User Manual

Ifasha (version 0.3)

David S. Kammer
davekammer@gmail.com

December 1, 2015

Contents

1 How to Start

This code can be used to

1. post-process data that was dumped by IOHelper during simulations within Akantu,
2. access efficiently data that was post-processed,
3. compute analytic solutions of solid mechanics and fracture mechanics (LEFM).

By adding the new directory to the python path, you can use the method everywhere on your system. On possibility is to add the following line to your *.bashrc* file.

```
export PYTHONPATH=$PYTHONPATH:/home/user/sources
```

and then put entire ifasha into the *sources* folder.

2 FieldId

The FieldId is the way an output field is identified. It needs the following information:

- name:
  - *description*: the field's name, as given by the simulation
- dir:
  - *description*: the direction of the field. x=0, y=1, z=2

```
from ifasha.datamanager import FieldId
fid = FieldId(name, 0)
```

The time fields are global fields and currently there are: `step` and `time`

3 DataManager

This is the base class that handles multiple data output of a single simulation. The inherited class DataManagerAnalysis can be used to access data for further analysis.

3.1 init

The DataManager is created by providing its name.

```
from ifasha.datamanager import DataManager
dm = DataManager(name, [path='./'], [create=False])
```

- **name**: name of simulation to be loaded
- **path**: path to where the simulation is saved (default value = './')
- **create**: if data does not exist, should it be created?

3.2 destroy

This destroys all data corresponding to this DataManager.

```
dm.destroy()
```

3.3 get_field_collection

Returns a FieldCollection object of data contained in the present DataManager.

```
from ifasha.datamanager import FieldCollection
fc = dm.get_field_collection(name)
```

- **name**: name of FieldCollection to be returned

3.4 get_new_field_collection

Creates a new FieldCollection in the DataManager and returns it.

```
from ifasha.datamanager import FieldCollection
fc = dm.get_new_field_collection(name)
```

- **name**: name of FieldCollection to be created and returned.

If a FieldCollection with the same name exists already, it will raise a RuntimeError.

3.5 remove_field_collection

Removes a FieldCollection from the DataManager and destroys it.

```
dm.remove_field_collection{name}
```

- **name**: name of FieldCollection to be removed and destroyed.

3.6 add_supplementary

Add an supplementary file with auxiliary information (*e.g.,* input file or output file)

```
dm.add_supplementary(fname, path, [replace=False])
```

- **fname**: name to be given to the file stored into this DataManager
- **path**: path to file that will be copied into the DataManager
- **replace**: should it be replaced, if a file with same *fname* exists already?

3.7 remove_supplementary

Remove a supplementary file from the DataManager

```
dm.remove_supplementary(fname)
```

- **fname**: name of file to be removed

If file does not exist, this method does nothing.

3.8 Examples

-To do.

4 DataManagerAnalysis

This class adds a method to the DataManager that helps you to get a FieldCollectionAnalysis object.

4.1 init

Similar to the DataManager, this initiates an object. The main difference is that it does not create a new object, if none with the provided name exists. This is to avoid the creating of new object by mistake (because you don't know how to write the name of your data).

```
from ifasha.datamanager import DataManagerAnalysis
dma = DataManagerAnalysis(name, [path='./'])
```

- **name**: name of simulation to be loaded
- **path**: path to where the simulation is saved (default value = './')

4.2 operator()

This operator provides you with the possibility of getting a FieldCollectionAnalysis object from the DataManagerAnalysis.

```
fca = dma(name)
```

- **name**: name of FieldCollection you would like to access as a FieldCollectionAnalysis object.

5 FieldCollectionAnalysis

The FieldCollectionAnalysis helps to extract data from the DataManager in order to plot and analysis it. It inherits all methods from FieldCollection. See init for how to get an instant from the DataManagerAnalysis.

Different methods exist to access data:

- full field: get_full_field
- at a time: get_field_at_t, get_t_index, get_index_of_closest_time, get_field_at_t_index
- at a position: get_index_of_closest_position, get_field_at_node_index
- 2d plot: get_x_y_plot, get_sliced_x_y_plot, get_nb_points_in_direction, get_x_t_plot, get_sliced_x_t_plot

5.1 get_full_field

Returns the MemMap of the field. For nodal and elemental fields, the first dimension is the index of time and the second dimension the index of the node/element. For global fields (*e.g.* step, time), there is only the first dimension for the time index and no second dimension exists.

```
fmmap = fca.get_full_field(fldid)
```

- **fldid**: FieldId of the field
- **fmmap**: MemMap with data.
  - nodal/elemental field: `fmmap[tindex,xindex]` with the time index `tindex` and the node/element index `xindex`
  - global field: `fmmap[tindex]` with the time index `tindex`

5.2 get_field_at_t

Returns an array of the field as close to a given time, where the time can also be a step.

```
farray = data.get_field_at_t(fldid, tid, ts)
```

- **fldid**: FieldId of the field.
- **tid**: FieldId of the time field. Depending on whether `ts` are a times or a time steps.

- **ts**: list of time steps or times
- **farray**: array of the fields at the times, see get_field_at_t_index.

5.3 get_t_index

Returns a time index or a list of time indexes for a given indicator or list of indicators.

```
idxs = fca.get_t_index(inds)
```

- **inds**: an indicator or a list of indicators. Currently there are: `"first" "middle" "last"`
- **idxs**: time index or list of time indexes, depending whether `inds` is a list.

5.4 get_index_of_closest_time

Returns a time index or a list of time indexes for a given time field (*e.g.* `step`, `time`) and a "time" or list of "times".

```
idxs = fca.get_index_of_closest_time(fldid, tms)
```

- **fldid**: FieldId of any time field
- **tms**: time/step or list of times/steps
- **idxs**: time index or list of time indexes, depending whether `tms` is a list.

5.5 get_field_at_t_index

Returns an array of the field at the given indexes.

```
farray = fca.get_field_at_t_index(fldid, idxs)
```

- **fldid**: FieldId of the field
- **idxs**: list of time indexes
- **farray**: array of field
  - nodal/elemental field: `farray[i,xindex]`
  - global field: `farray[i]`

  with `i` the index of the time index in the `idxs` list, and `xindex` the index of the node/element.

5.6 get_index_of_closest_position

Returns the index of the closest node for a given point.

```
idx = fca.get_index_of_closest_position(fldids, vlus)
```

- **fldids**: list of FieldId.
- **vlus**: list of values of the position
- **idx**: node index of closest node

This method uses the first (in time) entry of the given field. Considering this is the position, which does not change. It cannot find the closest position at a position for a later point.

Example for finding a node that is closest to the point $x = 3.$ and $y = 5.$:

```
idx = fca.get_index_of_closest_position([FieldId("position",0),
                                         FieldId("position",1)],[3.,5.])
```

5.7 get_field_at_node_index

Returns the field for a list of given nodes indicated by their indexes.

```
farray = fca.get_field_at_node_index(fldid, idxs)
```

- **fldid**: FieldId of the field
- **idxs**: list of node indexes
- **farray**: array of the field with `farray[tindex,i]` where `tindex` is the time index and `i` is the index of the node indexes in the `idxs` list.

5.8 get_nb_points_in_direction

Returns the number of points for a 2D plot in the direction of the field position field provided.

```
nbpts = fca.get_nb_points_in_direction(xfldid,tidx)
```

- **xfldid**: FieldId of the position field
- **tidx**: time index (with default value = 0)
- **nbpts**: number of points in the direction of the position field

5.9 get_x_y_plot

Returns three MemMaps that can be used to plot as a 2D color plot.

```
X,Y,Z = fca.get_x_y_plot(xfldid, yfldid, zfldid, tidx)
```

- **xfldid**: FieldId of the X field
- **yfldid**: FieldId of the Y field
- **zfldid**: FieldId of the Z field
- **tidx**: time index
- **X,Y,Z**: MemMaps that can be used for 2D color plots.

For an example see get_sliced_x_sliced_y_plot.

5.10 get_sliced_x_sliced_y_plot

Returns three MemMaps that can be used to plot as a 2D color plot (see get_x_y_plot) where X and Y are sliced.

```
X,Y,Z = fca.get_x_y_plot(xfldid, xslice, yfldid, yslice, zfldid, tidx)
```

- **xfldid**: FieldId of the X field
- **xslice**: slices in X direction
- **yfldid**: FieldId of the Y field
- **yslice**: slices in Y direction
- **zfldid**: FieldId of the Z field
- **tidx**: time index
- **X,Y,Z**: MemMaps that can be used for 2D color plots.

Example showing how to use this method:

```
#!/usr/bin/env python

import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import axis3d

X,Y,Z = fca.get_sliced_x_sliced_y_plot(FieldId("position",0),
                                       np.arange(0,201,2),
                                       FieldId("position",1),
                                       np.arange(1,21,3),
                                       FieldId("velocity",0))

fig = plt.figure()
ax = fig.add_subplot(111)
ax.pcolor(X,Y,Z)
plt.show()
```

5.11 get_sliced_x_y_plot

Returns three MemMaps that can be used to plot as a 2D color plot (see get_x_y_plot) where X is sliced.

```
X,Y,Z = fca.get_sliced_x_y_plot(xfldid, xslice, yfldid, zfldid, tidx)
```

- **xfldid**: FieldId of the X field
- **xslice**: slices in X direction
- **yfldid**: FieldId of the Y field

- **zfldid**: FieldId of the Z field
- **tidx**: time index
- **X,Y,Z**: MemMaps that can be used for 2D color plots.

5.12 get_x_sliced_y_plot

Returns three MemMaps that can be used to plot as a 2D color plot (see get_x_y_plot) where Y is sliced.

```
X,Y,Z = fca.get_x_sliced_y_plot(xfldid, yfldid, yslice, zfldid, tidx)
```

- **xfldid**: FieldId of the X field
- **yfldid**: FieldId of the Y field
- **yslice**: slices in Y direction
- **zfldid**: FieldId of the Z field
- **tidx**: time index
- **X,Y,Z**: MemMaps that can be used for 2D color plots.

5.13 get_x_t_plot

Returns three MemMaps that can be used to plot as a 2D color plot. This works only for one dimensional data.

```
X,T,V = fca.get_x_t_plot(xfldid, tfldid, vfldid)
```

- **xfldid**: FieldId of the X field
- **tfldid**: FieldId of the time field T
- **vfldid**: FieldId of the V field
- **X,T,V**: MemMaps that can be used for 2D space-time color plots

For an example see: get_sliced_x_sliced_t_plot

5.14 get_sliced_x_sliced_t_plot

Returns three MemMaps that can be used to plot as a 2D color plot (see get_x_t_plot) where X and T are sliced. This works only for one dimensional data.

```
X,T,V = fca.get_sliced_x_sliced_t_plot(xfldid, xslice, tfldid, tslice, vfldid)
```

- **xfldid**: FieldId of the X field
- **xslice**: slices in X direction

- **tfldid**: FieldId of the time field T
- **tslice**: slices in time T
- **vfldid**: FieldId of the V field
- **X,T,V**: MemMaps that can be used for 2D space-time color plots

Example showing how to use this method:

```
#!/usr/bin/env python

import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import axis3d

X,T,Z = data.get_sliced_x_sliced_t_plot(FieldId("position",0),
                                        np.arange(0,201,2),
                                        FieldId("time"),
                                        np.arange(1,1000,10),
                                        FieldId("velocity",0))

fig = plt.figure()
ax = fig.add_subplot(111)
ax.pcolor(X,T,Z)
plt.show()
```

5.15 get_sliced_x_t_plot

Returns three MemMaps that can be used to plot as a 2D color plot (see get_x_t_plot) where X is sliced. This works only for one dimensional data.

```
X,T,V = fca.get_sliced_x_t_plot(xfldid, xslice, tfldid, vfldid)
```

- **xfldid**: FieldId of the X field
- **xslice**: slices in X direction
- **tfldid**: FieldId of the time field T
- **vfldid**: FieldId of the V field
- **X,T,V**: MemMaps that can be used for 2D space-time color plots

5.16 get_x_sliced_t_plot

Returns three MemMaps that can be used to plot as a 2D color plot (see get_x_t_plot) where T is sliced. This works only for one dimensional data.

```
X,T,V = fca.get_x_sliced_t_plot(xfldid, tfldid, tslice, vfldid)
```

- **xfldid**: FieldId of the X field

- **tfldid**: FieldId of the time field T
- **tslice**: slices in time T
- **vfldid**: FieldId of the V field
- **X,T,V**: MemMaps that can be used for 2D space-time color plots

5.17 make_it_pretty

This is a static method which improves the X,Y or X,T matrices for pretty plots. The shape of the matrices is increased in both dimension by one and values are changed such that colour plots have squares with the centre position of the value

```
Xp, Yp = FieldCollectionAnalysis.make_it_pretty(X,Y)
```

- **X**: X MemMap from get_x_y_plot or get_x_t_plot.
- **Y**: Y MemMap from get_x_y_plot or T MemMap from get_x_t_plot
- **Xp**: X MemMap for pretty plot
- **Yp**: Y MemMap for pretty plot

Do not use values in Xp and Yp for computation, only for pretty plots!

5.18 Examples

Plot global variables:

```
#!/usr/bin/env python

from ifasha.datamanager import FieldId
from ifasha.datamanager import DataManagerAnalysis
from ifasha.datamanager import FieldCollectionAnalysis

import matplotlib.pyplot as plt

# location of simulation data
sim_name = "test"
fc_name = "global"
path = "/home/researcher/post_process"

# load the simulation data
dma = DataManagerAnalysis(sim_name,path)
fca = dma(fc_name)

# get time as well as kinetic and potential energy (all global variables)
time = fca.get_full_field(FieldId("time"))
Ekin = fca.get_full_field(FieldId("kinetic_energy"))
Epot = fca.get_full_field(FieldId("potential_energy"))

# prepare plot
fig = plt.figure()
```

```
ax = fig.add_subplot(111)

# plot energies w.r.t. time
ax.plot(time, Ekin, ".-", label="kinetic energy")
ax.plot(time, Epot, ".-", label="potential energy")
ax.plot(time, Ekin+Epot, ".-", label="total energy")

# finalize plot
ax.set_xlabel(r"time $t$ (s)")
ax.set_ylabel(r"energy $E$ (GJ)")
ax.legend()

plt.show()
```

Plot data on a boundary at a given time:

```
#!/usr/bin/env python


from ifasha.datamanager import FieldId
from ifasha.datamanager import DataManagerAnalysis
from ifasha.datamanager import FieldCollectionAnalysis

import matplotlib.pyplot as plt

# location of simulation data
sim_name = "test"
fc_name = "global"
path = "/home/researcher/post_process"

# load the simulation data
dma = DataManagerAnalysis(sim_name,path)
fca = dma(fc_name)

# define time at which to plot
time_t = 0.1
tfld = FieldId("time")
tidx = fca.get_index_of_closest_time(tfld,time_t)

# get position in x and velocity in x and y
positionX = fca.get_field_at_t_index(FieldId("position",0),tidx)
velocityX = fca.get_field_at_t_index(FieldId("velocity",0),tidx)
velocityY = fca.get_field_at_t_index(FieldId("velocity",1),tidx)

# sort all fields together w.r.t. the position x
# the 0 indicates that this is for the first time (tidx) given above
# tidx could be a list: there would be more than one entry in the first dimension
fltr = positionX.argsort()
positionX = positionX[0,fltr]
velocityX = velocityX[0,fltr]
velocityY = velocityyy[0,fltr]

# prepare plot
fig = plt.figure()
```

```
ax = fig.add_subplot(111)

# plot energies w.r.t. time
ax.plot(positionX[0,:], velocityX[0,:], ".-", label="velocity in x")
ax.plot(positionX[0,:], velocityY[0,:], ".-", label="velocity in y")

# finalize plot
ax.set_xlabel(r"position $x$ (m)")
ax.set_ylabel(r"velocity $v$ (m/s)")
ax.legend()

plt.show()
```

6 RuptureHunter

Tracks ruptures from simulation data.

```
from ifasha.ruptureshunter import RuptureHunter
hunter = RuptureHunter()
```

6.1 load

Loads input from the simulation data, using the FieldCollectionAnalysis.

```
hunter.load(position, time, stick)
```

- **position**: get data using get_field_at_t_index at `idxs = 0`.
- **time**: get data using get_full_field and the `time` FieldId
- **stick**: get data using getFullFieldfcaget_full_field and the `is_sticking` FieldId

6.2 hunt

Tracks the ruptures.

```
hunter.hunt()
```

Data has to be loaded before with load.

6.3 free_small_ruptures

Deletes all ruptures up to the indicated length.

```
hunter.free_small_ruptures(max)
```

- **max**: the maximal length of ruptures that are deleted

6.4 free

Deletes all ruptures given by a set.

```
hunter.free(rpts)
```

- **rpts**: set of rupture index.

6.5 renumber

Renumbers the indexes of the ruptures in order to: 1) have continuous rupture indexes, and 2) to start with the first rupture in time and to end with the last rupture in time.

```
hunter.renumber()
```

6.6 get_expansions

Returns a list of expansion data for all ruptures (sorted in indexes of ruptures)

```
exps = hunter.get_expansions()
```

- **exps**: list of expansion data for each rupture (sorted w.r.t. the rupture indexes). `exps[ri,0]` is the smallest and `exps[ri,1]` the largest position for the rupture of index `ri`, respectively.

6.7 get_fronts

Returns a list of arrays for sorted (w.r.t. the rupture index) fronts of all ruptures. It is the same order as the rupture indexes given by get_rupture_indexes.

```
fronts = hunter.get_fronts()
```

- **fronts**: list of rupture fronts for all ruptures, sorted w.r.t. to rupture index. A front `f = fronts[ri]` of rupture with index `ri` is sorted w.r.t. to the position and is structured as `f[x,t]` with `x` the coordinate and `t` the time.

6.8 get_backs

Returns a list of arrays for sorted (w.r.t. the rupture index) backs of all ruptures. It is the same order as the rupture indexes given by get_rupture_indexes.

```
backs = hunter.get_backs()
```

- **backs**: list of rupture backs for all ruptures, sorted w.r.t. to rupture index. A front `b = backs[ri]` of rupture with index `ri` is sorted w.r.t. to the position and is structured as `b[x,t]` with `x` the coordinate and `t` the time.

6.9 get_rupture_indexes

Returns a list of the rupture indexes. It is sorted by index.

```
rptidxs = hunter.get_rupture_indexes()
```

- **rptidxs**: list of rupture indexes sorted by index.

6.10 get_rupture

Returns a rupture for a given index.

```
rpt = hunter.get_rupture(rptidx)
```

- **rptidx**: index of rupture
- **rpt**: Rupture

6.11 Examples

```
#!/usr/bin/env python

from ifasha.datamanager import FieldId
from ifasha.datamanager import FieldCollectionAnalysis
from ifasha.ruptureshunter import RuptureHunter

import matplotlib.pyplot as plt

# location of simulation data
sim_name = "test"
fc_name = "interface"
path = "/home/researcher/post_process"

# load the simulation data
dma = DataManagerAnalysis(sim_name, path)
fca = dma.load(fc_name)

# get position
index    = fca.get_t_index("first")
position = fca.get_field_at_t_index(FieldId("position",0),index)

# get time and stick data
time  = data.get_full_field(FieldId("time"))
stick = data.get_full_field(FieldId("is_sticking"))

# load RuptureHunter and hunt ruptures
hunter = RuptureHunter()
hunter.load(position, time, stick)
hunter.hunt()

# delete Ruptures that have propagate not further than two position
hunter.free(2)
```

```
hunter.renumber() # renumber in order to have continuous rupture indexes

# get the expansion of all ruptures
rptidxs   = hunter.get_rupture_indexes()
fronts    = hunter.get_fronts()
backs     = hunter.get_backs()
expansions = hunter.get_expansions()

# prepare plot
fig = plt.figure()

# plot fronts and backs
ax = fig.add_subplot(211)
for ft, bk in zip(fronts,backs):
  ax.plot(ft[:,0], ft[:,1], "ko", label="front")
  ax.plot(bk[:,0], bk[:,1], "ro", label="back")
ax.set_xlabel(r"position $x$")
ax.set_ylabel(r"time $t$")
ax.legend()

# plot expansions
ax = fig.add_subplot(212)
ax.plot(expansions[:,0], rptidxs, "bo", label="min")
ax.plot(expansions[:,1], rptidxs, "go", label="max")
ax.set_xlabel(r"position $x$")
ax.set_ylabel(r"rupture index")
ax.legend()

plt.show()
```