ATOM

ANTENNA TOOLBOX FOR MATLAB

Documentation

23 Mar 2020

Contents

Namespace

+models/+geom

Class +models/+geom/@Boolean

AToM:+models:+geom:@Boolean:subtract

subtract: perform boolean operation subtract on objects

This perfoirms subtract Boolean operation on specified objects.

Inputs

obj: Boolean object, [1 x 1]

names1: name of Object 1, char [1 x N]

names2: name of Object 2, char [1 x N]

type1: optional, type of Object 1, char [1 x N]

type2: optional, type of Object 2, char [1 x N]

callerName: optional, name of calling function, char [1 x N]

Syntax

```
obj.subtract(names1, names2)
```

Objects specified by *names1* and *names2* are subtracted.

```
obj.subtract(names1, names2, type1, type2, callerName)
```

Objects are searched faster according to hints in *type1* and *type2*. Valid values of *callerName* are: 'recomputeCommands' (do not write to History), 'user' (write to history).

AToM:+models:+geom:@Boolean:unite

unite: perform boolean operation unite on objects

This perfoirms unite Boolean operation on specified objects.

Inputs

obj: Boolean object, [1 x 1]

names1: name of Object 1, char [1 x N]

names2: name of Object 2, char [1 x N]

type1: optional, type of Object 1, char [1 x N]

type2: optional, type of Object 2, char [1 x N]

callerName: optional, name of calling function, char [1 x N]

Syntax

```
obj.unite(names1, names2)
```

Objects specified by *names1* and *names2* are subtracted.

```
obj.unite(names1, names2, type1, type2, callerName)
```

Objects are searched faster according to hints in *type1* and *type2*. Valid values of *callerName* are: 'recomputeCommands' (do not write to History), 'user' (write to history).

Class +models/+geom/@GeomObject

AToM:+models:+geom:@GeomObject:areLinesInObject2

areLinesInObject: determine if lines lie inside of an object

This method determines if 3D lines lie inside or outside of specified object.

Inputs

```
obj: object if interest, GeomObject [1 x 1]
lines: 3D lines, struct [1 x nLines]
      .startPoint: start points, double [nLines x 3]
      .endPoint: end point, double [nLines x 3]
isDivided: default false, are Segments of obj divided, logical [1 x 1]
```

Outputs

```
areIn: are lines inside object, logical [nP x 1]
incStruct: struct [1 x n]
  .segmentation Parts #, double [1 x nSPIN]
  .curveNums: seg. curves # in corresponding segPartNum, double [1 x nSPIN]
```

Syntax

```
[areIn, incStruct] = obj.areLinesInObject(lines)
```

Method areLinesInObject determines if lines specified by a struct *lines* lie totaly inside or outside the GeomObject *obj*. The line is defined by a struct with properties: lines.startPoint and .endPoint.

```
[areIn, incStruct] = obj.areLinesInObject(lines, isDivided)
```

If *isDivided* set to true, the object contour segments are divided, if line's end point is on it. If *isDivided* set to false, the contour is noy changed.

AToM:+models:+geom:@GeomObject:arePointsInObject

arePointsInObject: determine if points lie inside of an object

This method determines if 3D points lie inside or outside of specified object.

Inputs

obj: `object if interest, GeomObject [1 x 1]`

points: `3D points, double [nP x 3]`

parts: `optional, part # that should be checked, double [n x 1]`

Outputs

areIn: `are points inside object, logical [nP x 1]`

partNums: `part # where both IN, cell [1 x nLines], double [nParts x 1]`

Syntax

```
areIn = obj.arePointsInObject(points)
```

Method arePointsInObject determines if 3D *points* lie inside GeomObject *obj* or not.

AToM:+models:+geom:@GeomObject:getLocalCoordinateSystem

getLocalCoordinateSystem: get objects local coordinate system

This method determines local coordinate system of specified object.

Inputs

obj: `object if interest, GeomObject [1 x 1]`

Outputs

origin: `coordinate system origin if interest, double [nParts x 3]`

localX: `X axis direction, double [nParts x 3]`

localX: `Y axis direction, double [nParts x 3]`

localX: `Z axis direction, double [nParts x 3]`

Syntax

```
[origin, localX, localY, localZ] = obj.getLocalCoordinateSystem
```

Method getLocalCoordinateSystem computes one point (*origin*) and thre vectors (*localX*, *localY*, *localZ*) that determines local coordinate system of specified object *obj*.

AToM:+models:+geom:@GeomObject:getSymmetrySegmentation

getSymmetrySegmentation: get segmentation when symmetries applied

This method gives back segmentation

Inputs

obj: object of interest, GeomObject [1 x 1]

symmTypes: types of symmetry ('xy', 'xz', 'yz'), cell [1 x nSymm]

Outputs

symmSeg: segmentation (viz Segmentation doc) of part, Segmentation [1 x 1]
onSymmetry, is some segPart on Symmetry Plane, cell{1, nSegParts} of chars
intersectLines, iintersetLine segments, IntersectLine [1, nISL]

Syntax

```
[symmSeg, onSymmetry, intersectLine] = getSymmetrySegmentation( ...
obj, symmTypes)
```

Method getSymmetrySegmentation computes for segmentation of part of symmetry GeomObject *obj*. The part of segmentation is stored in _symmSeg, which is an object of class Segmentation.

Class +models/+geom/@Geom

AToM:+models:+geom:@Geom:addCircle

addCircle: include Ellipse of circular shape to Geom

This method adds a new object Ellipse to object container of class Geom. It returns objName of the new object.

Inputs

obj: Geom object

center: circle's center point, char [1 x N]/double [1 x 3]

radius: circle's radius, char [1 x N]/double [1 x 3]

normal: optional, rectangle's normal spec. (default 'z'), char [1 x 1]

name: optional, name of object, char [1 x N]

Outputs

objName: name of new GeomObject, char [1 x N]

Syntax

```
objName = obj.addCircle(center, radius)
```

The object of type Ellipse specified by center point *center* and *radius* is created in Geom.

```
objName = obj.addCircle(center, radius, normal)
```

The object of type Ellipse specified by center point *center*, *radius* and normal direction *norm* is created in Geom.

```
id = obj.addCircle(center, radius, normal, name)
```

The object name is set to *name*.

AToM:+models:+geom:@Geom:addCircleArc

addCircleArc: include EllipseArc of circular shape to Geom

This method adds a new object EllipseArc to object container of class Geom. It returns objName of the new object.

Inputs

obj: Geom object

center: circle's center point, char [1 x N]/double [1 x 3]

radius: circle's radius, char [1 x N]/double [1 x 3]

normal: optional, circle's normal spec. (default 'z'), char [1 x 1]

name: optional, name of object, char [1 x N]

Outputs

objName: name of new GeomObject, char [1 x N]

Syntax

```
objName = obj.addCircleArc(center, radius)
```

The object of type EllipseArc specified by center point *center* and *radius* is created in Geom.

```
objName = obj.addCircleArc(center, radius, normal)
```

The object of type EllipseArc specified by center point *center*, *radius* and normal direction *norm* is created in Geom.

```
id = obj.addCircleArc(center, radius, normal, name)
```

The object name is set to *name*.

AToM:+models:+geom:@Geom:addEllipse

addEllipse: include Ellipse to Geom object container

This method adds a new object Ellipse to object container of class Geom. It returns objName of the new object.

Inputs

obj: Geom object

cP: center point, char [1 x N]/double [1 x 3]

majV: major axis vertex point, char [1 x N]/double [1 x 3]

minV: minor axis vertex point, char [1 x N]/double [1 x 3]

sA: start angle of Ellipse, char [1 x N]/double [1 x 1]

a: angle of Ellipse, char [1 x N]/double [1 x 1]

name: optional, name of object, char [1 x N]

Outputs

objName: name of new GeomObject, char [1 x N]

Syntax

```
objName = obj.addEllipse(cP, majV, minV, sA, a)
```

The object of type Ellipse specified by *cP*, *majV*, *minV*, *_sA* and *a* is created in Geom.

```
objName = obj.addEllipse(cP, majV, minV, sA, a, name)
```

The object of type Ellipse specified by *cP*, *majV*, *minV*, *_sA* and *a* is created in Geom. The name of new object is set to *name*.

AToM:+models:+geom:@Geom:addEllipseArc

addEllipseArc: include EllipseArc to Geom object container

This method adds a new object EllipseArc to object container of class Geom. It returns objName of the new object.

Inputs

obj: Geom object

cP: center point, char [1 x N]/double [1 x 3]

majV: major axis vertex point, char [1 x N]/double [1 x 3]

minV: minor axis vertex point, char [1 x N]/double [1 x 3]

sA: start angle of EllipseArc, char [1 x N]/double [1 x 1]

a: angle of EllipseArc, char [1 x N]/double [1 x 1]

name: optional, name of object, char [1 x N]

Outputs

objName: name of new GeomObject, char [1 x N]

Syntax

```
objName = obj.addEllipseArc(cP, majV, minV, sA, a)
```

The object of type EllipseArc specified by *cP*, *majV*, *minV*, *_sA* and *a* is created in Geom.

```
objName = obj.addEllipseArc(cP, majV, minV, sA, a, name)
```

The object of type EllipseArc specified by *cP*, *majV*, *minV*, *_sA* and *a* is created in Geom. The name of new object is set to *name*.

AToM:+models:+geom:@Geom:addEquationCurve

addEquatoionCurve: include EquatoionCurve to Geom

This method adds a new object EquatoionCurve to object container of class Geom. It returns objName of the new object.

Inputs

obj: Geom object

eqX: handle function to X coordinate, char [1 x N]

eqY: handle function to Y coordinate, char [1 x N]

eqZ: handle function to Z coordinate, char [1 x N]

parInt: interval for parameter, char [1 x N]/double [1 x 2]

Outputs

objName: name of new GeomObject, char [1 x N]

Syntax

The object of type EquatoionCurve specified by handle functions *eqX*, *eqY*, *eqZ and parameter interval _parInt* is created in Geom.

```
objName = obj.addLine(points, name)
```

The object of type EquatoionCurve specified by handle functions *eqX*, *eqY*, *eqZ and parameter interval _parInt* is created in Geom. The name of new object is set to *name*.

AToM:+models:+geom:@Geom:addLine

addLine: include Line to Geom

This method adds a new object Line to object container of class Geom. It returns objName of the new object.

Inputs

obj: Geom object

points: expression for coordinates definition, char [1 x N]/double [2 x 3]

name: optional, name of object, char [1 x N]

Outputs

objName: name of new GeomObject, char [1 x N]

Syntax

```
objName = obj.addLine(points)
```

The object of type Line with start and end points specified by *points* is created in Geom.

```
objName = obj.addLine(points, name)
```

The object of type Line with start and end points specified by *points* is created in Geom. The name of new object is set to *name*.

AToM:+models:+geom:@Geom:addParallelogram

addParallelogram: include Parallelogram to Geom object container

This method adds a new object Parallelogram to object container of class Geom. It returns objName of the new object.

Inputs

obj: Geom object

lLC: low left corner position, char [1 x N]/double [1 x 3]

lRC: low right corner position, char [1 x N]/double [1 x 3]

hLC: high left corner position, char [1 x N]/double [1 x 3]

name: optional, name of object, char [1 x N]

Outputs

objName: name of new GeomObject, char [1 x N]

Syntax

```
objName = obj.addParallelogram(lLC, lRC, hLC)
```

The object of type Parallelogram specified by three corners *lLC*, *lRC*, *hLC* is created in Geom.

```
objName = obj.addParallelogram(lLC, lRC, hLC, name)
```

The object of type Parallelogram specified by *lLC*, *lRC*, *hLC* is created in Geom. The name of new object is set to *name*.

AToM:+models:+geom:@Geom:addParallelogramFrame

addParallelogramFrame: include ParallelogramFrame to Geom

This method adds a new object ParallelogramFrame to object container of class Geom. It returns objName of the new object.

Inputs

obj: Geom object

lLC: low left corner, char [1 x N]/double [1 x 3]

lRC: low right corner, char [1 x N]/double [1 x 3]

hLC: high left corner, char [1 x N]/double [1 x 3]

name: optional, name of object, char [1 x N]

Outputs

objName: name of new GeomObject, char [1 x N]

Syntax

```
objName = obj.addParallelogramFrame(lLC, lRC, hLC)
```

The object of type ParallelogramFrame specified by three corners *lLC*, *lRC*, _hLC is created in Geom.

```
objName = obj.addParallelogramFrame(lLC, lRC, hLC, name)
```

The object of type ParallelogramFrame specified by three corners *lLC*, *lRC*, *hLC is created in Geom. The name of new object is set to _name.*

AToM:+models:+geom:@Geom:addPoint

addPoint: include Point to Geom

This method adds a new object Point to object container of class Geom. It returns objName of the new object.

Inputs

obj: Geom object

coords: expression for coordinates definition, char [1 x N]/double [1 x 3]

name: optional, name of object, char [1 x N]

Outputs

objName: name of new GeomObject, char [1 x N]

Syntax

```
objName = obj.addPoint(coords)
```

The object of type Point with coordinates *coords* is created in Geom.

```
objName = obj.addPoint(coords, name)
```

The object of type Point with coordinates *coords* is created in Geom. The name of new object is set to *name*.

AToM:+models:+geom:@Geom:addPolyLine

addPolyLine: include broken line to Geom

This method adds a new object PolyLine to object container of class Geom. It returns objName of the new object.

Inputs

obj: Geom object

points: expression for coordinates definition, char [1 x N]/double [N x 3]

name: optional, name of object, char [1 x N]

Outputs

objName: name of new GeomObject, char [1 x N]

Syntax

```
objName = obj.addPolyLine(points)
```

The object of type PolyLine with start defined as first row of *points* and end in last row of *points* is created in Geom.

```
objName = obj.addPolyLine(points, name)
```

The name of new object is set to *name*.

AToM:+models:+geom:@Geom:addPolyLoop

addPolyLoop: include PolyLoop to Geom

This method adds a new object PolyLoop to object container of class Geom. It returns objName of the new object.

Inputs

`obj`: Geom object

`curveNames`: names of individual curves to be added, char [1 x N]

`name`: optional, name of object, char [1 x N]

Outputs

`objName`: name of new GeomObject, char [1 x N]

Syntax

```
objName = obj.addPolyLoop(curveNames)
```

The object of type PolyLoop is created from curves specified by *curveNames* from Geom.

```
objName = obj.addPolyLoop(curveNames, name)
```

The object of type PolyLoop is created from curves specified by *curveNames* from Geom. The name of new object is set to *name*.

AToM:+models:+geom:@Geom:addPolygon

addPolygon: include arbitrary Polygon to Geom

This method adds a new object Polygon to object container of class Geom. It returns objName of the new object.

Inputs

obj: Geom object

points: expression for coordinates definition, char [1 x N]/double [N x 3]

name: optional, name of object, char [1 x N]

Outputs

objName: name of new GeomObject, char [1 x N]

Syntax

```
objName = obj.addPolygon(points)
```

The object of type Polygon with start defined as first row of *points* and end in last row of *points* is created in Geom. If first point and last point are not the same, the first point is copied to the end of points to form closed Polygon.

```
objName = obj.addPolygon(points, name)
```

The name of new object is set to *name*.

AToM:+models:+geom:@Geom:addRectangle

addRectangle: include Parallelogram of rectangle shape to Geom

This method adds a new object Parallelogram to object container of class Geom. It returns objName of the new object.

Inputs

obj: Geom object

center: rectangle's center point, char [1 x N]/double [1 x 3]

width: rectangle's width, char [1 x N]/double [1 x 3]

height: rectangle's height, char [1 x N]/double [1 x 3]

normal: optional, rectangle's normal spec. (default 'z'), char [1 x 1]

name: optional, name of object, char [1 x N]

Outputs

objName: name of new GeomObject, char [1 x N]

Syntax

```
objName = obj.addRectangle(center, width, height)
```

The object of type Parallelogram specified by center point *center*, *width* and *height* is created in Geom.

```
objName = obj.addRectangle(center, width, height, normal)
```

The object of type Parallelogram specified by center point *center*, *width*, *height* and normal direction *norm* is created in Geom.

```
id = obj.addRectangle(center, width, height, normal, name)
```

The object name is set to *name*.

AToM:+models:+geom:@Geom:addRectangleFrame

addRectangleFrame: include ParallelogramFrame of rectangle shape to Geom

This method adds a new object ParallelogramFrame to object container of class Geom. It returns objName of the new object.

Inputs

obj: Geom object

center: rectangle's center point, char [1 x N]/double [1 x 3]

width: rectangle's width, char [1 x N]/double [1 x 3]

height: rectangle's height, char [1 x N]/double [1 x 3]

normal: optional, rectangle's normal spec. (default 'z'), char [1 x 1]

name: optional, name of object, char [1 x N]

Outputs

objName: name of new GeomObject, char [1 x N]

Syntax

```
objName = obj.addRectangleFrame(center, width, height)
```

The object of type ParallelogramFrame specified by center point *center*, *width* and *height* is created in Geom.

```
objName = obj.addRectangleFrame(center, width, height, normal)
```

The object of type ParallelogramFrame specified by center point *center*, *width*, *height* and normal direction *norm* is created in Geom.

```
objName = obj.addRectangleFrame(center, width, height, normal, name)
```

The object name is set to *name*.

ATоM:+models:+geom:@Geom:copyObject

copy: copy object along vector

This function copies specified object along vector N-times.

Inputs

`obj`: Geom [1 x 1]

`objName`: GeomObject, char [1 x N]

`vect`: defines where object should be copied, double [1 x 3]

`nNew`: optional, number of copied objects, double [1 x 1]

`newName`: optional, names for new objects, char [1 x N]

`type`: optional, type of GeomObject

Syntax

```
obj.copyObject(objName, vect)
```

New object is placed along vector *vect* having same properties as origin object *objName*.

```
obj.copyObject(objName, vect, nNew)
```

Several new objects are produced based on definition of *objName*. Number of objects is defined by *nNew*. The distance between two neighbours is defined by euclidean distance of vector *vect*.

```
obj.copyObject(objName, vect, nNew, newName)
```

Produced objects ara named according to char specified in *newName* numbered from 1.

ATоM:+models:+geom:@Geom:deleteObject

deletObject: delete object from Geom

This method removes an object specified by its name name from Geom *obj*.

Inputs

`obj`: Geom object, [1 x 1]

`name`: object name, char [1 x N]

`type`: optional type of object, char [1 x N]

Outputs

`isDeleted`: logical [1 x 1]

Syntax

The object specified by *name* is removed from Geom object *obj*. If *type* of object is set, the search is performed faster just within objects of specified type. Possible types are objects of class GeomObjectType: 'Point', 'Line', 'EllipseArc', 'EquationCurve', 'ParallelogramFrame', 'PolyLine', 'PolyCurve', 'Parallelogram', 'Ellipse', 'PolyLoop', 'Polygon', 'Curve', 'Shape'.

AToM:+models:+geom:@Geom:moveCommandObject

moveCommandObject: move command to new position in history of object

This method removes a command from history of object's transformations.

Inputs

obj: Geom object, [1 x 1]

name: object name, char [1 x N]

oldNum: id of command to be moved, double [1 x 1]

newNum: new position of comand in History, double [1 x 1]

type: optional type of object, char [1 x N]

Outputs

isMoved: logical [1 x 1]

Syntax

```
oldNum = obj.removeCommandObject(name, oldNum, newNum, type)
```

The command specified by command number *oldNum* of object specified by *name* is moved in object's history to new position specified by *newNum* }if the object is found in Geom object *obj*).After the command is removed, the object history is recomputed. If *type* of object is set, the search is performed faster just within objects of specified type. Possible types are objects of class GeomObjectType: Point, Line, EllipseArc, EquationCurve, ParallelogramFrame, PolyCurve, Parallelogram, Ellipse, PolyLoop, Curve, Shape.

AToM:+models:+geom:@Geom:reconstructObject

reconstructObject: reconstruct object from Geom

This method reinitiates an object specified by its *name* saved in Geom. The definig properties of the object are set to initiate expresssion value or to new values.

Inputs

obj: Geom object, [1 x 1]

name: object name, char [1 x N]

varargin:
```
        type: optional, type of object, char [1 x N]
        property-value pairs, with new values for defining prop. [1 x 2N]
```

Outputs

isModified: logical [1 x 1]

Syntax

```
isModified = obj.reconstructObject(name)
```

The object specified by *name* is reconstructed (if found in Geom object *obj*). The objects defining properties are reconstructed to initial expression.

```
isModified = obj.reconstructObject(name, type)
```

The object specified by *name* is reconstructed (if found in Geom object *obj*). If *type* of object is set, the search is performed faster just within objects of specified type. Possible types are objects of class GeomObjectType: 'Point', 'Line', 'EllipseArc', 'EquationCurve', 'ParallelogramFrame', 'PolyCurve', 'Parallelogram', 'Ellipse', 'PolyLoop', 'Curve', 'Shape'. The objects defining properties are reconstructed to initial expression.

```
isModified = obj.reconstructObject(name, type, varargin)
```

The objects defining properties are to values defined by 'property'-'value' pairs in *varargin*.

AToM:+models:+geom:@Geom:redrawObject

redrawObject: change number of drawPoints for GeomObjects

Number of drawPoints of GeomObjects is changed by user.

Inputs

obj: Geom object [1 x 1]

name: names of objects to be modified, char [1 x N]

nPoints: number of points to be used for visualization, double [1 x N]

type: optional, types of Geom Object, char [1 x N]

Syntax

```
obj.redrawObject(names, nPoints)
```

The objects of Geom (*obj*) specified in *namse* are changed. Number of drawPoints is set according to *nPoints*.

```
obj.redrawObject(names, nPoints, types)
```

If *types* is specified, the search within Geom is performed faster.

AToM:+models:+geom:@Geom:removeCommandObject

removeCommandObject: remove command from history of object

This method removes a command from history of object's transformations.

Inputs

obj: Geom object, [1 x 1]

name: object name, char [1 x N]

cmdNum: id of command to be removed, double [1 x 1]

type: optional type of object, char [1 x N]

Outputs

isRemoved: logical [1 x 1]

Syntax

```
isRemoved = obj.removeCommandObject(name, cmdNum,  type)
```

The command specified by command number *cmdNum* of object specified by *name* is removed from object's history if the object is found in Geom object *obj*. After the command is removed, the object history is recomputed. If *type* of object is set, the search is performed faster just within objects of specified type. Possible types are objects of class GeomObjectType: 'Point', 'Line', 'EllipseArc', 'EquationCurve', 'ParallelogramFrame', 'PolyLine', 'PolyCurve', 'Parallelogram', 'Ellipse', 'PolyLoop', 'Polygon', 'Curve', 'Shape'.

AToM:+models:+geom:@Geom:renameObject

renameObject: rename object in Geom

This method renames an object specified by its name *oldName* from Geom *obj*.

Inputs

obj: Geom object, [1 x 1]

oldNname: current object name, char [1 x N]

newNname: new name specified by user, char [1 x N]

type: optional type of object, char [1 x N]

Outputs

isRenamed: logical [1 x 1]

Syntax

```
isRenamed = renameObject(obj, oldName, newName, type)
```

The object specified by *oldName* is removed from Geom object *obj*. If *type* of object is set, the search is performed faster just within objects of specified type. Possible types are objects of class GeomObjectType: 'Point', 'Line', 'EllipseArc', 'EquationCurve', 'ParallelogramFrame', 'PolyLine', 'PolyCurve', 'Parallelogram', 'Ellipse', 'PolyLoop', 'Polygon', 'Curve', 'Shape'. The object is now named using name specified by user *newName*. If *newName* is allready used in Geom, user is asked to set different name.

AToM:+models:+geom:@Geom:rotateObject

rotateObject: rotate object from Geom

This method rotates an object specified by its name saved in Geom.

Inputs

obj: `Geom object, [1 x 1]`

name: `object name, char [1 x N]`

vect: `rotation axis, double [1/2 x 3]`

angle: `rotation angle, double [1 x 1]`

type: `optional type of object, char [1 x N]`

callerName: `optional, caller name to control notifications, char [1 x N]`

Outputs

isModified: `logical [1 x 1]`

Syntax

```
isModified = obj.rotateObject(name, vect, angle, type, callerName)
```

The object specified by *name* is rotated (if found in Geom object *obj*). If *type* of object is set, the search is performed faster just within objects of specified type. Possible types are objects of class GeomObjectType: 'Point', 'Line', 'EllipseArc', 'EquationCurve', 'ParallelogramFrame', 'PolyLine' 'PolyCurve', 'Parallelogram', 'Ellipse', 'PolyLoop', 'Polygon', 'Curve', 'Shape'. The object is rotated around vector specified by *vect* around angle determined by *angle*.

AToM:+models:+geom:@Geom:rotateXObject

rotateXObject: rotate object from Geom around X axis

This method rotates an object specified by its name saved in Geom.

Inputs

obj: `Geom object, [1 x 1]`

name: `object name, char [1 x N]`

angle: `rotation angle, double [1 x 1]`

type: `optional type of object, char [1 x N]`

callerName: `optional, caller name to control notifications, char [1 x N]`

Outputs

isModified: `logical [1 x 1]`

Syntax

```
isModified = obj.rotateXObject(name, angle, type, callerName)
```

The object specified by *name* is rotated (if found in Geom object *obj*). If *type* of object is set, the search is performed faster just within objects of specified type. Possible types are objects of class GeomObjectType: 'Point', 'Line', 'EllipseArc', 'EquationCurve', 'ParallelogramFrame', 'PolyLine' 'PolyCurve', 'Parallelogram', 'Ellipse', 'PolyLoop', 'Polygon', 'Curve', 'Shape'. The object is rotated around X axis [1, 0, 0] around angle determined by *angle*.

ATOM:+models:+geom:@Geom:rotateYObject

rotateYObject: rotate object from Geom around Y axis

This method rotates an object specified by its name saved in Geom.

Inputs

obj: Geom object, [1 x 1]

name: object name, char [1 x N]

angle: rotation angle, double [1 x 1]

type: optional type of object, char [1 x N]

callerName: optional, caller name to control notifications, char [1 x N]

Outputs

isModified: logical [1 x 1]

Syntax

```
isModified = obj.rotateYObject(name, angle, type, callerName)
```

The object specified by *name* is rotated (if found in Geom object *obj*). If *type* of object is set, the search is performed faster just within objects of specified type. Possible types are objects of class GeomObjectType: 'Point', 'Line', 'EllipseArc', 'EquationCurve', 'ParallelogramFrame', 'PolyLine' 'PolyCurve', 'Parallelogram', 'Ellipse', 'PolyLoop', 'Polygon', 'Curve', 'Shape'. The object is rotated around Y axis [0, 1, 0] around angle determined by *angle*.

AToM:+models:+geom:@Geom:rotateZObject

rotateZObject: rotate object from Geom around Z axis

This method rotates an object specified by its name saved in Geom.

Inputs

obj: `Geom object, [1 x 1]`

name: `object name, char [1 x N]`

angle: `rotation angle, double [1 x 1]`

type: `optional type of object, char [1 x N]`

callerName: `optional, caller name to control notifications, char [1 x N]`

Outputs

isModified: `logical [1 x 1]`

Syntax

```
isModified = obj.rotateZObject(name, angle, type, callerName)
```

The object specified by *name* is rotated (if found in Geom object *obj*). If *type* of object is set, the search is performed faster just within objects of specified type. Possible types are objects of class GeomObjectType: 'Point', 'Line', 'EllipseArc', 'EquationCurve', 'ParallelogramFrame', 'PolyLine' 'PolyCurve', 'Parallelogram', 'Ellipse', 'PolyLoop', 'Polygon', 'Curve', 'Shape'. The object is rotated around Z axis [0, 0, 1] around angle determined by *angle*.

AToM:+models:+geom:@Geom:scaleObject

scaleObject: scale object from Geom according to vector

This method scales an object specified by its name saved in Geom.

Inputs

obj: Geom object, [1 x 1]

name: object name, char [1 x N]

vect: scaling vector, double [1 x 3]

type: optional type of object, char [1 x N]

callerName: optional, caller name to control notifications, char [1 x N]

Outputs

isModified: logical [1 x 1]

Syntax

```
isModified = obj.scaleObject(name, vect, type, callerName)
```

The object specified by *name* is scaled (if found in Geom object *obj*). If *type* of object is set, the search is performed faster just within objects of specified type. Possible types are objects of class GeomObjectType: 'Point', 'Line', 'EllipseArc', 'EquationCurve', 'ParallelogramFrame', 'PolyLine' 'PolyCurve', 'Parallelogram', 'Ellipse', 'PolyLoop', 'Polygon', 'Curve', 'Shape'. The object is scaled according to vector specified by *vect*.

AToM:+models:+geom:@Geom:translateObject

translateObject: translate object from Geom according to vector

This method translates an object specified by its name saved in Geom.

Inputs

obj: `Geom object, [1 x 1]`

name: `object name, char [1 x N]`

vect: `translation vector, double [1 x 3]`

type: `optional type of object, char [1 x N]`

callerName: `optional, caller name to control notifications, char [1 x N]`

Outputs

isModified: `logical [1 x 1]`

Syntax

```
isModified = obj.translateObject(name, angle, type, callerName)
```

The object specified by *name* is translated (if found in Geom object *obj*). If *type* of object is set, the search is performed faster just within objects of specified type. Possible types are objects of class GeomObjectType: 'Point', 'Line', 'EllipseArc', 'EquationCurve', 'ParallelogramFrame', 'PolyLine' 'PolyCurve', 'Parallelogram', 'Ellipse', 'PolyLoop', 'Polygon', 'Curve', 'Shape'. 'Shape'. The object is translated according to vector specified by *vect*.

Namespace

+models/+mesh

Class +models/+mesh/@Mesh

AToM:+models:+mesh:@Mesh:convertToImportedMesh

convertToImportedMesh: convers geometry to imported mesh

```
Converts mesh from atom geometry to imported mesh.
```

Inputs

`obj`: Mesh object

`name`: object name, char [1 x N]

Syntax

```
obj.convertToImportedMesh();
obj.convertToImportedMesh(name);
```

AToM:+models:+mesh:@Mesh:deleteEdges1D

deleteEdges1D: deletes 1D edges given by edge indices

```
This function deletes 1D edges given by edge indices.
```

Inputs

`obj`: Mesh object, [1 x 1]

`edgesToDelete`: list of 1D edges to remove, double [N x 1]

Syntax

```
obj.deleteEdges1D(edgesToDelete);
```

AToM:+models:+mesh:@Mesh:deleteMesh

deleteMesh: deletes selected mesh object

```
Takes mesh object name as input. When an object with a given name is found,
it's directly deleted if mesh was imported, otherwise user is prompted that
object was created from AToM geometry.
```

Inputs

`mesh`: Mesh object, Mesh [1 x 1]

`name`: name of Mesh object to delete, char [1 x N]

Syntax

```
obj.deleteMesh(name);
```

AToM:+models:+mesh:@Mesh:deleteNodes

deleteNodes: deletes nodes of specific meshObject

This function deletes nodes of specific object.

Inputs

obj: Mesh object, [1 x 1]

nodes: list of nodes to remove, double [N x 1]

Syntax

```
obj.deleteNodes(nodesToDelete);
```

AToM:+models:+mesh:@Mesh:deleteTriangles

deleteTriangles: deletes triangles of specific meshObject

This function deletes triangles of specific object.

Inputs

obj: Mesh object, [1 x 1]

trianglesToDelete: list of triangles to remove, double [N x 1]

Syntax

```
obj.deleteTriangles(trianglesToDelete);
```

AToM:+models:+mesh:@Mesh:exportMesh

exportMesh: Exports mesh in specified format

Inputs

filePath: path to output directory, char [1 x N]

name: name of the export file, char [1 x N]

type: type of exported format, char [1 x N]

Syntax

```
obj.exportMesh(type, name, filePath);
```

AToM:+models:+mesh:@Mesh:getBoundaryMesh

getBoundaryMesh: create boundary mesh of 2D mesh objects

```
This function returns boundary of 2D mesh objects.
```

Syntax

```
obj.getBoundaryMesh();
```

AToM:+models:+mesh:@Mesh:getCircumsphere

getCircumsphere: computes mesh circumsphere

```
A circumsphere is computed for the whole mesh and optionaly for each object.
If eachObject is true, first line is circumsphere of each object and next
lines are circumsphere of each Mesh Object.
```

Inputs

```
obj: object of class Mesh, [1 x 1]
```

Outputs

```
coordinates:
  -radius: radius of a circumsphere, double [N x 1]
  -center: center of a circumsphere, double [N x 3]
```

Syntax

```
[coordinates] = obj.getCircumsphere();
```

AToM:+models:+mesh:@Mesh:getEdges

getEdges: get 1D and 2D mesh edges

Outputs

```
edges1D: 1D element edges, double [N x 2]
edges2D: 2D element edges, double [N x 2]
```

Syntax

```
[edges1D, edges2D] = obj.getEdges();
```

AToM:+models:+mesh:@Mesh:getMesh

getMesh: loads all curves from geom a generates 1D mesh

```
This function calls all specific functions to load geometry and generate 1D
and 2D mesh.
```

Syntax

```
obj.getMesh();
```

AToM:+models:+mesh:@Mesh:getMeshData1D

getMeshData1D: computes information necessary for 1D mesh solvers

```
This function loads data from mesh and outputs struct with data necessary
for 1D mesh solvers.
```

Outputs

```
meshData: structure with following items
            -nodes, triangulation nodes, double [N x 3]
            -edges, triangulation edges, double [N x 2]
            -edgeLengths, center point of each edge, double [N x 3]
            -edgeCentroids, length of each edge, double [N x 1]
            -nNodes, number of nodes, double [1 x 1]
            -nEdges, number of edges, double [1 x 1]
            -nNodesBasic, number of nodes before symmetry, double [1 x 1]
            -nEdgesBasic, number of edges before symmetry, double [1 x 1]
```

Syntax

```
obj.getMeshData1D();
```

AToM:+models:+mesh:@Mesh:getMeshData2D

getMeshData2D: computes information necessary for MoM computations

```
This function loads data from mesh and outputs struct with data necessary
for MoM
```

Outputs

```
meshData: structure with following items
             -nodes, triangulation nodes, double [N x 3]
             -connectivityList, triangulation connectivity list, double [N x 3]
             -edges, triangulation edges, double [N x 2]
             -triangleEdgeCenters, center point of each edge, double [N x 3]
             -triangleEdgeLengths, length of each edge, double [N x 1]
             -triangleAreas, area of each triangle, double [N x 1]
             -triangleCentroids, center points of each triangle, double [N x 3]
             -triangleEdges, indices to edges, double [N x 3]
             -nNodes, number of nodes, double [1 x 1]
             -nEdges, number of edges, double [1 x 1]
             -nTriangles, number of triangles [1 x 1]
             -normDistanceA, radius of circumsphere, double [1 x 1]
             -nNodesBasic, number of nodes before symmetry, double [1 x 1]
             -nEdgesBasic, number of edges before symmetry, double [1 x 1]
             -nTrianglesBasic, number of triangles
              before symmetry, double [1 x 1]
             -edgeOrigins, edges origins before symmetry, double [N x 1]
             -triangleOrigins, triangles origins before symmetry, double [N x 1]
```

Syntax

```
obj.getMeshData2D();
```

AToM:+models:+mesh:@Mesh:getMeshStatistics

getMeshStatistics: computes statistics for the Mesh and each MeshObject

```
This function loads data from Mesh and Mesh Objects and outputs number
of triangle, number of nodes, average triangle quality and minimal
triangle quality, for Mesh and each MeshObject separately.
```

Outputs

```
meshStatistics: structure with following items
                 -numNodes, number of triangulation nodes, double [1 x 3]
                 -numTriangles, number of triangles in
                  the triangulation, double [1 x 3]
                 -minQuality, minimal triangle quality, double [1 x 1]
                 -avgQuality, average triangle quality, double [1 x 1]
                 -objects, above mentioned statistics for each
                  mesh object separately [N x 1]
```

Syntax

```
obj.getMeshStatistics();
```

AToM:+models:+mesh:@Mesh:import1DMesh

import1DMesh: Imports mesh node coordinates and connectivity

Inputs

nodes: node coordinates, double [N x 3]

connectivityList: mesh connectivity, double [N x 2]

name: name of created MeshObject, char [1 x N]

Syntax

```
obj.import1DMesh(nodes, connectivityList, name);
```

AToM:+models:+mesh:@Mesh:import2DMesh

import2DMesh: Imports mesh node coordinates and connectivity

Inputs

nodes: node coordinates, double [N x 3]

connectivityList: mesh connectivity, double [N x 3]

name: name of created MeshObject, char [1 x N]

Syntax

```
obj.import2DMesh(nodes, connectivityList, name);
```

AToM:+models:+mesh:@Mesh:importMesh

importMesh: Imports mesh from specific format

```
SUPPORTED FORMATS
 *.mphtxt
 *.nas - NASTRAN in high-presition data format
 *.geo
```

Inputs

obj: Mesh object, [1 x 1]

fileName: name of imported file, char [1 x N]

name: imported mesh name, char [1 x N]

Syntax

```
obj.importMesh(fileName);
obj.importMesh(fileName, name);
```

AToM:+models:+mesh:@Mesh:meshToPolygon

meshToPolygon: creates AToM geometry polygon from a MeshObject

```
Mesh is converted into polygon and a new Geom object is created. This
function supports only conversion of planar meshes of arbitrary shapes,
with arbitrary number of holes.
```

Inputs

mesh: Mesh object, Mesh [1 x 1]

name: name of Mesh object to delete, char [1 x N]

Syntax

```
obj.meshToPolygon(name);
```

AToM:+models:+mesh:@Mesh:mirrorImportedMesh

mirrorImportedMesh: mirror mesh

Mirrors mesh according to a mirror plane given by its normal.

Inputs

obj: Mesh object, [1 x 1]

name: imported mesh object name, char [1 x N]

normal: mirror plane normal, double [1 x 3]

numCopies: number of copies, double [1 x 1]

origin: point on the mirror plane, double [1 x 3]

Syntax

```
obj.mirrorImportedMesh(name, normal);
obj.mirrorImportedMesh(name, normal, numCopies);
obj.mirrorImportedMesh(name, normal, numCopies, origin);
```

AToM:+models:+mesh:@Mesh:plotMesh

plotMesh: plots 2D mesh

Inputs

TODO options

Syntax

```
obj.plotMesh();
```

AToM:+models:+mesh:@Mesh:plotMeshBoundary

plotMeshBoundary:: plots boudary edges and nodes

Plots boudary edges and nodes of triangulation given by
nodes and connectivityList

Syntax

```
obj.plotMeshBoundary();
```

AToM:+models:+mesh:@Mesh:plotMeshCircumsphere

plotMeshCircumsphere:: plots mesh and its circumpshere

```
Plots circumsphere of triangulation given by nodes and connectivityList
```

Syntax

```
obj.plotMeshCircumsphere();
```

AToM:+models:+mesh:@Mesh:renameImportedMesh

renameImportedMesh: renames imported mesh

```
Renames only imported meshes. Meshes created from AToM geometry share
names with Geom objects.
```

Inputs

obj: Mesh object, [1 x 1]

currentName: object to be removed, char [1 x N]

newName: new mesh object name, char [1 x N]

Syntax

```
obj.renameImportedMesh(currentName, newName);
```

AToM:+models:+mesh:@Mesh:rotateImportedMesh

rotateImportedMesh: scales imported mesh

```
Rotates imported mesh by scaleVector.
```

Inputs

obj: Mesh object, [1 x 1]

name: imported mesh object name, char [1 x N]

rotateAngles: rotate angles, double [1 x 3]

numCopies: number of copies, double [1 x 1]

origin: point on the mirror plane, double [1 x 3]

Syntax

```
obj.rotateImportedMesh(name, rotateAngles);
obj.rotateImportedMesh(name, rotateAngles, numCopies);
obj.rotateImportedMesh(name, rotateAngles, numCopies, origin);
```

AToM:+models:+mesh:@Mesh:scaleImportedMesh

scaleImportedMesh: scales imported mesh

```
Scales imported mesh by scaleVector.
```

Inputs

obj: Mesh object, [1 x 1]

name: imported mesh object name, char [1 x N]

scaleVector: scale vector, double [1 x 3]

numCopies: number of copies, double [1 x 1]

origin: point on the mirror plane, double [1 x 3]

Syntax

```
obj.scaleImportedMesh(name, scaleVector);
obj.scaleImportedMesh(name, scaleVector, numCopies);
obj.scaleImportedMesh(name, scaleVector, numCopies, origin);
```

AToM:+models:+mesh:@Mesh:setElementSizeFromFrequency

setElementSizeFromFrequency: set property lengthFromFrequency of object which is specified by its name

```
This function finds object specified by its name and
changes its lengthFromFrequency
```

Inputs

obj: object of class Mesh, [1 x 1]

name: name of the object, char [1 x N]

value: new value for lengthFromFrequency, logical [1 x 1]

Syntax

```
obj.setElementSizeFromFrequency(name, value);
```

AToM:+models:+mesh:@Mesh:setGlobalDensityFunction

setGlobalDensityFunction: sets property densityFunction of object which is specified by name

This function changes global densityFunction.

Inputs

obj: object of class Mesh, [1 x 1]
func: function handle

TODO: description of allowed function

Syntax

```
obj.setGlobalDensityFunction(functionHandle);
```

AToM:+models:+mesh:@Mesh:setGlobalMeshDensity

setGlobalMeshDensity: set property meshSize to all objects based on frequency

This function finds all objects and changes their meshSize parameter.

Inputs

obj: object of class Mesh, [1 x 1]
densityOfElements: number of elements per wavelength, double [1 x 1]

Syntax

```
obj.setGlobalMeshDensity(densityOfElements);
```

AToM:+models:+mesh:@Mesh:setLocalDensityFunction

setLocalDensityFunction: sets property densityFunction of object which is specified by name

```
This function finds object specified by name and changes its
densityFunction.
```

Inputs

`obj`: object of class Mesh, [1 x 1]

`name`: name of the object, char [1 x N]

`func`: function handle

TODO: description of allowed function

Syntax

```
obj.setLocalDensityFunction(name, functionHandle);
```

AToM:+models:+mesh:@Mesh:setLocalMeshDensity

setLocalMeshDensity: set property meshSize of object which is specified by its name

```
This function finds object specified by its name and changes its meshSize
```

Inputs

`obj`: object of class Mesh, [1 x 1]

`name`: name of the object, char [1 x N]

`meshDensity`: maximal size of mesh, double [1 x 1]

Syntax

```
obj.setLocalMeshDensity(name, meshDensity);
```

AToM:+models:+mesh:@Mesh:setMaxElement

setMaxElement: set property maxElement of object which is specified by name

This function finds object specified by name and changes its maximal element size.

Inputs

obj: object of class Mesh, [1 x 1]

name: name of the object, char [1 x N]

size: maximal element size, double [1 x 1]

Syntax

```
obj.setMaxElement(name, size);
```

AToM:+models:+mesh:@Mesh:setQualityPriority

setQualityPriority: set property qualityPriority

This function enables quality priority settings for mesh. This is rather experimental feature and may help in cases when a standard mesh has too many triangles.

Inputs

obj: object of class Mesh, [1 x 1]

value: new value for flag qualityPriority, logical [1 x 1]

Syntax

```
obj.setQualityPriority(value);
```

AToM:+models:+mesh:@Mesh:setUniformMeshType

setUniformMeshType: set property uniformMeshType

This function sets type of elements used in uniform triangulations

Inputs

obj: object of class Mesh, [1 x 1]

elemType: 'equilateral' or 'right', char [1 x N]

Syntax

```
obj.setUniformMeshType(elemType);
```

AToM:+models:+mesh:@Mesh:setUseLocalMeshDensity

setUseLocalMeshDensity: set property useLocalMeshSize of object which is specified by its name

```
This function finds object specified by its name and changes its useLocalMeshSize
```

Inputs

obj: object of class Mesh, [1 x 1]

name: name of the object, char [1 x N]

value: new value for useLocalMeshSize, logical [1 x 1]

Syntax

```
obj.setUseLocalMeshDensity(name, value);
```

AToM:+models:+mesh:@Mesh:setUseUniformTriangulation

setUseUniformTriangulation: set property isUniform

```
This function finds changes property isUniform
```

Inputs

obj: object of class Mesh, [1 x 1]

value: new value for flag isUniform, logical [1 x 1]

Syntax

```
obj.setUseUniformTriangulation(value);
```

AToM:+models:+mesh:@Mesh:translateImportedMesh

translateImportedMesh: translates imported mesh

Translates imported mesh by translateVector

Inputs

obj: Mesh object, [1 x 1]

name: imported mesh object name, char [1 x N]

translateVector: translate vector, double [1 x 3]

numCopies: number of copies, double [1 x 1]

Syntax

```
obj.translateImportedMesh(name, translateVector);
obj.translateImportedMesh(name, translateVector, numCopies);
```

Namespace

+models/+solvers/+GEP/+customFunctions

Class

AToM:+models:+solvers:+GEP:+customFunctions:postEigSMatrixDecomposition

postEigSMatrixDecomposition: is used as post-eigs function

This function is called after eig/eigs when S matrix is used for computing characteristic modes and eigen-vectors and eigen-numbers are postprocessed here.

Inputs

eigVec: eigen-vectors, double [nEdges x nModes x nFreq]

eigNum: eigen-numbers, double [nModes x nFreq]

iFreq: number of frequency sample, double [1 x 1]

objGEP: GEP object, GEP [1 x 1]

dataFromPreProc: data prom preprocessing function, struct [1 x 1]

Outputs

eigVec: updated eigen-vectors, double [nEdges x nModes x nFreq]

eigNum: updated eigen-numbers, double [nModes x nFreq]

Syntax

```
[eigVec, eigNum] = postEigSMatrixDecomposition(eigVec, eigNum, ...
   iFreq, objGEP, dataFromPreproc)
```

AToM:+models:+solvers:+GEP:+customFunctions:preEigSMatrixDecomposition

preEigSMatrixDecomposition: is used as pre-eigs function

This function is called before eig/eigs when S matrix is used for computing characteristic modes and input matrices are pre-processed here.

Inputs

```
data: input matrices, struct [1 x 1]
  .A: input matrix, double [nEdges x nEdges]
  .B: input matrix, double [nEdges x nEdges]
  .N: normalized matrix, double [nEdges x nEdges]
iFreq: number of frequency sample, double [1 x 1]
objGEP: GEP object, GEP [1 x 1]
```

Outputs

```
data: updated input matrices, struct [1 x 1]
  .A: input matrix, double [nEdges x nEdges]
  .B: input matrix, double [nEdges x nEdges]
  .N: normalized matrix, double [nEdges x nEdges]
dataForPostproc: data prom preprocessing function, struct [1 x 1]
```

Syntax

```
[data, dataForPostproc] = preEigSMatrixDecomposition(data, iFreq, ...
   objGEP)
```

AToM:+models:+solvers:+GEP:+customFunctions:solveSMatrixDecomposition

solveSMatrixDecomposition: run solver for SMatrix decomposition

This function run custom inner solver in GEP when S matrix is used for computing characteristic modes.

Inputs

```
objSolver: object of inner solver, struct [1 x 1]
  .solver: reference to MoM2D, solver [1 x 1]
  .S: alocation for S matrix, double [0 x 0]
  .X: alocation for X matrix, double [0 x 0]
frequencyList: list of frequencies, double [nFreq x 1]
waitBar: waitbar in GEP status window, waitbar [1 x 1]
```

Outputs

```
objSolver: object of inner solver, struct [1 x 1]
  .solver: reference to MoM2D, solver [1 x 1]
  .S: S matrix, double [maxAlpha x nEdges x nFreq]
  .X: X matrix, double [nEdges x nEdges x nFreq]
```

Syntax

```
objSolver = solveSMatrixDecomposition(objSolver, frequencyList, waitBar)
```

```
Note:  _objSolver_ is in general named as solver, but here it is struct.
```

AToM:+models:+solvers:+GEP:+customFunctions:solverSMatrixDecomposition

solverSMatrixDecomposition: create solver for SMatrix decomposition

This function create solver when S matrix is used for computing characteristic modes.

Inputs

```
objGEP: GEP object, GEP [1 x 1]
```

Outputs

```
mySolver: structure of my solver, struct [1 x 1]
 .solver: reference to MoM2D, solver [1 x 1]
 .S: alocation for S matrix, double [0 x 0]
 .X: alocation for X matrix, double [0 x 0]
```

Syntax

```
mySolver = solverSMatrixDecomposition(objGEP)
```

Namespace

+models/+solvers/+GEP

Class +models/+solvers/+GEP/@GEP

AToM:+models:+solvers:+GEP:@GEP:GEP

GEP: creates solver using General Eigenvalue Problem

Main class of GEP

Syntax

```
myGEP = models.solvers.GEP()
```

AToM:+models:+solvers:+GEP:@GEP:clearInputs

clearInputs: clear inputs

Clear input matrices, frequency list and inner solver object.

Syntax

```
objGEP.clearInputs()
```

AToM:+models:+solvers:+GEP:@GEP:clearOutputs

clearOutputs: clear outputs

Clear all outputs in results structure.

Syntax

```
objGEP.clearOutpus()
```

AToM:+models:+solvers:+GEP:@GEP:defaultControls

defaultControls: provide struct of control handles for given inner solver

Provide struct of control handles for given inner solver. Fields contains string with handles which are set to corresponding GEP propertis when solver is started.

Inputs

innerSolver: name of inner solver, char [1 x N]

Outputs

defControls: handles for controlling inner solver, struct [1 x 1]
  .innerSolverHndl: get object of inner solver, char [1 x N]
  .innerSolverSolve: solve inner solver, char [1 x N]
  .innerSolverGetA: get matrix A from inner solver, char [1 x N]
  .innerSolverGetB: get matrix B from inner solver, char [1 x N]
  .innerSolverGetN: get matrix N from inner solver, char [1 x N]
  .eigRunPreAndPostprocessing: run function before and after eig/eigs, logical [1 x 1]

eigPreprocessing: eig/eigs pre-processing, char [1 x N]

eigPostprocessing: eig/eigs post-processing, char [1 x N]

Syntax

```
defControls = objGEP.defaultControls(innerSolver)
```

AToM:+models:+solvers:+GEP:@GEP:getDefaultProperties

getDefaultProperties: returns structure of default GEP properties

Structure with default values for GEP properties.

Outputs

defaultProperties: struture of default properties, struct [1 x 1]
  .propertyName: contain default value, any

Syntax

```
defaultProperties = objGEP.getDefaultProperties()
```

AToM:+models:+solvers:+GEP:@GEP:getPropertyList

getPropertyList: returns names of properties

Get names of GEP properties which can be set by method setProperties().

Outputs

`defaultProperties`: struture of default properties, struct [1 x 1]

Syntax

```
defaultProperties = objGEP.getDefaultProperties()
```

AToM:+models:+solvers:+GEP:@GEP:resetPropertiesToDefault

resetPropertiesToDefault: reset properties of GEP to default values

Reser properties to default

Syntax

```
objGEP.resetPropertiesToDefault()
```

AToM:+models:+solvers:+GEP:@GEP:setCorrInputData

setCorrInputData: set corrInputData as corrInputData to GEP properties

Store required data for correlation from inner solver.

Inputs

`corrInputData`: structure with datas, struct [1 x 1]

Syntax

```
objGEP.setCorrInputData(corrInputData)
```

AToM:+models:+solvers:+GEP:@GEP:setFrequencyList

setFrequencyList: set list of frequencies to GEP properties

Set frequency list

Inputs

frequencyList: frequency list, double [nFreq x 1]

Syntax

```
objGEP.setFrequencyList(frequencyList)
```

AToM:+models:+solvers:+GEP:@GEP:setMatrices

setMatrices: set all input matrices to GEP properties

Set all three matrices as input

Inputs

A: input matrix, double [nEdges x nEdges x nFreq]

B: input matrix, double [nEdges x nEdges x nFreq]

N: normalized matrix, double [nEdges x nEdges x nFreq]

Syntax

```
objGEP.setMatrices(A, B, N)
```

AToM:+models:+solvers:+GEP:@GEP:setMatrix

setMatrix: set data to given input to GEP properties

Set selected matrix

Inputs

nameOfMatrix: matrix name ('A' or 'B' or 'N'), char [1 x 1]

matrix: input matrix, double [nEdges x nEdges x nFreq]

Syntax

```
objGEP.setMatrix(nameOfMatrix, matrix)
```

AToM:+models:+solvers:+GEP:@GEP:setMatrixA

setMatrixA: set A as matrixA to GEP properties

Set matrix A.

Inputs

A: input matrix, double [nEdges x nEdges x nFreq]

Syntax

```
objGEP.setMatrixA(A)
```

AToM:+models:+solvers:+GEP:@GEP:setMatrixB

setMatrixB: set B as matrixB to GEP properties

Set matrix B.

Inputs

B: input matrix, double [nEdges x nEdges x nFreq]

Syntax

```
objGEP.setMatrixB(B)
```

AToM:+models:+solvers:+GEP:@GEP:setMatrixN

setMatrixN: set N as matrixN to GEP properties

Set matrix N.

Inputs

N: normalized matrix, double [nEdges x nEdges x nFreq]

Syntax

```
objGEP.setMatrixN(N)
```

AToM:+models:+solvers:+GEP:@GEP:solve

solve: solve GEP

Run GEP solver

Inputs

frequencyList: list of frequencies, double [nFreq x 1]

Syntax

```
objGEP.solve()
objGEP.solve(frequencyList)
```

If *frequencyList* is set, frequencies in objGEP.frequencyList are ingored.

AToM:+models:+solvers:+GEP:@GEP:updateResult

updateResult: update given result of GEP

Set new data to given result.

Inputs

result: result, char [1 x N]

newData: new data, any

Syntax

```
objGEP.updateResult(result, newData)
```

Class

AToM:+models:+solvers:+GEP:assignEigNumbers

assignEigNumbers: assign eigen-numbers to modes track

This function assign unsorted eigen-numbers and eigen-vectors to track defined in *modesTrack* and make modes sorted.

Inputs

eigVec: original eigen vectors, double [nEdges x nModes x nFreq]

eigNum: original eigen numbers, double [nModes x nFreq]

modesTrack: modes tracking matrix, double [nModes x nFreq]

gepOptions: options settings, struct [1 x 1]

Outputs

assignedEigVec: assigned eigen vectors, double [nEdges x maxUsedModeNumber x nFreq]

assignedEigNum: assigned eigen numbers, double [maxUsedModeNumber x nFreq]

Syntax

```
[assignedEigVec, assignedEigNum] = ...
   assignEigNumbers(eigVec, eigNum, modesTrack)
[assignedEigVec, assignedEigNum] = ...
   assignEigNumbers(eigVec, eigNum, modesTrack, gepOptions)
```

AToM:+models:+solvers:+GEP:computeAlpha

computeAlpha: compute matrix of alpha coefficients

Coeffieicnet alpha is computed from modal excitation coefficient.

Inputs

Vi: excitation vectors [nEdges x nFreq]

eigVec: eigen vectors [nEdges x nModes x nFreq]

eigNum: eigen numbers, double [nModes x nFreq]

Outputs

alpha: alpha coefiffients [nModes x nFreq]

Syntax

```
alpha = computeAlpha(Vi, eigVec, eigNum)
```

AToM:+models:+solvers:+GEP:computeBeta

computeBeta: compute matrix of beta coefficients

Coefficient betta is computed from alpha coefficient

Inputs

Vi: excitation vectors, double [nEdges x nFreq]

eigVec: eigen vectors, double [nEdges x nModes x nFreq]

eigNum: eigen numbers, double [nModes x nFreq]

Outputs

beta: beta coefficients, double [nModes x nModes x nFreq]

Syntax

```
beta = computeBeta(Vi, eigVec, eigNum)
```

AToM:+models:+solvers:+GEP:computeCorrTable

computeCorrTable: compute correlation table between eigen-vectors

General function deciding which core for correlation table computing will be used.

Inputs

eigVec: eigvectors, double [nEdges x nModes x nFreq]

corrInputData: data for computing correlation, struct, [1 x 1]

gepOptions: options settings, struct [1 x 1]

corrTable: correlation table, double [nModes x nModes x nFreq-1]

statusWindow: status window for GUI, GEP status window [1 x 1]

Outputs

corrTable: correlation coeficients, double [nModes x nModes x nFreq-1]

Syntax

```
[corrTable, corrInputData] = computeCorrTable(eigVec)
[corrTable, corrInputData] = computeCorrTable(eigVec, corrInputData)
[corrTable, corrInputData] = computeCorrTable(eigVec, corrInputData, ...
   gepOptions)
[corrTable, corrInputData] = computeCorrTable(eigVec, corrInputData, ...
   gepOptions, corrTable,)
[corrTable, corrInputData] = computeCorrTable(eigVec, corrInputData, ...
   gepOptions, corrTable, statusWindow)
```

If *corrTable* is given as inputs, new values are computed only on positions, where corrTable(:, :, n) is matrix of NaNs.

AToM:+models:+solvers:+GEP:computeCorrTableFF

computeCorrTableFF: compute correlation using Far-Field computation

Far-field for each mode is computed and than 2D correlation coefficient is stored in *corrTable*.

Inputs

eigVec: eigvectors, double [nEdges x nModes x nFreq]

corrInputData: data for computing correlation, struct [1 x 1]
  .mesh: required - mesh structure, struct [1 x 1]
  .basisFcns: required - basis functions, struct [1 x 1]
  .frequencyList: required - list of frequencies, double [nFreq x 1]
  .FF: optional - result struct with modal far-fields, struct [1 x 1]

gepOptions: options settings, struct [1 x 1]

corrTable: correlation table, double [nModes x nModes x nFreq-1]

statusWindow: status window for GUI, GEP status window [1 x 1]

Outputs

corrTable: correlation coeficients, double [nModes x nModes x nFreq-1]

corrInputData: data for computing correlation, struct [1 x 1]

Syntax

```
[corrTable, corrInputData] = computeCorrTableFF(eigVec)
[corrTable, corrInputData] = computeCorrTableFF(eigVec, corrInputData)
[corrTable, corrInputData] = computeCorrTableFF(eigVec, ...
   corrInputData, gepOptions)
[corrTable, corrInputData] = computeCorrTableFF(eigVec, ...
   corrInputData, gepOptions, corrTable)
[corrTable, corrInputData] = computeCorrTableFF(eigVec, ...
   corrInputData, gepOptions, corrTable, statusWindow)
```

If *corrTable* is given as inputs, new values are computed only on positions, where corrTable(:, :, n) is matrix of NaNs.
If *corrInputData* contains far-fields, new values are computed only on positions, where farFields(:, :, iFreq, iMode) is matrix of NaNs.

AToM:+models:+solvers:+GEP:computeCorrTableII

computeCorrTableII: compute correlation between eigen-vectors

Correaltion coefficient is computed as correlation between two eigen- vectors on next frequency sample.

Inputs

eigVec: eigvectors, double [nEdges x nModes x nFreq]

corrInputData: data for computing correlation, struct [1 x 1]

gepOptions: options settings, struct [1 x 1]

corrTable: correlation table, double [nModes x nModes x nFreq-1]

statusWindow: status window for GUI, GEP status window [1 x 1]

Outputs

corrTable: correlation coeficients, double [nModes x nModes x nFreq-1]

corrInputData: data for computing correlation, struct [1 x 1]

Syntax

```
[corrTable, corrInputData] = computeCorrTableII(eigVec)
[corrTable, corrInputData] = computeCorrTableII(eigVec, corrInputData)
[corrTable, corrInputData] = computeCorrTableII(eigVec, ...
   corrInputData, gepOptions)
[corrTable, corrInputData] = computeCorrTableII(eigVec, ...
   corrInputData, gepOptions, corrTable)
[corrTable, corrInputData] = computeCorrTableII(eigVec, ...
   corrInputData, gepOptions, corrTable, statusWindow)
```

If *corrTable* is given as inputs, new values are computed only on positions, where corrTable(:, :, n) is matrix of NaNs.

AToM:+models:+solvers:+GEP:computeCorrTableIRI

computeCorrTableIRI: compute correlation table using surface correlation

Surface correaltion is used to compute correlation coefficient

Inputs

eigVec: eigvectors, double [nEdges x nModes x nFreq]

corrInputData: data for computing correlation, struct [1 x 1]
  .R: required - real part of impedance matrix, double
  [nEdges x nEdges x nFreq]

gepOptions: options settings, struct [1 x 1]

corrTable: correlation table, double [nModes x nModes x nFreq-1]

statusWindow: status window for GUI, GEP status window [1 x 1]

Outputs

corrTable: correlation coeficients, double [nModes x nModes x nFreq-1]

corrInputData: data for computing correlation, struct [1 x 1]

Syntax

```
[corrTable, corrInputData] = computeCorrTableIRI(eigVec)
[corrTable, corrInputData] = computeCorrTableIRI(eigVec, corrInputData)
[corrTable, corrInputData] = computeCorrTableIRI(eigVec, ...
   corrInputData, gepOptions)
[corrTable, corrInputData] = computeCorrTableIRI(eigVec, ...
   corrInputData, gepOptions, corrTable)
[corrTable, corrInputData] = computeCorrTableIRI(eigVec, ...
   corrInputData, gepOptions, corrTable, statusWindow)
```

If *corrTable* is given as inputs, new values are computed only on positions, where corrTable(:, :, n) is matrix of NaNs.

AToM:+models:+solvers:+GEP:computeCorrelation

computeCorrelation: compute correlation of eig vectors

Compute correlation coefficients between given vectors or between vector and matrix.

Inputs

vec0: eigvector of this Mode at this frequency, double [nMode x 1]

VEC1: eigvector of all modes at next frequency, double [nMode x nMode]

Outputs

corrTable: table of correlation between modes, double, [1 x nMode]

Syntax

```
corrTable = computeCorrelation(vec0, VEC1)
```

AToM:+models:+solvers:+GEP:computeCorrelation2D

computeCorrelation2D: compute correlation of 2D matrixes

Compute correlation coefficient between two 2D matrices.

Inputs

a, b: 2D matrices [double]

Outputs

corrCoeff: correlation coefficient, double [1 x 1]

Syntax

```
corrCoeff = computeCorrelation2D(a, b)
```

AToM:+models:+solvers:+GEP:computeFF

computeFF: compute modal far-fields

Compute modal far-fields.

Inputs

eigVec: eigvectors, double [nEdges x nModes x nFreq]

mesh: mesh structure, struct [1 x 1]

basisFcns: basis functions, struct [1 x 1]

frequencyList: list of frequencies, double [nFreq x 1]

theta: vector of theta points, double [nTheta x 1]

phi: vector of phi points, double [nPhi x 1]

gepOptions: options settings, struct [1 x 1]

FF: far-field matrice, double [nTheta x nPhi x nFreq x nMode]

statusWindow: status window for GUI, GEP status window [1 x 1]

Outputs

FF: far-field matrice, double [nTheta x nPhi x nFreq x nMode]

Syntax

```
FF = computeFF(eigVec, mesh, basisFcns, frequencyList, theta, phi)
FF = computeFF(eigVec, mesh, basisFcns, frequencyList, theta, phi, ...
   gepOptions)
FF = computeFF(eigVec, mesh, basisFcns, frequencyList, theta, phi, ...
   gepOptions, FF)
FF = computeFF(eigVec, mesh, basisFcns, frequencyList, theta, phi, ...
   gepOptions, FF, statusWindow)
```

AToM:+models:+solvers:+GEP:computeModalExcitation

computeModalExcitation: compute matrix of modal excitation factors

Modal Excitation coeeficients are computed as

Inputs

`Vi`: excitation vectors [nEdges x nFreq]

`eigVec`: eigen vectors [nEdges x nModes x nFreq]

Outputs

`modalE`: modal excitation factors [nModes x nModes x nFreq]

Syntax

```
modalE = computeModalExcitation(Vi, eigVec)
```

AToM:+models:+solvers:+GEP:computeModalQ

computeModalQ: compute modal quality factor Q

Quality factor Q is computed as

Inputs

`frequencyList`: list of frequencies [nFreq x 1]

`eigNum`: eigen vectors [nModes x nFreq]

Outputs

`modalQ`: modalQ [nModes x nFreq]

Syntax

```
modalQ = computeModalQ(frequencyList, eigNum)
```

AToM:+models:+solvers:+GEP:computeModalSignificance

computeModalSignificance: compute matrix of modal significance factors

Modal significance factor is computed as abs(1/(1 + 1i*lambda))

Inputs

eigNum: eigen numbers, double [nModes x nFreq]

Outputs

modalS: modal significance factors [nModes x nFreq]

Syntax

```
modalS = computeModalSignificance(eigNum)
```

AToM:+models:+solvers:+GEP:computePiFactor

computePiFactor: compute matrix of Pi factors

Pi factor is computed as max(abs(J_n)) / (1 + lambda_n^2)

Inputs

IorJ: eigVec or abs(J) [nEdges x nModes x nFreq]

eigNum: eigen numbers, double [nModes x nFreq]

mesh: mesh structure, struct [1 x 1]

basisFcns: basis functions, struct [1 x 1]

Outputs

PiFac: Pi factor [nModes x nFreq]

Syntax

```
PiFac = computePiFactor(Jabs, eigNum)
PiFac = computePiFactor(eigVec, eigNum, mesh, basisFcns)
```

Reference
J. L. T. Ethier and D. A. Mcnamara, "Modal significance measure in characteristic mode analysis of radiating structures," in Electronics Letters, vol. 46, no. 2, pp. 107-108, January 21 2010.

AToM:+models:+solvers:+GEP:connectModes

connectModes: connect interupted modes

Try to connect modes with previously closed modes.

Inputs

eigVec: eigvectors, double [nEdges x nModes x nFreq]

modesTrack: modes tracking matrix, double [nModes x nFreq]

gepOptions: options settings, struct [1 x 1]

Outputs

modesTrack: modes tracks with connections, double [nModes x nFreq]

Syntax

```
modesTrack = connectModes(eigVec, modesTrack)
modesTrack = connectModes(eigVec, modesTrack, gepOptions)
```

AToM:+models:+solvers:+GEP:delNegValues

delNegValues: delete negative eigen-values of matrix

Eigen numbers of matrix A are computed, negative are discarded and from positive are constrinct new matrix A

Inputs

A: input matrix, double [N x M]

Outputs

A: output matrix, double [N x M]

Syntax

```
A = delNegValues(A)
```

AToM:+models:+solvers:+GEP:discardModes

discardModes: discard modes according to specification

Discard modes according to settings set by user.

Inputs

eigVec: eigen vectors, double [nEdges x maxUsedModeNumber x nFreq]

eigNum: eigen numbers, double [maxUsedModeNumber x nModes]

modesTrack: modes tracking matrix, double [nModes x nFreq]

opt: options settings, struct [1 x 1]

Outputs

eigVec: eigen vector, double [nEdges x maxUsedModeNumber* x nFreq]

eigNum: eigen numbers, double [maxUsedModeNumber* x nFreq]

*****: Number of output modes may be different than number of input modes.

Syntax

```
[eigVec, eigNum] = discardModes(eigVec, eigNum, modesTrack)
[eigVec, eigNum] = discardModes(eigVec, eigNum, modesTrack, gepOptions)
```

AToM:+models:+solvers:+GEP:findMaxUsedModeNumber

findMaxUsedModeNumber: find max used mode number

Find maximal used mode number in *modesTrack* matrix.

Inputs

modesTrack: modes tracking matrix, double [nModes x nFreq]

Outputs

maxUsedModeNumber: maximal used mode number, double [1 x 1]

Syntax

```
maxUsedModeNumber = findMaxUsedModeNumber(modesTrack)
```

AToM:+models:+solvers:+GEP:gep

gep: solve Generalized Eigenvalue Problem

Compute GEP: A eigVec = eigNum B eigVec

Inputs

A: `input matrix, double [nEdges x nEdges]`

B: `input matrix, double [nEdges x nEdges]`

N: `normalized matrix, double [nEdges x nEdges]`

gepOptions: `options settings, struct [1 x 1]`

if N is empty or NaN, B is used to normalization

Outputs

eigVec: `eigen-vectors, double [nEdges x nModes]`

eigNum: `eigen-numbers, double [nModes x 1]`

INI: `reacted power, double, [nModes x 1]`

Syntax

```
[eigVec, eigNum, INI] = gep(A, B)
[eigVec, eigNum, INI] = gep(A, B, N)
[eigVec, eigNum, INI] = gep(A, B, N, gepOptions)
```

AToM:+models:+solvers:+GEP:getOption

getOption: return vale of option

Return value of required option from options structure.

Inputs

optionsStruct: `structure with options, struct [1 x 1]`

optionName: `name of required option, char [1 x N]`

Outputs

value: `value of required option, any`

Syntax

```
value = getOption(optionsStruct, optionName)
```

description

AToM:+models:+solvers:+GEP:prepareResultStruct

prepareResultStruct: prepare struct for result

Create structure for result

Inputs

description: name of this result, char [1 x N]

dimensions: dimensions of this result, char [1 x N]

freqDepDim: frequency-dependent dimension, double [1 x 1]

units: units of this result, char [1 x N]

data: data of this result, double [any]

Outputs

```
resultStruct: structure of result, struct [1 x 1]
  .description: description of result, char [1 x N]
  .data: results data, double [any]
  .size: size of data, double [1 x N]
  .dimensions: description of dimensions, char [1 x N]
  .units: units of data, char, [1 x N]
  .frequencyDependentDimension: number of frequency-dependent
                                           dimension, double [1 x 1]
```

Syntax

```
resultStruct = prepareResultStruct(description, dimensions, ...
   freqDepDim)
resultStruct = prepareResultStruct(description, dimensions, ...
   freqDepDim, units)
resultStruct = prepareResultStruct(description, dimensions, ...
   freqDepDim, units, data)
```

If result does not have frequency-dependent dimension, set $freqDepDim = 0$.

AToM:+models:+solvers:+GEP:procgep

procgep: solve Generalized Eigenvalue Problem with pre-&post- processing

Generalized Eigenvalue Problem: A eigVec = eigNum B eigVec

Inputs

`A`: input matrix, double [nEdges x nEdges]

`B`: input matrix, double [nEdges x nEdges]

`N`: normalized matrix, double [nEdges x nEdges]

`gepOptions`: options settings, struct [1 x 1]

if N is empty or NaN, B is used to normalization

Outputs

`eigVec`: eigen-vectors, double [nEdges x nModes]

`eigNum`: eigen-numbers, double [nModes x 1]

Syntax

```
[eigVec, eigNum, INI] = procgep(A, B)
[eigVec, eigNum, INI] = procgep(A, B, N)
[eigVec, eigNum, INI] = procgep(A, B, N, gepOptions)
```

AToM:+models:+solvers:+GEP:scanModesProperties

scanModesProperties: returns modes properties from given modesTrack

Return informations about modes from *modesTrack* matrix

Inputs

`modesTrack`: modes tracking matrix, double [nModes x nFreq]

`gepOptions`: options settings, struct [1 x 1]

Outputs

```
modesProp: output structure, struct [1 x 1]
  .maxUsedModeNumber: maximal value in modesTrack, double, [1 x 1]
  .start: indicator where modes start, double [1 x maxUsedModeNumber]
  .end: indicator where modes end, double [1 x maxUsedModeNumber]
  .length: length of modes, double [1 x maxUsedModeNumber]
```

Syntax

```
modesProp = scanModesProperties(modesTrack)
modesProp = scanModesProperties(modesTrack, gepOptions)
```

AToM:+models:+solvers:+GEP:solve

solve: run GEP solver

Run GEP solver

Inputs

solver: GEP solver object, GEP [1 x 1]

frequencyList: list of frequencies, double [nFreq x 1]

Syntax

```
solve(objGEP)
solve(objGEP, frequencyList)
```

AToM:+models:+solvers:+GEP:symmetrizeMatrix

symmetrizeMatrix: make the matrix symmetric

Make the matrix symmetric

Inputs

A: input matrix, double [N x M]

Outputs

symA: symmetrized matrix, double [N x M]

Syntax

```
symA = symmetrizeMatrix(A)
```

AToM:+models:+solvers:+GEP:trackingCM

trackingCM: track modes with respect to corrTable

Track and sort modes according to correlation coefficients in correlation table.

Inputs

eigVec: unsorted eigen-vectors, double [nEdges x nModes x nFreq]

eigNum: unsorted eigen-numbers, double [nModes x nFreq]

corrInputData: additional data for correlation comp., struct [1 x 1]

gepOptions: options settings, struct [1 x 1]

corrTable: correlationTable, double [nModes x nModes x nFreq-1]

Outputs

outStruct: output structure, struct [1 x 1]
  .modesTrack: modes track in unsorted values, double, [nModes x nFreq]
  .eigNumSorted: sorted eigen numbers, double [maxUsedModeNumber x nFreq]
  .eigVecSorted: sorted eigen vectors, double [nEdges x maxUsedModeNumber x nFreq]
  .corrTable: correlation table, double [nModes x nModes x nFreq-1]

corrInputData: additional data for correlation comp., struct [1 x 1]

Syntax

```
[outStruct, corrInputData] = trackingCM(eigVec, eigNum)
[outStruct, corrInputData] = trackingCM(eigVec, eigNum, ...
   corrInputData)
[outStruct, corrInputData] = trackingCM(eigVec, eigNum, ...
   corrInputData, gepOptions)
[outStruct, corrInputData] = trackingCM(eigVec, eigNum, ...
   corrInputData, gepOptions, corrTable)
[outStruct, corrInputData] = trackingCM(eigVec, eigNum, ...
   corrInputData, gepOptions, corrTable, statusWindow)
```

Namespace

+models/+solvers/+MoM2D/+computation

AToM:+models:+solvers:+MoM2D:+computation:getJInPoints

getJInPoints: returns values of current density in general points

This method returns values of the current density and its divergence evaluated in general points given by the user or in the triangle centroids. The last output variable *points* contains coordinates of the points for which the values were computed. Procedure: ~~~~~~~ Get topology 1.) find triangles for points 2.) find basis functions for triangles Solve topology 3.) accumulate basis function contributions to triangles 4.) accumulate triangle contributions to points

Inputs

mesh: computational mesh

basisFcns: information about basis functions

iVec: current density coefficients, double [#unknowns x #frequencies]

points: Cartesian coordinates of the points, double [#points x 3]
This parameter is optional.

Outputs

Jx: x component of the current density, double [#unknowns x #frequencies]

Jy: y component of the current density, double [#unknowns x #frequencies]

Jz: z component of the current density, double [#unknowns x #frequencies]

divJ: divergence of the current density, double [#unknowns x #frequencies]

points: Cartesian coordinates of the points, double [#points x 3]

Syntax

```
getJInPoints(obj, jVec, points)
```

User defines points where the current density will be evaluated.

```
getJInPoints(obj, jVec)
```

The points are not user-defined. Triangle centroids are used instead.

Namespace

+models/+solvers/+MoM2D

ATоM:+models:+solvers:+MoM2D:possibleResultRequests

possibleResultRequests: returns list of output request which can be used

List of the result requests is formed as a list of result definitions in namespace "resultDefs".

Namespace

+models/+solvers

Class +models/+solvers/@BEM

AToM:+models:+solvers:@BEM:getCurrentsOnPorts

returns: current on ports

Method returns scalar or vector of currents on every single port. These currents is needed for computing of driving impedance resp. s-parameters

Outputs

`currentOnPorts`: current on ports, complex [N x 1]

Syntax

```
currentOnPorts = getCurrentsOnPorts()
```

Returned currents on the ports is a vector [N x 1]

AToM:+models:+solvers:@BEM:getParametersS

returns: mutual s-parameters

Method returns mutual s-parameters based on mutual z-parameters. For driving s-parameters is need to use computed voltage on ports (using getVoltage method) and also currents on ports (using getCurrentOnPorts). Method returns mutual s-parameters based on mutual z-parameters. For driving s-parameters is need to use computed voltage on ports (using getVoltage method) and also currents on ports (using getCurrentOnPorts).

Inputs

`Z0`: characteristic impedance, double [1 x 1]

Outputs

`sParameters`: mutual s-parameters, complex [N x 1 x N]

Syntax

```
sParameters = getParametersS(Z0)
```

Method returns mutual s-parameters base on characteristic impedance Z0.

AToM:+models:+solvers:@BEM:getParametersZ

returns: z-parameters from BEM

Method returns mutual z-parameters based on impedance matrix obtained by Boundary Elements method. If the impedance matrix is not computed yet, comment to command window warns user, that solving is needed first.

Outputs

`parametersZ`: matrix of z-parameters, complex [N x N x N]

Syntax

```
parametersZ = getParametersZ
```

Method returns matrix of the z-parameters. The third dimension determines frequency depedency.

AToM:+models:+solvers:@BEM:getPeripherySamples

returns: the coordinaate of samples on circuit periphery

Method returns the (x,y,z) coordinates of samples on periphery, including the periphery of holes. This data are needed for radiation pattern computing, also for displaying of voltage on the periphery.

Outputs

`mesh.boundaryPoints`: coordinates of periphery samples, struct

Syntax

```
mesh = getPeripherySamples()
```

Method returns the x,y,z coordinates of periphery samples for other computing.

AToM:+models:+solvers:@BEM:getVoltage

returns: the voltage on samples of periphery

Method is computing both, the voltage on periphery of investigated planar circuit and voltage on every single port. If the impedance matrix is known and at least one port is feeding the circuit, the method returns voltage on periphery for all set frequencies.

Outputs

peripheryVoltage: `Voltage on periphery for all frequencies, complex [N x 1 x N]`

portVoltage: `Voltage on every single port for all frequencies, complex [N x 1 x N]`

Syntax

```
peripheryVoltage = getVoltage()
```

Returned voltage on the periphery is a vector [N x 1] listed to third dimension coresponding with frequency.

```
[peripheryVoltage, portVoltage] = getVoltage()
```

Returned both, voltage on the periphery is a vector [N x 1] and voltage on every single port, listed to third dimension coresponding with frequency.

AToM:+models:+solvers:@BEM:initializeSideAccess

initializeSideAccess: initializes BEM object for work from side access

The method is for initialization of BEM class without AToM session. User has to keep the structure of inputs needed for computing of the impedance matrix. The method is only for skilled users!

Inputs

mesh: `mesh.getBoundaryMesh.nodes11 = boundaryPoints, double [N x 2]`
`mesh.getBoundaryMesh.nodes12 = holeBoundaryPoints, double [N x 2]`

physics:

Syntax

```
initializeSideAccess(boundaryPoints, nodes, physics, materials, d)
```

This is only one syntax for initialization for side access.

AToM:+models:+solvers:@BEM:setDielectric

setDielectric: sets dielectric part of computed substrate.

The method sets metal part of substrate from dielectric database. Dielectric is chosen according to his name

Inputs

dielectricName: `name of dielectric material, Char [1 x N]`

Syntax

```
setDielectric(dielectricName)
```

Dielectric is set using his name in database of materials

AToM:+models:+solvers:@BEM:setMetal

setMetal: sets metal part of the computed substrate.

The method sets metal part of substrate from metal database. Metal is chosen according to his name

Inputs

metalName: `name of metal material, Char [1 x N]`

Syntax

```
setMetal(metalName)
```

Metal is set using his name in database of materials.

AToM:+models:+solvers:@BEM:setSubstrateHeight

setSubstrateHeight: sets Substrate height (for BEM solver)

The method sets substrate height (in meters)

Inputs

value: `value of substrate height, double [1 x 1]`

Syntax

```
setSubstrateHeight(value)
```

Substrate height is set to "value" in meter

AToM:+models:+solvers:@BEM:solve

solve: starts calculate impedance matrix by Boundary Elements Method

This method validates if important properties of BEM class are set correctly. Then it sets everything else for calculation of impedance matrix. Result of computation is the impedance matrix.

Inputs

`userFrequency`: frequencies defined by user, double [N x 1]

Syntax

`solve()`

Method starts compute impedance matrix using of frequency list from physics class.

`solve(userFrequency)`

Method starts compute impedance matrix using the frequency list from function input.

Namespace

+models/+utilities/+geomPublic

Class

AToM:+models:+utilities:+geomPublic:arePointsInPolygon

arePointsInPolygon: determine if points are in polygon or not

This function determines if 2D points are inside or outside of specified 2D polygon.

Inputs

`points`: set of points, double [nPoints x 2]

`polygon`: nodes of polygon in CCW order, double [nNodes x 3]

`holeSeg`: hole indicator (1 solid, >1 hole), double [nNodes x 1]

`tol`: geom precision, double [1 x 1]

Outputs

`areIn`: are points in or not, logical [nPoints x 1]

Syntax

```
areIn = models.utilities.geomPublic.arePointsInPolygon(points, polygon)
```

Function arePointsInPolygon determines if 2D points *points* are inside or outside of a polygon *polygon*. The computation is based on winding number according to http://geomalgorithms.com/a03-_inclusion.html.

AToM:+models:+utilities:+geomPublic:arePointsInSamePlane

arePointsInSamePlane: determine if points are in same plane

This function determines if a set of 3d points is co-planar / all points lie in teh same plane.

Inputs

`points`: set of points, double [nPoints x 2]

`tol`: optional, geom precision, double [1 x 1]

Outputs

`areInSame`: are points in same plane, logical [1 x 1]

`areColinear`: are points in one line, logical [1 x 1]

Syntax

```
[areInSame, areColinear] = ...
models.utilities.geomPublic.arePointsInSamePlane(points, tol)
```

Function arePointsInSamePlane determines if 3D points *points* lie all in the sam plane according to numerical precision *tol*.

AToM:+models:+utilities:+geomPublic:checkSamePoints

checkSamePoints: determine if points are same according to tolerance

This function determines if points are same (distance lower than tolerance) and replaces points with same entries.

Inputs

`points`: set of points, double [nPoints x 2/3]

`tolernace`: geom precision, double [1 x 1]

Outputs

`checkedPoints`: points rounded to tolerance, doble [nPoints x 2/3]

Syntax

```
checkedPoints = models.utilities.geomPublic.checkSamePoints( ...
points, tolerance)
```

Function checkSamePoints determines if 2D points *points* have smae points, that are closer than tolernace. In that case, points are rplaced bz the first occurance.

AToM:+models:+utilities:+geomPublic:crossProduct

crossProduct: find cross product between two sets of vectors

This function determines crossproduct of two sets of 3D vectors.

Inputs

`vect1`: set of vectors, double [nV1 x 3]

`vect2`: set of vectors, double [nV2 x 3]

`mode`: optional, normalization of cross product vector, char [1 x N]

Outputs

```
crossProd: cross product, struct [1 x 1]
  .x: x-coordinates, double [nV1 x nV2]
  .y: y-coordinates, double [nV1 x nV2]
  .z: z-coordinates, double [nV1 x nV2]
```

Syntax

```
crossProd = models.utilities.geomPublic.crossProduct(vect1, vect2)
```

Function crossProduct determines cross product of two sets of 3D vectors determined bz *vect1* and *vect2*.

```
crossProd = models.utilities.geomPublic.crossProduct(vect1, vect2, mode)
```

Mode of output vector specified by user in *mode*. Default mode is 'normalized', other option is 'notModified'.

ATоM:+models:+utilities:+geomPublic:distanceFromPointsToLines

distanceFromPointsToLines: compute distance from points to lines

This function computes perpendicular distance between sets of points and lines defined by two points in 3D.

Inputs

```
points: set of points, double [nPoints x 3]
lines: struct [1 x nLines]
      .startPoint: start points, double [nLines x 3]
      .endPoint: end point, double [nLines x 3]
```

Outputs

```
distMatrix: distance from Points to Lines, double [nPoints x nLines]
parameters: parametric projection of P. to L., double [nPoints x nLines]
```

Syntax

```
[distMatrix, parameters] = models.utilities.geomPublic. ...
distanceFromPointsToLines(points, lines)
```

Function distanceFromPointsToLines computes pairwise distances between set of points defined in *points* and set of lines defined in struct *lines*. This struct is formed by two points (*lines.startPoint* and *lines.endPoint*).

ATоM:+models:+utilities:+geomPublic:distanceFromPointsToPlanes

distanceFromPointsToPlanes: compute distance from points to planes

This function computes distance between sets of points and planes defined in 3D.

Inputs

```
points: set of points, double [nPoints x 3]
planes: struct [1 x nPlanes]
      .normal: normal vector, double [1 x 3]
      .pointIn: point on a plane, double [1 x 3]
```

Outputs

```
distMatrix: distance between points, double [nPoints x nPlanes]
```

Syntax

```
distMatrix = models.utilities.geomPublic. ...
distanceFromPointsToPlanes(points, planes)
```

Function distanceFromPointsToPlanes computes pairwise distances between set of points defined in *points* and set of planes defined in struct *planes*. This struct is formed by planes normal vector (*planes.normal*) and a point on the plane (*planes.pointIn*).

ATOM:+models:+utilities:+geomPublic:dotProduct

dotProduct: find dot product between two sets of vectors

This function determines dotproduct of two sets of 3D vectors.

Inputs

vect1: `set of vectors, double [nV1 x 3]`

vect2: `set of vectors, double [nV2 x 3]`

mode: `optional, normalization of dot product vector, char [1 x N]`

Outputs

dotProd: `dot product, struct [nV1 x nV2]`

Syntax

```
dotProd = models.utilities.geomPublic.dotProduct(vect1, vect2)
```

Function dotProduct determines dot product of two sets of 3D vectors determined bz *vect1* and *vect2*.

```
dotProd = models.utilities.geomPublic.dotProduct(vect1, vect2, mode)
```

Mode of output vector specified by user in *mode*. Default mode is 'normalized', other option is 'notModified'.

ATOM:+models:+utilities:+geomPublic:euclideanDistanceBetweenTwoSets

euclideanDistanceBetweenTwoSets: compute distance between two sets of points

This function computes Euclidean distance between two sets of points.

Inputs

set1: `first set of points, double [nPoints x nDims]`

set2: `second set of points, double [mPoints x nDims]`

Outputs

distMatrix: `distance between points, double [nPoints x mPoints]`

Syntax

```
distMatrix = models.utilities.geomPublic. ...
euclideanDistanceBetweenTwoSets(set1, set2)
```

Function euclideanDistanceBetweenTwoSets computes pairwise distances between two sets of points defined in *set1* and *set2*.

AToM:+models:+utilities:+geomPublic:euclideanDistanceBetweenTwoSetsSqrt

euclideanDistanceBetweenTwoSetsSqrt: compute distance between sets of points

This function computes Euclidean distance between two sets of points. It is more robust variant of function: models.utilities.geomPublic.euclideanDistanceBetweenTwoSets

Inputs

set1: first set of points, double [nPoints x nDims]

set2: second set of points, double [mPoints x nDims]

tol: optional, tolerance before sqrt, double [1 x 1]

Outputs

distMatrix: distance between points, double [nPoints x mPoints]

Syntax

```
distMatrix = models.utilities.geomPublic. ...
euclideanDistanceBetweenTwoSets(set1, set2)
```

Function euclideanDistanceBetweenTwoSets computes pairwise distances between two sets of points defined in *set1* and *set2*.

AToM:+models:+utilities:+geomPublic:findNumberOfOccurances

findNumberOfOccurances: find number of occurances of element in other vector

This function searches for number of occurances of each element of specified *alphabet* in defined vector with repetitions *vect*.

Inputs

alphabet: values to be found in vect, double [1 x N]

vect: vector of interest, double [1 x nVectSize]

Outputs

nOccur: number of occurances of alphabet elements in vect, double [1 x N]

Syntax

```
nOccur = models.utilities.geomPublic.findNumberOfOccurances(alphabet, vect)
```

Function findNumberOfOccurances computes how many times elements of *alphabet* occur in vector of interest *vect*.

AToM:+models:+utilities:+geomPublic:gen-ifs-fractal

:

AToM:+models:+utilities:+geomPublic:geomUnique

geomUnique: finds unique rows according to relative tolerance

This function determines unique rows in set of points.

Inputs

`points`: set of points, double [nPoints x 1/2/3]

`tol`: geom relative precision, double [1 x 1]

Outputs

`uniquePoints`: set of unique points, double [nUniques x 1/2/3]

Syntax

```
[uniquePoints, indA, indC] = models.utilities.geomPublic.geomUnique( ...
points, tol)
```

Function geomUnique determines unique points *uniquePoints* from set of points *points*. Also index vectors *indA*: indices of unique rows in initial set) and *indC*: indices of initial points in unique set.

AToM:+models:+utilities:+geomPublic:getAngleBetweenVectors

getAngleBetweenVectors: compute angle between two vectors

This function computes angle between vector 1 defined in 3D Euclidean space by *vect1* and 3D vector 2 *vect2* .

Inputs

`vect1`: 3D vector, double [1 x 3]

`vect2`: 3D vector, double [1 x 3]

Outputs

`angle`: angle between vector 1 and vector 2, double [1 x 1] in [rad]

Syntax

```
angle = models.utilities.geomPublic.getAngleBetweenVectors(vect1, vect2)
```

Function getAngleBetweenVectors computes angle between two vectors in 3D defined by *vect1* and *vect2*.

AToM:+models:+utilities:+geomPublic:getEllipseArcLength

getEllipseArcLength: compute length of ellipsearc

This static method computes length of part of ellipseac specified by majorRadius, minorRadius, startAngle and arc angle.

Inputs

`majorRadius`: major axis radius, double [N x 1]

`minorRadius`: minor axis radius, double [N x 1]

`startAngle`: start angle of arc, double [N x 1]

`angle`: arc length, double [N x 1]

`tolerance`: numrical tolerance, double [1 x 1]

Outputs

`length`: length of ellipsearcs, double [N x 1]

Syntax

```
length = models.utilities.geomPublic.getEllipseArcLength(majorRadius, ...
minorRadius, startAngle, angle)
```

Function getEllipseArcLength computes length of ellipsearcs specified by its properties: *majorRadius*, *minorRadius*, *startAngle*, *angle*.

AToM:+models:+utilities:+geomPublic:getLineIntersectingTwoPlanes

getLineIntersectingTwoPlanes: find intersection line between two planes

This function computes line that is intersecting both planes defined in 3D.

Inputs

norm1: normal of planes from set 1, double [nP1 x 3]

point1: point on planes from set 1, double [nP1 x 3]

norm2: normal of planes from set 2, double [nP2 x 3]

point2: point on planes from set 2, double [nP2 x 3]

Outputs

isParallel: are planes parallel, logical [nP1 x nP2]

interVect: vector of line, double [nP1 x 3*nP2]
- **.x**: x-coordinates, double [nV1 x nV2]
- **.y**: y-coordinates, double [nV1 x nV2]
- **.z**: z-coordinates, double [nV1 x nV2]

interPoint: point on line, double [nP1 x 3*nP2]
- **.x**: x-coordinates, double [nV1 x nV2], in [m]
- **.y**: y-coordinates, double [nV1 x nV2], in [m]
- **.z**: z-coordinates, double [nV1 x nV2], in [m]

Syntax

```
[isParallel, interVect, interPoint] = models.utilities.geomPublic. ...
getLineIntersectingTwoPlanes(norm1, point1, norm2, point2)
```

Function getLineIntersectingTwoPlanes finds intersection line of planes defined by their normal (*norm1* and *norm2*) and point (*point1* and *point2*). The curve is found in form of point (*interPoint*) and vector (*interVecto*).

AToM:+models:+utilities:+geomPublic:getPointsOnEllipseArc

getPointsOnEllipseArc: compute points on ellipse arc

This function computes position of points that defines an EllipseArc.

Inputs

nPoints: number of points on EllipseArc, double [1 x 1]

center: EllipseArc center position, double [1 x 3]

majorVertex: EllipseArc major vertex point, double [1 x 3]

minorVector: EllipseArc minor vertex point, double [1 x 3]

startAngle: EllipseArc start angle, double [1 x 1]

angle: EllipseArc angle, double [1 x 1]

Outputs

dP: drawPoints, double [nPoints x 3]

Syntax

```
dP = getPointsOnEllipseArc(nPoints, center, majorVector, minorVector,
startAngle, angle)
```

Points *dP* are computed on the EllipseArc defined by *center*, *majorVertex*, *minorVertex*, *startAngle*, *angle*.

AToM:+models:+utilities:+geomPublic:getPointsOnEquationCurve

getPointsOnEquationCurve: compute points on EquationCurve

This function computes position of points that defines EquationCurve.

Inputs

nPoints: number of points on EllipseArc, double [1 x 1]

interval: parameter interval, double [1 x 2]

eqX: handle_function [1 x 1]

eqY: handle_function [1 x 1]

eqZ: handle_function [1 x 1]

Outputs

dP: points on line, double [nPoints x 3]

Syntax

```
dP = getPointsOnEquationCurve(nPoints, interval, eqX, eqY, eqZ)
```

Points *dP* are computed on the EquationCurve defined by *interval*, *eqX*, *eqY*, *eqZ*.

AToM:+models:+utilities:+geomPublic:getPointsOnLine

getPointsOnLine: compute points on line segment

This function computes position of points that defines Line.

Inputs

`nPoints: number of points on Line, double [1 x 1]`
`startPoint: double [1 x 3]`
`endPoint: double [1 x 3]`

Outputs

`dP: points on Line, double [nPoints x 3]`

Syntax

```
dP = obj.getPointsOnLine(nPoints, startPoint, endPoint)
```

Points *dP* are computed on the Line defined by *startPoint*, *endPoint*.

AToM:+models:+utilities:+geomPublic:getPolygonArea

getPolygonArea: compute area of 2D polygon in 3D

This function computes area of a 2d polzgon (flat) in 3d space.

Inputs

`points: 3D polygon nodes, double [N x 3]`

Outputs

`area: area of polygon, double [1 x 3] in [rad]`

Syntax

```
[area, inLine, normal] = models.utilities.geomPublic.getPolygonArea(points)
```

Function getPolygonArea computes area of polygon defined in 3D space by points specified in *points*. In case all the polygon segments specified by *points* are parallel, *inLine* is set to true and area contains NaN.

AToM:+models:+utilities:+geomPublic:getTriangleArea

getTriangleArea: compute signed area of triangle

This function computes signed area of triuangle defined by three points: *point1*, *point2* and *point3*.

Inputs

`point1`: first point position, double [1 x 3] in [m]

`point2`: second point position, double [1 x 3] in [m]

`point3`: third point position, double [1 x 3] in [m]

Outputs

`area`: signed area of triangle, double [1 x 1] in [m^2]

Syntax

```
area = models.utilities.geomPublic.getTriangleArea(point1, point2, point3)
```

Function getTriangleArea computes signed area of triangle defined by three points: *point1*, *point2* and *point3*. The resulting area is: I] area > 0 - points are in CCW order, II] area < 0 - points in CW order, III] area = 0 - points are in one line.

AToM:+models:+utilities:+geomPublic:getVectorAngles

getVectorAngles: compute angles between vector and coordinate axes X, Y, Z

This function computes angles between vector defined in 3D Euclidean space and coordinate axes X [1 0 0], Y [0 1 0] and Z {0 0 1}.

Inputs

`vect`: 3D vector, double [1 x 3]

Outputs

`angles`: angles between vector and axes, double [1 x 3] in [rad]

Syntax

```
angles = models.utilities.geomPublic.getVectorAngles(vect)
```

Function getVectorAngles computes abgles between vector *vect* in 3D and axes x [1, 0, 0], y [0, 1, 0] and z [0, 0, 1].

AToM:+models:+utilities:+geomPublic:getVectorNorm

getVectorNorm: compute norm of vector in 3D

This static method computes norm of vector defined in 3D Euclidean space.

Inputs

vect: `3D vector, double [N x 3]`

dim: `dimension, double [1 x 1]`

Outputs

vNorm: `norm of the vector, double [N x 1]`

Syntax

```
vNorm = models.utilities.geomPublic.getVectorNorm(vect)
```

Function getVectorNorm computes norm of vector *vect* in 3D.

AToM:+models:+utilities:+geomPublic:intersectLines2D

intersectLines2D: find intersection points between two sets of lines

This static method finds intersection between two sets of lines in 2D plane.

Inputs

line1: `points of set1 lines, double [2 x 2*N1]`

lines: `points of set2 lines, double [2 x 2*N2]`

tol: `geometrical precision, double [1 x 1]`

Outputs

points: `intersection points between curves, cell [N1 x N2]`

status: `double [N1 x N2]`

```
       0 - no intersection
       1 - intersection in one point
       2 - overlappping
```

param: `parametric position of intersection points, cell [N1 x N2]`

Syntax

```
[points, status, param] = models.utilities.geomPublic. ...
intersectLines2D(line1, line2)
```

Intersection points *points* between a set of lines *line1* and set of lines *line2* are computed in 2D. The variable *status* indicates how intersection ends: 0 - no intersection, 1 - lines intersect in one point, 2 - lines overlap between *points(1,:)* and *points(2,:)*.

AToM:+models:+utilities:+geomPublic:intersectLines3D

intersectLines3D: computes intersection point of two lines in 3D.

Implemented according to: https://math.stackexchange.com/questions/270767/find-intersection-of-two-3d-lines This is public function.

Inputs

X1, X2: `start and end point of first line, double [1 x 3]`

Y1, Y2: `start and end point of second line, double [1 x 3]`

Outputs

I: `intersection point coordinates, double [1 x 3]`

t: `parameter of intersection point. If is in interval [0,1], intersection point lies between defining points, double [1 x 1]`

AToM:+models:+utilities:+geomPublic:isPolygonCounterClockWise

isPolygonCounterClockWise: find out if polygon is CCW or not

This function finds out if polygon specified by ordered points is in CCW (counterclockwise] order or in CW (clockwise) order.

Inputs

points: `points of polygon, double [N x 3] in [m]`

varargin:
`tolerance: geometry tolerance, double [1 x 1] in [m]`

Outputs

isCCW: `true = counterclockwise, false = clockwise, logical [1 x 1]`

inLine: `true = all points in one line, false = triangle, logical [1 x 1]`

Syntax

```
[isCCW, inLine, area] = ...
models.utilities.geomPublic.isPolygonCounterClockWise(points, tolerance)
```

Function isPolygonCounterClockWise is used to determine if polygon specified by N 3D points *points* is in CCW order ($isCCW$ = true) or in CW order ($isCCW$ = false). The tolerance of geometry is set to first value in *varargin*.

AToM:+models:+utilities:+geomPublic:isTriangleCounterClockWise

isTriangleCounterClockWise: find out if triangle is CCW or not

This function finds out if triangle specified by three points in order: *point1*, *point2* and *point3* is in CCW }counterclockwise] order or in CW (clockwise order).

Inputs

point1: first point position, double [1 x 3] in [m]

point2: second point position, double [1 x 3] in [m]

point3: third point position, double [1 x 3] in [m]

varargin:
    tolerance: geometry tolerance, double [1 x 1] in [m]

Outputs

isCCW: true = counterclockwise, false = clockwise, logical [1 x 1]

inLine: true = all points in one line, false = triangle, logical [1 x 1]

Syntax

```
[isCCW, inLine] = models.utilities.geomPublic.isTriangleCounterClockWise( ...
point1, point2, point3, tolerance)
```

Function isTriangleCounterClockWise is used to determine if triangle specified by three points *point1*, *point2* and *point3* is in CCW order ($isCCW$ = true) or in CW order ($isCCW$ = false). The tolerance of geometry is set to first value in *varargin*.

AToM:+models:+utilities:+geomPublic:makeVectorsPerpendicular

makeVectorsPerpendicular: force two vectors to be perpendicular

This function forces two vectors to be perpendicular. First vector remains the same and the second one is rotated in the plane defined by the vectors so that they are perpendicular.

Inputs

vect1: 3D vector, double [1 x 3]

vect2: 3D vector, double [1 x 3]

Outputs

vect2: 3D vector, double [1 x 3]

Syntax

```
vect2 = models.utilities.geomPublic.makeVectorsPerpendicular(vect1, vect2)
```

Function makeVectorsPerpendicular forces two vectors to be perpendicular by rotating the second vector in plane defined by the vectors.

AToM:+models:+utilities:+geomPublic:pointsEuclidDistance

pointsEuclidDistance: computes Euclidean distances between points in 3D

This function computes distance between individual points and total length of the segments connecting all points.

Inputs

points: 3D points, double [N x 3] in [m]

Outputs

totalLength: length of the whole refracted line, double [1 x 1] in [m]

sectionLengths: length of individual segments, double [(N - 1) x 1] in [m]

Syntax

```
[totalLength, sectionLengths] = ...
models.utilities.geomPublic.pointsEuclidDistance(points)
```

Function pointsEuclidDistance computes Euclidean distances between 3D points.

AToM:+models:+utilities:+geomPublic:pointsGlobal2LocalCoords

pointsGlobal2LocalCoords: transform object from global to local coordinates

This function transforms an object from global coordinate system ([1, 0, 0], [0, 1, 0], [0, 0, 1]) to local one {defined by object} *origin*, *localX*, *localY* and *localZ*.

Inputs

points: 3D points, double [N x 3]

origin: object center position, origin of coord system, double [1 x 3]

localX: orientation of object's coordinate system X, double [1 x 3]

localY: orientation of object's coordinate system Y, double [1 x 3]

localZ: orientation of object's coordinate system Y, double [1 x 3]

varargin:
tolerance: optional, geometry tolerance, double [1 x 1], in [m]

Outputs

points: new position of points, double [N x 1]

Syntax

```
points = models.utilities.geomPublic.pointsGlobal2LocalCoords(points,
origin, localX, localY, localZ, tolerance)
```

Function pointsGlobal2LocalCoords transforms points from global coordinate system to local one defined by three vectors *localX*, *localY*, *localZ* and center point *origin*.

AToM:+models:+utilities:+geomPublic:pointsLocal2GlobalCoords

pointsLocal2GlobalCoords: transforms object from local to global coordinates

This function transforms an object from local coordinate system (defined for the object) to global one specified by three vectors globalX, globalY and globalZ.

Inputs

points: `3D points, double [N x 3]`

origin: `object center position, double [1 x 3]`

globalX: `orientation of object in global coordinate system, double [1 x 3]`

globalY: `orientation of object in global coordinate system, double [1 x 3]`

globalZ: `orientation of object in global coordinate system, double [1 x 3]`

varargin:
```
tolerance: optional, geometry tolerance, double [1 x 1], in [m]
```

Outputs

points: `new position of points, double [N x 1]`

Syntax

```
points = models.utilities.geomPublic.pointsLocal2GlobalCoords(points,
origin, globalX, globalY, globalZ)
```

Function pointsLocal2GlobalCoords transforms points from their local coordinate system to global one defined by three vectors *globalX*, *globalY*, *globalZ* and *origin*.

AToM:+models:+utilities:+geomPublic:pointsRotate

pointsRotate: rotate points in 3D around vector by angle

This static method rotates points by angle around line defined by vector either starting at origin O and defined by *vect* or gooing through two points defined in *vect*.

Inputs

points: points in 3D, double [N x 3]

vect: definition of rotation axis, double [1or2 x 3]

angle: rotation angle, double [1 x 1] in [rad]

Outputs

points: transformed points in 3D, double [N x 3]

transformMatrix: double [4 x 4]

Syntax

```
[points, transformMatrix] = ...
models.utilities.geomPublic.pointsRotate(points, vect, angle)
```

Object obj is rotated by angle in radians around axis specified by *vect*. If *vect* has one row, the rotation is made around line defined by Origin and point saved in *vect*. If *vect* has two rows, the rotatiuon is made around line defined by two points in *vect*.

AToM:+models:+utilities:+geomPublic:pointsRotateX

pointsRotateX: rotate points around X-axis by angle

This function rotates points by angle in rad around global X-axis [1, 0, 0].

Inputs

points: points in 3D, double [N x 3] in [m]

angle: rotation angle, double [1 x 1] in [rad]

Outputs

points: transformed points in 3D, double [N x 3] in [m]

transformMatrix: double [4 x 4]

Syntax

```
[points, transformMatrix] = ...
models.utilities.geomPublic.pointsRotateX(points, angle)
```

Points are rotated by angle in radians around X-axis [1, 0, 0].

AToM:+models:+utilities:+geomPublic:pointsRotateY

pointsRotateY: rotate points around Y-axis by angle

This function rotates points by angle in rad around Y-axis [0, 1, 0].

Inputs

`points`: points in 3D, double [N x 3] in [m]

`angle`: rotation angle, double [1 x 1] in [rad]

Outputs

`points`: transformed points in 3D, double [N x 3] in [m]

`transformMatrix`: double [4 x 4]

Syntax

```
[points, transformMatrix] = models.utilities.geomPublic. ...
pointsRotateY(points, angle)
```

Points are rotated by angle in radians around Y-axis [0, 1, 0].

AToM:+models:+utilities:+geomPublic:pointsRotateZ

pointsRotateZ: rotate points around Z-axis by angle

This function rotates points by angle in rad around Z-axis [0, 0, 1].

Inputs

`points`: points in 3D, double [N x 3] in [m]

`angle`: rotation angle, double [1 x 1] in [rad]

Outputs

`points`: transformed points in 3D, double [N x 3] in [m]

`transformMatrix`: double [4 x 4]

Syntax

```
[points, transformMatrix] = ...
models.utilities.geomPublic.pointsRotateZ(points, angle)
```

Points are rotated by angle in radians around Z-axis [0, 0, 1].

AToM:+models:+utilities:+geomPublic:pointsScale

pointsScale: scale points according to vector

This function scales specified points according to vector. The individual dimensions of points are multiplied by values from specified vector.

Inputs

points: points in 3D, double [N x 3] in [m]

vect: scaling vector, double [1 x 3]

center: optional, cneter of transformation (default [0 0 0]), double [1 x 3]

Outputs

points: transformed points in 3D, double [N x 3] in [m]

transformMatrix: double [4 x 4]

Syntax

```
[points, transformMatrix] = ...
models.utilities.geomPublic.pointsScale(points, vect, center)
```

Points are scaled so that any dimension of values in *points* is multiplied by corresponding value from vector *vect*. The object is moved to *center* before scale operation, and then back after the scale operation.

AToM:+models:+utilities:+geomPublic:pointsTranslate

pointsTranslate: translates object according to vector

This function translates specified points by vector in meters.

Inputs

points: points in 3D, double [N x 3] in [m]

vect: translation vector, double [1 x 3] in [m]

Outputs

points: transformed points in 3D, double [N x 3] in [m]

transformMatrix: double [4 x 4]

Syntax

```
[points, transformMatrix] = ...
models.utilities.geomPublic.pointsTranslate(points, vect)
```

Points are translated to new position according to vector *vect* in meters.

AToM:+models:+utilities:+geomPublic:repelem

repelem: repeats elements of vect rep-times

This function repeats elements of n-times according to rep values.

Inputs

vect: vector of values, double [1 x N]

rep: count how many times should be repeated, double [1 x N]

Outputs

newVect: vector of repeated values, double [1 x M]

Syntax

```
newVect = models.utilities.geomPublic.repelem(vect, rep)
```

Function repelem repeates all elements of vector *vect* according to values in vector *rep* having the same size.

AToM:+models:+utilities:+geomPublic:roundToRelativeTolerance

roundToRelativeTolerance: round to relative tolerance

This function round values to relative tolerance according to their max abs value.

Inputs

values: set of values, double [nVals1 x nVals2]

tol: geom relative precision, double [1 x 1]

Outputs

uniquePoints: set of unique points, double [nUniques x 1/2/3]

Syntax

```
values = models.utilities.geomPublic.roundToRelativeTolerance(values, tol)
```

Function roundToRelativeTolerance throws back values _*values rounded to a relative tolerance _tol* accrodnig to their max abs value.

Namespace

+models/+utilities/+matrixOperators/+SMatrix

AToM:+models:+utilities:+matrixOperators:+SMatrix:computeDS

computeDS: derivative of S matrix

S matrix assembled according to https://arxiv.org/pdf/1709.09976.pdf
matrix properties:

```
odd rows  - TE modes
even rows - TM modes
```

Inputs

mesh: mesh structure, struct [1 x 1]

basisFcns: basis functions structure, struct [1 x 1]

frequency: frequency, double [1 x 1]

```
INUPUTS
 (optional)
 maxDegreeL:      maximal degree of used spherical functions,
                   double [1 x 1], default value 15
 quadratureOrder: order of Gaussian quadrature, double [1 x 1],
                   integers <1 , 12>, default value 1
```

Outputs

dS: S matrix derivative, double [N x M]

Syntax

```
dS = computeDS(mesh, basisFcns, frequency)
dS = computeDS(mesh, basisFcns, frequency)
dS = computeDS(mesh, basisFcns, frequency, ...
                         maxDegreeL, quadratureOrder)
```

AToM:+models:+utilities:+matrixOperators:+SMatrix:computeS

computeS: Calculates S matrix

S matrix assembled according to https://arxiv.org/pdf/1709.09976.pdf
matrix properties:

```
odd rows  - TE modes
even rows - TM modes
```

Inputs

```
mesh: mesh structure, struct [1 x 1]

basisFcns: basis functions structure, struct [1 x 1]

frequency: frequency, double [1 x 1]
```

Inputs

```
(optional)

maxDegreeL:      maximal degree of used spherical functions,
                  double [1 x 1], default value 15

quadratureOrder: order of Gaussian quadrature, double [1 x 1],
                  integers <1 , 12>, default value 1

wavesType: type of waves, double [1 x 1]
           1 - regular waves, z = spherical Bessel function
           2 - irregular waves, z = spherical Neumann function
           3 - ingoing waves, z = spherical Hankel function 1
           4 - outgoing waves, z = spherical Hankel function 2
```

Outputs

```
S:           S matrix,   double [N x M]

indexMatrix: matrix of ordering in S matrix, double [5 x N]
```

Syntax

```
S = computeS(mesh, basisFcns, frequency)
[S, indexMatrix] = computeS(mesh, basisFcns, frequency)
[S, indexMatrix] = computeS(mesh, basisFcns, frequency, ...
                       maxDegreeL, quadratureOrder)
[S, indexMatrix] = computeS(mesh, basisFcns, frequency, ...
                       maxDegreeL, quadratureOrder, wavesType)
```

AToM:+models:+utilities:+matrixOperators:+SMatrix:functionR

functionR: radial part of spherical waves

Inputs

```
degreeL: vector of degrees L, double [N x 1]
kR:      vector of radial coordinates, double [M x 1]
p:       type of waves, double [1 x 1]
           1 - regular waves, z = spherical Bessel function
           2 - irregular waves, z = spherical Neumann function
           3 - ingoing waves, z = spherical Hankel function 1
           4 - outgoing waves, z = spherical Hankel function 2
```

Outputs

```
R1:   R1 radial function, complex double [N x M]
R2:   R2 radial function, complex double [N x M]
R3:   R3 radial function, complex double [N x M]
zD:   derivatve of propper spherical bessel function,
      complex double [N x M]
```

Syntax

AToM:+models:+utilities:+matrixOperators:+SMatrix:functionU

functionU: spherical vector waves u

Inputs

degreeL: vector of degrees L, double [N x 1]

orderM: vector of orderes M, double [N x 1]

theta: vector of theta coordinates, double [M x 1]

phi: vector of phi coordinates, double [M x 1]

kR: vector of radial coordinates, double [M x 1]

p: type of waves, double [1 x 1]
1 – regular waves, z = spherical Bessel function
2 – irregular waves, z = spherical Neumann function
3 – ingoing waves, z = spherical Hankel function 1
4 – outgoing waves, z = spherical Hankel function 2

Outputs

u12: spherical vector wave u1 with sigma = 2, complex double [N x M x 3]

u11: spherical vector wave u1 with sigma = 1, complex double [N x M x 3]

u22: spherical vector wave u2 with sigma = 2, complex double [N x M x 3]

u21: spherical vector wave u2 with sigma = 1, complex double [N x M x 3]

u32: spherical vector wave u3 with sigma = 2, complex double [N x M x 3]

u31: spherical vector wave u3 with sigma = 1, complex double [N x M x 3]

Syntax

AToM:+models:+utilities:+matrixOperators:+SMatrix:functionY

functionY: vector spherical harmonics Y

Inputs

degreeL: vector of degrees L, double [N x 1]
orderM: vector of orderes M, double [N x 1]
theta: vector of theta coordinates, double [M x 1]
phi: vector of phi coordinates, double [M x 1]

Outputs

Y1: Y1 vector spherical hamonic, complex double [N x M x 3]
Y2: Y2 vector spherical hamonic, complex double [N x M x 3]
Y3 Y3 vector spherical hamonic, complex double [N x M x 3]

Syntax

AToM:+models:+utilities:+matrixOperators:+SMatrix:lmax

lmax: gives estimate of highest L order for spherical expansion

Inputs

ka: normalized electrical size

Outputs

Lmax: highest degree of Legendre polynomial to be used

Syntax

```
Lmax = lmax(ka)
```

See [1] Tayli, Capek, Akrou, Losenicky, Jelinek, Gustafsson: Accurate and Efficient Evaluation of Characteristic Modes, IEEE TAP, 2018. https://arxiv.org/pdf/1709.09976.pdf

AToM:+models:+utilities:+matrixOperators:+SMatrix:totalSphericalModes

totalSphericalModes: determines how many spherical waves are used

Inputs

Lmax: normalized electrical size

Outputs

sphWaves: highest degree of Legendre polynomial to be used

Syntax

```
sphWaves = totalSphericalModes(Lmax)
```

See [1] Tayli, Capek, Akrou, Losenicky, Jelinek, Gustafsson: Accurate and Efficient Evaluation of Characteristic Modes, IEEE TAP, 2018. https://arxiv.org/pdf/1709.09976.pdf

Namespace

+models/+utilities/+matrixOperators/+TMatrix

AToM:+models:+utilities:+matrixOperators:+TMatrix:computeT

computeT: Calculates T matrix

Inputs

```
mesh:             mesh structure, struct [1 x 1]
basisFcns:        basis functions structure, struct [1 x 1]
frequency:        frequency, double [1 x 1]
maxDegreeL:       maximal degree of used spherical functions,
                    double [1 x 1], default value 15
quadratureOrder: order of Gaussian quadrature, double [1 x 1],
                  integers <1 , 12>, default value 1
```

Outputs

```
T:       T matrix, double [N x N]
```

Syntax

AToM:+models:+utilities:+matrixOperators:+TMatrix:computeTSphere

computeTSphere: Calculates analytic T matrix for scattering of sphere

Inputs

```
radius:       radius of the sphere in meters, double [1 x 1]
frequency:    frequency, double [1 x 1]
maxDegreeL:   maximal degree of used spherical functions,
                  double [1 x 1], default value 15
```

Outputs

```
T:       T matrix, double [N x N]
```

Syntax

AToM:+models:+utilities:+matrixOperators:+TMatrix:computeT-Z

computeT_Z: Calculates T matrix with precalculated impedance matrix

Inputs

Z: impedance matrix, double [M x M]

mesh: mesh structure, struct [1 x 1]

basisFcns: basis functions structure, struct [1 x 1]

frequency: frequency, double [1 x 1]

maxDegreeL: maximal degree of used spherical functions, double [1 x 1], default value 15

quadratureOrder: order of Gaussian quadrature, double [1 x 1], integers <1 , 12>, default value 1

Outputs

T: T matrix, double [N x N]

Syntax

Namespace

+models/+utilities/+matrixOperators/+electricMoment

AToM:+models:+utilities:+matrixOperators:+electricMoment:computeP

computeP: compute electric moment operator

Compute electric moment operator If frequency is not set, results corresponds to p0 in p = 1i / (2*pi*frequency) * p0; i.e. to results with frequency = 1i / (2*pi).

Inputs

mesh: `mesh structure, struct [1 x 1]`

basisFcns: `basis functions, struc [1 x 1]`

frequency: `frequency list, double [nFreq x 1]`

Outputs

P: `electric moment, double [nEdges x nEdges]`

p:

Syntax

```
[P, p] = computeP(mesh, basisFcns)
[P, p] = computeP(mesh, basisFcns, frequency)
```

Namespace

+models/+utilities/+matrixOperators/+farfield

AToM:+models:+utilities:+matrixOperators:+farfield:computeU

computeU: compute radiation intensity matrix

Compute radiation intensity matrix

Inputs

mesh: mesh structure, struct [1 x 1]

basisFcns: basis functions, struc [1 x 1]

frequency: frequency, double [1 x 1]

theta: theta angle, double [1 x 1]

phi: phi angle, double, [1 x 1]

component:

Outputs

U: radiation intensity matrix, double [nEdges x nEdges]

F_phi:

Syntax

```
[U, F_phi, F_theta] = computeU(mesh, basisFcns, frequency, theta, phi)
[U, F_phi, F_theta] = computeU(mesh, basisFcns, frequency, theta, ...
  phi, component)
```

see Jelinek, Capek: Optimal Currents on Arbitrarily Shaped Surfaces IEEE-TAP, 2017, Eqs. (49)-(56)
Far-field vector F defined as F(e,k) = -1j*k*Z0/(4*pi) * int(e .* J(r) * exp(1j*k .* r0) dS U-matrix defined as U(k) = (F'*F) / (2*Z0)

Namespace

+models/+utilities/+matrixOperators/+magneticMoment

AToM:+models:+utilities:+matrixOperators:+magneticMoment:computeM

computeM: compute magnetic moment operator

Compute magnetic moment operator

Inputs

mesh: mesh structure, struct [1 x 1]

basisFcns: basis functions, struc [1 x 1]

Outputs

M: magnetic moment, double [nEdges x nEdges]

m:

Syntax

```
[M, m] = computeM(mesh, basisFcns)
```

Namespace

+models/+utilities/+matrixOperators/+ohmicLosses

AToM:+models:+utilities:+matrixOperators:+ohmicLosses:computeL

computeL: Compute L matrix for calculation of ohmic losses

Compute L matrix for calculation of ohmic losses

Inputs

mesh: mesh structure, struct [1 x 1]

basisFcns: basis functions, struc [1 x 1]

rho: [nTria x 1] for mode='triangles', [nEdges x 1] for mode='edges'

mode:

Outputs

L: lossy matrix [nEdges x nEdges]

AToM:+models:+utilities:+matrixOperators:+ohmicLosses:lossyMatrix

lossymatrix: calculate L matrix for calculation of ohmic losses

calculate L matrix for calculation of ohmic losses

Inputs

mesh: mesh structure, struct [1 x 1]

basisFcns: basis functions, struc [1 x 1]
rhoTria [nTria x 1]

Outputs

```
L .. lossy matrix [nEdges x nEdges]
T
F
FL .. transform structure for rhoEdge -> rhoTria [6 x 5 x nTria]
 1st dimension: 6 edges for each triangle
 2nd dimension:
  FL(:, 1, iTria) .. number of edge 1
  FL(:, 2, iTria) .. number of edge 2
  FL(:, 3, iTria) .. contribution to lossy matrix
  FL(:, 4, iTria) .. sign of edge 1
  FL(:, 5, iTria) .. sign of edge 2
```

ATOM:+models:+utilities:+matrixOperators:+ohmicLosses:rhoEdge2rhoTria

rhoEdge2rhoTria: recalculate resistivitiy of triangles from edges

Inputs

mesh: mesh structure, struct [1 x 1]

FL: transform structure for rhoEdge -> rhoTria [6 x 5 x nTria]

rhoEdge: [nEdges x 1]

I: [nEdges x 1]

Outputs

rhoTria: [nTria x 1]

ATOM:+models:+utilities:+matrixOperators:+ohmicLosses:thinSheetCoef

thinSheetCoef: calculate lossy coeficient derived for thin-sheet approximation

Inputs

```
frequency
sigma
t
```

Outputs

```
F [1x1]
```

Namespace

+models/+utilities/+meshPublic

AToM:+models:+utilities:+meshPublic:commonEdgeOfTwoTriangles

commonEdgeOfTwoTriangles: Returns ID of common edge of two adjacent triangles

The function returns ID of edge between two adjacent triangles.

AToM:+models:+utilities:+meshPublic:deleteEdges

deleteEdges: deletes edges from given mesh

```
This function takes nodes a their connections, which form mesh in 3D.
It outputs new mesh without given edges.
```

Inputs

nodes: point coordinates, double [N x 3]

connectivityList: connectivity of nodes, double [N x 3]

edgesToDelete: edges to delete from the mesh, double [N x 1]

Outputs

newNodes: new set of nodes, double [N x 3]

newConnectivityList: new set of connections, double [N x 3]

Syntax

```
[newNodes, newConnectivityList] =
models.utilities.meshPublic.deleteEdges(nodes, connectivityList,edgesToDelete);
```

AToM:+models:+utilities:+meshPublic:deleteNodes

deleteNodes: deletes nodes from given mesh

This function takes points a their connections, which form mesh in 3D.
It outputs new mesh without nodes specified in nodes.

Inputs

nodes: point coordinates, double [N x 3]

connectivityList: ratio for scaling points, double [N x 3]

nodes: points to delete from the mesh, double [N x 1]

Outputs

newNodes: new set of points, double [N x 3]

newConnectivityList: new set of connections, double [N x 3]

Syntax

```
[newNodes, newConnectivityList] =
models.utilities.meshPublic.deleteNodes(nodes, connectivityList, nodesToDelete);
```

AToM:+models:+utilities:+meshPublic:deleteTriangles

deleteTriangles: deletes triangles from a given mesh

This function takes nodes and their connections, which form mesh in 3D.
It outputs new mesh without given triangles.

Inputs

nodes: node coordinates, double [N x 3]

connectivityList: triangle connectivity, double [N x 3]

trianglesToDelete: triangles to delete from the mesh, double [N x 1]

Outputs

newNodes: new set of nodes, double [N x 3]

newConnectivityList: new set of connections, double [N x 3]

newNodeNumbering: new node numbering, double [N x 1]

Syntax

```
[newNodes, newConnectivityList] =
models.utilities.meshPublic.deleteTriangles(nodes, connectivityList,
trianglesToDelete);
```

AToM:+models:+utilities:+meshPublic:edgeSymPlanes

edgeSymPlanes: get information about edges touching symmetry plane

```
The function returns information of edges belonging to symmetry planes.
The _symPlaneInfo_ contains 1 on positions where the edge touches given
symmetry plane ([0 1 0] means that edge is lying in plane XZ).
```

Inputs

nodes: node coordinates

edges: node IDs of edges

Outputs

symPlaneInfo: symmetry plane information

Syntax

```
symPlaneInfo = models.utilities.meshPublic.edgeSymPlanes(nodes, edges);
```

AToM:+models:+utilities:+meshPublic:exportGeo

exportGeo: exports mesh to GEO file

Inputs

nodes: coordinates of points, double [N x 3]

triangles: pointers on nodes which represents triangles of mesh, double [N x 3]

filePath: path to output directory, char [1 x N]

fileName: name of the GEO file, char [1 x N]

Syntax

```
models.utilities.meshPublic.exportGeo(nodes, connectivityList, path, name);
```

AToM:+models:+utilities:+meshPublic:exportNastran

exportNastran: exports mesh to NASTRAN file

```
Creates file is NASTRAN - high-precision data format.
Data format: 8/16/16/16/16
             8/16
```

Inputs

nodes: coordinates of points, double [N x 3]

edges: pointers on nodes which represents edges of mesh, double [N x 2]

triangles: pointers on nodes which represents triangles of mesh, double [N x 3]

filePath: path to output directory, char [1 x N]

fileName: name of the NASTRAN file, char [1 x N]

Syntax

```
models.utilities.meshPublic.exportNastran(nodes, edges, triangles, path, name);
```

AToM:+models:+utilities:+meshPublic:fullSymmetryInfo

:

AToM:+models:+utilities:+meshPublic:getAreaTriangle

getAreaTriangles: calculate area of triangles.

Inputs

p: points coordinates, double [N x 3]

t: points number for each triangle, double [N x 3]

Outputs

area: areas of triangles, double [N x 1]

Syntax

```
area = models.utilities.meshPublic.getAreaTriangles(p,t);
```

AToM:+models:+utilities:+meshPublic:getBoundary2D

getBoundary2D: returns outer edges of planar triangulation

```
This function returns set of boundary edges which are specified by
triangulation connectivityList and nodes.
```

Inputs

nodes: point coordinates, double [N x 3]

connectivityList: triangle vertices, double [N x 3]

Outputs

edges: set of boundary edges in the triangulation, double [N x 2]

boundaryNodes: set of boundary nodes in the triangulation, double [N x 3]

Syntax

```
[edges, boundaryNodes] = models.utilities.meshPublic.getBoundary2D(nodes,
connectivityList);
```

AToM:+models:+utilities:+meshPublic:getBoundary3D

getBoundary3D: returns outer edges of of connected planar triangulations in 3D

```
This function returns set of boundary edges which are specified by
triangulation given by connectivityList and nodes.
```

Inputs

nodes: point coordinates, double [N x 3]

connectivityList: triangle vertices, double [N x 3]

Outputs

E: set of boundary edges in the triangulation, double [N x 2]

P: set of boundary nodes in the triangulation, double [N x 3]

Syntax

```
[E, P] = models.utilities.meshPublic.getBoundary3D(nodes, connectivityList);
```

AToM:+models:+utilities:+meshPublic:getCenterSegment

getCenterSegment: center of segment

Inputs

nodes: nodes coordinates, double [N x 3]

edges: nodes number for each segment, double [N x 2]

Outputs

center: coorddinates of center, double [N x 3]

Syntax

```
center = models.utilities.meshPublic.getCenterSegment(nodes, edges);
```

AToM:+models:+utilities:+meshPublic:getCenterTriangle

getCenterTriangle: calculate area of triangles

Inputs

nodes: points coordinates, double [N x 3]

connectivityList: points number for each triangle, double [N x 3]

Outputs

center: center of triangle, double [N x 3]

Syntax

```
center = models.utilities.meshPublic.getCenterTriangle(nodes,
connectivityList);
```

AToM:+models:+utilities:+meshPublic:getCircuitTriangle

getCircuitTriangles: calculate circuits and edges length of triangles

Inputs

nodes: points coordinates, double [N x 3]

connectivityList: nodes number for each triangle, double [N x 3]

Outputs

circuit: circuit, double [N x 1]

Syntax

```
circuit =
models.utilities.meshPublic.getCircuitTriangle(nodes,connectivityList);
```

AToM:+models:+utilities:+meshPublic:getCircumsphere

Radius: and center of the smallest circumscribing sphere

This function evaluates radius and center of the smallest circumscribing sphere
INPUTS nodes: points, double [N x 3]
OUTPUTS r: radius, double [N x 1] center: radius vector of the center, double [1 x 3]
SYNTAX
[r, center] = getCircumsphere (nodes)

AToM:+models:+utilities:+meshPublic:getEdgeLengthTriangle

getEdgeLengthTriangle: calculate edges length of triangle

Inputs

p: points coordinates, double [N x 3]

t: points number for each triangle, double [N x 3]

Outputs

edge: edge length, double [N x 3]

Syntax

```
edge = models.utilities.meshPublic.getEdgeLengthTriangle(p,t);
```

AToM:+models:+utilities:+meshPublic:getEdges

getEdges: returns edges in triangulation

This function returns set of edges which are specified by triangulation connectivityList.

Inputs

nodes: node coordinates, double [N x 3]

connectivityList: triangle vertices, double [N x 3]

Outputs

edges: set of edges in triangulation connectivityList, double [N x 2]

Syntax

```
[edges] = models.utilities.meshPublic.getEdges(nodes, connectivityList);
```

AToM:+models:+utilities:+meshPublic:getInnerNodes

getInnerEdges: returns edges in triangulation

This function returns set of inner edges which are specified by connectivityList on nodes.

Inputs

nodes: point coordinates, double [N x 3]

connectivityList: triangle vertices, double [N x 3]

Outputs

innerNodes: set of inner nodes in triangulation, double [N x 2]

Syntax

```
[innerNodes] = models.utilities.meshPublic.getInnerNodes(nodes,
connectivityList);
```

AToM:+models:+utilities:+meshPublic:getLengthSegment

getLengthSegment: calculate length of segment

Inputs

nodes: points coordinates, double [N x 3]

edges: points number for each segment, double [N x 2]

Outputs

length: length of segment, double [N x 1]

Syntax

```
lengthSegment = models.utilities.meshPublic.getLengthSegment(nodes,edges);
```

AToM:+models:+utilities:+meshPublic:getLocalCoordinateSystem

getLocalCoordinateSystem: get objects local coordinate system

This method determines local coordinate system of a polygon.

Inputs

points: 3D points, double [N x 3]

Outputs

origin: coordinate system origin if interest, double [1 x 3]

localX: X axis direction, double [1 x 3]

localX: Y axis direction, double [1 x 3]

localX: Z axis direction, double [1 x 3]

Syntax

```
[origin, localX, localY, localZ] = models.utilities.meshPublic.
   getLocalCoordinateSystem(points);
```

AToM:+models:+utilities:+meshPublic:getMeshData2D

getMeshData2D: computes information necessary for MoM computations

This function loads data from mesh and outputs struct with data necessary for MoM.

Inputs

nodes: point coordinates, double [N x 3]

connectivityList: triangle vertices, double [N x 3]

Outputs

meshData: structure with following items
```
-nodes, triangulation nodes, double [N x 3]
-connectivityList, triangulation connectivity list, double [N x 3]
-edges, triangulation edges, double [N x 2]
-edgeCentroids, center point of each edge, double [N x 3]
-edgeLengths, length of each edge, double [N x 1]
-triangleAreas, area of each triangle, double [N x 1]
-triangleCentroids, center points of each triangle, double [N x 3]
-triangleEdges, indices to edges, double [N x 3]
-nNodes, number of nodes, double [1 x 1]
-nEdges, number of edges, double [1 x 1]
-nTriangles, number of triangles, double [1 x 1]
-normDistanceA, radius of circumsphere, double [1 x 1]
```

Syntax

```
[meshData] = models.utilities.meshPublic.getMeshData2D(nodes,
connectivityList);
```

AToM:+models:+utilities:+meshPublic:getTriangleAreas

getTriangleAreas: return triangle areas

Inputs

nodes: node coordinates, double [nNodes x 3]

triangleNodes: triangle node indices, double [nTriangles x 3]

edgeLengths: triangle edge lengths, double [nTriangles x 1]

triangleEdges: triangle edge indices, double [nTriangles x 3]

Outputs

triangleAreas: triangle areas, double [N x 1]

Syntax

```
triangleAreas = getTriangleAreas(nodes, triangleNodes)
```

AToM:+models:+utilities:+meshPublic:getTriangleCentroids

getTriangleCentroids: returns centroids of all trinangles

AToM:+models:+utilities:+meshPublic:getTriangleCircumferences

triangleCircumferences: returns circumferences of all trinangles

AToM:+models:+utilities:+meshPublic:getTriangleEdgeIndices

getTriangleEdgeIndices: creates list of triangle edges according to triangle nodes

```
This function constructs local edges inside given triangles, makes unique list
of them (_edges_) and expresses local edges as pointers (_connectivityList_).
```

Inputs

connectivityList: list of triangle nodes, double [N x 3]

edges: sorted list of edges, double [N x 2]

Outputs

triangleEdges: pointers to global list of edges, double [nTriangles x 3]

Syntax

```
[triangleEdges] =
models.utilities.meshPublic.getTriangleEdgeIndices(connectivityList, edges);
```

AToM:+models:+utilities:+meshPublic:getTriangleQuality

getTriangleQuality: calculate area of triangles

Inputs

nodes: points coordinates, double [N x 3]

connectivityList: node numbers for each triangle, double [N x 3]

Outputs

quality: quality of triangles, double [N x 1]

Syntax

```
quality = models.utilities.meshPublic.getTriangleQuality(nodes,
connectivityList);
```

AToM:+models:+utilities:+meshPublic:importGeo

importGeo: imports mesh from GEO files

Inputs

filePath: path to imported file, char [1 x N]

Outputs

nodes: coordinates of points, double [N x 3]

connectivityList: subscripts into nodes, double [N x 3]

fileIsReadable: informs whether file can be read, logical [1 x 1]

Syntax

```
[nodes, connectivityList, fileIsReadable] =
models.utilities.meshPublic.importGeo(filePath);
```

AToM:+models:+utilities:+meshPublic:importMphtxt

importMphtxt: Imports mesh from mphtxt file

Inputs

fileName: name of imported file, char [1 x N]

Outputs

nodes: coordinates of points, double [N x 3]

connectivityList: pointers on nodes which represents triangles of mesh, double [N x 3]

fileIsReadable: informs whether file can be read, logical [1 x 1]

Syntax

```
[nodes, connectivityList, fileIsReadable] =
models.utilities.meshPublic.importMphtxt(fileName);
```

AToM:+models:+utilities:+meshPublic:importNastran

importNastran: Imports mesh from NASTRAN file

Inputs

fileName: name of imported file

Outputs

nodes: coordinates of points, double [N x 3]

edges: pointers on nodes which represents edges of mesh, double [N x 2]

connectivityList: pointers on nodes which represents connectivityList of mesh, double [N x 3]

tetrahedrons: not used, double [N x 4]

fileIsReadable: informs whether file can be read, logical [1 x 1]

Syntax

```
[nodes, edges, connectivityList, tetrahedrons, fileIsReadable] =
models.utilities.meshPublic.importNastran(filePath);
```

AToM:+models:+utilities:+meshPublic:meshToPolygon

meshToPolygon: creates polygon from mesh

Creates counter clockwise representation of mesh boundary.

Inputs

nodes: set of nodes, double [N x 2]

connectivityList: set of node connections, double [N x 3]

Outputs

polygons: cell of points for each polygon, cell [1 x N]

err: error when points aren't in one plane, logical [1 x 1]

Syntax

```
[polygons] = models.utilities.meshPublic.meshToPolygon(nodes,
connectivityList);
```

AToM:+models:+utilities:+meshPublic:mirrorMesh

:

Mirrors mesh according to a mirror plane given by its normal.

Inputs

obj: Mesh object, [1 x 1]

nodes: mesh nodes, double [N x 3]

normal: mirror plane normal, double [1 x 3]

origin: mirror plane origin, double [1 x 3]

Outputs

Syntax

```
[newNodes] = models.utilities.meshPublic.mirrorMesh(nodes, normal, origin);
```

AToM:+models:+utilities:+meshPublic:nodeReferences

nodeReferences: counts references of nodes in connectivityList

Inputs

nodes: double, [N x 3]

connectivityList: double [N x X]

Outputs

countedReferences: number represents how many times is each node referenced in connectivityList, double [N x 1]

isReferenced: is node referenced in connectivityList, double [N x 1]

referencedShift: shift of values in connectivityList if you

take only nodes(isReferenced,:), double [N x 1]

Syntax

```
[countedReferences, isReferenced, referencedShift] =
models.utilities.meshPublic.nodeReferences(nodes, connectivityList);
```

AToM:+models:+utilities:+meshPublic:pixelGridToMesh

pixelGridToMesh: generates mesh from matrix full of integer numbers

```
Each entry is considered as a pixel of size d (by default d = 1)
```

Inputs

```
M: matrix of integer numbers denoting type of mesh in that pixel
```

Inputs

```
(optional)
  d: size of the pixel (by default d = 1)
```

Outputs

```
nodes:
connectivityList:
Mesh: mesh grid in AToM format
```

Syntax

```
[nodes, connectivityList, Mesh = ...
       models.utilities.meshPublic.meshFromPixels(ones(10, 5), 1);


Type of elementary mesh cells:
 M(i,j) = +1 pixel has two triangles with "7-2 hours" diagonal
 M(i,j) = -1 pixel has two triangles with "5-11 hours" diagonal
 M(i,j) = +2 pixel has four triangles devided by two diagonals X


The final mesh is centered around the origin of the coordinate system.
```

AToM:+models:+utilities:+meshPublic:plotMeshBoundary

plotMeshBoundary:: plots boudary edges and nodes

```
Plots boudary edges and nodes of triangulation given by
nodes and connectivityList.
```

Inputs

nodes: point coordinates, double [N x 3]

connectivityList: triangle vertices, double [N x 3]

Syntax

```
models.utilities.meshPublic.plotMeshBoundary(nodes, connectivityList);
```

AToM:+models:+utilities:+meshPublic:plotMeshCircumsphere

plotMeshCircumsphere: plots mesh and its circumpshere

```
Plots circumsphere of triangulation given by nodes and connectivityList.
```

Inputs

nodes: point coordinates, double [N x 3]

connectivityList: triangle vertices, double [N x 3]

Syntax

```
models.utilities.meshPublic.plotMeshCircumsphere(nodes, connectivityList);
```

AToM:+models:+utilities:+meshPublic:rotateMesh

rotateMesh: rotates given set of points by given angles

```
This function takes nodes in 3D and rotates them by angles specified
in angles matrix. The coordinate system is right handed.
```

Inputs

nodes: node coordinates, double[N x 3]

angles: angles for point rotation, double [1 x 3]

Outputs

newNodes: nodes rotated by angles, double [N x 3]

Syntax

```
[newNodes] = models.utilities.meshPublic.rotateMesh(nodes, angles);
```

AToM:+models:+utilities:+meshPublic:scaleMesh

scaleMesh: rotates given set of points by given angles

This function takes nodes in 3D and scales them uniformly by given ratio.

Inputs

nodes: node coordinates, double [N x 3]

ratio: ratio for scaling points, double [1 x 1]

Outputs

newNodes: points scaled by ratio, double [N x 3]

Syntax

```
[newNodes] = models.utilities.meshPublic.scaleMesh(nodes, ratio);
```

AToM:+models:+utilities:+meshPublic:scaleNonUniformMesh

scaleNonUniformMesh: rotates given set of points by given angles

This function takes points in 3D and scales them non-uniformly by given ratio.

Inputs

points: point coordinates, double [N x 3]

ratio: ratio for scaling points, double [1 x 3]

Outputs

p: points scaled by ratio, double [N x 3]

Syntax

```
p = models.utilities.meshPublic.scaleNonUniformMesh(points, ratio);
```

AToM:+models:+utilities:+meshPublic:searchForEdgeIDs

Find: IDs of edges contributing to the points

Location of all points is compared with location of all edges. The point coordinates are expressed in terms of simplex coordinates of the triangle(s). If the simplex coordinates satisfy all the necessary conditions, it means that the point is touching given edge.

```
     o-------------------o
    1          a          2
```

AToM:+models:+utilities:+meshPublic:searchForTriangleIDs

Find: IDs of triangles contributing to the points

Location of all points is compared with location of all triangles. The point coordinates are expressed in terms of simplex coordinates of the triangle(s). If the simplex coordinates satisfy all the necessary conditions, it means that the point is touching area of given trinagle.

```
              3                  S = triangle area
              o                  S1 = area of triangle "point-2-3"
             /|                 S2 = area of triangle "point-3-1"
            / |                 S3 = area of triangle "point-1-2"
           /  |              /    |d3              d1 = distance to node 1
       c/     |     _          d2 = distance to node 2
       /  S2 | S1                d3 = distance to node 3
      /     _o_              / d1_-    -_d2            simplex1,2,3 = simplex coordinates
    /_--    S3    --_       o--------------------o
  1          a          2
```

AToM:+models:+utilities:+meshPublic:symmetrizeMesh

:

Inputs

nodes: mesh nodes, double [N x 3]

connectivityList: nodes connectivity, double [N x 3]

plane: symmetry plane, double [3 x 3]

Outputs

Syntax

```
[newNodes, newConnectivityList, mirroredTriangles] = models.utilities
   .meshPublic.symmetrizeMesh(nodes, connectivityList, plane);
```

AToM:+models:+utilities:+meshPublic:translateMesh

translateMesh: rotates given set of points by given angles

This function takes nodes in 3D and translates them by given vector specified in variable shift.

Inputs

nodes: node coordinates, double [N x 3]

shift: vector for translation, double [1 x 3]

Outputs

newNodes: nodes translated by shift, double [N x 3]

Syntax

```
[newNodes] = models.utilities.meshPublic.translateMesh(nodes, shift);
```

AToM:+models:+utilities:+meshPublic:uniformTriangulation2D

uniformTriangulation2D: creates regular uniform triangulation over given polygon in 3D

This function takes boundary nodes of polygon which should be meshed. Boundary and hole points must be ordered counter clockwise.

Inputs

boundaryNodes: point coordinates, double [N x 2]

holes: a cell with points of holes, might be an empty cell, cell [1 x 1] containint matrices double [N x 2]

elemSize: euclidean distance between points, double [1 x 1]

meshType: type of triangles used for meshing, string [1 x N] options: right, equilateral

Outputs

nodes: new set of nodes, double [N x 2]

connectivityList: new set of connections, double [N x 3]

Syntax

```
[nodes, connectivityList] = models.utilities.meshPublic.
   uniformTriangulation2D(boundaryPoints, holes, elemSize, meshType);
```

AToM:+models:+utilities:+meshPublic:uniformTriangulation3D

uniformTriangulation3D: creates regular uniform triangulation over given polygon in 3D

This function takes boundary nodes of polygon which should be meshed. Boundary and hole points must be ordered counter clockwise.

Inputs

boundaryNodes: point coordinates, double [N x 3]

holes: a cell with points of holes, might be an empty cell, cell [1 x 1] containint matrices double [N x 3]

elemSize: euclidean distance between points, double [1 x 1]

meshType: type of triangles used for meshing, char [1 x N]
options: right, equilateral

Outputs

nodes: new set of nodes, double [N x 3]

connectivityList: new set of connections, double [N x 3]

Syntax

```
[nodes, connectivityList] = models.utilities.meshPublic.
  uniformTriangulation3D(boundaryPoints, holes, elemSize, meshType);
```

AToM:+models:+utilities:+meshPublic:uniquetol

uniquetol: Unique values with tolerance

This function is fallback for versions of Matlab (<2015a) which don't have built-in uniquetol.

Inputs

toUnique: elements to unique, double [N x X]

epsilon: tolerance, double [1 x 1]

param1: ByRows parameter, char [1 x N]

val1: value of parameter 1, logical [1 x 1]

Outputs

res: unique elements from toUnique, double [N x X]

ia:

Syntax

```
[res, ia, ic] = models.utilities.meshPublic.uniquetol(toUnique, epsilon,
param1, val1);
```

AToM:+models:+utilities:+meshPublic:uniquetolSorted

uniquetol: Unique values with tolerance and outputs are in the original order

This function is fallback for versions of Matlab (<2015a) which don't have built-in uniquetol.

Inputs

toUnique: elements to unique, double [N x X]

epsilon: tolerance, double [1 x 1]

param1: ByRows parameter, char [1 x N]

val1: value of parameter 1, logical [1 x 1]

Outputs

res: unique elements from toUnique, double [N x X]

ia:

Syntax

```
[res, ia, ic] = models.utilities.meshPublic.uniquetolSorted(toUnique,
epsilon, param1, val1);
```

AToM:+models:+utilities:+meshPublic:uniteMeshes

uniteMeshes: creates one mesh from 2 sets of nodes and connectivity lists

This function connects two meshes into one connectivity list and one unique set of nodes.

Inputs

nodes1: node coordinates 1, double [N x X]

connectivityList1: nodes1 connectivity, double [N x X]

nodes2: node coordinates 2, double [N x X]

connectivityList2: nodes2 connectivity, double [N x X]

Outputs

newNodes: unique set of nodes, double [N x X]

newConnectivityList: new connectivity List, double [N x X]

Syntax

```
[newNodes, newConnectivityList] = models.utilities.meshPublic.
   uniteMeshes(nodes1, connectivityList1, nodes2, connectivityList2);
```

Namespace

+models/+utilities/+subregionMatrices

AToM:+models:+utilities:+subregionMatrices:computeCMat

computeCMat: computes subregion matrix C

Computes subregion matrix C which can be use to 'cut' any vector/matrix quantity connected to given mesh and basis functions structure. C matrix allows to reduce any vector/matrix quantity and cut out any part of the structure or create 'holes' into structure which is described by this matrix, these parts are defined by one or more polygons.
Properies of C matrix:

```
zMatSmall = C' * zMatBig * C
ISmall = C' * IBig
IBig = C * ISmall
```

where

```
zMatBig   - impedance matrix of full structure,
zMatSmall - impedance matrix of structure with 'holes',
IBig      - vector of expansion coefficient which belongs to zMatBig,
ISmall    - reduced vector of expansion coefficients which belongs
            to zMatSmall
```

Inputs

```
mesh:      mesh struct created by AToM, struct [1 x 1]
basisFcns: basis functions struct created by AToM, struct [1 x 1],
polygons:  cell of polygons defined by border points, cell[N x 1]
            N ~ number of polygons
            polygons are defined by border points using coordinates
            [x, y], double [M x 2]
            M ~ number of border points
type:      type of cut flag, double [1 x 1],
            >= 0 ~ cut out part of the structure (positive cut)
            < 0  ~ create 'holes' into the structure (negative cut)
```

Outputs

```
C:               subregion matrix, double [P X Q]
                  P ~ size of 'big' quantity
                  Q ~ size of 'small' quantity
newMesh:         new mesh structure, struct [1 x 1]
newBasisFcns:    basis functions struct, modified to be in proper
                 order to 'small' quantities, struct [1 x 1]
basisFcnsOrder:  vector which describes new order of basis funtions
                 corresponding to 'small' quantities, double [Q x 1]
```

Syntax

```
[C, newMesh] = computePositiveCMat(mesh, basisFcns, polygons)
[C, newMesh, newBasisFcns] = computePositiveCMat( ...
  mesh, basisFcns, polygons, 1)
```

Namespace +results

AToM:+results:calculateCharacteristicAngle

calculateCharacteristicAngle: calculate characteristic angle from eigennumber

Calculates characteristic angle.

Inputs

```
eigennumber:  eigen numbers, double [N x M]
               N - number of modes
               M - number of frequencies
```

Outputs

```
characteristicAngle:  characteristic angles, double [N x M]
                       N - number of modes
                       M - number of frequencies
```

Syntax

```
characteristicAngle = calculateCharacteristicAngle(eigennumber)
```

AToM:+results:calculateCharge

calculateCharge: calculate charge density on given structure

Calculates charge distribution.

Inputs

```
mesh:       mesh struct created by AToM, struct [1 x 1]
basisFcns:  basis functions struct created by AToM, struct [1 x 1]
iVec:       vector of expansion coeficients from AToM MoM,
            double [N x 1]
```

Inputs

(optional)

```
points:   Cartesian coordinates of the points, double [N x 3]
```

Outputs

```
divJ:     divergence of the current density, double [N x 1]
points:   Cartesian coordinates of the points, double [N x 3]
```

Syntax

```
divJ = results.calculateCharge(mesh, basisFcns, iVec)
[divJ, points] = results.calculateCharge(mesh, basisFcns, iVec, points)
```

AToM:+results:calculateCurrent

calculateCurrent: calculate current density on given structure

Calculates current density from given results comming from MOM.

Inputs

```
mesh:      mesh struct created by AToM, struct [1 x 1]
basisFcns: basis functions struct created by AToM, struct [1 x 1]
iVec:      vector of expansion coeficients from AToM MoM,
            double [N x 1]
```

Inputs

```
(optional)
points:   Cartesian coordinates of the points, double [N x 3]
```

Outputs

```
Jx:       x component of the current density, double [N x 1]
Jy:       y component of the current density, double [N x 1]
Jz:       z component of the current density, double [N x 1]
points:   Cartesian coordinates of the points, double [N x 1]
```

Syntax

```
[Jx, Jy, Jz] = results.calculateCurrent(mesh, basisFcns, iVec)
[Jx, Jy, Jz, points] = results.calculateCurrent(mesh, basisFcns, ...
                       iVec, points)
```

AToM:+results:calculateCurrentDecomposition

calculateCurrentDecomposition: calculates current decoposition

This decomposition is based on S matatrix which is defined in https://arxiv.org/pdf/1709.09976.pdf.

Inputs

```
mesh:      mesh struct created by AToM, struct [1 x 1]
basisFcns: basis functions struct created by AToM, struct [1 x 1]
iVec:      vector of expansion coeficients, double [M x N]
            M - number of basis functions
            N - number of frequency points
frequency: vector of frequencies, double [1 x N]
```

Inputs

```
(parameters)
'maxDegreeL'       maximal degree of used spherical harmonics,
                    double [1 x 1], default: 5
'quadratureOrder'  used gaussian quadrature order in integration,
                    double [1 x 1], integer <1, 12>, default: 1
```

Outputs

```
decomposition:   decomposition matrix, double [N x M]
                  N - number of used modes
                  M - number of frequencies
indexMatrix:     indexation matrix used to identify modes,
                  double [5 x N]
                  N - number of used modes
```

Syntax

```
decomposition = results.calculateCurrentDecomposition(mesh, ...
   basisFcns, iVec, frequency)
[decomposition, I] = results.calculateCurrentDecomposition(mesh, ...
   basisFcns, iVec, frequency, 'maxDegreeL', 10, 'quadratureOrder', 2);
```

AToM:+results:calculateEigennumber

calculateEigennumber: calculates eigennumber from characteristic angle

Calculates eigennumbers.

Inputs

characteristicAngle:

```
characteristic angles, double [N x M]
 N - number of modes
 M - number of frequencies
```

Outputs

eigennumber:

```
eigen numbers, double [N x M]
 N - number of modes
 M - number of frequencies
```

Syntax

```
eigennumber = calculateEigennumber(characteristicAngle)
```

AToM:+results:calculateFarField

calculateFarField: computes far-field for given structure and current

Calculates far-field.

Inputs

mesh: mesh struct created by AToM, struct [1 x 1]

frequency: value of frequency, double [1 x 1]

Inputs

```
(parameters)
'basisFcns'  basis functions struct created by AToM, struct [1 x 1]
'iVec'       vector of expansion coeficients, double [N x 1]
'theta'      vector of points in theta spherical coordinate,
              double [1 x N], default: linspace(0, pi, 46)
'phi'        vector of points in phi spherical coordinate,
              double [1 x N], default: linspace(0, 2*pi, 91)
'J'          current density to be plotted, double [N x 3]
'Jx'         x component of current density to be plotted,
              double [N x 1]
'Jy'         y component of current density to be plotted,
              double [N x 1]
'Jz'         z component of current density to be plotted,
              double [N x 1]
'V'          voltage distribution over boundary points, only BEM,
              double[N x 1]
'quadOrder'  quadrature order, double [1 x 1]]
```

Outputs

farFieldStructure: structure with all computed quantities, struct [1 x 1]

Syntax

```
farFieldStructure = results.calculateFarField(mesh, frequency, ...
  'basisFcns', basisFcns);
farFieldStructure = results.calculateFarField(mesh, frequency, ...
  'basisFcns', basisFcns, 'theta', linspace(0, pi, 46), ...
  'phi', linspace(0, 2*pi, 91));
```

AToM:+results:calculateNearField

calculateNearField: computes near-field for given structure and current

Calculates near-field.

Inputs

```
mesh:       mesh struct created by AToM, struct [1 x 1]
basisFcns:  basis functions struct created by AToM, struct [1 x 1]
iVec:       vector of expansion coeficients, double [N x 1]
frequency:  value of frequency, double [1 x 1]
uPoints:    vector of points in first dimension, double [1 x N]
vPoints:    vector of points in secont dimension, double [1 x M]
plane:      near field plane, 'x', 'y', 'z'
distance:   perpendicular distance from origin, double [1 x 1]
```

Outputs

```
nearFieldStructure:   structure with all computed quantities,
                       struct [1 x 1]
```

Syntax

```
nearFieldStructure = results.calculateNearField(mesh, basisFcns, ...
   iVec, frequency, uPoints, vPoints, plane, distance)
```

AToM:+results:calculateQFBW

calculateQFBW: computes Q_FBW

Calculates quality factor Q_FBW.

Inputs

```
zIn:        input impedance, double [N x 1]
             N - number of frequencies
frequency:  frequency list, double [N x 1], [1 x N]
alpha:      FBW threshold, double [1 x 1]
```

Outputs

```
QFBW:       quality factor QFBW, double [M x 1]
f           frequencies corresponding to QFBW, double [M x 1]
```

Syntax

```
[QFBW, f] = results.calculateQFBW(zIn, frequency, alpha)
```

AToM:+results:calculateQZ

calculateQZ: computes Q_Z

Calculates quality factor Q_Z.

Inputs

```
zIn:       input impedance, double [N x 1]
            N - number of frequencies
frequency: frequency list, double [N x 1], [1 x N]
```

Outputs

```
QZ:      quality factor QZ, double [M x 1]
QZTuned: quality factor QZ tuned to resonance, double [M x 1]
```

Syntax

```
[QZ, QZTuned] = results.calculateQZ(zIn, frequency)
```

AToM:+results:calculateRCS

calculateRCS: computes monostatic/bistatic radar cross section

Calculates RCS.

Inputs

```
mesh:       mesh struct created by AToM, struct [1 x 1]
basisFcns:  basis functions struct created by AToM, struct [1 x 1]
iVec:       vector of expansion coeficients from AToM MoM,
             double [N x 1]
frequency:  value of frequency, double [1 x 1]
```

Inputs

```
(parameters)
 'theta'     vector of points in theta spherical coordinate,
              double [1 x N], default: linspace(0, pi, 46)
 'phi'       vector of points in phi spherical coordinate,
              double [1 x M], default: linspace(0, 2*pi, 91)
 'component' specify componenit of used radiation intesity,
              char [1 x N], 'theta', 'phi', 'total', default: 'total'
```

Outputs

```
RCS:   radar cross section (RCS), double [N x M]
        N - number of points in theta
        M - number of points in phi
theta: vector of points in theta spherical coordinate, double [1 x N]
phi:   vector of points in phi spherical coordinate, double [1 x M]
```

Syntax

```
[RCS, theta, phi] = calculateRCS(mesh, basisFcns, iVec, frequency);
[RCS, theta, phi] = calculateRCS(mesh, basisFcns, iVec, frequency, ...
  'component', 'theta');
```

AToM:+results:calculateS

calculateS: computes s parameter from z parameters

Calculates S parameters.

Inputs

```
zIn:        input impedance, double [N x N x M]
             N - number of ports
             M - number of frequencies
```

Inputs

```
 (parameters)
 'z0'       characteristic impedance, double [1 x 1], default 50 Ohm
```

Outputs

```
S:          s parameters, double [N x N x M]
```

Syntax

```
S = results.calculateS(zIn);
S = results.calculateS(zIn, 'z0', 50);
```

AToM:+results:plotBasisFcns

plotBasisFcns: generates plot of given basis functions

Creates plot of basis functions.

Inputs

```
mesh:      mesh struct created by AToM, struct [1 x 1]
basisFcns: basis functions struct created by AToM, struct [1 x 1]
```

Inputs

```
(parameters)
 'options'     ploting options, list below, struct [1 x 1]
 'handles'     handles to the modification, struct [1 x 1]
 'template'    template containing graphic rules, struct [1 x 1]

Options structure, logical [1 x 1] in each field
 options.showBasisFcns         generate basis functions
 options.showBasisFcnsNumbers  generate numbers of basis functions
```

Outputs

```
handles:   structure with all graphic objects, struct [1 x 1]
```

Syntax

```
handles = results.plotBasisFcns(mesh, basisFcns)
handles = results.plotBasisFcns(mesh, basisFcns, 'options', options)
```

AToM:+results:plotCharacteristicAngle

plotCharacteristicAngle: generates plot of given characteristic angle

Creates plot of characteristic angle.

Inputs

```
(parameters)
 'characteristicAngle'     characteristic angles, double [N x M]
                            N - number of modes
                            M - number of frequencies
 'eigennumber'             eigen numbers, double [N x M]
                            N - number of modes
                            M - number of frequencies
 'frequency'               frequency list, double [M x 1]
 'handles'                 handles to the modification, struct [1 x 1]
 'template'                template containing graphic rules,
                            struct [1 x 1]
```

Outputs

```
 handles:   structure with all graphic objects, struct [1 x 1]
```

Syntax

```
handles = results.plotCharacteristicAngle('frequency', frequency, ...
  'characteristicAngle', characteristicAngle);
handles = results.plotCharacteristicAngle('frequency', frequency, ...
  'eigennumber', eigennumber);
```

AToM:+results:plotCharge

plotCharge: generates plot of charge density on given structure

Creates plot of charge density.

Inputs

```
mesh:      mesh struct created by AToM, struct [1 x 1]
```

Inputs

```
(parameters)
 'basisFcns'   basis functions struct created by AToM, struct [1 x 1]
 'iVec'        vector of expansion coeficients, double [N x 1]
 'divJ'        vector of charge computed in triangle centroid,
                double [N x 1]
 'divJnodes'   vector of charge computed in mesh nodes, double [N x 1]
 'part'        part of plotted current, {'re', 'im', 'abs'}
 'options'     ploting options, list below, struct [1 x 1]
 'handles'     handles to the modification, struct [1 x 1]
 'template'    template containing graphic rules, struct [1 x 1]

Options structure, logical [1 x 1] in each field
 options.showCharge              generate triangles with color map
                                according to calculated charge density
 options.colorbar                show colorbar
```

Outputs

```
handles:   structure with all graphic objects, struct [1 x 1]
```

Syntax

```
handles = results.plotCharge(mesh, basisFcns, iVec)
handles = results.plotCharge(mesh, basisFcns, iVec)
handles = results.plotCharge(mesh, basisFcns, iVec, 'part', 'abs')
```

AToM:+results:plotCurrent

plotCurrent: Generates plot of current density on given structure

Creates plot of current density.

Inputs

```
mesh:      mesh struct created by AToM, struct [1 x 1]
```

Inputs

```
(parameters)
 'basisFcns'   basis functions struct created by AToM, struct [1 x 1]
 'iVec'        vector of expansion coeficients, double [N x 1]
 'J'           current density to be plotted, double [N x 3]
 'Jnodes'      current density to be plotted computed in mesh nodes,
                double [N x 3], important for interpolated colors
 'Jx'          x component of current density to be plotted,
                double [N x 1]
 'Jy'          y component of current density to be plotted,
                double [N x 1]
 'Jz'          z component of current density to be plotted,
                double [N x 1]
 'part'        part of plotted current, {'re', 'im', 'abs'}
 'scale'       sets the scale of the color map,
                {'linear', 'normalized','logarithmic'}
 'arrowScale'  sets the scale of the arrows,
                {'uniform', 'proportional'}
 'arrowLength' sets maximal absolute length of arrow, double [1 x 1]
 'options'     ploting options, list below, struct [1 x 1]
 'handles'     handles to the modification, struct [1 x 1]
 'template'    template containing graphic rules, struct [1 x 1]

Options structure, logical [1 x 1] in each field
 options.showCurrentIntensity   generate triangles with color map
                               accordingto calculated current density
 options.showCurrentArrows      generate arrows according to current
                               density
 options.colorbar               show colorbar
```

Outputs

```
handles:   structure with all graphic objects, struct [1 x 1]
```

Syntax

```
results.plotCurrent(mesh, 'basisFcns', basisFcns, 'iVec', iVec)
handles = results.plotCurrent(mesh, 'basisFcns', basisFcns, 'iVec', ...
 iVec)
handles = results.plotCurrent(mesh,  'basisFcns', basisFcns, ...
 'iVec', iVec, 'part', 'abs', 'scale', 'linear')
```

AToM:+results:plotCurrentDecomposition

plotCurrentDecomposition: generates plot of current decomposition

Creates plot of current decomposition.

Inputs

```
(parameters)
 'decomposition'   decomposition matrix, double [N x M]
                    N - number of used modes
                    M - number of frequencies
 'indexMatrix'     indexation matrix used to identify modes,
                    double [5 x N]
                    N - number of used modes
 'frequency'       frequency list, double [M x 1]
 'mesh'            mesh struct created by AToM, struct [1 x 1]
 'basisFcns'       basis functions struct created by AToM, struct [1 x 1]
 'iVec'            vector of expansion coeficients, double [N x 1]
 'threshold'       threshold to filter modes, double [1 x 1]
 'options'         ploting options, list below, struct [1 x 1]
 'handles'         handles to the modification, struct [1 x 1]
 'template'        template containing graphic rules, struct [1 x 1]
```

Outputs

handles: structure with all graphic objects, struct [1 x 1]

Syntax

```
handles = results.plotCurrentDecomposition('mesh', mesh, ...
  'basisFcns', basisFcns, 'iVec', iVec, 'frequency', frequency)
handles = results.plotCurrentDecomposition( ...
  'decomposition', decomposition, 'frequency', frequency)
handles = results.plotCurrentDecomposition( ...
  'decomposition', decomposition, 'indexMatrix', indexMatrix, ...
  'frequency', frequency)
```

AToM:+results:plotEigennumber

plotEigennumber: generates plot of given eigen numbers

Creates plot of eigennumbers.

Inputs

```
(parameters)
 'eigennumber'              eigen numbers, double [N x M]
                             N - number of modes
                             M - number of frequencies
 'characteristicAngle'      characteristic angles, double [N x M]
                             N - number of modes
                             M - number of frequencies
 'frequency'                frequency list, double [M x 1]
 'handles'     handles to the modification, struct [1 x 1]
 'template'    template containing graphic rules, struct [1 x 1]
```

Outputs

```
handles:   structure with all graphic objects, struct [1 x 1]
```

Syntax

```
handles = results.plotEigennumbers('frequency', frequency, ...
  'eigennumber', eigennumber);
handles = results.plotEigennumbers('frequency', frequency, ...
  'characteristicAngle', characteristicAngle);
```

ATᴏM:+results:plotFarField

plotFarField: generates plot of far-field

Creates plot of far-field.

Inputs

```
 (parameters)
  'mesh'          mesh struct created by AToM, struct [1 x 1]
  'basisFcns'     basis functions struct created by AToM, struct [1 x 1]
  'iVec'          vector of expansion coeficients, double [N x 1]
  'theta'         vector of points in theta spherical coordinate,
                   double [1 x N]
  'phi'           vector of points in phi spherical coordinate,
                   double [1 x N]
  'frequency'     value of frequency, double [1 x 1]
  'farField'      data for given theta and phi, double [N x M]
  'options'       ploting options, list below, struct [1 x 1]
  'handles'       handles to the modification, struct [1 x 1]
  'template'      template containing graphic rules, struct [1 x 1]
  'radius'        value for scaling size of plotted far-field,
                   double [1 x 1]
  'userFunction'  function handle used to choose what to plot,
                   function handle [1 x 1],
                   default function @(ff) abs(ff.D)

 Options structure, logical [1 x 1] in each field
  options.showFarField                generate surface of far-field
  options.showSphericalCoordinates    generate spherical coordinates to
                                     figure
  options.showCartesianCoordinates    generate cartesian coordinates to
                                     figure
  options.showColorBar                show colorbar
```

Outputs

```
  handles:   structure with all graphic objects, struct [1 x 1]
```

Syntax

```
 handles = results.plotFarField('mesh', mesh, 'basisFcns', bf, ...
  'iVec', iVec, 'frequency', frequency);
 handles = results.plotFarField('theta', thetaVector, ...
 'phi', phiVector, 'farField', farFieldMatrix)
```

AToM:+results:plotFarFieldCut

plotFarFieldCut: generates plot of far-field cut

Creates plot of far-field cut.

Inputs

```
(parameters)
 'farField'    data for given theta and phi, double [N x M]
 'theta'       vector of points in theta spherical coordinate,
              double [1 x N]
 'phi'         vector of points in phi spherical coordinate,
              double [1 x N]
 'thetaCut'    value of theta for cut in theta, double [1 x 1]
 'phiCut'      value of phi for cut in phi, double [1 x 1]
 'handles'     handles to the modification, struct [1 x 1]
 'template'    template containing graphic rules, struct [1 x 1]
```

Outputs

```
 handles:   structure with all graphic objects, struct [1 x 1]
```

SYNTAX

```
results.plotFarFieldCut('farField', farField, 'theta', theta, ...
  'phi', phi, 'thetaCut', thetaCut);
handles = results.plotFarField('farField', farField, 'theta', theta, ...
 'phi', phi, 'thetaCut', thetaCut);
```

AToM:+results:plotMesh

plotMesh: generates plot of given structure

Creates plot of mesh.

Inputs

```
mesh:      mesh struct created by AToM, struct [1 x 1]

Options structure, logical [1 x 1] in each field
 options.showNodes              generate point for each node
 options.showNodeNumbers        generate numbers of nodes
 options.showEdges              generate edges (line object)
 options.showEdgesNumbers       generate numbers of edges
 options.showEdgesArrows        generate arrows to show orientation of
                                 edges
 options.showTriangles          generate triangles (patch object)
 options.showTriangleNumbers    generate numbers of triangles
```

Outputs

```
handles:   structure with all graphic objects, struct [1 x 1]
```

Syntax

```
handles = results.plotMesh(mesh)
handles = results.plotMesh(mesh, 'options', options)
```

AToM:+results:plotNearField

plotNearField: generates plot of near-field

Creates plot of near-field.

Inputs

```
(parameters)
 'mesh'          mesh struct created by AToM, struct [1 x 1]
 'basisFcns'     basis functions struct created by AToM, struct [1 x 1]
 'iVec'          vector of expansion coeficients, double [N x 1]
 'frequency'     value of frequency, double [1 x 1]
 'uPoints'       vector of points in first dimension, double [1 x N]
 'vPoints'       vector of points in secont dimension, double [1 x M]
 'plane'         near field plane, {'x', 'y', 'z'}
 'distance'      perpendicular distance from origin, double [1 x 1]
 'nearField'     data for given uPoints and vPoints, double [N x M]
 'options'       ploting options, list below, struct [1 x 1]
 'handles'       handles to the modification, struct [1 x 1]
 'template'      template containing graphic rules, struct [1 x 1]
 'userFunction'  function handle used to choose what to plot,
                  function handle [1 x 1], default function @(nf) nf.E
```

Outputs

```
handles:   structure with all graphic objects, struct [1 x 1]
```

Syntax

```
handles = results.plotNearField('nearField', nearField, ...
  'uPoints', uPoints,  'vPoints', vPoints, 'plane', 'x', ...
  'distance', distance);
```

AToM:+results:plotQ

plotQ: generates plot of quality factor Q

Creates plot of Q factor.

Inputs

```
Q:          quality factor Q, double [N x 1]
             N - number of frequencies
frequency: vector of frequencies, double [1 x N]
```

Inputs

```
(parameters)
 'handles'       handles to the modification, struct [1 x 1]
 'template'      template containing graphic rules, struct [1 x 1]
```

Outputs

```
handles:   structure with all graphic objects, struct [1 x 1]
```

Syntax

```
handles = results.plotQ(Q, frequency)
```

AToM:+results:plotRCS

plotRCS: generates plot of monostatic/bistatic radar cross section

Creates plot of RCS.

Inputs

```
(parameters)
 'mesh'                  mesh struct created by AToM, struct [1 x 1]
 'basisFcns'             basis functions struct created by AToM,
                          struct [1 x 1]
 'iVec'                  vector of expansion coeficients, double [N x 1]
 'theta'                 vector of points in theta spherical coordinate,
                          double [1 x N]
 'phi'                   vector of points in phi spherical coordinate,
                          double [1 x N]
 'frequency'             value of frequency, double [1 x N]
 'component'             specify componenit of used radiation intesity,
                          char [1 x N],
                          'theta', 'phi', 'total', default: 'total'
 'RCS'                   data for given theta and phi, double [L x M x N]
                          L - number of points in theta
                          M - number of points in phi
                          N - number of frequencies
 'independentVariable'   variable on x axis, char [1 x N]
                          'theta', 'phi', 'frequency', default: 'theta'
 'fixedDimensionTheta'   value of fixed dimension theta, double [1 x 1]
 'fixedDimensionPhi'     value of fixed dimension theta, double [1 x 1]
 'options'               ploting options, list below, struct [1 x 1]
 'handles'               handles to the modification, struct [1 x 1]
 'template'              template containing graphic rules, struct [1 x 1]
```

Outputs

```
 handles:   structure with all graphic objects, struct [1 x 1]
```

Syntax

```
results.plotRCS('RCS', RCS, 'theta', theta, 'phi', phi, ...
  'independentVariable', 'theta', 'fixedDimensionPhi', pi);
```

AToM:+results:plotS

plotS: generates plot of s parameters

Creates plot of S parameters.

Inputs

```
(parameters)
'zIn'          input impedance, double [N x N x M]
                N - number of ports
                M - number of frequencies
'z0'           characteristic impedance, double [1 x 1], default 50 Ohm
's'             s parameters, double [N x N x M]
'frequency'    list of frequencies, double [M x 1], [1 x M]
'select'       selection of visualised curves, double [P x 1]
                selection is based on MATLAB linear indexing

'scale'        select scale, char 'linear', 'dB', default: 'linear'
```

Outputs

handles: structure with all graphic objects, struct [1 x 1]

Syntax

```
handles = results.plotS('frequency', frequency, 'zIn', zIn, 'z0', 50);
handles = results.plotS('frequency', frequency, 's', S);
handles = results.plotS('frequency', frequency, 's', S, 'select', 1);
```

AToM:+results:standardizeFigure

standardizeFigure: standardize figure appearance

Controls the appearance of figure and ensures the normalization of the figure. Default profile is saved in results.figureProfiles. Another profile can be created. Create your own profile, place it to .\+results\+figureProfiles folder and then call it by its name.

Inputs

handles: structure of graphical objects, struct [1 x 1]

userProfileName: structure of graphical preferences, struct [1 x 1]
or name of figureProfile, char [1 x N]

Syntax

```
standardizeFigure(handles)
standardizeFigure(handles, 'userProfileName')
```