Documentation of the Janus Library

Release 0.0

S. Brisard

November 14, 2016
Janus is a Python library dedicated to the discretization of the Lippmann–Schwinger equation with periodic boundary conditions. The matrix of the resulting linear system is the sum of a block-diagonal matrix and a block-circulant matrix. Following the ideas initially introduced by Moulinec & Suquet (1998) matrix-vector products can then be computed efficiently by means of the Fast Fourier Transform. A matrix-free strategy is then adopted to solve the linear system iteratively, e.g. (non-exhaustive list)

- fixed-point iterations (Moulinec & Suquet, 1998),
- accelerated schemes (Eyre & Milton, 1999),
- augmented Lagrangians (Michel et al., 2001),
- Krylov subspace linear solvers (Brisard & Dormieux, 2010),
- polarization-based schemes (Monchiet & Bonnet, 2012),

see also Moulinec & Silva (2014) for a comparison of some of these iterative schemes.

The library provides tools to define the linear operator associated to the discretized Lippmann–Schwinger equation, and to compute the necessary matrix-vector products. Third-party iterative linear solvers (Scipy, petsc4py) can then be invoked to compute the solution.

The library is designed with performance in mind. It is fully parallelized (using MPI and mpi4py), and the critical parts of the code are written in Cython.
For the moment, no precompiled binaries are available, and Janus must be compiled from sources, using setuptools.

The sources can be retrieved from Github, https://github.com/sbrisard/janus.git.

### 2.1 Prerequisites

Janus requires Python 3k. The serial version depends on FFTW (version 3) only, while the parallel (MPI-based) version also requires mpi4py.

**Todo**

The present version of setup.py tries to install the parallel version of the code if it detects that mpi4py is installed. In other words, if mpi4py is installed, the MPI-enabled version of FFTW must be installed.

### 2.2 Configuration

Edit the section [build_ext] of the file setup.cfg. You must provide the following values:

```ini
[build_ext]
include_dirs = ...
library_dirs = ...
libraries = ...
```

**include_dirs**

The path to the FFTW headers.

**library_dirs**

The path to the FFTW shared library.

**libraries**

The name of the FFTW libraries (comma separated), including the MPI-enabled version if necessary.
2.3 Compilation and installation under Unix-like systems (Linux, OSX)

Set the following values:

```java
[build_ext]
include_dirs = /path/to/headers
library_dirs = /path/to/binaries
libraries = fftw3, fftw3_mpi
```

Then, issue the standard commands in a console:

```
python setup.py install --user
```

2.4 Compilation and installation under Windows

The parallel version of this code is not tested under Windows. You must first download and install the precompiled binaries of FFTW for Windows.

2.4.1 Compilation with the Windows SDK 7.1

Set the following values:

```java
[build_ext]
include_dirs = C:\PATH\TO\HEADERS
library_dirs = C:\PATH\TO\BINARIES
libraries = libfftw3-3
```

Then open the Windows SDK 7.1 Command Prompt, and issue the following command:

```
set DISTUTILS_USE_SDK=1
setenv /x64 /release
```

Change to the root directory of the Janus project, and issue the standard commands:

```
python setup.py install --user
```

2.4.2 Compilation with MinGW/MSYS

Set the following values:

```java
[build_ext]
include_dirs = C:\PATH\TO\HEADERS
library_dirs = C:\PATH\TO\BINARIES
libraries = fftw3-3
```

Todo

Complete installation procedure with MinGW.
2.5 Test your installation

Testing the installation of Janus requires `pytest`. To run all serial tests, issue the following command at the root of the project:

```
python -m pytest tests
```

To run all parallel tests (assuming you compiled the MPI-enabled version of Janus), issue the following command at the root of the project:

```
mpiexec -np 3 pytest tests/parallel
```

where the total number of processes can be adjusted (an odd number should preferably be used, as it is more likely to reveal bugs).

---

**Todo**

How to print only messages from root process with pytest?
Classes and functions for the definition of operators are provided in the `janus.operators` module. Operator should be understood in the most general sense, as a mapping from a (real) vector space to another. If the mapping is linear, then the resulting operator is a tensor.

The root of the hierarchy tree of operators is the class `AbstractOperator`. Instances of this class have two attributes, isize and osize which are the sizes of the input and output of the operator, respectively (in other words, these attributes are the dimensions of the domain and codomain, respectively).

Operator op is mapped to the vector x through the method `AbstractOperator.apply`:

\[ y = \text{op.apply}(x) \]

where x (resp. y) is a 1D array of length op.isize (resp. op.osize).

### 3.1 Structured operators

Structured operators are operators whose input and output are structured in multi-dimensional grids. The content of each cell might be tensorial, so that the input and output of 2D structured operators are 3-dimensional arrays. Likewise, the input and output arrays of 3D structured operators are 4-dimensional arrays. The local input and output then refer to data contained in one specific cell. For example, let \( x[:, :, :] \) (resp. \( y[:, :, :] \)) be the input (resp. output) of a 2D structured operator; the local input (resp. output) of cell \((i0, i1)\) is the 1D array \( x[i0, i1, :] \) (resp \( y[i0, i1, :] \)).

Two- and three-dimensional structured operators are defined in this module through the classes `AbstractStructuredOperator2D` and `AbstractStructuredOperator3D`. Instances of these classes have two attributes, ishape and oshape which are the shapes (tuples of dimensions) of the input and output of the operator, respectively. It should be noted that the data layout (the dimensions of the spatial grid) of the input and output are identical. In other words:

```python
op.ishape[:-1] == op.oshape[:-1]
```

for any structured operator op. Of course, op.ishape[-1] and op.oshape[-1] may differ. Structured operators are applied to multidimensional arrays as follows:

```python
y = op.apply(x)
```

where `x.shape == op.ishape` and `y.shape == op.oshape`.
3.2 Block-diagonal operators

Block-diagonal operators (BlockDiagonalOperator2D, BlockDiagonalOperator3D) are defined as structured operators for which the local output depends on the local input only. Any block diagonal operator can be represented as an array of local operators (of type AbstractOperator) loc. Then, the input x is mapped to the output y as follows

\[ y[i0, i1] = \text{loc}[i0, i1].\text{apply}(x[i0, i1]) \]

in 2D, and

\[ y[i0, i1, i2] = \text{loc}[i0, i1, i2].\text{apply}(x[i0, i1, i2]) \]

in 3D.

3.2.1 Block-diagonal linear operators

This can be further simplified in the case of linear, block-diagonal operators (BlockDiagonalLinearOperator2D, BlockDiagonalLinearOperator3D). Indeed, loc is then an array of matrices, which can be viewed as a higher-dimension array. Therefore, a block-diagonal linear operator can be defined through a float64 array a such that

\[ y[i0, i1, i2] = \text{sum}(a[i0, i1, i2, j2] \times x[i0, i1, j2], j2) \]

in 2D, and

\[ y[i0, i1, i2, i3] = \text{sum}(a[i0, i1, i2, i3, j3] \times x[i0, i1, i2, j3], j3) \]

in 3D.

Block-diagonal linear operators are created with the function block_diagonal_linear_operator, which takes as an input an array, whose last two dimensions correspond to the matrix of the local operator.

```
>>> import numpy as np
>>> a = np.arange(120., dtype=np.float64).reshape(2, 3, 4, 5)
>>> op = operators.block_diagonal_linear_operator(a)
>>> x = np.arange(30., dtype=np.float64).reshape(2, 3, 5)
>>> y = op.apply(x)
>>> yy = np.sum(a * x[:,:,np.newaxis,:], axis=-1)
>>> np.sqrt(np.sum((yy - y)**2))
0.0
```

3.3 Performing in-place operations

All types of operators define a method `apply(x, y)`, where `x` is a memoryview of the input and `y` is a memoryview of the output. If `y` is `None`, then `apply` returns a newly created memoryview. If `y` is not `None`, then `apply` returns a reference to `y`.

Depending on the implementation, some operators allow for in-place operations, which can further reduce memory allocations. In other words, `apply(x, x)` is valid for such operators and returns the expected value. Whether or not an operator allows for in-place operations is implementation dependent, and should be specified in the documentation. **Unless otherwise stated, it should be assumed that in-place operations are not supported.**

If relevant, the above also applies to the Cython method `c_apply(x, y)`. 
3.4 API of module `janus.operators`
Classes and functions for the definition of materials are provided in the `janus.material` package. This package is structured in three levels:

1. **the physical model** (e.g. elasticity, thermal conductivity, ...),
2. **linear/nonlinear constitutive law**, and
3. **material symmetries** (isotropic, transverse isotropic, ...).

Regardless of the actual constitutive law, an attempt is made to expose a unified interface. For example, a new instance of a specific material can be created through the function `create()` of the corresponding module.

### 4.1 Elastic materials

#### 4.1.1 Linear, elastic materials

**Isotropic, linear, elastic materials**

Such materials are created from their shear modulus ($\mu$) and Poisson ratio ($\nu$) as follows:

```python
>>> import janus.material.elastic.linear.isotropic as material
>>> mat = material.create(1.0, 0.3, 3)
>>> mat
IsotropicLinearElasticMaterial(g=1.0, nu=0.3, dim=3)
>>> mat.g # Shear modulus
1.0
>>> mat.nu # Poisson ratio
0.3
>>> mat.k # Bulk modulus
2.1666666666666665
```

The function `janus.material.elastic.linear.isotropic.create()` takes two positional arguments: the shear modulus and the Poisson ratio, and one optional argument: the dimension of the physical space, which can be 2 (plane strain elasticity) or 3 (3D elasticity); the default value is 3. To create a *plane stress* ($\mu, \nu$) material, a *plane strain* ($\mu, \nu'$) material should be created, with $\nu' = \nu / (1 + \nu)$.

A helper function, `janus.material.elastic.linear.isotropic.poisson_from_bulk_and_shear_moduli()` is also provided. It returns the Poisson ratio, computed from the bulk and shear moduli.

Green operators for strains associated with a given material are instanciated with the `green_operator()` method, like so:

```python
```
>>> green = mat.green_operator()
>>> green
Green Operator(IsotropicLinearElasticMaterial(g=1.0, nu=0.3, dim=3))

The returned operator can then be manipulated frequency-wise

```python
>>> import numpy as np
>>> k = np.array([1.0, 2.0, 3.0])
>>> green.set_frequency(k)
```
Computing discrete Fourier transforms

Discrete Fourier transforms are computed through the Fast Fourier Transform method (FFT) implemented in the FFTW library. Module janus.fft provides a Python wrapper to this C library. This module exposes both serial and parallel (MPI) implementations through a unified interface.

Before the main methods and functions of the janus.fft module are introduced, an important design issue should be mentioned. In the present implementation of the module, input data (to be transformed) is not passed directly to FFTW. Rather, a local copy is first made, and FFTW then operates on this local copy. This allows reusing the same plan to perform many transforms (which is advantageous in the context of iterative solvers). This certainly induces a performance hit, which is deemed negligible for transforms of large 2D or 3D arrays.

Although not essential, it might be useful to have a look to the FFTW manual. For the time being, only two and three dimensional real-to-complex transforms are implemented.

5.1 Serial computations

The following piece of code creates an object transform which can perform real FFTs on 32x64 grids of real numbers.

```python
>>> import janus.fft.serial
>>> transform = janus.fft.serial.create_real((32, 64))
```

The function janus.fft.serial.create_real() can be passed planner flags (see Planner Flags in the FFTW manual). The attributes of the returned object are

- transform.global_ishape contains the global shape of the input array,
- transform.ishape contains the local shape of the input (real) array,
- transform.global_oshape contains the global shape of the output (complex) array,
- transform.oshape contains the local shape of the output (complex) array. For serial transforms, local and global output shapes coincide.

For serial transforms, local and global shapes coincide.

```python
>>> transform.global_ishape
(32, 64)
>>> transform.ishape
(32, 64)
>>> transform.global_oshape
(32, 66)
>>> transform.oshape
(32, 66)
```
It should be noted that complex-valued tables are stored according to the FFTW library: even (resp. odd) values of the fast index correspond to the real (resp. imaginary) part of the complex number (see also Multi-Dimensional DFTs of Real Data in the FFTW manual).

Direct (real-to-complex) transforms are computed through the method `transform.r2c()`, which takes as input a `MemoryView` of shape `transform.ishape`, and returns a `MemoryView` of shape `transform.oshape`.

```python
>>> import numpy as np
>>> np.random.seed(20150223)
>>> x = np.random.rand(*transform.ishape)
>>> y1 = transform.r2c(x)
```

It should be noted that `y1` is a `MemoryView`, not a `numpy` array; it can, however, readily be converted into an array.

```python
>>> print(y1)
<MemoryView of 'array' object>
>>> y1 = np.asarray(y1)
>>> type(y1)
<class 'numpy.ndarray'>
```

The output can be converted to an array of complex numbers

```python
>>> actual = y1[..., 0::2] + 1j * y1[..., 1::2]
>>> actual.shape
(32, 33)
```

and compared to the FFT of `x` computed by means of the `numpy.fft` module.

```python
>>> expected = np.fft.rfftn(x)
>>> expected.shape
(32, 33)
>>> abs_delta = np.abs(expected - actual)
>>> abs_exp = np.abs(expected)
>>> error = np.sqrt(np.sum(abs_delta**2) / np.sum(abs_exp**2))
>>> assert error < 1E-15
```

Inverse discrete Fourier transform is computed through the method `transform.c2r()`

```python
>>> x1 = transform.c2r(y1)
>>> error = np.sqrt(np.sum((x1 - x)**2) / np.sum(x**2))
>>> assert error < 1E-15
```

It should be noted that the output array can be passed as an argument to both `transform.r2c()` and `transform.c2r()`

```python
>>> y2 = np.empty(transform.oshape)
>>> out = transform.r2c(x, y2)
>>> assert out.base is y2
>>> assert np.sum((y2 - y1)**2) == 0.0
```

and `transform.c2r()`

```python
>>> x2 = np.empty(transform.ishape)
>>> out = transform.c2r(y1, x2)
>>> assert out.base is x2
>>> assert np.sum((x2 - x1)**2) == 0.0
```
5.2 Parallel computations

The module `janus.fft.parallel` is a wrapper around the `fftw3-mpi` library (refer to Distributed-memory FFTW with MPI in the FFTW manual for the inner workings of this library). This module must be used along with the `mpi4py` module to handle MPI communications.

The Python API is very similar to the API for serial transforms. However, computing a parallel FFT is slightly more involved than computing a serial FFT, because the data must be distributed across the processes. The computation must go through the following steps:

1. create input data (root process),
2. create a transform object (all processes),
3. gather local shapes (root process),
4. scatter the input data according to the previously gathered local sizes (root process),
5. compute the transform (all processes),
6. gather the results (root process).

This is illustrated in the step-by-step tutorial below. This tutorial aims again at computing a $32 \times 64$ real Fourier transform. The full source can be downloaded here, it must be run through the following command line:

```
$ mpiexec -np 2 python3 parallel_fft_tutorial.py
```

where the number of processes can be adjusted (all output produced below was obtained with two parallel processes).

Before we proceed with the description of the program, it should be noted that communication will be carried out with the uppercase versions `MPI.Comm.Gather` and `MPI.Comm.Scatter`. The lowercase versions of `MPI.Comm.scatter` and `MPI.Comm.gather` are slightly easier to use, but communicate objects through pickling. This approach fails with very large objects (the size limit is much lower than the intrinsic MPI size limit). With `MPI.Comm.Gather` and `MPI.Comm.Scatter`, the intrinsic MPI size limit is restored. The FFT objects defined in the module `janus.fft.parallel` provide attributes to help call these methods.

A few modules must first be imported

```python
import numpy as np
import janus.fft.parallel
from mpi4py import MPI
```

Then, some useful variables are created

```python
if __name__ == '__main__':
    comm = MPI.COMM_WORLD
    root = 0
    shape = (32, 64)
```

Then, the transform objects (one for each process) are created (step 2), and their various shapes are printed out.

```python
transform = janus.fft.parallel.create_real(shape, comm)
if comm.rank == root:
    print('global_ishape = {}'.format(transform.global_ishape))
    print('global_oshape = {}'.format(transform.global_oshape))
    print('ishape = {}'.format(transform.ishape))
    print('oshape = {}'.format(transform.oshape))
```

This code snippet outputs the following messages
The `transform.shape` attribute refers to the `global` (logical) shape of the transform. Since the data is distributed across all processes, the `local` size in memory of the input and output data differ from `transform.shape`. Accordingly, the `transform.rshape` (resp. `transform.cshape`) attribute refers to the local shape of the real, input (resp. complex, output) data, for the current process. As expected with FFTW, it is observed that the data is distributed with respect to the first dimension. Indeed, the global, first dimension is 64, and the above example is run with 2 processes; therefore, the local first dimension is $64 \div 2 = 32$.

In order to figure out how to scatter the input data, the root process then gathers all local sizes and displacements, and the parameters to be passed to `mpi4py.MPI.Scatterv()` and `mpi4py.MPI.Gatherv()` are prepared:

```python
counts_and_displs = comm.gather(sendobj=(transform isize, transform idispl,
transform osize, transform odispl),
root=root)

if comm.rank == root:
    np.random.seed(20150310)
x = np.random.rand(*shape)
icounts, idispls, ocounts, odispls = zip(*counts_and_displs)
else:
x, icounts, idispls, ocounts, odispls = None, None, None, None, None
```

Then the input data `x` is scattered across all processes:

```python
x_loc = np.empty(transform ishape, dtype=np.float64)
comm.Scatterv([x, icounts, idispls, MPI.DOUBLE], x_loc, root)
```

Each process then executes its transform:

```python
y_loc = transform.r2c(x_loc)
```

and the root process finally gathers the results:

```python
if comm.rank == root:
y = np.empty(transform global oshape, dtype=np.float64)
else:
y = None
comm.Gatherv(y_loc, [y, ocounts, odispls, MPI.DOUBLE], root)
```

To check that the computation is correct, the same transform is finally computed locally by the root process:

```python
if comm.rank == root:
    serial_transform = janus.fft.serial.create_real(shape)
y_ref = np.asarray(serial_transform.r2c(x))
err = np.sum((y-y_ref)**2) / np.sum(y_ref**2)
assert err <= np.finfo(np.float64).eps
```

## 5.2.1 The complete program

```python
# Imports
import numpy as np
import janus.fft.parallel
from mpi4py import MPI
```

Chapter 5. Computing discrete Fourier transforms
# Init some variables
if __name__ == '__main__':
    comm = MPI.COMM_WORLD
    root = 0
    shape = (32, 64)

# Create transform objects
transform = janus.fft.parallel.create_real(shape, comm)
if comm.rank == root:
    print('global_ishape = {}'.format(transform.global_ishape))
    print('global_oshape = {}'.format(transform.global_oshape))
    print('ishape = {}'.format(transform.ishape))
    print('oshape = {}'.format(transform.oshape))

# Prepare communications
counts_and_displs = comm.gather(sendobj=(transform.isize, transform.idispl, transform.osize, transform.odispl),
                                    root=root)
if comm.rank == root:
    np.random.seed(20150310)
x = np.random.rand(*shape)
icounts, idispls, ocounts, odispls = zip(*counts_and_displs)
else:
x, icounts, idispls, ocounts, odispls = None, None, None, None, None

# Scatter input data
x_loc = np.empty(transform.ishape, dtype=np.float64)
comm.Scatterv([x, icounts, idispls, MPI.DOUBLE], x_loc, root)

# Execute transform
y_loc = transform.r2c(x_loc)

# Gather output data
if comm.rank == root:
y = np.empty(transform.global_oshape, dtype=np.float64)
else:
y = None

comm.Gatherv(y_loc, [y, ocounts, odispls, MPI.DOUBLE], root)

# Validate result
if comm.rank == root:
    serial_transform = janus.fft.serial.create_real(shape)
y_ref = np.asarray(serial_transform.r2c(x))
err = np.sum((y-y_ref)**2) / np.sum(y_ref**2)
assert err <= np.finfo(np.float64).eps
6.1 Description of the problem

In this tutorial, we will compute the effective elastic properties of a simple 2D microstructure in plane strain. More precisely, we consider a periodic microstructure made of a square inclusion of size $a$, embedded in a unit-cell of size $L$ (see Fig. 6.1).

The effective properties of this periodic microstructure are derived from the solution to the so-called corrector problem

\begin{align}
\nabla \cdot \sigma &= 0, \\
\sigma &= C : \varepsilon, \\
\varepsilon &= E + \nabla^* u,
\end{align}

where $u$ denotes the unknown, periodic displacement, $\varepsilon$ (resp. $\sigma$) is the local strain (resp. stress) and $C$ is the local stiffness (inclusion or matrix). From the solution to the above problem, the effective stiffness $C^{\text{eff}}$ is defined as the tensor mapping the macroscopic (imposed) strain $E = \langle \varepsilon \rangle$ to the macroscopic stress $\sigma = \langle \sigma \rangle$ (where quantities between angle brackets denote volume averages)

\[
C^{\text{eff}} : E = \frac{1}{L^d} \int_{(0,L)^d} \sigma(x_1, x_2) \, dx_1 \cdots dx_d.
\]
Note: This example is illustrated in two dimensions \((d = 2)\). However, it is implemented so as to be dimension independent, so that \(d = 3\) should work out of the box.

In the present tutorial, we shall concentrate on the \(1212\) component of the effective stiffness, that is to say that the following macroscopic strain will be imposed

\[
E = E_{12} \left( e_1 \otimes e_2 + e_2 \otimes e_1 \right),
\]

and the volume average \(\langle \sigma_{12} \rangle\) will be evaluated. To do so, the boundary value problem \(\ ???\) is transformed into an integral equation, known as the Lippmann–Schwinger equation \((\text{Korringa, 1973}; \text{Zeller & Dederichs, 1973}; \text{Kerner, 1974})\). This equation reads

\[
\varepsilon + \Gamma_0[(C - C_0) : \varepsilon] = E,
\]

where \(C_0\) denotes the stiffness of the reference material, \(\Gamma_0\) the related Green operator for strains, and \(\varepsilon\) the local strain tensor. We will assume that the reference material is isotropic, with shear modulus \(\mu_0\) and Poisson ratio \(\nu_0\).

Following \textit{Moulinec and Suquet (1998)}, the above Lippmann–Schwinger equation \((6.5)\) is solved by means of fixed point iterations

\[
\varepsilon^{k+1} = E - \Gamma_0[(C - C_0) : \varepsilon^k].
\]

Finally, the above iterative scheme is discretized over a regular grid, leading to the basic uniform grid, periodic Lippmann–Schwinger solver.

### 6.2 Implementation of the Lippmann–Schwinger operator

We will call the operator

\[
\varepsilon \mapsto E - \Gamma_0[(C - C_0) : \varepsilon]
\]

the \textit{Lippmann–Schwinger operator}. In the present section, we show how this operator is implemented as a class with Janus. This will be done by composing two successive operators, namely (i) the local operator

\[
\varepsilon \mapsto \tau = (C - C_0) : \varepsilon,
\]

where \(\tau\) denotes the stress-polarization, and (ii) the Green operator for strains

\[
\tau \mapsto \Gamma_0[\tau].
\]

For the implementation of the local operator defined by Eq. \((6.7)\), it is first observed that \(C_0\), \(C_i\) and \(C_m\) being isotropic materials, \(C - C_0\) is an isotropic tensor at any point of the unit-cell. In other words, both \(C_i - C_0\) and \(C_m - C_0\) will be defined as instances of \texttt{FourthRankIsotropicTensor}.

Furthermore, this operator is \textit{local}. In other words, the output value in cell \((i_0, i_1)\) depends on the input value in the same cell only (the neighboring cells are ignored). More precisely, we assume that a uniform grid of shape \((n, n)\) is used to discretized Eq. \((6.6)\). Then the material properties are constant in each cell, and we define \(\text{delta}_C[i0, i1, :, :]\) the matrix representation of \(C - C_0\) (see \textit{Mandel notation}). Likewise, \(\text{eps}[i0, i1, :, :]\) is the vector representation of the strain tensor in cell \((i0, i1)\). Then, the stress-polarization \((C - C_0) : \varepsilon\) in cell \((i0, i1)\) is given by the expression:

\[
\text{tau}[i0, i1] = \text{delta}_C[i0, i1] \otimes \text{eps}[i0, i1],
\]

where \(\otimes\) denotes the matrix multiplication operator. It results from the above relation that the local operator defined by \((6.7)\) should be implemented as a \texttt{BlockDiagonalOperator2D}. As for the non-local operator, it is instantiated by a simple call to the \textit{green operator} method of the relevant material (see \textit{Materials}).

The script starts with imports from the standard library, the SciPy stack and Janus itself:
import itertools
import numpy as np
import matplotlib.pyplot as plt
import janus.green as green
import janus.fft.serial as fft
import janus.material.elastic.linear.isotropic as material
import janus.operators as operators
from janus.operators import isotropic_4

We then define a class `Example`, which represents the microstructure described above. The first few lines of its initializer are pretty simple

```python
class Example:
    def __init__(self, mat_i, mat_m, mat_0, n, a=0.5, dim=3):
        self.mat_i = mat_i
        self.mat_m = mat_m
        self.n = n
        shape = tuple(itertools.repeat(n, dim))
```

`mat_i` (resp. `mat_m`, `mat_0`) are the material properties of the inclusion (resp. the matrix, the reference material); `n` is the number of grid cells along each side, `a` is the size of the inclusion, and `dim` is the dimension of the physical space. The `shape` of the grid is stored in a tuple, the length of which depends on `dim`.

**Note:** As much as possible, keep your code dimension-independent. This means that the spatial dimension (2 or 3) should not be hard-coded. Rather, you should make it a rule to always parameterize the spatial dimension (use a variable `dim`), even if you do not really intend to change this dimension. Janus object sometimes have different implementations depending on the spatial dimension. For example, the abstract class `FourthRankIsotropicTensor` has two concrete daughter classes `FourthRankIsotropicTensor2D` and `FourthRankIsotropicTensor3D`. However, both can be instantiated through the unique function `isotropic_4`, where the spatial dimension can be specified.

We then define the local operators $C_i - C_0$ and $C_m - C_0$ as `FourthRankIsotropicTensor`. It is recalled that the stiffness $C$ of a material with bulk modulus $\kappa$ and shear modulus $\mu$ reads

$$C = d\kappa J + 2\mu K,$$

where $d$ denotes the dimension of the physical space and $J$ (resp. $K$) denote the spherical (resp. deviatoric) projector tensor. In other words, the spherical and deviatoric projections of $C$ are $d\kappa$ and $2\mu$, respectively. As a consequence, the spherical and deviatoric projections of $C - C_0$ are $d(\kappa - \kappa_0)$ and $2(\mu - \mu_0)$, respectively. This leads to the following definitions

```python
# ...
delta_C_i = isotropic_4(dim*(mat_i.k-mat_0.k),
                      2*(mat_i.g-mat_0.g), dim)
delta_C_m = isotropic_4(dim*(mat_m.k-mat_0.k),
                      2*(mat_m.g-mat_0.g), dim)
# ...
```

Now, `delta_C_i` and `delta_C_m` are used to create the operator $\varepsilon \mapsto (C - C_0) : \varepsilon$ as a `BlockDiagonalOperator2D`. Block-diagonal operators are initialized from an array of local operators, called `ops` below.

**Todo**

6.2. Implementation of the Lippmann--Schwinger operator
This code snippet is not dimension independent.

```python
# ...
ops = np.empty(shape, dtype=object)
ops[:, :] = delta_C_m
imax = int(np.ceil(n*a-0.5))
ops[:imax, :imax] = delta_C_i
self.eps_to_tau = operators.BlockDiagonalOperator2D(ops)
# ...
```

The upper-left quarter of the unit-cell is filled with $\Delta C_i (C_i - C_0)$, while the remainder of the unit-cell receives $\Delta C_m (C_m - C_0)$. Finally, a `BlockDiagonalOperator2D` is created from the array of local operators. It is called `eps_to_tau` as it maps the strain ($\epsilon$) to the stress-polarization ($\tau$).

**Note:** `eps_to_tau` is not a method. Rather, it is an attribute, which turns out to be a function.

Finally, the discrete Green operator for strains associated with the reference material $C_0$ is created. This requires first to create a FFT object (see *Computing discrete Fourier transforms*).

**Todo**

Document Green operators for strains.

```python
# ...
self.green = green.truncated(mat_0.green_operator(),
                               shape, 1.,
                               fft.create_real(shape))
```

The Lippmann–Schwinger operator $\epsilon \mapsto \Gamma_0[(C - C_0) : \epsilon]$ is then defined by composition

```python
def apply(self, x, out=None):
    if out is None:
        out = np.zeros_like(x)
    self.eps_to_tau.apply(x, out)
    self.green.apply(out, out)
```

which closes the definition of the class **Example**.

**Note:** Note how we allowed for the output array to be passed by reference, thus allowing for memory reuse.

### 6.3 The main block of the script

It starts with the definition of a few parameters

```python
if __name__ == '__main__':
    dim = 2  # Spatial dimension
    sym = (dim*(dim+1))/2  # Dim. of space of second rank, symmetric tensors
    n = 256  # Number of cells along each side of the grid
    mu_i, nu_i = 100, 0.2  # Shear modulus and Poisson ratio of inclusion
    mu_m, nu_m = 1, 0.3  # Shear modulus and Poisson ratio of matrix
    mu_0, nu_0 = 50, 0.3  # Shear modulus and Poisson ratio of ref. mat.
    num_cells = n**dim  # Total number of cells
```
Then, an instance of class `Example` is created

```python
example = Example(mat_i=material.create(mu_i, nu_i, dim),
                   mat_m=material.create(mu_m, nu_m, dim),
                   mat_0=material.create(mu_0, nu_0, dim),
                   n=n,
                   dim=dim)
```

We then define `eps_macro`, which stores the imposed value of the macroscopic strain $\mathbf{E}$, and `eps` and `eps_new`, which hold two successive iterates of the local strain field $\mathbf{\varepsilon}$. 

```python
avg_eps = np.zeros((sym,), dtype=np.float64)
avg_eps[-1] = 1.0
eps = np.empty(example.green.ishape, dtype=np.float64)
new_eps = np.empty_like(eps)
```

**Note:** The shape of the arrays `eps` and `eps_new` is simply inferred from the shape of the input of the Green operator for strains $\Gamma_0$.

We will not implement a stopping criterion for this simple example. Rather, a fixed number of iterations will be specified. Meanwhile, the residual

\[
\left( \frac{1}{L^d} \int_{(0,L)^d} \left( \mathbf{\varepsilon}^{k+1} - \mathbf{\varepsilon}^k \right) : (\mathbf{\varepsilon}^{k+1} - \mathbf{\varepsilon}^k) \, dx_1 \cdots dx_d \right)^{1/2},
\]

will be computed and stored at each iteration through the following estimate

\[
\text{norm(new_eps-eps)/sqrt(num_cells)/norm(avg_eps)},
\]

where normalization (using $\|\mathbf{E}\|$) is also applied.

**Note:** Note that the quantity defined by Eq. (6.9) is truly a residual. Indeed, it is the norm of the difference between the left- and right-hand side in Eq. (6.5), since $\mathbf{\varepsilon}^{k+1} - \mathbf{\varepsilon}^k = \mathbf{E} - \Gamma_0[(\mathbf{C} - \mathbf{C}_0) : \mathbf{\varepsilon}^k] - \mathbf{\varepsilon}^k$.

The fixed-point iterations defined by Eq. (6.6) are then implemented as follows

```python
num_iter = 400
res = np.empty((num_iter,), dtype=np.float64)
eps[...] = avg_eps
normalization = 1/np.sqrt(num_cells)/np.linalg.norm(avg_eps)
for i in range(num_iter):
    example.apply(eps, out=new_eps)
    np.subtract(avg_eps, new_eps, out=new_eps)
    res[i] = normalization*np.linalg.norm(new_eps-eps)
    eps, new_eps = new_eps, eps
```

and the results are post-processed

```python
tau = example.eps_to_tau.apply(eps)
avg_tau = np.mean(tau, axis=tuple(range(dim)))
C_1212 = mu_0+0.5*avg_tau[-1]/avg_eps[-1]
print(C_1212)
```

6.3. The main block of the script 23
To compute the macroscopic stiffness, we recall the definition of the stress-polarization from which we find

\[ C_{\text{eff}}^E : E = \langle \sigma \rangle = \langle C : \varepsilon + \tau \rangle = C : E + \langle \tau \rangle. \]

Then, from the specific macroscopic strain \( E \) that we considered [see Eq. (6.4)]

\[ C_{1212}^{\text{eff}} = C_{0,1212} + \frac{\langle \tau_{12} \rangle}{2E_{12}} = C_{0,1212} + \frac{[\langle \tau \rangle]_{-1}}{2[E]_{-1}} = \mu_0 + \frac{[\langle \tau \rangle]_{-1}}{2[E]_{-1}} \]

where brackets refer to the Mandel notation, and the -1 index denotes the last component of the column-vector (which, in Mandel’s notation, refers to the 12 component of second-rank symmetric tensors, both in two and three dimensions). We get the following approximation

\[ C_{1212} \approx 1.41903971282, \]

and the map of the local strains is shown in Fig. 6.2, while Fig. 6.3 shows that the residual decreases (albeit slowly) with the number of iterations. This completes this tutorial.

6.4 The complete program

The complete program can be downloaded here.
Fig. 6.3: The normalized residual as a function of the number of iterations.
# Begin: imports
import itertools
import numpy as np
import matplotlib.pyplot as plt
import janus.green as green
import janus.fft.serial as fft
import janus.material.elastic.linear.isotropic as material
import janus.operators as operators

from janus.operators import isotropic_4
# End: imports

# Begin: init
class Example:
    def __init__(self, mat_i, mat_m, mat_0, n, a=0.5, dim=3):
        self.mat_i = mat_i
        self.mat_m = mat_m
        self.n = n
        shape = tuple(itertools.repeat(n, dim))
        # ...
        # End: init
    # Begin: create (C_i - C_0) and (C_m - C_0)
    delta_C_i = isotropic_4(dim*(mat_i.k-mat_0.k),
                            2*(mat_i.g-mat_0.g), dim)
    delta_C_m = isotropic_4(dim*(mat_m.k-mat_0.k),
                            2*(mat_m.g-mat_0.g), dim)
    # ...
    # End: create (C_i - C_0) and (C_m - C_0)
    # Begin: create local operator \( \epsilon (C-C_0) \):
    ops = np.empty(shape, dtype=object)
    ops[:, :] = delta_C_m
    imax = int(np.ceil(n*a-0.5))
    ops[:imax, :imax] = delta_C_i
    self.eps_to_tau = operators.BlockDiagonalOperator2D(ops)
    # ...
    # End: create local operator \( \epsilon (C-C_0) \):
    # Begin: create non-local operator \( \Gamma_0[\epsilon] \):
    self.green = green.truncated(mat_0.green_operator(),
                                  shape, 1.,
                                  fft.create_real(shape))
    #End: create non-local operator \( \Gamma_0[\epsilon] \)

# Begin: apply
    def apply(self, x, out=None):
        if out is None:
            out = np.zeros_like(x)
        self.eps_to_tau.apply(x, out)
        self.green.apply(out, out)
    # End: apply

# Begin: params
```python
if __name__ == '__main__':
    dim = 2  # Spatial dimension
    sym = (dim*(dim+1))//2  # Dim. of space of second rank, symmetric tensors
    n = 256  # Number of cells along each side of the grid
    mu_i, nu_i = 100, 0.2  # Shear modulus and Poisson ratio of inclusion
    mu_m, nu_m = 1, 0.3  # Shear modulus and Poisson ratio of matrix
    mu_0, nu_0 = 50, 0.3  # Shear modulus and Poisson ratio of ref. mat.
    num_cells = n**dim  # Total number of cells
    # End: params
    # Begin: instantiate example
    example = Example(mat_i=material.create(mu_i, nu_i, dim),
                      mat_m=material.create(mu_m, nu_m, dim),
                      mat_0=material.create(mu_0, nu_0, dim),
                      n=n,
                      dim=dim)
    # End: instantiate example
    # Begin: define strains
    avg_eps = np.zeros((sym,), dtype=np.float64)  # Begin: define strains
    avg_eps[-1] = 1.0
    eps = np.empty(example.green.ishape, dtype=np.float64)
    new_eps = np.empty_like(eps)
    # End: define strains
    # Begin: iterate
    num_iter = 400
    res = np.empty((num_iter,), dtype=np.float64)
    eps[..., ] = avg_eps
    normalization = 1/np.sqrt(num_cells)/np.linalg.norm(avg_eps)
    for i in range(num_iter):
        example.apply(eps, out=new_eps)
        np.subtract(avg_eps, new_eps, out=new_eps)
        res[i] = normalization*np.linalg.norm(new_eps-eps)
        eps, new_eps = new_eps, eps
    # End: iterate
    # Begin: post-process
    tau = example.eps_to_tau.apply(eps)
    avg_taus = np.mean(tau, axis=tuple(range(dim)))
    C_1212 = mu_0+0.5*avg_taus[-1]/avg_eps[-1]
    print(C_1212)
    fig, ax = plt.subplots()
    ax.set_xlabel('Number of iterations')
    ax.set_ylabel('Normalized residual')
    ax.loglog(res)
    fig.tight_layout(pad=0.2)
    fig.savefig('residual.png', transparent=True)
    fig, ax_array = plt.subplots(nrows=1, ncols=3)
    width, height = fig.get_size_inches()
    fig.set_size_inches(width, width/3)
    for i, ax in enumerate(ax_array):
        ax.set_axis_off()
        ax.imshow(eps[... ,i], interpolation='nearest')
    fig.tight_layout(pad=0)
    fig.savefig('eps.png', transparent=True)
    # End: post-process
```

6.4. The complete program
CHAPTER 7

API Reference

7.1 janus.fft module

7.2 janus.fft.serial module

7.3 janus.fft.parallel module
TODO List

Todo
The present version of \texttt{setup.py} tries to install the parallel version of the code if it detects that \texttt{mpi4py} is installed. In other words, if \texttt{mpi4py} is installed, the MPI-enabled version of \texttt{FFTW} must be installed.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/janus/checkouts/latest/docs/installation.rst, line 16.)

Todo
Complete installation procedure with MinGW.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/janus/checkouts/latest/docs/installation.rst, line 88.)

Todo
How to print only messages from root process with pytest?

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/janus/checkouts/latest/docs/installation.rst, line 103.)

Todo
This code snippet is not dimension independent.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/janus/checkouts/latest/docs/tutorials/square_basic/square_basic.rst, line 126.)

Todo
Document Green operators for strains.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/janus/checkouts/latest/docs/tutorials/square_basic/square_basic.rst, line 140.)
Copyright (c) 2013-2015, S. Brisard

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
Janus makes heavy use of Mandel’s representation of symmetric, second rank tensors as column vectors, and fourth rank tensors with minor symmetries as matrices. This representation is defined below. Furthermore, the properties of the matrix representation are summarized (see also Wikipedia).

10.1 Mandel notation in 3D

In this section, the notation is introduced for tensors of the three dimensional space.

10.1.1 Second rank, symmetric tensors

Let $\varepsilon$ be a second rank, symmetric tensor

$$\varepsilon_{ij} = \varepsilon_{ji}.$$  

Its Mandel representation $[\varepsilon]$ (as a column-vector) is defined as follows

$$[\varepsilon] = [\varepsilon_{11}, \varepsilon_{22}, \varepsilon_{33}, \sqrt{2} \varepsilon_{23}, \sqrt{2} \varepsilon_{31}, \sqrt{2} \varepsilon_{12}]^T,$$

where the cross-component $\varepsilon_{ij} (i \neq j)$ appears at the $3+k$-th line, with $k \neq i \neq j$. The $\sqrt{2}$ prefactors ensure that the standard scalar product of column vectors coincides with the double contraction of tensors. Indeed

$$\sigma : \varepsilon = \sigma_{ij} \varepsilon_{ij},$$

$$= \sigma_{11} \varepsilon_{11} + \sigma_{22} \varepsilon_{22} + \sigma_{33} \varepsilon_{33} + 2 \sigma_{23} \varepsilon_{23} + 2 \sigma_{31} \varepsilon_{31} + 2 \sigma_{12} \varepsilon_{12},$$

$$= [\sigma]^T \cdot [\varepsilon].$$

10.1.2 Fourth rank tensors with minor symmetries

Let $C$ be a fourth rank tensor with minor symmetries

$$C_{ijkl} = C_{jikl} = C_{ijlk}.$$  

Its Mandel representation $[C]$ (as a square matrix) is defined as follows

$$[C] = \begin{bmatrix} 
C_{1111} & C_{1122} & C_{1133} & \sqrt{2} C_{1123} & \sqrt{2} C_{1131} & \sqrt{2} C_{1112} \\
C_{2211} & C_{2222} & C_{2233} & \sqrt{2} C_{2223} & \sqrt{2} C_{2231} & \sqrt{2} C_{2212} \\
C_{3311} & C_{3322} & C_{3333} & \sqrt{2} C_{3323} & \sqrt{2} C_{3331} & \sqrt{2} C_{3312} \\
\sqrt{2} C_{2311} & \sqrt{2} C_{2322} & \sqrt{2} C_{2333} & 2 C_{2323} & 2 C_{2331} & 2 C_{2312} \\
\sqrt{2} C_{3111} & \sqrt{2} C_{3122} & \sqrt{2} C_{3133} & 2 C_{3123} & 2 C_{3131} & 2 C_{3112} \\
\sqrt{2} C_{1211} & \sqrt{2} C_{1222} & \sqrt{2} C_{1233} & 2 C_{1223} & 2 C_{1231} & 2 C_{1212} 
\end{bmatrix}.$$
where the numbering of the cross-components $C_{ijkl}$ with $i \neq j$ or $k \neq l$ is consistent with the numbering of cross-components of second rank tensors. Again, the $\sqrt{2}$ and $2$ prefactors ensure that matrix-matrix and matrix-vector products coincide with the double contraction of tensors.

More precisely, the Mandel representation of the second rank tensor $\mathbf{\sigma} = C : \mathbf{\varepsilon}$ is the column vector

$$[\mathbf{\sigma}] = [C : \mathbf{\varepsilon}] = C_{ijkl} \varepsilon_{kl} = [C] : [\mathbf{\varepsilon}].$$

Likewise, if $S$ is another fourth rank tensor with minor symmetries, then

$$[C : S] = [C] \cdot [S],$$

where it is recalled that the $(i, j, k, l)$ component of $C : S$ is $C_{ijmn} S_{mnkl}$. It results from the above formula that the Mandel representation of the inverse of a fourth rank tensor is the inverse of the Mandel representation of this tensor

$$[C^{-1}] = [C]^{-1}.$$

Finally, it is readily verified that the Mandel representation of the transpose is the transpose of the Mandel representation

$$[C^T] = [C]^T.$$

### 10.2 Mandel notation in 2D

The above formulas are readily extended to two dimensions, so that we only recall the matrix representation of second rank, symmetric tensors and fourth rank tensors with minor symmetries. The properties of these matrix representations are unchanged.

#### 10.2.1 Second rank, symmetric tensors

The Mandel representation $[\mathbf{\varepsilon}]$ (as a column-vector) of any second rank, symmetric tensor $\mathbf{\varepsilon}$ is defined as follows

$$[\mathbf{\varepsilon}] = [\varepsilon_{11}, \varepsilon_{22}, \sqrt{2} \varepsilon_{12}]^T.$$

#### 10.2.2 Fourth rank tensors with minor symmetries

The Mandel representation $[C]$ (as a square matrix) of any fourth rank tensor $C$ with minor symmetries is defined as follows

$$[C] = \begin{bmatrix} C_{1111} & C_{1122} & \sqrt{2}C_{1112} \\ C_{2211} & C_{2222} & \sqrt{2}C_{2212} \\ \sqrt{2}C_{1211} & \sqrt{2}C_{1222} & 2C_{1212} \end{bmatrix}.$$
Bibliography


Indices and tables

- genindex
- modindex
- search
j

janus.fft.parallel, 29
janus.fft.serial, 29
janus.operators, 9
L
libraries (built-in variable), 3
library_dirs (built-in variable), 3