PeriPy

Jul 24, 2021

Contents

CHAPTER 1

PeriPy

[![Build Status](https://travis-ci.com/alan-turing-institute/PeriPy.svg?branch=master){]}(https://travis-ci.com/alan-turing-institute/PeriPy) [![codecov](https://codecov.io/gh/alan-turing-institute/PeriPy/branch/master/graph/badge.svg){]}(https://codecov.io/gh/alan-turing-institute/PeriPy)

PeriPy, a collaboration between Exeter, Cambridge &amp; the Alan Turing Institute, is a lightweight, open-source and high-performance python package for solving bond-based peridynamics (BBPD) problems in solid mechanics. It is implemented in [Python](https://www.python.org/) and the performance critical parts are implemented in [Cython](https://cython.org/) and [PyOpenCL](https://documen.tician.de/pyopencl/).

PeriPy allows users to write their code in pure Python. Simulations are then executed seamlessly using high performance OpenCL code.

1.1 Features

- Easy to use. Get started with the latest documentation at peripy.readthedocs.org
- 2-10x faster than exisiting OpenCL solvers
- 'Outer-loop' applications including uncertainty quantification, optimisation and feature recognition are made possible
- Support for both regular and irregular mesh files. See [meshio](https://github.com/nschloe/meshio) for the full list of mesh formats
- Support for composite and interface material models
- Support for arbitrary n-linear 'microelastic' damage models
- Simulates force or displacement controlled boundary conditions and initial conditions
- Arbitrary subsets of particles are easily measured for their displacements, damages etc.
- Output files can be viewed in [Paraview](https://www.paraview.org/)
- Various 'partial volume correction' algorithms, 'surface correction' algorithms and 'micromodulus functions' are included. The code is easily extended to define your own

- Velocity-Verlet, Euler and Euler-Cromer integrators are included and the code is easily extended to define your own higher order and/or adaptive integrators

1.2 Get started (preferred)

Building and Installation

- The package requires Python 3.7+
- Install pyopencl, a build dependency, by following these instructions https://documen.tician.de/pyopencl/misc.html
- To install pyopencl, note that pyopencl may need to be installed from *(base)* environment
- If using pyopencl on Windows with a CPU (rather than a GPU), first, ensure the C++ build tools for Visual Studio are installed (e.g., see https://youtu.be/KUTVnxCeC50)
- Make sure the OpenCL device driver is installed for your device: ` `python import pyopencl pyopencl.get_platforms()` ` The error *pyopencl._cl.LogicError: clGetPlatformIDs failed: PLATFORM_NOT_FOUND_KHR* means that the OpenCL device driver is not correctly installed.
- Install cython, a build dependency, *pip install cython*
- Install PeriPy *pip install peripy*

Running examples

- Run the first example by typing *peripy run example1* on the command line
- You can show the example code by typing *peripy run example1 –cat*
- Type *peripy run –list* for a list of examples
- For usage, type *peripy run –help*

Running the tests

The tests for this project use [pytest](https://pytest.org/en/latest/). To run the tests yourself,

- Install pytest using pip *pip install pytest*
- Type *peripy test* on the command line
- For coverage install *pytest-cov* and type *peripy coverage* on the command line

1.3 Get started from the GitHub repository (for developers)

Building and Installation

- The package requires Python 3.7+
- Install pyopencl, a build dependency, by following these instructions https://documen.tician.de/pyopencl/misc.html
- To install pyopencl, note that pyopencl may need to be installed from *(base)* environment
- If using pyopencl on Windows with a CPU (rather than a GPU), first, ensure the C++ build tools for Visual Studio are installed (e.g., see https://youtu.be/KUTVnxCeC50)
- Make sure the OpenCL device driver is installed for your device ` `python import pyopencl pyopencl.get_platforms()` ` The error *pyopencl._cl.LogicError: clGetPlatformIDs failed: PLATFORM_NOT_FOUND_KHR* means that the OpenCL device driver is not correctly installed.

- Install cython, a build dependency, *pip install cython*
- Clone the repository *git clone git@github.com:alan-turing-institute/peripy.git*
- Install using pip *pip install -e .* from the root directory of the repository

Running examples

- You can find examples of how to use the package under:*peripy/examples/*. Run the first example by typing *python peripy/examples/example1/example.py*

Running the tests

The tests for this project use [pytest](https://pytest.org/en/latest/). To run the tests yourself,

- If you haven't already, install using pip *pip install -e .* from the root directory of the repository
- Install pytest using pip *pip install pytest*
- Run *pytest* from the root directory of the repository
- For coverage install *pytest-cov* and run *pytest –cov=./peripy*

CHAPTER 2

Quickstart

2.1 Examples

You can find an examples of how to use the package under `peripy/examples/`.

There are two examples that will output mesh files that can be viewed in Paraview.

Example 1

Run the first example by typing `python peripy/examples/example1/example.py`

Example 1 is a simple, 2D peridynamics simulation example. This example is a 1.0m x 1.0m 2D plate with a central pre-crack subjected to uniform velocity displacements on the left-hand side and right-hand side of 2.5x10^-6 metres per time-step. The `--opencl` argument toggles between OpenCL and cython implementations. The `--profile` argument generates profiling information for the example.

Example 2

Run the second example by typing `python peripy/examples/example2/example.py`

Example 2 is a simple, 3D peridynamics simulation example. This example is a 1.65m x 0.25m x 0.6m plain concrete canteliver beam with no pre-crack subjected to force controlled loading on the right-hand side of the beam which linearly increases up to 45kN. In this example, the first time the volume, family and connectivity of the model are calculated, they are also stored in file '1650beam13539_model.h5'. In subsequent simulations, the arrays are loaded from this h5 file instead of being calculated again, therefore reducing the overhead of initiating the model. The `--profile` argument generates profiling information for the example

2.2 The Model class

The *`peripy.model.Model`* class allows users to define a bond-based peridynamics model for composite materials with non-linear micromodulus functions, stiffness correction factors and boundary conditions. The model is defined by parameters and a set of initial conditions (coordinates, connectivity and optionally bond_types and stiffness_corrections). For this an *`peripy.integrators.Integrator`* is required, and optionally functions implementing the boundarys.

2.3 The Integrator class

The *peripy.integrators.Integrator* is the explicit time integration method, see *peripy.integrators* for options. Any integrator with the suffix 'CL' uses OpenCL kernels to calculate the bond force and displacement update, resulting in orders of magnitude faster simulation time when compared to using the cython implementation, *peripy.integrators.Euler*. OpenCL is 'heterogeneous' which means the 'CL' integrator classes will work on a CPU device as well as a GPU device. The preferable (faster) CL device will be chosen automatically.

```
>>> from peridynamics import Model
>>> from  peripy.integrators import EulerCL
>>>
>>> def is_displacement_boundary(x):
>>>     # Node does not live on a boundary
>>>     bnd = [None, None, None]
>>>     # Node does live on a boundary
>>>     if x[0] < 1.5 * 0.1:
>>>         # These displacement boundary conditions
>>>         # are applied in the negative x direction
>>>         bnd[0] = -1
>>>     elif x[0] > 1.0 - 1.5 * 0.1:
>>>         # These displacement boundary conditions
>>>         # are applied in the positive x direction
>>>         bnd[0] = 1
>>>     return bnd
>>>
>>> # for the cython implementation, use euler = Euler(dt)
>>> euler = EulerCL(dt=1e-3)
>>>
>>> model = Model(
>>>     mesh_file,
>>>     integrator=euler,
>>>     horizon=0.1,
>>>     critical_stretch=0.005,
>>>     bond_stiffness=18.00 * 0.05 / (np.pi * 0.1**4),
>>>     is_displacement_boundary=is_displacement_boundary,
>>>     )
```

2.4 Defining a crack

To define a crack in the inital configuration, you may supply a list of pairs of nodes between which the crack is.

```
>>> initial_crack = [(1,2), (5,7), (3,9)]
>>> model = Model(
>>>     mesh_file,
>>>     integrator=euler,
>>>     horizon=0.1,
>>>     critical_stretch=0.005,
>>>     bond_stiffness=18.00 * 0.05 / (np.pi * 0.1**4),
>>>     is_displacement_boundary=is_displacement_boundary,
>>>     initial_crack=initial_crack
>>>     )
```

If it is more convenient to define the crack as a function you may also pass a function to the constructor which takes the array of coordinates as its only argument and returns a list of tuples as described above. The `peripy.model.`

`initial_crack_helper()` decorator has been provided to easily create a function of the correct form from one which tests a single pair of node coordinates and returns *True* or *False*.

```
>>> from peridynamics import initial_crack_helper
>>>
>>> @initial_crack_helper
>>> def initial_crack(x, y):
>>>     ...
>>>     if crack:
>>>         return True
>>>     else:
>>>         return False
>>>
>>> model = Model(
>>>     mesh_file,
>>>     integrator=euler,
>>>     horizon=0.1,
>>>     critical_stretch=0.005,
>>>     bond_stiffness=18.00 * 0.05 / (np.pi * 0.1**4),
>>>     is_displacement_boundary=is_displacement_boundary,
>>>     initial_crack=initial_crack
>>>     )
```

2.5 Conducting a simulation

The *`peripy.model.Model.simulate()`* method can be used to conduct a peridynamics simulation. Here it is possible to define the boundary condition magnitude throughout the simulation.

```
>>> model = Model(...)
>>>
>>> # Number of time-steps
>>> steps = 1000
>>>
>>> # Boundary condition magnitude throughout the simulation
>>> displacement_bc_array = np.linspace(2.5e-6, 2.5e-3, steps)
>>>
>>> (u,
>>>  ud,
>>>  udd,
>>>  force,
>>>  body_force,
>>>  damage,
>>>  nlist,
>>>  n_neigh) = model.simulate(
>>>     steps=steps,
>>>     displacement_bc_magnitudes=displacement_bc_array,
>>>     write=100
>>>     )
```

2.6 Conducting a simulation with initial conditions

It is possible to define initial conditions such as the displacement vector *u*, the velocity vector *ud* and the *connectivity* which is a *tuple*, (*nlist*, *n_neigh*). In this example the first 1000 steps have been simulated, generating the initial conditions for the next 1000 steps. The first step has been set to 1000 in the second simulation.

```
>>> model = Model(...)
>>>
>>> # Number of time-steps
>>> steps = 1000
>>>
>>> # Boundary condition magnitude throughout the simulation
>>> displacement_bc_array = np.linspace(2.5e-6, 2.5e-3, steps)
>>>
>>>  (u,
>>>  ud,
>>>  udd,
>>>  force,
>>>  body_force,
>>>  damage,
>>>  nlist,
>>>  n_neigh) = model.simulate(
>>>      ...displacement_bc_magnitudes=displacement_bc_array,
>>>      ...)
>>>
>>> # Boundary condition magnitude throughout the simulation
>>> displacement_bc_array = np.linspace(2.5025e-3, 5.0e-3, steps)
>>>
>>> u, *_ = model.simulate(
>>>     u=u,
>>>     ud=ud,
>>>     connectivity=(nlist, n_neigh),
>>>     steps=steps,
>>>     first_step=1000,
>>>     displacement_bc_magnitudes=displacement_bc_array,
>>>     write=100
>>>     )
```

CHAPTER 3

PeriPy Reference

Release

Date Jul 24, 2021

This reference manual details functions, modules, and objects included in PeriPy, describing what they are and what they do. For learning how to use PeriPy, see the *complete documentation*.

3.1 Model documentation

3.1.1 Model Class

class `peripy.model.`**`Model`**(*mesh_file*, *integrator*, *horizon*, *critical_stretch*, *bond_stiffness*, *transfinite=0*, *volume_total=None*, *write_path=None*, *connectivity=None*, *family=None*, *volume=None*, *initial_crack=None*, *dimensions=2*, *is_density=None*, *is_bond_type=None*, *is_displacement_boundary=None*, *is_force_boundary=None*, *is_tip=None*, *density=None*, *bond_types=None*, *stiffness_corrections=None*, *surface_correction=None*, *volume_correction=None*, *micromodulus_function=None*, *node_radius=None*)

A peridynamics model.

This class allows users to define a bond-based peridynamics model for composite materials with non-linear micromodulus functions and stiffness correction factors. The model is defined by parameters and a set of initial conditions (coordinates, connectivity and optionally bond_types and stiffness_corrections). For this an `peripy.integrators.Integrator` is required, and optionally functions implementing the boundarys. The `Model.simulate()` method can be used to conduct a peridynamics simulation.

For learning how to use PeriPy, see *complete documentation*, and for getting started, please see *quickstart*.

Create a `Model` object.

Note that nnodes is the number of nodes in the mesh. nbond_types is the number of different bonds, i.e. the number of damage models (e.g. there might be a damage model for each material and interface in a composite). nregimes is the number of linear splines that define the damage model (e.g. An n-linear damage model has nregimes = n. The bond-based prototype microelastic brittle (PMB) model has nregimes = 1). Note that nregimes and nbond_types are defined by the size of the critical_stretch and bond_stiffness positional arguments.

Parameters

- **`mesh_file`** (*`str`*) – Path of the mesh file defining the systems nodes and connectivity.
- **`integrator`** (*`peripy.integrators.Integrator`*) – The integrator to use, see *`peripy.integrators`* for options.
- **`horizon`** (*`float`*) – The horizon radius. Nodes within *horizon* of another interact with that node and are said to be within its neighbourhood.
- **`critical_stretch`** (`numpy.ndarray` or float) – An (nregimes, nbond_types) array of critical stretch values, each corresponding to a bond type and a regime, or a float value of the critical stretch of the Peridynamic bond-based prototype microelastic brittle (PMB) model.
- **`bond_stiffness`** (`numpy.ndarray` or float) – An (nregimes, nbond_types) array of bond stiffness values, each corresponding to a bond type and a regime, or a float value of the bond stiffness the Peridynamic bond-based prototype microelastic brittle (PMB) model.
- **`transfinite`** (*`bool`*) – Set to 1 for Cartesian cubic (tensor grid) mesh. Set to 0 for a tetrahedral mesh (default). If set to 1, the volumes of the nodes are approximated as the average volume of nodes on a cuboidal tensor-grid mesh.
- **`volume_total`** (*`float`*) – Total volume of the mesh. Must be provided if transfinite mode (transfinite=1) is used.
- **`write_path`** (*`path-like or str`*) – The path where the model arrays, (volume, family, connectivity, stiffness_corrections, bond_types) should be written to file to avoid doing time expensive calculations each time that the model is initiated.
- **`connectivity`** (tuple(`numpy.ndarray`, `numpy.ndarray`)) – The initial connectivity for the model. A tuple of a neighbour list and the number of neighbours for each node. If None the connectivity at the time of construction of the *`Model`* object will be used. Default None.
- **`family`** (`numpy.ndarray`) – The family array. An array of the intial number of nodes within the horizon of each node. If None the family at the time of construction of the *`Model`* object will be used. Default None.
- **`volume`** (`numpy.ndarray`) – Array of volumes for each node. If None the volume at the time of construction of the *`Model`* object will be used. Default None.
- **`initial_crack`** (*`list(tuple(int, int)) or function`*) – The initial crack of the system. The argument may be a list of tuples where each tuple is a pair of integers representing nodes between which to create a crack. Alternatively, the arugment may be a function which takes the (nnodes, 3) `numpy.ndarray` of coordinates as an argument, and returns a list of tuples defining the initial crack. Default is None
- **`dimensions`** (*`int`*) – The dimensionality of the model. The default is 2.
- **`is_density`** (*`function`*) – A function that returns a float of the material density, given a node coordinate as input.
- **`is_bond_type`** (*`function`*) – A function that returns an integer value (a flag) of the bond_type, given two node coordinates as input.

- **is_displacement_boundary** (*function*) – A function to determine if a node is on the boundary for a displacement boundary condition, and if it is, which direction and magnitude the boundary conditions are applied (positive or negative cartesian direction). It has the form is_displacement_boundary(numpy.ndarray). The argument is the initial coordinates of a node being simulated. *is_displacement_boundary* returns a (3) list of the boundary types in each cartesian direction. A boundary type with an int value of None if the node is not on a displacement controlled boundary, a value of 1 if is is on a boundary and displaced in the positive cartesian direction, a value of -1 if it is on the boundary and displaced in the negative direction, and a value of 0 if it is clamped.
- **is_force_boundary** (*function*) – As 'is_displacement_boundary' but applying to force boundary conditions as opposed to displacement boundary conditions.
- **is_tip** (*function*) – A function to determine if a node is to be measured for its state variables or reaction force over time, and if it is, which cartesian direction the measurements are made. It has the form is_tip(numpy.ndarray). The argument is the initial coordinates of a node being simulated. *is_tip* returns a (3) list of the tip types in each cartesian direction: A value of None if the node is not on the *tip*, and a value of not None (e.g. a string or an int) if it is on the *tip* and to be measured.
- **density** (numpy.ndarray) – An (nnodes,) array of node density values, each corresponding to a material
- **bond_types** (numpy.ndarray) – The bond_types for the model. If None the bond_types at the time of construction of the *Model* object will be used. Default None.
- **stiffness_corrections** (numpy.ndarray) – The stiffness_corrections for the model. If None the stiffness_corrections at the time of construction of the *Model* object will be used. If not None then these stiffness corrections will be used, overiding the positional arguments surface_correction, volume_correction and micromodulus_function. Default None.
- **surface_correction** (*int*) – A flag variable only denoting the algorithm for correcting the peridynamic surface softening effect. Le and Bobaru [Q. V. Le, F. Bobaru, Surface corrections for peridynamic models in elasticity and fracture, Computational Mechanics 61 (2018) 499–518. URL: https://link.springer.com/article/10.1007/s00466-017-1469-1.] provide a detailed comparison of the most common surface correction techniques. The 'volume method' proposed by Bobaru et al. [Chapter 2 in Bobaru F, Foster JT, Geubelle PH, Silling SA, Handbook of peridynamic modeling, (2017) (p51 – 52)] is used when surface_correction= 1 or 0. Set to 1: Surface corrections are calculated more accurately using actual nodal volumes. Set to 0: Surface corrections are calculated using an average nodal volume. Set to None: All no surface correction is applied. Default None.
- **volume_correction** (*int*) – A flag variable denoting the algorithm for approximation of the partial nodal volumes applied when integrating the bond force over the horizon. Set to 0: The 'Partial Volume algorithm' as proposed by Hu, Ha, and Bobaru [W. Hu, Y.D. Ha and F. Bobaru, Numerical integration in peridynamics, Tech. Rep., University of Nebraska-Lincoln, Department of Mechanical & Materials Engineering (September 2010)] is used. Set to None: The 'Full Volume algorithm' is used; partial nodal volumes are approximated by their full nodal volumes. Defauly None.
- **micromodulus_function** (*int*) – A flag variable denoting the normalised micromodulus function. Set to 0: A conical micromodulus function is used, which is normalised such that the maximum value of the micromodulus function is the bond stiffness. Set to None: A constant noramlised micromodulus function is used such that the maximum value of the micromodulus function is the bond stiffness. Default None.
- **node_radius** (*float*) – Average peridynamic node radius . Must be provided if volume

corrections (volume_correction=1) are applied.

Raises

- ***DimensionalityError*** – when an invalid *dimensions* argument is provided.
- ***FamilyError*** – when a node has no neighbours (other nodes it interacts with) in the initial state.

Returns A new *Model* object.

Return type *Model*

simulate(*steps*, *u=None*, *ud=None*, *connectivity=None*, *regimes=None*, *critical_stretch=None*, *bond_stiffness=None*, *displacement_bc_magnitudes=None*, *force_bc_magnitudes=None*, *first_step=1*, *write=None*, *write_path=None*)

Simulate the peridynamics model.

Parameters

- **steps** (*int*) – The number of simulation steps to conduct.
- **u** (numpy.ndarray) – The initial displacements for the simulation. If None the displacements will be initialised to zero. Default None.
- **ud** (numpy.ndarray) – The initial velocities for the simulation. If None the velocities will be initialised to zero. Default None.
- **connectivity** (tuple(numpy.ndarray, numpy.ndarray)) – The initial connectivity for the simulation. A tuple of a neighbour list and the number of neighbours for each node. If None the connectivity at the time of construction of the *Model* object will be used. Default None.
- **regimes** (numpy.ndarray) – The initial regimes for the simulation. A (*nodes*, *max_neighbours*) array of type numpy.ndarray of the regimes of the bonds.
- **critical_stretch** (numpy.ndarray or float) – An (nregimes, nbond_types) array of critical stretch values, each corresponding to a bond type and a regime, or a float value of the critical stretch of the Peridynamic bond-based prototype microelastic brittle (PMB) model.
- **bond_stiffness** (numpy.ndarray or float) – An (nregimes, nbond_types) array of bond stiffness values, each corresponding to a bond type and a regime, or a float value of the bond stiffness the Peridynamic bond-based prototype microelastic brittle (PMB) model.
- **displacement_bc_magnitudes** (numpy.ndarray) – (steps,) array of the magnitude applied to the displacement boundary conditions over time.
- **force_bc_magnitudes** (numpy.ndarray) – (steps,) array of the magnitude applied to the force boundary conditions over time.
- **first_step** (*int*) – The starting step number. This is useful when restarting a simulation.
- **write** (*int*) – The frequency, in number of steps, to write the system to a mesh file by calling *Model.write_mesh()*. If None then no output is written. Default None.
- **write_path** (*path-like or str*) – The path where the periodic mesh files should be written.

Returns A tuple of the final displacements (*u*); damage, a tuple of the connectivity; the final node forces (*force*); the final node velocities (*ud*) and a dictionary object containing the displacement, velocity and acceleration (average of), and the forces and body forces for each of the writes (read 'over time'), for each unique tip_type (read 'for each of the set of nodes the user has chosen to measure datum for, as defined by the *is_tip* function).

Return type tuple(`numpy.ndarray`, `numpy.ndarray`, tuple(`numpy.ndarray`, `numpy.ndarray`), `numpy.ndarray`, `numpy.ndarray`, dict)

`write_mesh`(*filename*, *damage=None*, *displacements=None*, *file_format=None*)
Write the model's nodes, connectivity and boundary to a mesh file.

Parameters

- **`filename`** (*`str`*) – Path of the file to write the mesh to.
- **`damage`** (`numpy.ndarray`) – The damage of each node. Default is None.
- **`displacements`** (`numpy.ndarray`) – An array with shape (nnodes, dim) where each row is the displacement of a node. Default is None.
- **`file_format`** (*`str`*) – The file format of the mesh file to write. Inferred from *filename* if None. Default is None.

Returns None

Return type NoneType

3.1.2 Decorators

3.1.3 Warnings

`model.`**`this_may_take_a_while`**(*calculation*)
Raise a UserWarning if nnodes is over 9000! (an arbitrarily large value).

Parameters

- **`nnodes`** (*`int`*) – The number of nodes.
- **`calculation`** (*`str`*) – A string message of the calculation being performed.

Return type `UserWarning` or None

3.1.4 Exceptions

`exception` `peripy.model.`**`DimensionalityError`**(*dimensions*)
An invalid dimensionality argument used to construct a model.

Construct the exception.

Parameters **`dimensions`** (*`int`*) – The number of dimensions passed as an argument to *`Model()`*.

Return type *`DimensionalityError`*

`exception` `peripy.model.`**`FamilyError`**(*family*)
One or more nodes have no bonds in the initial state.

Construct the exception.

Parameters **`family`** (`numpy.ndarray`) – The family array.

Return type *`FamilyError`*

exception `peripy.model.`**`DamageModelError`**(*critical_stretch*)
An invalid critical stretch argument was used to construct a model.

Construct the exception.

Parameters **`critical_stretch`** (`numpy.ndarray` or list) – The critical_stretch array.

Return type *`DamageModelError`*

exception `peripy.model.`**`InvalidIntegrator`**(*integrator*)
An invalid integrator has been passed to *simulate*.

Construct the exception.

Parameters **`integrator`** – The object passed to *`Model.simulate()`* as the integrator argument.

Return type *`InvalidIntegrator`*

3.2 Integrators documentation

3.2.1 Integrator Base Class

class `peripy.integrators.`**`Integrator`**(*dt*, *context=None*)
Base class for integrators.

All integrators must define an init method, which may or may not inherit Integrator as a parent class using *super()*. All integrators that inherit Integrator are OpenCL implementations that can use GPU or CPU. All integrators must also define a call method which performs one integration step, a *_build_special* method which builds the OpenCL programs which are special to the integrator, and a *_create_special_buffers* method which creates the OpenCL buffers which are special to the integrator.

Create an *`Integrator`* object.

This method should be implemented in every concrete integrator.

Parameters

- **`dt`** (*`float`*) – The length of time (in seconds [s]) of one time-step.
- **`context`** (`pyopencl._cl.Context` or NoneType) – Optional argument for the user to provide a context with a single suitable device, default is None.

Returns A *`Integrator`* object

`build`(*nnodes*, *degrees_freedom*, *max_neighbours*, *coords*, *volume*, *family*, *bc_types*, *bc_values*, *force_bc_types*, *force_bc_values*, *stiffness_corrections*, *bond_types*, *densities*)
Build OpenCL programs.

Builds the programs that are common to all integrators and the buffers which are independent of *`peripy.model.Model.simulate()`* parameters.

`create_buffers`(*nlist*, *n_neigh*, *bond_stiffness*, *critical_stretch*, *plus_cs*, *u*, *ud*, *udd*, *force*, *body_force*, *damage*, *regimes*, *nregimes*, *nbond_types*)
Initialise the OpenCL buffers.

Initialises only the buffers which are dependent on *`peripy.model.Model.simulate()`* parameters.

`write`(*u*, *ud*, *udd*, *force*, *body_force*, *damage*, *nlist*, *n_neigh*)
Copy the state variables from device memory to host memory.

3.2.2 Euler

class `peripy.integrators.`**`Euler`**(*dt*)

Euler integrator for cython.

C implementation of the Euler integrator generated using Cython. Uses CPU only. The Euler method is a first-order numerical integration method. The integration is given by,

$$u(t + \delta t) = u(t) + \delta t f(t),$$

where $u(t)$ is the displacement at time t, $f(t)$ is the force density at time t, δt is the time step.

Create an `Euler` integrator object.

Parameters **dt** (`float`) – The length of time (in seconds [s]) of one time-step.

Returns An `Euler` object

`__call__`(*displacement_bc_magnitude*, *force_bc_magnitude*)

Conduct one iteration of the integrator.

Parameters

- **displacement_bc_magnitude** (`float`) – the magnitude applied to the displacement boundary conditions for the current time-step.
- **force_bc_magnitude** (`float`) – the magnitude applied to the force boundary conditions for the current time-step.

`build`(*nnodes*, *degrees_freedom*, *max_neighbours*, *coords*, *volume*, *family*, *bc_types*, *bc_values*, *force_bc_types*, *force_bc_values*, *stiffness_corrections*, *bond_types*, *densities*)

Initiate integrator arrays.

Since `Euler` uses cython in place of OpenCL, there are no OpenCL programs or buffers to be built/created. Instead, this method instantiates the arrays and variables that are independent of `peripy.model.Model.simulate()` parameters as python objects that are used as arguments of the cython functions.

`create_buffers`(*nlist*, *n_neigh*, *bond_stiffness*, *critical_stretch*, *plus_cs*, *u*, *ud*, *udd*, *force*, *body_force*, *damage*, *regimes*, *nregimes*, *nbond_types*)

Initiate arrays that are dependent on simulation parameters.

Initiates arrays that are dependent on `peripy.model.Model.simulate()` parameters. Since `Euler` uses cython in place of OpenCL, there are no buffers to be created, just python objects that are used as arguments of the cython functions.

`write`(*damage*, *u*, *ud*, *udd*, *force*, *body_force*, *nlist*, *n_neigh*)

Return the state variable arrays.

3.2.3 EulerCL

class `peripy.integrators.`**`EulerCL`**(**args*, ***kwargs*)

Euler integrator for OpenCL.

The Euler method is a first-order numerical integration method. The integration is given by,

$$u(t + \delta t) = u(t) + \delta t f(t),$$

where $u(t)$ is the displacement at time t, $f(t)$ is the force density at time t, δt is the time step.

Create an `EulerCL` integrator object.

Returns An `EulerCL` object

`__call__`(*displacement_bc_magnitude*, *force_bc_magnitude*)
Conduct one iteration of the integrator.

Parameters

- **`displacement_bc_magnitude`** (`float`) – the magnitude applied to the displacement boundary conditions for the current time-step.
- **`force_bc_magnitude`** (`float`) – the magnitude applied to the force boundary conditions for the current time-step.

3.2.4 EulerCromerCL

class `peripy.integrators.`**`EulerCromerCL`**(*damping*, **args*, ***kwargs*)
Euler Cromer integrator for OpenCL which can use GPU or CPU.

The Euler-Cromer method is a first-order numerical integration method. The integration is given by,

$$\dot{u}(t + \delta t) = \dot{u}(t) + \delta t \ddot{u}(t),$$

$$u(t + \delta t) = u(t) + \delta t \dot{u}(t + \delta t),$$

where $u(t)$ is the displacement at time t, $\dot{u}(t)$ is the velocity at time t, $\ddot{u}(t)$ is the acceleration at time t, and δt is the time step.

A dynamic relaxation damping term $\eta \dot{u}(t)$ is added to the equation of motion so that the solution to quickly converges to a steady state solution in quasi-static problems. Given the velocity and displacement vectors of each node at time step t, the acceleration at time step t is given by the equation of motion,

$$\ddot{u}(t) = \frac{f(t) - \eta \dot{u}(t)}{\rho},$$

where $f(t)$ is the force density at time t, η is the dynamic relaxation damping constant and ρ is the density.

Create an `EulerCromerCL` integrator object.

Parameters **`damping`** (`float`) – The dynamic relaxation damping constant with units [kg/(m^3 s)]

Returns An `EulerCromerCL` object

`__call__`(*displacement_bc_magnitude*, *force_bc_magnitude*)
Conduct one iteration of the integrator.

Parameters

- **`displacement_bc_magnitude`** (`float`) – the magnitude applied to the displacement boundary conditions for the current time-step.
- **`force_bc_magnitude`** (`float`) – the magnitude applied to the force boundary conditions for the current time-step.

3.2.5 VelocityVerletCL

class `peripy.integrators.`**`VelocityVerletCL`**(*damping*, **args*, ***kwargs*)
Velocity-Verlet integrator for OpenCL.

The Velocity-Verlet method is a second-order numerical integration method. The integration is given by,

$$\dot{u}(t + \frac{\delta t}{2}) = \dot{u}(t) + \frac{\delta t}{2}\ddot{u}(t),$$

$$u(t + \delta t) = u(t) + \delta t \dot{u}(t) + \frac{\delta t}{2}\ddot{u}(t),$$

$$\dot{u}(t + \delta t) = \dot{u}(t + \frac{\delta t}{2}) + \frac{\delta t}{2}\ddot{u}(t + \delta t),$$

where $u(t)$ is the displacement at time t, $\dot{u}(t)$ is the velocity at time t, $\ddot{u}(t)$ is the acceleration at time t and δt is the time step.

A dynamic relaxation damping term $\eta \dot{u}(t)$ is added to the equation of motion so that the solution to quickly converges to a steady state solution in quasi-static problems. Given the displacement vectors of each node at time step t, and half-step velocity vectors of each node at time step $t + \frac{\delta t}{2}$, the acceleration at time step $t + \delta t$ is given by the equation of motion,

$$\ddot{u}(t + \delta t) = \frac{f(t + \delta t) - \eta \dot{u}(t + \frac{\delta t}{2})}{\rho},$$

where $f(t)$ is the force density at time t, η is the dynamic relaxation damping constant and ρ is the density.

Create an *VelocityVerletCL* integrator object.

Parameters **damping** (*float*) – The dynamic relaxation damping constant with units [kg/(m^3 s)]

Returns A *VelocityVerletCL* object

__call__(*displacement_bc_magnitude*, *force_bc_magnitude*)
Conduct one iteration of the integrator.

Parameters

- **displacement_bc_magnitude** (*float*) – the magnitude applied to the displacement boundary conditions for the current time-step.
- **force_bc_magnitude** (*float*) – the magnitude applied to the force boundary conditions for the current time-step.

3.2.6 Exceptions

exception peripy.integrators.**ContextError**
No suitable context was found by `get_context()`.

Exception constructor.

3.3 Spatial documentation

peripy.spatial.**euclid**()
Calculate the Euclidean distance between two, three-dimensional coordinates.

Parameters

- **r1** (numpy.ndarray) – A (3,) array representing the first coordinate.
- **r2** (numpy.ndarray) – A (3,) array representing the second coordinate.

Returns The Euclidean distance between *r1* and *r2*

Return type *numpy.float64*

`peripy.spatial.`**`strain`**`()`

Calculate the strain between two particles given their current and initial positions.

Parameters

- **`r1`** (`numpy.ndarray`) – A (3,) array giving the coordinate of the first particle.
- **`r2`** (`numpy.ndarray`) – A (3,) array giving the coordinate of the second particle.
- **`r10`** (`numpy.ndarray`) – A (3,) array giving the initial coordinate of the first particle.
- **`r20`** (`numpy.ndarray`) – A (3,) array giving the initial coordinate of the second particle.

Returns The strain.

Return type *numpy.float64*

`peripy.spatial.`**`strain2`**`()`

Calculate the strain between two particles given their current distance and initial positions.

Parameters

- **`l`** (*numpy.float64*) – The Euclidean distance between the two particles.
- **`r10`** (`numpy.ndarray`) – A (3,) array giving the initial coordinate of the first particle.
- **`r20`** (`numpy.ndarray`) – A (3,) array giving the initial coordinate of the second particle.

Returns The strain.

Return type *numpy.float64*

3.4 Correction factors documentation

`peripy.correction.`**`set_imprecise_surface_correction`**`()`

Calculate the surface corrections using an average nodal volume.

Uses the 'volume method' proposed by Bobaru et al. [Chapter 2 in Bobaru F, Foster JT, Geubelle PH, Silling SA, Handbook of peridynamic modeling, (2017) (p51 – 52)].

Parameters

- **`stiffness_corrections`** (`numpy.ndarray`) – The stiffness corrections.
- **`nlist`** (`numpy.ndarray`) – The neighbour list
- **`n_neigh`** (`numpy.ndarray`) – The number of neighbours for each node.
- **`average_volume`** (*`double`*) – The average nodal volume.
- **`family_volume_bulk`** (*`double`*) – Volume of a family in the bulk material.

`peripy.correction.`**`set_micromodulus_function`**`()`

Calculate the normalised conical micromodulus function values given the initial coordinates and peridynamic horizon.

Parameters

- **`micromodulus_values`** (`numpy.ndarray`) – The micromodulus values.
- **`r0`** (`numpy.ndarray`) – The initial coordinates of each node.
- **`nlist`** (`numpy.ndarray`) – The neighbour list

- **n_neigh** (numpy.ndarray) – The number of neighbours for each node.
- **horizon** (*float*) – The critical strain.
- **micromodulus_function** (*int*) – A flag variable denoting the micromodulus function used. Set to 0: Uses the normalised connical micromodulus function. Otherwise: Uses a constant micromodulus function.

peripy.correction.**set_precise_surface_correction**()

Calculate the surface corrections given actual nodal volumes.

Uses the 'volume method' proposed by Bobaru et al. [Chapter 2 in Bobaru F, Foster JT, Geubelle PH, Silling SA, Handbook of peridynamic modeling, (2017) (p51 – 52)].

Parameters

- **stiffness_corrections** (numpy.ndarray) – The stiffness corrections.
- **nlist** (numpy.ndarray) – The neighbour list
- **n_neigh** (numpy.ndarray) – The number of neighbours for each node.
- **volume** (numpy.ndarray) – The nodal volumes.
- **family_volume_bulk** (*double*) – Volume of a family in the bulk material.

peripy.correction.**set_volume_correction**()

Calculate the partial volume corrections given the initial coordinates, peridynamic horizon and node radius.

Parameters

- **volume_corrections** (numpy.ndarray) – The volume corrections.
- **r0** (numpy.ndarray) – The initial coordinates of each node.
- **nlist** (numpy.ndarray) – The neighbour list
- **n_neigh** (numpy.ndarray) – The number of neighbours for each node.
- **horizon** (*float*) – The critical strain.
- **node_radius** (*float*) – The node radius.
- **volume_correction** (*int*) – A flag variable denoting the algorithm used. Set to 0: Uses the 'Partial Area/Volume HHB algorithm', as proposed by Hu, Ha, and Bobaru [W. Hu, Y.D. Ha and F. Bobaru, Numerical integration in peridynamics, Tech. Rep., University of Nebraska-Lincoln, Department of Mechanical & Materials Engineering (September 2010)]. Otherwise: The 'Full Volume algorithm' is used; partial nodal volumes are approximated by their full nodal volumes.

3.5 Create crack documentation

peripy.create_crack.**create_crack**()

Create a crack by removing selected pairs from the neighbour list.

Parameters

- **crack** (numpy.ndarray) – An array giving the pairs between which to create the crack. Each row of this array should be the index of two nodes.
- **nlist** (numpy.ndarray) – The neighbour list
- **n_neigh** (numpy.ndarray) – The number of neighbours for each node.

3.6 Peridynamics documentation

Note: These are the peridynamics methods for the cython implementation. For the OpenCL implementation

`peripy.peridynamics.bond_force()`
Calculate the force due to bonds on each node.

Parameters

- **r** (numpy.ndarray) – The current coordinates of each node.
- **r0** (numpy.ndarray) – The initial coordinates of each node.
- **nlist** (numpy.ndarray) – The neighbour list
- **n_neigh** (numpy.ndarray) – The number of neighbours for each node.
- **volume** (numpy.ndarray) – The volume of each node.
- **bond_stiffness** (*float*) – The bond stiffness.
- **force_bc_values** (numpy.ndarray) – The force boundary condition values for each node.
- **force_bc_types** (numpy.ndarray) – The force boundary condition types for each node.
- **bc_scale** (*double*) – The scalar value applied to the force boundary conditions.

`peripy.peridynamics.break_bonds()`
Update the neighbour list and number of neighbours by breaking bonds which have exceeded the critical strain.

Parameters

- **r** (numpy.ndarray) – The current coordinates of each node.
- **r0** (numpy.ndarray) – The initial coordinates of each node.
- **nlist** (numpy.ndarray) – The neighbour list
- **n_neigh** (numpy.ndarray) – The number of neighbours for each node.
- **critical_strain** (*float*) – The critical strain.

`peripy.peridynamics.damage()`
Calculate the damage for each node.

Damage is defined as the ratio of broken bonds at a node to the total number of bonds at that node.

Parameters

- **n_neigh** (numpy.ndarray) – The current number of neighbours for each node, *i.e.* the number of unbroken bonds. dtype=numpy.int32.
- **family** (numpy.ndarray) – The total (initial) number of bonds for each node. dtype=numpy.int32.

`peripy.peridynamics.update_displacement()`
Update the displacement of each node for each node using an Euler integrator.

Parameters

- **u** (numpy.ndarray) – The current displacements of each node.
- **bc_values** (numpy.ndarray) – An (n,3) array of the boundary condition values.

- **`bc_types`** (`numpy.ndarray`) – An (n,3) array of the boundary condition types, where a zero value represents an unconstrained node.
- **`force`** (`numpy.ndarray`) – The force due to bonds on each node.
- **`bc_scale`** (*`float`*) – The scalar value applied to the displacement boundary conditions.
- **`dt`** (*`float`*) – The length of the timestep in seconds.

3.7 Utilities documentation

Utility functions that are unrelated to peridynamics.

`peripy.utilities.`**`read_array`**(*read_path*, *dataset*)

Read a :class numpy.ndarray: from a HDF5 file.

Parameters

- **`read_path`** (*`path-like or str`*) – The path to which the HDF5 file is written.
- **`dataset`** (*`str`*) – The name of the dataset stored in the HDF5 file.

Returns An array which was stored on disk.

Return type

class numpy.ndarray

`peripy.utilities.`**`write_array`**(*write_path*, *dataset*, *array*)

Write a :class: numpy.ndarray to a HDF5 file.

Parameters

- **`write_path`** (*`path-like or str`*) – The path to which the HDF5 file is written.
- **`dataset`** (*`str`*) – The name of the dataset stored in the HDF5 file.

Array The array to be written to file.

Returns None

Return type None type

3.8 Indices and tables

- genindex
- modindex
- search

CHAPTER 4

How to contribute

File bug reports or feature requests, and make contributions (e.g. code patches), by opening a “new issue” on GitHub:

- PeirPy Issues: https://github.com/alan-turing-institute/Peripy/issues

Please give as much information as you can in the ticket. It is extremely useful if you can supply a small self-contained code snippet that reproduces the problem. Also specify the version you are referring to.

CHAPTER 5

MIT License

Copyright (c) 2020 Tim Dodwell

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Python Module Index

Index