MuPIF Reference manual Documentation

Release 2.0.0

Bořek Patzák and Vít Šmilauer

December 22, 2017

CHAPTER 1

Introduction

Multi-Physics Integration Framework (MuPIF) is an integration framework, that will facilitate the implementation of multi-physic and multi-level simulations, built from independently developed components. The principal role of the framework is to steer individual components (applications) and to provide high-level data-exchange services. Each application should implement an interface that allows to steer application and execute data requests. The design supports various coupling strategies, discretization techniques, and also the distributed applications. The platform development is hosted on GitHub (https://github.com/mupif/mupif/).

The approach followed in this project is based on an object-oriented approach, consisting in designing a system of interacting objects for the purpose of solving a software problem. The identification of individual objects and their mutual interaction has been based on expertise of project partners, and later refined by analysis of simulation scenarios considered in the project. The main advantage of this approach lies in independence on particular data format(s), as the exchanged data (fields, properties) are represented as abstract classes. Therefore, the focus on services is provided by objects (object interfaces) and not on underlying data itself.

The integration framework is implemented in Python3. Python is an interpreted, interactive, object-oriented programming language. It runs on many Unix/Linux platforms, on the Mac, and on PCs under MS-DOS, Windows, Windows NT, and OS/2. The Python language is enriched by new objects/classes to describe and to represent complex simulation chains. Such approach allows profiting from the capabilities of established scripting environment, including numerical libraries, serialization/persistence support, VPN, and remote communication.

The proposed abstract classes are designed to represent the entities in a model space, including simulation tools, fields, discretizations, properties, etc. The purpose of these abstract classes is to define a common interface that needs to be implemented by any derived class. Such interface concept allows using any derived class on a very abstract level, using common interface for services, without being concerned with the implementation details of an individual software component.

To facilitate execution and development of the simulation workflows, the platform provides the transparent communication mechanism that will take care of the network communication between the objects. An important feature is the transparency, which hides the details of remote communication to the user and allows working with local and remote objects in the same way. The communication layer is built on Pyro4 library, which provides a transparent distributed object system fully integrated into Python. It takes care of the network communication between the objects when they are distributed over different machines on the network. The platform is designed to work on virtually any distributed platform, including grid and cloud infrastructure.

In addition to this MuPIF reference manual, a user manual from https://github.com/mupif/mupif/tree/master/mupif/doc/userManual can be obtained, showing details on API implementation, installation, networking and providing several examples in local/distributed setups.

mupif package

Subpackages

mupif.Physics package

Submodules

mupif.Physics.NumberDict module

Dictionary storing numerical values

class `mupif.Physics.NumberDict.`**`NumberDict`**

Bases: `dict`

Dictionary storing numerical values

Constructor: NumberDict()

An instance of this class acts like an array of number with generalized (non-integer) indices. A value of zero is assumed for undefined entries. NumberDict instances support addition, and subtraction with other NumberDict instances, and multiplication and division by scalars.

mupif.Physics.PhysicalQuantities module

Physical quantities with units.

This module provides a data type that represents a physical quantity together with its unit. It is possible to add and subtract these quantities if the units are compatible, and a quantity can be converted to another compatible unit. Multiplication, subtraction, and raising to integer powers is allowed without restriction, and the result will have the correct unit. A quantity can be raised to a non-integer power only if the result can be represented by integer powers of the base units.

The values of physical constants are taken from the 1986 recommended values from CODATA. Other conversion factors (e.g. for British units) come from various sources. I can't guarantee for the correctness of all entries in the unit table, so use this at your own risk.

SI derived units; these automatically get prefixes: Y (1E+24), Z (1E+21), E (1E+18), P (1E+15), T (1E+12), G (1E+09), M (1E+06), k (1E+03), h (1E+02), da (1E+01), d (1E-01), c (1E-02), m (1E-03), mu (1E-06), n (1E-09), p (1E-12), f (1E-15), a (1E-18), z (1E-21), y (1E-24)

Hz Hertz 1/s N Newton m*kg/s**2 Pa Pascal N/m**2 J Joule N*m W Watt J/s C Coulomb s*A V Volt W/A F Farad C/V ohm Ohm V/A S Siemens A/V Wb Weber V*s T Tesla Wb/m**2 H Henry Wb/A lm Lumen cd*sr lx Lux lm/m**2 Bq Becquerel 1/s Gy Gray J/kg Sv Sievert J/kg

Prefixed units for ohm:

Yohm, Zohm, Eohm, Pohm, Tohm, Gohm, Mohm, kohm, hohm, daohm, dohm, cohm, mohm, muohm, nohm, pohm, fohm, aohm, zohm, yohm

Prefixed units for rad:

Yrad, Zrad, Erad, Prad, Trad, Grad, Mrad, krad, hrad, darad, drad, crad, mrad, murad, nrad, prad, frad, arad, zrad, yrad

Prefixed units for mol:

Ymol, Zmol, Emol, Pmol, Tmol, Gmol, Mmol, kmol, hmol, damol, dmol, cmol, mmol, mumol, nmol, pmol, fmol, amol, zmol, ymol

Prefixed units for cd:

Ycd, Zcd, Ecd, Pcd, Tcd, Gcd, Mcd, kcd, hcd, dacd, dcd, ccd, mcd, mucd, ncd, pcd, fcd, acd, zcd, ycd

Prefixed units for Pa:

YPa, ZPa, EPa, PPa, TPa, GPa, MPa, kPa, hPa, daPa, dPa, cPa, mPa, muPa, nPa, pPa, fPa, aPa, zPa, yPa

Prefixed units for Hz:

YHz, ZHz, EHz, PHz, THz, GHz, MHz, kHz, hHz, daHz, dHz, cHz, mHz, muHz, nHz, pHz, fHz, aHz, zHz, yHz

Prefixed units for Wb:

YWb, ZWb, EWb, PWb, TWb, GWb, MWb, kWb, hWb, daWb, dWb, cWb, mWb, muWb, nWb, pWb, fWb, aWb, zWb, yWb

Prefixed units for lm:

Ylm, Zlm, Elm, Plm, Tlm, Glm, Mlm, klm, hlm, dalm, dlm, clm, mlm, mulm, nlm, plm, flm, alm, zlm, ylm

Prefixed units for Bq:

YBq, ZBq, EBq, PBq, TBq, GBq, MBq, kBq, hBq, daBq, dBq, cBq, mBq, muBq, nBq, pBq, fBq, aBq, zBq, yBq

Prefixed units for lx:

Ylx, Zlx, Elx, Plx, Tlx, Glx, Mlx, klx, hlx, dalx, dlx, clx, mlx, mulx, nlx, plx, flx, alx, zlx, ylx

Prefixed units for A:

YA, ZA, EA, PA, TA, GA, MA, kA, hA, daA, dA, cA, mA, muA, nA, pA, fA, aA, zA, yA

Prefixed units for C:

YC, ZC, EC, PC, TC, GC, MC, kC, hC, daC, dC, cC, mC, muC, nC, pC, fC, aC, zC, yC

Prefixed units for F:

YF, ZF, EF, PF, TF, GF, MF, kF, hF, daF, dF, cF, mF, muF, nF, pF, fF, aF, zF, yF

Prefixed units for H:

YH, ZH, EH, PH, TH, GH, MH, kH, hH, daH, dH, cH, mH, muH, nH, pH, fH, aH, zH, yH

Prefixed units for K:

YK, ZK, EK, PK, TK, GK, MK, kK, hK, daK, dK, cK, mK, muK, nK, pK, fK, aK, zK, yK

Prefixed units for J:

YJ, ZJ, EJ, PJ, TJ, GJ, MJ, kJ, hJ, daJ, dJ, cJ, mJ, muJ, nJ, pJ, fJ, aJ, zJ, yJ

Prefixed units for Sv:

YSv, ZSv, ESv, PSv, TSv, GSv, MSv, kSv, hSv, daSv, dSv, cSv, mSv, muSv, nSv, pSv, fSv, aSv, zSv, ySv

Prefixed units for N:

YN, ZN, EN, PN, TN, GN, MN, kN, hN, daN, dN, cN, mN, muN, nN, pN, fN, aN, zN, yN

Prefixed units for S:

YS, ZS, ES, PS, TS, GS, MS, kS, hS, daS, dS, cS, mS, muS, nS, pS, fS, aS, zS, yS

Prefixed units for T:

YT, ZT, ET, PT, TT, GT, MT, kT, hT, daT, dT, cT, mT, muT, nT, pT, fT, aT, zT, yT

Prefixed units for W:

YW, ZW, EW, PW, TW, GW, MW, kW, hW, daW, dW, cW, mW, muW, nW, pW, fW, aW, zW, yW

Prefixed units for V:

YV, ZV, EV, PV, TV, GV, MV, kV, hV, daV, dV, cV, mV, muV, nV, pV, fV, aV, zV, yV

Prefixed units for none:

Ynone, Znone, Enone, Pnone, Tnone, Gnone, Mnone, knone, hnone, danone, dnone, cnone, mnone, munone, nnone, pnone, fnone, anone, znone, ynone

Prefixed units for g:

Yg, Zg, Eg, Pg, Tg, Gg, Mg, kg, hg, dag, dg, cg, mg, mug, ng, pg, fg, ag, zg, yg

Prefixed units for sr:

Ysr, Zsr, Esr, Psr, Tsr, Gsr, Msr, ksr, hsr, dasr, dsr, csr, msr, musr, nsr, psr, fsr, asr, zsr, ysr

Prefixed units for m:

Ym, Zm, Em, Pm, Tm, Gm, Mm, km, hm, dam, dm, cm, mm, mum, nm, pm, fm, am, zm, ym

Prefixed units for Gy:

YGy, ZGy, EGy, PGy, TGy, GGy, MGy, kGy, hGy, daGy, dGy, cGy, mGy, muGy, nGy, pGy, fGy, aGy, zGy, yGy

Prefixed units for s:

Ys, Zs, Es, Ps, Ts, Gs, Ms, ks, hs, das, ds, cs, ms, mus, ns, ps, fs, as, zs, ys

Fundamental constants: c speed of light 299792458.*m/s mu0 permeability of vacuum 4.e-7*pi*N/A**2 eps0 permittivity of vacuum 1/mu0/c**2 Grav gravitational constant 6.67259e-11*m**3/kg/s**2 hplanck Planck constant 6.6260755e-34*J*s hbar Planck constant / 2pi hplanck/(2*pi) e elementary charge 1.60217733e-19*C me electron mass 9.1093897e-31*kg mp proton mass 1.6726231e-27*kg Nav Avogadro number 6.0221367e23/mol k Boltzmann constant 1.380658e-23*J/K

Time units: min minute 60*s h hour 60*min d day 24*h wk week 7*d yr year 365.25*d

Length units: inch inch 2.54*cm ft foot 12*inch yd yard 3*ft mi (British) mile 5280.*ft nmi Nautical mile 1852.*m Ang Angstrom 1.e-10*m lyr light year c*yr Bohr Bohr radius 4*pi*eps0*hbar**2/me/e**2

Area units: ha hectare 10000*m**2 acres acre mi**2/640 b barn 1.e-28*m**2

Volume units: l liter dm**3 dl deci liter 0.1*l cl centi liter 0.01*l ml milli liter 0.001*l tsp teaspoon 4.92892159375*ml tbsp tablespoon 3*tsp floz fluid ounce 2*tbsp cup cup 8*floz pt pint 16*floz qt quart 2*pt galUS US gallon 4*qt galUK British gallon 4.54609*l

Mass units: amu atomic mass units 1.6605402e-27*kg oz ounce 28.349523125*g lb pound 16*oz ton ton 2000*lb

Force units: dyn dyne (cgs unit) 1.e-5*N

Energy units: erg erg (cgs unit) 1.e-7*J eV electron volt e*V Hartree Wavenumbers/inverse cm me*e**4/16/pi**2/eps0**2/hbar**2 Ken Kelvin as energy unit k*K cal thermochemical calorie 4.184*J kcal thermochemical kilocalorie 1000*cal cali international calorie 4.1868*J kcali international kilocalorie 1000*cali Btu British thermal unit 1055.05585262*J

Prefixed units for eV:

YeV, ZeV, EeV, PeV, TeV, GeV, MeV, keV, heV, daeV, deV, ceV, meV, mueV, neV, peV, feV, aeV, zeV, yeV

Power units: hp horsepower 745.7*W

Pressure units: bar bar (cgs unit) 1.e5*Pa atm standard atmosphere 101325.*Pa torr torr = mm of mercury atm/760 psi pounds per square inch 6894.75729317*Pa

Angle units: deg degrees pi*rad/180

Temperature units: degR degrees Rankine (5./9.)*K degC degrees Celcius <PhysicalUnit degC> degF degree Fahrenheit <PhysicalUnit degF>

class `mupif.Physics.PhysicalQuantities.`**`PhysicalQuantity`**`(`**args*`)`

Bases: `future.types.newobject.newobject`

Physical quantity with units

PhysicalQuantity instances allow addition, subtraction, multiplication, and division with each other as well as multiplication, division, and exponentiation with numbers. Addition and subtraction check that the units of the two operands are compatible and return the result in the units of the first operand. A limited set of mathematical functions (from module Numeric) is applicable as well:

- sqrt: equivalent to exponentiation with 0.5.
- **sin, cos, tan: applicable only to objects whose unit is** compatible with 'rad'.

See the documentation of the PhysicalQuantities module for a list of the available units.

Here is an example on usage:

```
>>> from PhysicalQuantities import PhysicalQuantity as p  # short hand
>>> distance1 = p('10 m')
>>> distance2 = p('10 km')
>>> total = distance1 + distance2
>>> total
PhysicalQuantity(10010.0,'m')
>>> total.convertToUnit('km')
>>> total.getValue()
10.01
>>> total.getUnitName()
'km'
>>> total = total.inBaseUnits()
>>> total
PhysicalQuantity(10010.0,'m')
>>>
>>> t = p(314159., 's')
>>> # convert to days, hours, minutes, and second:
>>> t2 = t.inUnitsOf('d','h','min','s')
>>> t2_print = ' '.join([str(i) for i in t2])
>>> t2_print
'3.0 d 15.0 h 15.0 min 59.0 s'
>>>
>>> e = p('2.7 Hartree*Nav')
>>> e.convertToUnit('kcal/mol')
>>> e
PhysicalQuantity(1694.2757596034764,'kcal/mol')
```

```
>>> e = e.inBaseUnits()
>>> str(e)
'7088849.77818 kg*m**2/s**2/mol'
>>>
>>> freeze = p('0 degC')
>>> freeze = freeze.inUnitsOf ('degF')
>>> str(freeze)
'32.0 degF'
>>>
m = PQ(12,'kg')
a = PQ('0.88 km/s**2')
F = m*a
print F
```

#vector valued quantities: a = PQ((1,2,3),'m') scalar = PQ(2.0, 's') a.convertToUnit('km') a.inUnitsOf('dm') a*3.0 a*scalar

F = F.inBaseUnits() print F

print F.isCompatible('MN') print F.isCompatible('m')

F.convertToUnit('MN') # convert to Mega Newton print F F = F + PQ(0.1, 'kPa*m**2') # kilo Pascal m^2 print F print str(F)

value = float(str(F).split()[0]) print value

convertToUnit (*unit*)

Change the unit and adjust the value such that the combination is equivalent to the original one. The new unit must be compatible with the previous unit of the object.

Parameters **unit** (`C{str}`) – a unit

Raises **TypeError** – if the unit string is not a know unit or a unit incompatible with the current one

cos ()

getUnitName ()

Return unit (string) of physical quantity.

getValue ()

Return value (float) of physical quantity (no unit).

inBaseUnits ()

Returns the same quantity converted to base units, i.e. SI units in most cases

Return type L{PhysicalQuantity}

inUnitsOf (**units*)

Express the quantity in different units. If one unit is specified, a new PhysicalQuantity object is returned that expresses the quantity in that unit. If several units are specified, the return value is a tuple of PhysicalObject instances with with one element per unit such that the sum of all quantities in the tuple equals the the original quantity and all the values except for the last one are integers. This is used to convert to irregular unit systems like hour/minute/second.

Parameters **units** (`C{str} or sequence of C{str}`) – one or several units

Returns one or more physical quantities

Return type L{PhysicalQuantity} or C{tuple} of L{PhysicalQuantity}

Raises **TypeError** – if any of the specified units are not compatible with the original unit

isCompatible(*unit*)

Parameters **unit** (*C{str}*) – a unit

Returns C{True} if the specified unit is compatible with the one of the quantity

Return type C{bool}

sin()

sqrt()

tan()

class mupif.Physics.PhysicalQuantities.**PhysicalUnit**(*names*, *factor*, *powers*, *offset=0*)

Bases: future.types.newobject.newobject

Physical unit

A physical unit is defined by a name (possibly composite), a scaling factor, and the exponentials of each of the SI base units that enter into it. Units can be multiplied, divided, and raised to integer powers.

conversionFactorTo(*other*)

Parameters **other** (*L{PhysicalUnit}*) – another unit

Returns the conversion factor from this unit to another unit

Return type C{float}

Raises **TypeError** – if the units are not compatible

conversionTupleTo(*other*)

Parameters **other** (*L{PhysicalUnit}*) – another unit

Returns the conversion factor and offset from this unit to another unit

Return type (C{float}, C{float})

Raises **TypeError** – if the units are not compatible

isAngle()

isCompatible(*other*)

Parameters **other** (*L{PhysicalUnit}*) – another unit

Returns C{True} if the units are compatible, i.e. if the powers of the base units are the same

Return type C{bool}

isDimensionless()

name()

setName(*name*)

mupif.Physics.PhysicalQuantities.**assertPhysicalUnitEqual**(*first*, *second*, *msg=None*)

mupif.Physics.PhysicalQuantities.**description**()

Return a string describing all available units.

mupif.Physics.PhysicalQuantities.**getDimensionlessUnit**()

return dimensionless unit

mupif.Physics.PhysicalQuantities.**isPhysicalQuantity**(*x*)

Parameters **x** (*any*) – an object

Returns C{True} if x is a L{PhysicalQuantity}

Return type C{bool}

`mupif.Physics.PhysicalQuantities.`**`isPhysicalUnit`**(*x*)

Parameters **x** (*any*) – an object

Returns C{True} if x is a L{PhysicalUnit}

Return type C{bool}

Module contents

Submodules

mupif.APIError module

exception `mupif.APIError.`**`APIError`**

Bases: `exceptions.Exception`

This class serves as a base class for exceptions thrown by the framework. Raising an exception is a way to signal that a routine could not execute normally - for example, when an input argument is invalid (e.g. value is outside of the domain of a function) or when a resource it relies on is unavailable (like a missing file, a hard disk error, or out-of-memory errors)

Exceptions provide a way to react to exceptional circumstances (like runtime errors) in programs by transferring control to special functions called handlers. To catch exceptions, a portion of code is placed under exception inspection. This is done by enclosing that portion of code in a try-block. When an exceptional circumstance arises within that block, an exception is thrown that transfers the control to the exception handler. If no exception is thrown, the code continues normally and all handlers are ignored.

An exception is thrown by using the throw keyword from inside the "try" block. Exception handlers are declared with the keyword "except", which must be placed immediately after the try block.

mupif.Application module

class `mupif.Application.`**`Application`**(*file=''*, *workdir=''*)

Bases: *`mupif.MupifObject.MupifObject`*

An abstract class representing an application and its interface (API).

The purpose of this class is to define abstract services for data exchange and steering. This interface has to be implemented/provided by any application. The data exchange is performed by the means of new data types introduced in the framework, namely properties and fields. New abstract data types (properties, fields) allow to hide all implementation details related to discretization and data storage.

`__init__`(*file=''*, *workdir=''*)

Constructor. Initializes the application.

Parameters

- **file** (*str*) – Name of file
- **workdir** (*str*) – Optional parameter for working directory

finishStep (*tstep*)
Called after a global convergence within a time step is achieved.

Parameters **tstep** (TimeStep) – Solution step

getAPIVersion ()

Returns Returns the supported API version

Return type str, int

getApplicationSignature ()
Get application signature.

Returns Returns the application identification

Return type str

getAssemblyTime (*tstep*)
Returns the assembly time related to given time step. The registered fields (inputs) should be evaluated in this time.

Parameters **tstep** (TimeStep) – Solution step

Returns Assembly time

Return type Physics.PhysicalQuantity, TimeStep

getCriticalTimeStep ()
Returns a critical time step for an application.

Returns Returns the actual (related to current state) critical time step increment

Return type Physics.PhysicalQuantity

getField (*fieldID*, *time*)
Returns the requested field at given time. Field is identified by fieldID.

Parameters

- **fieldID** (FieldID) – Identifier of the field
- **time** (*Physics.PhysicalQuantity*) – Target time

Returns Returns requested field.

Return type *Field*

getFieldURI (*fieldID*, *time*)
Returns the uri of requested field at given time. Field is identified by fieldID.

Parameters

- **fieldID** (FieldID) – Identifier of the field
- **time** (*Physics.PhysicalQuantity*) – Target time

Returns Requested field uri

Return type Pyro4.core.URI

getFunction (*funcID*, *objectID=0*)
Returns function identified by its ID

Parameters

- **funcID** (FunctionID) – function ID

- **objectID** (*int*) – Identifies optional object/submesh on which property is evaluated (optional, default 0)

Returns Returns requested function

Return type *Function*

getMesh (*tstep*)
Returns the computational mesh for given solution step.

Parameters **tstep** (TimeStep) – Solution step

Returns Returns the representation of mesh

Return type *Mesh*

getProperty (*propID*, *time*, *objectID=0*)
Returns property identified by its ID evaluated at given time.

Parameters

- **propID** (PropertyID) – property ID
- **time** (*Physics.PhysicalQuantity*) – Time when property should to be evaluated
- **objectID** (*int*) – Identifies object/submesh on which property is evaluated (optional, default 0)

Returns Returns representation of requested property

Return type *Property*

getURI ()

Returns Returns the application URI or None if application not registered in Pyro

Return type str

isSolved ()
Check whether solve has completed.

Returns Returns true or false depending whether solve has completed when executed in background.

Return type bool

registerPyro (*pyroDaemon*, *pyroNS*, *pyroURI*, *appName=None*, *externalDaemon=False*)
Register the Pyro daemon and nameserver. Required by several services

Parameters

- **pyroDaemon** (*Pyro4.Daemon*) – Optional pyro daemon
- **pyroNS** (*Pyro4.naming.Nameserver*) – Optional nameserver
- **PyroURI** (*string*) – Optional URI of receiver
- **appName** (*string*) – Optional application name. Used for removing from pyroNS
- **externalDaemon** (*bool*) – Optional parameter when daemon was allocated externally.

removeApp (*nameServer=None*, *appName=None*)
Removes (unregisters) application from the name server.

Parameters

- **nameServer** (*Pyro4.naming.Nameserver*) – Optional instance of a nameServer
- **appName** (*str*) – Optional name of the application to be removed

restoreState (*tstep*)
Restore the saved state of an application. :param TimeStep tstep: Solution step

setField (*field*)
Registers the given (remote) field in application.

Parameters **field** (`Field`) – Remote field to be registered by the application

setFunction (*func*, *objectID=0*)
Register given function in the application.

Parameters

- **func** (`Function`) – Function to register
- **objectID** (*int*) – Identifies optional object/submesh on which property is evaluated (optional, default 0)

setProperty (*property*, *objectID=0*)
Register given property in the application

Parameters

- **property** (`Property`) – Setting property
- **objectID** (*int*) – Identifies object/submesh on which property is evaluated (optional, default 0)

solveStep (*tstep*, *stageID=0*, *runInBackground=False*)
Solves the problem for given time step.

Proceeds the solution from actual state to given time. The actual state should not be updated at the end, as this method could be called multiple times for the same solution step until the global convergence is reached. When global convergence is reached, finishStep is called and then the actual state has to be updated. Solution can be split into individual stages identified by optional stageID parameter. In between the stages the additional data exchange can be performed. See also wait and isSolved services.

Parameters

- **tstep** (`TimeStep`) – Solution step
- **stageID** (*int*) – optional argument identifying solution stage (default 0)
- **runInBackground** (*bool*) – optional argument, defualt False. If True, the solution will run in background (in separate thread or remotely).

storeState (*tstep*)
Store the solution state of an application.

Parameters **tstep** (`TimeStep`) – Solution step

terminate ()
Terminates the application. Shutdowns daemons if created internally.

wait ()
Wait until solve is completed when executed in background.

class `mupif.Application.`**RemoteApplication** (*decoratee*, *jobMan=None*, *jobID=None*, *appTunnel=None*)
Bases: `object`

Remote Application instances are normally represented by auto generated pyro proxy. However, when application allocated using JobManager or ssh tunnel needs to be established, the proper termination of the tunnel or job manager task is required.

This class is a decorator around pyro proxy object represeting application storing the reference to job manager and related jobID or/and ssh tunnel.

These extermal attributes could not be injected into Application instance, as it is remote instance (using proxy) and the termination of job and tunnel has to be done from local computer, which has the neccesary communication link established (ssh tunnel in particular, when port translation takes place)

getJobID()

terminate()
Terminates the application. Terminates the allocated job at jobManager

mupif.BBox module

class mupif.BBox.**BBox**(*coords_ll*, *coords_ur*)
Bases: `future.types.newobject.newobject`

Represents a bounding box - a rectange in 2D and prism in 3D. Its geometry is described using two points - lover left and upper right corners. The bounding box class provides fast and efficient methods for testing whether point is inside it and whether intersection with other BBox exist.

__init__(*coords_ll*, *coords_ur*)
Constructor.

Parameters

- **coords_ll** (*tuple*) – Tuple with coordinates of lower left corner
- **coords_ur** (*tuple*) – Tuple with coordinates of uper right corner

__str__()

Returns Returns lower left and upper right coordinate of the bounding box

Return type str

containsPoint(*point*)
Check whether a point lies within a receiver.

Parameters **point** (*tuple*) – 1D/2D/3D position vector

Returns Returns True if point is inside receiver, otherwise False

Return type bool

intersects(*bbox*)
Check intersection of a receiver with a bounding box

Parameters **bbox** (BBox) – an instance of BBox class

Returns Returns True if receiver intersects given bounding box, otherwise False

Return type bool

merge(*entity*)
Merges receiver with given entity (position vector or a BBox).

Parameters

- **entity** (BBox) – 1D/2D/3D position vector or
- **entity** – an instance of BBox class

mupif.Cell module

class `mupif.Cell.`**`Brick_3d_lin`**(*mesh*, *number*, *label*, *vertices*)

Bases: *`mupif.Cell.Cell`*

Unstructured 3d tetrahedral element with linear interpolation

`_evalN`(*lc*)

Evaluates shape functions at given point (given in parametric coordinates) :param tuple lc: A local coordinate :return: shape function :rtype: float

`containsPoint`(*point*)

Check if a cell contains a point.

Parameters **`point`** (*`tuple`*) – 1D/2D/3D position vector

Returns Returns True if cell contains a given point

Return type bool

`copy`()

This will copy the receiver, making a deep copy of all atributes EXCEPT mesh attribute.

Returns A deep copy of a receiver

Return type *Cell*

classmethod **`getGeometryType`**()

Returns geometry type of receiver.

Returns Returns geometry type of receiver

Return type *CellGeometryType*

`getTransformationJacobian`(*coords*)

Returns the transformation jacobian (the determinant of jacobian) of the receiver

Parameters **`coords`** (*`tuple`*) – local (parametric) coordinates of the point

Returns jacobian

Return type float

`glob2loc`(*coords*)

Converts global coordinate to local (area) coordinate.

Parameters **`coords`** (*`tuple`*) – A coordinate in global system

Returns local (area) coordinate

Return type tuple

`interpolate`(*point*, *vertexValues*)

Interpolates given vertex values to a given point.

Parameters

- **`point`** (*`tuple`*) – 1D/2D/3D position vector
- **`vertexValues`** (*`tuple`*) – A tuple containing vertex values

Returns Interpolated value at a given point

Return type tuple

`loc2glob`(*lc*)

Converts local (parametric) coordinates to global ones

Parameters **`lc`** (*`tuple`*) – A local coordinate

Returns global coordinate

Return type tuple

class `mupif.Cell.`**`Cell`**(*mesh*, *number*, *label*, *vertices*)

Bases: `future.types.newobject.newobject`

Representation of a computational cell.

The solution domain is composed of cells (e.g. finite element), whose geometry is defined using vertices (e.g. nodes). Cells provide interpolation over their associated volume, based on given vertex values. Derived classes will be implemented to support common interpolation cells (finite elements, FD stencils, etc.)

`__init__`(*mesh*, *number*, *label*, *vertices*)

Initializes the cell.

Parameters

- **`mesh`** (`Mesh`) – The mesh to which a cell belongs to
- **`number`** (*`int`*) – A local cell number. Local numbering should start from 0 and should be continuous.
- **`label`** (*`int`*) – A cell label. Arbitrary unique number.
- **`vertices`** (*`tuple`*) – A cell vertices (local numbers)

`containsPoint`(*point*)

Check if a cell contains a point.

Parameters **`point`** (*`tuple`*) – 1D/2D/3D position vector

Returns Returns True if cell contains a given point

Return type bool

`copy`()

This will copy the receiver, making a deep copy of all attributes EXCEPT a mesh attribute

Returns A deep copy of a receiver

Return type *Cell*

`getBBox`(*relPad=1e-05*)

Return bounding box. The box is by default slightly enlarged via *relPad* to avoid finite-precision issues when testing for a boundary point being inside the box.

Parameters **`relPad`** (*`float`*) – relative padding of the box; tight (geometrical) bbox will be enlarged along each axis by *relPad* times size along that axis, in both directions.

Returns Returns a bounding box of the receiver

Return type *BBox*

static **`getClassForCellGeometryType`**(*cgt*)

Return class object (not instance) for given cell geometry type. Does introspection of all subclasses of Cell caches the result.

classmethod **`getGeometryType`**()

Returns geometry type of receiver.

Returns Returns geometry type of receiver

Return type *CellGeometryType*

getNumberOfVertices()

Returns Number of vertices

Return type int

getTransformationJacobian(*coords*)

Returns the transformation jacobian (the determinant of jacobian) of the receiver

Parameters **coords** (*tuple*) – local (parametric) coordinates of the point

Returns jacobian

Return type float

getVertices()

Returns The list of cell vertices

Return type tuple

interpolate(*point*, *vertexValues*)

Interpolates given vertex values to a given point.

Parameters

- **point** (*tuple*) – 1D/2D/3D position vector
- **vertexValues** (*tuple*) – A tuple containing vertex values

Returns Interpolated value at a given point

Return type tuple

class mupif.Cell.**Quad_2d_lin**(*mesh*, *number*, *label*, *vertices*)

Bases: *mupif.Cell.Cell*

Unstructured 2d quad element with linear interpolation

containsPoint(*point*)

Check if a cell contains a point.

Parameters **point** (*tuple*) – 1D/2D/3D position vector

Returns Returns True if cell contains a given point

Return type bool

copy()

This will copy the receiver, making deep copy of all atributes EXCEPT mesh attribute.

Returns A deep copy of a receiver

Return type *Cell*

classmethod getGeometryType()

Returns geometry type of receiver.

Returns Returns geometry type of receiver

Return type *CellGeometryType*

getTransformationJacobian(*coords*)

Returns the transformation jacobian (the determinant of jacobian) of the receiver

Parameters **coords** (*tuple*) – local (parametric) coordinates of the point

Returns jacobian

Return type float

glob2loc (*coords*)
Converts global coordinate to local (area) coordinate.

Parameters **coords** (*tuple*) – A coordinate in global system

Returns local (area) coordinate

Return type tuple

interpolate (*point*, *vertexValues*)
Interpolates given vertex values to a given point.

Parameters

- **point** (*tuple*) – 1D/2D/3D position vector
- **vertexValues** (*tuple*) – A tuple containing vertex values

Returns Interpolated value at a given point

Return type tuple

loc2glob (*lc*)
Converts local (parametric) coordinates to global ones.

Parameters **lc** (*tuple*) – A local coordinate

Returns global coordinate

Return type tuple

class mupif.Cell.**Tetrahedron_3d_lin** (*mesh*, *number*, *label*, *vertices*)
Bases: *mupif.Cell.Cell*

Unstructured 3d tetrahedral element with linear interpolation.

containsPoint (*point*)
Check if a cell contains a point.

Parameters **point** (*tuple*) – 1D/2D/3D position vector

Returns Returns True if cell contains a given point

Return type bool

copy ()
This will copy the receiver, making a deep copy of all atributes EXCEPT mesh attribute.

Returns A deep copy of a receiver

Return type *Cell*

classmethod getGeometryType ()
Returns geometry type of receiver.

Returns Returns geometry type of receiver

Return type *CellGeometryType*

getTransformationJacobian (*coords*)
Returns the transformation jacobian (the determinant of jacobian) of the receiver

Parameters **coords** (*tuple*) – local (parametric) coordinates of the point

Returns jacobian

Return type float

glob2loc (*coords*)
Converts global coordinate to local (area) coordinate.

Parameters **coords** (*tuple*) – A coordinate in global system

Returns local (area) coordinate

Return type tuple

interpolate (*point*, *vertexValues*)
Interpolates given vertex values to a given point.

Parameters

- **point** (*tuple*) – 1D/2D/3D position vector
- **vertexValues** (*tuple*) – A tuple containing vertex values

Returns Interpolated value at a given point

Return type tuple

loc2glob (*lc*)
Converts local (parametric) coordinates to global ones

Parameters **lc** (*tuple*) – A local coordinate

Returns global coordinate

Return type tuple

class mupif.Cell.**Triangle_2d_lin** (*mesh*, *number*, *label*, *vertices*)
Bases: *mupif.Cell.Cell*

Unstructured 2D triangular element with linear interpolation Node numbering convention:

2 | | | | 0—–1

containsPoint (*point*)
Check if a cell contains a point.

Parameters **point** (*tuple*) – 1D/2D/3D position vector

Returns Returns True if cell contains a given point

Return type bool

copy ()
This will copy the receiver, making a deep copy of all atributes EXCEPT mesh attribute.

Returns A deep copy of a receiver

Return type *Cell*

classmethod getGeometryType ()
Returns geometry type of receiver.

Returns Returns geometry type of receiver

Return type *CellGeometryType*

getTransformationJacobian (*coords*)
Returns the transformation jacobian (the determinant of jacobian) of the receiver

Parameters **coords** (*tuple*) – local (parametric) coordinates of the point

Returns jacobian

Return type float

glob2loc(*coords*)
Converts global coordinate to local (area) coordinate.

Parameters **coords** (*tuple*) – A coordinate in global system

Returns local (area) coordinate

Return type tuple

interpolate(*point*, *vertexValues*)
Interpolates given vertex values to a given point.

Parameters

- **point** (*tuple*) – 1D/2D/3D position vector
- **vertexValues** (*tuple*) – A tuple containing vertex values

Returns Interpolated value at a given point

Return type tuple

loc2glob(*lc*)
Converts local (parametric) coordinates to global ones.

Parameters **lc** (*tuple*) – A local coordinate

Returns global coordinate

Return type tuple

class mupif.Cell.**Triangle_2d_quad**(*mesh*, *number*, *label*, *vertices*)
Bases: *mupif.Cell.Cell*

Unstructured 2D triangular element with quadratic interpolation Node numbering convention:

2 | | 5 4 | | 0–3—1

containsPoint(*point*)
Check if a cell contains a point.

Parameters **point** (*tuple*) – 1D/2D/3D position vector

Returns Returns True if cell contains a given point

Return type bool

copy()
This will copy the receiver, making a deep copy of all atributes EXCEPT mesh attribute.

Returns A deep copy of a receiver

Return type *Cell*

classmethod getGeometryType()
Returns geometry type of receiver.

Returns Returns geometry type of receiver

Return type *CellGeometryType*

getTransformationJacobian(*coords*)
Returns the transformation jacobian (the determinant of jacobian) of the receiver

Parameters **coords** (*tuple*) – local (parametric) coordinates of the point

Returns jacobian

Return type float

glob2loc(*coords*)
Converts global coordinate to local (area) coordinate.

Parameters **coords** (*tuple*) – A coordinate in global system

Returns local (area) coordinate

Return type tuple

interpolate(*point*, *vertexValues*)
Interpolates given vertex values to a given point.

Parameters

- **point** (*tuple*) – 1D/2D/3D position vector
- **vertexValues** (*tuple*) – A tuple containing vertex values

Returns Interpolated value at a given point

Return type tuple

loc2glob(*lc*)
Converts local (parametric) coordinates to global ones.

Parameters **lc** (*tuple*) – A local coordinate

Returns global coordinate

Return type tuple

mupif.CellGeometryType module

Enumeration defining the supported cell geometries

mupif.EnsightReader2 module

mupif.EnsightReader2.**readEnsightField**(*name*, *parts*, *partRec*, *type*, *fieldID*, *mesh*, *units*, *time*)
Reads either Per-node or Per-element variable file and returns corresponding Field representation.

Parameters

- **name** (*str*) – Input field name with variable data
- **parts** (*tuple*) – Only parts with id contained in partFiler will be imported
- **partRec** (*list*) – A list containing info about individual parts (number of elements per each element type).
- **type** (*int*) – Determines type of field values: type = 1 scalar, type = 3 vector, type = 6 tensor
- **fieldID** (FieldID) – Field type (displacement, strain, temperature ...)
- **mesh** (Mesh) – Corresponding mesh
- **units** (PhysicalUnit) – field units
- **time** (PhysicalQuantity) – time

Returns FieldID for unknowns

Return type *Field*

mupif.EnsightReader2.**readEnsightGeo**(*name*, *partFilter*, *partRec*)

Reads Ensight geometry file (Ensight6 format) and returns corresponding Mesh object instance. Supports only unstructured meshes.

Parameters

- **name** (*str*) – Path to Ensight geometry file (*.geo)
- **partFiler** (*tuple*) – Only parts with id contained in partFiler will be imported
- **partRec** (*list*) – A list containing info about individual parts (number of elements). Needed by readEnsightField

Returns mesh

Return type *Mesh*

mupif.EnsightReader2.**readEnsightGeo_Part**(*f*, *line*, *mesh*, *enum*, *cells*, *vertexMapping*, *partnum*, *partdesc*, *partRec*)

Reads single cell part geometry from an Ensight file.

Parameters

- **f** (*File*) – File object
- **line** (*str*) – Current line to process (should contain element type)
- **mesh** (Mesh) – Mupif mesh object to accommodate new cells
- **enum** (*int*) – Accumulated cell number
- **cells** (*list*) – List of individual Cells
- **vertexMapping** (*dict*) – Map from vertex label (as given in Ensight file) to local number
- **partnum** (*int*) – Part number
- **partdesc** (*list*) – Partition description record
- **partRec** (*list*) – Output agrument (list) containing info about individual parts (number of elements). Needed by readEnsightField

Returns tuple (line, cell number)

Return type tuple (line, enum)

mupif.Field module

class mupif.Field.**Field**(*mesh*, *fieldID*, *valueType*, *units*, *time*, *values=None*, *fieldType=1*)

Bases: *mupif.MupifObject.MupifObject*, *mupif.Physics.PhysicalQuantities.PhysicalQuantity*

Representation of field. Field is a scalar, vector, or tensorial quantity defined on a spatial domain. The field, however is assumed to be fixed at certain time. The field can be evaluated in any spatial point belonging to underlying domain.

Derived classes will implement fields defined on common discretizations, like fields defined on structured/unstructured FE meshes, FD grids, etc.

__init__(*mesh*, *fieldID*, *valueType*, *units*, *time*, *values=None*, *fieldType=1*)

Initializes the field instance.

Parameters

- **mesh** (`Mesh`) – Instance of a Mesh class representing the underlying discretization
- **fieldID** (`FieldID`) – Field type (displacement, strain, temperature ...)
- **valueType** (`ValueType`) – Type of field values (scalear, vector, tensor). Tensor is a tuple of 9 values. It is changed to 3x3 for VTK output automatically.
- **units** (*Physics.PhysicalUnits*) – Field value units
- **time** (*Physics.PhysicalQuantity*) – Time associated with field values
- **values** (*list of tuples representing individual values*) – Field values (format dependent on a particular field type, however each individual value should be stored as tuple, even scalar value)
- **fieldType** (`FieldType`) – Optional, determines field type (values specified as vertex or cell values), default is FT_vertexBased

_evaluate (*position*, *eps*)
Evaluates the receiver at a single spatial position.

Parameters

- **position** (*tuple*) – 1D/2D/3D position vector
- **eps** (*float*) – Optional tolerance

Returns field value

Return type tuple of doubles

Note: This method has some issues related to https://sourceforge.net/p/mupif/tickets/22/ .

commit ()
Commits the recorded changes (via setValue method) to a primary field.

dumpToLocalFile (*fileName*, *protocol=2*)
Dump Field to a file using a Pickle serialization module.

Parameters

- **fileName** (*str*) – File name
- **protocol** (*int*) – Used protocol - 0=ASCII, 1=old binary, 2=new binary

evaluate (*positions*, *eps=0.0*)
Evaluates the receiver at given spatial position(s).

Parameters

- **position** (*tuple, a list of tuples*) – 1D/2D/3D position vectors
- **eps** (*float*) – Optional tolerance for probing whether the point belongs to a cell (should really not be used)

Returns field value(s)

Return type Physics.PhysicalQuantity with given value or tuple of values

field2Image2D (*plane='xy'*, *elevation=(-1e-06, 1e-06)*, *numX=10*, *numY=20*, *interp='linear'*, *fieldComponent=0*, *vertex=True*, *colorBar='horizontal'*, *colorBarLegend=''*, *barRange=(None, None)*, *barFormatNum='%.3g'*, *title=''*, *xlabel=''*, *ylabel=''*, *fileName=''*, *show=True*, *figsize=(8, 4)*, *matPlotFig=None*)
Plots and/or saves 2D image using a matplotlib library. Works for structured and unstructured 2D/3D fields. 2D/3D fields need to define plane. This method gives only basic viewing options, for aesthetic and

more elaborated output use e.g. VTK field export with postprocessors such as ParaView or Mayavi. Idea from https://docs.scipy.org/doc/scipy/reference/tutorial/interpolate.html#id1

Parameters

- **field** (Field) – field of unknowns
- **plane** (*str*) – what plane to extract from field, valid values are 'xy', 'xz', 'yz'
- **elevation** (*tuple*) – range of third coordinate. For example, in plane='xy' is grabs z coordinates in the range
- **numX** (*int*) – number of divisions on x graph axis
- **numY** (*int*) – number of divisions on y graph axis
- **interp** (*str*) – interpolation type when transferring to a grid. Valid values 'linear', 'nearest' or 'cubic'
- **fieldComponent** (*int*) – component of the field
- **vertex** (*bool*) – if vertices shoud be plot as points
- **colorBar** (*str*) – color bar details. Valid values '' for no colorbar, 'vertical' or 'horizontal'
- **colorBarLegend** (*str*) – Legend for color bar. If '', current field name and units are printed. None prints nothing.
- **barRange** (*tuple*) – min and max bar range. If barRange=('NaN','NaN'), it is adjusted automatically
- **barFormatNum** (*str*) – format of color bar numbers
- **title** (*str*) – title
- **xlabel** (*str*) – x axis label
- **ylabel** (*str*) – y axis label
- **fileName** (*str*) – if nonempty, a filename is written to the disk, usually png, pdf, ps, eps and svg are supported
- **show** (*bool*) – if the plot should be showed
- **figsize** (*tuple*) – size of canvas in inches. Affects only showing a figure. Image to a file adjust one side automatically.
- **matPlotFig** (*obj*) – False means plot window remains in separate thread, True waits until a plot window becomes closed

Returns handle to matPlotFig

Return type matPlotFig

field2Image2DBlock()

Block an open window from matPlotLib. Waits until closed.

field2VTKData(*name=None*, *lookupTable=None*)

Creates VTK representation of the receiver. Useful for visualization. Requires pyvtk module.

Parameters

- **name** (*str*) – human-readable name of the field
- **lookupTable** (*pyvtk.LookupTable*) – color lookup table

Returns Instance of pyvtk

Return type pyvtk

getCellValue(*componentID*)
Returns the value associated with a given integration point on a cell.

Parameters **componentID** (*tuple*) – A tuple identifying a component: vertex (vertexID,) or integration point (CellID, IPID)

Returns The value

Return type Physics.PhysicalQuantity

getFieldID()
Returns FieldID, e.g. FID_Displacement, FID_Temperature.

Returns Returns field ID

Return type *FieldID*

getFieldIDName()
Returns name of the field.

Returns Returns fieldID name

Return type string

getFieldType()
Returns receiver field type (values specified as vertex or cell values)

Returns Returns fieldType id

Return type *FieldType*

getMartixForTensor(*values*)
Reshape values to a list with 3x3 arrays. Usable for VTK export.

Parameters **values** (*list*) – List containing tuples of 9 values, e.g. [(1,2,3,4,5,6,7,8,9), (1,2,3,4,5,6,7,8,9), ...]

Returns List containing 3x3 matrices for each tensor

Return type list

getMesh()
Obtain mesh.

Returns Returns a mesh of underlying discretization

Return type *Mesh*

getRecordSize()
Return the number of scalars per value, depending on `valueType` passed when constructing the instance.

Returns number of scalars (1,3,9 respectively for scalar, vector, tensor)

Return type int

getTime()
Get time of the field.

Returns Time of field data

Return type Physics.PhysicalQuantity

getUnits()

Returns Returns units of the receiver

Return type Physics.PhysicalUnits

getValueType()

Returns ValueType of the field, e.g. scalar, vector, tensor.

Returns Returns value type of the receiver

Return type *ValueType*

getVertexValue(*componentID*)

Returns the value associated with a given vertex component

Parameters **componentID** (*tuple*) – A tuple identifying a component: vertex (vertexID,)

Returns The value

Return type Physics.PhysicalQuantity

giveValue(*componentID*)

Returns the value associated with a given component (vertex or integration point on a cell).

Parameters **componentID** (*tuple*) – A tuple identifying a component: vertex (vertexID,) or integration point (CellID, IPID)

Returns The value

Return type tuple

inUnitsOf(**units*)

Should return a new instance. As deep copy is expensive, this operation should be avoided. Better to use convertToUnits method performing in place conversion.

classmethod loadFromLocalFile(*fileName*)

Alternative constructor which loads instance directly from a Pickle module.

Parameters **fileName** (*str*) – File name

Returns Returns Field instance

Return type *Field*

static makeFromHdf5(*fileName*, *group='component1/part1'*)

Restore Fields from HDF5 file.

Parameters

- **fileName** (*str*) – HDF5 file
- **group** (*str*) – HDF5 group the data will be read from (IOError is raised if the group does not exist).

Returns list of new *Field* instances

Return type [Field,Field,...]

Note: This method has not been tested yet.

static makeFromVTK2(*fileName, unit, time=0, skip=['coolwarm']*)

Return fields stored in *fileName* in the VTK2 (`.vtk`) format.

Parameters

- **fileName** (*str*) – filename to load from
- **PhysicalUnit** (*unit*) – physical unit of filed values

- **time** (*float*) – time value for created fields (time is not saved in VTK2, thus cannot be recovered)
- **skip** (*[string,]*) – file names to be skipped when reading the input file; the default value skips the default coolwarm colormap.

Returns one field from VTK

Return type *Field*

static makeFromVTK3 (*fileName*, *units*, *time=0*, *forceVersion2=False*)

Create fields from a VTK unstructured grid file (`.vtu`, format version 3, or `.vtp` with *forceVersion2*); the mesh is shared between fields.

`vtk.vtkXMLGenericDataObjectReader` is used to open the file (unless *forceVersion2* is set), but it is checked that contained dataset is a `vtk.vtkUnstructuredGrid` and an error is raised if not.

Note: Units are not supported when loading from VTK, all fields will have `None` unit assigned.

Parameters

- **fileName** (*str*) – VTK (`*.vtu`) file
- **units** (`PhysicalUnit`) – units of read values
- **time** (*float*) – time value for created fields (time is not saved in VTK3, thus cannot be recovered)
- **forceVersion2** (*bool*) – if `True`, `vtk.vtkGenericDataObjectReader` (for VTK version 2) will be used to open the file, isntead of `vtk.vtkXMLGenericDataObjectReader`; this also supposes *fileName* ends with `.vtk` (not checked, but may cause an error).

Returns list of new *Field* instances

Return type [Field,Field,...]

static manyToVTK3 (*fields*, *fileName*, *ascii=False*, *compress=True*)

Save all fields passed as argument into VTK3 Unstructured Grid file (`*.vtu`).

All *fields* must be defined on the same mesh object; exception will be raised if this is not the case.

Parameters

- **fileName** – output file name
- **ascii** (*bool*) – write numbers are ASCII in the XML-based VTU file (rather than base64-encoded binary in XML)
- **compress** (*bool*) – apply compression to the data

merge (*field*)

Merges the receiver with given field together. Both fields should be on different parts of the domain (can also overlap), but should refer to same underlying discretization, otherwise unpredictable results can occur.

Parameters **field** (`Field`) – given field to merge with.

setValue (*componentID*, *value*)

Sets the value associated with a given component (vertex or integration point on a cell).

Parameters

- **componentID** (*tuple*) – A tuple identifying a component: vertex (vertexID,) or integration point (CellID, IPID)
- **value** (*tuple*) – Value to be set for a given component, should have the same units as receiver

---

Note: If a mesh has mapping attached (a mesh view) then we have to remember value locally and record change. The source field values are updated after commit() method is invoked.

---

toHdf5 (*fileName*, *group='component1/part1'*)

Dump field to HDF5, in a simple format suitable for interoperability (TODO: document).

Parameters

- **fileName** (*str*) – HDF5 file
- **group** (*str*) – HDF5 group the data will be saved under.

The HDF hierarchy is like this:

```
group
  |
  +--- mesh_01 {hash=25aa0aa04457}
  |      +--- [vertex_coords]
  |      +--- [cell_types]
  |      \--- [cell_vertices]
  +--- mesh_02 {hash=17809e2b86ea}
  |      +--- [vertex_coords]
  |      +--- [cell_types]
  |      \--- [cell_vertices]
  +--- ...
  +--- field_01
  |      +--- -> mesh_01
  |      \--- [vertex_values]
  +--- field_02
  |      +--- -> mesh_01
  |      \--- [vertex_values]
  +--- field_03
  |      +--- -> mesh_02
  |      \--- [cell_values]
  \--- ...
```

where `plain` names are HDF (sub)groups, `[bracketed]` names are datasets, `{name=value}` are HDF attributes, `->` prefix indicated HDF5 hardlink (transparent to the user); numerical suffixes (`_01`, ...) are auto-allocated. Mesh objects are hardlinked using HDF5 hardlinks if an identical mesh is already stored in the group, based on hexdigest of its full data.

Note: This method has not been tested yet. The format is subject to future changes.

---

toVTK2 (*fileName*, *format='ascii'*)

Save the instance as Unstructured Grid in VTK2 format (`.vtk`).

Parameters

- **fileName** (*str*) – where to save
- **format** (*str*) – one of `ascii` or `binary`

toVTK3 (*fileName*, ***kw*)

Save the instance as Unstructured Grid in VTK3 format (.vtu). This is a simple proxy for calling *manyToVTK3* with the instance as the only field to be saved. If multiple fields with identical mesh are to be saved in VTK3, use *manyToVTK3* directly.

Parameters

- **fileName** – output file name
- ****kw** – passed to *manyToVTK3*

class mupif.Field.**FieldType**

Bases: future.types.newobject.newobject

Represent the supported values of FieldType, i.e. FT_vertexBased or FT_cellBased.

FT_cellBased = 2

FT_vertexBased = 1

mupif.Function module

class mupif.Function.**Function** (*funcID*, *objectID=0*)

Bases: future.types.newobject.newobject

Represents a function.

Function is an object defined by mathematical expression. Function can depend on spatial position and time. Derived classes should implement evaluate service by providing a corresponding expression.

Example: f(x,t)=sin(2*3.14159265*x(1)/10.)

__init__ (*funcID*, *objectID=0*)

Initializes the function.

Parameters

- **funcID** (FunctionID) – function ID, e.g. FuncID_ProbabilityDistribution
- **objectID** (*int*) – Optional ID of associated subdomain, default 0

evaluate (*d*)

Evaluates the function for given parameters packed as a dictionary.

A dictionary is container type that can store any number of Python objects, including other container types. Dictionaries consist of pairs (called items) of keys and their corresponding values.

Example: d={'x':(1,2,3), 't':0.005} initializes dictionary contaning tuple (vector) under 'x' key, double value 0.005 under 't' key. Some common keys: 'x': position vector 't': time

Parameters **d** (*dictionary*) – Dictionaty containing function arguments (number and type depends on particular function)

Returns Function value evaluated at given position and time

Return type int, float, tuple

getID ()

Obtain function's ID.

Returns Returns receiver's ID.

Return type int

getObjectID()
Get optional ID of associated subdomain.

Returns Returns receiver's object ID,

Return type int

mupif.IntegrationRule module

class `mupif.IntegrationRule.`**GaussIntegrationRule**
Bases: *mupif.IntegrationRule.IntegrationRule*

Gauss integration rule.

getIntegrationPoints(*cgt*, *npt*)
See *IntegrationRule.getIntegrationPoints()*.

getRequiredNumberOfPoints(*cgt*, *order*)
See *IntegrationRule.getRequiredNumberOfPoints()*.

class `mupif.IntegrationRule.`**IntegrationRule**
Bases: `future.types.newobject.newobject`

Represent integration rule to be used on cells.

__init__()

getIntegrationPoints(*cgt*, *npt*)
Returns a list of integration points and corresponding weights.

Parameters

- **cgt** (`CellGeometryType`) – Type of underlying cell geometry (e.g. linear triangle CGT_TRIANGLE_1)
- **npt** (*int*) – Number of desired integration points

Returns A list of tuples containing natural coordinates of integration point and weights, i.e. [((c1_ksi, c1_eta), weight1), ((c2_ksi, c2_eta), weight2)]

Return type a list of tuples

getRequiredNumberOfPoints(*cgt*, *order*)
Returns required number of integration points to exactly integrate polynomial of order approxOrder on a given cell type.

Parameters

- **cgt** (`CellGeometryType`) – Type of underlying cell geometry (e.g. linear triangle CGT_TRIANGLE_1)
- **order** (*int*) – Target polynomial order

mupif.JobManager module

exception `mupif.JobManager.`**JobManException**
Bases: `exceptions.Exception`

This class serves as a base class for exceptions thrown by the job manager.

exception `mupif.JobManager.`**`JobManNoResourcesException`**

Bases: *`mupif.JobManager.JobManException`*

This class is thrown when there are no more available resources.

class `mupif.JobManager.`**`JobManager`**(*appName*, *jobManWorkDir*, *maxJobs=1*)

Bases: `future.types.newobject.newobject`

An abstract (base) class representing a job manager. The purpose of the job manager is the following:

- To allocate and register the new instance of application (called job)
- To query the status of job
- To cancel the given job
- To register its interface to pyro name server

`__init__`(*appName*, *jobManWorkDir*, *maxJobs=1*)

Constructor. Initializes the receiver.

Parameters

- **`appName`** (*`str`*) – Name of receiver (used also by NS)
- **`jobManWorkDir`** (*`str`*) – Absolute path for storing data, if necessary
- **`maxJobs`** (*`int`*) – Maximum number of jobs to run simultaneously

`allocateJob`(*user*, *natPort*)

Allocates a new job.

Parameters

- **`user`** (*`str`*) – user name
- **`natPort`** (*`int`*) – NAT port used in ssh tunnel

Returns tuple (error code, None). errCode = (JOBMAN_OK, JOBMAN_ERR, JOBMAN_NO_RESOURCES). JOBMAN_OK indicates sucessfull allocation and JobID contains the PYRO name, under which the new instance is registered (composed of application name and a job number (allocated by jobmanager), ie, Miccress23). JOBMAN_ERR indicates an internal error, JOBMAN_NO_RESOURCES means that job manager is not able to allocate new instance of application (no more recources available)

Return type tuple

Except JobManException when allocation of new job failed

`getJobStatus`(*jobID*)

Returns the status of the job.

Parameters **`jobID`** (*`str`*) – jobID

`getNSName`()

`getPyroFile`(*jobID*, *filename*, *buffSize=1024*)

Returns the (remote) PyroFile representation of given file. To create local copy of file represented by PyroFile, use PyroUtil.downloadPyroFile, see `PyroUtil.downloadPyroFile()`

Parameters

- **`jobID`** (*`str`*) – job identifier (jobID)
- **`filename`** (*`str`*) – source file name (on remote server). The filename should contain only base filename, not a path, which is determined by jobManager based on jobID.

Returns PyroFile representation of given file

Return type *PyroFile*

getStatus()

registerPyro(*daemon*, *ns*, *uri*, *appName*, *externalDaemon*)
Possibility to register the Pyro daemon and nameserver.

Parameters

- **pyroDaemon** (*Pyro4.Daemon*) – Optional pyro daemon
- **pyroNS** (*Pyro4.naming.Nameserver*) – Optional nameserver
- **PyroURI** (*string*) – Optional URI of receiver
- **externalDaemon** (*bool*) – Optional parameter when damon was allocated externally.

terminate()
Terminates job manager itself.

terminateJob(*jobID*)
Terminates the given job, frees the associated recources.

Parameters **jobID** (*str*) – jobID

Returns JOBMAN_OK indicates sucessfull termination, JOBMAN_ERR means internal error

Return type str

uploadFile(*jobID*, *filename*, *pyroFile*)
Uploads the given file to application server, files are uploaded to dedicated jobID directory :param str jobID: jobID :param str filename: target file name :param PyroFile pyroFile: source pyroFile

class mupif.JobManager.**RemoteJobManager**(*decoratee*, *sshTunnel=None*)
Bases: `future.types.newobject.newobject`

Remote jobManager instances are normally represented by auto generated pyro proxy. However, when ssh tunneled connection is established to connect to remote job manager, its instance must be properly terminated. This class is a decorator around pyro proxy object represeting jobManager storing the reference to the ssh tunnel established. Note in case of VPN or direct (plain) connection, the plain Pyro proxy should be used.

The attribute could not be injected into remote instance (using proxy) as the termination has to be done from local computer, where the ssh tunnel has been created. Also different connections (proxies) to the same jobManager can exist.

terminate()
Terminates the application. Terminates the allocated job at jobManager

mupif.Localizer module

class mupif.Localizer.**Localizer**
Bases: `future.types.newobject.newobject`

A Localizer is an abstract class representing an algorithm used to partition space and quicly localize the contained objects.

delete(*item*)
Deletes the given object from Localizer data structure.

Parameters **item** (*object*) – Object to be removed

evaluate(*functor*)

Returns the list of all objects for which the functor is satisfied.

Parameters **functor** (`object`) – The functor is a class which defines two methods: giveBBox() which returns an initial functor bbox and evaluate(obj) which should return True if the functor is satisfied for a given object.

Returns List of all objects

Return type tuple

giveItemsInBBox(*bbox*)

Parameters **bbox** (`BBox`) – Bounding box

Returns List of all objects which bbox contains and intersects

Return type tuple

insert(*item*)

Inserts given object to Localizer. Object is assume to provide giveBBox() method returning bounding volume if itself.

Parameters **item** (`object`) – Inserted object

mupif.Mesh module

class `mupif.Mesh.`**Mesh**

Bases: `future.types.newobject.newobject`

Abstract representation of a computational domain. Mesh contains computational cells and vertices. Derived classes represent structured, unstructured FE grids, FV grids, etc.

Mesh is assumed to provide a suitable instance of cell and vertex localizers.

__init__()

asHdf5Object(*parentgroup*, *newgroup*)

Return the instance as HDF5 object. Complementary to *`makeFromHdf5Object`* which will restore the instance from that data.

asVtkUnstructuredGrid()

Return object as a vtk.vtkUnstructuredMesh instance.

Note: This method uses the compiled vtk module (which is a wrapper atop the c++ VTK library) – in contrast to *`UnstructuredMesh.getVTKRepresentation`*, which uses the pyvtk module (python-only implementation of VTK i/o supporting only VTK File Format version 2).

cellLabel2Number(*label*)

Returns local cell number corresponding to given label. If no label found, throws an exception.

Parameters **label** (`str`) – Cell label

Returns Cell number

Return type int

Except Label not found

cells()

Iterator over cells.

Returns Iterator over cells

Return type *MeshIterator*

copy()

Returns a copy of the receiver.

Returns A copy of the receiver

Return type Copy of the receiver, e.g. Mesh

Note: DeepCopy will not work, as individual cells contain mesh link attributes, leading to underlying mesh duplication in every cell!

---

dumpToLocalFile(*fileName*, *protocol=2*)

Dump Mesh to a file using a Pickle serialization module.

Parameters

- **fileName** (*str*) – File name
- **protocol** (*int*) – Used protocol - 0=ASCII, 1=old binary, 2=new binary

getCell(*i*)

Returns i-th cell.

Parameters **i** (*int*) – i-th cell

Returns cell

Return type *Cell*

getCells()

Return all cells as 2x numpy.array; each i-th row contains vertex indices for i-th cell. Does in 2 passes, first to determine maximum number of vertices per cell (to shape the field accordingly). For cells with less vertices than the maximum, excess ones are assigned the invalid value of -1.

Returns (cell_types,cell_vertices)

Return type (numpy.array,numpy.array)

Note: This method has not been tested yet.

---

getMapping()

Get mesh mapping.

Returns The mapping associated to a mesh

Return type defined by API

getNumberOfCells()

Return number of cells (finite elements).

Returns The number of Cells

Return type int

getNumberOfVertices()

Get number of vertices (nodes).

Returns Number of Vertices

Return type int

getVertex (*i*)
Returns i-th vertex.

Parameters **i** (*int*) – i-th vertex

Returns vertex

Return type *Vertex*

getVertices ()
Return all vertex coordinates as 2D (Nx3) numpy.array; each i-th row contains 3d coordinates of the i-th vertex.

Returns vertices

Return type numpy.array

Note: This method has not been tested yet.

internalArraysDigest ()
Internal function returning hash digest of all internal data, for the purposes of identity test.

classmethod **loadFromLocalFile** (*fileName*)
Alternative constructor which loads an instance from a Pickle module.

Parameters **fileName** (*str*) – File name

Returns Returns Mesh instance

Return type *Mesh*

static **makeFromHdf5Object** (*h5obj*)
Create new `Mesh` instance from given hdf5 object. Complementary to `asHdf5Object`.

Returns new instance

Return type `Mesh` or its subclass

vertexLabel2Number (*label*)
Returns local vertex number corresponding to given label. If no label found, throws an exception.

Parameters **label** (*str*) – Vertex label

Returns Vertex number

Return type int

Except Label not found

vertices ()
Iterator over vertices.

Returns Iterator over vertices

Return type *MeshIterator*

class mupif.Mesh.**MeshIterator** (*mesh, type*)
Bases: `future.types.newobject.newobject`

Class implementing iterator on Mesh components (vertices, cells).

__init__ (*mesh, type*)
Constructor.

Parameters

- **mesh** (Mesh) – Given mesh
- **type** (*str*) – Type of mesh, e.g. VERTICES or CELLS

`__iter__`()

Returns Itself

Return type *MeshIterator*

`__next__`()

Returns Returns next Mesh components.

Return type *MeshIterator*

next()

Python 2.x compatibility, see `MeshIterator.__next__()`

class mupif.Mesh.**UnstructuredMesh**

Bases: `mupif.Mesh.Mesh`

Represents unstructured mesh. Maintains the list of vertices and cells.

The class contains:

- vertexList: list of vertices
- cellList: list of interpolation cells
- vertexOctree: vertex spatial localizer
- cellOctree: cell spatial localizer
- vertexDict: vertex dictionary
- cellDict: cell dictionary

`__init__`()

Constructor.

`__buildVertexLabelMap__`()

Create a custom dictionary between vertex's label and Vertex instance.

`__buildCellLabelMap__`()

Create a custom dictionary between cell's label and Cell instance.

cellLabel2Number(*label*)

See `Mesh.cellLabel2Number()`

copy()

See `Mesh.copy()`

getCell(*i*)

See `Mesh.getCell()`

getNumberOfCells()

See `Mesh.getNumberOfCells()`

getNumberOfVertices()

See `Mesh.getNumberOfVertices()`

getVTKRepresentation()

Get VTK representatnion of the mesh.

return: VTK representation of the receiver. Requires pyvtk module. :rtype: pyvtk.UnstructuredGrid

getVertex (*i*)
See *Mesh.getVertex()*

giveCellLocalizer ()
Get the cell localizer.

Returns Returns the cell localizer.

Return type *Octree*

giveVertexLocalizer ()

Returns Returns the vertex localizer.

Return type *Octree*

static **makeFromPyvtkUnstructuredGrid** (*ugr*)
Create a new instance of *UnstructuredMesh* based on pyvtk.UnstructuredGrid object. Cell types are mapped between pyvtk and mupif (supported: triangle, tetra, quad, hexahedron).

Parameters **ugr** – instance of pyvtk.UnstructuredGrid

Returns new instance of *UnstructuredMesh*

static **makeFromVtkUnstructuredGrid** (*ugrid*)
Create a new instance of *UnstructuredMesh* based on VTK's unstructured grid object. Cell types are mapped between VTK and mupif (supported: vtkTriangle, vtkQuadraticTriangle, vtkQuad, vtkTetra, vtkHexahedron).

Parameters **ugrid** – instance of vtk.vtkUnstructuredGrid

Returns new instance of *UnstructuredMesh*

merge (*mesh*)
Merges receiver with a given mesh. This is based on merging mesh entities (vertices, cells) based on their labels, as they refer to global IDs of each entity, that should be unique.

The procedure used here is based on creating a dictionary for every componenet from both meshes, where the key is component label so that the entities with the same ID could be easily identified.

Parameters **mesh** (Mesh) – Source mesh for merging

setup (*vertexList*, *cellList*)
Initializes the receicer according to given vertex and cell lists.

Parameters

- **vertexList** (*tuple*) – A tuple of vertices
- **cellList** (*tuple*) – A tuple of cells

vertexLabel2Number (*label*)
See *Mesh.vertexLabel2Number()*

mupif.MetadataKeys module

Definition of common metadata keys

mupif.MupifObject module

class `mupif.MupifObject.`**`MupifObject`**

Bases: `future.types.newobject.newobject`

An abstract class representing a base Mupif object.

The purpose of this class is to represent any mupif object; it introduce basic methods for getting and setting object metatdata.

`__init__``()`

Constructor. Initializes the object

`getMetadata``(`*key*`)`

Returns metadata associated to given key :param key: unique metadataID :return: metadata associated to key, throws TypeError if key does not exist :raises: TypeError

`hasMetadata``(`*key*`)`

Returns true if key defined :param key: unique metadataID :return: true if key defined, false otherwise :rtype: bool

`setMetadata``(`*key*, *val*`)`

Sets metadata associated to given key :param key: unique metadataID :param val: any type

`toJSON``()`

JSON serialization method :return: string

mupif.Octree module

class `mupif.Octree.`**`Octant`**`(`*octree*, *parent*, *origin*, *size*`)`

Bases: `future.types.newobject.newobject`

Defines Octree Octant: a cell containing either terminal data or its child octants. Octree is used to partition space by recursively subdividing the root cell (square or cube) into octants. Octants can be terminal (containing the data) or can be further subdivided into children octants. Each terminal octant contains the objects with bounding box within the octant.

`__init__``(`*octree*, *parent*, *origin*, *size*`)`

The contructor. Octant class contains:

- data: Container storing the indexed objects (cells, vertices, etc)
- children: Container storing the children octants (if not terminal).
- octree: Link to octree object
- parent: Link to parent Octant
- origin: Coordinates of Octant lower left corner
- size: Dimension of Octant

Parameters

- **`octree`** (`Octree`) – Link to octree object
- **`parent`** (`Octree`) – Link to parent Octant
- **`origin`** (*`tuple`*) – coordinates of octant lower left corner
- **`size`** (*`float`*) – Size (dimension) of receiver

childrenIJK()

Returns iterator over receiver children

Returns iterator over 3-tuples with child indices; functionally equivalent to 3 nested loops, a bit faster and more readable.

containsBBox(*_bbox*)

Returns True if BBox contains or intersects the receiver.

delete(*item*, *itemBBox=None*)

Deletes/removes the given object from receiver

Parameters

- **item** (`object`) – object to remove
- **itemBBox** (`BBox`) – Optional parameter to specify bounding box of the object to be removed

divide()

Divides the receiver locally, creating child octants.

evaluate(*functor*)

Evaluate the given functor on all containing objects. The functor should define getBBox() function to return functor bounding box. Only the objects within this bouding box will be processed. Functor should also define evaluate method accepting object as a parameter.

Parameters **functor** (`object`) – Functor

giveDepth()

Returns Returns the depth (the subdivision level) of the receiver (and its children)

giveItemsInBBox(*itemList*, *bbox*)

Returns the list of objects inside the given bounding box. Note: an object can be included several times, as can be assigned to several octants.

Parameters

- **itemList** (`list`) – list containing the objects matching the criteria
- **bbox** (`BBox`) – target bounding box

giveMyBBox()

Returns Receiver's BBox

Return type *BBox*

insert(*item*, *itemBBox=None*)

Insert given object into receiver container. Object is inserted only when its bounding box intersects the bounding box of the receiver. If the number of stored objects exceeds the limit, the receiver is adaptively refined and objects distributed to children octants.

Parameters

- **item** (`object`) – object to insert
- **itemBBox** (`BBox`) – Optional parameter determining the BBox of the object

isTerminal()

Returns True if octree is the terminal cell

class mupif.Octree.**Octree**(*origin*, *size*, *mask*)

Bases: *mupif.Localizer.Localizer*

An octree is used to partition space by recursively subdividing the root cell (square or cube) into octants. Octants can be terminal (containing the data) or can be further subdivided into children octants partitioning the parent. Each terminal octant contains the objects with bounding box within the octant. Octree contains at least one octant, called root octant, with geometry large enough to contain all potential objects. Such a partitiong can significantly speed up spatial serches on objects.

Each object that can be inserted is assumed to provide giveBBox() returning its bounding box.

Octree implementation supports 1D, 2D and 3D setting. This is controlled by Octree mask. Octree mask is a tuple containing 0 or 1 values. If corresponding mask value is nonzero, receiver is subdivided in corresponding coordinate direction.

__init__(*origin*, *size*, *mask*)

The constructor.

Parameters

- **origin** (*tuple*) – coordinates of lower left corner of the root octant.
- **size** (*float*) – dimension (size) of the root octant
- **mask** (*tuple*) – boolean tuple, where true values determine the coordinate indices in which octree octants are subdivided

delete(*item*)

Removes the given object from octree. See *Octant.delete()*

evaluate(*functor*)

Evaluate the given functor on all containing objects. See *Octant.evaluate()*

giveDepth()

See *Octant.giveDepth()*

giveItemsInBBox(*bbox*)

Returns the list of objects inside the given bounding box. See *Octant.giveItemsInBBox()*

insert(*item*)

Inserts given object into octree. See *Octant.insert()*

mupif.Property module

class mupif.Property.**ConstantProperty**(*value*, *propID*, *valueType*, *units*, *time=None*, *objectID=0*)

Bases: *mupif.Property.Property*

Property is a characteristic value of a problem, that does not depend on spatial variable, e.g. homogenized conductivity over the whole domain. Typically, properties are obtained by postprocessing results from lover scales by means of homogenization and are parameters of models at higher scales.

Property value can be of scalar, vector, or tensorial type. Property keeps its value, objectID, time and type.

__init__(*value*, *propID*, *valueType*, *units*, *time=None*, *objectID=0*)

Initializes the property.

Parameters

- **value** (*tuple*) – A tuple (array) representing property value
- **propID** (*PropertyID*) – Property ID

- **valueType** (ValueType) – Type of a property, i.e. scalar, vector, tensor. Tensor is by default a tuple of 9 values, being compatible with Field's tensor.
- **time** (*Physics.PhysicalQuantity*) – Time when property is evaluated. If None (default), no time dependence
- **units** (*Physics.PhysicalUnits or string*) – Property units or string
- **objectID** (*int*) – Optional ID of problem object/subdomain to which property is related, default = 0

convertToUnit (*unit*)

Change the unit and adjust the value such that the combination is equivalent to the original one. The new unit must be compatible with the previous unit of the object.

Parameters **unit** (*C{str}*) – a unit

Raises **TypeError** – if the unit string is not a know unit or a unit incompatible with the current one

dumpToLocalFile (*fileName*, *protocol=2*)

Dump Property to a file using Pickle module

Parameters

- **fileName** (*str*) – File name
- **protocol** (*int*) – Used protocol - 0=ASCII, 1=old binary, 2=new binary

getTime ()

Returns Receiver time

Return type PhysicalQuantity or None

getValue (*time=None*, ***kwargs*)

Returns the value of property in a tuple. :param Physics.PhysicalQuantity time: Time of property evaluation :param **kwargs: None.

Returns Property value as an array

Return type tuple

inUnitsOf (**units*)

Express the quantity in different units. If one unit is specified, a new PhysicalQuantity object is returned that expresses the quantity in that unit. If several units are specified, the return value is a tuple of PhysicalObject instances with with one element per unit such that the sum of all quantities in the tuple equals the original quantity and all the values except for the last one are integers. This is used to convert to irregular unit systems like hour/minute/second.

Parameters **units** (*C{str}*) – one units

Returns one physical quantity

Return type L{PhysicalQuantity} or C{tuple} of L{PhysicalQuantity}

Raises **TypeError** – if any of the specified units are not compatible with the original unit

classmethod loadFromLocalFile (*fileName*)

Alternative constructor from a Pickle module

Parameters **fileName** (*str*) – File name

Returns Returns Property instance

Return type *Property*

class mupif.Property.**Property**(*propID*, *valueType*, *units*, *objectID=0*)
Bases: *mupif.MupifObject.MupifObject*, *mupif.Physics.PhysicalQuantities.PhysicalQuantity*

Property is a characteristic value of a problem, that does not depend on spatial variable, e.g. homogenized conductivity over the whole domain. Typically, properties are obtained by postprocessing results from lover scales by means of homogenization and are parameters of models at higher scales.

Property value can be of scalar, vector, or tensorial type. Property keeps its value, objectID, time and type.

__init__(*propID*, *valueType*, *units*, *objectID=0*)
Initializes the property.

Parameters

- **value** (*tuple*) – A tuple (array) representing property value
- **propID** (PropertyID) – Property ID
- **valueType** (ValueType) – Type of a property, i.e. scalar, vector, tensor. Tensor is by default a tuple of 9 values, being compatible with Field's tensor.
- **time** (*Physics.PhysicalQuantity*) – Time
- **units** (*Physics.PhysicalUnits or string*) – Property units or string
- **objectID** (*int*) – Optional ID of problem object/subdomain to which property is related, default = 0

getObjectID()
Returns property objectID.

Returns Object's ID

Return type int

getPropertyID()
Returns type of property.

Returns Receiver's property ID

Return type *PropertyID*

getUnits()
Returns representation of property units.

Returns Returns receiver's units (Units)

Return type *PhysicalQuantity*

getValue(*time=None*, ***kwargs*)
Returns the value of property in a tuple. :param Physics.PhysicalQuantity time: Time of property evaluation :param **kwargs: Arbitrary keyword arguments,see documentation

of derived classes.

Returns Property value as an array

Return type tuple

getValueType()
Returns the value type of property.

Returns Property value type

Return type *mupif.PropertyID*

mupif.PyroFile module

class `mupif.PyroFile.`**`PyroFile`**(*filename*, *mode*, *buffsize=1024*, *compressFlag=False*)

Bases: `object`

Helper Pyro class providing an access to local file. It allows to receive/send the file content from/to remote site (using Pyro) in chunks of configured size.

`close`()

Closes the associated file handle.

`getChunk`()

Reads and returns next buffsize bytes from open (should be opened in read mode). The returned chunk may contain less bytes if not enouch data can be read, or can be empty if end-of-file is reached. :return: Returns next chunk of data read from the file :rtype: str

`getTerminalChunk`()

Reads and returns the terminal bytes from source. In case of of source without compression, an empty string should be returned, in case of compressed stream the termination sequence is returned (see zlib flush(Z_FINAL)) :rtype: str

`setBuffSize`(*buffSize*)

Allows to set the receiver buffer size. :param int buffSize: new buffer size

`setChunk`(*buffer*)

Writes the given chunk of data into the file, which should be opened in write mode.

Parameters **`buffer`** (*`str`*) – data chunk to append

`setCompressionFlag`()

Sets the compressionFlag to True

mupif.PyroUtil module

class `mupif.PyroUtil.`**`SSHContext`**(*userName='‘*, *sshClient='manual'*, *options='‘*, *sshHost='‘*)

Bases: `object`

Helper class to store ssh tunnel connection details. It is parameter to different methods (connectJobManager, allocateApplicationWithJobManager, etc.). When provided, the corresponding ssh tunnel connection is established and associated to proxy using decorator class to make sure it can be terminated properly.

`mupif.PyroUtil.`**`allocateApplicationWithJobManager`**(*ns*, *jobMan*, *natPort*, *hkey*, *sshContext=None*)

Request new application instance to be spawned by given jobManager.

Parameters

- **`ns`** (*`Pyro4.naming.Nameserver`*) – running name server
- **`jobManager`** (*`jobManager`*) – jobmanager to use
- **`natPort`** (*`int`*) – nat port on a local computer for ssh tunnel for the application
- **`hkey`** (*`str`*) – A password string
- **`sshContext`** (*`sshContext`*) – describing optional ssh tunnel connection detail

Returns Application instance

Return type *Application.RemoteApplication*

Raises **Exception** – if allocation of job fails

`mupif.PyroUtil.`**allocateNextApplication**(*ns*, *jobMan*, *natPort*, *sshContext=None*)

Request new application instance to be spawned by given jobManager

Parameters **ns** (*Pyro4.naming.Nameserver*) – running name server

:param jobManager jobmanager to use :param int natPort: nat port on a local computer for ssh tunnel for the application :param sshContext describing optional ssh tunnel connection detail

Returns Application instance

Return type *Application.RemoteApplication*

Raises **Exception** – if allocation of job fails

`mupif.PyroUtil.`**connectApp**(*ns*, *name*, *hkey=None*, *sshContext=None*)

Connects to a remote application, creates the ssh tunnel if necessary

Parameters

- **ns** (*Pyro4.naming.Nameserver*) – Instance of a nameServer
- **name** (*str*) – Name of the application to be connected to
- **hkey** (*str*) – A password string

Returns Application Decorator (docorating pyro proxy with ssh tunnel instance)

Return type Instance of an application decorator

Raises **Exception** – When cannot find registered server or Cannot connect to application

`mupif.PyroUtil.`**connectApplicationsViaClient**(*fromContext*, *fromApplication*, *toApplication*)

Create a reverse ssh tunnel so one server application can connect to another one.

Typically, steering_computer creates connection to server1 and server2. However, there is no direct link server1-server2 which is needed for Field operations (getField, setField). Assume a working connection server1-steering_computer on NAT port 6000. This function creates a tunnel steering_computer:6000 and server2:7000 so server2 has direct access to server1's data.

steering_computer / from server1:6000 to server2:7000

Parameters

- **fromContext** (`SSHContext`) – Remote application
- **fromApplication** (`Application`) – Application object from which we want to create a tunnel
- **toApplication** (`Application`) – Application object to which we want to create a tunnel

Returns Instance of sshTunnel class

Return type *sshTunnel*

`mupif.PyroUtil.`**connectJobManager**(*ns*, *jobManName*, *hkey=None*, *sshContext=None*)

Connect to jobManager described by given jobManRec and create an optional ssh tunnel

:param jobManName name under which jobmanager is registered on NS :param str hkey: A password string :param sshContext describing optional ssh tunnel connection detail

Returns (JobManager proxy, jobManager Tunnel)

Return type *JobManager.RemoteJobManager*

Raises **Exception** – if creation of a tunnel failed

mupif.PyroUtil.**connectNameServer**(*nshost*, *nsport*, *hkey*, *timeOut=3.0*)
Connects to a NameServer.

Parameters

- **nshost** (*str*) – IP address of nameServer
- **nsport** (*int*) – Nameserver port.
- **hkey** (*str*) – A password string
- **timeOut** (*float*) – Waiting time for response in seconds

Returns NameServer

Return type Pyro4.naming.Nameserver

Raises **Exception** – When can not connect to a LISTENING port of nameserver

mupif.PyroUtil.**downloadPyroFile**(*newLocalFileName*, *pyroFile*, *compressFlag=False*)
Allows to download remote file (pyro ile handle) to a local file.

Parameters

- **newLocalFileName** (*str*) – path to a new local file on a client.
- **pyroFile** (PyroFile) – representation of existing remote server's file
- **compressFlag** (*bool*) – will activate compression during data transfer (zlib)

mupif.PyroUtil.**downloadPyroFileFromServer**(*newLocalFileName*, *pyroFile*, *compressFlag=False*)
See :func:'downloadPyroFileFromServer'

mupif.PyroUtil.**getIPfromUri**(*uri*)
Returns IP address of the server hosting given URI, e.g. return 127.0.0.1 from string PYRO:obj_b178eed8e1994135adf9864725f1d50f@127.0.0.1:5555 :param str uri: URI from an object

Returns IP address

Return type string

mupif.PyroUtil.**getNATfromUri**(*uri*)
Return NAT port from URI, e.g. return 5555 from string PYRO:obj_b178eed8e1994135adf9864725f1d50f@127.0.0.1:5555

Parameters **uri** (*str*) – URI from an object

Returns NAT port number

Return type int

mupif.PyroUtil.**getNSAppName**(*jobname*, *appname*)
Get application name.

Parameters

- **jobname** (*str*) – Arbitrary string concatenated in the outut
- **appname** (*str*) – Arbitrary string concatenated in the outut

Returns String of concatenated arguments

Return type str

mupif.PyroUtil.**getNSConnectionInfo**(*ns*, *name*)
Returns component connection information stored in name server :return (host, port, nathost, natport) tuple :rtype: tuple

`mupif.PyroUtil.`**`getNSmetadata`**`(`*ns*, *name*`)`

Returns name server metadata for given entry identified by name :return entry metadata :rtype: list of strings

`mupif.PyroUtil.`**`getUserInfo`**`()`

Returns tuple containing (username, hostname)

Return type tuple of strings

`mupif.PyroUtil.`**`runAppServer`**`(`*server*, *port*, *nathost*, *natport*, *nshost*, *nsport*, *appName*, *hkey*, *app*, *daemon=None*`)`

Runs a simple application server

Parameters

- **server** (*str*) – Host name of the server (internal host name)
- **port** (*int*) – Port number on the server where daemon will listen (internal port number)
- **nathost** (*str*) – Hostname of the server as reported by nameserver, for secure ssh tunnel it should be set to 'localhost' (external host name)
- **natport** (*int*) – Server NAT port as reported by nameserver (external port)
- **nshost** (*str*) – Hostname of the computer running nameserver
- **nsport** (*int*) – Nameserver port
- **appName** (*str*) – Name of registered application
- **app** (*instance*) – Application instance
- **hkey** (*str*) – A password string
- **daemon** – Reference to already running daemon, if available. Optional parameter.

Raises **`Exception`** – if can not run Pyro4 daemon

`mupif.PyroUtil.`**`runDaemon`**`(`*host*, *port*, *nathost=None*, *natport=None*, *hkey=None*`)`

Runs a daemon without registering to a name server :param str(int) host: Host name where daemon runs. This is typically a localhost :param int port: Port number where daemon will listen (internal port number) :param str(int) nathost: Hostname of the server as reported by nameserver, for secure ssh tunnel it should be set to 'localhost' (external host name) :param int natport: Server NAT port, optional (external port) :param str hkey: A password string

:return Instance of the running daemon, None if a problem :rtype Pyro4.Daemon

`mupif.PyroUtil.`**`runJobManagerServer`**`(`*server*, *port*, *nathost*, *natport*, *nshost*, *nsport*, *appName*, *hkey*, *jobman*, *daemon=None*`)`

Registers and runs given jobManager server

Parameters

- **server** (*str*) – Host name of the server (internal host name)
- **port** (*int*) – Port number on the server where daemon will listen (internal port number)
- **nathost** (*str*) – Hostname of the server as reported by nameserver, for secure ssh tunnel it should be set to 'localhost' (external host name)
- **natport** (*int*) – Server NAT port as reported by nameserver (external port)
- **nshost** (*str*) – Hostname of the computer running nameserver
- **nsport** (*int*) – Nameserver port
- **appName** (*str*) – Name of job manager to be registered at nameserver

- **hkey** (*str*) – A password string
- **app** (*instance*) – Application instance
- **daemon** – Reference to already running daemon, if available. Optional parameter.

mupif.PyroUtil.**runServer**(*server, port, nathost, natport, nshost, nsport, appName, hkey, app, daemon=None, metadata=None*)
Runs a simple application server

Parameters

- **server** (*str*) – Host name of the server (internal host name)
- **port** (*int*) – Port number on the server where daemon will listen (internal port number)
- **nathost** (*str*) – Hostname of the server as reported by nameserver, for secure ssh tunnel it should be set to 'localhost' (external host name)
- **natport** (*int*) – Server NAT port as reported by nameserver (external port)
- **nshost** (*str*) – Hostname of the computer running nameserver
- **nsport** (*int*) – Nameserver port
- **appName** (*str*) – Name of registered application
- **app** (*instance*) – Application instance
- **hkey** (*str*) – A password string
- **daemon** – Reference to already running daemon, if available. Optional parameter.
- **metadata** – set of strings that will be the metadata tags associated with the object registration. See PyroUtil.py for valid tags. The metadata string "connection:server:port:nathost:natport" will be authomatically generated.

Raises Exception – if can not run Pyro4 daemon

class mupif.PyroUtil.**sshTunnel**(*remoteHost, userName, localPort, remotePort, sshClient='ssh', options='', sshHost='', Reverse=False*)
Bases: object

Helper class to represent established ssh tunnel. It defines terminate and __del__ method to ensure correct tunnel termination.

terminate()
Terminate the connection.

mupif.PyroUtil.**uploadPyroFile**(*clientFileName, pyroFile, hkey, size=1024, compressFlag=False*)
Allows to upload given local file to a remote location (represented by Pyro file hanfdle).

Parameters

- **clientFileName** (*str*) – path to existing local file on a client where we are
- **pyroFile** (PyroFile) – represenation of remote file, this file will be created
- **hkey** (*str*) – A password string
- **size** (*int*) – optional chunk size. The data are read and written in byte chunks of this size
- **compressFlag** (*bool*) – will activate compression during data transfer (zlib)

mupif.PyroUtil.**uploadPyroFileOnServer**(*clientFileName, pyroFile, size=1024, compressFlag=False*)
See :func:'downloadPyroFile'

mupif.RemoteAppRecord module

class `mupif.RemoteAppRecord.`**`RemoteAppRecord`**(*app*, *appTunnel*, *jobMan*, *jobManTunnel*, *jobID*)

Bases: `future.types.newobject.newobject`

Class keeping internal data on remote application. The data contain: * appTunnel: reference to application ssh tunnel * jobMan: reference to jobManager * jobManTunnel: reference to jobManager tunnel representation * jobID: jobID of application .. automethod:: __init__

`appendNextApplication`(*app*, *appTunnel*, *jobID*)

Append next application on existing instance :param Application app: application instance :param subprocess.Popen appTunnel: ssh tunnel subprocess representing ssh tunnel to application process :param string jobID: application jobID

`getApplication`(*num=0*)

Returns application instance :param int num: number of application, default 0 :return: Instance of Application

`getApplicationUri`(*num=0*)

Returns application uri :param int num: number of application, default 0 :return: uri

`getJobID`(*num=0*)

`getJobManager`()

`terminateAll`()

Terminates all remote applications in app[] including their ssh tunnels. Terminates also jobManager and the associated ssh tunnel.

`terminateApp`(*num*)

Terminates app[num] and its ssh tunnel. Job manager and its tunnel remains untouched. :param int num: number of application

mupif.SimpleJobManager module

class `mupif.SimpleJobManager.`**`SimpleJobManager`**(*daemon*, *ns*, *appAPIClass*, *appName*, *jobManWorkDir*, *maxJobs=1*)

Bases: *`mupif.JobManager.JobManager`*

Simple job manager using Pyro thread pool based server. Requires Pyro servertype=thread pool based (SERVERTYPE config item). This is the default value. For the thread pool server the amount of worker threads to be spawned is configured using THREADPOOL_SIZE config item (default value set to 16).

However, dee to GIL (Global Interpreter Lock of python the actual level of achievable concurency is low. The threads created from a single python context are executed sequentially. This implementation is suitable only for servers with a low workload.

`__init__`(*daemon*, *ns*, *appAPIClass*, *appName*, *jobManWorkDir*, *maxJobs=1*)

Constructor.

Parameters

- **daemon** (*`Pyro4.Daemon`*) – running daemon for SimpleJobManager
- **ns** (*`Pyro4.naming.Nameserver`*) – running name server
- **appAPIClass** (`Application`) – application class
- **appName** (*`str`*) – application name

- **jobManWorkDir** (*str*) – see `JobManager.__init__()`
- **maxJobs** (*int*) – see `JobManager.__init__()`

allocateJob (*user*, *natPort*)
Allocates a new job.

See `JobManager.allocateJob()`

Except unable to start a thread, no more resources

getApplicationSignature ()

Returns application name

Return type str

getStatus ()
Returns a list of tuples for all running jobIDs :return: a list of tuples (jobID, running time, user) :rtype: a list of (str, float, str)

terminateJob (*jobID*)
Terminates the given job, frees the associated recources.

See `JobMSimpleJobManageranager.terminateJob()`

class mupif.SimpleJobManager.**SimpleJobManager2** (*daemon*, *ns*, *appAPIClass*, *appName*, *portRange*, *jobManWorkDir*, *serverConfigPath*, *serverConfigFile*, *serverConfigMode*, *jobMan2CmdPath*, *maxJobs=1*, *jobMancmdCommPort=10000*)

Bases: *mupif.JobManager.JobManager*

Simple job manager 2. This implementation avoids the problem of GIL lock by running applicaton server under new process with its own daemon.

__init__ (*daemon*, *ns*, *appAPIClass*, *appName*, *portRange*, *jobManWorkDir*, *serverConfigPath*, *serverConfigFile*, *serverConfigMode*, *jobMan2CmdPath*, *maxJobs=1*, *jobMancmdCommPort=10000*)
Constructor.

See *SimpleJobManager.__init__()* :param tuple portRange: start and end ports for jobs which will be allocated by a job manager :param str serverConfigFile: path to serverConfig file :param str jobMan2CmdPath: path to JobMan2cmd.py

Parameters

- **jobMancmdCommPort** (*int*) – optional communication port to communicate with jobman2cmd
- **configFile** (*str*) – path to server config file

allocateJob (*user*, *natPort*)
Allocates a new job.

See `JobManager.allocateJob()` :except: unable to start a thread, no more resources

getApplicationSignature ()
See *SimpleJobManager.getApplicationSignature()*

getPyroFile (*jobID*, *filename*, *mode='r'*, *buffSize=1024*)
See `JobManager.getPyroFile()`

getStatus()
See `JobManager.getStatus()`

terminate()
Terminates job manager itself.

terminateJob(*jobID*)
Terminates the given job, frees the associated recources.

See `JobManager.terminateJob()`

uploadFile(*jobID*, *filename*, *pyroFile*)
See `JobManager.uploadFile()`

mupif.TimeStep module

class `mupif.TimeStep.`**TimeStep**(*t*, *dt*, *targetTime*, *units=None*, *n=1*)
Bases: `future.types.newobject.newobject`

Class representing a time step. The folowing attributes are used to characterize a time step:

|l- - -|l(time-dt)- - - i-th time step(dt) - - -|l(time)- - - -|l- - - -|l(targetTime)

Note: Individual models (applications) assemble theit governing equations at specific time, called asssembly-Time, this time is reported by individual models. For explicit model, asssembly time is equal to timeStep.time-timestep.dt, for fully implicit model, assembly time is equal to timeStep.time

__init__(*t*, *dt*, *targetTime*, *units=None*, *n=1*)
Initializes time step.

Parameters

- **t** (*float or Physics.PhysicalQuantity*) – Time(time at the end of time step)
- **dt** (*float or Physics.PhysicalQuantity*) – Step length (time increment), type depends on 'units'
- **targetTime** (*float or Physics.PhysicalQuantity. targetTime is not related to particular time step rather to the material model (load duration, relaxation spectra etc.)*) – target simulation time (time at the end of simulation, not of a single TimeStep)
- **units** (*Physics.PhysicalUnit*) – optional units for t, dt, targetTime if given as float values
- **n** (*int*) – Optional, solution time step number, default = 1

getNumber()

Returns Receiver's solution step number

Return type int

getTargetTime()

Returns Target time

Return type Physics.PhysicalQuantity

getTime()

Returns Time

Return type Physics.PhysicalQuantity

getTimeIncrement()

Returns Time increment

Return type Physics.PhysicalQuantity

mupif.Timer module

class mupif.Timer.**Timer**

Bases: `future.types.newobject.newobject`

Class for measuring time.

__enter__()

Remembers time at calling this function.

__exit__(**args*)

Remembers time at calling this function and calculates the difference to __enter__().

mupif.Util module

mupif.Util.**NoneOrInt**(*arg*)

mupif.Util.**changeRootLogger**(*newLoggerName*)

Change root logger by giving a new file name. Useful in parallel processes on a single machine.

Returns Nothing

mupif.Util.**getParentParser**()

Parent parser for controling running mode. Used in MuPIF's examples. Mode 0-local (default), 1-ssh, 2-VPN with option -m.

Returns parent parser object

Return type argparse object

mupif.Util.**quadratic_real**(*a*, *b*, *c*)

Finds real roots of quadratic equation: ax^2 + bx + c = 0. By substituting x = y-t and t = a/2, the equation reduces to y^2 + (b-t^2) = 0 which has easy solution y = +/-sqrt(t^2-b)

Parameters

- **a** (*float*) – Parameter from quadratic equation
- **b** (*float*) – Parameter from quadratic equation
- **c** (*float*) – Parameter from quadratic equation

Returns Two real roots if they exist

Return type tuple

mupif.Util.**setupLogger**(*fileName*, *level=10*)

Set up a logger which prints messages on the screen and simultaneously saves them to a file. The file has the suffix '.log' after a loggerName.

Parameters

- **fileName** (*str*) – file name, the suffix '.log' is appended.

- **level** (*object*) – logging level. Allowed values are CRITICAL, ERROR, WARNING, INFO, DEBUG, NOTSET

Return type logger instance

mupif.ValueType module

Enumeration defining supported types of field and property values, e.g. scalar, vector, tensor

`mupif.ValueType.`**`fromNumberOfComponents`**`(`*i*`)`

Parameters **i** (*int*) – number of components

Returns value type corresponding to the number of components

RuntimeError is raised if *i* does not match any value known.

mupif.Vertex module

class `mupif.Vertex.`**`Vertex`**`(`*number*, *label*, *coords=None*`)`

Bases: `future.types.newobject.newobject`

Represent a vertex. Vertices define the geometry of interpolation cells. Vertex is characterized by its position, number and label. Vertex number is locally assigned number, while label is a unique number referring to source application.

`__init__``(`*number*, *label*, *coords=None*`)`

Initializes the vertex.

Parameters

- **number** (*int*) – Local vertex number
- **label** (*int*) – Vertex label
- **coords** (*tuple*) – 3D position vector of a vertex

`__repr__``()`

Returns Receiver's number, label, coordinates

Return type string

`getBBox``()`

Returns Receiver's bounding-box (containing only one point)

Return type *mupif.BBox.BBox*

`getCoordinates``()`

Returns Receiver's coordinates

Return type tuple

`getNumber``()`

Returns Number of the instance

Return type int

mupif.VtkReader2 module

`mupif.VtkReader2.`**`patched_polydata_fromfile`**(*f*, *self*)
Use VtkData(<filename>).

`mupif.VtkReader2.`**`patched_scalars_fromfile`**(*f*, *n*, *sl*)

`mupif.VtkReader2.`**`pyvtk_monkeypatch`**()
Apply monkey-patches to work around https://github.com/pearu/pyvtk/wiki/unexpectedEOF in pyvtk without changing the source code.

`mupif.VtkReader2.`**`readField`**(*mesh*, *Data*, *fieldID*, *units*, *time*, *name*, *filename*, *type*)

Parameters

- **mesh** (`Mesh`) – Source mesh
- **Data** (*vtkData*) – vtkData obtained by pyvtk
- **fieldID** (`FieldID`) – Field type (displacement, strain, temperature ...)
- **units** (`PhysicalUnit`) – field units
- **time** (`PhysicalQuantity`) – time
- **name** (*str*) – name of the field to visualize
- **type** (*int*) – type of value of the field (1:Scalar, 3:Vector, 6:Tensor)

Returns Field of unknowns

Return type *Field*

`mupif.VtkReader2.`**`readMesh`**(*numNodes*, *nx*, *ny*, *nz*, *coords*)
Reads structured 3D mesh

Parameters

- **numNodes** (*int*) – Number of nodes
- **nx** (*int*) – Number of elements in x direction
- **ny** (*int*) – Number of elements in y direction
- **nz** (*int*) – Number of elements in z direction
- **coords** (*tuple*) – Coordinates for each nodes

Returns Mesh

Return type *Mesh*

mupif.Workflow module

class `mupif.Workflow.`**`Workflow`**(*file=''*, *workdir=''*, *targetTime=PhysicalQuantity(0.0, 's')*)
Bases: *mupif.Application.Application*

An abstract class representing a workflow and its interface (API).

The purpose of this class is to represent a workflow, its abstract services for data exchange and steering. This interface has to be implemented/provided by any workflow. The Workflow class inherits from Application allowing to treat any workflow as model(application) in high-level workflow.

__init__ (*file='', workdir='', targetTime=PhysicalQuantity(0.0, 's')*)
Constructor. Initializes the workflow

Parameters

- **file** (*str*) – Name of file
- **workdir** (*str*) – Optional parameter for working directory
- **targetTime** (PhysicalQuantity) – target simulation time

getAPIVersion ()

Returns Returns the supported API version

Return type str, int

getApplicationSignature ()
Get application signature.

Returns Returns the application identification

Return type str

solve (*runInBackground=False*)
Solves the workflow.

The default implementation solves the problem in series of time steps using solveStep method (inheritted) until the final time is reached.

Parameters **runInBackground** (*bool*) – optional argument, default False. If True, the solution will run in background (in separate thread or remotely).

mupif.fieldID module

class mupif.fieldID.**FieldID**
Bases: enum.IntEnum

This class represent the supported values of field IDs, e.g. displacement, strain, temperature. Immutable class Enum allows accessing members by .name and .value methods

FID_Concentration = <FieldID.FID_Concentration: 6>

FID_Displacement = <FieldID.FID_Displacement: 1>

FID_Humidity = <FieldID.FID_Humidity: 5>

FID_Material_number = <FieldID.FID_Material_number: 9>

FID_Strain = <FieldID.FID_Strain: 2>

FID_Stress = <FieldID.FID_Stress: 3>

FID_Temperature = <FieldID.FID_Temperature: 4>

FID_Thermal_absorption_surface = <FieldID.FID_Thermal_absorption_surface: 8>

FID_Thermal_absorption_volume = <FieldID.FID_Thermal_absorption_volume: 7>

mupif.functionID module

class `mupif.functionID.`**`FunctionID`**

Bases: `enum.IntEnum`

This classenumeration represent the supported values of FunctionID, e.g. FuncID_ProbabilityDistribution

`FuncID_ProbabilityDistribution` = <FunctionID.FuncID_ProbabilityDistribution: 1>

mupif.operatorUtil module

class `mupif.operatorUtil.`**`OperatorEMailInteraction`**(*From*, *To*, *smtpHost*, *smtpUser=''*, *smtpPsswd=''*, *smtpSSL=False*, *smtpTLS=False*, *smtpPort=25*, *imapHost=''*, *imapUser=''*, *imapPsswd=''*, *imapPort=993*, *imapSSL=True*)

Bases: `mupif.operatorUtil.OperatorInteraction`

Constructor setting up communication channels. @param From(str): email address where the response will be send to @param To(str): email address of the operator @param smtpHost(str): internet address of SMTP server for sending a message @param smtpUser(str): username for authentication on SMTP server @param smtpPsswd(str): optional string of SMTP authentication. If empty, it will ask @param smtpSSL(bool): if SSL encryption on STMP server should be used @param smtpTLS(bool): if TLS mode of IMAP server should be used @param smtpPort(int): port of SMTP server @param imapHost(str): IMAP server where the response is stored @param imapUser(str): IMAP user where the response is stored @param imapPsswd(str): optional string of IMAP user's password. If empty, it will ask @param imapPort(int): port of IMAP server @param imapSSL(bool): if SSL encryption on IMAP server should be used

`checkOperatorResponse`(*workflowID*, *jobID*)

`contactOperator`(*workflowID*, *jobID*, *msgBody*)

class `mupif.operatorUtil.`**`OperatorInteraction`**

Contact operator. @param workflowID(str): unique workflow ID @param jobID(str): unique jobID @param msgBody(str): message to operator. Recomended to store all paramaters into dictionary and convert dictionary into json string representation.

`checkOperatorResponse`(*workflowID*, *jobID*)

`contactOperator`(*workflowID*, *jobID*, *msgBody*)

`mupif.operatorUtil.`**`log`** = <logging.RootLogger object>

Generic class to represent interaction with an operator. Derived classes implement different communication channels.

mupif.propertyID module

Module defining PropertyID as enumeration, e.g. concentration, velocity. class Enum allows accessing members by .name and .value

class `mupif.propertyID.`**`PropertyID`**

Bases: `enum.IntEnum`

Enumeration class defining Property IDs. These are used to uniquely determine the canonical keywords identifiing individual properties.

`PID_AsorptionSpectrum` = <PropertyID.PID_AsorptionSpectrum: 26>

`PID_ChipSpectrum` = <PropertyID.PID_ChipSpectrum: 17>

`PID_Concentration` = <PropertyID.PID_Concentration: 1>

`PID_CumulativeConcentration` = <PropertyID.PID_CumulativeConcentration: 2>

`PID_Demo_Integral` = <PropertyID.PID_Demo_Integral: 9992>

`PID_Demo_Max` = <PropertyID.PID_Demo_Max: 9991>

`PID_Demo_Min` = <PropertyID.PID_Demo_Min: 9990>

`PID_Demo_Value` = <PropertyID.PID_Demo_Value: 9994>

`PID_Demo_Volume` = <PropertyID.PID_Demo_Volume: 9993>

`PID_EmissionSpectrum` = <PropertyID.PID_EmissionSpectrum: 24>

`PID_ExcitationSpectrum` = <PropertyID.PID_ExcitationSpectrum: 25>

`PID_InverseCumulativeDist` = <PropertyID.PID_InverseCumulativeDist: 28>

`PID_KPI01` = <PropertyID.PID_KPI01: 9996>

`PID_LEDCCT` = <PropertyID.PID_LEDCCT: 20>

`PID_LEDColor_x` = <PropertyID.PID_LEDColor_x: 18>

`PID_LEDColor_y` = <PropertyID.PID_LEDColor_y: 19>

`PID_LEDRadiantPower` = <PropertyID.PID_LEDRadiantPower: 21>

`PID_LEDSpectrum` = <PropertyID.PID_LEDSpectrum: 16>

`PID_NumberOfFluorescentParticles` = <PropertyID.PID_NumberOfFluorescentParticles: 29>

`PID_NumberOfRays` = <PropertyID.PID_NumberOfRays: 15>

`PID_ParticleMu` = <PropertyID.PID_ParticleMu: 30>

`PID_ParticleNumberDensity` = <PropertyID.PID_ParticleNumberDensity: 22>

`PID_ParticleRefractiveIndex` = <PropertyID.PID_ParticleRefractiveIndex: 23>

`PID_ParticleSigma` = <PropertyID.PID_ParticleSigma: 31>

`PID_PhosphorEfficiency` = <PropertyID.PID_PhosphorEfficiency: 32>

`PID_RefractiveIndex` = <PropertyID.PID_RefractiveIndex: 14>

`PID_ScatteringCrossSections` = <PropertyID.PID_ScatteringCrossSections: 27>

`PID_UserTimeStep` = <PropertyID.PID_UserTimeStep: 9995>

`PID_Velocity` = <PropertyID.PID_Velocity: 3>

`PID_conductivity_green_phosphor` = <PropertyID.PID_conductivity_green_phosphor: 9>

`PID_conductivity_red_phosphor` = <PropertyID.PID_conductivity_red_phosphor: 8>

`PID_effective_conductivity` = <PropertyID.PID_effective_conductivity: 5>

`PID_mean_radius_green_phosphor` = <PropertyID.PID_mean_radius_green_phosphor: 11>

`PID_mean_radius_red_phosphor` = <PropertyID.PID_mean_radius_red_phosphor: 10>

`PID_standard_deviation_green_phosphor` = <PropertyID.PID_standard_deviation_green_phosphor: 13>

`PID_standard_deviation_red_phosphor` = <PropertyID.PID_standard_deviation_red_phosphor: 12>

PID_transient_simulation_time = <PropertyID.PID_transient_simulation_time: 4>

PID_volume_fraction_green_phosphor = <PropertyID.PID_volume_fraction_green_phosphor: 7>

PID_volume_fraction_red_phosphor = <PropertyID.PID_volume_fraction_red_phosphor: 6>

Module contents

class mupif.**FieldID**

Bases: enum.IntEnum

This class represent the supported values of field IDs, e.g. displacement, strain, temperature. Immutable class Enum allows accessing members by .name and .value methods

FID_Concentration = <FieldID.FID_Concentration: 6>

FID_Displacement = <FieldID.FID_Displacement: 1>

FID_Humidity = <FieldID.FID_Humidity: 5>

FID_Material_number = <FieldID.FID_Material_number: 9>

FID_Strain = <FieldID.FID_Strain: 2>

FID_Stress = <FieldID.FID_Stress: 3>

FID_Temperature = <FieldID.FID_Temperature: 4>

FID_Thermal_absorption_surface = <FieldID.FID_Thermal_absorption_surface: 8>

FID_Thermal_absorption_volume = <FieldID.FID_Thermal_absorption_volume: 7>

class mupif.**FunctionID**

Bases: enum.IntEnum

This classenumeration represent the supported values of FunctionID, e.g. FuncID_ProbabilityDistribution

FuncID_ProbabilityDistribution = <FunctionID.FuncID_ProbabilityDistribution: 1>

class mupif.**PropertyID**

Bases: enum.IntEnum

Enumeration class defining Property IDs. These are used to uniquely determine the canonical keywords identifiing individual properties.

PID_AsorptionSpectrum = <PropertyID.PID_AsorptionSpectrum: 26>

PID_ChipSpectrum = <PropertyID.PID_ChipSpectrum: 17>

PID_Concentration = <PropertyID.PID_Concentration: 1>

PID_CumulativeConcentration = <PropertyID.PID_CumulativeConcentration: 2>

PID_Demo_Integral = <PropertyID.PID_Demo_Integral: 9992>

PID_Demo_Max = <PropertyID.PID_Demo_Max: 9991>

PID_Demo_Min = <PropertyID.PID_Demo_Min: 9990>

PID_Demo_Value = <PropertyID.PID_Demo_Value: 9994>

PID_Demo_Volume = <PropertyID.PID_Demo_Volume: 9993>

PID_EmissionSpectrum = <PropertyID.PID_EmissionSpectrum: 24>

PID_ExcitationSpectrum = <PropertyID.PID_ExcitationSpectrum: 25>

PID_InverseCumulativeDist = <PropertyID.PID_InverseCumulativeDist: 28>

PID_KPI01 = <PropertyID.PID_KPI01: 9996>

PID_LEDCCT = <PropertyID.PID_LEDCCT: 20>

PID_LEDColor_x = <PropertyID.PID_LEDColor_x: 18>

PID_LEDColor_y = <PropertyID.PID_LEDColor_y: 19>

PID_LEDRadiantPower = <PropertyID.PID_LEDRadiantPower: 21>

PID_LEDSpectrum = <PropertyID.PID_LEDSpectrum: 16>

PID_NumberOfFluorescentParticles = <PropertyID.PID_NumberOfFluorescentParticles: 29>

PID_NumberOfRays = <PropertyID.PID_NumberOfRays: 15>

PID_ParticleMu = <PropertyID.PID_ParticleMu: 30>

PID_ParticleNumberDensity = <PropertyID.PID_ParticleNumberDensity: 22>

PID_ParticleRefractiveIndex = <PropertyID.PID_ParticleRefractiveIndex: 23>

PID_ParticleSigma = <PropertyID.PID_ParticleSigma: 31>

PID_PhosphorEfficiency = <PropertyID.PID_PhosphorEfficiency: 32>

PID_RefractiveIndex = <PropertyID.PID_RefractiveIndex: 14>

PID_ScatteringCrossSections = <PropertyID.PID_ScatteringCrossSections: 27>

PID_UserTimeStep = <PropertyID.PID_UserTimeStep: 9995>

PID_Velocity = <PropertyID.PID_Velocity: 3>

PID_conductivity_green_phosphor = <PropertyID.PID_conductivity_green_phosphor: 9>

PID_conductivity_red_phosphor = <PropertyID.PID_conductivity_red_phosphor: 8>

PID_effective_conductivity = <PropertyID.PID_effective_conductivity: 5>

PID_mean_radius_green_phosphor = <PropertyID.PID_mean_radius_green_phosphor: 11>

PID_mean_radius_red_phosphor = <PropertyID.PID_mean_radius_red_phosphor: 10>

PID_standard_deviation_green_phosphor = <PropertyID.PID_standard_deviation_green_phosphor: 13>

PID_standard_deviation_red_phosphor = <PropertyID.PID_standard_deviation_red_phosphor: 12>

PID_transient_simulation_time = <PropertyID.PID_transient_simulation_time: 4>

PID_volume_fraction_green_phosphor = <PropertyID.PID_volume_fraction_green_phosphor: 7>

PID_volume_fraction_red_phosphor = <PropertyID.PID_volume_fraction_red_phosphor: 6>

CHAPTER 3

Acknowledgement

Since 2017, the development of the platform has been funded by H2020 project COMPOSELECTOR: “Multi-scale Composite Material Selection Platform with a Seamless Integration of Materials Models and Multidisciplinary Design Framework” with Grant agreement no: 721105.

During 2014-2016, the development of the platform was funded by FP7 project under NMP-2013-1.4-1 call 1.4-1 “Development of an integrated multi-scale modelling environment for nanomaterials and systems by design” with Grant agreement no: 604279.

During 2010-2012, the development of MuPIF was funded by Grant Agency of the Czech Republic, Project “MuPIF - a Multi-Physic Integration Framework” No. P105/10/1402.

Python Module Index

m

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W