify the content of the field
- **Dim** – spatial dimension of the tensor
- **NbMemory** – number of previous values to store

using **T4StateFieldMap** = StaticStateFieldMap<T, Mutability, *internal*::*MatrixMap*<T, Dim * Dim, Dim * Dim>, NbMemory, *Iteration*::*QuadPt*>

Alias of *muGrid::StaticStateFieldMap* over a fourth-rank tensor field you wish to iterate over quadrature point by quadrature point.

Template Parameters

- **T** – scalar type stored in the field, must be one of `muGrid::Real`, `muGrid::Int`, `muGrid::Uint`, `muGrid::Complex`
- **Mutability** – whether or not the map allows to modify the content of the field
- **Dim** – spatial dimension of the tensor
- **NbMemory** – number of previous values to store

using **T4Mat** = *Eigen*::Matrix<T, Dim * Dim, Dim * Dim>

simple adapter function to create a matrix that can be mapped as a tensor

using **T4MatMap** = std::conditional_t<ConstMap, *Eigen*::Map<const *T4Mat*<T, Dim>>, *Eigen*::Map<*T4Mat*<T, Dim>>>

Map onto `muGrid::T4Mat`

Enums

enum **Iteration**

Used to specify whether to iterate over pixels or quadrature points in field maps

Values:

enumerator **Pixel**

enumerator **QuadPt**

enum **Mapping**

Maps can give constant or mutable access to the mapped field through their iterators or access operators.

Values:

enumerator **Const**

enumerator **Mut**

Functions

template<*Dim_t* **order**, *Dim_t* **dim**, typename **Fun_t**>
inline decltype(auto) **call_sizes**(*Fun_t* &&fun)

takes a lambda and calls it with the proper `Eigen::Sizes` unpacked as arguments. Is used to call constructors of a `Eigen::Tensor` or map thereof in a context where the spatial dimension is templated

static constexpr *Dim_t* **ct_sqrt**(*Dim_t* res, *Dim_t* l, *Dim_t* r)

static constexpr *Dim_t* **ct_sqrt**(*Dim_t* res)

template<*Dim_t* **dim**>

inline decltype(auto) **logm**(const *log_comp*::*Mat_t*<*dim*> &mat)

computes the matrix logarithm efficiently for dim=1, 2, or 3 for a diagonizable tensor. For larger tensors, better use the direct eigenvalue/vector computation

template<class **Derived**, template<class Matrix_t> class **DecompType** = *Eigen*::SelfAdjointEigenSolver>
inline decltype(auto) **spectral_decomposition**(const *Eigen*::MatrixBase<*Derived*> &mat)

compute the spectral decomposition

template<*Dim_t* **Dim**>
inline decltype(auto) **logm_alt**(const *Decomp_t*<*Dim*> &spectral_decomp)

Uses a pre-existing spectral decomposition of a matrix to compute its logarithm

Parameters
spectral_decomp – spectral decomposition of a matrix

Template Parameters
Dim – spatial dimension (i.e., number of rows and colums in the matrix)

template<class **Derived**>
inline decltype(auto) **logm_alt**(const *Eigen*::MatrixBase<*Derived*> &mat)

compute the matrix log with a spectral decomposition. This may not be the most efficient way to do this

template<*Dim_t* **Dim**, template<class Matrix_t> class **DecompType** = *Eigen*::SelfAdjointEigenSolver>
inline decltype(auto) **expm**(const *Decomp_t*<*Dim*> &spectral_decomp)

Uses a pre-existing spectral decomposition of a matrix to compute its exponential

Parameters
spectral_decomp – spectral decomposition of a matrix

Template Parameters
Dim – spatial dimension (i.e., number of rows and colums in the matrix)

template<class **Derived**>
inline decltype(auto) **expm**(const *Eigen*::MatrixBase<*Derived*> &mat)

compute the matrix exponential with a spectral decomposition. This may not be the most efficient way to do this

template<typename **T**, size_t **Dim**>
Eigen::Map<*Eigen*::Matrix<*T*, *Dim*, 1>> **eigen**(std::array<*T*, *Dim*> &coord)

return a Eigen representation of the data stored in a std::array (e.g., for doing vector operations on a coordinate)

template<typename **T**, size_t **Dim**>
Eigen::Map<const *Eigen*::Matrix<*T*, *Dim*, 1>> **eigen**(const std::array<*T*, *Dim*> &coord)

return a constant Eigen representation of the data stored in a std::array (e.g., for doing vector operations on a coordinate)

template<typename **T**, size_t **MaxDim**>
Eigen::Map<*Eigen*::Matrix<*T*, *Eigen*::Dynamic, 1>> **eigen**(*DynCcoord*<*MaxDim*, *T*> &coord)

return a Eigen representation of the data stored in a std::array (e.g., for doing vector operations on a coordinate)

template<typename **T**, size_t **MaxDim**>
Eigen::Map<const *Eigen*::Matrix<*T*, *Eigen*::Dynamic, 1>> **eigen**(const *DynCcoord*<*MaxDim*, *T*> &coord)

return a const Eigen representation of the data stored in a std::array (e.g., for doing vector operations on a coordinate)

template<typename **T**>

std::ostream &**operator<<**(std::ostream &os, const std::vector<*T*> &values)

Allows inserting `std::vector` into `std::ostreams`

template<typename **T**, size_t **dim**>
std::ostream &**operator<<**(std::ostream &os, const std::array<*T*, *dim*> &values)

Allows inserting `muGrid::Ccoord_t` and `muGrid::Rcoord_t` into `std::ostreams`

template<size_t **MaxDim**, typename **T**>
std::ostream &**operator<<**(std::ostream &os, const *DynCcoord*<*MaxDim*, *T*> &values)

Allows inserting *muGrid::DynCcoord* into `std::ostreams`

template<size_t **dim**>
Rcoord_t<*dim*> **operator/**(const *Rcoord_t*<*dim*> &a, const *Rcoord_t*<*dim*> &b)

element-wise division

template<size_t **dim**>
Rcoord_t<*dim*> **operator/**(const *Rcoord_t*<*dim*> &a, const *Ccoord_t*<*dim*> &b)

element-wise division

template<typename **R**, typename **I**>
constexpr *R* **ipow**(*R* base, *I* exponent)

compile-time potentiation required for field-size computations

template<typename **T**>
std::vector<*Dim_t*> **numpy_copy**(const TypedFieldBase<*T*> &field, pybind11::array_t<*T*, pybind11::array::f_style> array)

template<typename **T**>
pybind11::array_t<*T*, pybind11::array::f_style> **numpy_wrap**(const TypedFieldBase<*T*> &field, std::vector<*Dim_t*> components_shape = std::vector<*Dim_t*>{})

template<typename **T**>
pybind11::tuple **to_tuple**(*T* a)

template<typename **T4**>
inline auto **get**(const *Eigen*::MatrixBase<*T4*> &t4, *Dim_t* i, *Dim_t* j, *Dim_t* k, *Dim_t* l) -> decltype(auto)

provides index-based access to fourth-order Tensors represented by square matrices

template<typename **T4**>
inline auto **get**(*Eigen*::MatrixBase<*T4*> &t4, *Dim_t* i, *Dim_t* j, *Dim_t* k, *Dim_t* l) -> decltype(*t4*.coeffRef(*i*, *j*))

provides constant index-based access to fourth-order Tensors represented by square matrices

Variables

constexpr *Dim_t* **oneD** = {1}

constant for a one-dimensional problem

constexpr *Dim_t* **twoD** = {2}

constant for a two-dimensional problem

constexpr *Dim_t* **threeD** = {3}

constant for a three-dimensional problem

constexpr *Dim_t* **firstOrder** = {1}

constant for vectors

constexpr *Dim_t* **secondOrder** = {2}

constant second-order tensors

constexpr *Dim_t* **fourthOrder** = {4}

constant fourth-order tensors

constexpr *Dim_t* **OneQuadPt** = {1}

constant for 1 quadrature point/pixel

constexpr *Real* **pi** = {3.14159265358979323846264 33}

convenience definitions

static constexpr *Dim_t* **Unknown** = {-1}

constant used to explicitly denote unknown positive integers

namespace **CcoordOps**

Functions

Dim_t **get_index**(const *DynCcoord_t* &nb_grid_pts, const *DynCcoord_t* &locations, const *DynCcoord_t* &ccoord)

get the linear index of a pixel in a given grid

Real **compute_volume**(const *DynRcoord_t* &lenghts)

these functions can be used whenever it is necessary to calcluate the volume of a cell or each pixle of the cell

Real **compute_pixel_volume**(const *DynCcoord_t* &nb_grid_pts, const *DynRcoord_t* &lenghts)

template<size_t **dim**, typename **T**>
constexpr std::array<*T*, *dim*> **get_cube**(*T* nb_grid_pts)

returns a grid of equal number of grid points in each direction

template<size_t **MaxDim** = *threeD*>
DynCcoord<*MaxDim*> **get_cube**(const *Dim_t* &dim, const *Dim_t* &nb_grid_pts)

returns a grid of equal number of grid points in each direction

template<size_t **dim**>
Eigen::Matrix<*Real*, *dim*, 1> **get_vector**(const *Ccoord_t*<*dim*> &ccoord, *Real* pix_size = 1.)

return physical vector of a cell of cubic pixels

template<size_t **dim**, typename **T**>

Eigen::Matrix<*T*, *dim*, 1> **get_vector**(const *Ccoord_t*<*dim*> &ccoord, *Eigen*::Matrix<*T*, *Dim_t*(*dim*), 1> pix_size)

return physical vector of a cell of general pixels

template<size_t **dim**, typename **T**>
Eigen::Matrix<*T*, *dim*, 1> **get_vector**(const *Ccoord_t*<*dim*> &ccoord, const std::array<*T*, *dim*> &pix_size)

return physical vector of a cell of general pixels

template<size_t **dim**, size_t **MaxDim**, typename **T**>
Eigen::Matrix<*T*, *dim*, 1> **get_vector**(const *Ccoord_t*<*dim*> &ccoord, const *DynCcoord*<*MaxDim*, *T*> &pix_size)

return physical vector of a cell of general pixels

template<size_t **dim**>
Eigen::Matrix<*Real*, *dim*, 1> **get_vector**(const *DynCcoord_t* &ccoord, *Real* pix_size = 1.)

return physical vector of a cell of cubic pixels

template<size_t **dim**, typename **T**>
Eigen::Matrix<*T*, *dim*, 1> **get_vector**(const *DynCcoord_t* ccoord, *Eigen*::Matrix<*T*, *Dim_t*(*dim*), 1> pix_size)

return physical vector of a cell of general pixels

template<size_t **dim**, typename **T**>
Eigen::Matrix<*T*, *dim*, 1> **get_vector**(const *DynCcoord_t* ccoord, const std::array<*T*, *dim*> &pix_size)

return physical vector of a cell of general pixels

template<size_t **dim**, size_t **MaxDim**, typename **T**>
Eigen::Matrix<*T*, *dim*, 1> **get_vector**(const *DynCcoord_t* ccoord, const *DynCcoord*<*MaxDim*, *T*> &pix_size)

return physical vector of a cell of general pixels

template<size_t **dim**>
constexpr *Ccoord_t*<*dim*> **get_default_strides**(const *Ccoord_t*<*dim*> &nb_grid_pts)

get all strides from a column-major grid

template<size_t **MaxDim**>
constexpr *DynCcoord*<*MaxDim*> **get_default_strides**(const *DynCcoord*<*MaxDim*> &nb_grid_pts)

get all strides from a row-major grid

template<size_t **dim**>
constexpr *Ccoord_t*<*dim*> **get_ccoord**(const *Ccoord_t*<*dim*> &nb_grid_pts, const *Ccoord_t*<*dim*> &locations, *Dim_t* index)

get the i-th pixel in a grid of size nb_grid_pts

template<size_t **dim**, size_t... **I**>
constexpr *Ccoord_t*<*dim*> **get_ccoord**(const *Ccoord_t*<*dim*> &nb_grid_pts, const *Ccoord_t*<*dim*> &locations, *Dim_t* index, std::index_sequence<*I*...>)

get the i-th pixel in a grid of size nb_grid_pts

template<size_t... **I**>
constexpr *Ccoord_t*<1> **get_ccoord**(const *Ccoord_t*<1> &nb_grid_pts, const *Ccoord_t*<1> &locations, *Dim_t* index, std::index_sequence<*I*...>)

get the i-th pixel in a grid of size nb_grid_pts - specialization for one dimension

template<size_t **dim**>

constexpr *Ccoord_t*<*dim*> **get_ccoord_from_strides**(const *Ccoord_t*<*dim*> &nb_grid_pts, const *Ccoord_t*<*dim*> &locations, const *Ccoord_t*<*dim*> &strides, *Dim_t* index)

get the i-th pixel in a grid of size nb_grid_pts

template<size_t **MaxDim**>
inline *DynCcoord*<*MaxDim*> **get_ccoord_from_strides**(const *DynCcoord*<*MaxDim*> &nb_grid_pts, const *DynCcoord*<*MaxDim*> &locations, const *DynCcoord*<*MaxDim*> &strides, *Dim_t* index)

get the i-th pixel in a grid of size nb_grid_pts

template<size_t **dim**>
constexpr *Dim_t* **get_index**(const *Ccoord_t*<*dim*> &nb_grid_pts, const *Ccoord_t*<*dim*> &locations, const *Ccoord_t*<*dim*> &ccoord)

get the linear index of a pixel in a given grid

template<size_t **dim**>
constexpr *Dim_t* **get_index_from_strides**(const *Ccoord_t*<*dim*> &strides, const *Ccoord_t*<*dim*> &ccoord)

get the linear index of a pixel given a set of strides

template<size_t **MaxDim**>
Dim_t **get_index_from_strides**(const *DynCcoord*<*MaxDim*> &strides, const *DynCcoord*<*MaxDim*> &ccoord)

get the linear index of a pixel given a set of strides

template<size_t **dim**>
constexpr size_t **get_size**(const *Ccoord_t*<*dim*> &nb_grid_pts)

get the number of pixels in a grid

template<size_t **MaxDim**>
size_t **get_size**(const *DynCcoord*<*MaxDim*> &nb_grid_pts)

get the number of pixels in a grid

template<size_t **dim**>
constexpr size_t **get_size_from_strides**(const *Ccoord_t*<*dim*> &nb_grid_pts, const *Ccoord_t*<*dim*> &strides)

get the number of pixels in a grid given its strides

namespace **internal**

Functions

template<typename **T**>
constexpr *T* **ret**(*T* val, size_t)

simple helper returning the first argument and ignoring the second

template<*Dim_t* **Dim**, typename **T**, size_t... **I**>
constexpr std::array<*T*, *Dim*> **cube_fun**(*T* val, std::index_sequence<*I*...>)

helper to build cubes

template<*Dim_t* **Dim**, size_t... **I**>

constexpr *Ccoord_t*<*Dim*> **herm**(const *Ccoord_t*<*Dim*> &nb_grid_pts, std::index_sequence<*I*...>)

computes hermitian size according to FFTW

template<*Dim_t* **Dim**>
constexpr *Dim_t* **stride**(const *Ccoord_t*<*Dim*> &nb_grid_pts, const size_t index)

compute the stride in a direction of a column-major grid

template<*Dim_t* **Dim**, size_t... **I**>
constexpr *Ccoord_t*<*Dim*> **compute_strides**(const *Ccoord_t*<*Dim*> &nb_grid_pts,
std::index_sequence<*I*...>)

get all strides from a column-major grid (helper function)

namespace **EigenCheck**

namespace **internal**

Functions

template<*Dim_t* **Dim**, *Dim_t* **NbRow**, *Dim_t* **NbCol**>
inline constexpr *Dim_t* **get_rank**()

determine the rank of a Dim-dimensional tensor represented by an `Eigen::Matrix` of shape NbRow × NbCol

Template Parameters

- **Dim** – spatial dimension
- **NbRow** – number of rows
- **NbCol** – number of columns

namespace **internal**

Typedefs

using **MatrixMap** = *EigenMap*<T, *Eigen*::Matrix<T, NbRow, NbCol>>

internal convenience alias for creating maps iterating over statically sized `Eigen::Matrix`s

using **ArrayMap** = *EigenMap*<T, *Eigen*::Array<T, NbRow, NbCol>>

internal convenience alias for creating maps iterating over statically sized `Eigen::Array`s

namespace **log_comp**

Typedefs

using **Mat_t** = *Eigen*::Matrix<*Real*, dim, dim>

Matrix type used for logarithm evaluation.

using **Vec_t** = *Eigen*::Matrix<*Real*, dim, 1>

Vector type used for logarithm evaluation.

Functions

template<*Dim_t* **dim**, *Dim_t* **i**>
inline decltype(auto) **P**(const *Vec_t*<*dim*> &eigs, const *Mat_t*<*dim*> &T)

Product term.

template<*Dim_t* **dim**>
inline decltype(auto) **Sum**(const *Vec_t*<*dim*> &eigs, const *Mat_t*<*dim*> &T)

sum implementation

namespace **Matrices**

Typedefs

using **Tens2_t** = *Eigen*::Matrix<*Real*, dim, dim>

second-order tensor representation

using **Tens4_t** = *T4Mat*<*Real*, dim>

fourth-order tensor representation

Functions

template<*Dim_t* **dim**>
inline constexpr *Tens2_t*<*dim*> **I2**()

compile-time second-order identity

template<typename **T1**, typename **T2**>
inline decltype(auto) constexpr **outer**(*T1* &&A, *T2* &&B)

compile-time outer tensor product as defined by Curnier R_ijkl = A_ij.B_klxx 0123 01 23

template<typename **Derived1**, typename **Derived2**>
inline decltype(auto) constexpr **outer_under**(const *Eigen*::MatrixBase<*Derived1*> &A, const *Eigen*::MatrixBase<*Derived2*> &B)

compile-time underlined outer tensor product as defined by Curnier R_ijkl = A_ik.B_jlxx 0123 02 13 0213 01 23 <- this defines the shuffle order

template<typename **T1**, typename **T2**>
inline decltype(auto) constexpr **outer_over**(*T1* &&A, *T2* &&B)

compile-time overlined outer tensor product as defined by Curnier R_ijkl = A_il.B_jkxx 0123 03 12 0231 01 23 <- this defines the shuffle order

template<typename **T4**, typename **T2**>

inline constexpr auto **tensmult**(const *Eigen*::MatrixBase<*T4*> &A, const *Eigen*::MatrixBase<*T2*> &B) -> *Tens2_t*<*T2*::RowsAtCompileTime>

Standart tensor multiplication

template<*Dim_t* **dim**>
inline constexpr *Tens4_t*<*dim*> **Itrac**()

compile-time fourth-order tracer

template<*Dim_t* **dim**>
inline constexpr *Tens4_t*<*dim*> **Iiden**()

compile-time fourth-order identity

template<*Dim_t* **dim**>
inline constexpr *Tens4_t*<*dim*> **Itrns**()

compile-time fourth-order transposer

template<*Dim_t* **dim**>
inline constexpr *Tens4_t*<*dim*> **Isymm**()

compile-time fourth-order symmetriser

template<*Dim_t* **Dim**, class **T1**, class **T2**>
decltype(auto) **dot**(*T1* &&t1, *T2* &&t2)

simple contraction between two tensors. The result depends on the rank of the tesnors, see documentation for `muGrid::internal::Dotter`

template<*Dim_t* **Dim**, class **T1**, class **T2**>
decltype(auto) **ddot**(*T1* &&t1, *T2* &&t2)

double contraction between two tensors. The result depends on the rank of the tesnors, see documentation for `muGrid::internal::Dotter`

namespace **internal**

namespace **Tensors**

Typedefs

using **Tens2_t** = *Eigen*::TensorFixedSize<*Real*, *Eigen*::Sizes<dim, dim>>

second-order tensor representation

using **Tens4_t** = *Eigen*::TensorFixedSize<*Real*, *Eigen*::Sizes<dim, dim, dim, dim>>

fourth-order tensor representation

Functions

template<*Dim_t* **dim**>
inline constexpr *Tens2_t*<*dim*> **I2**()

compile-time second-order identity

template<*Dim_t* **dim**, typename **T1**, typename **T2**>

inline decltype(auto) constexpr **outer**(*T1* &&A, *T2* &&B)

compile-time outer tensor product as defined by Curnier R_ijkl = A_ij.B_klxx 0123 01 23

template<*Dim_t* **dim**, typename **T1**, typename **T2**>
inline decltype(auto) constexpr **outer_under**(*T1* &&A, *T2* &&B)

compile-time underlined outer tensor product as defined by Curnier R_ijkl = A_ik.B_jlxx 0123 02 13 0213 01 23 <- this defines the shuffle order

template<*Dim_t* **dim**, typename **T1**, typename **T2**>
inline decltype(auto) constexpr **outer_over**(*T1* &&A, *T2* &&B)

compile-time overlined outer tensor product as defined by Curnier R_ijkl = A_il.B_jkxx 0123 03 12 0231 01 23 <- this defines the shuffle order

template<*Dim_t* **dim**>
inline constexpr *Tens4_t*<*dim*> **I4S**()

compile-time fourth-order symmetrising identity

namespace **muSpectre**

Typedefs

using **MatrixXXc** = *Eigen*::Matrix<Complex, *Eigen*::Dynamic, *Eigen*::Dynamic>

convenience alias

using **Grad_t** = Matrices::Tens2_t<Dim>

Field type that solvers expect gradients to be expressed in

using **LoadSteps_t** = std::vector<*Eigen*::MatrixXd>

Input type for specifying a load regime

Enums

enum **RotationOrder**

The rotation matrices depend on the order in which we rotate around different axes. See [[https://en.wikipedia.org/wiki/Euler_angles#Rotation_matrix]] to find the matrices

Values:

enumerator **Z**

enumerator **XZXEuler**

enumerator **XYXEuler**

enumerator **YXYEuler**

enumerator **YZYEuler**

enumerator **ZYZEuler**

enumerator **ZXZEuler**

enumerator **XZYTaitBryan**

enumerator **XYZTaitBryan**

enumerator **YXZTaitBryan**

enumerator **YZXTaitBryan**

enumerator **ZYXTaitBryan**

enumerator **ZXYTaitBryan**

enum **Formulation**

continuum mechanics flags

Values:

enumerator **finite_strain**

causes evaluation in PK1(F)

enumerator **small_strain**

causes evaluation in ()

enumerator **small_strain_sym**

symmetric storage as vector

enumerator **native**

causes the material's native measures to be used in evaluation

enum **SplitCell**

split cell flags

Values:

enumerator **laminate**

enumerator **simple**

enumerator **no**

enum **FiniteDiff**

finite differences flags

Values:

enumerator **forward**

f/x (f(x+x) - f(x))/x

enumerator **backward**

f/x (f(x) - f(x-x))/x

enumerator **centred**

f/x (f(x+x) - f(x-x))/2x

enum **StressMeasure**

Material laws can declare which type of stress measure they provide, and µSpectre will handle conversions

Values:

enumerator **Cauchy**

Cauchy stress

enumerator **PK1**

First Piola-Kirchhoff stress.

enumerator **PK2**

Second Piola-Kirchhoff stress.

enumerator **Kirchhoff**

Kirchhoff stress

enumerator **Biot**

Biot stress.

enumerator **Mandel**

Mandel stress.

enumerator **no_stress_**

only for triggering static_asserts

enum **StrainMeasure**

Material laws can declare which type of strain measure they require and µSpectre will provide it

Values:

enumerator **Gradient**

placement gradient (y/x)

enumerator **Infinitesimal**

small strain tensor .5(u + u)

enumerator **GreenLagrange**

Green-Lagrange strain .5(F·F - I)

enumerator **Biot**

Biot strain.

enumerator **Log**

logarithmic strain

enumerator **Almansi**

Almansi strain.

enumerator **RCauchyGreen**

Right Cauchy-Green tensor.

enumerator **LCauchyGreen**

Left Cauchy-Green tensor.

enumerator **no_strain_**

only for triggering static_assert

enum **ElasticModulus**

all isotropic elastic moduli to identify conversions, such as E = µ(3 + 2µ)/(+µ). For the full description, see https://en.wikipedia.org/wiki/Lam%C3%A9_parameters Not all the conversions are implemented, so please add as needed

Values:

enumerator **Bulk**

Bulk modulus K.

enumerator **K**

alias for `ElasticModulus::Bulk`

enumerator **Young**

Young's modulus E.

enumerator **E**

alias for `ElasticModulus::Young`

enumerator **lambda**

Lamé's first parameter

enumerator **Shear**

Shear modulus G or µ

enumerator **G**

alias for `ElasticModulus::Shear`

enumerator **mu**

alias for `ElasticModulus::Shear`

enumerator **Poisson**

Poisson's ratio

enumerator **nu**

alias for `ElasticModulus::Poisson`

enumerator **Pwave**

P-wave modulus M.

enumerator **M**

alias for `ElasticModulus::Pwave`

enumerator **no_modulus_**

enum **IsStrainInitialised**

Values:

enumerator **True**

enumerator **False**

Functions

template<class **FFTEngine** = *muFFT*::*FFTWEngine*>
inline std::unique_ptr<*ProjectionBase*> **cell_input**(const *DynCcoord_t* &nb_grid_pts, const *DynRcoord_t* &lengths, const *Formulation* &form, *muFFT*::*Gradient_t* gradient, const *muFFT*::*Communicator* &comm = *muFFT*::*Communicator*())

Convenience function to create consistent input for the constructor of * *muSpectre::Cell*. Creates a unique ptr to a Projection operator (with appropriate FFT_engine) to be used in a cell constructor

Parameters

- **nb_grid_pts** – resolution of the discretisation grid in each spatial directional
- **lengths** – length of the computational domain in each spatial direction
- **form** – problem formulation (small vs finite strain)

- **gradient** – gradient operator to use (i.e., "exact" Fourier derivation, finite differences, etc)
- **comm** – communicator used for solving distributed problems

template<class **FFTEngine** = *muFFT*::*FFTWEngine*>
inline std::unique_ptr<*ProjectionBase*> **cell_input**(const *DynCcoord_t* &nb_grid_pts, const *DynRcoord_t* &lengths, const *Formulation* &form, const *muFFT*::*Communicator* &comm = *muFFT*::*Communicator*())

Convenience function to create consistent input for the constructor of * `muSpectre::Cell`. Creates a unique ptr to a Projection operator (with appropriate FFT_engine) to be used in a cell constructor. Uses the "exact" fourier derivation operator for calculating gradients

Parameters

- **nb_grid_pts** – resolution of the discretisation grid in each spatial directional
- **lengths** – length of the computational domain in each spatial direction
- **form** – problem formulation (small vs finite strain)
- **comm** – communicator used for solving distributed problems

template<typename **Cell_t** = *Cell*, class **FFTEngine** = *muFFT*::*FFTWEngine*>
inline *Cell_t* **make_cell**(*DynCcoord_t* nb_grid_pts, *DynRcoord_t* lengths, *Formulation* form, *muFFT*::*Gradient_t* gradient, const *muFFT*::*Communicator* &comm = *muFFT*::*Communicator*())

convenience function to create a cell (avoids having to build and move the chain of unique_ptrs

Parameters

- **nb_grid_pts** – resolution of the discretisation grid in each spatial directional
- **lengths** – length of the computational domain in each spatial direction
- **form** – problem formulation (small vs finite strain)
- **gradient** – gradient operator to use (i.e., "exact" Fourier derivation, finite differences, etc)
- **comm** – communicator used for solving distributed problems

template<typename **Cell_t** = *Cell*, class **FFTEngine** = *muFFT*::*FFTWEngine*>
inline *Cell_t* **make_cell**(*DynCcoord_t* nb_grid_pts, *DynRcoord_t* lengths, *Formulation* form, const *muFFT*::*Communicator* &comm = *muFFT*::*Communicator*())

convenience function to create a cell (avoids having to build and move the chain of unique_ptrs. Uses the "exact" fourier derivation operator for calculating gradients

Parameters

- **nb_grid_pts** – resolution of the discretisation grid in each spatial directional
- **lengths** – length of the computational domain in each spatial direction
- **form** – problem formulation (small vs finite strain)
- **comm** – communicator used for solving distributed problems

template<typename **Cell_t** = *CellSplit*, class **FFTEngine** = *muFFT*::*FFTWEngine*>
inline *Cell_t* **make_cell_split**(*DynCcoord_t* nb_grid_pts, *DynRcoord_t* lengths, *Formulation* form, *muFFT*::*Gradient_t* gradient, const *muFFT*::*Communicator* &comm = *muFFT*::*Communicator*())

template<typename **Cell_t** = *CellSplit*, class **FFTEngine** = *muFFT*::*FFTWEngine*>
std::unique_ptr<*Cell_t*> **make_cell_ptr**(const *DynCcoord_t* &nb_grid_pts, const *DynRcoord_t* &lengths, const *Formulation* &form, *muFFT*::*Gradient_t* gradient, const *muFFT*::*Communicator* &comm = *muFFT*::*Communicator*())

std::ostream &**operator<<**(std::ostream &os, *Formulation* f)

inserts `muSpectre::Formulations` into `std::ostreams`

std::ostream &**operator<<**(std::ostream &os, *StressMeasure* s)

inserts `muSpectre::StressMeasures` into `std::ostreams`

std::ostream &**operator<<**(std::ostream &os, *StrainMeasure* s)

inserts `muSpectre::StrainMeasures` into `std::ostreams`

void **banner**(std::string name, Uint year, std::string cpy_holder)

Copyright banner to be printed to the terminal by executables Arguments are the executable's name, year of writing and the name

- address of the copyright holder

template<bool **sym** = true>
constexpr Dim_t **vsize**(Dim_t dim)

compile time computation of voigt vector

constexpr Dim_t **dof_for_formulation**(const *Formulation* form, const Dim_t dim)

compute the number of degrees of freedom to store for the strain tensor given dimension dim

inline constexpr bool **operator<**(*ElasticModulus* A, *ElasticModulus* B)

define comparison in order to exploit that moduli can be expressed in terms of any two other moduli in any order (e.g. K = K(E,) = K(, E)

constexpr *StrainMeasure* **get_stored_strain_type**(*Formulation* form)

Compile-time function to g strain measure stored by muSpectre depending on the formulation

constexpr *StressMeasure* **get_stored_stress_type**(*Formulation* form)

Compile-time function to g stress measure stored by muSpectre depending on the formulation

constexpr *StrainMeasure* **get_formulation_strain_type**(*Formulation* form, *StrainMeasure* expected)

Compile-time functions to get the stress and strain measures after they may have been modified by choosing a formulation.

For instance, a law that expecs a Green-Lagrange strain as input will get the infinitesimal strain tensor instead in a small strain computation

template<typename **T**>
T **modulo**(*T* a, *T* b)

bool **check_symmetry**(const *Eigen*::Ref<const *Eigen*::ArrayXXd> &eps, Real rel_tol)

check whether a strain is symmetric, for the purposes of small strain problems

Eigen::IOFormat format (Eigen::FullPrecision, 0, ", ", ",\n", "[", "]", "[", "]")

produces numpy-compatible full precision text output. great for debugging

std::vector<*OptimizeResult*> **newton_cg**(*Cell* &cell, const *LoadSteps_t* &load_steps, *SolverBase* &solver, Real newton_tol, Real equil_tol, Dim_t verbose, *IsStrainInitialised* strain_init)

Uses the Newton-conjugate Gradient method to find the static equilibrium of a cell given a series of mean applied strain(for Formulation::small_strain and H (=F-I) for Formulation::finite_strain). The initial macroscopic strain state is set to zero in cell initialisation.

std::vector<*OptimizeResult*> **de_geus**(*Cell* &cell, const *LoadSteps_t* &load_steps, *SolverBase* &solver, Real newton_tol, Real equil_tol, Dim_t verbose, *IsStrainInitialised* strain_init)

Uses the method proposed by de Geus method to find the static given a series of mean applied strain(for Formulation::small_strain and H (=F-I) for Formulation::finite_strain). The initial macroscopic strain state is set to zero in cell initialisation.

inline *OptimizeResult* **newton_cg**(*Cell* &cell, const *Eigen*::Ref<*Eigen*::MatrixXd> load_step, *SolverBase* &solver, Real newton_tol, Real equil_tol, Dim_t verbose = 0, *IsStrainInitialised* strain_init = *IsStrainInitialised*::*False*)

Uses the Newton-conjugate Gradient method to find the static equilibrium of a cell given a mean applied strain.

inline *OptimizeResult* **de_geus**(*Cell* &cell, const *Eigen*::Ref<*Eigen*::MatrixXd> load_step, *SolverBase* &solver, Real newton_tol, Real equil_tol, Dim_t verbose = 0, *IsStrainInitialised* strain_init = *IsStrainInitialised*::*False*)

Uses the method proposed by de Geus method to find the static equilibrium of a cell given a mean applied strain.

namespace **internal**

Functions

template<size_t **DimS**, class **FFTEngine**>
inline std::unique_ptr<*ProjectionBase*> **cell_input_helper**(const *DynCcoord_t* &nb_grid_pts, const *DynRcoord_t* &lengths, const *Formulation* &form, *muFFT*::*Gradient_t* gradient, const *muFFT*::*Communicator* &comm = *muFFT*::*Communicator*())

function to create consistent input for the constructor of *muSpectre::Cell*. Users should never need to call this function, for internal use only

namespace **MatTB**

Enums

enum **NeedTangent**

Flag used to designate whether the material should compute both stress and tangent moduli or only stress

Values:

enumerator **yes**

compute both stress and tangent moduli

enumerator **no**

compute only stress

Functions

template<*StrainMeasure* **In**, *StrainMeasure* **Out**, class **Strain_t**>
decltype(auto) **convert_strain**(*Strain_t* &&strain)

set of functions returning one strain measure as a function of another

template<*ElasticModulus* **Out**, *ElasticModulus* **In1**, *ElasticModulus* **In2**>
inline constexpr Real **convert_elastic_modulus**(const Real &in1, const Real &in2)

allows the conversion from any two distinct input moduli to a chosen output modulus

template<Dim_t **Dim**, *FiniteDiff* **FinDif** = *FiniteDiff*::*centred*, class **FunType**, class **Derived**>
inline *muGrid*::*T4Mat*<Real, *Dim*> **compute_numerical_tangent**(*FunType* &&fun, const *Eigen*::MatrixBase<*Derived*> &strain, Real delta)

Helper function to numerically determine tangent, intended for testing, rather than as a replacement for analytical tangents

template<Dim_t **DimM**>
inline *Eigen*::Matrix<Real, *DimM*, *DimM*> **compute_deviatoric_stress**(const *Eigen*::Matrix<Real, *DimM*, *DimM*> &PK2)

Computes the deviatoric stress _{dev}=-{1}{3} tr()*I, on each pixel from a given stress, first only for PK2.

template<Dim_t **DimM**>
inline decltype(auto) **compute_equivalent_von_Mises_stress**(const *Eigen*::Map<const *Eigen*::Matrix<Real, *DimM*, *DimM*>> PK2)

Computes the equivalent von Mises stress _{eq} on each pixel from a given PK2 stress.

template<*Formulation* **Form**, class **Material**, class **Strains**, class **Stresses**>
void **constitutive_law**(*Material* &mat, *Strains* &&strains, *Stresses* &stresses, const size_t &quad_pt_id, const Real &ratio)

template<*Formulation* **Form**, class **Material**, class **Strains**, class **Stresses**>
void **constitutive_law**(*Material* &mat, *Strains* &&strains, *Stresses* &stresses, const size_t &quad_pt_id)

template<*Formulation* **Form**, class **Material**, class **Strains**, class **Stresses**>
void **constitutive_law_tangent**(*Material* &mat, *Strains* &&strains, *Stresses* &stresses, const size_t &quad_pt_id)

template<*Formulation* **Form**, class **Material**, class **Strains**, class **Stresses**>
void **constitutive_law_tangent**(*Material* &mat, *Strains* &&strains, *Stresses* &stresses, const size_t &quad_pt_id, const Real &ratio)

template<Dim_t **DimM**, class **Derived1**, class **Derived2**>
void **make_C_from_C_voigt**(const *Eigen*::MatrixBase<*Derived1*> &C_voigt, *Eigen*::MatrixBase<*Derived2*> &C_holder)

template<*StressMeasure* **StressM**, *StrainMeasure* **StrainM**, class **Stress_t**, class **Strain_t**>
decltype(auto) **PK1_stress**(*Strain_t* &&strain, *Stress_t* &&stress)

set of functions returning an expression for PK1 stress based on

template<*StressMeasure* **StressM**, *StrainMeasure* **StrainM**, class **Stress_t**, class **Strain_t**, class **Tangent_t**>
decltype(auto) **PK1_stress**(*Strain_t* &&strain, *Stress_t* &&stress, *Tangent_t* &&tangent)

set of functions returning an expression for PK1 stress based on

template<*StressMeasure* **StressM**, *StrainMeasure* **StrainM**, class **Stress_t**, class **Strain_t**>

decltype(auto) **PK2_stress**(*Strain_t* &&strain, *Stress_t* &&stress)

set of functions returning an expression for PK2 stress based on

template<*StressMeasure* **StressM**, *StrainMeasure* **StrainM**, class **Stress_t**, class **Strain_t**, class **Tangent_t**>
decltype(auto) **PK2_stress**(*Strain_t* &&strain, *Stress_t* &&stress, *Tangent_t* &&tangent)

set of functions returning an expression for PK2 stress based on

template<*StressMeasure* **StressM**, *StrainMeasure* **StrainM**, class **Stress_t**, class **Strain_t**>
decltype(auto) **Kirchhoff_stress**(*Strain_t* &&strain, *Stress_t* &&stress)

set of functions returning an expression for Kirchhoff stress based on

namespace **internal**

namespace **std_replacement**

Functions

template<class **F**, class ...**ArgTypes**>
auto **invoke**(*F* &&f, *ArgTypes*&&... args) noexcept(noexcept(*detail*::*INVOKE*(std::forward<*F*>(*f*), std::forward<*ArgTypes*>(*args*)...))) -> decltype(*detail*::*INVOKE*(std::forward<*F*>(*f*), std::forward<*ArgTypes*>(*args*)...))

from cppreference

template<class **F**, class **Tuple**>
decltype(auto) constexpr **apply**(*F* &&f, *Tuple* &&t)

from cppreference

namespace **detail**

Functions

template<class **Base**, class **T**, class **Derived**, class ...**Args**>
auto **INVOKE**(*T Base*::* pmf, *Derived* &&ref, *Args*&&... args) noexcept(noexcept((std::forward<*Derived*>(*ref*) .* *pmf*)(std::forward<*Args*>(*args*)...))) -> std::enable_if_t<std::is_function<*T*>::value && std::is_base_of<*Base*, std::decay_t<*Derived*>>::value, decltype((std::forward<*Derived*>(*ref*) .* *pmf*)(std::forward<*Args*>(*args*)...))>

from cppreference

template<class **Base**, class **T**, class **RefWrap**, class ...**Args**>
auto **INVOKE**(*T Base*::* pmf, *RefWrap* &&ref, *Args*&&... args) noexcept(noexcept((*ref*.get() .* *pmf*)(std::forward<*Args*>(*args*)...))) -> std::enable_if_t<std::is_function<*T*>::value && *is_reference_wrapper*<std::decay_t<*RefWrap*>>::value, decltype((*ref*.get() .* *pmf*)(std::forward<*Args*>(*args*)...))>

from cppreference

template<class **Base**, class **T**, class **Pointer**, class ...**Args**>

auto **INVOKE**(*T Base*::* pmf, *Pointer* &&ptr, *Args*&&... args)
noexcept(noexcept(((*std::forward<*Pointer*>(*ptr*)) .* *pmf*)(std::forward<*Args*>(*args*)...))) ->
std::enable_if_t<std::is_function<*T*>::value &&
!*is_reference_wrapper*<std::decay_t<*Pointer*>>::value && !std::is_base_of<*Base*,
std::decay_t<*Pointer*>>::value, decltype(((*std::forward<*Pointer*>(*ptr*)) .*
pmf)(std::forward<*Args*>(*args*)...))>

from cppreference

template<class **Base**, class **T**, class **Derived**>
auto **INVOKE**(*T Base*::* pmd, *Derived* &&ref) noexcept(noexcept(std::forward<*Derived*>(*ref*) .* *pmd*)) ->
std::enable_if_t<!std::is_function<*T*>::value && std::is_base_of<*Base*,
std::decay_t<*Derived*>>::value, decltype(std::forward<*Derived*>(*ref*) .* *pmd*)>

from cppreference

template<class **Base**, class **T**, class **RefWrap**>
auto **INVOKE**(*T Base*::* pmd, *RefWrap* &&ref) noexcept(noexcept(*ref*.get() .* *pmd*)) ->
std::enable_if_t<!std::is_function<*T*>::value &&
is_reference_wrapper<std::decay_t<*RefWrap*>>::value, decltype(*ref*.get() .* *pmd*)>

from cppreference

template<class **Base**, class **T**, class **Pointer**>
auto **INVOKE**(*T Base*::* pmd, *Pointer* &&ptr) noexcept(noexcept((*std::forward<*Pointer*>(*ptr*)) .* *pmd*)) ->
std::enable_if_t<!std::is_function<*T*>::value &&
!*is_reference_wrapper*<std::decay_t<*Pointer*>>::value && !std::is_base_of<*Base*,
std::decay_t<*Pointer*>>::value, decltype((*std::forward<*Pointer*>(*ptr*)) .* *pmd*)>

from cppreference

template<class **F**, class ...**Args**>
auto **INVOKE**(*F* &&f, *Args*&&... args) noexcept(noexcept(std::forward<*F*>(*f*)(std::forward<*Args*>(*args*)...)))
-> std::enable_if_t<!std::is_member_pointer<std::decay_t<*F*>>::value,
decltype(std::forward<*F*>(*f*)(std::forward<*Args*>(*args*)...))>

from cppreference

template<class **F**, class **Tuple**, std::size_t... **I**>
decltype(auto) constexpr **apply_impl**(*F* &&f, *Tuple* &&t, std::index_sequence<*I*...>)

from cppreference

file **cell.cc**

#include "cell_adaptor.hh"*#include* "cell.hh"*#include* <libmugrid/field_map.hh>*#include* <libmugrid/field_map_static.hh>*#include* <*set*> implementation for the Cell class

Author

Till Junge till.junge@altermail.ch

Date

05 Oct 2019

You should have received a copy of the GNU Lesser General Public License along with µSpectre; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **cell.hh**

#include "common/muSpectre_common.hh"*#include* "materials/material_base.hh"*#include* "projection/projection_base.hh"*#include* <libmugrid/ccoord_operations.hh>*#include* <*memory*> Class for the representation of a homogenisation problem in µSpectre.

Copyright © 2019 Till Junge

Author
Till Junge till.junge@epfl.ch

Date
13 Sep 2019

µSpectre is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µSpectre is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µSpectre; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **cell_adaptor.hh**

#include <common/muSpectre_common.hh>*#include* <*Eigen/IterativeLinearSolvers*> Cell Adaptor implements the matrix-vector multiplication and allows the adapted cell to be used like a spacse matrix in conjugate-gradient-type solvers.

Copyright © 2019 Till Junge

Author
Till Junge till.junge@epfl.ch

Date
13 Sep 2019

µSpectre is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µSpectre is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µSpectre; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **cell_factory.hh**

#include "common/muSpectre_common.hh"#include "cell/cell.hh"#include "projection/projection_finite_strain_fast.hh"#include "projection/projection_small_strain.hh"#include <libmugrid/ccoord_operations.hh>*#include* <libmufft/derivative.hh>*#include* <libmufft/fftw_engine.hh>*#include* <*memory*> Cell factories to help create cells with ease.

Copyright © 2017 Till Junge

Author
Till Junge till.junge@epfl.ch

Date
15 Dec 2017

µSpectre is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µSpectre is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µSpectre; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330,

- Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **cell_split.cc**

*#include "*cell/cell_split.hh*"* Implementation for cell base class.

Copyright © 2017 Till Junge

Author
Ali Falsafi ali.faslafi@epfl.ch

Date
10 Dec 2019

µSpectre is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µSpectre is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µSpectre; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **cell_split.hh**

#include "cell/cell.hh"*#include* "common/muSpectre_common.hh"*#include* "common/intersection_octree.hh"*#include* "materials/material_base.hh"*#include* "projection/projection_base.hh"*#include* "libmugrid/ccoord_operations.hh"*#include* "libmugrid/field.hh"*#include* <*vector*>*#include* <*memory*>*#include* <*tuple*>*#include* <*functional*>*#include* <*sstream*>*#include* <*algorithm*> Base class representing a unit cell able to handle split material assignments.

Copyright © 2017 Till Junge

Author
Ali Falsafi ali.falsafi@epfl.ch

Date
10 Dec 2019

µSpectre is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µSpectre is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µSpectre; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **cell_split_factory.hh**

#include "common/muSpectre_common.hh"*#include* "libmugrid/ccoord_operations.hh"*#include* "cell/cell_split.hh"*#include* "projection/projection_finite_strain_fast.hh"*#include* "projection/projection_small_strain.hh"*#include* "libmufft/fftw_engine.hh"*#include* "cell/cell_factory.hh"*#include* <*memory*> Implementation for cell base class.

Author
Ali Falsafi ali.falsafi@epfl.ch

Date
01 Nov 2018

file **cell_traits.hh**

*#include "*common/muSpectre_common.hh*"#include <Eigen/IterativeLinearSolvers>* Provides traits for Eigen solvers to be able to use Cells.

Author
Till Junge till.junge@epfl.ch

Date
19 Jan 2018

file **common.cc**

*#include "*common/muSpectre_common.hh*"#include <stdexcept>#include <iostream>* Implementation for common functions.

Author
Till Junge till.junge@epfl.ch

Date
15 Nov 2017

file **geometry.hh**

#include "common/muSpectre_common.hh"#include <libmugrid/tensor_algebra.hh>#include <libmugrid/eigen_tools.hh>#include <Eigen/Dense>#include <Eigen/Geometry>#include <array>#include <memory> Geometric calculation helpers.

Author
Till Junge till.junge@altermail.ch

Date
18 Apr 2018

file **intersection_octree.cc**

#include "common/intersection_octree.hh" Oct tree for obtaining and calculating the intersection with pixels.

Author
Ali Falsafi ali.falsafi@epfl.ch

Date
May 2018

file **intersection_octree.hh**

#include "common/muSpectre_common.hh"*#include* "cell/cell.hh"*#include* "materials/material_base.hh"*#include* "common/intersection_volume_calculator_corkpp.hh"*#include* "libmugrid/ccoord_operations.hh"*#include* <*vector*>*#include* <*array*>*#include* <*algorithm*> octree algorithm employed to accelerate precipitate pixel assignment

Author
Ali Falsafi ali.falsafi@epfl.ch

Date
May 2018

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **intersection_volume_calculator_corkpp.hh**

*#include "cork_interface.hh"#include "*libmugrid/grid_common.hh*"#include <vector>#include <fstream>#include <math.h>* Calculation of the intersection volume of percipitates and pixles.

Copyright © 2018 Ali Falsafi

Author
Ali Falsafi ali.falsafi@epfl.ch

Date
04 June 2018

µSpectre is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µSpectre is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µSpectre; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **muSpectre_common.hh**

#include <libmugrid/grid_common.hh>*#include* <libmugrid/tensor_algebra.hh>*#include* <libmufft/mufft_common.hh>*#include <string>* Small definitions of commonly used types throughout µSpectre.

Author
Till Junge till.junge@epfl.ch

Date
01 May 2017

7.1 LICENSE

Copyright © 2017 Till Junge

µSpectre is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µSpectre is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µSpectre; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file `voigt_conversion.cc`

*#include "*common/voigt_conversion.hh*"* specializations for static members of voigt converter

Copyright © 2017 Till Junge

Author
Till Junge till.junge@epfl.ch

Author
Ali Falsafi ali.falsafi@epfl.ch

Date
04 May 2017

µSpectre is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µSpectre is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µSpectre; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330,

- Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file `voigt_conversion.hh`

*#include "*common/muSpectre_common.hh*"#include <Eigen/Dense>#include <unsupported/Eigen/CXX11/Tensor>#include <iostream>* utilities to transform vector notation arrays into voigt notation arrays and vice-versa

Copyright © 2017 Till Junge

Author
Till Junge till.junge@epfl.ch

Date
02 May 2017

µSpectre is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µSpectre is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µSpectre; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file `communicator.cc`

*#include "*communicator.hh*"#include <sstream>* implementation for mpi abstraction layer

Copyright © 2019 Till Junge

Author
Till Junge till.junge@epfl.ch

Date
02 Oct 2019

µFFT is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µFFT is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µFFT; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file `communicator.hh`

*#include <type_traits>#include "*mufft_common.hh*"#include <Eigen/Dense>* abstraction layer for the distributed memory communicator object

Copyright © 2017 Till Junge

Author
Lars Pastewka lars.pastewka@imtek.uni-freiburg.de

Date
07 Mar 2018

µFFT is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µFFT is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µFFT; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330,

- Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **derivative.cc**

*#include <iostream>#include "*derivative.hh*"*

file **derivative.cc**

#include "projection/derivative.hh"

file **derivative.hh**

*#include <memory>#include "*common/muSpectre_common.hh*"#include <*libmugrid/ccoord_operations.hh*>*
Representation of finite-differences stencils.

Copyright © 2019 Lars Pastewka

Author
Richard Leute richard.leute@imtek.uni-freiburg.de Lars Pastewka lars.pastewka@imtek.uni-freiburg.de

Date
05 June 2019

µFFT is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µFFT is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µFFT; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **fft_engine_base.cc**

#include “fft_engine_base.hh”*#include* “fft_utils.hh” implementation for FFT engine base class

Author
Till Junge till.junge@altermail.ch

Date
03 Dec 2017

file **fft_engine_base.hh**

#include <libmugrid/ccoord_operations.hh>*#include* <libmugrid/field_collection_global.hh>*#include* <libmugrid/field_typed.hh>*#include* “communicator.hh”*#include* “mufft_common.hh” Interface for FFT engines.

Author
Till Junge till.junge@epfl.ch

Date
01 Dec 2017

file **fft_utils.cc**

*#include "*fft_utils.hh*"* implementation of fft utilities

Author
Till Junge till.junge@altermail.ch

Date
11 Dec 2017

file **fft_utils.hh**

*#include "*mufft_common.hh*"#include <Eigen/Dense>#include <array>#include <valarray>* collection of functions used in the context of spectral operations

Author
Till Junge till.junge@epfl.ch

Date
06 Dec 2017

file **fftw_engine.cc**

*#include <sstream>#include "*fftw_engine.hh*"#include <*libmugrid/ccoord_operations.hh*>* implements the fftw engine

Author
Till Junge till.junge@altermail.ch

Date
03 Dec 2017

file **fftw_engine.hh**

*#include "*fft_engine_base.hh*"#include <fftw3.h>* FFT engine using FFTW.

Author
Till Junge till.junge@altermail.ch

Date
03 Dec 2017

file **fftwmpi_engine.cc**

*#include "*fftwmpi_engine.hh*"#include <*libmugrid/ccoord_operations.hh*>* implements the MPI-parallel fftw engine

Author
Lars Pastewka lars.pastewka@imtek.uni-freiburg.de

Date
06 Mar 2017

file **fftwmpi_engine.hh**

*#include "*fft_engine_base.hh*"#include <fftw3-mpi.h>* FFT engine using MPI-parallel FFTW.

Author
Lars Pastewka lars.pastewka@imtek.uni-freiburg.de

Date
06 Mar 2017

file **mufft_common.hh**

#include <libmugrid/grid_common.hh> Small definitions of commonly used types throughout µFFT.

Copyright © 2019 Till Junge

Author
Till Junge till.junge@epfl.ch

Date
24 Jan 2019

µFFT is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µFFT is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µFFT; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **pfft_engine.cc**

*#include "*pfft_engine.hh*"#include* <libmugrid/ccoord_operations.hh> implements the MPI-parallel pfft engine

Copyright © 2017 Till Junge

Author
Lars Pastewka lars.pastewka@imtek.uni-freiburg.de

Date
06 Mar 2017

µFFT is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µFFT is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µFFT; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330,

- Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **pfft_engine.hh**

*#include "*fft_engine_base.hh*"#include <pfft.h>* FFT engine using MPI-parallel PFFT.

Author
Lars Pastewka lars.pastewka@imtek.uni-freiburg.de

Date
06 Mar 2017

file **ccoord_operations.cc**

*#include <iostream>#include "*ccoord_operations.hh*"* pre-compilable pixel operations

Author
Till Junge till.junge@epfl.ch

Date
01 Oct 2019

file **ccoord_operations.hh**

#include <functional>#include <numeric>#include <utility>#include <Eigen/Dense>#include "grid_common.hh"#include "iterators.hh" common operations on pixel addressing

Copyright © 2017 Till Junge

Author
Till Junge till.junge@epfl.ch

Date
29 Sep 2017

µGrid is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µGrid is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µGrid; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330,

- Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **cpp_compliance.hh**

#include <tuple>#include <experimental/optional> additions to the standard name space to anticipate C++17 features

Copyright © 2017 Till Junge

Author
Till Junge till.junge@epfl.ch

Date
17 Nov 2017

µGrid is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µGrid is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µGrid; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330,

- Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **eigen_tools.hh**

*#include "*grid_common.hh*"#include <Eigen/Dense>#include <unsupported/Eigen/CXX11/Tensor>#include <type_traits>#include <utility>* small tools to be used with Eigen

Copyright © 2017 Till Junge

Author
Till Junge till.junge@epfl.ch

Date
20 Sep 2017

µGrid is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µGrid is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µGrid; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330,

- Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **field.cc**

*#include "*field.hh*"#include "*field_collection.hh*"#include "*field_collection_global.hh*"* implementation of Field

Copyright © 2019 Till Junge

Author
Till Junge till.junge@altermail.ch

Date
11 Aug 2019

µGrid is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µGrid is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µGrid; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file `field.hh`

*#include "*grid_common.hh*"#include <string>#include <typeinfo>* Base class for fields.

Copyright © 2019 Till Junge

Author
Till Junge till.junge@altermail.ch

Date
10 Aug 2019

µGrid is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µGrid is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µGrid; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file `field_collection.cc`

*#include "*field_collection.hh*"#include "*field.hh*"#include "*state_field.hh*"#include "*field_typed.hh*"* Implementations for field collections.

Copyright © 2019 Till Junge

Author
Till Junge till.junge@altermail.ch

Date
11 Aug 2019

µGrid is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µGrid is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µGrid; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **field_collection.hh**

*#include "*grid_common.hh*"#include <map>#include <string>#include <memory>#include <sstream>#include <stdexcept>#include <vector>* Base class for field collections.

Copyright © 2019 Till Junge

Author
Till Junge till.junge@altermail.ch

Date
10 Aug 2019

µGrid is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µGrid is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µGrid; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **field_collection_global.cc**

*#include "*field_collection_global.hh*"#include <iostream>* Implementation of GlobalFieldCollection.

Copyright © 2019 Till Junge

Author
Till Junge till.junge@altermail.ch

Date
11 Aug 2019

µGrid is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µGrid is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µGrid; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file `field_collection_global.hh`

*#include "*field_collection.hh*"#include "*ccoord_operations.hh*"* Global field collections.

Copyright © 2019 Till Junge

Author
Till Junge till.junge@altermail.ch

Date
11 Aug 2019

µGrid is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µGrid is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µGrid; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file `field_collection_local.cc`

*#include "*field_collection_local.hh*"* implementation of local field collection

Copyright © 2019 Till Junge

Author
Till Junge till.junge@epfl.ch

Date
12 Aug 2019

µGrid is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µGrid is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µGrid; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **field_collection_local.hh**

*#include "*field_collection.hh*"#include "*field_collection_global.hh*"* Local field collection.

Copyright © 2019 Till Junge

Author
Till Junge till.junge@epfl.ch

Date
12 Aug 2019

µGrid is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µGrid is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µGrid; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **field_map.cc**

*#include "*field_map.hh*"#include "*field_typed.hh*"#include "*field_collection.hh*"#include "*iterators.hh*"#include <sstream>#include <iostream>* Implementation for basic FieldMap.

Copyright © 2019 Till Junge

Author
Till Junge till.junge@epfl.ch

Date
15 Aug 2019

µGrid is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µGrid is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µGrid; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **field_map.hh**

#include "grid_common.hh"#include "iterators.hh"#include "field_collection.hh"#include <type_traits>#include <memory>#include <functional> Implementation of the base class of all field maps.

Copyright © 2019 Till Junge

Author
Till Junge till.junge@epfl.ch

Date
15 Aug 2019

µGrid is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µGrid is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µGrid; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **field_map_static.hh**

#include "field.hh"#include "field_typed.hh"#include "field_map.hh"#include "T4_map_proxy.hh"#include <sstream> header-only implementation of field maps with statically known iterate sizes

Copyright © 2019 Till Junge

Author
Till Junge till.junge@epfl.ch

Date
20 Aug 2019

µGrid is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µGrid is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µGrid; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **field_typed.cc**

*#include <sstream>#include "*field_typed.hh*"#include "*field_collection.hh*"#include "*field_map.hh*"* Implementation for typed fields.

Copyright © 2019 Till Junge

Author
Till Junge till.junge@epfl.ch

Date
13 Aug 2019

µGrid is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µGrid is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µGrid; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **field_typed.hh**

*#include "*field.hh*"#include "*grid_common.hh*"#include <Eigen/Dense>#include <vector>#include <memory>* Field classes for which the scalar type has been defined.

Copyright © 2019 Till Junge

Author
Till Junge till.junge@altermail.ch

Date
10 Aug 2019

µGrid is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µGrid is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µGrid; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **grid_common.hh**

#include <Eigen/Dense>#include <array>#include <cmath>#include <complex>#include <type_traits>#include <initializer_list>#include <algorithm>#include <vector>#include "cpp_compliance.hh" Small definitions of commonly used types throughout µgrid.

Copyright © 2019 Till Junge

Author
Till Junge till.junge@epfl.ch

Date
24 Jan 2019

µGrid is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µGrid is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µGrid; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **iterators.hh**

#include <tuple>#include <utility> iterator interfaces

Copyright (©) 2010-2011 EPFL (Ecole Polytechnique Fédérale de Lausanne) Laboratory (LSMS - Laboratoire de Simulation en Mécanique des Solides)

Author
Nicolas Richart

Date
creation Wed Jul 19 2017

Akantu is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

Akantu is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with Akantu. If not, see http://www.gnu.org/licenses/.

Above block was left intact as in akantu. µGrid exercises the right to redistribute and modify the code below

file `mapped_field.hh`

#include "field_map_static.hh"*#include* "field_collection.hh"*#include* "field_typed.hh"*#include* <*string*> convenience class to deal with data structures common to most internal variable fields in materials

Copyright © 2019 Till Junge

Author
Till Junge till.junge@epfl.ch

Date
04 Sep 2019

µGrid is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µGrid is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µGrid; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file `mapped_state_field.hh`

#include "state_field_map_static.hh"*#include* "state_field.hh"*#include* "field_collection.hh" Convenience class extending the mapped field concept to state fields.

Copyright © 2019 Till Junge

Author
Till Junge till.junge@epfl.ch

Date
09 Sep 2019

µGrid is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µGrid is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µGrid; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **numpy_tools.hh**

#include <algorithm>#include <pybind11/numpy.h>#include "field_typed.hh"*#include* "field_collection_global.hh" Convenience function for working with (pybind11's) numpy arrays.

Author
Lars Pastewka lars.pastewka@imtek.uni-freiburg.de

Author
Till Junge till.junge@epfl.ch

Date
02 Dec 2019

file **ref_array.hh**

#include <array>#include <initializer_list>#include "iterators.hh" convenience class to simulate an array of references

Author
Till Junge till.junge@epfl.ch

Date
04 Dec 2018

You should have received a copy of the GNU Lesser General Public License along with µGrid; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **ref_vector.hh**

#include <*vector*> convenience class providing a vector of references

Copyright © 2019 Till Junge

Author
Till Junge till.junge@epfl.ch

Date
21 Aug 2019

µGrid is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µGrid is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µGrid; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **state_field.cc**

*#include "*state_field.hh*"#include "*field.hh*"#include "*field_typed.hh*"#include "*field_collection.hh*"#include* <*sstream*> implementation for state fields

Copyright © 2019 Till Junge

Author
Till Junge till.junge@altermail.ch

Date
20 Aug 2019

µGrid is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µGrid is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µGrid; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file `state_field.hh`

#include "grid_common.hh"#include "ref_vector.hh"#include "state_field_map.hh"#include <string>#include <vector>#include <utility> A state field is an abstraction of a field that can hold current, as well as a chosen number of previous values. This is useful for instance for internal state variables in plastic laws, where a current, new, or trial state is computed based on its previous state, and at convergence, this new state gets cycled into the old, the old into the old-1 etc. The state field abstraction helps doing this safely (i.e. only const references to the old states are available, while the current state can be assigned to/modified), and efficiently (i.e., no need to copy values from new to old, we just cycle the labels). This file implements the state field as well as state maps using the Field, FieldCollection and FieldMap abstractions of µGrid.

Copyright © 2019 Till Junge

Author
Till Junge till.junge@altermail.ch

Date
20 Aug 2019

µGrid is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µGrid is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µGrid; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file `state_field_map.cc`

#include "state_field_map.hh"#include "state_field.hh"#include "field_map.hh"#include "field_typed.hh"#include "field_collection.hh"#include "field.hh" implementation of state field maps

Copyright © 2019 Till Junge

Author
Till Junge till.junge@epfl.ch

Date
22 Aug 2019

µGrid is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µGrid is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µGrid; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **state_field_map.hh**

*#include "*grid_common.hh*"#include "*field_map.hh*"#include "*ref_vector.hh*"#include <vector>#include <memory>* implementation of state field maps

Copyright © 2019 Till Junge

Author
Till Junge till.junge@epfl.ch

Date
22 Aug 2019

µGrid is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µGrid is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µGrid; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **state_field_map_static.hh**

*#include "*state_field_map.hh*"#include "*field_map_static.hh*"#include "*field_typed.hh*"#include <array>#include <sstream>#include <utility>* header-only implementation of state field maps with statically known iterate sizes

Copyright © 2019 Till Junge

Author
Till Junge till.junge@epfl.ch

Date
27 Aug 2019

µGrid is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µGrid is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µGrid; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **T4_map_proxy.hh**

*#include "*eigen_tools.hh*"#include <Eigen/Dense>#include <Eigen/src/Core/util/Constants.h>#include <type_traits>* Map type to allow fourth-order tensor-like maps on 2D matrices.

Copyright © 2017 Till Junge

Author
Till Junge till.junge@altermail.ch

Date
19 Nov 2017

µGrid is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µGrid is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µGrid; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330,

- Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **tensor_algebra.hh**

*#include "*grid_common.hh*"#include "*T4_map_proxy.hh*"#include "*eigen_tools.hh*"#include <Eigen/Dense>#include <unsupported/Eigen/CXX11/Tensor>#include <type_traits>* collection of compile-time quantities and algrebraic functions for tensor operations

Copyright © 2017 Till Junge

Author
Till Junge till.junge@epfl.ch

Date
05 Nov 2017

µGrid is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µGrid is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µGrid; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330,

- Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file `iterable_proxy.hh`

*#include "*common/muSpectre_common.hh*"* transitional class for iterating over materials and their strain and stress fields

Copyright © 2019 Till Junge

Author
Till Junge till.junge@epfl.ch

Date
08 Nov 2019

µSpectre is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µSpectre is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µSpectre; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file `laminate_homogenisation.cc`

*#include "*laminate_homogenisation.hh*"* : Implementation of functions of internal laminate solver used in MaterialLaminate

Author
Ali Falsafi ali.falsafi@epfl.ch

Date
28 Sep 2018

file **laminate_homogenisation.hh**

#include "common/geometry.hh"*#include* "common/muSpectre_common.hh"*#include* "libmugrid/field_map.hh"*#include* "material_linear_anisotropic.hh"*#include* "materials_toolbox.hh"*#include* "material_muSpectre_base.hh"*#include* <*tuple*> Laminatehomogenisation enables one to obtain the resulting stress and stiffness tensors of a laminate pixel that is consisted of two materialswith a certain normal vector of their interface plane. note that it is supposed to be used in static way. so it does note any data member. It is merely a collection of functions used to calculate effective stress and stiffness.

Author
Ali Falsafi ali.falsafi@epfl.ch

Date
28 Sep 2018

file **material_base.cc**

*#include "*materials/material_base.hh*"#include* <libmugrid/field.hh>*#include* <libmugrid/field_typed.hh> implementation of material

Author
Till Junge till.junge@epfl.ch

Date
01 Nov 2017

file **material_base.hh**

*#include "*common/muSpectre_common.hh*"#include "*materials/materials_toolbox.hh*"#include* <libmugrid/field_collection_local.hh>*#include* <libmugrid/field_typed.hh>*#include* <libmugrid/mapped_field.hh>*#include* <*string*>*#include* <*tuple*> Base class for materials (constitutive models)

Author
Till Junge till.junge@epfl.ch

Date
25 Oct 2017

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **material_evaluator.hh**

#include "common/muSpectre_common.hh"*#include* "materials/materials_toolbox.hh"*#include* <libmugrid/T4_map_proxy.hh>*#include* <libmugrid/ccoord_operations.hh>*#include* <libmugrid/mapped_field.hh>*#include* <*exception*>*#include* <*memory*>*#include* <*sstream*> Helper to evaluate material laws.

Copyright © 2018 Till Junge

Author
Till Junge till.junge@altermail.ch

Date
12 Dec 2018

µSpectre is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µSpectre is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µSpectre; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **material_hyper_elasto_plastic1.cc**

#include "common/muSpectre_common.hh"*#include* "materials/stress_transformations_Kirchhoff.hh"*#include* "materials/material_hyper_elasto_plastic1.hh"*#include* <libmugrid/T4_map_proxy.hh> implementation for MaterialHyperElastoPlastic1

Copyright © 2018 Till Junge

Author
Till Junge till.junge@epfl.ch

Date
21 Feb 2018

µSpectre is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µSpectre is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µSpectre; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **material_hyper_elasto_plastic1.hh**

#include "materials/material_muSpectre_base.hh"*#include* "materials/materials_toolbox.hh"*#include* <libmugrid/eigen_tools.hh>*#include* <libmugrid/mapped_field.hh>*#include* <libmugrid/mapped_state_field.hh>*#include* <*algorithm*> Material for logarithmic hyperelasto-plasticity, as defined in de Geus 2017 (https://doi.org/10.1016/j.cma.2016.12.032) and further explained in Geers 2003 (https://doi.org/10.1016/j.cma.2003.07.014)

Copyright © 2018 Till Junge

Author
Till Junge till.junge@epfl.ch

Date
20 Feb 2018

µSpectre is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µSpectre is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µSpectre; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330,

- Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **material_hyper_elasto_plastic2.cc**

#include "common/muSpectre_common.hh"*#include* "materials/stress_transformations_Kirchhoff.hh"*#include* "materials/material_hyper_elasto_plastic2.hh"*#include* <libmugrid/T4_map_proxy.hh> copy of material_hyper_elasto_plastic1 with Young, Poisson, yield criterion and hardening modulus per pixel. As defined in de Geus 2017 (https://doi.org/10.1016/j.cma.2016.12.032) and further explained in Geers 2003 (https://doi.org/10.1016/j.cma.2003.07.014).

Copyright © 2019 Till Junge

Author
Richard Leute richard.leute@imtek.uni-freiburg.de

Date
08 Jul 2019

µSpectre is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µSpectre is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µSpectre; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file `material_hyper_elasto_plastic2.hh`

*#include "*materials/material_muSpectre_base.hh*"#include "*materials/materials_toolbox.hh*"#include* <libmugrid/eigen_tools.hh>*#include* <libmugrid/mapped_state_field.hh>*#include* <*algorithm*> copy of material_hyper_elasto_plastic1 with Young, Poisson, yield criterion and hardening modulus per pixel. As defined in de Geus 2017 (https://doi.org/10.1016/j.cma.2016.12.032) and further explained in Geers 2003 (https://doi.org/10.1016/j.cma.2003.07.014).

Copyright © 2019 Till Junge

Author
Richard Leute richard.leute@imtek.uni-freiburg.de

Date
08 Jul 2019

µSpectre is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µSpectre is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µSpectre; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file `material_laminate.cc`

#include "material_laminate.hh" material that uses laminae homogenisation

Author
Ali Falsafi ali.falsafi@epfl.ch

Author
Till Junge till.junge@altermail.ch

Date
04 Jun 2018

file `material_laminate.hh`

#include "common/muSpectre_common.hh"*#include* "materials/material_muSpectre_base.hh"*#include* "materials/materials_toolbox.hh"*#include* "materials/material_evaluator.hh"*#include* "materials/laminate_homogenisation.hh"*#include* "common/intersection_octree.hh"*#include* "cell/cell.hh"*#include* "libmugrid/T4_map_proxy.hh"*#include* <*vector*> material that uses laminae homogenisation

Author
Ali Falsafi ali.falsafi@epfl.ch

Author
Till Junge till.junge@altermail.ch

Date
04 Jun 2018

You should have received a copy of the GNU Lesser General Public License along with µSpectre; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file `material_linear_anisotropic.cc`

#include "material_linear_anisotropic.hh" Implementation of general anisotropic linear constitutive model.

Copyright © 2017 Till Junge

Author
Ali Falsafiali.falsafi@epfl.ch

Date
09 Jul 2018

µSpectre is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µSpectre is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µSpectre; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file `material_linear_anisotropic.hh`

#include "materials/stress_transformations_PK2.hh"*#include* "materials/material_base.hh"*#include* "materials/material_muSpectre_base.hh"*#include* "materials/materials_toolbox.hh"*#include* "common/muSpectre_common.hh"*#include* "common/voigt_conversion.hh"*#include* "libmugrid/T4_map_proxy.hh"*#include* "libmugrid/tensor_algebra.hh"*#include* "libmugrid/eigen_tools.hh"*#include* "libmugrid/mapped_field.hh" defenition of general anisotropic linear constitutive model

Copyright © 2017 Till Junge

Author
Ali Falsafiali.falsafi@epfl.ch

Date
9 Jul 2018

µSpectre is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µSpectre is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µSpectre; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **material_linear_elastic1.cc**

*#include "*materials/material_linear_elastic1.hh*"* Implementation for materiallinearelastic1.

Copyright © 2017 Till Junge

Author
Till Junge till.junge@epfl.ch

Date
14 Nov 2017

µSpectre is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µSpectre is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µSpectre; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **material_linear_elastic1.hh**

*#include "*common/muSpectre_common.hh*"#include "*materials/stress_transformations_PK2.hh*"#include "*materials/material_muSpectre_base.hh*"#include "*materials/materials_toolbox.hh*"#include <*libmugrid/field_map_static.hh*>* Implementation for linear elastic reference material like in de Geus.

Copyright © 2017 Till Junge

Author
Till Junge till.junge@epfl.ch

Date
13 Nov 2017

```
    2017. This follows the simplest and likely not most efficient
    implementation (with exception of the Python law)
```

µSpectre is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µSpectre is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µSpectre; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330,

- Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file `material_linear_elastic2.cc`

*#include "*materials/material_linear_elastic2.hh*"* implementation for linear elastic material with eigenstrain

Copyright © 2018 Till Junge

Author
Till Junge till.junge@altermail.ch

Date
04 Feb 2018

µSpectre is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µSpectre is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µSpectre; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330,

- Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file `material_linear_elastic2.hh`

*#include "*materials/material_linear_elastic1.hh*"#include* <libmugrid/mapped_field.hh>*#include* <*Eigen/Dense*> linear elastic material with imposed eigenstrain and its type traits. Uses the MaterialMuSpectre facilities to keep it simple

Author
Till Junge till.junge@altermail.ch

Date
03 Feb 2018

file `material_linear_elastic3.cc`

*#include "*materials/material_linear_elastic3.hh*"* implementation for linear elastic material with distribution of stiffness properties. Uses the MaterialMuSpectre facilities to keep it simple.

Author
Richard Leute <richard.leute@imtek.uni-freiburg.de

Date
20 Feb 2018

file `material_linear_elastic3.hh`

#include "materials/material_linear_elastic1.hh"#include <libmugrid/mapped_field.hh>#include <Eigen/Dense> linear elastic material with distribution of stiffness properties. Uses the MaterialMuSpectre facilities to keep it simple.

Author
Richard Leute richard.leute@imtek.uni-freiburg.de

Date
20 Feb 2018

file `material_linear_elastic4.cc`

#include "material_linear_elastic4.hh" linear elastic material with distribution of stiffness properties. In difference to material_linear_elastic3 two Lame constants are stored per pixel instead of the whole elastic matrix C. Uses the MaterialMuSpectre facilities to keep it simple.

Author
Richard Leute <richard.leute@imtek.uni-freiburg.de

Date
15 March 2018

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **material_linear_elastic4.hh**

#include "materials/material_linear_elastic1.hh"#include "libmugrid/mapped_field.hh"#include <Eigen/Dense> linear elastic material with distribution of stiffness properties. In difference to material_linear_elastic3 two Lame constants are stored per pixel instead of the whole elastic matrix C. Uses the MaterialMuSpectre facilities to keep it simple.

Copyright © 2018 Till Junge

Author
Richard Leute richard.leute@imtek.uni-freiburg.de

Date
15 March 2018

µSpectre is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µSpectre is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µSpectre; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330,

- Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **material_linear_elastic_generic1.cc**

#include "materials/material_linear_elastic_generic1.hh"#include "common/voigt_conversion.hh" implementation for MaterialLinearElasticGeneric

Copyright © 2018 Till Junge

Author
Till Junge till.junge@altermail.ch

Date
21 Sep 2018

µSpectre is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µSpectre is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µSpectre; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file `material_linear_elastic_generic1.hh`

*#include "*common/muSpectre_common.hh*"#include "*materials/stress_transformations_PK2.hh*"#include "*materials/material_muSpectre_base.hh*"#include* <libmugrid/T4_map_proxy.hh>*#include* <libmugrid/field_map_static.hh>*#include* <*memory*> Implementation fo a generic linear elastic material that stores the full elastic stiffness tensor. Convenient but not the most efficient.

Copyright © 2018 Till Junge

Author
Till Junge till.junge@altermail.ch

Date
21 Sep 2018

µSpectre is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µSpectre is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µSpectre; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330,

- Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file `material_linear_elastic_generic2.cc`

*#include "*material_linear_elastic_generic2.hh*"* Implementation for generic linear elastic law with eigenstrains.

Copyright © 2018 Till Junge

Author
Till Junge till.junge@epfl.ch

Date
20 Dec 2018

µSpectre is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µSpectre is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µSpectre; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file `material_linear_elastic_generic2.hh`

*#include "*material_linear_elastic_generic1.hh*"#include "*libmugrid/mapped_field.hh*"* implementation of a generic linear elastic law with eigenstrains

Copyright © 2018 Till Junge

Author
Till Junge till.junge@epfl.ch

Date
20 Dec 2018

µSpectre is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µSpectre is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µSpectre; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file `material_linear_orthotropic.cc`

*#include "*material_base.hh*"#include "*common/muSpectre_common.hh*"#include "*material_linear_anisotropic.hh*"#include "*material_linear_orthotropic.hh*"* Implementation of general orthotropic linear constitutive model.

Copyright © 2017 Till Junge

Author
Ali Falsafiali.falsafi@epfl.ch

Date
11 Jul 2018

µSpectre is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µSpectre is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µSpectre; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **material_linear_orthotropic.hh**

#include "stress_transformations_PK2.hh"*#include* "material_base.hh"*#include* "material_muSpectre_base.hh"*#include* "material_linear_anisotropic.hh"*#include* "common/muSpectre_common.hh"*#include* "cell/cell.hh"*#include* "libmugrid/field_map_static.hh" defenition of general orthotropic linear constitutive model

Copyright © 2017 Till Junge

Author
Ali Falsafiali.falsafi@epfl.ch

Date
11 Jul 2018

µSpectre is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µSpectre is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µSpectre; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **material_muSpectre_base.hh**

#include "common/muSpectre_common.hh"*#include* "materials/material_base.hh"*#include* "materials/materials_toolbox.hh"*#include* "materials/material_evaluator.hh"*#include* "materials/iterable_proxy.hh"*#include* "cell/cell.hh"*#include* "libmugrid/field_map_static.hh"*#include* <tuple>*#include* <type_traits>*#include* <iterator>*#include* <stdexcept>*#include* <functional>*#include* <util-

ity>#include "sstream" Base class for materials written for µSpectre specifically. These can take full advantage of the configuration-change utilities of µSpectre. The user can inherit from them to define new constitutive laws and is merely required to provide the methods for computing the second Piola-Kirchhoff stress and Tangent. This class uses the "curiously recurring template parameter" to avoid virtual calls.

Author
Till Junge till.junge@epfl.ch

Date
25 Oct 2017

file **material_stochastic_plasticity.cc**

*#include "*materials/material_stochastic_plasticity.hh*"#include <sstream>*

material for stochastic plasticity as described in Z. Budrikis et al. Nature Comm. 8:15928, 2017. It only works together with "python

-script", which performes the avalanche loop. This makes the material slower but more easy to modify and test. (copied from *material_linear_elastic4.cc*)

Author
Richard Leute richard.leute@imtek.uni-freiburg.de

Date
24 Jan 2019

You should have received a copy of the GNU Lesser General Public License along with µSpectre; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **material_stochastic_plasticity.hh**

#include "common/muSpectre_common.hh"#include "materials/material_linear_elastic1.hh"#include "cell/cell.hh"#include <libmugrid/mapped_field.hh>

material for stochastic plasticity as described in Z. Budrikis et al. Nature Comm. 8:15928, 2017. It only works together with "python

-script", which performes the avalanche loop. This makes the material slower but more easy to modify and test. (copied from *material_linear_elastic4.hh*)

Copyright © 2019 Till Junge

Author
Richard Leute richard.leute@imtek.uni-freiburg.de

Date
24 Jan 2019

µSpectre is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µSpectre is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µSpectre; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **materials_toolbox.hh**

#include "common/muSpectre_common.hh"#include "materials/stress_transformations_PK1.hh"#include "common/voigt_conversion.hh"#include <libmugrid/eigen_tools.hh>*#include* <libmugrid/T4_map_proxy.hh>*#include* <libmugrid/tensor_algebra.hh>*#include <Eigen/Dense>#include <unsupported/Eigen/MatrixFunctions>#include <exception>#include <sstream>#include <iostream>#include <tuple>#include <type_traits>* collection of common continuum mechanics tools

Copyright © 2017 Till Junge

Author
Till Junge till.junge@epfl.ch

Date
02 Nov 2017

µSpectre is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µSpectre is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µSpectre; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file `s_t_material_linear_elastic_generic1.cc`

*#include "*materials/s_t_material_linear_elastic_generic1.hh*"* the implemenation of the methods of the class STMateriallinearelasticgeneric1

Copyright © 2020 Ali Falsafi

Author
Ali Falsafi ali.falsafi@epfl.ch

Date
20 Jan 2020

µSpectre is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µSpectre is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µSpectre; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file `s_t_material_linear_elastic_generic1.hh`

*#include "*materials/material_linear_elastic_generic1.hh*"#include "*materials/stress_transformations_PK1.hh*"#include "*materials/stress_transformations_Kirchhoff.hh*"* Material that is merely used to behave as an intermediate convertor for enablling us to conduct tests on stress_transformation usogn MaterialLinearelasticgeneric1.

Copyright © 2020 Ali Falsafi

Author
Ali Falsafi ali.falsafi@epfl.ch

Date
20 Jan 2020

file **stress_transformations.hh**

*#include "*common/muSpectre_common.hh*"#include* <libmugrid/eigen_tools.hh> isolation of stress conversions for quicker compilation

Author
Till Junge till.junge@altermail.ch

Date
29 Oct 2018

file **stress_transformations_default_case.hh**

*#include "*common/muSpectre_common.hh*"#include* <libmugrid/T4_map_proxy.hh> default structure for stress conversions

Author
Till Junge till.junge@altermail.ch

Date
29 Oct 2018

file **stress_transformations_Kirchhoff.hh**

#include "materials/stress_transformations_default_case.hh"*#include* "materials/stress_transformations_Kirchhoff_impl.hh"*#include* "stress_transformations.hh" Stress conversions for Kirchhoff stress ()

Author
Till Junge till.junge@altermail.ch

Date
29 Oct 2018

file **stress_transformations_Kirchhoff_impl.hh**

Implementation of stress conversions for Kirchhoff stress.

Author
Till Junge till.junge@altermail.ch

Date
29 Oct 2018

file `stress_transformations_PK1.hh`

#include "materials/stress_transformations_default_case.hh"*#include* "materials/stress_transformations_PK1_impl.hh"*#include* "materials/stress_transformations.hh" stress conversion for PK1 stress

Author
Till Junge till.junge@altermail.ch

Date
29 Oct 2018

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **stress_transformations_PK1_impl.hh**

#include "common/muSpectre_common.hh"*#include* <libmugrid/T4_map_proxy.hh> implementation of stress conversion for PK1 stress

Copyright © 2018 Till Junge

Author
Till Junge till.junge@altermail.ch

Date
29 Oct 2018

µSpectre is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µSpectre is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µSpectre; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **stress_transformations_PK2.hh**

#include "materials/stress_transformations_default_case.hh"*#include* "materials/stress_transformations_PK2_impl.hh"*#include* "materials/stress_transformations.hh" stress conversions for PK2 stress

Copyright © 2018 Till Junge

Author
Till Junge till.junge@altermail.ch

Date
29 Oct 2018

µSpectre is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µSpectre is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µSpectre; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file `stress_transformations_PK2_impl.hh`

*#include "*common/muSpectre_common.hh*"#include <*libmugrid/T4_map_proxy.hh*>* Implementation of stress conversions for PK2 stress.

Copyright © 2018 Till Junge

Author
Till Junge till.junge@altermail.ch

Date
29 Oct 2018

µSpectre is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µSpectre is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µSpectre; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file `projection_base.cc`

*#include <sstream>#include "*projection/projection_base.hh*"* implementation of base class for projections

Copyright © 2017 Till Junge

Author
Till Junge till.junge@epfl.ch

Date
06 Dec 2017

µSpectre is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µSpectre is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µSpectre; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **projection_base.hh**

#include <libmugrid/field_collection.hh>*#include* <libmugrid/field_typed.hh>*#include* <libmufft/fft_engine_base.hh>*#include* "common/muSpectre_common.hh"*#include* <*memory*> Base class for Projection operators.

Copyright © 2017 Till Junge

Author
Till Junge till.junge@altermail.ch

Date
03 Dec 2017

µSpectre is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µSpectre is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µSpectre; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **projection_default.cc**

#include "projection/projection_default.hh"*#include* <libmufft/fft_engine_base.hh> Implementation default projection implementation.

Copyright © 2018 Till Junge

Author
Till Junge till.junge@altermail.ch

Date
14 Jan 2018

µSpectre is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µSpectre is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µSpectre; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **projection_default.hh**

#include <libmugrid/field_map_static.hh>*#include* <libmufft/derivative.hh>*#include* "projection/projection_base.hh" virtual base class for default projection implementation, where the projection operator is stored as a full fourth-order tensor per k-space point (as opposed to 'smart' faster implementations, such as ProjectionFiniteStrainFast

Copyright (C) 2018 Till Junge

Author
Till Junge till.junge@altermail.ch

Date
14 Jan 2018

µSpectre is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µSpectre is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µSpectre; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **projection_finite_strain.cc**

#include "projection/projection_finite_strain.hh"*#include* <libmugrid/iterators.hh>*#include* <libmufft/fft_utils.hh>*#include* <libmufft/fftw_engine.hh>*#include* *"Eigen/Dense"* implementation of the finite strain projection operator

Copyright © 2017 Till Junge

Author
Till Junge till.junge@altermail.ch Richard Leute richard.leute@imtek.uni-freiburg.de Lars Pastewka lars.pastewka@imtek.uni-freiburg.de

Date
05 Dec 2017

file **projection_finite_strain.hh**

*#include "*projection/projection_default.hh*"* Class for discrete finite-strain gradient projections.

Author
Till Junge till.junge@altermail.ch Richard Leute richard.leute@imtek.uni-freiburg.de Lars Pastewka lars.pastewka@imtek.uni-freiburg.de

Date
16 Apr 2019

file **projection_finite_strain_fast.cc**

*#include "*projection/projection_finite_strain_fast.hh*"#include* <libmufft/fft_utils.hh>*#include* <libmugrid/iterators.hh> implementation for fast projection in finite strain

Author
Till Junge till.junge@epfl.ch Lars Pastewka lars.pastewka@imtek.uni-freiburg.de

Date
12 Dec 2017

file **projection_finite_strain_fast.hh**

#include <libmugrid/field_collection.hh>*#include* <libmugrid/field_map_static.hh>*#include* <libmufft/derivative.hh>*#include* "common/muSpectre_common.hh"*#include* "projection/projection_base.hh" Faster alternative to ProjectionFinitestrain.

Author
Till Junge till.junge@epfl.ch Lars Pastewka lars.pastewka@imtek.uni-freiburg.de

Date
12 Dec 2017

file **projection_small_strain.cc**

*#include "*projection/projection_small_strain.hh*"#include* <libmufft/fft_utils.hh> Implementation for ProjectionSmallStrain.

Author
Till Junge till.junge@altermail.ch

Date
14 Jan 2018

file **projection_small_strain.hh**

*#include "*projection/projection_default.hh*"*

Small strain projection operator as defined in Appendix A1 of DOI: 10.1002/nme.5481 ("A finite element perspective on nonlinear

FFT-based micromechanical simulations", Int. J. Numer. Meth. Engng 2017; 111 :903–926)

Author
Till Junge till.junge@altermail.ch

Date
14 Jan 2018

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **solver_base.cc**

*#include "*solver/solver_base.hh*"* implementation of SolverBase

Copyright © 2018 Till Junge

Author
Till Junge till.junge@epfl.ch

Date
24 Apr 2018

µSpectre is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µSpectre is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µSpectre; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330,

- Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **solver_base.hh**

*#include "*solver/solver_common.hh*"#include "*cell/cell.hh*"#include <Eigen/Dense>* Base class for iterative solvers for linear systems of equations.

Copyright © 2018 Till Junge

Author
Till Junge till.junge@epfl.ch

Date
24 Apr 2018

µSpectre is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µSpectre is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µSpectre; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330,

- Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **solver_cg.cc**

*#include "*solver/solver_cg.hh*"#include "*cell/cell_adaptor.hh*"#include <*libmufft/communicator.hh*>#include <iomanip>#include <sstream>#include <iostream>* implements SolverCG

Copyright © 2018 Till Junge

Author
Till Junge till.junge@epfl.ch

Date
24 Apr 2018

µSpectre is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µSpectre is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µSpectre; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330,

- Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **solver_cg.hh**

*#include "*solver/solver_base.hh*"* class fo a simple implementation of a conjugate gradient solver. This follows algorithm 5.2 in Nocedal's Numerical Optimization (p 112)

Copyright © 2018 Till Junge

Author
Till Junge till.junge@epfl.ch

Date
24 Apr 2018

µSpectre is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µSpectre is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µSpectre; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330,

- Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file `solver_common.cc`

*#include "*solver/solver_common.hh*"* implementation for solver utilities

Copyright © 2018 Till Junge

Author
Till Junge till.junge@epfl.ch

Date
15 May 2018

µSpectre is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µSpectre is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µSpectre; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330,

- Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file `solver_common.hh`

*#include "*common/muSpectre_common.hh*"#include <Eigen/Dense>#include <stdexcept>* Errors raised by solvers and other common utilities.

Copyright © 2017 Till Junge

Author
Till Junge till.junge@altermail.ch

Date
28 Dec 2017

µSpectre is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µSpectre is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µSpectre; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330,

- Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **solver_eigen.cc**

*#include "*solver/solver_eigen.hh*"#include <iomanip>#include <sstream>* Implementations for bindings to Eigen's iterative solvers.

Copyright © 2018 Till Junge

Author
Till Junge till.junge@epfl.ch

Date
15 May 2018

µSpectre is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µSpectre is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µSpectre; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330,

- Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **solver_eigen.hh**

*#include "*solver/solver_base.hh*"#include "*cell/cell.hh*"#include "*cell/cell_adaptor.hh*"#include <Eigen/IterativeLinearSolvers>#include <iostream>#include <unsupported/Eigen/IterativeSolvers>* Bindings to Eigen's iterative solvers.

Copyright © 2018 Till Junge

Author
Till Junge till.junge@epfl.ch

Date
15 May 2018

µSpectre is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µSpectre is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µSpectre; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330,

- Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **solvers.cc**

*#include "*solver/solvers.hh*"#include* <libmugrid/iterators.hh>*#include* <libmugrid/mapped_field.hh>*#include <Eigen/Dense>#include <iomanip>#include <iostream>* implementation of dynamic newton-cg solver

Copyright © 2018 Till Junge

Author
Till Junge till.junge@epfl.ch

Date
24 Apr 2018

µSpectre is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

µSpectre is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µSpectre; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330,

- Boston, MA 02111-1307, USA.

Additional permission under GNU GPL version 3 section 7

If you modify this Program, or any covered work, by linking or combining it with proprietary FFT implementations or numerical libraries, containing parts covered by the terms of those libraries' licenses, the licensors of this Program grant you additional permission to convey the resulting work.

file **solvers.hh**

*#include "*solver/solver_base.hh*"#include <Eigen/Dense>#include <vector>#include <string>* Free functions for solving rve problems.

Copyright © 2018 Till Junge

Author
Till Junge till.junge@epfl.ch

Date
24 Apr 2018

group **Coordinates**

Typedefs

using **Ccoord_t** = std::array<Dim_t, Dim>

Ccoord_t are cell coordinates, i.e. integer coordinates.

using **Rcoord_t** = std::array<Real, Dim>

Real space coordinates.

using **DynCcoord_t** = DynCcoord<threeD>

usually, we should not need omre than three dimensions

using **DynRcoord_t** = DynCcoord<threeD, Real>

usually, we should not need omre than three dimensions

dir **/home/docs/checkouts/readthedocs.org/user_builds/muspectre/checkouts/master/src/cell**

dir **/home/docs/checkouts/readthedocs.org/user_builds/muspectre/checkouts/master/src/common**

dir
/home/docs/checkouts/readthedocs.org/user_builds/muspectre/checkouts/master/src/libmufft

dir
/home/docs/checkouts/readthedocs.org/user_builds/muspectre/checkouts/master/src/libmugrid

dir
/home/docs/checkouts/readthedocs.org/user_builds/muspectre/checkouts/master/src/materials

dir /home/docs/checkouts/readthedocs.org/user_builds/muspectre/checkouts/master/src/projection

dir /home/docs/checkouts/readthedocs.org/user_builds/muspectre/checkouts/master/src/solver

dir /home/docs/checkouts/readthedocs.org/user_builds/muspectre/checkouts/master/src

CHAPTER

EIGHT

LICENSE

μSpectre is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

μSpectre is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

You are **not** allowed to use μSpectre in commercial products.

8.1 GNU LESSER GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

This version of the GNU Lesser General Public License incorporates the terms and conditions of version 3 of the GNU General Public License, supplemented by the additional permissions listed below.

8.1.1 0. Additional Definitions.

As used herein, "this License" refers to version 3 of the GNU Lesser General Public License, and the "GNU GPL" refers to version 3 of the GNU General Public License.

"The Library" refers to a covered work governed by this License, other than an Application or a Combined Work as defined below.

An "Application" is any work that makes use of an interface provided by the Library, but which is not otherwise based on the Library. Defining a subclass of a class defined by the Library is deemed a mode of using an interface provided by the Library.

A "Combined Work" is a work produced by combining or linking an Application with the Library. The particular version of the Library with which the Combined Work was made is also called the "Linked Version".

The "Minimal Corresponding Source" for a Combined Work means the Corresponding Source for the Combined Work, excluding any source code for portions of the Combined Work that, considered in isolation, are based on the Application, and not on the Linked Version.

The "Corresponding Application Code" for a Combined Work means the object code and/or source code for the Application, including any data and utility programs needed for reproducing the Combined Work from the Application, but excluding the System Libraries of the Combined Work.

8.1.2 1. Exception to Section 3 of the GNU GPL.

You may convey a covered work under sections 3 and 4 of this License without being bound by section 3 of the GNU GPL.

8.1.3 2. Conveying Modified Versions.

If you modify a copy of the Library, and, in your modifications, a facility refers to a function or data to be supplied by an Application that uses the facility (other than as an argument passed when the facility is invoked), then you may convey a copy of the modified version:

- a) under this License, provided that you make a good faith effort to ensure that, in the event an Application does not supply the function or data, the facility still operates, and performs whatever part of its purpose remains meaningful, or
- b) under the GNU GPL, with none of the additional permissions of this License applicable to that copy.

8.1.4 3. Object Code Incorporating Material from Library Header Files.

The object code form of an Application may incorporate material from a header file that is part of the Library. You may convey such object code under terms of your choice, provided that, if the incorporated material is not limited to numerical parameters, data structure layouts and accessors, or small macros, inline functions and templates (ten or fewer lines in length), you do both of the following:

- a) Give prominent notice with each copy of the object code that the Library is used in it and that the Library and its use are covered by this License.
- b) Accompany the object code with a copy of the GNU GPL and this license document.

8.1.5 4. Combined Works.

You may convey a Combined Work under terms of your choice that, taken together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and reverse engineering for debugging such modifications, if you also do each of the following:

- a) Give prominent notice with each copy of the Combined Work that the Library is used in it and that the Library and its use are covered by this License.
- b) Accompany the Combined Work with a copy of the GNU GPL and this license document.
- c) For a Combined Work that displays copyright notices during execution, include the copyright notice for the Library among these notices, as well as a reference directing the user to the copies of the GNU GPL and this license document.
- d) Do one of the following: - 0) Convey the Minimal Corresponding Source under the terms of

  this License, and the Corresponding Application Code in a form suitable for, and under terms that permit, the user to recombine or relink the Application with a modified version of the Linked Version to produce a modified Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.

  - 1) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (a) uses at run time a copy of the Library already present on the user's computer system, and (b) will operate properly with a modified version of the Library that is interface-compatible with the Linked Version.

- e) Provide Installation Information, but only if you would otherwise be required to provide such information under section 6 of the GNU GPL, and only to the extent that such information is necessary to install and execute a modified version of the Combined Work produced by recombining or relinking the Application with a modified version of the Linked Version. (If you use option 4d0, the Installation Information must accompany the Minimal Corresponding Source and Corresponding Application Code. If you use option 4d1, you must provide the Installation Information in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.)

8.1.6 5. Combined Libraries.

You may place library facilities that are a work based on the Library side by side in a single library together with other library facilities that are not Applications and are not covered by this License, and convey such a combined library under terms of your choice, if you do both of the following:

- a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms of this License.
- b) Give prominent notice with the combined library that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

8.1.7 6. Revised Versions of the GNU Lesser General Public License.

The Free Software Foundation may publish revised and/or new versions of the GNU Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library as you received it specifies that a certain numbered version of the GNU Lesser General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that published version or of any later version published by the Free Software Foundation. If the Library as you received it does not specify a version number of the GNU Lesser General Public License, you may choose any version of the GNU Lesser General Public License ever published by the Free Software Foundation.

If the Library as you received it specifies that a proxy can decide whether future versions of the GNU Lesser General Public License shall apply, that proxy's public statement of acceptance of any version is permanent authorization for you to choose that version for the Library.

8.2 GNU GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow. TERMS AND CONDITIONS

8.2.1 0. Definitions.

"This License" refers to version 3 of the GNU General Public License.

"Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

"The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based on the Program.

To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License,

and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

8.2.2 1. Source Code.

The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

8.2.3 2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

8.2.4 3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

8.2.5 4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

8.2.6 5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

a) The work must carry prominent notices stating that you modified it, and giving a relevant date.

b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".

c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

8.2.7 6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

8.2.8 7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or

b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or

c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

d) Limiting the use for publicity purposes of names of licensors or authors of the material; or

e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8.2.9 8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

8.2.10 9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

8.2.11 10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

8.2.12 11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly

relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

8.2.13 12. No Surrender of Others' Freedom.

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

8.2.14 13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

8.2.15 14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

8.2.16 15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

8.2.17 16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

8.2.18 17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

μSpectre is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3, or (at your option) any later version.

μSpectre is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with µSpectre; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

A

C

D

E

M

R

S