fedempy

Dec 20, 2023

TABLE OF CONTENTS

Welcome to the `fedempy` python package!

These pages contain the documentation of the modules of `fedempy`, and some HOWTO pages with various topics on the usage of it.

CHAPTER

ONE

FEDEMPY HOWTO

1.1 Installing fedempy

To install the latest available `fedempy` package on Windows, just give the following command in the console window:

```
pip install fedempy
```

For this to work, you first need to copy the file pip.conf into *%APPDATA%\pip\pip.ini*, where *%APPDATA%* is the environment variable connected to your user profile that points to where applications will have data and settings stored. Notice the extension of the copied file needs to be *.ini* (as opposed to *.conf* on Linux). If you you are running python in a virtual environment, the file can also be placed in the root folder of the virtual environment file system instead.

Before using the `fedempy` modules, the following environment variables need to be set:

FEDEM_REDUCER = path to the Fedem reducer shared object library (*fedem_reducer_core.dll*)
FEDEM_SOLVER = path to the Fedem solver shared object library (*fedem_solver_core.dll*)
FEDEM_MDB = path to the Fedem mechanism database shared object library (*FedemDB.dll*)

The current version of `fedempy` is #FEDEMPY_VERSION# and is compatible with **Fedem #FEDEM_VERSION#**.

1.2 Creating/editing Fedem models

This section gives a brief introduction on how you can use the `fedempy` package to generate new Fedem models through python scripting. It relies on the *`modeler`* module.

The modeling methods are collected in the class `FedemModeler`, so to access them, start your python script by:

```
from fedempy.modeler import FedemModeler
from fedempy.enums import FmDof, FmDofStat, FmLoadType, FmType, FmVar
```

You also need to set the environment variable **FEDEM_MDB** to point to the shared object library of the Fedem mechanism model database, before executing the script. This library is named *libFedemDB.so* on Linux (*FedemDB.dll* on Windows).

1.2.1 Opening a new/existing model

To establish a Fedem model object, use:

```
myModel = FedemModeler("mymodel.fmm")
```

If the file "mymodel.fmm" exists, this will open that model, and any subsequent modeling operations will alter or append objects to that model. If you want to force opening a new empty model, specify *True* as the second parameter, viz.:

```
myModel = FedemModeler("mymodel.fmm", True)
```

The specified model file will then be overwritten if it already exists when saving the model.

1.2.2 Saving and closing current model

To save the current model, use:

```
if not myModel.save():
    raise Exception({"Error": "Failed to save current model"})
```

To give it a new name (Save As):

```
if not myModel.save("newname.fmm"):
    raise Exception({"Error": "Failed to save to newname.fmm"})
```

When finished, the model should be closed to release all internal memory:

```
myModel.close()
```

1.2.3 Creating objects

The `FedemModeler` class has several methods for creating mechanism objects of different type. They all have a signature like `make_<object_type> ("description", [attributes])` and return an integer which is (with two exceptions, see below) the base Id of the generated object. This value can be used as a handle to that object in other modeling operations. If the object could not be created or an error occurred, either zero or a negative value is returned. Your modeling script should therefore always check that the returned value is positive before continuing.

See the *modeler.FedemModeler* documentation for a full overview of the available methods. In the following, some example statements for creating objects are presented.

FE parts

A FE model stored in one of the supported FE data file formats can be imported as an FE part into the current Fedem model, by using:

```
p1 = myModel.make_fe_part(fe_data_file)
print("Created a FE Part with base Id", p1)
```

where *fe_data_file* is the full path of the FE data file to be imported. No other attributes of the part can be specified with this method. See *Editing parts* below, for how to change it's properties.

Triads

To create a triad object at the global location (1.0, 2.0, 3.5), use:

```
t1 = myModel.make_triad("My first triad", (1, 2, 3.5))
print("Created a Triad with base Id", t1)
```

If the triad should be attached to a part, the base Id of that part may be specified using the optional *on_part* argument, as follows:

```
t2 = myModel.make_triad("My second triad", (1, 2, 3.5), on_part=part_id)
print("Created a Triad with base Id", t2, "on Part", part_id)
```

This assumes that the coordinates provided match a nodal point in the FE model of the part specified. If the triad should be attached to ground, the base Id of the Reference plane is specified instead, which usually equals 2, e.g.:

```
t3 = myModel.make_triad("My third triad", (1, 2, 3.5), on_part=2)
print("Created a grounded Triad with base Id", t3)
```

Finally, if the nodal point of the triad to be created is known, you may specify that instead of the coordinates, viz.:

```
t4 = myModel.make_triad("My fourth triad", node=node_id, on_part=part_id)
print("Created a Triad with base Id", t4, "on Part", part_id)
```

where *node_id* is the Id of a nodal point in the FE model of the part *part_id*.

See *Editing triads* below, for how to further modify the properties of created triads.

Beam elements

To create a string of three beam elements connected to the four triads, *t1*, *t2*, *t3* and *t4*, you can use:

```
beams = myModel.make_beam("My beams", [t1, t2, t3, t4], prop_id)
print("Created beam elements with base Ids", beams)
```

This method returns a list of base Id values for the created beam elements (or *None* if an error occured). The last parameter in the above call is the base Id of a cross section property, which is created by:

```
mat_id = myModel.make_beam_material("Steel", (7850, 2.1e11, 0.3))
prop_id = myModel.make_beam_section("Pipe", mat_id, (0.5, 0.45))
```

or:

```
prop_id = myModel.make_beam_section("General", 0, section_data)
```

The first variant above creates a pipe cross section with outer diameter 0.5 and inner diameter 0.45, and connected to a material object with mass density 7850.0, Young's modulus 2.1e11 and Poisson's ratio 0.3. The second variant creates a generic cross section, where `section_data` is a list of up to 10 cross section property values:

```
section_data = [EA, EIy, EIz, GIt, rhoL, RhoIp, GAy, GAsz, sy, sz]
```

If you specify less than 10 values, the remaining values will be assumed equal to zero.

Joints

To attach a triad to ground using a rigid joint, use:

```
joint1 = myModel.make_joint("Fixed", FmType.RIGID_JOINT, t1)
```

To connect two triads via a revolute joint:

```
joint2 = myModel.make_joint("Hinge", FmType.REVOLUTE_JOINT, t2, t3)
```

The first triad specified (t2) will then be the dependent joint triad and the second triad (t3) will be the independent triad. You can also specify `FmType.BALL_JOINT` and `FmType.FREE_JOINT` as joint type.

To connect triads via a cylindric joint, use:

```
joint3 = myModel.make_joint ("Cylindric", FmType.CYLINDRIC_JOINT, t0, [t1, t2, ..., tn])
```

After the dependent triad (t0), a list of independent triads (t1, t2, ..., tn) is specified. The first two list items (t1 and t2) are taken as the start and end position of the joint, and the subsequent triads (t3, ...), will by the in-between triads. The latter must lie on a straight line through the start and end triads, otherwise they won't be taken into account.

To create a prismatic joint, use the enum value `FmType.PRISMATIC_JOINT` instead.

See *Editing joints* below, for how to further modify the properties of created joints.

Springs and Dampers

To create an axial spring with a piece-wise linear stiffness function, the following will work:

```
spr1 = myModel.make_spring("My first spring", (t1, t2),
                           xy=[[-0.1, 10.0], [0.0, 0.1], [0.1, 10.0]],
                           extrapol_type="FLAT")

spr2 = myModel.make_spring("My second spring", (t3, t4), init_Stiff_Coeff=1000.0)
```

The first example above creates an axial spring connected to triad (base Id) *t1* and *t2*, with a constant stiffness equal to 10.0 outside the interval [-0.1,0.1], and a V-shaped stiffness function in between with minimum value 0.1 at zero spring deflection. The second example creates a spring with a constant stiffness.

It is also possible to create several axial springs in one go by specifying a list of *(int, int)* tuples, and referring an already existing spring stiffness function using the *fn* keyword argument, as follows:

```
spr = myModel.make_spring("My springs", [(t1, t2), (t3, t4), (t5, t6)], fn=spr_func)
```

where *spr_func* is the base Id of an existing spring stiffness function.

To create axial dampers, there exists a method *make_damper* with a similar set of arguments as the *make_spring* method. That is, you can create one or several dampers with a piece-wise linear, or constant, damping coefficient, e.g.:

```
dmp = myModel.make_damper("My damper", (t3, t4), init_Damp_Coeff=100.0)
```

which creates an axial damper between the triads *t3* and *t4* with a constant damping coefficient of 100.0.

Refer to the documentation of *modeler.FedemModeler.make_spring()* and *modeler.FedemModeler.make_damper()* for an overview of all the keyword arguments that may be used for these two methods.

External loads

To create an external load on a triad, acting in the positive global Z-direction, with a time-dependent magnitude, you can use:

```
ldir = (0, 0, 1)  # Positive Z-direction
load = myModel.make_load("Sine", FmLoadType.FORCE, t3, ldir, "1E6*sin(5*x)")
print("Created an external load with base Id", load)
```

where `FmLoadType.TORQUE` as second parameter if a torque load is wanted instead. The last argument is a string with a math expression giving the load magnitude as function of time (here represented by the variable "x"), in this case a sinusoidal function with amplitude 1000000.0 and angular frequencey 5.0.

Alternatively, you may specify a general function as the load magnitude:

```
load = myModel.make_load("My load", FmLoadType.FORCE, t3, ldir, fn=funcId)
```

where *funcId* is the user Id of an existing general function, see *General functions* below.

Sensors

To create a sensor measuring the Z-displacement at a triad, use the following:

```
s1 = myModel.make_sensor("Displacement", t3, FmVar.POS, FmDof.TX)
print("Created sensor", s1)
```

where the third parameter can be any of `FmVar.POS`, `FmVar.LOCAL_VEL`, `FmVar.GLOBAL_VEL`, `FmVar.LOCAL_ACC`, `FmVar.GLOBAL_ACC`, `FmVar.LOCAL_FORCE` and `FmVar.GLOBAL_FORCE`, whereas the fourth parameter can be `FmDof.TX`, `FmDof.TY`, `FmDof.TZ`, `FmDof.RX`, `FmDof.RY` or `FmDof.RZ`.

To create a relative sensor between two triads, you can use:

```
s1 = myModel.make_sensor("Relative displacement", (t3, t2), FmVar.POS, FmDof.TX)
print("Created relative sensor", s2)
```

where the third parameter can be either `FmVar.POS`, `FmVar.VEL`, or `FmVar.ACC`.

Note: This method returns the user Id of the created sensor - not its base Id.

General functions

To create a general function of time, any of the following can be used:

```
# Polyline
f1 = myModel.make_function("My func 1", xy=[[0,0], [1,1], [2,3], [3,0.5]], extrapol_type=
↪"FLAT")
# Polyline from file
f2 = myModel.make_function("My func 2", filename="data.asc", ch_name="Force")
# Sine
f3 = myModel.make_function("My func 3", frequency=1.23, amplitude=2.5)
# Math expression
f4 = myModel.make_function("My func 4", expression="1.2+3.0*x^2")
# Constant function
f5 = myModel.make_function("My func 5", value=3.5)
```

(continues on next page)

(continued from previous page)

```
# Linear function
f6 = myModel.make_function("My func 6", slope=8.13)
# Ramp function
f7 = myModel.make_function("My func 7", start_val=0.3, start_ramp=1.46, slope=8.13)
# Limited Ramp function
f8 = myModel.make_function("My func 8", start_val=0.3, start_ramp=1.46, end_ramp=3.48,␣
↪slope=8.13)
# External function (no arguments)
f9 = myModel.make_function("My func 9")
```

What type of function to create is determined by the presence of keywords in the function argument list, as follows:

- *xy* : Polyline
- *filename* : Polyline from file
- *frequency* : Sine
- *expression* : Math expression
- *value* : Constant
- *slope* : Linear
- *start_ramp* : Ramp
- *end_ramp* : Limited Ramp

If no keywords (except the function name) are specified, an external function (whose value is assigned directly through the method *solver.FedemSolver.set_ext_func()*) will be created. The other function types may take other arguments in addition to those shown above, refer to the method documentation *modeler.FedemModeler.make_function()* for a full overview.

Note: This method returns the user Id of the created function - not its base Id.

See *Editing functions* below, for how to further modify the properties of created functions.

Strain rosettes

To create a strain rosette on an FE part, you can either specify the 3-4 node numbers to connect the strain rosette element to, or the coordinates of 3 or 4 spatial points if the node numbers are not known. It will then search for and use the closest node for each point:

```
# Using FE node numbers
r1 = myModel.make_strain_rosette("Gage A", part_id,
                                 nodes=[121, 122, 123],
                                 direction=(0, 1, 0))
# Using spatial point coordinates
r2 = myModel.make_strain_rosette("Gage B", part_id,
                                 pos=[(-1.702537, -0.5171, 1.702752),
                                      (-1.649538, -0.5171, 1.658139),
                                      (-1.630592, -0.5171, 1.73142)],
                                 direction=(0, 1, 0))
print("Created strain rosettes", [r1, r2], "on FE part", part_id)
```

You may also specify other keywords, see *modeler.FedemModeler.make_strain_rosette()* for the full documentation of this method.

User-defined elements

User-defined elements can be included in a Fedem model, if you specify the path to the plugin shared object library containing your element implementation when creating the `FedemModeler` object, e.g.:

```
myModel = FedemModeler("mymodel.fmm", True, "/usr/local/lib/libMyElmPlugin.so")
```

Please refer to the Fedem User's Guide for details on how the create a plugin library for user-defined elements. With this, you can create a string of three 2-noded elements connected to the four triads, *t1*, *t2*, *t3* and *t4*, using:

```
elms = myModel.make_udelm("My elements", [t1, t2, t3, t4], alpha1=0.03, alpha2=0.05)
print("Created user-defined elements with base Ids", elms)
```

Currently, only two-noded elements are supported in `fedempy`. The path to the plugin library will be stored in the created Fedem model file. Therefore, there is no need to specify it when *Solving Fedem models* through a `FmmSolver` object.

1.2.4 Modifying existing objects

The `FedemModeler` class also has some methods for modifying existing objects in the current model. They all have a signature like `edit_<object_type> (base_id, **kwarg)` and return the bool value True on success, otherwise False. The ***kwarg* argument represents a varying list of *keyword=value* pairs with the properties to assign to the object.

Editing triads

To change the position of a triad with base Id *tid*, use the following:

```
if not myModel.edit_triad(tid, Tx=1.0, Ty=0.2, Tz=3.4, Rx=30, Ry=10, Rz=5):
    print(" *** Failed to move Triad", tid)
```

The values specified are considered as offsets to the current position. Thus, you can leave out those coordinate directions which should not change. The rotational values (Rx, Ry, Rz) are Euler-ZYX angles (in degrees). That is, first the Rz rotation is applied, then Ry and finally Rx. If you need to rotate in a different order, that can be achieved by multiple *edit_triad* calls.

To adjust the DOF status of a triad, use something like:

```
if not myModel.edit_triad(tid, constraints={
                          "Tx" : FmDofStat.FIXED,
                          "Ty" : FmDofStat.PRESCRIBED,
                          "Tz" : FmDofStat.FREE_DYN,
                          "Rx" : FmDofStat.FIXED,
                          }):
    print(" *** Failed to constrain Triad", tid)
```

In this example, the triad is fixed in X-translation and X-axis rotation, prescribed in Y-translation, and fixed during initial equilibrium only in Z-translation. The last two DOFs (Ry and Rz) remain free.

If you need to constrain all DOFs in a triad, you can alternatively use the keyword "All" to shorten the statement, viz.:

```
if not myModel.edit_triad(tid, constraints={"All" : FmDofStat.FIXED}):
    print(" *** Failed to constrain Triad", tid)
```

This will then be equivalent to attaching the triad to ground.

To assign a constant load and/or prescribed motion to a triad, you can do:

```
if not myModel.edit_triad(tid, load={"Tz" : 1000.0, "Ry" : 123.4},
                          motion={"Ty" : 0.01}):
    print(" *** Failed to assign load/motion to Triad", tid)
```

This will assign constant loads in the 3rd and 5th local DOF, and a prescribed motion in the 2nd local DOF of the triad.

To assign a non-constant load or motion, just specify the user Id of the general function defining the load magnitude instead of the constant value. For instance, the following will assign a sinusoidal load in the *Tx*-DOF:

```
lid = myModel.make_function("My load", frequency=12.5, amplitude=1000.0)
if not myModel.edit_triad(tid, load={"Tx" : lid}):
    print(" *** Failed to assign load to Triad", tid)
```

The convention is that an integer value in the *load* and *motion* dictionary argument is assumed to be the user Id of an existing general function, whereas a real value is taken as the constant load/motion magnitude to be assigned.

Editing joints

For joints, you can edit the same properties as shown for triads above, except that the DOF status now can also be set to `FmDofStat.SPRING` or `FmDofStat.SPRING_DYN`. The latter is the same as the former, except that the DOF is kept fixed during initial equilibrium and (optionally) during eigenvalue analysis. For joint DOFs with either of these status codes, you can then assign constant *spring* and *damper* properties as well as stress-free length change function, as follows:

```
if not myModel.edit_joint(jid, spring={"Tx" : 1000.0, "Ry" : 1234.5},
                          damper={"Tx" : 100.0, "Ry" : 222.2},
                          length={"Tx" : len_id}):
    print(" *** Failed to assign spring/damper properties to joint", jid)
```

where *jid* is the base Id of the joint to modify and *len_id* is the user Id of an existing general function defining the stress-free length change of the *Tx*-DOF of the joint.

Editing parts

To change the position of a part with base Id *pid*, use the following:

```
if not myModel.edit_part(pid, Tx=1.0, Ty=0.2, Tz=3.4, Rx=30, Ry=10, Rz=5):
    print(" *** Failed to move Part", pid)
```

The interpretation of the keywords Tx,Ty,...,Rz is here similar as for triads, as explained above in *Editing triads*.

In addition, the structural damping and some reduction options can be changed using the edit_part method, viz.:

```
if not myModel.edit_part(pid, alpha1=0.001, alpha2=0.03,
                         component_modes=20, consistent_mass=True):
    print(" *** Failed to change properties for Part", pid)
```

Editing functions

The method *make_function* described above in *General functions* will make a general function of time, by default. To change the argument to other response variables, you can use the following:

```
if not myModel.edit_function(fid, t1, FmVar.POS, FmDof.TX):
    print(" *** Failed to change argument of Function", fid)
```

Except for the first argument, which here is the user Id of the general function to modify, this method takes the same set of arguments as the *make_sensor* method discussed in *Sensors* above.

1.2.5 Using tags as object identifiers

Each object in a Fedem model is assigned a unique base Id when it is created. This is a positive integer value which is returned by the *make*-methods, and can be used to refer to existing objects in other statements creating or modifying objects. However, it is often more convenient to use a user-defined *tag* to refer to an object, or a group of objects.

For this purpose, all *make*-methods accept the keyword *tag* for assigning a tag, e.g., for triads:

```
myModel.make_triad("My first Triad", (1.0, 0.0, 0.0), tag="T1")
myModel.make_triad("My second triad", (2.0, 0.0, 0.0), tag="T2")
myModel.make_triad("My third triad", (3.0, 1.5, 0.0), tag="T3")
```

Then, to change their properties, specify a string instead of the base Id:

```
if not myModel.edit_triad("T.", constraints={"Tz" : FmDofStat.FIXED}):
    print(" *** Failed to constrain Triads")
```

The string may contain a regular expression and will expand into all objects with a matching tag. The above example will therefore constrain the three triads “T1”, “T2” and “T3” in the Z-axis direction.

1.2.6 Example

A sample python script using `modeler` to generate a simple model is provided here. Another script that generates and solves the model which is used in the Car Suspension regression test is available here.

1.3 No-code modeling

This section gives a brief introduction on how you can use `fedempy` to create/edit Fedem models without the need of writing any python code yourself. Instead, the model definition is encoded in a YAML-formatted input file. This input file is parsed and converted into an equivalent Fedem model file (*.fmm*), through the use of the *`yaml_parser`* module.

1.3.1 YAML input file syntax

TODO: Descripe the file format here, listing the available keywords, etc.

1.3.2 Creating/editing a Fedem model with YAML input

When you have finished the YAML input file, e.g., "myModel.yaml", execute the following command to process it:

```
python -m fedempy.yaml_parser --input-file myModel.yaml --solve
```

The option *–solve* will execute the dynamics solver on the generated model.

1.3.3 Sample YAML input file

See here for a sample YAML input file, which will create the classical Loader model.

1.4 Solving Fedem models

This section gives a brief introduction on how you can use the `fedempy` package to execute the Fedem dynamics solver through python scripting. It relies on the *fmm_solver* module.

1.4.1 Prerequisites

Before you start, you need to set the environment variable **FEDEM_SOLVER** to point to the shared object library of the Fedem dynamics solver. This library is named *libfedem_solver_core.so* on Linux and *fedem_solver_core.dll* on Windows. You also need to set the environment variable **FEDEM_MDB** to point to the shared object library of the Fedem mechanism model database. This library is named *libFedemDB.so* on Linux (*FedemDB.dll* on Windows).

The methods for conducting dynamics simulation of a Fedem model are collected in the class `FmmSolver` which is a sub-class of `FedemSolver`. To access this class, start your python script by:

```
from fedempy.fmm_solver import FmmSolver
```

1.4.2 Starting a new simulation

To start the solver on an existing Fedem model "mymodel.fmm", use:

```
mySolver = FmmSolver("mymodel.fmm")
```

This will parse the model file, write out some key model size parameters, and generate the necessary input files for the dynamics solver in the appropriate directory structure. The solver will then start and halt after the setup stage, during which the mechanism model is initialized in memory. If an error condition occurs during this stage, an exception is raised and the script will abort.

You may also perform this in two separate operations, like:

```
mySolver = FmmSolver()

status = mySolver.start("mymodel.fmm", False, True)
if status < 0:
    print(" *** Something bad happened, check fedem_solver.res", status)
```

This way, the script will not abort if the solver fails, and you can take the proper action by checking the return value *status* instead. You may then also reuse the FmmSolver object in subsequent simulation within the same script.

If the model contains FE parts, they will be reduced if needed, if you specify *True* as the third parameter in the call to the `start` method. Otherwise, it is assumed the FE models have already been reduced and the solver will fail if the reduced matrix files are not found.

You also need to set the environment variable **FEDEM_REDUCER** to point to the shared object library of the Fedem FE part reducer for this to work. This library is named *libfedem_reducer_core.so* on Linux and *fedem_reducer_core.dll* on Windows.

1.4.3 Solving

There are many methods in `FmmSolver` via its parent class `FedemSolver` for running through the time stepping of the simulation. See the *fmm_solver.FmmSolver* and *solver.FedemSolver* documentation for a full overview of the available methods.

The easiest way to just step through the simulation would be as follows:

```
function_id = 1  # User Id of a function measuring the response variable
while mySolver.solve_next():
    res = mySolver.get_function(function_id)
    print("Time = ", mySolver.get_current_time(), " response =", res)
if mySolver.solver_done() == 0 and mySolver.ierr.value == 0:
    print("   * Time step loop finished, solver is closed")
else:
    print(" *** Solver failed, see fedem_solver.res", mySolver.ierr.value)
```

The while loop will continue as long there are more time steps defined, and no error condition occurred. In the above, the current simulation time and a response variable extracted using `get_function` is printed during the simulation. Here you can insert any other processing instead.

If you only need to run through the model without accessing any response variables, the above code block can be replaced by one line:

```
mySolver.solve_all("mymodel.fmm", True, True)
```

This will also reduce the FE parts, if needed, before the solver is started, see *fmm_solver.FmmSolver.solve_all()*.

1.4.4 Saving/closing current model

To close model after the simulation has finished, use:

```
mySolver.close_model(True)
```

This will first update the model file with references to the latest simulation results in the model database, and then release the model from core memory. You may also close the model without saving it (e.g., if the simulation failed) by specifying *False* as parameter.

If you used the `solve_all` method, you don't need to invoke `close_model` aftwards as this is included in the former method.

1.4.5 Solving models with external functions

In digital twin applications, the input loads and/or motions in a Fedem model are defined by means of External Functions. These are function objects that are assigned their value by an external process, for instance by using the method *solver.FedemSolver.set_ext_func()* within the time integration loop.

During the development of a digital twin model, it is often convenient to take the external function values from a file instead, such that the model can be solved directly from the Fedem Desktop, or by using the `solve_all` method. This can be done by specifying the following additional option for the dynamics solver:

```
-externalfuncfile <datafile.asc>
```

where <*datafile.asc*> is the name of a multi-column ASCII file containing the external function values as columns. The first column is assumed to contain the time of each step and is not used. The columns need to be labeled with the *tag* of the external function objects, by specifying the following as a comment line before the numerical data:

```
#DESCRIPTION <func1-tag> <func2-tag> <func3-tag> ...
```

where each <*func#-tag*> identifies the respective column. The order of the columns in the file is arbitrary, and it may also contain some columns that are not used, since a search will be performed for each function.

For instance, assume you created a model using *modeler.FedemModeler* and included the following external functions:

```
myModel.make_function("Input function A", tag="FuncA");
myModel.make_function("Input function B", tag="FuncB");
```

Then a *datafile.asc* containing the following will work:

```
#DESCRIPTION Func_A unused Func_B
0.0          1.2345  0.123 2.3456
0.1          2.2345  0.234 3.3456
...
```

Note: It is assumed that the time steps of the simulation match those of the specified file, and such that a new line of data is read and the function values are updated prior to each time step of the simulation. No interpolation is performed if the times do not match. If the file contains fewer data lines than the total number of time steps in the simulation, all external functions will remain constant and equal to the values of the last line for the remaining steps throughout the simulation.

CHAPTER

TWO

FEDEMPY MODULES

2.1 divergence module

Convenience module for automatic simulation restart of diverging models.

divergence.**jump_state**(*lib_dir*)

Checks whether the current state should be jumped over or not.

divergence.**ramp_dataframe**(*dfr*, *start_pos=0*)

Creating a s-shape ramp, starting with 0 and ending with 1 Second derivative at start/end will be zero Multiplying each column in dataframe with ramp-array

divergence.**restart**(*solver*, *df*, *state*, *lib_dir*, *out_id*, *c_opts*, *x_times=None*)

Restarts the simulation over a time window in case of divergence issues. When a divergent solution is detected, one (or more) of the following approaches may be attempted:

- *TOL*: Increasing tolerance(s) for convergence checks (tolDispNorm, tolVecNorm, tolEnerSum, maxit).
- *STATE_DYNAMIC*: Use dynamic equilibrium at restart. Restart can be performed at a specified number of steps before the divergent step, optionally with ramp-scaling activated (default activated).
- *STATE_STATIC*: Use quasi-static equilibrium for a specified number of steps at restart. Restart can be performed at a specified number of steps before the divergent step, optionally with ramp-scaling activated (default activated) done n-steps backwards.
- *NO_STATE_WITH_RAMP*: Restart of specified number of steps before the diverged step. The loads are ramped up from zero and the state is not used.
- *JUMP_OVER*: Skip the rest if the current window and restart from the next, with the loads ramped up from zero.
- *NO_CONV*: No convergence is obtained.
- *FAILURE*: Other solver failure during re-initialization.

Parameters

solver
[FedemSolver] Fedem dynamics solver instance

df
[DataFrame] Input function values

state
[list of float] State vector to restart simulation from

lib_dir
[str] Path to the input/output files

out_id
[list of int] List of user Ids identifying the output sensors in the model

c_opts
[dictionary] Settings for the Fedem solver

x_times
[list of float, default=None] Time list linked to the input

Returns

list of float
Output sensor values for each time step

str
How the simulation was actually restarted (or not)

2.2 enums module

This module contains some enum definitions for the `fedempy` package. They are python equivalents to corresponding enum definitions in the C++ code of the *FedemDB* shared object library (see the *fedem_mdb* repository). The numerical value of each enum value in this module should therefore not be changed without a similar change in the C++ equivalent, to preserve consistency.

`class enums.FmDof(`*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*`)`

Bases: `Enum`

This enumerator identifies the local DOF components in an object. The values corresponds to the `FmIsMeasuredBase::SensorDof` enum values (see the *fedem_mdb* repository).

GAGE_1 = 17

GAGE_2 = 18

GAGE_3 = 19

LENGTH = 6

MAX_PR = 13

MIN_PR = 14

RX = 3

RY = 4

RZ = 5

SA_MAX = 15

TX = 0

TY = 1

TZ = 2

VMISES = 16

class enums.**FmDofStat**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

This enumerator identifies the available DOF constraint types. The values corresponds to the FmHasDOFsBase::DOFStatus enum values (see the *fedem_mdb* repository).

FIXED = 1

FREE = 0

FREE_DYN = 3

PRESCRIBED = 2

SPRING = 4

SPRING_DYN = 5

class enums.**FmLoadType**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

This enumerator identifies the two external load types a Fedem model. The values corresponds to the enum values in the C++ class FmLoad. (see the *fedem_mdb* repository).

FORCE = 0

TORQUE = 1

class enums.**FmType**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

This enumerator identifies the various object types a Fedem mechanism model may consist of. They are mapped onto a corresponding type ID value of the C++ classes in the *FedemDB.C* source file (see the *fedem_mdb* repository).

ALL = 0

AXIAL_DAMPER = 18

AXIAL_SPRING = 17

BALL_JOINT = 9

BEAM = 2

BEAM_PROP = 4

CAM_JOINT = 13

CYLINDRIC_JOINT = 12

FEPART = 3

FREE_JOINT = 10

FUNCTION = 15

JOINT = 6

LOAD = 14

MAT_PROP = 5

PRISMATIC_JOINT = 11

REVOLUTE_JOINT = 8

RIGID_JOINT = 7

SENSOR = 16

STRAIN_ROSETTE = 19

TRIAD = 1

class enums.**FmVar**(*value*, *names=None*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

This enumerator identifies the result quantities that may be measured. The values corresponds to the FmIsMeasuredBase::SensorEntity enum values (see the *fedem_mdb* repository).

ACC = 7

DAMP_ANG = 12

DAMP_FORCE = 14

DAMP_VEL = 13

DEFLECTION = 16

DISTANCE = 5

FORCE = 17

GLOBAL_ACC = 4

GLOBAL_FORCE = 19

GLOBAL_VEL = 2

LENGTH = 15

LOCAL_ACC = 3

LOCAL_FORCE = 18

LOCAL_VEL = 1

POS = 0

REL_POS = 8

SPR_ANG = 9

SPR_DEFL = 10

SPR_FORCE = 11

STRAIN = 24

STRESS = 25

VEL = 6

2.3 exporter module

Python wrapper for the native VTFX exporter.

class exporter.**Exporter**(*fe_parts*, *vis_parts*, *lib_path*, *vtfx_path=None*)

Bases: object

This class provides functionality for exporting fedem animations to Ceetron CUG format.

Parameters

fe_parts
[dict] Dictionary of finite element parts to include in visualization

vis_parts
[dict] Dictionary of visualization/vrml parts to include in visualization

lib_path
[str] Absolute path to the shared object library vtfxExporter.so

vtfx_path
[str, default=None] Absolute path to vtfx-file, use a temporary file if None

Methods

do_step:	Executes a step of visualization export with the provided input data
clean:	Converts temporary generated vtfx-file to CUG database and cleans up

clean(*time_step*, *lib_dir*, *out_dir=None*, *cug_path='/usr/bin/CugComposer'*)

Clears all data about included FE-parts and frs-files, and removes all transformation, stress and deformation results. Closes the vtfx-file and converts it to a CUG database.

Parameters

time_step
[double] Time step to use for animation speed

lib_dir
[str] Absolute path to app location

out_dir
[str, default=None] Absolute path for output folder of the CUG database If None, CUG data is not generated and the vtfx-file is retained

cug_path
[str, default="/usr/bin/CugComposer"] Absolute path to Ceetron CUG composer executable

do_step(*time*, *transformation_in*, *deformation_in*, *stress_in*)

Execute a step of visualization export with the provided input data.

Parameters

time
[float] Simulation time

transformation_in
[list of c_double] Array of transformation data from the fedem solver

deformation_in
[dict] Arrays of part deformation data from the fedem solver

stress_in
[dict] Arrays of part stress data from the fedem solver

exception exporter.ExporterException(*rc=0*)

Bases: Exception

General exception-type for exporter exceptions.

Parameters

rc
[int, default=0] Return code

2.4 fmm module

Python wrapper for the Fedem Mechanism Model database library. Used for convenience in order to hide native type convertions.

class fmm.FedemModel(*lib_path*, *plugin1=None*, *plugin2=None*)

Bases: object

This class wraps the Fedem model file handling functionality.

Parameters

lib_path
[str] Absolute path to the Fedem model database shared object library

plugin1
[str, default=None] Absolute path to user-defined element plugin (optional)

plugin2
[str, default=None] Absolute path to user-defined function plugin (optional)

Methods

fm_open:	Opens the specified Fedem model file
fm_save:	Saves current model with updated results to file
fm_new:	Initializes an empty model
fm_close:	Closes current model
fm_count:	Counts mechanism objects of specified type
fm_get_objects:	Returns a list of base Ids for all objects of specified type
fm_tag_object:	Tags the specified object(s) in the model
fm_get_pos:	Returns the global position of the specified object
fm_get_node:	Returns the FE node number matching a spatial point
fm_sync_part:	Syncronizes an FE part with content on disk
fm_write_reducer:	Writes reducer input files for an FE part
fm_write_solver:	Writes solver input files
fm_solver_setup:	Defines some basis solver setup parameters
fm_solver_tol:	Defines the solver convergence tolerances
def fm_get_func_tag:	Returns the tag of an indexed external function

`fm_close`(*remove_singletons=False*)

This method closes the currently open Fedem model to clean up memory.

Parameters

remove_singletons
[bool, default=False] If True, heap-allocated singelton objects are also released

Returns

bool
Always True

`fm_count`(*object_type=FmType.ALL*)

This method returns the number of objects of the given type.

Parameters

object_type
[FmType, default=ALL] Enum value identifying which object type to count the instances of

Returns

int
Number of instances of the indicated object type

`fm_get_func_tag`(*channel*)

This method returns the tag of an indexed external function.

Parameters

channel
[int] Channel index of the external function

Returns

str
The tag of the function, None if channel is out of range

fm_get_node(*part_id*, *pos*)

This method returns the FE node number matching a spatial point.

Parameters

part_id
[int] Base Id or tag of the FE part that the node belongs to

pos
[(float, float, float)] X-, Y-, and Z-coordinate of the node

Returns

int
Node Id

fm_get_objects(*object_type=FmType.ALL*, *tag=None*)

This method returns the base Id of all objects of the given type, and/or those with the specified tag.

Parameters

object_type
[FmType, default=ALL] Enum value identifying which object type to return base Id of

tag
[str or list of str, default=None] Return objects having this tag only, unless None

Returns

list
Base Id list of all instances in the model of the specified type

fm_get_pos(*object_id*)

This method returns the global position of the specified object.

Parameters

object_id
[int or str] Base Id or tag of the object to return the position for

Returns

(float, float, float)
Global X-, Y-, and Z-coordinate

fm_new(*fname=None*)

This method initializes an empty Fedem model.

Parameters

fname
[str, default=None] Absolute path to the fmm-file to write to on next save. If not given, the name "untitled.fmm" will be used.

Returns

bool
Always True

fm_open(*fname*)

This method opens the specified Fedem model file and reads its content into the internal datastructure.

Parameters

fname
[str] Absolute path to the fmm-file to read

Returns

bool
True on success, otherwise False

`fm_save`(*fname=None*)

This method saves the currently open Fedem model such that the model file is updated to reflect the current results data found on disk.

Parameters

fname
[str, default=None] Absolute path to the fmm-file to write to. If not given, the current model file as specified by the last call to fm_open() or fm_new() is overwritten.

Returns

bool
True on success, otherwise False

`fm_solver_setup`(*t_start=0.0*, *t_end=1.0*, *t_inc=0.01*, *t_quasi=0.0*, *n_modes=0*, *e_inc=0.0*, *add_opt=None*)

This method (re)defines some basis solver setup parameters.

Parameters

t_start
[float, default=0] Start time

t_end
[float, default=1] Stop time

t_inc
[float, default=0.01] Time step size

t_quasi
[float, default=0] Stop time for quasi-static simulation. If equal to t_start, also perform initial equilibrium analysis before starting the time stepping.

n_modes
[int, default=0] If non-zero, perform eigenvalue analysis during the simulation, and calculate this number of modes each time

e_inc
[float, default=0] Time between each eigenvalue analysis

add_opt
[str, default=None] Additional solver options

`fm_solver_tol`(*tol_ene=-1.0*, *tol_dis=-1.0*, *tol_vel=-1.0*, *tol_res=-1.0*)

This method (re)defines the solver convergence tolerances.

Parameters

tol_ene
[float, default=-1.0] Energy norm tolerance

tol_dis
[float, default=-1.0] Displacement norm tolerance

tol_vel
[float, default=-1.0] Velocity norm tolerance

tol_res
[float, default=-1.0] Residual force norm tolerance

`fm_sync_part`(*base_id*)

This method syncronizes the currently open Fedem model with contents on disk for the specified FE part. It is typically used after the reduction process for an FE part is finished.

Parameters

base_id
[int] Base Id of the FE part to syncronize the model and disk content for

Returns

bool
True on success, otherwise (e.g., if base_id is invalid) False

`fm_tag_object`(*base_id*, *tag*)

This method tags the specified object(s) identified by the base Id.

Parameters

base_id
[int or list of int] Base Id of the object(s) to tag

tag
[str] The tag to assign

Returns

int
Number of tagged objects

`fm_write_reducer`(*base_id*)

This method writes reducer input files for the specified FE part, unless that part is already reduced.

Parameters

base_id
[int] Base Id of the FE part to generate reducer input files for

Returns

str
Absolute path to the working directory of the reduction process

`fm_write_solver`(*keep_old_res=False*, *ude=None*, *udf=None*)

This method writes solver input files for the currently loaded model.

Parameters

keep_old_res
[bool, default=False] Option to not overwrite any existing res-file in the RDB directory

ude
[str, default=None] Absolute path to user-defined element plugin (optional)

udf
[str, default=None] Absolute path to user-defined function plugin (optional)

Returns

str
Absolute path to the working directory of the solver process

2.5 fmm_solver module

Python wrapper for the native fedem dynamics solver which operates directly on fedem mechanism model files (*.*fmm*).

This module relies on the following environment variables:

FEDEM_REDUCER = Full path to the Fedem reducer shared object library
FEDEM_SOLVER = Full path to the Fedem solver shared object library
FEDEM_MDB = Full path to the Fedem model database shared object library

The first variable needs to be set only if FE model reduction is to be performed. The other two variables are mandatory.

This module can also be launched directly, to run a specified model, using the syntax

```
python -m fedempy.fmm_solver -f mymodel.fmm [--reduce-fem] [--save-model]
```

This will then invoke the method *fmm_solver.FmmSolver.solve_all()* on the specified model-file (*mymodel.fmm*). If you already have a directory populated with Fedem solver input files, you can run the solver directly on those files by launching this module without arguments, i.e.:

```
python -m fedempy.fmm_solver
```

class fmm_solver.**FmmInverse**(*model_file*, *config*, *use_internal_state=False*, *keep_res=False*)

Bases: *FmmSolver*, InverseSolver

This class augments FmmSolver with inverse solution capabilities.

Parameters

model_file
[str] Absolute path to the fedem model file to start the simulation on

config
[dictionary] Inverse solver configuration

use_internal_state
[bool, default=False] If True, internal state arrays are allocated (for microbatching)

keep_res
[bool, default=False] Option to not overwrite any existing res-file in the RDB directory

Methods

add_rhs_vector(r_vec)	Utility updating the content of the system right-hand-side vector.
check_times(xtimes[, use_times])	Utility checking that a given time series starts with the current solver time.
close_model([save, remove_singletons])	Closes the currently open model.
compute_int_forces_from_displ(disp, ids)	This method computes beam section forces in triads for the given displacement field.
compute_rel_dist_from_displ(disp, ids)	This method computes the relative distance at sensors for the given displacement field.
compute_spring_var_from_displ(disp, ids)	This method computes one of the spring variables (length, deflection, force) at sensors for the given displacement field.
compute_strains_from_displ(disp, gauge_ids)	This method computes the strain tensor at gauges for the given displacement field, or from current state if no displacements provided.
convert_rev_joint_force(data)	Modify the data set for spring force input for revolute joints with defined spring forces Conversation from force to lenght x=F/k (const.
finish_step()	This method completes current time (or load) step, by iterating the linearized equation system until convergence is achieved.
get_current_strains(gauge_ids)	This method computes the strain tensor at gauges from current state.
get_current_time()	Utility returning the current physical time of the simulation.
get_damping_matrix()	Utility returning current content of the system damping matrix.
get_element_stiffness_matrix(bid)	Utility returning the initial content of an element stiffness matrix.
get_equations(bid)	Utility returning the equation numbers for the DOFs of the object with the specified base Id (bid).
get_external_force_vector()	Utility returning current content of the external force vector.
get_function([uid, tag, arg])	Utility evaluating a general function in the model, identified by the specified user Id *uid* or *tag*, and with the function argument *arg*.
get_function_ids(tags)	Utility returning a list of user Ids of tagged general functions.
get_functions(uids)	Utility evaluating a list of general functions for current state.
get_gauge_size()	Utility returning the required size of the initial strain gauge array which is used when restarting a simulation from an in-core array.
get_joint_spring_stiffness(bid)	Get joint spring stiffness coefficient(s).
get_mass_matrix()	Utility returning current content of the system mass matrix.
get_newton_matrix()	Utility returning current content of the system Newton matrix.
get_next_time()	Utility returning the physical time of the next step of the simulation.

continues on next page

Table 1 – continued from previous page

get_part_deformation_state_size(base_id)	Utility returning the required length of the state vector which stores deformation data for the FE Part with the given base Id.
get_part_stress_state_size(base_id)	Utility returning the required length of the state vector which stores von Mises stresses for the FE Part with the given base Id.
get_rhs_vector()	Utility returning current content of the system right-hand-side vector.
get_state_size()	Utility returning the required length of the state vector which is used when restarting a simulation from an in-core array.
get_stiffness_matrix()	Utility returning current content of the system stiffness matrix.
get_system_dofs()	Utiloty returning the total number of DOFs of the system.
get_system_size()	Utility returning the dimension (number of equations) of the system.
get_transformation_state_size()	Utility returning the required length of the vector which stores the transformation matrices (rotation and translation) for Triads, Parts and Beams.
restart_from_state(state_data[, write_to_rdb])	This method re-initializes the mechanism objects with data from the provided state array, such that the simulation can continue from there.
run_all(options)	This method runs the dynamics solver with given command-line *options*, without any user intervention.
run_inverse(inp_data, out_def)	Collector for different inverse methods.
run_inverse_dyn(inp_data, out_def)	Inverse solution driver (dynamic case).
run_inverse_fedem(inp_data, out_def)	This method uses fedem's inverse solution in fortran
save_gauges()	This method stores initial gauge strains in the self.gauge_data array.
save_part_state(base_id, def_state, str_state)	This method stores current deformation- and stress states for the specified FE Part in the provided core arrays.
save_state()	This method stores current solver state in the self.state_data array.
save_transformation_state(state_data[, ndat])	This method stores current transformation state for Triads, Parts and Beams in the provided core array.
set_ext_func(func_id[, value])	This method may be used prior to the solve_next call, to assign a sensor value from a physical twin to the specified actuator or load in the model, identified by the argument func_id (external function Id).
set_input(func_tag[, value])	This method may be used prior to the solve_next call, to assign a sensor value from a physical twin to the specified actuator or load in the model, identified by the argument func_tag.
set_rhs_vector(r_vec)	Utility replacing current content of the system right-hand-side vector.
solve_all(model_file[, save_model, reduce_fem])	Starts and runs through a simulation on the specified model file.
solve_inverse(x_val, x_def, g_def[, out_def])	This method solves the inverse problem at current time/load step, assuming small deformations only (linear response).

continues on next page

Table 1 – continued from previous page

solve_iteration()	This method solves the current linearized equation system and updates all state variables.
solve_modes(n_modes[, dof_order, use_lapack])	This method solves the eigenvalue problem at current time step, and returns the computed eigenvalues and associated eigenvectors.
solve_next([inp, inp_def, out_def, time_next])	This method advances the solution one time/load step forward.
solve_window(n_step[, inputs, f_out, xtimes])	This method solves the problem for a time/load step window, with given values for the external functions, and extraction of results from another set of general functions in the model.
solver_close()	This method needs to be used if solver_done() was invoked with its *remove_singletons* argument set to False.
solver_done([remove_singletons])	This method should be used when the time/load step loop is finished.
solver_init(options[, fsi, state_data, ...])	This method processes the input and sets up necessary data structures prior to the time integration loop.
start(model_file[, keep_old_res, ...])	Starts a simulation on the specified model file.
start_step()	This method starts a new time (or load) step, by calculating the predicted response, the coefficient matrix and right-hand-side vector of the first nonlinear iteration.

class fmm_solver.**FmmSolver**(*model_file=None*, *use_internal_state=False*, *keep_old_res=False*)

Bases: FedemSolver

This subclass of FedemSolver adds the possibility to start a simulation on a specified fedem model file.

Parameters

model_file
[str, default=None] Absolute path to the fedem model file to start the simulation on

use_internal_state
[bool, default=False] If True, internal state arrays are allocated (for microbatching)

keep_old_res
[bool, default=False] Option to not overwrite any existing res-file in the RDB directory

Methods

start:	Starts a simulation on the specified model file
close_model:	Closes the currently open model
solve_all:	Starts and runs through a simulation on the specified model file
set_input:	Assigns sensor value to a tagged external function

close_model(*save=False*, *remove_singletons=False*)

Closes the currently open model.

Parameters

save
[bool, default=False] If True, the model file is updated based on current result files

remove_singletons
[bool, default=False] If True, heap-allocated singelton objects are also released

Returns

bool
True on success, otherwise False

`set_input`(*func_tag*, *value=None*)

This method may be used prior to the solve_next call, to assign a sensor value from a physical twin to the specified actuator or load in the model, identified by the argument func_tag.

Parameters

func_tag
[str] Tag of the external function to be assigned new value

value
[float, default=None] The value to be assigned

Returns

bool
Always True, unless the specified function is not found

`solve_all`(*model_file*, *save_model=True*, *reduce_fem=False*)

Starts and runs through a simulation on the specified model file.

Parameters

model_file
[str] Absolute path of the Fedem model file to run the solver on

save_model
[bool, default=True] If True, save the model file with new results when solver finished

reduce_fem
[bool, default=False] If True, and the model contains FE parts, they will be reduced before the solver is started, unless the reduced matrix files already exist

Returns

int
Zero on success, otherwise negative

`start`(*model_file*, *keep_old_res=False*, *close_model=False*, *reduce_fem=False*, *state_data=None*, *gauge_data=None*, *extf_input=None*, *time_start=None*)

Starts a simulation on the specified model file.

Parameters

model_file
[str] Absolute path of the Fedem model file to run the solver on

keep_old_res
[bool, default=False] Option to not overwrite any existing res-file in the RDB directory

close_model
[bool, default=False] If True, release the model from memory before solver start

reduce_fem
[bool, default=False] If True, and the model contains FE parts, they will be reduced before the solver is started, unless the reduced matrix files already exist

state_data
[list of float, default=None] Complete state vector to restart simulation from

gauge_data
[list of float, default=None] Initial strain gauge values for restart

extf_input
[list of float, default=None] Initial external function values, for initial equilibrium iterations

time_start
[float, default=None] Optional start time of simulation, override setting in model file

Returns

int
Zero on success, otherwise negative

2.6 inverse module

Python implementation of inverse solution methods with Fedem.

`class inverse.FedemRun(`*wrkdir*, *config*`)`

Bases: `FedemSolver`, *`InverseSolver`*

This class augments FedemSolver with inverse solution capabilities.

Parameters

wrkdir
[str] Current working directory for the fedem dynamics solver

config
[dictionary] Content of yaml input file

Methods

`add_rhs_vector`(r_vec)	Utility updating the content of the system right-hand-side vector.
`check_times`(xtimes[, use_times])	Utility checking that a given time series starts with the current solver time.
`compute_int_forces_from_displ`(disp, ids)	This method computes beam section forces in triads for the given displacement field.
`compute_rel_dist_from_displ`(disp, ids)	This method computes the relative distance at sensors for the given displacement field.
`compute_spring_var_from_displ`(disp, ids)	This method computes one of the spring variables (length, deflection, force) at sensors for the given displacement field.
`compute_strains_from_displ`(disp, gauge_ids)	This method computes the strain tensor at gauges for the given displacement field, or from current state if no displacements provided.

continues on next page

Table 2 – continued from previous page

convert_rev_joint_force(data)	Modify the data set for spring force input for revolute joints with defined spring forces Conversation from force to lenght x=F/k (const.
finish_step()	This method completes current time (or load) step, by iterating the linearized equation system until convergence is achieved.
get_current_strains(gauge_ids)	This method computes the strain tensor at gauges from current state.
get_current_time()	Utility returning the current physical time of the simulation.
get_damping_matrix()	Utility returning current content of the system damping matrix.
get_element_stiffness_matrix(bid)	Utility returning the initial content of an element stiffness matrix.
get_equations(bid)	Utility returning the equation numbers for the DOFs of the object with the specified base Id (bid).
get_external_force_vector()	Utility returning current content of the external force vector.
get_function([uid, tag, arg])	Utility evaluating a general function in the model, identified by the specified user Id *uid* or *tag*, and with the function argument *arg*.
get_function_ids(tags)	Utility returning a list of user Ids of tagged general functions.
get_functions(uids)	Utility evaluating a list of general functions for current state.
get_gauge_size()	Utility returning the required size of the initial strain gauge array which is used when restarting a simulation from an in-core array.
get_joint_spring_stiffness(bid)	Get joint spring stiffness coefficient(s).
get_mass_matrix()	Utility returning current content of the system mass matrix.
get_newton_matrix()	Utility returning current content of the system Newton matrix.
get_next_time()	Utility returning the physical time of the next step of the simulation.
get_part_deformation_state_size(base_id)	Utility returning the required length of the state vector which stores deformation data for the FE Part with the given base Id.
get_part_stress_state_size(base_id)	Utility returning the required length of the state vector which stores von Mises stresses for the FE Part with the given base Id.
get_rhs_vector()	Utility returning current content of the system right-hand-side vector.
get_state_size()	Utility returning the required length of the state vector which is used when restarting a simulation from an in-core array.
get_stiffness_matrix()	Utility returning current content of the system stiffness matrix.
get_system_dofs()	Utiloty returning the total number of DOFs of the system.
get_system_size()	Utility returning the dimension (number of equations) of the system.

continues on next page

Table 2 – continued from previous page

get_transformation_state_size()	Utility returning the required length of the vector which stores the transformation matrices (rotation and translation) for Triads, Parts and Beams.
restart_from_state(state_data[, write_to_rdb])	This method re-initializes the mechanism objects with data from the provided state array, such that the simulation can continue from there.
run_all(options)	This method runs the dynamics solver with given command-line *options*, without any user intervention.
run_inverse(inp_data, out_def)	Collector for different inverse methods.
run_inverse_dyn(inp_data, out_def)	Inverse solution driver (dynamic case).
run_inverse_fedem(inp_data, out_def)	This method uses fedem's inverse solution in fortran
save_gauges()	This method stores initial gauge strains in the self.gauge_data array.
save_part_state(base_id, def_state, str_state)	This method stores current deformation- and stress states for the specified FE Part in the provided core arrays.
save_state()	This method stores current solver state in the self.state_data array.
save_transformation_state(state_data[, ndat])	This method stores current transformation state for Triads, Parts and Beams in the provided core array.
set_ext_func(func_id[, value])	This method may be used prior to the solve_next call, to assign a sensor value from a physical twin to the specified actuator or load in the model, identified by the argument func_id (external function Id).
set_rhs_vector(r_vec)	Utility replacing current content of the system right-hand-side vector.
solve_inverse(x_val, x_def, g_def[, out_def])	This method solves the inverse problem at current time/load step, assuming small deformations only (linear response).
solve_iteration()	This method solves the current linearized equation system and updates all state variables.
solve_modes(n_modes[, dof_order, use_lapack])	This method solves the eigenvalue problem at current time step, and returns the computed eigenvalues and associated eigenvectors.
solve_next([inp, inp_def, out_def, time_next])	This method advances the solution one time/load step forward.
solve_window(n_step[, inputs, f_out, xtimes])	This method solves the problem for a time/load step window, with given values for the external functions, and extraction of results from another set of general functions in the model.
solver_close()	This method needs to be used if solver_done() was invoked with its *remove_singletons* argument set to False.
solver_done([remove_singletons])	This method should be used when the time/load step loop is finished.
solver_init(options[, fsi, state_data, ...])	This method processes the input and sets up necessary data structures prior to the time integration loop.
start_step()	This method starts a new time (or load) step, by calculating the predicted response, the coefficient matrix and right-hand-side vector of the first nonlinear iteration.

exception inverse.**InverseException**(*method_name*, *ierr=None*)

Bases: FedemException

General exception type for inverse solver exceptions. Used to generalize error messages from the FedemSolver methods.

Parameters

method_name
[str] Name of the method that detected an error

ierr
[int, default=None] Error flag value that is embedded into the error message

class inverse.**InverseSolver**(*solver*, *config*)

Bases: object

This class handles the inverse solution through proper methods. It accesses the Fedem model through the provided FedemSolver instance.

Parameters

solver
[FedemSolver] The Fedem dynamics solver instance

config
[dictionary] Inverse solver configuration

Methods

run_inverse_dyn:	Performs the inverse solution (dynamic case)
run_inverse_fedem:	Performs the inverse static solution using the internal inverse solver
run_inverse:	Performs the inverse solution (static case)

convert_rev_joint_force(*data*)

Modify the data set for spring force input for revolute joints with defined spring forces Conversation from force to lenght x=F/k (const. stiffness assumption)

Parameters

data
[list of float] Input function/data values

Returns

list of int
revolute joint ID's and their modified input data

run_inverse(*inp_data*, *out_def*)

Collector for different inverse methods.

Parameters

inp_data
[list of float] Input function values

out_def
[list of int] User Ids of the functions to evaluate the response for

Returns

list of float
Evaluated response variables

`run_inverse_dyn`(*inp_data*, *out_def*)

Inverse solution driver (dynamic case).

Parameters

inp_data
[list of float] Input function values

out_def
[list of int] User Ids of the functions to evaluate the response for

Returns

list of float
Evaluated response variables

`run_inverse_fedem`(*inp_data*, *out_def*)

This method uses fedem's inverse solution in fortran

Parameters

inp_data
[list of float] Input function values

out_def
[list of int] User Ids of the functions to evaluate the response for

Returns

list of float
Evaluated response variables

2.7 modeler module

Python wrapper for the native fedem modeler.

This module provides functionality for creating new Fedem models, and for modifying existing ones.

`class modeler.FedemModeler`(*model_file=None*, *force_new=False*, *plugins=None*)

Bases: `FedemModel`

This subclass of *fmm.FedemModel* adds some basic modeling methods.

Parameters

model_file
[str, default=None] Absolute path to the fedem model file to open

force_new
[bool, default=False] If True, any existing model will be overwritten on save

plugins
[str or list of str, default=None] Plugin libraries with user-defined elements and functions

Methods

open:	Opens the specified model file
save:	Saves current model into the specified model file
close:	Closes the currently open model
make_triad:	Creates a triad at specified location or node
make_beam:	Creates a string of beam elements
make_beam_section:	Creates a beam cross section property object
make_beam_material:	Creates a material property object
make_spring:	Creates a spring element
make_damper:	Creates a damper element
make_joint:	Creates a joint object
make_load:	Creates an external load object
make_function:	Creates a general function object
make_sensor:	Creates a sensor object
make_fe_part:	Creates an FE part
make_strain_rosette:	Creates a strain rosette on an FE part
make_udelm:	Creates a string of user-defined elements
edit_triad:	Modifies an existing triad
edit_part:	Modifies an existing FE part
edit_joint:	Modifies an existing joint
edit_function:	Modifies an existing function
solver_setup:	Setting dynamics solver parameters

`close`(*save=False*, *remove_singletons=False*)

Closes the currently open model.

Parameters

save
[bool, default=False] If True, the model file is updated with the current model

remove_singletons
[bool, default=False] If True, heap-allocated singelton objects are also released

Returns

bool
True on success, otherwise False

`edit_function`(*func_id*, *obj*, *var=None*, *dof=None*)

Modifies an existing function by changing its argument.

Parameters

func_id
[int] User Id (or base Id, if negative) of the function to modify

obj
[int or str or (int, int) or (str, str)] Base Id or tag of object(s) producing the response quantity to use

var
[FmVar, default=None=0] Response quantity to use as function argument

dof
[FmDof, default=None=0] Which component of *var* to be used if a multi-DOF quantity

Returns

bool
True if the input is valid, otherwise False

`edit_joint`(*obj_id*, ***kwargs*)

Modifies an existing joint.

Parameters

obj_id
[int or str or list of int or list of str] Base Id or tag of the joint(s) to modify

kwargs
[dict] Keyword arguments containing the new joint attributes to assign. Currently, the following keywords are recognized:

- Tx, Ty, Tz : Translation offset in global X, Y, and Z-direction
- Rx, Ry, Rz : Euler angles (in degrees) for rotating the joint
- tra_ref : Base Id of object used as translation reference
- rot_ref : Base Id of object used as rotation reference
- constraints : Dictionary with keywords for constraing DOFs, i.e., `constraints={"Tx" : <value>, ..., "Rz" : <value>}`, where `<value>` can be any of the enum values *enums.FmDofStat*. Only the DOFs that should be changed need to be specified. You can also use "All" as key which implies all DOFs in the joint.
- init_vel : Dictionary with keywords specifying initial velocities, i.e., `init_vel={"Tx" : <u01>, ..., "Rz" : <u06>}`, where `<u0i>` is the initial velocity to be assigned local dof *i*.
- load : Dictionary with keywords specifying constant DOF loads, i.e., `load={"Tx" : <f1>, ..., "Rz" : <f6>}`, where `<fi>` is the user Id of a general function defining the load in local dof *i* if the type is int, otherwise it is taken as the constant load to be assigned local dof *i*.
- motion : Dictionary with keywords specifying prescribed motions, i.e., `motion={"Tx" : <u1>, ..., "Rz" : <u6>}`, where `<ui>` is the user Id of a general function defining the motion in local dof *i* if the type is int, otherwise it is taken as the constant motion to be prescribed in local dof *i*.
- spring : Dictionary with keywords specifying spring stiffnesses, i.e., `spring={"Tx" : <k1>, ..., "Rz" : <k6>}`, where `<ki>` is the constant stiffness to be assigned local dof *i*.
- damper : Dictionary with keywords specifying damping coefficients, i.e., `load={"Tx" : <c1>, ..., "Rz" : <c6>}`, where `<ci>` is the constant damping to be assigned local dof *i*.
- length : Dictionary with keywords specifying stress-free lengths, i.e., `length={"Tx" : <l1>, ..., "Rz" : <l6>}`, where `<li>` is the user Id of a general function defining the stress-free length in local dof *i*.

Returns

bool
True if the input is valid, otherwise False

`edit_part`(*obj_id*, ***kwargs*)

Modifies an existing FE part.

Parameters

obj_id
[int or str or list of int or list of str] Base Id or tag of the part(s) to modify

kwargs
[dict] Keyword arguments containing the new part attributes to assign. Currently, the following keywords are recognized:

- Tx, Ty, Tz : Translation offset in global X, Y, and Z-direction
- Rx, Ry, Rz : Euler angles (in degrees) for rotating the Part
- tra_ref : Base Id of object used as translation reference
- rot_ref : Base Id of object used as rotation reference
- alpha1 : Mass-proportional damping coefficient
- alpha2 : Stiffness-proportional damping coefficient
- component_modes : Number of component modes
- consistent_mass : If True, use consistent mass (lumped is default)
- recovery : Flag for activating FE part recovery during solve (0=off, 1=stress recovery, 2=strain gage recovery, 3=both)

Returns

bool
True if the input is valid, otherwise False

`edit_triad`(*obj_id*, ***kwargs*)

Modifies an existing triad.

Parameters

obj_id
[int or str or list of int or list of str] Base Id or tag of the triad(s) to modify

kwargs
[dict] Keyword arguments containing the new triad attributes to assign. Currently, the following keywords are recognized:

- Tx, Ty, Tz : Translation offset in global X, Y, and Z-direction
- Rx, Ry, Rz : Euler angles (in degrees) for rotating the Triad
- tra_ref : Base Id of object used as translation reference
- rot_ref : Base Id of object used as rotation reference
- mass : Additional mass (and inertia) on the Triad
- mass_func : User Id or base Id (if negative) of mass scaling function
- constraints : Dictionary with keywords for constraing DOFs, i.e., `constraints={"Tx" : <value>, ..., "Rz" : <value>}`, where `<value>` can be any of the enum values *enums.FmDofStat*. Only the DOFs that should be changed need to be specified. You can also use "All" as key which implies all DOFs in the triad.

- init_vel : Dictionary with keywords specifying initial velocities, i.e., `init_vel={"Tx" : <u01>, ..., "Rz" : <u06>}`, where `<u0i>` is the initial velocity to be assigned local dof *i*.
- load : Dictionary with keywords specifying constant DOF loads, i.e., `load={"Tx" : <f1>, ..., "Rz" : <f6>}`, where `<fi>` is the user Id of a general function defining the load in local dof *i* if the type is int, otherwise it is taken as the constant load to be assigned local dof *i*.
- motion : Dictionary with keywords specifying prescribed motions, i.e., `motion={"Tx" : <u1>, ..., "Rz" : <u6>}`, where `<ui>` is the user Id of a general function defining the motion in local dof *i* if the type is int, otherwise it is taken as the constant motion to be prescribed in local dof *i*.

Returns

bool
True if the input is valid, otherwise False

`make_assembly`(*name*, *objs=None*)

Creates a sub-assembly and moves the specified objects into it.

Parameters

name
[str] Description of the new sub-assembly

objs
[list of int or str or list of str, default=None] List of base Ids or tags of the objects to put into the new sub-assembly

Returns

int
Base Id of the created sub-assembly

`make_beam`(*name*, *triads*, *bprop=None*, *tag=None*)

Creates a string of beam elements.

Parameters

name
[str] Description of the new beam(s)

triads
[list of int or list of str] List of base Ids or tags of the connected triads

bprop
[int or str, default=None] Base Id or tag of beam property to use

tag
[str, default=None] Tag to associate the created beam(s) with

Returns

list of int
Base Ids of the created beams, None if error

`make_beam_material`(*name*, *mprops*, *tag=None*)

Creates a material property object.

Parameters

name
[str] Description of the new material property

mprop
[list of float] List of property data

tag
[str, default=None] Tag to associate the created material property with

Returns

int
Base Id of material property object, zero or negative on error

`make_beam_section`(*name*, *mat*, *bprops*, *tag=None*)

Creates a beam cross section property object.

Parameters

name
[str] Description of the new beam property

mat
[int or str] Cross section type flag. If zero, a Generic cross section is defined. If str or a non-zero int, a Pipe cross section is defined, and the value gives the tag or base Id of the material to use.

bprop
[list of float] List of property data

tag
[str, default=None] Tag to associate the created beam property with

Returns

int
Base Id of beam property object, zero or negative on error

`make_damper`(*name*, *triads*, ***kwargs*)

Creates axial damper elements.

Parameters

name
[str] Description of the new damper(s)

triads
[(str, str) or (int, int) or list of (int, int)] Tags or base Ids of the connected triads. If a list of tuples is specified, one damper is created for each tuple.

kwargs
[dict] Keyword arguments defining the damper properties. The following keywords are currently supported:

- *tag* : Tag to associate the created damper(s) with
- *def_vel_damper*: If True, use deformational velocity
- *init_Damp_Coeff* : Constant damping coefficient
- *fn*: Base Id of damping coefficient function
- *xy*: List of XY-pairs giving a piece-wise linear damping coefficient
- *extrapol_type*: String, either “NONE” (default), “FLAT” or “LINEAR”

- *damp_characteristics*: String, either "DA_TRA_COEFF" (default) or "DA_TRA_FORCE"

Returns

int or list of int
Base Id(s) of the created damper(s), None if an error occurs

`make_fe_part`(*file*, *name=None*, *tag=None*)

Creates an FE part.

Parameters

file
[str] Absolute path to the FE data file

name
[str, default=None] Description of the new part, use file basename if not specified

tag
[str, default=None] Tag to associate the created part with

Returns

int
Base Id of the FE part object, zero or negative on error

`make_function`(*name*, ***kwargs*)

Creates a general function of time.

The type of function to be created is determined by which keyword arguments are provided. The keyword determining the function type is below marked by an asterix (*) in each case.

Parameters

name
[str] Description of the new function

kwargs
[dict]

Keyword arguments depending on function type. The following **function types** and *keywords* are currently supported:

1. **Polyline**

   *xy**: List of XY-pairs giving a piece-wise linear curve
   extrapol_type: String, either "NONE" (default), "FLAT" or "LINEAR"

2. **Polyline-from-file**

   *filename**: Name of file containing XY-pairs
   ch_name: String identifying the column to use for multi-column files
   sc_factor: Scaling factor, default=1.0
   z_adjust: If True, the y-values are shifted such that the first value is zero, default=False
   v_shift: Additional shift of the y-values, default=0.0

3. **Sine**

   *frequency**: Angular frequency

amplitude: Scaling factor, default=1.0
delay: Phase shift, default=0.0
mean_value: Constant shift, default=0.0
end: Default=0.0, if > 0, the function value is constant for $x >$ *end*

The Sine function therefore evaluates to:

```
f(x) = amplitude*sin(frequency*x-delay) + mean_value
```

for x-values less than *end*, whereas `f(x) = f(end)` for $x >$ *end*.

4. **Constant**

*value**: The constant function value, i.e., `f(x) = value` for any x

5. **Linear**

*slope**: The scaling factor, i.e., `f(x) = slope*x`

6. **Ramp**

*start_ramp**: Start point of sloped function domain
start_val: Function value before *start_ramp*, default=0.0
slope: The scaling factor, default=1.0

The Ramp function therefore evaluates to:

```
f(x) = start_val + slope*(x-start_ramp)
```

for x-values greater than *start_ramp*,
whereas `f(x) = start_val` for $x \leq$ *start_ramp*.

7. **Limited Ramp**

start_ramp: Start point of sloped function domain, default=0.0
*end_ramp**: End point of sloped function domain
start_val: Function value before *start_ramp*, default=0.0
slope: The scaling factor, default=1.0

The Limited Ramp function therefore evaluates to:

```
f(x) = start_val + slope*(x-start_ramp)
```

for x-values in the range [*start_ramp, end_ramp*]
whereas `f(x) = start_val` for $x <$ *start_ramp*,
and `f(x) = f(end_ramp)` for $x >$ *end_ramp*.

8. **Math expression**

*expression**: String containing the expression to use

9. **External function**

   No keywords

In addition, the *tag* keyword is accepted for all function types, to associate a specified tag with the created function, and the *base_id* keyword is used to indicate if the base Id of the created function should be returned, instead of the default user Id.

Returns

int
User Id or base Id of created function object. Will be 0 or negative if an error occurs.

`make_joint`(*name*, *joint_type*, *follower*, *followed=None*, *tag=None*)

Creates a joint object.

Parameters

name
[str] Description of the new joint

joint_type
[FmType] Type of joint

follower
[int or str] Base Id or tag of the dependent joint triad

followed
[int or str or list of int, default=None] Base Id or tag of the independent joint triad(s). If None, the joint is connected to ground and the independent triad is created at the same location as the dependent triad. For point-to-path joints, the first two triads specified are taken as the end points of the glider.

tag
[str, default=None] Tag to associate the created joint with

Returns

int
Base Id of joint object, zero or negative on error

`make_load`(*name*, *load_type*, *triad*, *load_dir*, *magnitude=None*, *fn=0*, *tag=None*)

Creates an external load object.

Parameters

name
[str] Description of the new load

load_type
[FmLoadType] Type of load

triad
[int or str] Base Id or tag of triad where the load attacks

load_dir
[(float, float, float)] Load direction vector

magnitude
[str, default=None] Load magnitude expression

fn
[int, default=0] User Id of load magnitude function

tag
[str, default=None] Tag to associate the created load with

Returns

int
Base Id of load object, zero or negative on error

`make_sensor`(*name*, *obj*, *var*, *dof=None*, *tag=None*)

Creates a sensor object.

Parameters

name
[str] Description of the new sensor

obj
[int or str or (int, int) or (str, str)] Base Id or tag of object(s) to measure

var
[FmVar] The variable to measure

dof
[FmDof, default=None] Local DOF to measure if *var* is a multi-DOF quantity

tag
[str, default=None] Tag to associate the created sensor with

Returns

int
User Id of sensor object, zero or negative on error

`make_spring`(*name*, *triads*, ***kwargs*)

Creates axial spring elements.

Parameters

name
[str] Description of the new spring(s)

triads
[(str, str) or (int, int) or list of (int, int)] Tags or base Ids of the connected triads. If a list of tuples is specified, one spring is created for each tuple.

tag
[str, default=None] Tag to associate the created beam(s) with

kwargs
[dict] Keyword arguments defining the spring properties. The following keywords are currently supported:

- *tag* : Tag to associate the created spring(s) with
- *constDefl*: Constant stress free deflection
- *constLength*: Constant stress free length
- *length*: User Id of general function defining stress free length
- *init_Stiff_Coeff*: Constant spring stiffness coefficient
- *fn*: Base Id of spring stiffness function
- *xy*: List of XY-pairs giving a piece-wise linear spring stiffness

- *extrapol_type*: String, either "NONE" (default), "FLAT" or "LINEAR"
- *spring_characteristics*: String, either "SPR_TRA_STIFF" (default) or "SPR_TRA_FORCE"

Returns

int or list of int
Base Id(s) of the created spring(s), None if an error occurs

`make_strain_rosette`(*name*, *part_id*, ***kwargs*)

Creates a strain rosette on an FE part.

Parameters

name
[str] Description of the new strain rosette

part_id
[int or str] Base Id or tag of FE part on which the rosette will be created

kwargs
[dict] Keyword arguments defining the strain rosette properties. The following keywords are recognized:

- *tag* : Tag to associate the created strain rosette with
- *nodes*: List of nodes to connect the strain rosette to
- *pos*: List of connection point coordinates, on the format *[(x1,y1,z1), (x2,y2,z2), (x3,y3,z3), (x4,y4,z4)]*. The last tuple is skipped if a 3-noded rosette is desired.
- *direction*: Tuple (x,y,z) defining the local X-axis direction vector of the strain rosette
- *angle* : Angle between strain gage direction and local X-axis
- *start_at_zero* : If True, the start strains are set to zero

Returns

int
Base Id of the strain rosette, zero or negative on error

`make_triad`(*name*, *pos=None*, *rot=None*, *node=0*, *on_part=0*, *tag=None*)

Creates a new triad at specified location or nodal point.

Parameters

name
[str] Description of the new triad

pos
[(float, float, float)] Global XYZ-coordinates of new triad

rot
[(float, float, float), default=None] Global Euler angles giving the orientation of new triad

node
[int, default=0] FE node number to associate the triad with. Used only if *on_part* is specified.

on_part
[int or str, default=0] Base Id or tag of the part that this triad should be attached to. You can also specify the Reference plane here, to create a new triad that is attached to ground.

tag
[str, default=None] Tag to associate the created triad with

Returns

int
Base Id of new triad, zero or negative on error

`make_udelm`(*name*, *triads*, ***kwargs*)

Creates a string of 2-noded user-defined elements.

Parameters

name
[str] Description of the new element(s)

triads
[list of int or list of str] List of base Ids or tags of the connected triads

kwargs
[dict] Keyword arguments defining some element properties. Currently, the following keywords are recognized:

- *alpha1* : Mass-proportional damping coefficient
- *alpha2* : Stiffness-proportional damping coefficient
- *tag* : Tag to associate the created element with

Returns

list of int
Base Ids of the created elements, None if error

`open`(*model_file*)

Opens the specified model file and prints out some key model parameters.

Parameters

model_file
[str] Absolute path of the Fedem model file to open

Returns

bool
True on success, otherwise False

`save`(*model_file=None*)

Saves current model into the specified model file.

If no model_file is given, that last opened model file is overwritten.

Parameters

model_file
[str, default=None] Absolute path of the Fedem model file to save to

Returns

bool
True on success, otherwise False

`solver_setup`(***kwargs*)

Setting up solver parameters, t_quasi, t_inc and t_end. Switching off initial equilibrium by setting t_quasi negative.

Parameters

kwargs
[dict] Keyword arguments containing the solver settings to assign. Currently, the following keywords are recognized:

- t_start : Start time
- t_end : Stop time
- t_inc : Time step size
- **t_quasi**
  [Stop time for quasi-static simulation.] If equal to t_start, perform initial equilibrium analysis before starting the dynamics time integration.
- **n_modes**
  [If non-zero, perform eigenvalue analysis during the simulation,] and calculate this number of modes each time
- e_inc: Time between each eigenvalue analysis
- add_opt: Additional solver options
- tol_ene: Convergence tolerance in energy norm
- tol_dis: Convergence tolerance in displacement norm
- tol_vel: Convergence tolerance in velocity norm
- tol_res: Convergence tolerance in force residual norm

2.8 reducer module

Python wrapper for the native Fedem FE part reducer. Used for convenience in order to hide native type convertions.

`class reducer.FedemReducer`(*lib_path*, *solver_options=None*)

Bases: `object`

This class mirrors the functionality of the Fedem FE part reducer library (libfedem_reducer_core.so on Linux, fedem_reducer_core.dll on Windows).

Parameters

lib_path
[str] Absolute path to the reducer shared object library

solver_options
[list of str, default=None] List of command-line arguments passed to the reducer

Methods

solver_init:	Initializes the command-line handler of the reducer
run:	Invokes the reducer

`run`(*options=None*)

This function invokes the reducer.

Parameters

options
[list of str, default=None] List of command-line arguments passed to the reducer

Returns

int
Zero on success, a negative value indicates some error condition

`solver_init`(*options*)

This function initializes the command-line handler of the reducer.

See the Fedem R7.5 Users Guide, Appendix C.2 for a complete list of all command-line arguments that may be specified and their default values.

Parameters

options
[list of str] List of command-line arguments passed to the reducer

Returns

int
Zero on success, a negative value indicates some error condition

2.9 solver module

Python wrapper for the native fedem dynamics solver. Used for convenience in order to hide native type convertions.

`exception solver.FedemException`(*errmsg*)

Bases: `Exception`

General exception-type for solver exceptions.

Parameters

errmsg
[str] Error message to print.

`class solver.FedemSolver`(*lib_path*, *solver_options=None*, *use_internal_state=False*)

Bases: `object`

This class mirrors the functionality of the fedem dynamics solver library (libfedem_solver_core.so on Linux, fedem_solver_core.dll on Windows). See also the header file ../../../proprietary/vpmSolver/solverInterface.h

In addition to the methods for conducting the simulation itself, the class also contains several methods for accessing and manipulating the linearized equation system, to facilitate use by external solution algorithms.

Parameters

lib_path
[str] Absolute path to the solver shared object library

solver_options
[list of str, default=None] List of command-line arguments passed to the solver

use_internal_state
[bool, default=False] If True, internal state arrays are allocated (for microbatching)

Methods

solver_init:	Processes the input and sets up necessary data structures
restart_from_state:	Re-initializes the mechanism objects with data from state array
solve_window:	Solves the problem for a time/load step window
get_state_size:	Returns the length of the state vector
get_gauge_size:	Returns the length of the initial strain gauge array
get_transformation_state_size:	Returns the length of the state vector holding transformation matrices
get_part_deformation_state_size:	Returns the length of the state vector holding deformation data
get_part_stress_state_size:	Returns the length of the state vector holding von Mises stress data
save_state:	Stores current solver state in the self.state_data array
save_gauges:	Stores initial gauge strains in the self.gauge_data array
save_transformation_state:	Stores current transformation state in provided core array
save_part_state:	Stores current deformation- and stress states in provided core arrays
solve_next:	Advances the solution one time/load step forward
start_step:	Starts a new time (or load) step
solve_iteration:	Solves current linearized equation system and updates state variables
finish_step:	Completes current time (or load) step
solve_modes:	Solves the eigenvalue problem at current time step
solve_inverse:	Solves the inverse problem at current time/load step
solver_done:	Closes down the model and cleans up heap memory and things on disk
solver_close:	Cleans up heap memory (singleton objects) on close
run_all:	Runs through the dynamics solver without any user intervention.
set_ext_func:	Assigns new value to an external function
get_current_time:	Returns the current physical time of the simulation
get_next_time:	Returns the physical time of the next step of the simulation
get_function:	Evaluates a general function in the model and returns its value
get_functions:	Evaluates several general functions in the model and returns their value
get_function_ids:	Returns a list of user Ids of tagged general functions
get_equations:	Returns the equation numbers associated with the DOFs of an object
get_system_size:	Returns the number of equations in the linearized system
get_system_dofs:	Returns the number of DOFs in the system
get_newton_matrix:	Returns current content of the system Newton matrix
get_stiffness_matrix:	Returns current content of the system stiffness matrix
get_mass_matrix:	Returns current content of the system mass matrix
get_damping_matrix:	Returns current content of the system damping matrix
get_element_stiffness_matrix:	Returns the content of a (beam) element stiffness matrix
get_rhs_vector:	Returns the content of the system right-hand-side vector
get_external_force_vector:	Returns the content of the external force vector
set_rhs_vector:	Replaces current content of the system right-hand-side vector
add_rhs_vector:	Updates the content of the system right-hand-side vector
compute_strains_from_displ:	Computes the strain tensor at gauges for given displacement field
get_current_strains:	Returns the current strain tensor for the specified gauges

continues on next page

Table 3 – continued from previous page

compute_rel_dist_from_displ:	Computes relative distance at sensors for given displacement field
compute_int_forces_from_displ:	Computes beam section forces in triads for given displacement field
compute_spring_var_from_displ:	Computes one of the spring variables for given displacement field
get_joint_spring_stiffness:	Returns current joint spring stiffness coefficient(s)

`add_rhs_vector`(*r_vec*)

Utility updating the content of the system right-hand-side vector.

`check_times`(*xtimes*, *use_times=True*)

Utility checking that a given time series starts with the current solver time.

`compute_int_forces_from_displ`(*disp*, *ids*)

This method computes beam section forces in triads for the given displacement field.

There are 2 cases:

InternalForce component defined by an array
dofi contains dof number in a certain direction i
ids = [beamID1, triadID1, dof1, beamID2, triadID2, dof2, …..]

InternalSectionForces defined by an array
dofi contains always -1, 6 components are requested
ids = [beamID1, triadID1, -1, beamID2, triadID2, -1, …..]

Parameters

disp
[list of float] Displacement field

ids
[list of int] Array with beam and triad identification numbers

Returns

list of float
Force components

bool
Always True, unless the calculation failed

`compute_rel_dist_from_displ`(*disp*, *ids*)

This method computes the relative distance at sensors for the given displacement field.

Parameters

disp
[list of float] Displacement field

ids
[list of int] Array with function identification numbers

Returns

list of float
Displacement components

bool
Always True, unless the calculation failed

`compute_spring_var_from_displ`(*disp*, *ids*)

This method computes one of the spring variables (length, deflection, force) at sensors for the given displacement field.

Parameters

disp
[list of float] Displacement field

ids
[list of int] Array with function identification numbers

Returns

list of float
Displacement components

bool
Always True, unless the calculation failed

`compute_strains_from_displ`(*disp*, *gauge_ids*)

This method computes the strain tensor at gauges for the given displacement field, or from current state if no displacements provided.

Parameters

disp
[list of float] Displacement field to evaluate strains from

gauge_ids
[list of int] Array with gauge identification numbers

Returns

list of float
Strain components

bool
Always True, unless the calculation failed

`finish_step`()

This method completes current time (or load) step, by iterating the linearized equation system until convergence is achieved. The self.ierr variable has the value zero on a successful computation. A non-zero value indicates some error that requires the simulation to terminate.

Returns

bool
Always True, unless current time/load step failed to converge, or the end time of the simulation has been reached

`get_current_strains`(*gauge_ids*)

This method computes the strain tensor at gauges from current state.

Parameters

gauge_ids
[list of int] Array with gauge identification numbers

Returns

list of float
Strain components

bool
Always True, unless the calculation failed

`get_current_time()`

Utility returning the current physical time of the simulation. The self.ierr variable is not touched.

`get_damping_matrix()`

Utility returning current content of the system damping matrix.

`get_element_stiffness_matrix(bid)`

Utility returning the initial content of an element stiffness matrix. Use this method for beam elements only, 6 dof per node (total 12).

`get_equations(bid)`

Utility returning the equation numbers for the DOFs of the object with the specified base Id (bid).

`get_external_force_vector()`

Utility returning current content of the external force vector.

`get_function(uid=0, tag=None, arg=None)`

Utility evaluating a general function in the model, identified by the specified user Id *uid* or *tag*, and with the function argument *arg*. If *arg* is *None* or contains a negative value, the function is instead evaluated for current state of the sensor object that is defined as the function argument. The same is done if a *tag* is specified. If the specified function could not be evaluated, the self.ierr variable is decremented. Otherwise, it is not touched.

`get_function_ids(tags)`

Utility returning a list of user Ids of tagged general functions.

`get_functions(uids)`

Utility evaluating a list of general functions for current state. The *uids* argument can be a list of either the user Ids of the functions to be evaluated, or their corresponding objects tags. If the specified functions could not be evaluated, the self.ierr variable is decremented for each problem encountered. Otherwise, it is not touched.

`get_gauge_size()`

Utility returning the required size of the initial strain gauge array which is used when restarting a simulation from an in-core array. Returns 0 if the model does not contain any strain gauges.

`get_joint_spring_stiffness(bid)`

Get joint spring stiffness coefficient(s).

Parameters

bid
[int] Joint identification number (base ID)

Returns

list of float
Spring stiffness coefficients for the joint DOFs

bool
Always True, unless the extraction failed

get_mass_matrix()

Utility returning current content of the system mass matrix.

get_newton_matrix()

Utility returning current content of the system Newton matrix.

get_next_time()

Utility returning the physical time of the next step of the simulation. If the time step size is defined by a general function that could not be evaluated, the self.ierr variable is decremented. Otherwise, it is not touched.

get_part_deformation_state_size(*base_id*)

Utility returning the required length of the state vector which stores deformation data for the FE Part with the given base Id. Returns -1 if the specified Part does not exist.

The size/length of the state vector is:
(number of nodal points in the FE Part) * 3

get_part_stress_state_size(*base_id*)

Utility returning the required length of the state vector which stores von Mises stresses for the FE Part with the given base Id. Returns -1 if the specified Part does not exist, and 0 if the specified Part does not contain any (shell or solid) elements with stresses.

get_rhs_vector()

Utility returning current content of the system right-hand-side vector.

get_state_size()

Utility returning the required length of the state vector which is used when restarting a simulation from an in-core array.

get_stiffness_matrix()

Utility returning current content of the system stiffness matrix.

get_system_dofs()

Utiloty returning the total number of DOFs of the system.

get_system_size()

Utility returning the dimension (number of equations) of the system.

get_transformation_state_size()

Utility returning the required length of the vector which stores the transformation matrices (rotation and translation) for Triads, Parts and Beams.

The size/length of the state vector is:
3 +
(number of Triads) * 14 +
(number of Parts) * 14
(number of Beams) * 14

`restart_from_state`(*state_data*, *write_to_rdb=2*)

This method re-initializes the mechanism objects with data from the provided state array, such that the simulation can continue from there.

Parameters

state_data
[list of float] Complete state vector to restart simulation from

write_to_rdb
[int, default=2]

Flag for saving response variables to results database,
= 0 : No results saving,
= 1 : Append results to already opened results database,
= 2 : Increment the results database and write to new files

Returns

int
Zero on success, a negative value indicates some error

`run_all`(*options*)

This method runs the dynamics solver with given command-line *options*, without any user intervention.

`save_gauges`()

This method stores initial gauge strains in the self.gauge_data array.

Returns

bool
Always True, unless the self.gauge_data array is too small

`save_part_state`(*base_id*, *def_state*, *str_state*, *ndef=None*, *nstr=None*)

This method stores current deformation- and stress states for the specified FE Part in the provided core arrays.

Parameters

base_id
[int] Base Id of the FE Part to save state for

def_state
[list of c_double] Array to fill with deformation data

str_state
[list of c_couble] Array to fill with stress data

ndef
[int, default=None] Length of the def_state array, set to len(def_state) if none

nstr
[int, default=None] Length of the str_state array, set to len(str_state) if none

Returns

bool
Always True, unless one or both of the state arrays are too small

`save_state`()

This method stores current solver state in the self.state_data array.

Returns

bool
Always True, unless the self.state_data array is too small

`save_transformation_state`(*state_data*, *ndat=None*)

This method stores current transformation state for Triads, Parts and Beams in the provided core array.

The transformation state data is on the format:

[step number]
[current time]
[current time increment]
for each non-fixed Triad:
1
[rotMatrix column 1]
[rotMatrix column 2]
[rotMatrix column 3]
[translation vector]
for each Part and Beam:
2
[rotMatrix column 1]
[rotMatrix column 2]
[rotMatrix column 3]
[translation vector]

Parameters

state_data
[list of c_double] Array to fill with transformation data

ndat
[int, default=None] Length of the state_data array, set to len(state_data) if None

Returns

bool
Always True, unless the state_data array is too small

`set_ext_func`(*func_id*, *value=None*)

This method may be used prior to the solve_next call, to assign a sensor value from a physical twin to the specified actuator or load in the model, identified by the argument func_id (external function Id).

Parameters

func_id
[int] Id of the external function to be assigned new value

value
[float, default=None] The value to be assigned

Returns

bool
Always True, unless the specified function is not found

`set_rhs_vector`(*r_vec*)

Utility replacing current content of the system right-hand-side vector.

`solve_inverse`(*x_val*, *x_def*, *g_def*, *out_def=None*)

This method solves the inverse problem at current time/load step, assuming small deformations only (linear response). If an error condition that requires the simulation to terminate occurs, the self.ierr variable is assigned a non-zero value.

Parameters

x_val
[list of float] Specified displacement values at a set of degrees of freedom

x_def
[list of int] Equation numbers for the specified displacement values

g_def
[list of int] Equation numbers for the DOFs with unknown external forces

out_def
[list of int, default=None] User Ids of the functions to evaluate the response for

Returns

list of float, only if out_def is specified
Evaluated response variables

bool
Always True, unless the simulation has to stop due to some error, or the end of the simulation has been reached

`solve_iteration`()

This method solves the current linearized equation system and updates all state variables. Then it assembles the linearized system of equations for next iteration, unless convergence has been reached. The self.ierr variable has the value zero on a successful computation. A non-zero value indicates some error that requires the simulation to terminate.

Returns

bool
Always True, unless the simulation has to stop due to some error, or the current time/load step has converged.

`solve_modes`(*n_modes*, *dof_order=False*, *use_lapack=0*)

This method solves the eigenvalue problem at current time step, and returns the computed eigenvalues and associated eigenvectors. If an error condition that requires the simulation to terminate occurs, the self.ierr variable is assigned a negative value.

Parameters

n_modes
[int] Number of eigenmodes to calculate

dof_order
[bool, default=False] If True, the eigenvectors are returned in DOF-order instead of equation order which is the default

use_lapack
[int, default=0] Flag usage of LAPACK eigensolvers (0=No, 1=DSYGVX, 2=DGGEVX)

Returns

list of float
The computed eigenvalues

list of list of float
The computed eigenvectors

bool
Always True, unless the computation failed

`solve_next`(*inp=None*, *inp_def=None*, *out_def=None*, *time_next=None*)

This method advances the solution one time/load step forward. The self.ierr variable has the value zero on a successful computation. A non-zero value indicates some error that requires the simulation to terminate.

Parameters

inp
[list of float, default=None] Input function values

inp_def
[list of int, default=None] External function Ids of the functions to assign values

out_def
[list of int, default=None] User Ids of the functions to evaluate the response for

time_next
[float, default=None] Time of next step, to override time step size defined in the model

Returns

list of float, only if out_def is specified
Evaluated response variables

bool
Always True, unless current time/load step failed to converge, or the end time of the simulation has been reached

`solve_window`(*n_step*, *inputs=None*, *f_out=None*, *xtimes=None*)

This method solves the problem for a time/load step window, with given values for the external functions, and extraction of results from another set of general functions in the model.

A non-zero value on self.ierr on exit indicates that an error condition that will require the simulation to terminate has occurred.

Parameters

n_step
[int] Number of time/load steps to solve for from current state

inputs
[list of float, default=None] List of input sensor values for each time step. The length of this list must be equal to *n_step* times the number of input sensors.

f_out
[list of int, default=None] List of user Ids identifying the output sensors in the model

xtimes
[list of float, default=None] List of times associated with the inputs. The length of this list must be equal to *n_step*.

Returns

list of float
Output sensor values for each time step

bool
Always True, unless the end of the simulation has been reached

`solver_close()`

This method needs to be used if solver_done() was invoked with its *remove_singletons* argument set to False. It will delete those singleton objects here instead.

`solver_done(remove_singletons=None)`

This method should be used when the time/load step loop is finished. It closes down the model and cleans up heap memory and things on disk.

Parameters

remove_singletons
[bool default=None] If True or None, heap-allocated singelton objects are also released

Returns

int
Zero on success, non-zero values indicates errors.

`solver_init(options, fsi=None, state_data=None, gauge_data=None, extf_input=None)`

This method processes the input and sets up necessary data structures prior to the time integration loop. It also performs the license checks.

See the Fedem R7.5 Users Guide, Appendix C.3 for a complete list of all command-line arguments that may be specified and their default values.

If the argument fsi is None, the model is assumed defined in the file specified via the command-line argument -fsifile instead.

Parameters

options
[list of str] List of command-line arguments passed to the solver

fsi
[str, default=None] Content of the solver input file describing the model

state_data
[list of float, default=None] Complete state vector to restart simulation from

gauge_data
[list of float, default=None] Initial strain gauge values for restart

extf_input
[list of float, default=None] Initial external function values, for initial equilibrium iterations

Returns

int
A negative value indicates an error, otherwise success. A positive value indicates that initial equilibrium iterations was performed with external function values from *extf_input*

start_step()

This method starts a new time (or load) step, by calculating the predicted response, the coefficient matrix and right-hand-side vector of the first nonlinear iteration. It has to be followed up by a series of solve_iteration calls in order to continue the simulation, but the linear equation system can be manipulated in between. The self.ierr variable has the value zero on a successful computation. A non-zero value indicates some error that requires the simulation to terminate.

Returns

bool
Always True, unless the simulation has to stop due to some error, or the end time of the simulation has been reached

2.10 write_fmx module

Python wrapper for native FMX-writer library.

write_fmx.**read**(*fnam*, *ityp*, *data*)

This function reads a rectangular matrix from a binary FMX-file for FEDEM.

Parameters

fnam
[str] Absolute path to the fmx-file to be read

ityp
[int] Type of matrix to read, 1=stiffness matrix, 2=mass matrix, 3=gravity force vectors

data
[list of float] Matrix content, column-wise storage

Returns

int
Zero on success, otherwise negative

write_fmx.**write**(*fnam*, *ityp*, *data*)

This function writes a rectangular matrix as a binary FMX-file for FEDEM.

Parameters

fnam
[str] Absolute path to the fmx-file to be written

ityp
[int] Type of matrix to write, 1=stiffness matrix, 2=mass matrix, 3=gravity force vectors

data
[list of float] Matrix content, column-wise storage

Returns

int
Zero on success, otherwise negative

2.11 yaml_parser module

This module provides functionality for creating a Fedem model based on a YAML-formatted input file. The following keywords are recognized when parsing the input file:

target_file - specifies the name of the fmm-file to be created
source_file - specifies an fmm-file to be used as template for the model.
file_exists - specified what to do if the `target_file`already exists:
    USE_IT - opens this file for editing existing model
    OVERWRITE! - ignores and overwrites any existing file
    STOP! - aborts the execution
    index - creates a new non-existing file name
fe_parts - imports specified FE data files as a Parts
triads - creates triads at specified global points
triads_from_fe_parts - creates triads at nodal points on specified FE Parts
beam_materials - creates Beam material objects
beam_sections - creates Beam cross section objects
beams - creates Beam element objects
joints - creates Joint objects
loads - create Load objects
virtual_sensors - creates Sensor objects for extraction of response data
functions - creates Function objects (including input functions)
spring_dampers - creates Axial spring/damper objects
edit_fe_parts - modifies the properties of FE Part objects
edit_triads - modifies the properties of Triad objects
edit_joints - modifies the properties of Joint objects
edit_settings - modifies the solver settings

Each keyword (except the first three) are followed by an arbitrary number of lines, where each line defines one object to be created (or edited). The syntax of the input can be either list-based or dictionary based.

This module can also be launched directly using the syntax

```
python -m fedempy.yaml_parser -f mymodel.yaml
```

It will then invoke the method `yaml_parser.ModelYAML.main()` on the specified input file (*mymodel.yaml*).

class yaml_parser.**ModelYAML**(*input_file*)

Bases: `object`

This class contains methods for parsing a YAML-formated input file and creating a Fedem model from it. It is an extension of the class *modeler.FedemModeler*.

Parameters

input_file
[str] Absolute path to the YAML-formated input file

Methods

build:	Loads and parses all model attributes into the model instance
save:	Saves the model to the input file specified FEDEM model file
solve:	Executes the dynamics solver on the created model.

`build`(*dump_file=None*)

Loads and parses all model attributes into the model instance.

Parameters

dump_file
[str, default=None] Absolute path to dump the model data dictionary to

`save`(*save_as=False*)

Saves the model with the defined file name, or save it as a new file if *save_as* is set.

Parameters

save_as
[bool, default=False] If True, the model is stored with a new file name

`solve`()

Executes the dynamics solver on the created model.

yaml_parser.`main`(*input_file*, *dump_file=None*, *solve=False*)

Main driver.

Parameters

input_file
[str] Absolute path to the YAML-formated input file

dump_file
[str, default=None] Absolute path to json-file for dumping the model data dictionary

solve
[bool, default=False] If True, the dynamics solver is launched on the created model

CHAPTER

THREE

DTS_OPERATORS MODULE

3.1 dts_operators.stress_visualization

Convenience module for running Fedem stress recovery with CUG export as an operator. Usage: Invoke *run(df)* with the input data in the DataFrame *df*.

fedempy.dts_operators.stress_visualization.**run**(*df*, ***kwargs*)

Run Fedem simulation with stress recovery and create CUG visualization.

Parameters

df

[Dataframe] Input function values

kwargs

[dict] Dictionary containing configuration parameters and/or solver options

fedempy.dts_operators.stress_visualization.**stress_visualization_run**(*df*, ***kwargs*)

For backwards compatibility. Consider remove and update existing apps using this entry accordingly.

3.2 dts_operators.submodel

Convenience functions for running FEDEM sub-model simulations as an operator. To use, call sub_model_run(df) with input data in df.

fedempy.dts_operators.submodel.**sub_model_run**(*df*)

Run sub-model simulation. First solves the global model, then recursively solves the submodels. Reads external submodel_config.json file located in lib-folder.

Parameters

df

[Pandas dataframe] Dataframe containing input data

3.3 dts_operators.window

Convenience module for running Fedem simulations as an operator. Usage: Invoke *run(df)* with the input data in the DataFrame *df*.

fedempy.dts_operators.window.**run**(*df*, *dts=None*, ***kwargs*)

Run Fedem simulation over a time window with *df* as input.

Parameters

df
[DataFrame] Input function values

dts
[DTSContext, default=None] State data to be passed between each micro-batch

kwargs
[dict] Dictionary containing output definitions and/or solver options

Returns

DataFrame
Response values in output sensors

fedempy.dts_operators.window.**start_fmm_solver**(*fmm_file*, *lib_dir*, *use_state=False*, *old_state=None*, *ext_input=None*, *t_start=None*, *keep_old_res=True*)

Starts the dynamics solver on the provided Fedem model file.

Parameters

fmm_file
[str] The Fedem model file to run simulation on

lib_dir
[str] Path to where the model file is located, used only if *fmm_file* is a relative path

use_state
[bool, default=False] If True, internal state vectors will be allocated

old_state
[list of float, default=None] State vector to restart simulation from

ext_input
[list of float, default=None] Initial external function values, for initial equilibrium iterations

t_start
[float, default=None] Optional start time of simulation, override setting in model file

keep_old_res
[bool, default=True] Option to not overwrite any existing res-file in the RDB directory

Returns

FmmSolver
The dynamics solver object

int
Zero on success, negative values indicate errors

fedempy.dts_operators.window.**start_solver**(*solver_options*, *cw_dir*, *use_state=False*, *old_state=None*, *ext_input=None*, *tstart=None*)

Starts the dynamics solver with the provided command-line options.

Parameters

solver_options
[list of str] List of command-line arguments that are passed to the solver

cw_dir
[str] Path to working directory of the solver process

use_state
[bool, default=False] If True, internal state vectors will be allocated

old_state
[list of float, default=None] State vector to restart simulation from

ext_input
[list of float, default=None] Initial external function values, for initial equilibrium iterations

tstart
[float, default=None] Optional start time of simulation, override setting in model file

Returns

FedemSolver
The dynamics solver object

int
Zero on success, negative values indicate errors

3.4 dts_operators.driver

Convenience drivers for running FEDEM simulations as an operator.

These drivers can also take the model input as low-code (python) or no-code (yaml) files, and the corresponding fmm-file of the model will then be generated the first time the operator is invoked.

`fedempy.dts_operators.driver.`**`run_stress_visualization`**`(`*df*, ***kwargs*`)`

Run batch fedem simulation with *df* as input, to create CUG stress visualization of the FE-parts.

Parameters

df
[DataFrame] Input function values

kwargs
[dict] Dictionary containing output definitions and/or solver options

`fedempy.dts_operators.driver.`**`run_window`**`(`*df*, *dts=None*, ***kwargs*`)`

Run fedem simulation with *df* as input.

Parameters

df
[DataFrame] Input function values

dts
[DTSContext, default=None] State data to be passed between each micro-batch

kwargs
[dict] Dictionary containing output definitions and/or solver options

Returns

Dataframe
Response values in output sensors

3.5 dts_operators.fmm_creator

Utility module, for generating fmm-files from low/no-code model files.

fedempy.dts_operators.fmm_creator.**create_fmm**(*lib_dir*, *fmm_file=None*, *mod_file=None*)

Creates a Fedem model file (fmm) based on a low-code (python) or no-code (yaml) model file. If the *fmm_file* is specified directly, that file will be used instead. If the *lib_dir* folder already contains fmm-files, the first one found will be used. This will be the case when starting a new micro-batch in a streaming app.

Parameters

lib_dir
[str] Absolute path to directory where the fmm-file should be stored

fmm_file
[str, default=None] Name of fmm-file, if provided explicitly

mod_file
[str, default=None] Name of low/no-code model file to generate fmm-file from

Returns

str
Absolute path to (generated) fmm-file

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX