MuPIF Documentation

Release 0.11.4

Borek Patzak and Vit Smilauer and Guillaume Pacquaut

August 18, 2015

CONTENTS

Contents:

CHAPTER

ONE

MUPIF PACKAGE

1.1 Subpackages

1.1.1 mupif.Physics package

Submodules

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 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*)

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
```

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*)
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.description()`

Return a string describing all available units.

`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

1.2 Submodules

1.3 mupif.APIError module

exception `mupif.APIError.APIError(_msg)`

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.

`__init__` (*_msg*)
Constructor. Initializes the exception.

Parameters **_msg** (*str*) – Error message

`__str__` ()
Returns error message from the constructor.

Returns Returns string representation of the exception, ie. error message (string)

Return type str

1.4 mupif.Application module

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

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

`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` ()

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 float, TimeStep

`getCriticalTimeStep` ()

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

Return type float

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

Parameters

- **fieldID** (*FieldID*) – Identifier of the field
- **time** (*float*) – 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** (*float*) – 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** (*float*) – 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()

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

Return type bool

registerPyro(*pyroDaemon*, *pyroNS*, *pyroURI*)
Register the Pyro daemon and nameserver. Required by getFieldURI service

Parameters

- **pyroDaemon** (*Pyro4.Daemon*) – Optional pyro daemon
- **pyroNS** (*Pyro4.naming.Nameserver*) – Optional nameserver
- **PyroURI** (*string*) – Optional URI of receiver

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.

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

1.5 mupif.BBox module

class mupif.BBox.**BBox**(*coords_ll*, *coords_ur*)
Bases: object

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

1.6 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

`getGeometryType`()

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: `object`

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
- **label** (*int*) – A cell label
- **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`()

Returns Returns a bounding box of the receiver

Return type BBox

`getGeometryType`()

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

`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

`getGeometryType`()

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

`getGeometryType`()

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

`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

`getGeometryType`()

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

1.7 mupif.CellGeometryType module

Enumeration defining the supported cell geometries

1.8 mupif.EnsightReader2 module

`mupif.EnsightReader2.`**`readEnsightField`**(*name*, *parts*, *partRec*, *type*, *fieldID*, *mesh*)

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

Returns Field of unknowns??, why is FID_Temperature??

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. Why are these functions not under EnsightReader class in EnsightReader.py??

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)

1.9 mupif.Field module

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

Bases: `object`

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)
- **units** (*obj*) – Units of the field values
- **time** (*float*) – Time associated with filed values
- **values** (*tuple*) – Field values (format dependent on a particular field type)
- **fieldType** (*FieldType*) – Optional, determines field type (values specified as vertex or cell values), default is FT_vertexBased

`_evaluate`(*position*, *eps=0.001*)

Evaluates the receiver at a single spatial position.

Parameters

- **position** (*tuple*) – 1D/2D/3D position vector
- **eps** (*float*) – Optional tolerance, default 0.001

Returns field value

Return type tuple

`commit`()

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

`evaluate`(*positions*, *eps=0.001*)

Evaluates the receiver at given spatial position(s).

Parameters

- **position** (*tuple, a list of tuples*) – 1D/2D/3D position vectors
- **eps** (*float*) – Optional tolerance, default 0.001

Returns field value(s)

Return type tuple or a list of tuples

`field2VTKData`()

Creates VTK representation of the receiver. Useful for visualization.

Returns Instance of pyvtk

Return type pyvtk

getFieldID()

Returns Returns field ID

Return type FieldID

getMesh()

Returns Returns a mesh of underlying discretization

Return type Mesh

getUnits()

Returns Returns units of the receiver

Return type obj

getValueType()

Returns Returns value type of the receiver

Return type ValueType

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

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

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.

class mupif.Field.**FieldType**

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

FT_cellBased = 2

FT_vertexBased = 1

1.10 mupif.FieldID module

This class represent the supported values of field IDs, e.g. displacement, strain, temperature.

1.11 mupif.Function module

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

Bases: `object`

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`()

Returns Returns reciver's ID.

Return type int

`getObjectID`()

Returns Returns receiver's object ID

Return type int

1.12 mupif.FunctionID module

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

1.13 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: `object`

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

1.14 mupif.JobManager module

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

Bases: `object`

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 application
- **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

getJobStatus (*jobID*)

Returns the status of the job.

Parameters **jobID** (*str*) – jobID

getPyroFile (*jobID*, *filename*)

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 ()

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.`**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.JobManager.`**`SimpleJobManager2`**(*daemon*, *ns*, *appAPIClass*, *appName*, *portRange*, *jobManWorkDir*, *serverConfigPath*, *serverConfigFile*, *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*, *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'*)
See `JobManager.getPyroFile()`

getStatus()
See `JobManager.getStatus()`

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

See `JobManager.terminateJob()`

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

1.15 mupif.Localizer module

class mupif.Localizer.**Localizer**
Bases: `object`

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

1.16 mupif.Mesh module

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

Bases: `object`

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__``()`

`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``()`

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!

`getCell``(`*i*`)`

Returns i-th cell.

Parameters **i** (*int*) – i-th cell

Returns cell

Return type Cell

`getMapping``()`

Returns The mapping associated to a mesh

Return type defined by API

`getNumberOfCells``()`

Returns The number of Cells

Return type int

`getNumberOfVertices``()`

Returns Number of Vertices

Return type int

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

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

Returns vertex

Return type Vertex

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()

Returns Iterator over vertices

Return type MeshIterator

class mupif.Mesh.**MeshIterator**(*mesh, type*)
Bases: object

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` ()

Returns VTK representation of the receiver .Requires pyvtk module.

Return type pyvtk.UnstructuredGrid

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

`giveCellLocalizer` ()

Returns Returns the cell localizer.

Return type Octree

`giveVertexLocalizer` ()

Returns Returns the vertex localizer.

Return type Octree

`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()`

1.17 mupif.Octree module

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

Defines Octree Octant: a cell containing either terminal data or its child octants.

__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*) – lower left corner octant coordinates
- **size** (*float*) – Size (dimension) of receiver

containsBBox (*_bbox*)

Returns True if BBox contains or intersects the receiver.

delete (*item, itemBBox=None*)
Deletes 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 receiver locally.

evaluate (*functor*)
Evaluate the given functor on all containing objects. The functor should define getBBox() function to return 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.

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. Each terminal octant contains the objects with bounding box within the octant. Each object that can be inserted and is assumed to provide giveBBox() which returns its bounding box.

Octree mask is a tuple containing 0 or 1 values. If corresponding mask value is nonzero, receiver is subdivided in corresponding coordinate direction. The mask allows to create ocrtrees for various 2D and 1D settings.

__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*)
See `Octant.delete()`

evaluate(*functor*)
See `Octant.evaluate()`

giveDepth()
See `Octant.giveDepth()`

giveItemsInBBox(*bbox*)
See `Octant.giveItemsInBBox()`

insert (*item*)
See `Octant.insert()`

1.18 mupif.Property module

class `mupif.Property.`**`Property`** (*value, propID, valueType, time, units, objectID=0*)
Bases: `object`

Property is a characteristic value of a problem, that does not depend on spatial variable, e.g. homogenized conductivity over the whole domain.

Property represents characteristic value of the problem. It can represent value of scalar, vector, or tensorial type. Property keeps its value, objectID, time and type.

__init__ (*value, propID, valueType, time, 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
- **time** (*float*) – Time
- **units** (*PhysicalQuantity*) – Property units
- **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

getPropertID ()
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 ()
Returns the value of property in a tuple.

Returns Property value as array

Return type tuple

1.19 mupif.PropertyID module

Module defining PropertyID as enumeration, e.g. concentration, velocity.

1.20 mupif.PyroFile module

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

Bases: `object`

Class representing a remote 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

`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

1.21 mupif.PyroUtil module

`mupif.PyroUtil.`**`allocateApplicationWithJobManager`**(*ns*, *jobManRec*, *natPort*, *sshClient='ssh'*, *options=''*, *sshHost=''*)

Connect to jobManager described by given jobManRec

Parameters

- **ns** (*Pyro4.naming.Nameserver*) – running name server
- **jobManRec** (*tuple*) – tuple containing (jobManPort, jobManNatport, jobManHostname, jobManUserName, jobManDNSName), see clientConfig.py
- **natPort** (*int*) – nat port in local computer for ssh tunnel for the application
- **sshClient** (*str*) – client for ssh tunnel, see `sshTunnel()`, default 'ssh'
- **options** (*str*) – parameters for ssh tunnel, see `sshTunnel()`, default ''
- **sshHost** (*str*) – parameters for ssh tunnel, see `sshTunnel()`, default ''

Returns RemoteAppRecord containing application, tunnel to application, tunnel to jobman, jobid

Return type RemoteAppRecord

Except allocation of tunnel failed

`mupif.PyroUtil.`**`allocateNextApplication`**(*ns*, *jobManRec*, *natPort*, *appRec*)

Allocate next application on a running Job Manager

Parameters

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

- **jobManRec** (*tuple*) – tuple containing (jobManPort, jobManNatport, jobManHostname, jobManUserName, jobManDNSName), see clientConfig.py
- **natPort** (*int*) – nat port in local computer for ssh tunnel for the application
- **appRec** (*RemoteAppRecord*) – existing RemoteAppRecord where a new application will be added

Returns None

Except allocation or tunnel failed

`mupif.PyroUtil.`**`connectApp`**(*ns*, *name*)
Connects to a remote application.

Parameters

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

Returns Application

Return type Instance of an application

Except Cannot find registered server or Cannot connect to application

`mupif.PyroUtil.`**`connectJobManager`**(*ns*, *jobManRec*, *sshClient='ssh'*, *options=''*, *sshHost=''*)
Connect to jobManager described by given jobManRec and create a ssh tunnel

Parameters

- **jobManRec** (*tuple*) – tuple containing (jobManPort, jobManNatport, jobManHostname, jobManUserName, jobManDNSName), see client-conf.py
- **sshClient** (*str*) – client for ssh tunnel, see `sshTunnel()`, default 'ssh'
- **options** (*str*) – parameters for ssh tunnel, see `sshTunnel()`, default ''
- **sshHost** (*str*) – parameters for ssh tunnel, see `sshTunnel()`, default ''

Returns (JobManager proxy, jobManager Tunnel)

Return type tuple (JobManager, subprocess.Popen)

`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

Except Can not connect to a LISTENING port of nameserver

`mupif.PyroUtil.`**`downloadPyroFile`**(*filename*, *pyroFile*, *size=1024*)
Downloads the (remote) file, represented by given pyroFile into local file (determined by target path)

Parameters

- **filename** (*str*) – path to target file
- **pyroFile** (*PyroFile*) – represenation of source (remote) file
- **size** (*int*) – optional chunk size. The data are read and written in byte chunks of this size

`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.`**`getUserInfo`**()

Returns String assembled from username+"@"+hostname

Return type str

`mupif.PyroUtil.`**`runAppServer`**(*server*, *port*, *nathost*, *natport*, *nshost*, *nsport*, *nsname*, *hkey*, *app*)

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
- **nsname** (*str*) – Nameserver name to register application
- **hkey** (*str*) – A password string
- **app** (*instance*) – Application instance

Except Can not run Pyro4 daemon

`mupif.PyroUtil.`**`runDaemon`**(*host*, *port*, *nathost*, *natport*)

Runs a daemon without geristering 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)

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

`mupif.PyroUtil.`**`sshTunnel`**(*remoteHost*, *userName*, *localPort*, *remotePort*, *sshClient='ssh'*, *options=''*, *sshHost=''*)

Automatic creation of ssh tunnel, using putty.exe for Windows and ssh for Linux

Parameters

- **remoteHost** (*str*) – IP of remote host
- **userName** (*str*) – User name

- **localPort** (*int*) – Local port
- **remotePort** (*int*) – Remote port
- **sshClient** (*str*) – Path to executable ssh client (on Windows use double backslashes 'C:Program FilesPuttyputty.exe')
- **options** (*str*) – Arguments to ssh clinent, e.g. the location of private ssh keys
- **sshHost** (*str*) – Computer used for tunelling, optional. If empty, equals to remoteHost

Returns Instance of subprocess.Popen running the tunneling command

Return type subprocess.Popen

`mupif.PyroUtil.`**`uploadPyroFile`**(*filename*, *pyroFile*)

Uploads the given file (specified by given file name) into PyroFile handle.

Parameters

- **filename** (*str*) – path to source file name
- **pyroFile** (*PyroFile*) – representation of target (remote) file

1.22 mupif.RemoteAppRecord module

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

Bases: `object`

Class keeping data on remote application connection, such as ssh tunnels, etc.

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

Append next application on existing instance

Parameters

- **app** (*Application*) – application instance
- **appTunnel** (*subprocess.Popen*) – ssh tunnel subprocess representing ssh tunnel to application process
- **jobID** (*string*) – application jobID

`getApplication`(*num=0*)

Returns application instance

Parameters **num** (*int*) – number of application, default 0

Returns Instance of Application

`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.

Parameters **num** (*int*) – number of application

1.23 mupif.TimeStep module

class `mupif.TimeStep.`**`TimeStep`**(*t*, *dt*, *n=1*)

Bases: `object`

Class representing a time step.

`__init__`(*t*, *dt*, *n=1*)

Initializes time step.

Parameters

- **t** (*float*) – Time
- **dt** (*float*) – Step length (time increment)
- **n** (*int*) – Optional, solution time step number, default = 1

`getNumber`()

Returns Receiver's solution step number

Return type int

`getTime`()

Returns Time

Return type float

`getTimeIncrement`()

Returns Time increment

Return type float

1.24 mupif.Timer module

class `mupif.Timer.`**`Timer`**

Bases: `object`

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__().

1.25 mupif.Util module

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

Finds a 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

1.26 mupif.ValueType module

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

1.27 mupif.Vertex module

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

Bases: `object`

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

`getCoordinates`()

Returns Receiver's coordinates

Return type tuple

1.28 mupif.VtkReader2 module

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

Parameters

- **mesh** (*Mesh*) – Source mesh
- **Data** (*vtkData*) – vtkData obtained by pyvtk
- **fieldID** (*FieldID*) – Field type (displacement, strain, temperature ...)
- **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

1.29 Module contents

This is a MuPIF module (Multi-Physics Integration Framework)

CHAPTER TWO

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

m

Symbols

A

B

C

D

E

F

G

I

J

L

M

V

W