AToM Documentation

AToM Team

March 23, 2020

Contents

1 Kernel **1**
- 1.1 Start of AToM . . . 1
- 1.2 Termination of AToM . . . 1
- 1.3 How to forcibly restart AToM . . . 1
- 1.4 AToM Project . . . 2
  - 1.4.1 Creation of the AToM Project . . . 2
  - 1.4.2 Opening of the existing AToM project . . . 2
  - 1.4.3 How to switch between projects . . . 2
  - 1.4.4 Working with (selected) project in the MATLAB command line . . . 2
- 1.5 History . . . 2
- 1.6 Workspace . . . 2
  - 1.6.1 Public Methods . . . 3
  - 1.6.2 Public Properties . . . 4
  - 1.6.3 Preferences . . . 4
- 1.7 Workspace Viewer . . . 4
  - 1.7.1 Preferences . . . 4

2 Geom **7**
- 2.1 Geom class . . . 7
- 2.2 Geom Primitives . . . 7
- 2.3 Transformations . . . 16
- 2.4 Boolean operations . . . 17
- 2.5 Common methods . . . 18

3 Mesh **21**
- 3.1 General . . . 21
- 3.2 Creating meshes from AToM geometry . . . 21
- 3.3 Imported and exported meshes . . . 22
- 3.4 Getting data from mesh . . . 23
- 3.5 Public functions . . . 23

4 Physics **27**
- 4.1 Frequency List . . . 27
- 4.2 Symmetry Planes . . . 27
- 4.3 Materials . . . 27
- 4.4 Public Methods . . . 27
- 4.5 Public Properties . . . 29
- 4.6 Physics Viewer . . . 30
- 4.7 Feeding . . . 30
  - 4.7.1 Plane Waves . . . 30
  - 4.7.2 Geometry Discrete Ports . . . 31
  - 4.7.3 Mesh Ports . . . 32
  - 4.7.4 Delta Gap . . . 33

4.7.5 Lumped Circuits . . . 33
4.7.6 Public Methods . . . 35
4.7.7 Public Properties . . . 37

5 Solvers **39**
5.1 BEM . . . 39
5.1.1 General . . . 39
5.1.2 Public methods . . . 40
5.1.3 How to use BEM without AToM . . . 40
5.2 MoM1D . . . 41
5.2.1 Public Properties . . . 42
5.3 MoM2D . . . 43
5.3.1 Public Properties . . . 45

6 Generalized Eigenvalue Problem Solver **47**
6.1 Tracking . . . 47
6.1.1 Correlation Computing . . . 48
6.2 Adaptive Frequency Solver . . . 49
6.3 Public Functions . . . 50
6.4 Public Methods . . . 52
6.5 Public Properties . . . 53
6.5.1 General . . . 53
6.5.2 Pre-processing . . . 53
6.5.3 GEP . . . 54
6.5.4 Post-processing . . . 55
6.5.5 Tracking . . . 55
6.5.6 AFS . . . 55
6.6 How to use GEP . . . 56
6.6.1 Results . . . 58
6.6.2 Setting of properties . . . 58
6.6.3 Run AFS . . . 59
6.6.4 Using S-matrix decomposition . . . 59
6.7 Customization . . . 59
6.7.1 Customize Inner Solver . . . 59
6.7.2 User-defined `eig`/`eigs` preprocessing and postprocessing . . . 62
6.7.3 Customize Correlation Table . . . 64
6.8 GEP without AToM . . . 65

7 Results **69**
7.1 Overview . . . 69
7.2 Public functions . . . 70
7.3 Results class . . . 74
7.3.1 Structure of repository and data types . . . 74
7.3.2 GUI . . . 78
7.4 Templates and other user preferences . . . 82
7.4.1 Templates . . . 82
7.4.2 Standardize figure and user profile . . . 83

8 Utilities **85**
8.1 Matrix S . . . 85
8.2 Subregion matrix C . . . 85
8.3 Symmetry matrices . . . 86
8.3.1 Nomenclature . . . 86
8.3.2 Public functions . . . 87

8.3.3 Example 88

9 Graphical User Interface (GUI) 91

9.1 Viewers with separated figure 91
9.1.1 Public Methods 91
9.2 Viewers integrated in Main Viewer 92
9.3 Color Templates 92
9.4 Class GUI 93
9.4.1 Public Methods 93
9.4.2 Public Properties 93
9.4.3 Preferences 93
9.5 Design Viewer 94
9.5.1 Public Methods 94

Index 97

List of Figures

1.1 Workspace Viewer with several variables. 5

2.1 Point defined in 3D and its properties. 8
2.2 Line defined in 3D and its properties. 8
2.3 PolyLine defined in 3D and its properties. 8
2.4 RectangleFrame defined in 3D and its properties. 9
2.5 ParallelogramFrame defined in 3D and its properties. 10
2.6 CircleArc defined in 3D and its properties. 11
2.7 EllipseArc defined in 3D and its properties. 12
2.8 EquationCurve defined in 3D and its properties. 12
2.9 Rectangle defined in 3D and its properties. 13
2.10 Parallelogram defined in 3D and its properties. 14
2.11 Circle defined in 3D and its properties. 15
2.12 Ellipse defined in 3D and its properties. 15
2.13 Polygon defined in 3D and its properties. 16
2.14 Unite method performed on two 2D objects: `'circ'` + `'par'`. 18
2.15 Subtract method performed on two 2D objects: `'circ'` + `'par'`. 18

3.1 Example of density function from the Listing 3.1 and its impact on the mesh generation. 22
3.2 A discretized square using different types of triangulation. 22

4.1 Structure of modified bowtie antenna with two geometry symmetry planes. Main bowtie shape is drawn entirely, but an ending sheet is drawn on both sides of antenna differently. In the resulting mesh can be seen (symmetry planes are not shown), that mesh is symmetrized according to XY and YZ symmetry planes and was computed from geometry in quadrant with positive X and Z coordinate. 28
4.2 Physics Viewer. 30
4.3 Dialogue for plane wave definition. 30
4.4 Dialogues for adding discrete port on 2D geometry (left) and triangular mesh (right). Dialogues are accessible by icon in Design Viewer. 32
4.5 Example of correct and incorrect discrete 2D port positions. Port n. 3 is incorrect because it is not clear between which triangles in mesh current should flow. And port n. 5 is on the edge of geometry, where also current can not flow. On the other hand, ports n. 1, 2 and 4 has always simple planar part of geometry on both sides, *i.e.*, direction of current it is clear. 33
4.6 Example of correct and incorrect discrete 1D port positions. Port n. 1 is correctly placed on loop antenna, but port n. 2 is placed on the end of loop, which is invalid. 33
4.7 Example of definition mesh port on 2D mesh. Port n. 1 has `elements = [26 ... 2;28 20;35 39;40 45]`. In the first column are numbers of triangles with positive voltage. 34
4.8 Dialogue after adding port enables enabling and editing delta gap and lumped circuit on port. 34

4.9 Delta gap is always in series with lumped circuit, but lumped circuit itself can be serial, or parallel. It is also possible to disable the delta gap (`isEnaled = false`) and port is then represented just as lumped circuit. When some element of lumped circuit has to be intentionally unused, its value has to be `NaN`. Delta gap is in circuit schematically shown as voltage source, but can be current source as well, or can be disabled at all. . . . 35

5.1 MoM1D Graphical User Interface . . . 42
5.2 MoM2D Graphical User Interface . . . 45

6.1 Tracking algorithm . . . 48
6.2 Algorithm of Adaptive Frequency Solver (AFS). . . . 50
6.3 GUI of GEP Solver. Basic properties mentioned in Section 6.5 can be set in main window, the rest in Advance properties window. . . . 57
6.4 GEP Status Window . . . 58
6.5 Cutouts of GEP status Window showing gradual refinement around resonance using AFS option of GEP. . . . 59

7.1 Structure of the inner layers in Results . . . 76
7.2 Graphical user interface of Resultg - Load tab . . . 78
7.3 Results GUI - Compute tab . . . 80
7.4 Results GUI - Plot polar tab . . . 80
7.5 Results GUI - Plot 2D tab . . . 81
7.6 Results GUI - Plot 3D tab . . . 82
7.7 Example of combination more layers in one figure . . . 82
7.8 Results GUI - Explore tab . . . 83

8.1 Example how the **C** matrix affects the structure. . . . 86
8.2 Simple symmetric structure. . . . 88

9.1 Main Viewer. Model List Viewer is the tree representing simulation model, Project Viewer is on the bottom part of figure. . . . 92
9.2 Example of error dialogue which is shown every time when invalid step is made when working from GUI. Throwing error to Command Window depends on property `throwErrorToCommandWindow` of GUI. . . . 94
9.3 Design Viewer (DW) with mesh structure and one discrete port. . . . 94

Chapter 1

Kernel

1.1 Start of AToM

AToM can be started by following command written in the MATLAB command line:

```
atom = Atom.start();
```

The assignment into the variable `atom` is particularly important as this object is used for all operations with AToM from the MATLAB command line and it is also important for exit of the AToM (see the next section).

There are advanced possibilities how to start AToM:

- `atom = Atom.start(true);`
- `atom = Atom.start(false);`

The former option is equivalent to `atom = Atom.start();` and starts AToM with GUI, the latter option only start AToM and does not open GUI. For further details related to the start of AToM, see the documentation.

1.2 Termination of AToM

AToM package must be properly turned off by typing `atom.quit;` into the MATLAB command line. Alternatively, AToM can be shut down from GUI clicking on `AToM – Quit`.

Since AToM class is designed with the singleton pattern, there is always only one instance of the AToM. Therefore, in case the AToM is not terminated properly, new start of AToM only opens already opened session.

1.3 How to forcibly restart AToM

In the rare cases when AToM failed or something goes wrong, there is necessity to exit AToM in a non-standard way and restore the session by deleting all instances of the associated AToM classes. The following commands are recommended:

```
close all force
clear classes
```

Notice that all variables from MATLAB base workspace are deleted and all figures are closed. Therefore, you may backup/save your work before.

In principle, it may happen that the above mentioned procedure does not help. Then try to restart MATLAB.

1.4 AToM Project

1.4.1 Creation of the AToM Project

If the user wants to start with the antenna design from scratch, new AToM project has to be created. This can be done by the MATLAB command `atom.createProject('myProject1')` or from the main GUI by clicking on `Create project`. When new project is created, AToM automatically switch to this project.

1.4.2 Opening of the existing AToM project

AToM allows to open projects which were created previously and saved (as `*.atom` file). Then the project can be opened from the MATLAB command line as `atom.openProject('myProject1')` or from the main GUI clicking on `Open project`. Notice that in the command above the path to the file must be added if the project is not located in the actual folder. When new project is opened, AToM automatically switch to this project.

1.4.3 How to switch between projects

There can be many projects opened simultaneously. Then, the user can switch between them either in main GUI or by typing the following command in the MATLAB command line
`atom.selectProject('myProject1');` Alternatively, the projects can be selected with respect to their position as, *e.g.*, `atom.selectProject(1);`

1.4.4 Working with (selected) project in the MATLAB command line

Once the project is created or opened, the user can work with this project freely, either in GUI or in the MATLAB command line. Then the command always starts with
`atom.selectedProject.<command>` where `<command>` is an operation done by the user. The various commands are related to creation and adjustment of geometry, discretization, physics settings and simulation with the solvers.

1.5 History

While AToM project is opened, all following operations related to that projects are recorded as MATLAB-compatible commands. The user can see this list in the AToM History Viewer, which can be opened by clicking on History button in the main GUI, or by `atom.gui.historyViewer.open`

Remarkable properties of the History Viewer are:

- The ability to export the complete list of commands as AToM-executable standalone project. To do that, click on `Write script` button or use command
`atom.selectedProject.history.writeScript('myScript.m');` Notice that all the operations with the opened projects are also recorded in `<projectName.log>` files, which serve as the backup files. These log files cannot be deleted while the corresponding projects are opened.
- Add commentary between selected lines, to do that, click on `Add Comment`. The commentary can be also removed or edited.

1.6 Workspace

AToM Workspace enables creation and management of variables which allow easy parametrization of almost everything in AToM projects. Every variable in Workspace has its name, expression and description. All variables can be arbitrarily linked through expressions and arbitrary MATLAB functions in search path. Editing of one variable automatically cause change of other linked variables

and also cause immediate change of related project properties (mesh density, frequency points, size of geometry, ...). All values of variables are treated as unitless and its usage in AToM models presumes values in SI base units. Variables has these properties:

- `name` - unique identifier of variable. Has to fulfil MATLAB variable conventions (function `isvarname`) and it is not allowed to use any of these names: `'who'`, `'whos'`, `'which'`, `'clear'` and `'eval'`.
- `expression` - can be arbitrary row vector of characters containing valid MATLAB Expression (*e.g.*, `'cos(0:0.1:2*pi)'`, `'myFcn(pi)'`), numeric matrix (*e.g.*, `[1.1 1e5 exp(2)]`, `uint16(10)`, `magic(4)`) and matrix of logical values (*e.g.*, `true`, `[false(5), ~eye(5)]`). Expression can not contain commands `who`, `whos`, `which`, `clear`, `eval` and `close`. If expression is defined as numerical or logical matrix, passed values are converted by function `mat2str`, *i.e.*, actual computation command of expression will be lost (*e.g.*, `not(eye(2))` is converted to `'[false true;true false]'`). From this perspective using string expressions is preferred and more clear.
- `description` - can be arbitrary character row vector.

1.6.1 Public Methods

Every AToM project has its own Workspace and it is possible to access it in selected project via Command Window by `atom.selectedProject.workspace`. It is not possible to share variables between individual AToM projects.

`assignInMatlab`

`workspace.assignInMatlab()`

Assign all user-defined variables from AToM Workspace in MATLAB base workspace. Note that already existing variables will be replaced and editing of variables in MATLAB base workspace has no influence in variable in AToM Workspace.

`addVariable`

`workspace.addVariable(varName, varExpression, varDescription)`

Add variable with name **`varName`**, expression **`varExpression`** and description **`varDescription`** into workspace. All inputs can be `char` vectors. Moreover `varExpression` can be also numerical and logical 2-D matrix. In this case `mat2str(varExpression)` is used. Definition of description is optional.

`deleteVariable`

`workspace.deleteVariable(varName)`

Delete variable with name **`varName`** from Workspace. It is not possible to delete variable which is used in expression of another variable or in some AToM model.

`editDescription`

`workspace.editDescription(varName, newDescription)`

Edit description of variable **`varName`** to **`newDescription`**.

`editExpression`

`workspace.editExpression(varName, newExpression)`

Edit expression of variable **`varName`** to **`newExpression`**.

`editName`

`workspace.editName(oldName, newName)`

Edit name of variable **`oldName`** to **`newName`**. All expressions of other variables which contain name **`oldName`** will be edited to **`newName`**.

`evalExpression`

`value = workspace.evalExpression(expression)`

Evaluates expression **`expression`** in AToM Workspace. Value of expression is returned in

output argument `value`. Invalid Matlab expression leads to `value = NaN`.

`getValue`

```
varValue = workspace.getValue(varName)
```

Return value of variable `varName` into MATLAB base workspace as `varValue` variable.

`loadConstants`

```
workspace.loadConstants()
```

Import physical constants defined in `models.utilities.constants`. Imported are `c0` (speed of light, 299792458 ms^{-1}), `mu0` (permeability of vacuum, $4\pi \times 10^{-7}$ Hm^{-1}), `ep0` (permittivity of vacuum, $1/(\mu_0 c_0^2)$ Fm^{-1}) and `Z0` (impedance of free space, $c_0\mu_0$ Ω).

`recalculate`

```
workspace.recalculate()
```

Recalculate all variables in AToM Workspace. Useful when using variables calling external user-defined functions which were edited during work with AToM. NOTE: Will be removed in future release.

`recalculateVariable`

```
workspace.recalculateVariable(varName)
```

Force recalculation of variable `varName`. It is useful when expression of variable call external function, which was changed.

1.6.2 Public Properties

`table`

Return variable `workspaceTable` of class `table` into MATLAB base workspace. Table contain list and useful properties of all user-defined AToM variables.

1.6.3 Preferences

`nSignificantDigitsView`

`Double:` {5}

Number of significant digits of variable value shown in Workspace Viewer and in `workspace.table`.

1.7 Workspace Viewer

AToM Workspace has its own GUI called Workspace Viewer and it is accessible via Command Window by `atom.gui.workspaceViewer`. The only public methods are `workspaceViewer.open()` and `workspaceViewer.close()`. Appearance of Workspace Viewer is shown in Fig. 1.1. Viewer also contain class and size of variable's value and overall status of variables. Shown informations are the same as from `workspace.table`.

Adding variables is possible simply by typing its name and expression into empty cell in table. Fill description is optional. To delete variable select one or more cells in variable row by mouse and push “Delete Variable” button. It is possible to select more cells in more rows by Ctrl or Shift and left click. Then all variables on selected rows will be deleted. By pushing “Load Constants” button constants from `models.utilities.constants` will be loaded. By pushing “Assign In Matlab” all variables will be assigned in MATLAB base workspace. By pushing “Recalculate Var.” all selected rows with variables will recalculated.

1.7.1 Preferences

`columnWidth`

`Double:` {[100 100 60 120 50 50 40]}

Width of columns in Workspace Viewer. Double of size 1 x 7.

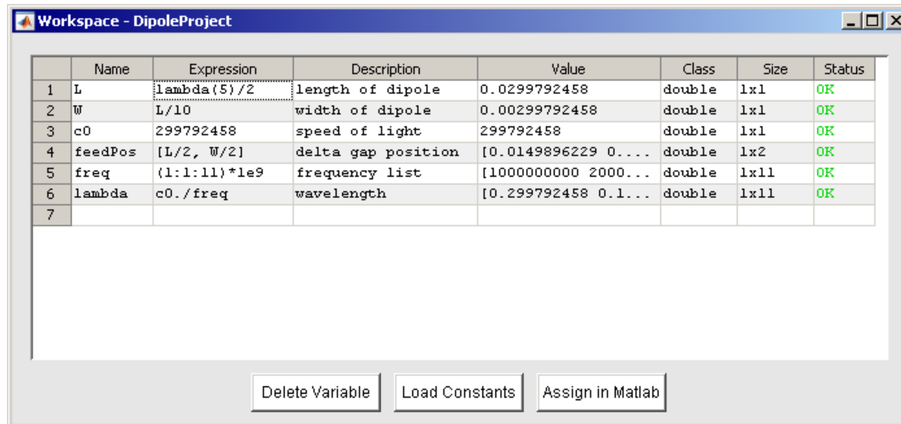


Figure 1.1: Workspace Viewer with several variables.

Chapter 2

Geom

2.1 Geom class

Geom is a standalone toolbox for creation, modification and combination of geometry objects. User can work with Geom using a GUI (see refdesignViewer) or using Matlab command window. In the later case, user needs a reference to refProject or directly to a Geom object (`geom = ... atom.selectedProject.geom`). When working with project every command is s saved to AToM session while when working directly with reference to Geom object not.

Geom methods accept as input variables Matlab values - double or char, user variables from current Matlab workspace or AToM Workspace Variables. In case of using Workspace Variables all objects influenced by the Varibles are instantly modified when the Variable is changed by user.

User defines the desired model from individual primitives. All the primitives can be defined in arbitrary 3D position. The portfolio of available object types is listed below:

- **0D: Point**
- **1D: Line, PolyLine, ParallelogramFrame, EllipseArc, and EquationCurve.**
- **2D: Parallelogram, Ellipse, Polygon.**

Geom object is a container for user-defined objects. All defined objects are sorted into corresponding property of geom object: lines to geom.line, ellipses to geom.ellipse etc. Every object has its unique name. Primitives can be combined to form more complex models using Boolean operations: 1D with 1D objects, 2D with 2D.

AToM Workspace enables creation and management of variables which allow easy parametrization of almost everything in AToM projects. Every variable in Workspace has its name, expression and description. All variables can be arbitrarily linked through expressions and arbitrary Matlab functions in search path. Editing of one variable automatically cause change of other linked variables and also cause immediate change of related project properties (mesh density, frequency points, size of geometry, ...). All values of variables are treated as unitless and its usage in AToM models presumes values in SI base units.

2.2 Geom Primitives

Point

Point is a 0D object created by a command:

```
atom.selectedProject.geom.addPoint(coordVal, name)
```

Variable `coordVal` is either double $[1 \times 3]$ or a Workspace Variable.

Optional variable `name` is char $[1 \times N]$. Workspace Variable cannot be used to define object's name!

Point and it's properties is depicted in Fig. 2.1. For the full header refer to addPoint.

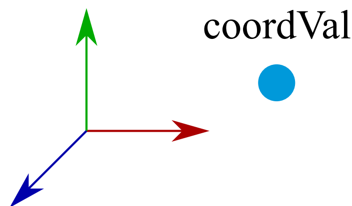


Figure 2.1: Point defined in 3D and its properties.

Line

Line is a 1D object created by a command:

```
atom.selectedProject.geom.addLine(points, name)
```

Variable `points` is either double $[2 \times 3]$ or a Workspace Variable.

Optional variable `name` is char $[1 \times N]$. Workspace Variable cannot be used to define object's name!

Line and it's properties is depicted in Fig. 2.2. For the full header refer to addLine.

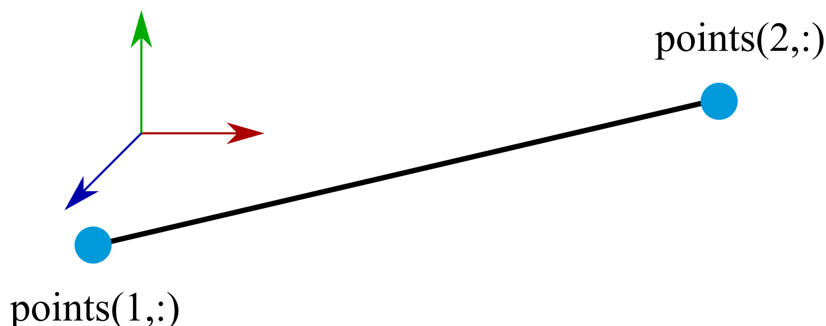


Figure 2.2: Line defined in 3D and its properties.

PolyLine

PolyLine is a 1D object created by a command:

```
atom.selectedProject.geom.addPolyLine(points, name)
```

Variable `points` is either double $[N \times 3]$ or a Workspace Variable.

Optional variable `name` is char $[1 \times N]$. Workspace Variable cannot be used to define object's name!

PolyLine and it's properties is depicted in Fig. 2.3. For the full header refer toaddPolyLine.

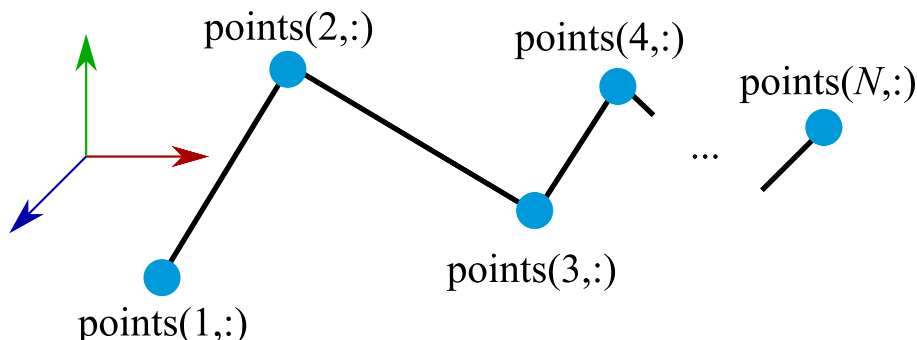


Figure 2.3: PolyLine defined in 3D and its properties.

RectangleFrame

RectangleFrame is a 1D object consisiting of Rectangle's border lines. As rectangle is degenerated form of Parallelogram, all RectangleFrames can be found in:

```
parFrames = atom.selectedProject.geom.parallelogramframe;
```

RectangleFrame can be created by a command:

```
atom.selectedProject.geom.addRectangleFrame(center, width, height, normal, ...
name)
```

Variable `center` is either double $[1 \times 3]$ or a Workspace Variable. It specifies a position of rectangle's center.

Variable `width` is either double $[1 \times 1]$ or a Workspace Variable.

Variable `height` is either double $[1 \times 1]$ or a Workspace Variable.

Optional variable `normal` defines orientation of the RectangleFrame. It has to be char $[1 \times 1]$. Possible values are: `'z'` (default), `'y'` or `'x'`. Object can be rotated arbitrarily using transformation methods - see Transformations

Optional variable `name` is char $[1 \times N]$. Workspace Variable cannot be used to define object's name!

RectangleFrame and it's properties is depicted in Fig. 2.4. For the full header refer to addRectangleFrame.

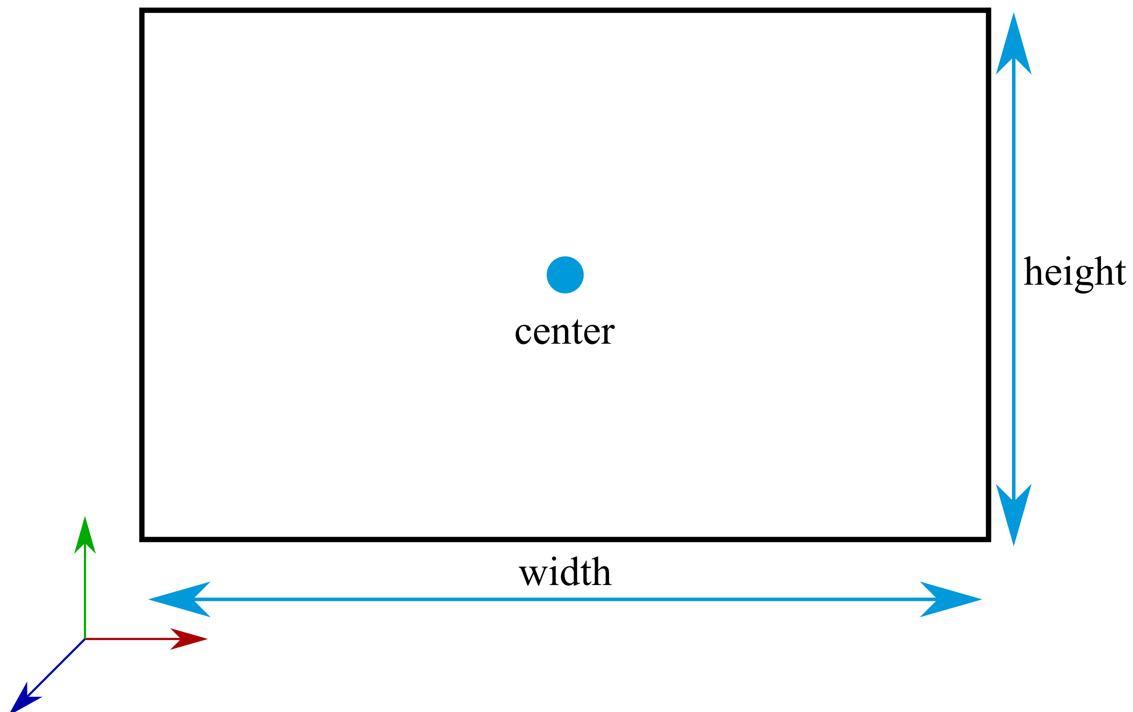


Figure 2.4: RectangleFrame defined in 3D and its properties.

ParallelogramFrame

ParallelogramFrame is a 1D object consisiting of Parallelogram's border lines. All created ParallelogramFrames can be found in:

```
parFrames = atom.selectedProject.geom.parallelogramframe;
```

ParallelogramFrame can be created by a command:

```
atom.selectedProject.geom.addRectangleFrame(lowLeftCorner, lowRightCorner, ...
highLeftCorner, name)
```

Variable `lowLeftCorner` is either double $[1 \times 3]$ or a Workspace Variable. It specifies a position of parallelogram's low-left corner, viewed from side where the object's normal points to.

Variable `lowRightCorner` is either double $[1 \times 3]$ or a Workspace Variable. It specifies a position of parallelogram's low-right corner.

Variable `highLeftCorner` is either double $[1 \times 1]$ or a Workspace Variable. It specifies a position of parallelogram's high-left corner.

Optional variable `name` is char $[1 \times N]$. Workspace Variable cannot be used to define object's name!

ParallelogramFrame and it's properties is depicted in Fig. 2.5. For the full header refer to addParallelogramFrame.

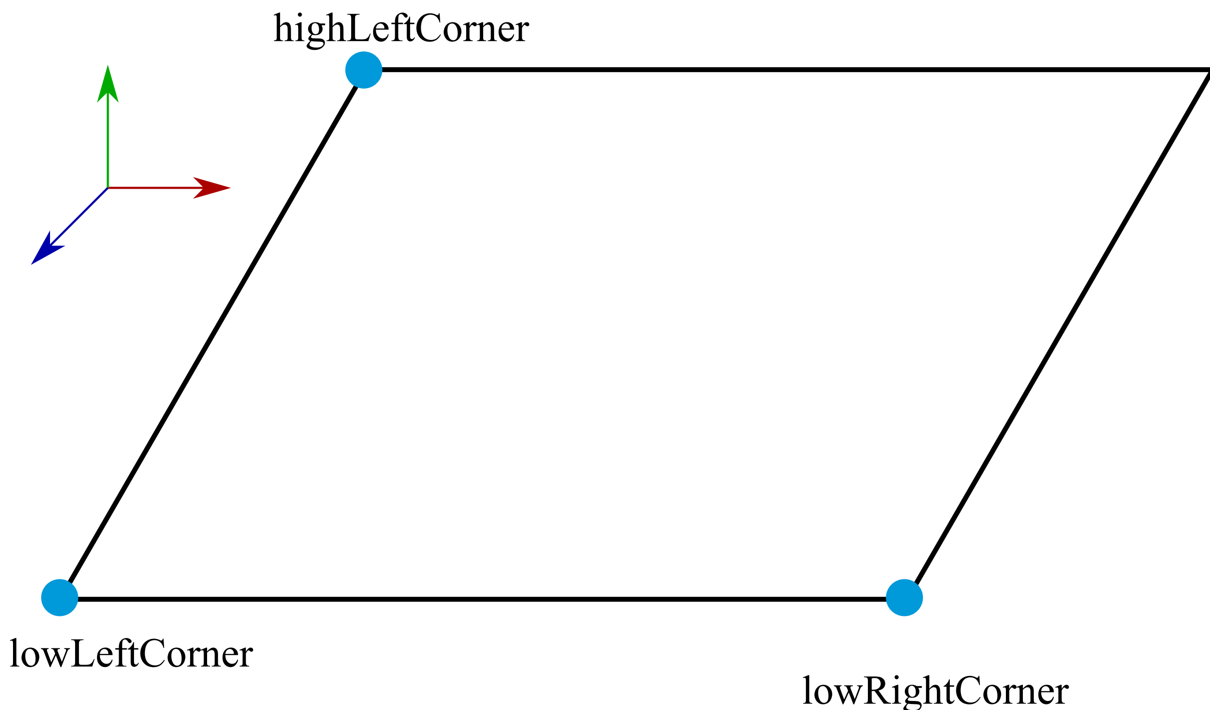


Figure 2.5: ParallelogramFrame defined in 3D and its properties.

CircleArc

CircleArc is a 1D object consisting of Circle's border curve. As circle is degenerated form of ellipse, all CircleArcs can be found in:

```
ellipseArcs = atom.selectedProject.geom.ellipsearc;
```

CircleArc can be created by a command:

```
atom.selectedProject.geom.addCircleArc(center, radius, normal, name);
```

Variable `center` is either double $[1 \times 3]$ or a Workspace Variable. It specifies a position of circle's center.

Variable `radius` is either double $[1 \times 1]$ or a Workspace Variable.

Optional variable `normal` defines orientation of the CircleArc. It has to be char $[1 \times 1]$. Possible values are: `'z'` (default), `'y'` or `'x'`. Object can be rotated arbitrarily using transformation methods - see Transformations

Optional variable `name` is char $[1 \times N]$. Workspace Variable cannot be used to define object's name!

CircleArc and it's properties is depicted in Fig. 2.6. For the full header refer to addCircleArc.

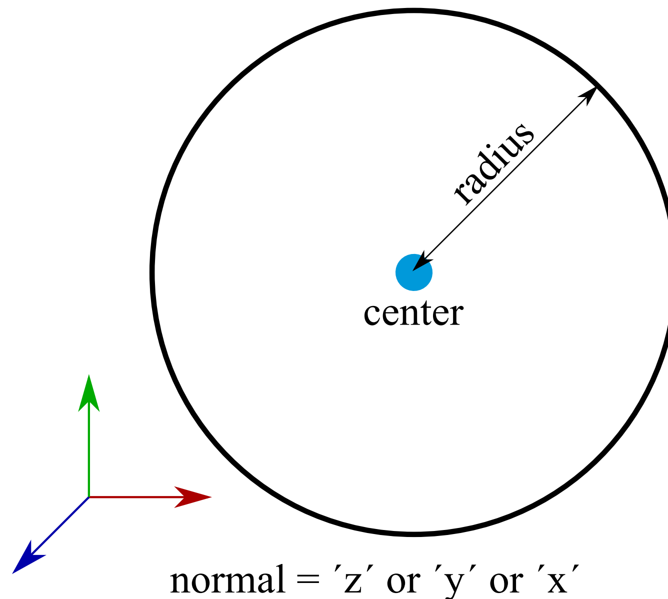


Figure 2.6: CircleArc defined in 3D and its properties.

EllipseArc

EllipseArc is a 1D object consisting of Ellipse's the whole border curve or its part define by angles. All created EllipseArcs can be found in:

```
ellipseArcs = atom.selectedProject.geom.ellipsearc;
```

EllipseArc can be created by a command:

```
atom.selectedProject.geom.addEllipseArc(center, majorVertex, minorVertex, ...
startAngle, angle, name);
```

Variable `center` is either double $[1 \times 3]$ or a Workspace Variable. It specifies a position of ellipse's center point.

Variable `majorVertex` is either double $[1 \times 3]$ or a Workspace Variable. It specifies a position of ellipse's major axis vertex. It defines orientation of ellipse in 3D in combination with `center` and `mainorVertex`.

Variable `minorVertex` is either double $[1 \times 3]$ or a Workspace Variable. It specifies a position of ellipse's minor axis vertex. It defines orientation of ellipse in 3D in combination with `center` and `majorVertex`.

Variable `startAngle` is either double $[1 \times 1]$ or a Workspace Variable. It specifies the angular start of an ellipse arc in radians.

Variable `angle` is either double $[1 \times 1]$ or a Workspace Variable. It specifies the angular length of an ellipse arc in radians.

Optional variable `name` is char $[1 \times N]$. Workspace Variable cannot be used to define object's name!

EllipseArc and it's properties is depicted in Fig. 2.7. For the full header refer to addEllipseArc.

EquationCurve

EquationCurve is a 1D object of arbitrary curve defined by three handle functions. All created EquationCurves can be found in:

```
equationCurves = atom.selectedProject.geom.equationcurve;
```

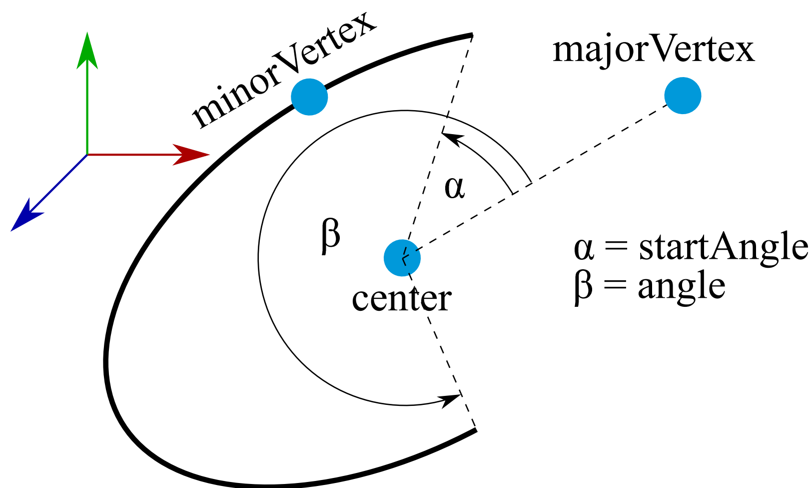


Figure 2.7: EllipseArc defined in 3D and its properties.

EquationCurve can be created by a command:

```
atom.selectedProject.geom.addEquationCurve(equationX, equationY, equationZ, ...
paramInterval, name);
```

Variable `equationX` is either char $[1 \times N]$ or a Workspace Variable. It contains Matlab definition of handle function defining equation for x-axis.

Variable `equationY` is either char $[1 \times N]$ or a Workspace Variable. It contains Matlab definition of handle function defining equation for y-axis.

Variable `equationZ` is either char $[1 \times N]$ or a Workspace Variable. It contains Matlab definition of handle function defining equation for z-axis.

Variable `paramInterval` is either double $[1 \times 2]$ or a Workspace Variable. It specifies the parameter start and end value used for handle functions.

Optional variable `name` is char $[1 \times N]$. Workspace Variable cannot be used to define object's name!

EquationCurve and it's properties is depicted in Fig. 2.8. For the full header refer to addEquationCurve.

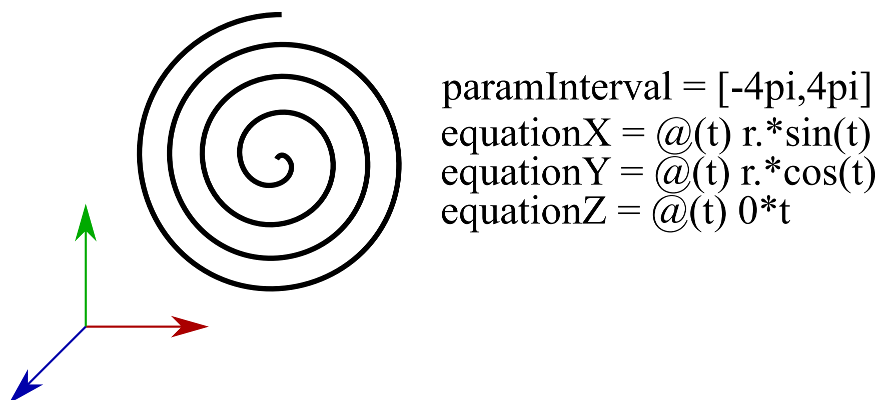


Figure 2.8: EquationCurve defined in 3D and its properties.

Rectangle

Rectangle is a 2D object. As rectangle is degenerated form of Parallelogram, all Rectangles can be found in:

```
parallelograms = atom.selectedProject.geom.parallelogram;
```

Rectangle can be created by a command:

```
atom.selectedProject.geom.addRectangle(center, width, height, normal, name);
```

Variable `center` is either double $[1 \times 3]$ or a Workspace Variable. It specifies a position of rectangle's center.

Variable `width` is either double $[1 \times 1]$ or a Workspace Variable.

Variable `height` is either double $[1 \times 1]$ or a Workspace Variable.

Optional variable `normal` defines orientation of the Rectangl. It has to be char $[1 \times 1]$. Possible values are: `'z'` (default), `'y'` or `'x'`. Object can be rotated arbitrarily using transformation methods - see Transformations

Optional variable `name` is char $[1 \times N]$. Workspace Variable cannot be used to define object's name!

Rectangle and it's properties is depicted in Fig. 2.9. For the full header refer to addRectangle.

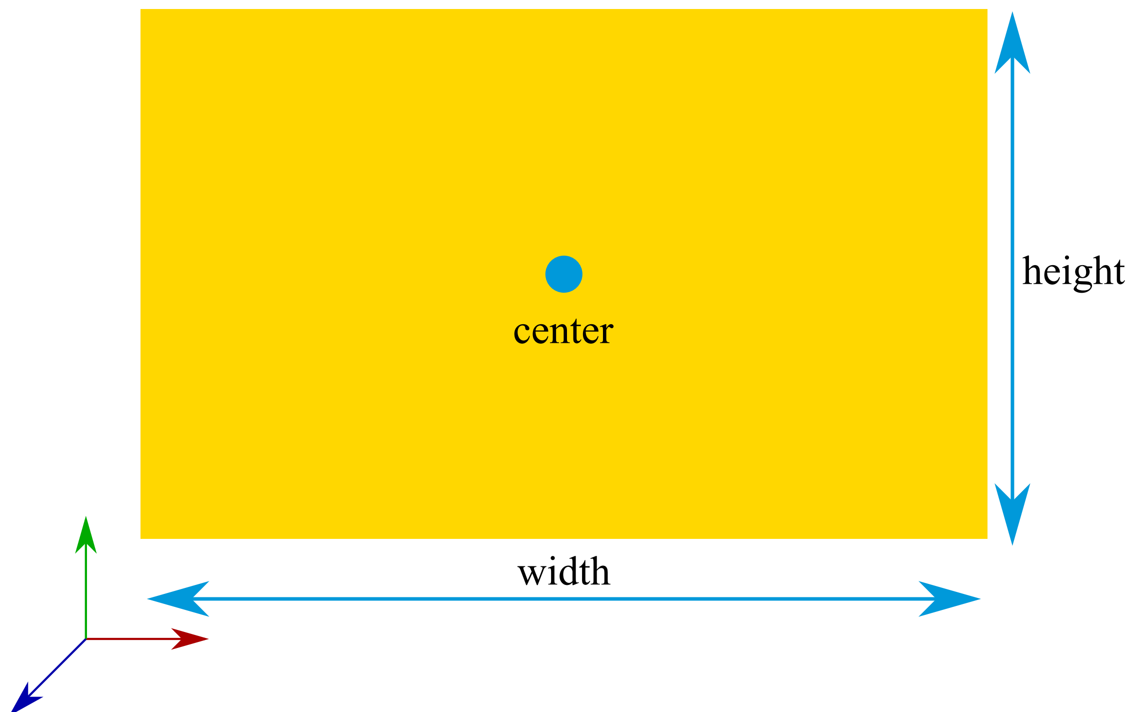


Figure 2.9: Rectangle defined in 3D and its properties.

Parallelogram

Parallelogram is a 2D object. All created Parallelograms can be found in:

```
parallelograms = atom.selectedProject.geom.parallelogram;
```

Parallelogram can be created by a command:

```
atom.selectedProject.geom.addParallelogram(lowLeftCorner, lowRightCorner, ...
highLeftCorner, name);
```

Variable `lowLeftCorner` is either double $[1 \times 3]$ or a Workspace Variable. It specifies a position of parallelogram's low-left corner, viewed from side where the object's normal points to.

Variable `lowRightCorner` is either double $[1 \times 3]$ or a Workspace Variable. It specifies a position of parallelogram's low-right corner.

Variable `highLeftCorner` is either double $[1 \times 1]$ or a Workspace Variable. It specifies a position of parallelogram's high-left corner.

Optional variable `name` is char $[1 \times N]$. Workspace Variable cannot be used to define object's name!

Parallelogram and it's properties is depicted in Fig. 2.10. For the full header refer to addParallelogram.

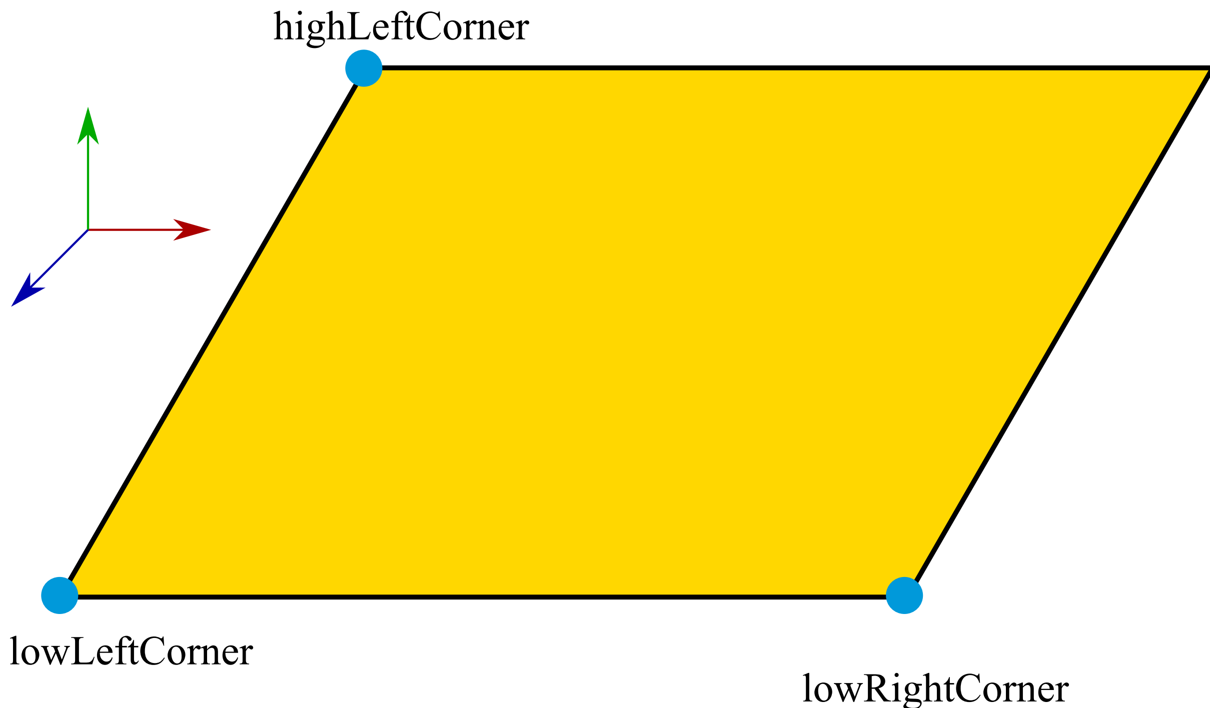


Figure 2.10: Parallelogram defined in 3D and its properties.

Circle

Circle is a 2D object. As circle is degenerated form of ellipse, all Circles can be found in:

```
ellipses = atom.selectedProject.geom.ellipse;
```

CircleArc can be created by a command:

```
atom.selectedProject.geom.addCircle(center, radius, normal, name);
```

Variable `center` is either double $[1 \times 3]$ or a Workspace Variable. It specifies a position of circle's center.

Variable `radius` is either double $[1 \times 1]$ or a Workspace Variable.

Optional variable `normal` defines orientation of the Circle. It has to be char $[1 \times 1]$. Possible values are: `'z'` (default), `'y'` or `'x'`. Object can be rotated arbitrarily using transformation methods - see Transformations

Optional variable `name` is char $[1 \times N]$. Workspace Variable cannot be used to define object's name!

Circle and it's properties is depicted in Fig. 2.11. For the full header refer to addCircle.

Ellipse

Ellipse is a 2D object. It can be either full ellipse or ellipse sector defined by angles. All created Ellipses can be found in:

```
ellipses = atom.selectedProject.geom.ellipse;
```

Ellipse can be created by a command:

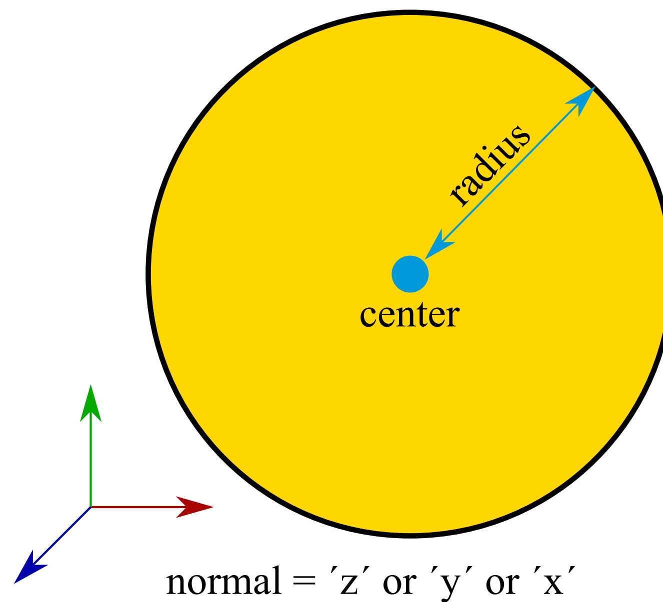


Figure 2.11: Circle defined in 3D and its properties.

```
atom.selectedProject.geom.addEllipse(center, majorVertex, minorVertex, startAngle, ...
angle, name);
```

Variable `center` is either double $[1 \times 3]$ or a Workspace Variable. It specifies a position of ellipse's center point.

Variable `majorVertex` is either double $[1 \times 3]$ or a Workspace Variable. It specifies a position of ellipse's major axis vertex. It defines orientation of ellipse in 3D in combination with `center` and `mainorVertex`.

Variable `minorVertex` is either double $[1 \times 3]$ or a Workspace Variable. It specifies a position of ellipse's minor axis vertex. It defines orientation of ellipse in 3D in combination with `center` and `majorVertex`.

Variable `startAngle` is either double $[1 \times 1]$ or a Workspace Variable. It specifies the angular start of an ellipse in radians.

Variable `angle` is either double $[1 \times 1]$ or a Workspace Variable. It specifies the angular length of an ellipse in radians.

Optional variable `name` is char $[1 \times N]$. Workspace Variable cannot be used to define object's name!

Ellipse and it's properties is depicted in Fig. 2.12. For the full header refer to addEllipse.

Polygon

Polygon is a 2D object. It can be created by a command:

```
atom.selectedProject.geom.addPolygon(points, name);
```

Variable `points` is either double $[N \times 3]$ or a Workspace Variable. If first and last point from `points` are different, first point is added to the end of `points`.

Optional variable `name` is char $[1 \times N]$. Workspace Variable cannot be used to define object's name!

Polygon and it's properties is depicted in Fig. 2.13. For the full header refer to addPolygon.

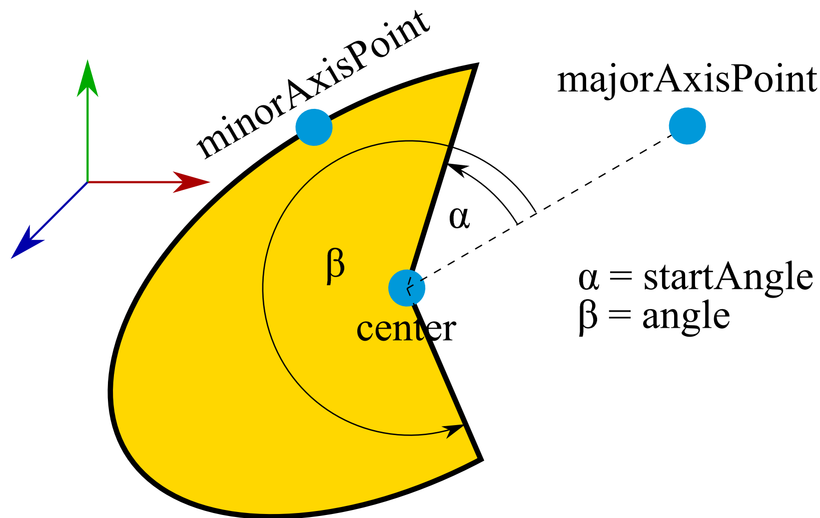


Figure 2.12: Ellipse defined in 3D and its properties.

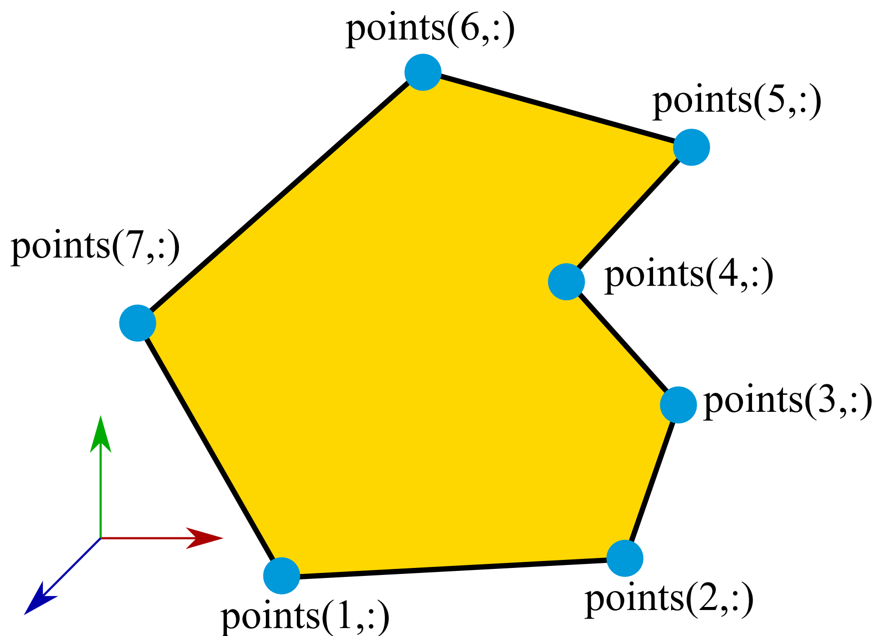


Figure 2.13: Polygon defined in 3D and its properties.

2.3 Transformations

Rotate

User can rotate arbitrary object using command:

```
atom.selectedProject.geom.rotateObject(name, vector, angle, type)
```

Variable `name` is char $[1 \times N]$ identifying object to be rotated.

Variable `vector` is double $[1 \text{ or } 2 \times N]$ or a Workspace Variable. If size of `vector` is $[1 \times N]$, rotation axis is directly `vector` (line going through origin and point specified in `vector`). If size of `vector` is $[2 \times N]$, rotation axis is a line going through two points specified in `vector(1,:)` and `vector(2,:)`.

Variable `angle` is either double $[1 \times 1]$ or a Workspace Variable. It specifies the angle of rotation in radians. Angle can be from interval $\langle 0; 2\pi \rangle$.

Optional variable `type` is object of enumeration `models.geom.GeomObjectType`. Specification of `type` can speed up the search of object. Possible types are listed in GeomObjectType.

RotateX

User can rotate arbitrary object around x-axis using command:

```
atom.selectedProject.geom.rotateXObject(name, angle, type)
```

Variable `name` is char $[1 \times N]$ identifying object to be rotated.

Variable `angle` is either double $[1 \times 1]$ or a Workspace Variable. It specifies the angle of rotation in radians. Angle can be from interval $\langle 0; 2\pi \rangle$.

Optional variable `type` is object of enumeration `models.geom.GeomObjectType`. Specification of `type` can speed up the search of object. Possible types are listed in GeomObjectType.

RotateY

User can rotate arbitrary object around y-axis using command:

```
atom.selectedProject.geom.rotateYObject(name, angle, type)
```

Variable `name` is char $[1 \times N]$ identifying object to be rotated.

Variable `angle` is either double $[1 \times 1]$ or a Workspace Variable. It specifies the angle of rotation in radians. Angle can be from interval $\langle 0; 2\pi \rangle$.

Optional variable `type` is object of enumeration `models.geom.GeomObjectType`. Specification of `type` can speed up the search of object. Possible types are listed in GeomObjectType.

RotateZ

User can rotate arbitrary object around z-axis using command:

```
atom.selectedProject.geom.rotateZObject(name, angle, type)
```

Variable `name` is char $[1 \times N]$ identifying object to be rotated.

Variable `angle` is either double $[1 \times 1]$ or a Workspace Variable. It specifies the angle of rotation in radians. Angle can be from interval $\langle 0; 2\pi \rangle$.

Optional variable `type` is object of enumeration `models.geom.GeomObjectType`. Specification of `type` can speed up the search of object. Possible types are listed in GeomObjectType.

Scale

User can scale arbitrary object in different directions using command:

```
atom.selectedProject.geom.scaleObject(name, vector, type)
```

Variable `name` is char $[1 \times N]$ identifying object to be rotated.

Variable `vector` is double $[1 \times N]$ or a Workspace Variable. Individual entries of `vector` specify scale in x, y or z axis.

Optional variable `type` is object of enumeration `models.geom.GeomObjectType`. Specification of `type` can speed up the search of object. Possible types are listed in GeomObjectType.

Translate

User can translate arbitrary object in a direction specified by vector using command: `atom.selectedProject.`

```
vector, type)
```

Variable `name` is char $[1 \times N]$ identifying object to be rotated.

Variable `vector` is double $[1 \times N]$ or a Workspace Variable. Individual entries of `vector` specify translation in x, y or z axis.

Optional variable `type` is object of enumeration `models.geom.GeomObjectType`. Specification of `type` can speed up the search of object. Possible types are listed in GeomObjectType.

2.4 Boolean operations

Boolean operations enable user to build more complex models from individual Geom Primitives. Boolean operations are always performed on a pair of objects of same kind (1D or 2D). First object becomes so called "master" and second one "slave". Master object contains information about the changed geometry, is displayed in Design Viewer and is further meshed. Slave object remains in the Geom class container but is not displayed and meshed. Every object can be used as slave only once! The master/slave bonding can be canceled by deleting the command specifying the Boolean operation from object's history - see 2.5.

Unite

Boolean operation `unite` makes union of two objects. It is performed using a command:

```
atom.selectedProject.geom.boolean.unite(name1, name2)
```

Variables `name1` and `name2` are names of objects to be united. Example union is displayed in Fig. 2.14. For the full header refer to unite.

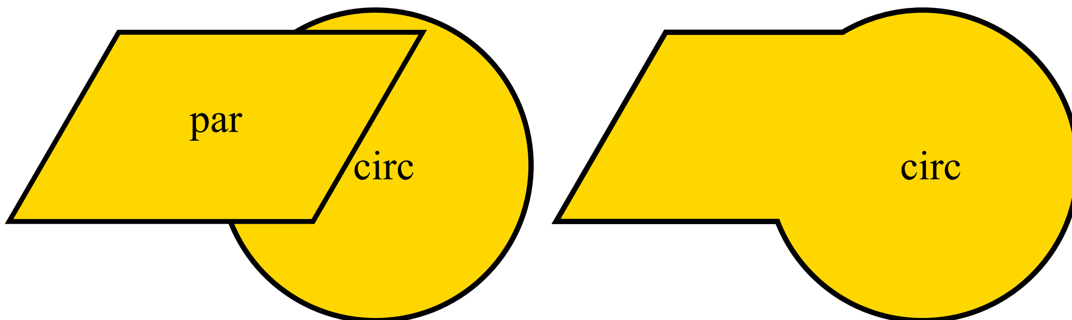


Figure 2.14: Unite method performed on two 2D objects: `'circ'` + `'par'`.

Subtract

Boolean operation `subtract` makes subtraction of two objects. It is performed using a command:

```
atom.selectedProject.geom.boolean.subtract(name1, name2)
```

Variables `name1` and `name2` are names of objects to be subtracted. Example subtraction is displayed in Fig. 2.15. For the full header refer to subtract.

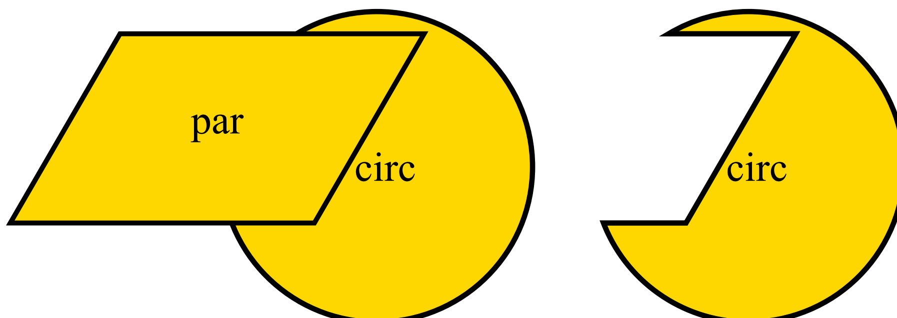


Figure 2.15: Subtract method performed on two 2D objects: `'circ'` + `'par'`.

2.5 Common methods

Delete Object

Every object can be deleted from the current Project using a command:

```
atom.selectedProject.geom.deleteObject(name, type)
```

Variable `name` is char $[1 \times N]$ identifying object to be deleted.

Optional variable `type` is object of enumeration `models.geom.GeomObjectType`. Specification of `type` can speed up the search of object. Possible types are listed in GeomObjectType.

If a slave object is deleted its master object is directly recomputed with mo influence of the deleted slave object.

If a master object is deleted, all of it's slave objects are deleted.

Rename

As name of object is it's only identification. user can specify it's name when the object is created. Also, after creation of object, it can be renamed using a command:

```
atom.selectedProject.geom.renameObject(oldName, newName, type)
```

Variable `oldName` is char $[1 \times N]$ identifying object to be renamed.

Variable `newName` is char $[1 \times N]$ specifying new object's name. It has to be unique throughout all the objects in Geom, regardless to object `type`.

Optional variable `type` is object of enumeration `models.geom.GeomObjectType`. Specification of `type` can speed up the search of object. Possible types are listed in GeomObjectType.

Copy Object

All objects defined by user can be copied using a command:

```
atom.selectedProject.geom.copyObject(name, vector, nNew, copyName, type)
```

Variable `name` is char $[1 \times N]$ identifying object to be copied.

Variable `vector` is double $[1 \times 3]$ identifying where copied objects are translated towards the initial position of the object to be copied.

Variable `nNew` is double $[1 \times 1]$ defining number of copies created.

Optional Variable `copyName` is char $[1 \times N]$ specifying new object's name. Number of copy is added to the name of every new object.

Optional variable `type` is object of enumeration `models.geom.GeomObjectType`. Specification of `type` can speed up the search of object. Possible types are listed in GeomObjectType.

Move Command

Every object tracks its own historz of commands. Commands Rotate, RotateX, RotateY, RotateZ, Scale, Translate and Boolean operations Unite and Subtract and their properties are saved in `GeomObject.history.commands`.

User can change the order of commands of selected object using a command:

```
atom.selectedProject.geom.moveCommandObject(name, oldNum, newNum, type)
```

Variable `name` is char $[1 \times N]$ identifying object to be redefined.

Variable `oldNum` is double $[1 \times 1]$ identifying number of command to be moved in history of commands.

Variable `newNum` is double $[1 \times 1]$ identifying new position in history of commands for the specified command.

Optional variable `type` is object of enumeration `models.geom.GeomObjectType`. Specification of `type` can speed up the search of object. Possible types are listed in GeomObjectType.

Remove Command

User can remove any command from history of commands including the Transformations and Transformations:

```
atom.selectedProject.geom.removeCommandObject(name, num, type)
```

Variable `name` is char $[1 \times N]$ identifying object to be redefined.

Variable `num` is double $[1 \times 1]$ identifying number of command that should be removed from history.

Optional variable `type` is object of enumeration `models.geom.GeomObjectType`. Specification of `type` can speed up the search of object. Possible types are listed in GeomObjectType.

Redraw Object

Objects composed of curved (not straight) lines have specified number of draw points - number of samples on every curve. If the number is insufficient from th user's viewpoint, it can be changed:

```
atom.selectedProject.geom.redrawObject(name, nPoints, type)
```

Variable `name` is char $[1 \times N]$ identifying object to be redefined.

Variable `nPoints` is double $[1 \times 1]$. It defines number of points used for discretization of curved lines.

Optional variable `type` is object of enumeration `models.geom.GeomObjectType`. Specification of `type` can speed up the search of object. Possible types are listed in GeomObjectType.

Chapter 3

Mesh

3.1 General

All discretized curves and planes are represented by `MeshObject`, which saves data of a given object – nodes, connectivity and local metadata. `Mesh` exists only in one instance per project and collects data from all `MeshObject` objects – nodes, elements, which are split into 1D edges for curves and 2D edges and connectivity for planar objects.
Metadata for the `Mesh` object are maximal element size, user defined density function and mesh density, which defines the number of elements per wavelength. Element size itself is dependent property[1] based on the value of mesh density in case you have enabled computing element size from frequency. Otherwise mesh size is absolute.
Other properties, even though it is possible to represent them in matrix form, are dependent because it is much easier to compute them dynamically then ensure data validity after each modification of the mesh.
Quality of every triangle is computed using following formula introduced in [?]:

$$q = \frac{4\sqrt{3}a}{h_1^2 + h_2^2 + h_3^2}, \tag{3.1}$$

where a is triangle area and h_i are edge lengths.

3.2 Creating meshes from AToM geometry

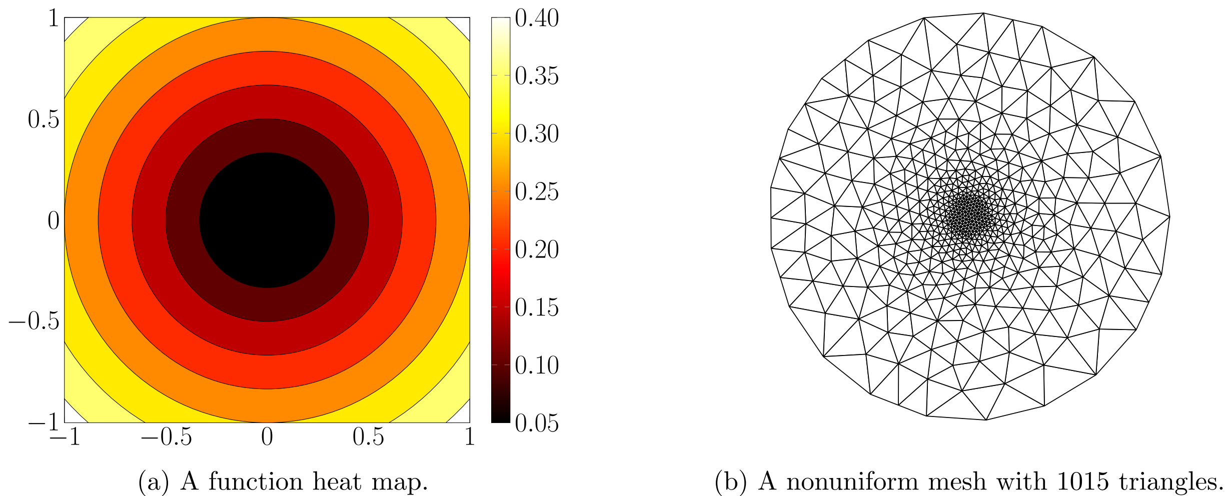
Built-in mesh generator can discretize each geometry into a mesh. A user can modify size of elements, density distribution and type of elements. Density functions and mesh size/number of elements per wavelength can be set either globaly, or you can enable local properties for selected objects.

Listing 3.1: Example of a non-uniform density function in MATLAB.

```
h = @(x, y) max(0.3*sqrt(x.*x + y.*y), 0.05)
```

Each geometry is uniform density function in default, but you can set non-uniform density functions as on the Figure 3.1. AToM supports two types of triangulations. First of them is Delaunay triangulation [?] which is so called standart with some very nice properties. Second type of triangulation is called uniform. All elements in this type have the same shape and size. You can choose from two types – equilateral and right. Note that both uniform triangulations aren't body conforming.
Next useful feature is called mesh on fly. When enabled, whenever you change anything which has direct impact on the mesh computation, mesh is recalculated with up to date settings.

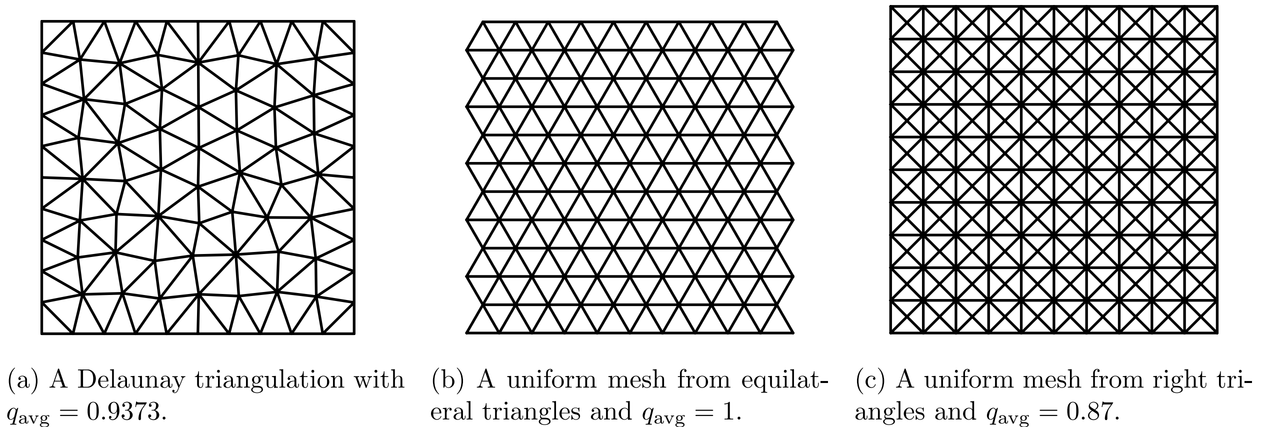
[1] A special type of property in MATLAB, whose value depends on other values and properties and it is computed when the property is queried.



(a) A function heat map.

(b) A nonuniform mesh with 1015 triangles.

Figure 3.1: Example of density function from the Listing 3.1 and its impact on the mesh generation.



(a) A Delaunay triangulation with $q_{\text{avg}} = 0.9373$.

(b) A uniform mesh from equilateral triangles and $q_{\text{avg}} = 1$.

(c) A uniform mesh from right triangles and $q_{\text{avg}} = 0.87$.

Figure 3.2: A discretized square using different types of triangulation.

3.3 Imported and exported meshes

Apart from creating meshes directly from ATоM geometry, you can use your own meshes in various formats.

- NASTRAN
- mphtxt
- GEO
- MATLAB matrices (nodes [N x 3], connectivity list [N x 2/3])

It is also possible to export meshes into nastrat and geo formats directly from ATоM.
Note that imported meshes don't have its ATоM geometry. This means you need to perform all transformations using `MeshObject`.
Unique atom feature is called mesh to polygon. It allows conversion of an imported mesh into standart ATоM geometry object. This geometry object is of type polygon and can be further remeshed and tranformed as other geometry objects.

3.4 Getting data from mesh

All data about mesh are stored in `Mesh` and `MeshObject` objects. You should never access these data directly. Always call appropriate mesh data functions.

Listing 3.2: Retrieving data from mesh.

```
% getting 1D mesh data
data1D = atom.selectedProject.mesh.getMeshData1D();

% getting 2D mesh data
data2D = atom.selectedProject.mesh.getMeshData2D();
```

3.5 Public functions

List of public methods which can be evaluated on Mesh object. Mesh object can be obtained as `objMesh = atom.selectedProject.mesh`

Setting mesh properties

setElementSizeFromFrequency more
: `objMesh.setElementSizeFromFrequency(name, value)`
Enables/disables mesh size from frequency.

setGlobalDensityFunction more
: `objMesh.setGlobalDensityFunction(functionHandle)`
Sets global mesh density function.

setGlobalMeshDensity more
: `objMesh.setGlobalMeshDensity(densityOfElements)`
Sets global mesh density.

setLocalDensityFunction more
: `objMesh.setLocalDensityFunction(name, functionHandle)`
Sets local mesh density function of `MeshObject`.

setLocalMeshDensity more
: `objMesh.setLocalMeshDensity(name, meshDensity)`
Sets local mesh density of `MeshObject`.

setMaxElement more
: `objMesh.setMaxElement(name, size)`
Sets maximal element size.

setQualityPriority more
: `objMesh.setQualityPriority(value)`
Enables/disables quality priority during mesh generation.

setUniformMeshType more
: `objMesh.setUniformMeshType(elemType)`
Sets type of uniform mesh.

setUseLocalMeshDensity more
: `objMesh.setUseLocalMeshDensity(name, value)`
Enables/disables use of local mesh density on a `MeshObject`.

setUseUniformTriangulation more
: `objMesh.setUseUniformTriangulation(value)`
Enables/disables uniform triangulation.

Mesh data

`getBoundaryMesh` more

`objMesh.getBoundaryMesh()`

Computes only 1D boundary mesh of 2D geometry.

`getCircumsphere` more

`[coordinates] = objMesh.getCircumsphere()`

Computes mesh circumsphere.

`getEdges` more

`[edges1D, edges2D] = objMesh.getEdges()`

Returns edges in mesh.

`getMesh` more

`objMesh.getMesh()`

Recomputes the whole mesh.

`getMeshData1D` more

`objMesh.getMeshData1D()`

Computes data necessary for 1D MoM computation.

`getMeshData2D` more

`objMesh.getMeshData2D()`

Computes data necessary for 2D MoM computation.

`getMeshStatistics` more

`objMesh.getMeshStatistics()`

Returns statisctics of computed mesh.

Mesh modification

`convertToImportedMesh` more

`objMesh.convertToImportedMesh(name)`

Converts mesh from AToM geometry object into `MeshObject` which behaves as imported `MeshObject`.

`deleteEdges1D` more

`objMesh.deleteEdges1D(edgesToDelete)`

Deletes selected edges from 1D Mesh.

`deleteMesh` more

`objMesh.deleteMesh(name)`

Deletes selected `MeshObject`.

`deleteNodes` more

`objMesh.deleteNodes(nodesToDelete)`

Deletes selected nodes.

`deleteTriangles` more

`objMesh.deleteTriangles(trianglesToDelete)`

Deletes selected triangles from mesh.

`meshToPolygon` more

`objMesh.meshToPolygon(name)`

Converts mesh into AToM geometry polygon.

Import/Export

`exportMesh` more

`objMesh.exportMesh(type, name, filePath)`

Exports mesh into a file in the selected format.

`import1DMesh` more

`objMesh.import1DMesh(nodes, connectivityList, name)`

Imports 1D mesh from MATLAB matrix.

`import2DMesh` more

`objMesh.import2DMesh(nodes, connectivityList, name)`

Imports 2D mesh from MATLAB matrix.

`importMesh` more

`objMesh.importMesh(fileName, name)`

Imports mesh from a file in the selected format.

`mirrorImportedMesh` more

`objMesh.mirrorImportedMesh(name, normal, numCopies, origin)`

Mirrors imported `MeshObject`.

`renameImportedMesh` more

`objMesh.renameImportedMesh(currentName, newName)`

Renames imported `MeshObject`.

`rotateImportedMesh` more

`objMesh.rotateImportedMesh(name, rotateAngles, numCopies, origin)`

Rotates imported `MeshObject`.

`scaleImportedMesh` more

`objMesh.scaleImportedMesh(name, scaleVector, numCopies, origin)`

Scales imported `MeshObject`.

`translateImportedMesh` more

`objMesh.translateImportedMesh(name, translateVector, numCopies)`

Translates imported `MeshObject`.

Visualization

`plotMesh` more

`objMesh.plotMesh()`

Plots the whole mesh from `Mesh` object

`plotMeshBoundary` more

`objMesh.plotMeshBoundary()`

Plots mesh with highlighted boundaries.

`plotMeshCircumsphere` more

`objMesh.plotMeshCircumsphere()`

Plots mesh altogether with its enclosing circumsphere.

Chapter 4

Physics

Physics class enables to define general properties of simulation task like frequency points, symmetries and materials for Boundary element method simulation. Every AToM project has its own Physics model and it is possible to access it in selected project via Command Window by `atom.selectedProject.physics`.

4.1 Frequency List

Frequency list can be defined as variable in Workspace. Value of this property can be numerical vector (row or column) with unique real values. Simulation of defined structure is performed on frequency points stated herein.

4.2 Symmetry Planes

In AToM is possible to define properties of three symmetry planes: XY, YZ and XZ. When some symmetry plane is active, it is possible to define just appropriate part of geometry, or whole geometry (example show on Fig. 4.1). Before meshing the structure, only appropriate part of geometry (half-space with one positive coordinate, quadrant with two positive coordinates, ...) is meshed and used for computations. Using symmetries thus saves computational time. Symmetries are implemented using so-called C-matrices which are introduced in Section 8.3. PEC, PMC and geometric planes are only special cases of the C-matrix. Possible types of planes are:

- `'none'` - symmetry plane is not applied
- `'electric'` - PEC wall ($E_{\mathrm{t}} = 0$)
- `'magnetic'` - PMC wall ($H_{\mathrm{t}} = 0$)
- `'geometric'` - geometric symmetry

4.3 Materials

Materials model manage predefined and user defined materials for Boundary element method simulation (Section 5.1). Only non dispersive dielectrics and metals can be specified. Materials are generally defined by loss tangent and conductivity. Dielectrics and metals are also specified by permittivity and permeability, respectively.

4.4 Public Methods

`getFrequencyListValues`

```
f = physics.getFrequencyListValues()
```

Return actual frequency list values from Physics model. If the frequency list is invalid, error

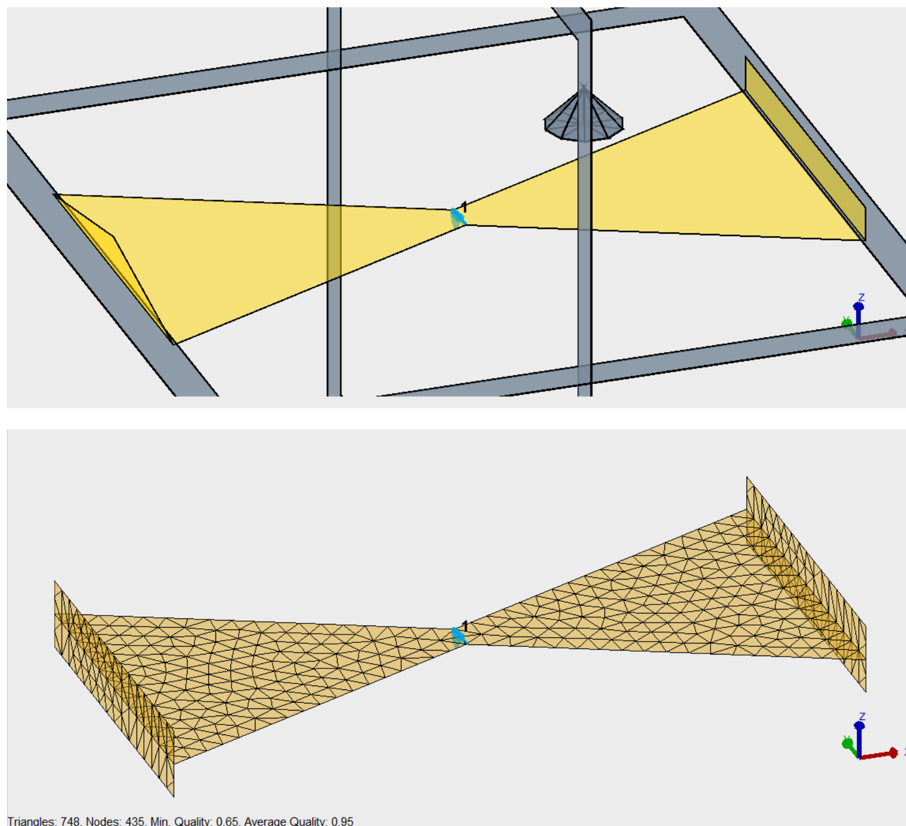


Figure 4.1: Structure of modified bowtie antenna with two geometry symmetry planes. Main bowtie shape is drawn entirely, but an ending sheet is drawn on both sides of antenna differently. In the resulting mesh can be seen (symmetry planes are not shown), that mesh is symmetrized according to XY and YZ symmetry planes and was computed from geometry in quadrant with positive X and Z coordinate.

is thrown.

`setFrequencyList`

```
physics.setFrequencyList(fListExpression)
```

Sets frequency list. It is possible to use arbitrary expression as char (*e.g.*, `'linspace(0.1, ... 1, 10)*1e9'`, `'(1:10)*1e8'`), or directly numerical vector. Valid value is vector of positive finite real and unique numbers.

`symmetryPlanes.(plane).setType`

```
physics.symmetryPlanes.XY.setType(symmType); physics.symmetryPlanes.YZ.setType(symmType); ...
physics.symmetryPlanes.XZ.setType(symmType)
```

Sets type of specific plane. `symmetryType` can be `'none'`, `'electric'`, `'magnetic'` or `'geometric'`.

`materials.addDielectric`

```
physics.materials.addDielectric(nameOfMaterial, varargin)
```

This method add dielectric to material container. First mandatory argument has to be an unique name of material as char vector. If no more parameter are passed, default material with vacuum properties is defined ($\epsilon_r = 1$, $\tan\delta = 0$, $\sigma = 0$). All other optional arguments are name-value pairs, *e.g.*, `addDielectric('myMaterial', 'Permittivity', 2.1, ... 'LossTangent', 'tgD', 'Conductivity', '0')`. All parameter values can be defined as workspace variables (as `'tgD'` in previous example). If some parameter is not defined, value from default vacuum is used.

`materials.addMetal`

`physics.materials.addMetal(nameOfMaterial, varargin)`
This method add metal to material container. First mandatory argument has to be an unique name of material as char vector. If no more parameter are passed, default material with PEC properties is defined ($\mu_r = 1$, $\tan\delta = 0$, $\sigma = \infty$). All other optional arguments are name-value pairs, *e.g.*, `addMetal('myMaterial', 'Permeability', 2, 'LossTangent', ... '0', 'Conductivity', 'sigmaMetal')`. All parameter values can be defined as workspace variables (as `'sigmaMetal'` in previous example). If some parameter is not defined, value from default PEC is used.

materials.editMedium

`physics.materials.editMedium(nameOfMaterial, varargin)`
This method allows editing of arbitrary material in `physics.material` container. First mandatory argument has in an unique name of material as char vector. All other optional arguments are name-value pairs, *e.g.*, `physics.editMedum('myMetal', 'Conductivity', ... 'sigmaMetal', 'Permeability', '1.1')`. All parameter values can be defined as workspace variables (as `'sigmaMetal'` in previous example).

materials.getDielectricTable

`dielTab = physics.materials.getDielectricTable()`
This method returns variable of table class with list of dielectric materials in `physics.material` container.

materials.getMetalTable

`metalTab = physics.materials.getMetalTable()`
This method returns variable of table class with list of metal materials in `physics.material` container.

materials.getTable

`materialTab = physics.materials.getTable()`
This method returns variable of table class with list of all materials in `physics.material` container.

materials.removeMedium

`physics.materials.removeMedium(materialName)`
This method deletes arbitrary material from `physics.material` container. `materialName` is name of material as char vector.

4.5 Public Properties

c0

Return speed of light in vacuum used in AToM for computations. This property is taken during project creation from public function `models.utilities.constants.c0`

ep0

Return permittivity of free space used in AToM for computations. This property is taken during project creation from public function `models.utilities.constants.ep0`.

gamma

Return Euler–Mascheroni constant used in AToM for computations. This property is taken during project creation from public function `models.utilities.constants.gamma`.

mu0

Return vacuum permeability used in AToM for computations. This property is taken during project creation from public function `models.utilities.constants.mu0`.

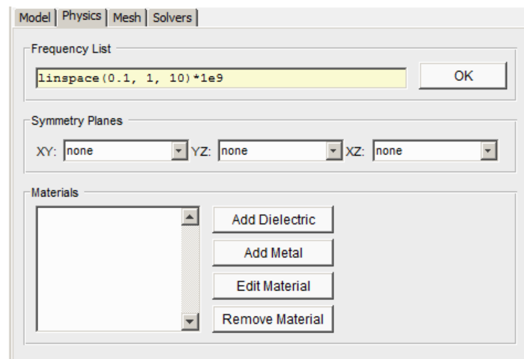


Figure 4.2: Physics Viewer.

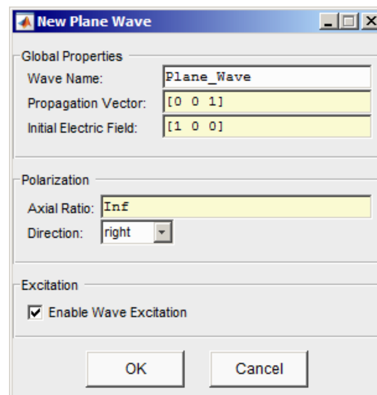


Figure 4.3: Dialogue for plane wave definition.

`symmetryPlanes.(plane).type`

> Return actual type of specific symmetry plane `plane`, *e.g.*,`planeType = ... physics.symmetryPlanes.XY.type`.

4.6 Physics Viewer

Physics Viewer (Fig. 4.2) is simple GUI, which provides interface for majority of Physics model methods mentioned above. Physics Viewer is part of Main Viewer and is accessible via Physics tab.

4.7 Feeding

Feeding is model aggregating and managing plane waves and feeders on geometry and mesh. Feeding is actually property of **`Physics`** model and is accessible by `atom.selectedProject.physics.feeding`.

4.7.1 Plane Waves

In AToM it is possible to define arbitrary number of plane waves, which are irradiating the structure. Definition of plane waves is possible via dialogue shown in Fig. 4.3. This dialogue is accessible via `Design Viewer`, or by right click on `Plane Waves` under `Feeding` item from tree in `Main ... Viewer`. For physical meaning of plane waves please see Section 5.3. Plane waves are defined by these properties:

- `name` - unique identifier of the wave. Name has to be `char` vector and fulfil MATLAB function `isvarname`.

- `propagationVector` - cartesian vector defining direction of propagation of plane wave. This vector can be defined via AToM **Workspace** as expression of variable, or directly like numerical vector of size $[1 \times 3]$. Default value is `[1, 0, 0]`.
- `initElectricField` - vector defining direction of electric field in coordinate origin at time $t = 0$. It is possible to define this vector as expression of variable in **Workspace**, or directly as numerical vector of size $[1 \times 3]$. `propagationVector` and `initElectricField` has to be perpendicular to each other. `initElectricField` can be complex and represents a phasor. By that it is possible to create plane waves with defined phase shift. In a GUI is shown a real component of the electric field. Default value is `[0, 0, 1]`.
- `axialRatio` - axial ratio of plane wave. It is possible to define it as expression of variable in `Workspace`, or as numerical scalar. Value `Inf` stands for linearly polarized wave, and value `1` stands for circularly polarized wave. Values `1 < axialRatio < Inf` represents elliptically polarized wave. Axial ratio is defined as ratio of magnitude of orthogonal components of electric filed. Generally for elliptically polarized wave it is ratio of magnitude of electric field in direction of major axis and minor axis. It is assumed that `initElectricField` represents direction of major axis. Default value is `Inf`.
- `direction` - direction of elliptically polarized wave. Possible values are `'left'` and `'right'` for left-handed polarization and right-handed polarization, respectively. Default value is `'right'`.
- `isEnabled` - allow to turn wave excitation on and off. Property can be defined as expression in `Workspace`, or directly as `logical` scalar. Default value is `true`.

4.7.2 Geometry Discrete Ports

It is possible to define discrete ports directly on 1D or 2D geometry objects before mesh is created. In case of 1D geometry discrete port is actually single points, which lie on arbitrary geometry object. Position of the port has to be defined as coordinate of single point lying directly on the so-called geometry base object. Coordinates of port can be defined in global coordinate system (numerical vector $[1 \times 3]$) or by parametrized position (see 2 for details). In case of discrete port on 2D geometry objects, port has always shape of line and is defined by global (numerical matrix $[2 \times 3]$) or parametrized position (numerical matrix $[2 \times 2]$) of start and end point. In all cases, port has to lie inside geometry object and no part of port can lie outside of the object (*e.g.*, over a hole in object, between two objects in vacuum, etc.). There are also some logical restrictions, where port can be located. General rule is, that always has to be clear, what is a direction of applied delta gap voltage, or more precisely, which part of structure is positive, and which is negative. It means that port can not lie on the edge of structure (in case when is not part of symmetry plane), or on the other hand, on intersection of more than two planar objects. And of course, ports can not intersect each other. See Fig. 4.5 and Fig. 4.6 for port positions examples in AToM Design Viewer. Example of correct position of port on geometry with symmetry planes is hown in Fig. 4.1. Port can lie inside symmetry planes and outside of planes in halfspace or quadrant with positive coordinates. But cutting the port by symmetry plane is forbidden.

When port is defined on the geometry structure, it does not automatically means that the port will feed the structure. Ports in AToM is meant to be just place, where is created fixed point or fixed edge, respectively, in computed mesh. To turn on feeding (delta gap feeder) or lumped circuit on created port requires additional steps. Dialogue for creating discrete port on 2D geometry is show in Fig. 4.4. Dialogue for adding port on 1D geometry is similar.

Discrete 1D and 2D ports on geometry has these properties:

- `number` - unique identifier of port. Can be integer scalar only.
- `baseObjName` - name of geometry base object, where is port connected. It has to be `char` vector with name of existing geometry object.

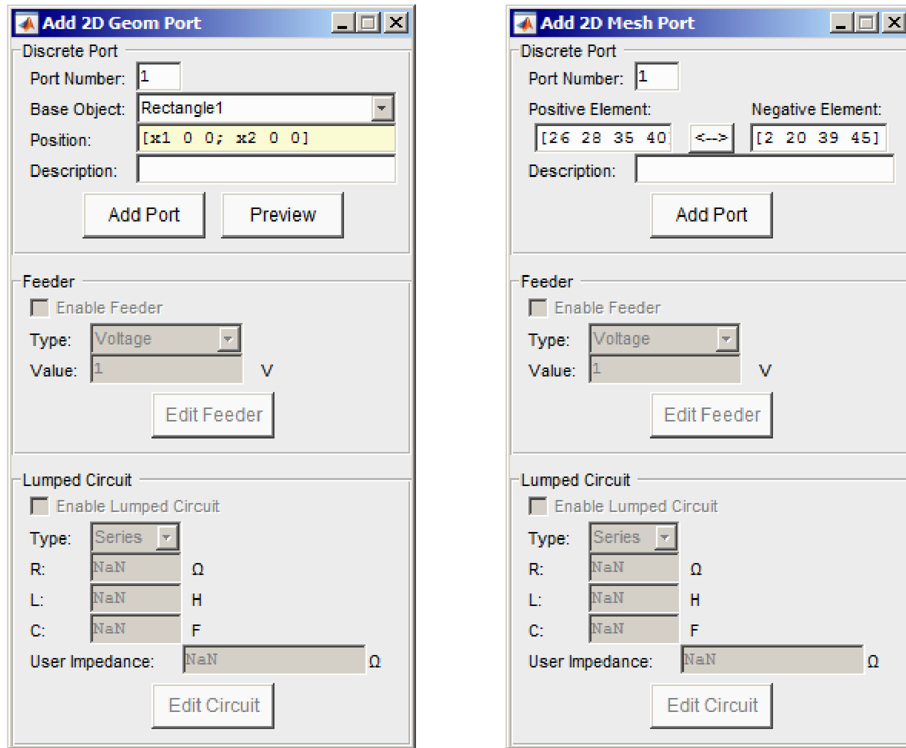


Figure 4.4: Dialogues for adding discrete port on 2D geometry (left) and triangular mesh (right). Dialogues are accessible by icon in Design Viewer.

- `positionPar` - position of port. It is global or parametrized position of single points in case od 1D port (numerical vector $[1 \times 3]$, or scalar from open interval $(0, 1)$), or positions of two points in case of 2D port (numerical matrix $[2 \times 3]$, or $[2 \times 2]$, respectively). Position can be setted as expression of variable in `Workspace` and during creating the port its position in relation with base object is checked. Correct position is also checked during changing port position or base object size/position via `Workspace`.
- `description` - description of port as `char` vector.

There also exists special type of port exclusively intended for BEM (Boundary Element Method, chapter 5.1). This port is possible to define only on planar 2D structures, which lies in XY plane only. BEM port has the same properties as 1D and 2D ports mentioned above, and moreover has property `radius`, which has to be real positive finite scalar (default value is `1e-4` m).

4.7.3 Mesh Ports

Definition of ports directly on port mesh elements is usually impractical, because you can not simply define position edges inside meshed structure. But in case of imported mesh, definition of ports on mesh is the only option. Dialogue for definition of mesh port on triangular mesh is show in Fig. 4.7 and resulting port is show in Fig. 4.4. Ports defined in 1D or 2D mesh has these properties:

- `number` - unique identifier of port. Can be integer scalar only.
- `elements` - numbers of elements, between which is port defined. Numbers of elements are numbers of mesh triangles in case od 2D mesh, or numbers of mesh edges in case of 1D mesh, respectively. It is defined as numerical vector of integers with size $[1 \times 2]$ for 1D mesh, where the first and second number is number of positive and negative edge, respectively. In case of 2D mesh, `elements` can be matrix of size $[N \times 2]$ where in the first and second column are numbers of positive and negative elements, respectively.
- `description` - description of port as `char` vector.

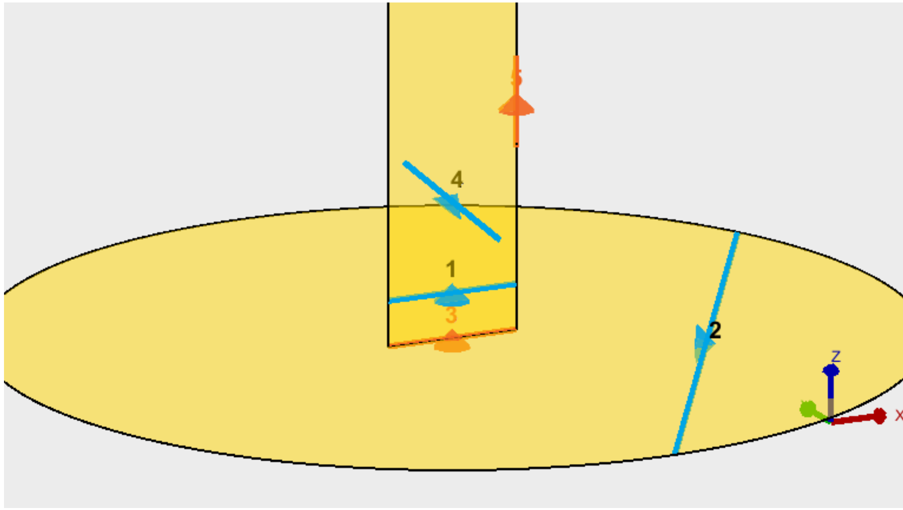


Figure 4.5: Example of correct and incorrect discrete 2D port positions. Port n. 3 is incorrect because it is not clear between which triangles in mesh current should flow. And port n. 5 is on the edge of geometry, where also current can not flow. On the other hand, ports n. 1, 2 and 4 has always simple planar part of geometry on both sides, *i.e.*, direction of current it is clear.

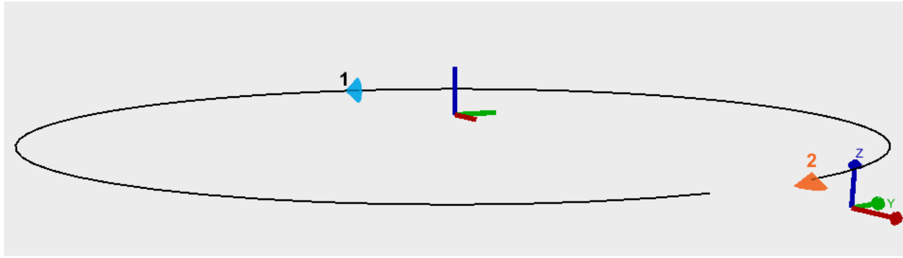


Figure 4.6: Example of correct and incorrect discrete 1D port positions. Port n. 1 is correctly placed on loop antenna, but port n. 2 is placed on the end of loop, which is invalid.

4.7.4 Delta Gap

Integral part of every discrete port is delta gap feeder. Delta gap represents voltage or current source between specific pairs of triangles in case of 2D geometry (mesh), or between two edges in case of 1D geometry (mesh). Delta gap is not active by default and it is necessary to turn it on when needed. Settings of delta gap is possible via the same dialogue, as adding port, and after hit “Add Port” button settings are accessible (Fig. 4.8). For physical meaning of gelta gaps please see Section 5.3.

Delta gap feeder has these properties:

- `isEnabled` - sets if delta gap on port is used for feeding, or not. Value can be directly logical scalar, or `char` vector as Workspace expression.
- `type` - type of delta gap. Possibilities are `'voltage'` and `'current'`.
- `value` - actual value of voltage or current source. Value can be directly numeric (finite complex scalar), or `char` vector as expression from Workspace.

4.7.5 Lumped Circuits

Integral part of every discrete port is simple lumped circuit. This circuit is always in series with delta gap and consists of lumped R, L, C and arbitrary user impedance components Zu (see Fig. 4.9). Lumped circuit itself consists of all components in series, or in parallel circuit. It is possible to set discrete port in a way, when delta gap is not active and port represents a lumped circuit only. For physical meaning of lumped circuits please see Section 5.3. Lumped circuit has these properties:

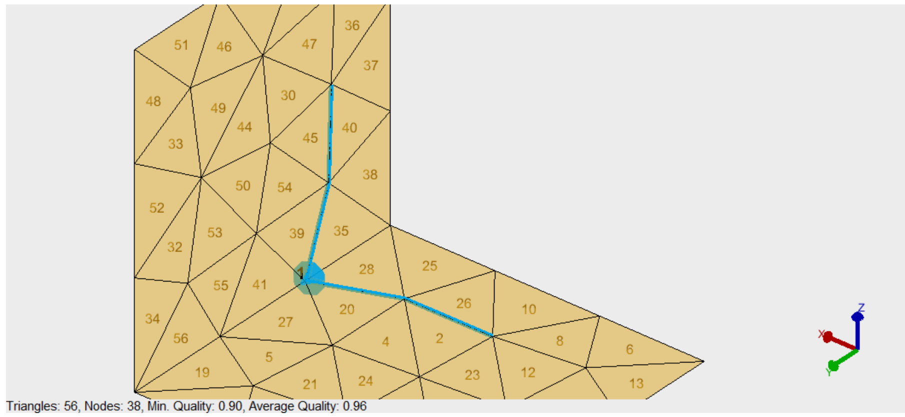


Figure 4.7: Example of definition mesh port on 2D mesh. Port n. 1 has `elements = [26 ... 2;28 20;35 39;40 45]`. In the first column are numbers of triangles with positive voltage.

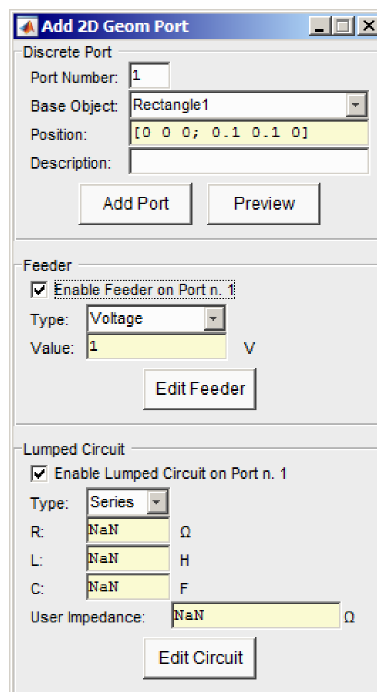


Figure 4.8: Dialogue after adding port enables enabling and editing delta gap and lumped circuit on port.

- `isEnabled` - sets if lumped circuit on port is considered, or not. Value can be directly logical scalar, or `char` vector as WORKSPACE expression.
- `type` - type of lumped circuit. Possibilities are `'ser'`, or `'par'` for serial and parallel arrangement, respectively (see Fig. 4.9).
- `R`, `L`, `C` - value of resistor in Ω, inductance in H and capacitance in F. Value equal to `NaN` means that lumped component is disconnected from circuit and is not take into account.
- `userImpedance` - impedance of user impedance component Zu. This property can be numerical matrix of size $[N \times 2]$, or `char` vector as expression from Workspace. Size N has to be the same as length of frequency list (Section 4.1) in `Physics`. The first column represents impedance of user impedance in Ω, the second column represents derivative of user impedance with respect to angular frequency. Derivative of impedance is used only in cases when derivative of impedance matrix is requested from MoM solver. In other cases, fill it, *e.g.*, with zeros. `NaN` value means disconnection of user impedance from circuit.

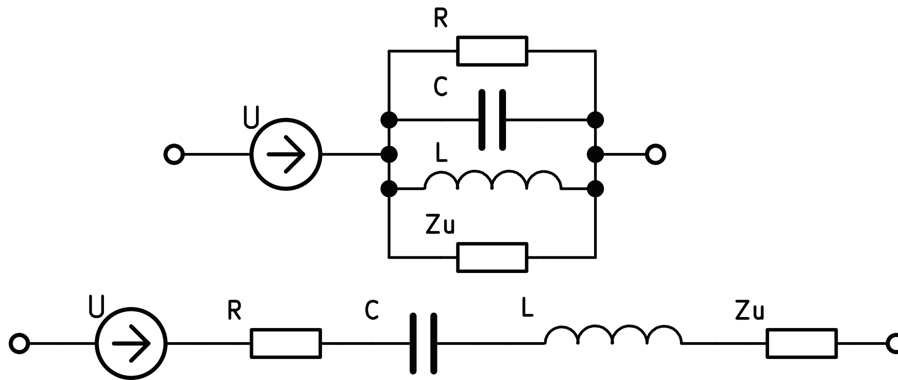


Figure 4.9: Delta gap is always in series with lumped circuit, but lumped circuit itself can be serial, or parallel. It is also possible to disable the delta gap (`isEnaled = false`) and port is then represented just as lumped circuit. When some element of lumped circuit has to be intentionally unused, its value has to be `NaN`. Delta gap is in circuit schematically shown as voltage source, but can be current source as well, or can be disabled at all.

4.7.6 Public Methods

Model `Feeding` is accessible by `atom.selectedProject.physics.feeding`. Several methods of `Feeding` use `inputParser` class for input argument management. It means that input arguments in propertyName-value pairs style can be significantly simplified. Name of properties can be stated in shorted form, *e.g.*, `'positionPar'` as `'pos'`, `'userImpedance'` as `'u'`, and case insensitive. Order of properties is moreover arbitrary and most of properties could be not defined, because have internally setted default value. Mandatory inputs are further specified.

add1DDiscretePort

```
feeding.add1DDiscretePort(n, 'baseObjName', objName, 'positionPar', position, ...
'description', portDescription)
```

Add 1D discrete port (Section 4.7.2) with number `n` into `Feeding` model lying on 1D geometry object `objName` on position `position` and description of port is `portDescription`. All properties except of `'description'` is mandatory to define.

add1DMeshPort

```
feeding.add1DMeshPort(n, 'elements', portElements, 'description', ...
portDescription)
```

Add 1D mesh port (Section 4.7.3) with number `n` into `Feeding` model lying on mesh node between two 1D mesh elements (edges) `portElements`. Description of port is `portDescription`. All properties except of `'description'` is mandatory to define.

add2DDiscretePort

```
feeding.add2DDiscretePort(n, 'baseObjName', objName, 'positionPar', position, ...
'description', portDescription)
```

Add 2D discrete port (Section. 4.7.2) with number `n` into `Feeding` model lying on 2D geometry object `objName` on position `position` and description of port is `portDescription`. All properties except of `'description'` is mandatory to define.

add2DMeshPort

```
feeding.add2DMeshPort(n, 'elements', portElements, 'description', ...
portDescription)
```

Add 2D mesh port (Section 4.7.3) with number `n` into `Feeding` model lying on mesh edge between 2D mesh elements (triangles) `portElements`. Description of port is `portDescription`. All properties except of `'description'` is mandatory to define.

addBEMDiscretePort

```
feeding.addBEMDiscretePort(n, 'baseObjName', objName, 'positionPar', ...
position, 'radius', portRadius, 'description', portDescription)
```

Add BEM discrete port (Section 4.7.2) with number `n` into `Feeding` model lying on 2D geom-

etry object `objName` on position `position` with virtual radius `portRadius` and description of port is `portDescription`. All properties except of `'description'` and `'radius'` is mandatory to define.

addPlaneWave

```
feeding.addPlaneWave(nameOfWave, 'direction', wavePolDir, ...
'propagationVector', propVec, 'initElectricField', initE, 'axialRatio', ...
axialRatio, 'isEnabled', isEnabled)
```

Add plane wave named `nameOfWave` (Section 4.7.1) into `Feeding` model with propagation vector `propVec`, axial ratio `axialRatio` and polarization direction . Only to define name is mandatory, rest of properties will have default values.

deletePort

```
feeding.deletePort(portNumbers)
```

This method deletes ports with numbers stated in numeric vector `portNumbers` from `Feeding` model.

deleteWave

```
feeding.deleteWave(nameOfWave)
```

This method deletes wave with name stated in `char` vector `nameOfWave` from `Feeding` model.

editFeeder

```
feeding.editFeeder(portNumber, 'isEnabled', enable, 'type', newType, 'value', ...
newValue)
```

This method enable editing of feeders (Section 4.7.4 and 4.7.2) on port number `portNumber`. Only first input argument is mandatory.

editLumpedCircuit

```
feeding.editLumpedCircuit(portNumber, 'isEnabled', enable, )
```

This method enable editing of lumped circuit (Section 4.7.5) on port number `portNumber`. Only first input argument is mandatory.

editPort

```
feeding.editPort(portNumber, 'number', newPortNumber, varargin)
```

Enable editing of discrete ports (Section 4.7.2 and 4.7.3) in `Feeding` model. Example stated above shows editing of port number. In `varargin` can be rest of propertyName-value pairs pertaining to specific class of edited port. Only first input argument is mandatory.

editWave

```
feeding.editWave(waveName, 'name', newWaveName, varargin)
```

Enable editing of plane waves (Section 4.7.1) in `Feeding` model. Example stated above shows editing of wave name. In `varargin` can be propertyName-value pairs of plane wave. Only first input argument is mandatory.

getPortValues

```
portValues = feeding.getPortValues(); portValues = feeding.getPortValues(frequencyList)
```

Method returns information about all ports which are affecting simulation task, *i.e.*, ports which has enabled lumped circuit or feeder. It is also possible to set custom frequencies for lumped circuit impedance computation.

getWaveValues

```
portValues = feeding.getWaveValues()
```

Method returns information about all plane waves which are affecting simulation task, *i.e.*, waves which are enabled.

4.7.7 Public Properties

table

This property contain list of all components managed by `Feeding` model.

Chapter 5

Solvers

5.1 BEM

5.1.1 General

It considers an arbitrarily shaped, triple planar circuit having coupling port. Solving the wave equation over the whole area is reduced into a contour-integral form, which relates the voltage and current along the periphery.

If the substrate height (z-dimension) will be much smaller than the wavelength, it is possible to assume that $\partial/\partial z = 0$ and $H_z = E_x = E_y = 0$. It leads to following equation

$$V(s) = \frac{1}{2\mathrm{j}} \oint\limits_{C} \left[k \cos(\theta) \mathrm{H}_1^{(2)}(kr) V(s_0) - \mathrm{j}\omega\mu d \mathrm{H}_0^{(2)}(kr) i_n(s_0) \right] \mathrm{d}s_0, \tag{5.1}$$

where $\mathrm{H}_0^{(2)}$ and $\mathrm{H}_1^{(2)}$ are the Hankel functions of the first and second kind. This approximation leads to definition of limitation. The height of substrate $d < \lambda/20$, where λ is a wavelength within the substrate.

Also segment width W should be smaller than $\lambda/10$. In other case, user will be informed, that the results are not valid, using the message "Mesh is poor". If the periphery is divided into N segments, that the magnetic and electric field intensities are constant over each width of segments, the integral equation results in a system of matrix equations

$$\sum_{j=1}^{N} u_{ij} V_j = \sum_{j=1}^{N} h_{ij} I_j \qquad (i = 1, 2, ...N), \tag{5.2}$$

where

$$u_{ij} = \delta_{ij} - \frac{k}{2\mathrm{j}} \int\limits_{W_j} \cos(\theta) \mathrm{H}_1^{(2)}(kr) \mathrm{d}s, \tag{5.3}$$

$$h_{ij} = \begin{cases} \frac{\omega\mu d}{2} \frac{1}{W_j} \int\limits_{W_j} \mathrm{H}_0^{(2)}(kr) \mathrm{d}s & (i \neq j) \\ \frac{\omega\mu d}{2} \left[1 - \frac{2\mathrm{j}}{\pi} \left(\ln\left(\frac{kW_i}{4}\right) - 1 + \gamma \right) \right] & (i = j) \end{cases}. \tag{5.4}$$

And his solving leads to obtain the rf voltage on each segment as

$$\boldsymbol{V} = \mathbf{U}^{-1} \mathbf{H} \boldsymbol{I}. \tag{5.5}$$

Then it is possible to show the impedance matrix

$$\mathbf{Z} = \mathbf{U}^{-1} \mathbf{H}. \tag{5.6}$$

This matrix is stored in property matrix for other computation in results class.

5.1.2 Public methods

`setSubstrateHeight` **more**

`BEM.setSubstrateHeight(height)`

Method sets the height of substrate. It is able to use absolute value or some variable from workspace. Note, that the height of the substrate has to be much smaller than wavelength.

`setDielectric` **more**

`BEM.setDielectric(dielectric)`

Method sets the dielectric part of the substrate using the name of material from the dielectric database.

`setMetal` **more**

`BEM.setMetal(metal)`

Method sets the metallic part of substrate using of metallic name within materials database. Both, patch and ground parts, are set as a same metallic material.

`solve` **more**

`BEM.solve(userFrequency)`

Method starts to calculate impedance matrix by Boundary Elements Method using of frequency list from physics class. If user defines specific frequency list to input of method, this frequency list will be used.

`getVoltage` **more**

`BEM.getVoltage()`

hod 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.

`getPeripherySamples` **more**

`BEM.getPeripherySamples`

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.

`getParametersZ` **more**

`BEM.getParametersZ`

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.

`getParametersS` **more**

`sParameters = getParametersS(Z0)`

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).

`getCurrentsOnPorts` **more**

`currentOnPorts = getCurrentsOnPorts`

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

5.1.3 How to use BEM without AToM

Boundary elements method is possible to use also without AToM session. This kind of using is only for skilled users. Firstly, it is needed to create initial object of BEM class using `models.solvers.BEM`. This object has method `initializeSideAccess()` which is able to set important parameters for computing.

Those parameters are divided to four parts (four structures). First is structure physics, which carries information about physical constants (as light speed, permittivity, permeability of vacuum, et.)

Second structure carries information about materials (dielectric and metal). These materials describe investigated planar structure.

Third input is a height of substrate and the last is a mesh.

`initializeSideAccess` more

```
initializeSideAccess(physics, materials, d, mesh)
```

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!

Geometrical parameters are samples on the periphery of inspected planar circuit. These samples are in structure `mesh.getBoundaryMesh.nodes{1}{1}`. If any hole is within this circuit, periphery of this hole is store into `mesh.getBoundaryMesh.nodes{1}{n}`. Where $n = 2$ and for more holes the increment n is increasing. More details are in example `launchBEM`.

5.2 MoM1D

MoM1D is a frequency domain solver based on method of moments which can solve current distribution on three-dimensional wire structures using electric field integral equation (EFIE). It uses piecewise-linear basis functions. Since most of the properties are the same as in the MoM2D, user is advised to read the documentation of the MoM2D.

Note that MoM1D does not support lumped elements, derivatives with respect to angular frequency and other Z-matrix related matrices.

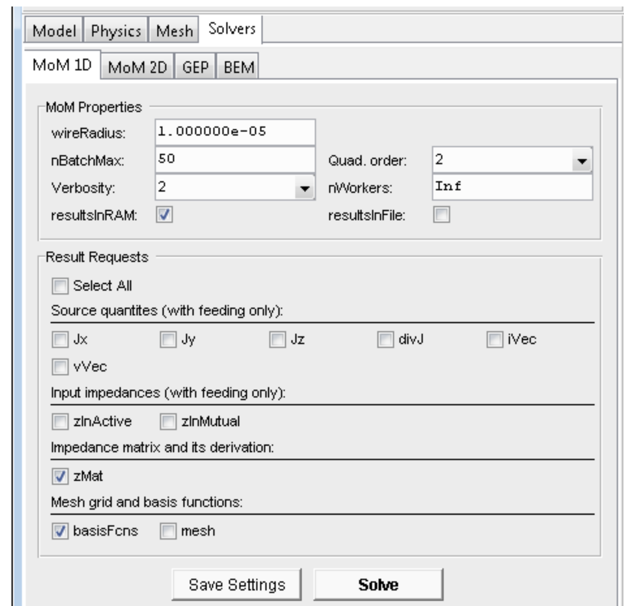


Figure 5.1: MoM1D Graphical User Interface

5.2.1 Public Properties

All properties can be set using method `atom.selectedProject.solvers.MoM1D.setProperties( ... 'prop1Name', prop1Value, 'prop2Name', prop2Value, .. )` or using graphical interface (see Fig. 5.1). Default values are shown in brackets {}.

`nBatchMax`

`Double: {500}`

Maximum amount of data to be processed at once. The value balances speed vs. memory requirements of the computation. Higher value should speed-up the computations but results in higher memory requirements. Note that after reaching certain value, the computation time will not decline any more. Based on observations, there is an optimal value for each system.

`nWorkers`

`Double: {0} .. Inf`

Number of parallel workers (threads) to be used. Note that monitoring of progress of parallel computation will not be frequent in MATLAB versions older than R2017a.

`nWorkers == 0`: Computation will be performed using single thread only (for-loop).

`nWorkers > 0`: Computation will be performed using number of threads given by value of nWorkers (parfor-loop). Parallel Computing Toolbox has to be installed.

`nWorkers == Inf`: Computation will be performed using preferred number of workers in a parallel pool set in MATLAB Parallel Computing Toolbox Preferences. Parallel Computing Toolbox has to be installed.

`quadOrder`

`Double: 1 | {2} | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12`

Order of numerical quadrature used for numerical integration. Higher order results in better precision but enlarges computational time.

`resultRequests`

`String: 'Jx'| 'Jy'| 'Jz'| {'basisFcns'} | 'divJ'| 'iVec'| {'mesh'} | 'vVec'| ... 'zInActive'| 'zInMutual'| {'zMat'}`

Selection of requested output quantities of the solver. The list can be obtained using possibleResultRequests(). You can request multiple results using comma-separated list, spaces are optional (*e.g.*`'Jx, Jy,Jz'`).

`resultsInFile`

`Logical: {false}`
If set to `true`, results will be saved into MAT-files.

`resultsInRAM`
`Logical: {true}`
If set to `true`, results will remain in memory.

`verbosity`
`Double: 0 | 1 | {2} | 3`
Controls amount of text being typed in the command line.
`verbosity == 0`: No information will be shown in the command line, no progress bar.
`verbosity == 1`: Progress bar will be shown.

`wireRadius`
`Double: {1e-5}`
Radius of all wire segments in the model.

5.3 MoM2D

MoM2D is a frequency domain solver based on method of moments which can solve current distribution on three-dimensional planar structures using electric field integral equation (EFIE). It uses the Rao-Wilton-Glisson (RWG) basis functions [?]. The method results in a solution of system of linear equations formed as

$$\mathbf{Z}\mathbf{I} = \mathbf{V}. \tag{5.7}$$

The $\mathbf{Z}$ is an impedance matrix. Values of the impedance matrix are given by geometrical properties of the analyzed structure

$$z_{mn} = \frac{\mathrm{j}Z_0}{k_0} \int\limits_{\Omega_m} \int\limits_{\Omega_n} \left\{ k_0^2 \boldsymbol{\psi}_m \cdot \boldsymbol{\psi}_n - \nabla \cdot \boldsymbol{\psi}_m \nabla' \cdot \boldsymbol{\psi}_n \right\} \frac{\mathrm{e}^{-\mathrm{j}k_0 R_{mn}}}{4\pi R_{mn}} \, \mathrm{d}S' \, \mathrm{d}S \tag{5.8}$$

where Z_0 is the impedance of free space, k_0 is the free space wave-number, $\boldsymbol{\psi}_m, \boldsymbol{\psi}_n$ are testing and basis functions and R_{mn} is a distance between source and observation point. The matrix can be requested by adding `'zMat'` into the list of requests `resultRequests`. If the `'zMat'` is present in the `resultRequests` and `resultsInRAM` is set to `true`, raw values of the impedance matrix will be available after successful computation in `atom.selectedProject.solvers.MoM2D.results.zMat.data`. Other data respect the same pattern:
`myData = atom.selectedProject.solvers.MoM2D.reults.<name>.data.`

For modal decomposition, it can be interesting to get just parts of the impedance matrix: magnetic (M) and electric (E), both of them can be further divided into quasi-static (0) and dynamic (k) part. Thanks to convenient divergence of the RWG testing/basis functions, we get

$$\mathrm{zMatMk_{mn}} = \chi_{m,n} \int\limits_{\Omega_m} \int\limits_{\Omega_n} \boldsymbol{\psi}_m \cdot \boldsymbol{\psi}_n \frac{\mathrm{e}^{-\mathrm{j}k_0 R_{mn}} - 1}{4\pi R_{mn}} \, \mathrm{d}S' \, \mathrm{d}S \tag{5.9}$$

$$\mathrm{zMatM0_{mn}} = \chi_{m,n} \int\limits_{\Omega_m} \int\limits_{\Omega_n} \boldsymbol{\psi}_m \cdot \boldsymbol{\psi}_n \frac{1}{4\pi R_{mn}} \, \mathrm{d}S' \, \mathrm{d}S \tag{5.10}$$

$$\mathrm{zMatEk_{mn}} = \chi_{m,n} \int\limits_{\Omega_m} \int\limits_{\Omega_n} \frac{\mathrm{e}^{-\mathrm{j}k_0 R_{mn}} - 1}{4\pi R_{mn}} \, \mathrm{d}S' \, \mathrm{d}S \tag{5.11}$$

$$\mathrm{zMatE0_{mn}} = \chi_{m,n} \int\limits_{\Omega_m} \int\limits_{\Omega_n} \frac{1}{4\pi R_{mn}} \, \mathrm{d}S' \, \mathrm{d}S \tag{5.12}$$

where

$$\chi_{m,n} = \begin{cases} \frac{l_m l_n}{A_m A_n}, & \text{for contributions} ++, -- \\ -\frac{l_m l_n}{A_m A_n}, & \text{for contributions} +-, -+ \end{cases}. \tag{5.13}$$

Remaining request related to impedance matrix is 'zMatLE', which is a part related to lumped elements. Using the requests 'zMatMk', 'zMatM0', 'zMatEk', 'zMatE0' and 'zMatLE', impedance matrix can be assembled as follows

$$\mathrm{zMat} = \mathrm{j}Z_0 k_0 \left(\mathrm{zMatMk} + \mathrm{zMatM0}\right) - \frac{\mathrm{j}Z_0}{k_0}\left(\mathrm{zMatEk} + \mathrm{zMatE0}\right) + \mathrm{zMatLE}. \tag{5.14}$$

Derivative of the impedance matrix with respect to angular frequency can be obtained using 'zMatD'. It can be also assembled using above mentioned requests and

$$\mathrm{tMatM_{mn}} = \chi_{m,n} \int\limits_{\Omega_m}\int\limits_{\Omega_n} \boldsymbol{\psi}_m \cdot \boldsymbol{\psi}_n \frac{\mathrm{e}^{-\mathrm{j}k_0 R_{mn}}}{4\pi}\,\mathrm{d}S'\,\mathrm{d}S \tag{5.15}$$

$$\mathrm{tMatE_{mn}} = \chi_{m,n} \int\limits_{\Omega_m}\int\limits_{\Omega_n} \frac{1}{4\pi}\,\mathrm{d}S'\,\mathrm{d}S. \tag{5.16}$$

Using additional requests 'tMatM', 'tMatE' and derivative of the lumped part 'zMatLED', derivative of the impedance matrix can be assembled in the following way:

$$\mathrm{zMatD} = \frac{\mathrm{j}Z_0}{\omega}\left(k_0\left(\mathrm{zMatMk} + \mathrm{zMatM0}\right) + \frac{1}{k_0}\left(\mathrm{zMatEk} + \mathrm{zMatE0}\right)\right) \tag{5.17}$$

$$+\frac{Z_0}{c_0}\left(k_0 \mathrm{tMatM} - \frac{1}{k_0}\mathrm{tMatE}\right) + \mathrm{zMatLED}. \tag{5.18}$$

The $\mathbf{I}$ (5.7) is an unknown vector of coefficients representing approximation of electric current density over the analysed structure. The current density is given as

$$J \approx \sum_{n=1}^{N} \boldsymbol{\psi}_n i_n \tag{5.19}$$

where $\boldsymbol{\psi}_n$ is n-th RWG basis function, i_n is n-th coefficient of the current density vector and N is total number of basis functions which is given by number of inner edges of the computational mesh. Since information about the basis functions can be requested using 'basisFcns' and coefficients using 'iVec', user is able to evaluate current density distribution at any point of the geometry. With `resultRequests` containing (at least) 'mesh, basisFcns, iVec', user can get the components of the current density and its divergence using getJInPoints() (see Listing 5.1).

Listing 5.1: Obtaining components of the current density.

```
% ...after solution:
mesh = atom.selectedProject.solvers.MoM2D.reults.mesh.data;
basisFcns = atom.selectedProject.solvers.MoM2D.reults.basisFcns.data;
iVec = atom.selectedProject.solvers.MoM2D.reults.iVec.data;

% obtaining components of the current density
[Jx, Jy, Jz, divJ, pointCoordinates] = ...
   models.solvers.MoM2D.computation.getJInPoints(...
   mesh, basisFcns, iVec);
```

In order to evaluate the quantities in centers of each triangle, user can just add requests 'Jx', 'Jy', 'Jz' or 'divJ' into the `resultRequests`. See Section 5.3.1 for list of possible result requests.

The $\mathbf{V}$ in (5.7) is a known excitation vector

$$v_m = \int\limits_{\Omega_m} \boldsymbol{\psi}_m\left(\mathbf{r}\right) \cdot \mathbf{E}^i\left(\mathbf{r}\right)\,\mathrm{d}S. \tag{5.20}$$

Its values can be requested by adding 'vVec' into the list of requests `resultRequests` (see Section 5.3.1).

The solver also provides active impedance on delta-gap ports 'zInActive' and mutual impedances between ports 'zInMutual'. Note that at least one valid delta-gap source must be defined in order to compute `zInActive` and at least two delta-gaps to compute `zInMutual`.

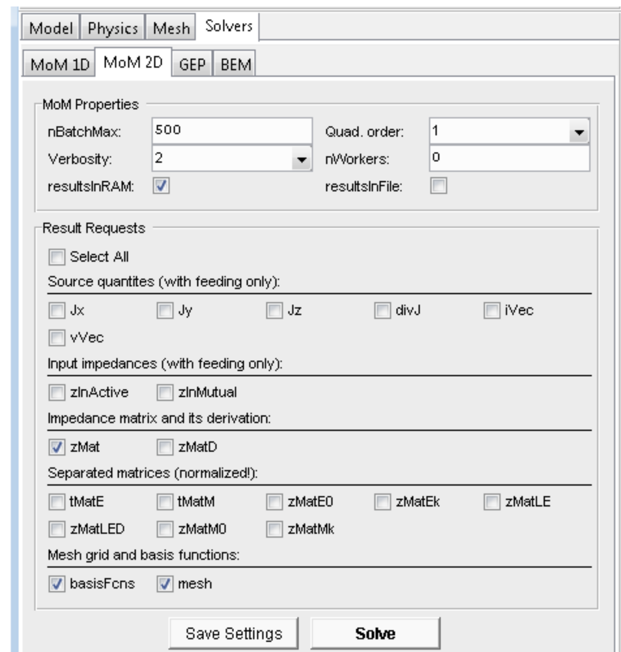


Figure 5.2: MoM2D Graphical User Interface

5.3.1 Public Properties

All properties can be set using method `atom.selectedProject.solvers.MoM2D.setProperties( ... 'prop1Name', prop1Value, 'prop2Name', prop2Value, .. )` or using graphical interface (see Fig. 5.2). Default values are shown in brackets {}.

`nBatchMax`

`Double: {500}`

Maximum amount of data to be processed at once. The value balances speed vs. memory requirements of the computation. Higher value should speed-up the computations but results in higher memory requirements. Note that after reaching certain value, the computation time will not decline any more. Based on observations, there is an optimal value for each system.

`nWorkers`

`Double: {0} .. Inf`

Number of parallel workers (threads) to be used. Note that monitoring of progress of parallel computation will not be frequent in MATLAB versions older than R2017a.

`nWorkers == 0`: Computation will be performed using single thread only (for-loop).

`nWorkers > 0`: Computation will be performed using number of threads given by value of nWorkers (parfor-loop). Parallel Computing Toolbox has to be installed.

`nWorkers == Inf`: Computation will be performed using preferred number of workers in a parallel pool set in MATLAB Parallel Computing Toolbox Preferences. Parallel Computing Toolbox has to be installed.

`quadOrder`

`Double: 1 | {2} | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12`

Order of numerical quadrature used for numerical integration. Higher order results in better precision but enlarges computational time.

`resultRequests`

`String: 'Jx'| 'Jy'| 'Jz'| {'basisFcns'} | 'divJ'| 'iVec'| {'mesh'} | 'tMatE'| ... 'tMatM'| 'vVec'| 'zInActive'| 'zInMutual'| {'zMat'} | 'zMatD'| 'zMatE0'| 'zMatEk'| ... 'zMatLE'| 'zMatLED'| 'zMatM0'| 'zMatMk'`

Selection of requested output quantities of the solver. The list can be obtained using possibleResultRequests(). You can request multiple results using comma-separated list, spaces

are optional (*e.g.*`'Jx, Jy,Jz'`).

`resultsInFile`

`Logical: {false}`

If set to `true`, results will be saved into MAT-files.

`resultsInRAM`

`Logical: {true}`

If set to `true`, results will remain in memory.

`verbosity`

`Double: 0 | 1 | {2} | 3`

Controls amount of text being typed in the command line.

`verbosity == 0`: No information will be shown in the command line, no progress bar.

`verbosity == 1`: Progress bar will be shown.

Chapter 6

Generalized Eigenvalue Problem Solver

Generalized Eigenvalue Problem Solver (GEP Solver) provide solution of generalized eigenvalue problem

$$\mathbf{A}\mathbf{I}_n = \lambda_n \mathbf{B}\mathbf{I}_n \tag{6.1}$$

with normalization to

$$\frac{1}{2}\mathbf{I}_m^{\mathrm{H}}\mathbf{N}\mathbf{I}_n = \delta_{mn}, \tag{6.2}$$

where $\mathbf{A}$ and $\mathbf{B}$ are input matrices, $\mathbf{N}$ is normalization matrix, λ_n represent the resulting eigen-numbers, $\mathbf{I}_n$ corresponding eigen-vectors and δ_{mn} is Kronecker delta. Superscript H denotes Hermitian transpose. For example for characteristic modes $\mathbf{A} = \mathbf{X}$ and $\mathbf{B} = \mathbf{N} = \mathbf{R}$, where $\mathbf{X}$ and $\mathbf{R}$ is imaginary and real part of impedance matrix, respectively. Input matrices should, according the theory [?], be symmetric, *i.e.*, $\mathbf{A} = \mathbf{A}^{\mathrm{T}}$ and $\mathbf{B} = \mathbf{B}^{\mathrm{T}}$.

The range of eigen-values $\lambda_n \in (-\infty, \infty)$ is too wide. It is plotted typically as $\log_{10} |\lambda_n|$ or the characteristic angle (eigen-angle) was defined for better resolution

$$\delta_n = 180^\circ - \arctan(\lambda_n), \tag{6.3}$$

which maps the eigenvalues into the limited range $\delta_n \in (90^\circ, 270^\circ)$ with resonance $\lambda_n = 0$ corresponding to $\delta_n = 180^\circ$.

The MATLAB has two functions for eigen-values and eigen-vectors computation, GEP Solver is based on them:

- `eig`: Use QZ algorithm (non-symmetrical matrices) or Choleski factorization (symmetrical matrices). [more in MATLAB documentation]
- `eigs`: Use Arnoldi package (ARPACK) to compute subset of eigen-values and eigen-vectors. [more in MATLAB documentation]

6.1 Tracking

The tracking problem appeared when eigen-values of modes are computed over frequency range, especially with more complex geometries (as compared with wired structures). The order of modes might be switched. Tracking algorithm is used to reorganize mode order over frequency range to get continuous eigen-value curves. Several tracking algorithms can be found, for example, in [?, ?, ?, ?, ?].

The principle of correlation based algorithm is illustrated in Fig. 6.1. The correlation table ρ_{mn} is computed for each eigen-vector at f_1 and all eigen-vectors at f_2. The highest correlation value, if the similarity is higher than set constant (typically $|\rho_{mn}| \geq 0.8$) connect the m-th mode at f_1 with n-th mode at f_2. In case there is not sufficient correlation value, tracking has failed. It could happen for several reasons. First is number of modes given to ARPACK to search for.

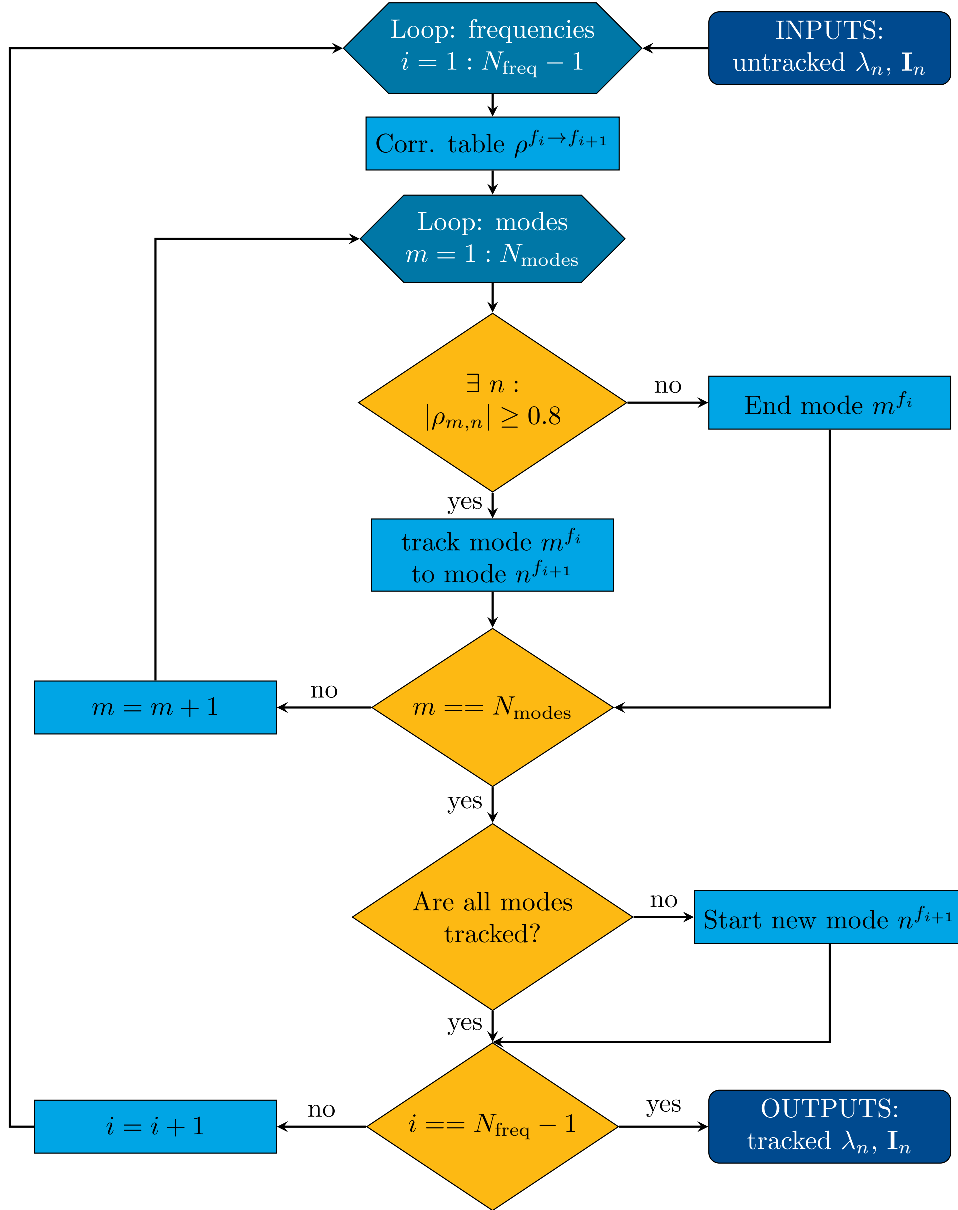


Figure 6.1: Tracking algorithm

When only the first several modes are required, it is possible, that at higher frequency the smallest eigen-values have different modes. Other reason could be a high gradient of the eigen-value (that is a reason, why large eigen-values are filtered in pre-processing). The algorithm can fail at higher frequency for insufficiently dense mesh, because these modes can have more complex surface current distributions.

Once all modes between frequencies f_1 and f_2 are connected or unconnected, modes are ended and new modes are opened, the algorithm moves to next frequencies f_2 and f_3. When it is finished connecting between frequencies $f_{N_{\text{freq}}-1}$ and $f_{N_{\text{freq}}}$, post-processing part is applied to tracked data.

6.1.1 Correlation Computing

Formulations for correlations are in detail described in [**?**]. Three types of correlation implemented in AToM are:

- Eigen-vector correlation

$$\rho_{mn} = \frac{\mathbf{I}_m^{\mathrm{T}} \mathbf{I}_n}{|\mathbf{I}_m|\,|\mathbf{I}_n|} \tag{6.4}$$

- Modal radiated power

$$\rho_{mn} = P_{\mathrm{rad}} = \frac{1}{2}\mathbf{I}_m^{\mathrm{T}}\mathbf{R}\mathbf{I}_n \tag{6.5}$$

- Far-field correlation [?]

$$\rho_{mn} = \frac{1}{2Z_0}\int\limits_{S_2} \mathbf{E}_m^*(\boldsymbol{r}) \cdot \mathbf{E}_n(\boldsymbol{r})\,\mathrm{d}S \tag{6.6}$$

  Modal far-field can be computed as

$$\mathbf{E}_n(\boldsymbol{r}) = -\mathrm{j}kZ_0\frac{\mathrm{e}^{-\mathrm{j}k|\boldsymbol{r}|}}{4\pi\,|\boldsymbol{r}|}\int\limits_{\Omega}\left(\boldsymbol{J}_n(\boldsymbol{r}') - \left(\boldsymbol{J}_n(\boldsymbol{r}')\cdot\hat{\mathbf{n}}\right)\hat{\mathbf{n}}\right)\mathrm{e}^{\mathrm{j}k\hat{\mathbf{n}}\cdot\boldsymbol{r}'}\,\mathrm{d}S, \tag{6.7}$$

  where $\hat{\mathbf{n}}$ denotes the unit vector from source point on surface $\boldsymbol{r}'$ to observed point $\boldsymbol{r}$, $\boldsymbol{J}_n$ is eigen-current, Z_0 is impedance of free space, Ω denote antenna's surface and S_2 represents unit sphere.

6.2 Adaptive Frequency Solver

The adaptive frequency solver was implemented as an additional "output analyzing" algorithm to increase precision of tracking.

At first the pre-processing is applied to computed eigen-vaules and eigen-vectors at originally given frequency samples — all modes with large eigen-values (typically $|\lambda_n| \geq 10^6$) are discarded as they have very small influence to the resonance. After that the main part of tracking algorithm is launched and than post-processing part with tracked data follows:

- Opened modes are checked with all previously closed modes if they can be connected. Correlation tables are computed between these modes and if the correlation is sufficient, modes are joined (with a gap in the middle of them).
- Short modes (typically with the length over only one or two frequency samples) are discarded — they are not interesting for the wideband frequency range.
- Modes running out of the resonance area (typically with eigen-angle δ_n out of range $[120°, 240°]$) are discarded — they are not interesting for resonating structure.

These procedures serve to make the resulting data more transparent. Note that all properties mentioned above are optional and can be controlled by user. The constants can be changed too.

After this one iteration of the tracking algorithm, the detecting algorithm of AFS is initialized. The tracked data are analyzed and it is decided, where the frequency range should be densed and new frequency samples are inserted here. At this new frequencies is inner solver computed again and eigen-values and eigen-vectors are decomposed. New results are placed at correct position between the older one and are re-tracked together. Number of AFS iterations is chosen.

The deciding criteria when insert more points between frequencies f_i and f_j can be:

- Resonance – if mode's track crossed value of $\delta_n = 180°$ between f_i and f_j.
- Gap in mode – if the mode has a gap at one of frequencies f_i, f_j.
- End of Mode – if the mode ends at f_i.
- Start of Mode – it the mode starts at f_j.
- Crossing of modes – if any traces of two modes are crossed between f_i and f_j.

The scheme of AFS algorithm is illustrated in Fig. 6.2.

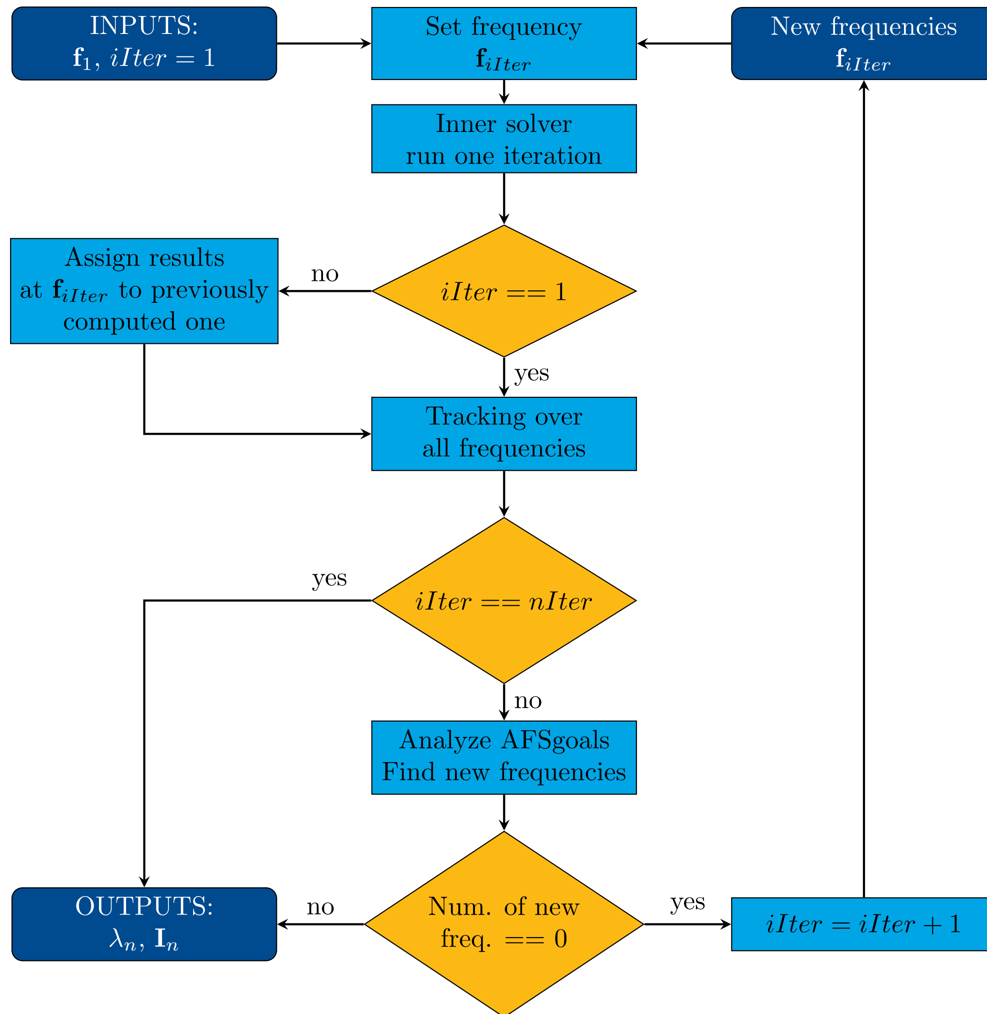


Figure 6.2: Algorithm of Adaptive Frequency Solver (AFS).

6.3 Public Functions

All functions described below are static functions and can be called without using AToM. They are located at GEP namespace.

Complete syntax is `outputs = models.solvers.GEP.functionName(inputs)`.

assignEigNumbers more

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

Assign eigen-numbers to modes track.

computeAlpha more

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

Compute matrix of alpha coefficients.

computeBeta more

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

Compute matrix of beta coefficients.

computeCorrelation more

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

Compute correlation of eigen-vectors.

computeCorrelation2D more

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

Compute correlation of 2D matrixes.

computeCorrTable more

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

Compute correlation table between eigen-vectors. General function deciding which core for correlation table computing will be used.

computeCorrTableFF more

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

Compute correlation table using correlation between far-field of modes.

computeCorrTableII more

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

Compute correlation table using correlation between eigen-vectors.

computeCorrTableIRI more

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

Compute correlation table using surface correlation.

computeFF more

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

Compute modal far-fields.

computeModalExcitation more

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

Compute matrix of modal excitation factors.

computeModalQ more

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

Compute matrix of modal excitation factors.

computeModalSignificance more

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

Compute matrix of modal significance factors.

computePiFactor more

```
PiFac = computePiFactor(IorJ, eigNum, mesh, basisFcns)
```

Compute matrix of Pi factors.

connectModes more

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

Connect interrupted modes.

delNegValues more

```
M = delNegValues(M)
```

Delete negative eigen-values of matrix.

discardModes more

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

Discard modes according to specification in options.

findMaxUsedModeNumber more

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

Find max used mode number in given modesTrack matrix.

gep more

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

Solve Generalized Eigenvalue Problem.

getOption more

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

Return value of required option from options structure.

prepareResultStruct **more**

`res = prepareResultStruct(description, dimensions, freqDepDim, units, data)`

Prepare structure for results.

procgep **more**

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

Solve Generalized Eigenvalue Problem with pre and post processing.

scanModesProperties **more**

`modesProp = scanModesProperties(modesTrack, gepOptions)`

Return modes properties from given modesTrack.

solve **more**

`solve(objGEP, frequencyList)`

Run GEP Solver.

symmetrizeMatrix **more**

`symA = symmetrizeMatrix(A)`

Make the matrix symmetric.

trackingCM **more**

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

Track modes between 2 frequency samples.

6.4 Public Methods

List of public method which can be evaluated on GEP object. GEP object can be obtained as `objGEP = atom.selectedProject.solver.GEP`.

clearInputs **more**

`objGEP.clearInputs()`

Clear input matrices, frequency list and inner solver object.

clearOutputs **more**

`objGEP.clearOutputs()`

Clear all outputs in results structure.

defaultControls **more**

`defControls = objGEP.defaultControls(innerSolver)`

Provide structure of control handles for given inner solver.

getDefaultProperties **more**

`defaultProperties = objGEP.getDefaultProperties()`

Return structure of default GEP properties.

getPropertyList **more**

`propertyList = objGEP.getDefaultProperties()`

Return names of properties.

resetPropertiesToDefault **more**

`objGEP.resetPropertiesToDefault()`

Reset properties of GEP to default values.

setCorrInputData **more**

`objGEP.setCorrInputData(corrInputData)`

Set corrInputData to GEP properties.

setFrequencyList **more**

`objGEP.setFrequencyList(frequencyList)`

Set list of frequencies to GEP properties.

`setMatrix` more

`objGEP.setMatrix(nameOfMatrix, matrix)`

Set data to given input to GEP properties.

`setMatrixA` more

`objGEP.setMatrixA(A)`

Set A as matrixA to GEP properties.

`setMatrixB` more

`objGEP.setMatrixB(B)`

Set B as matrixB to GEP properties.

`setMatrixN` more

`objGEP.setMatrixN(N)`

Set N as matrixN to GEP properties.

`solve` more

`objGEP.solve(frequencyList)`

Solve GEP.

`updateResult` more

`objGEP.updateResult(result, newData)`

Update new data to given result of GEP.

6.5 Public Properties

All properties can be set using method `atom.selectedProject.solvers.GEP.setProperties( ... 'prop1Name', prop1Value, 'prop2Name', prop2Value, .. )`. Default values are shown in brackets {}.

All properties can be readed from structure `atom.selectedProject.solvers.GEP.options;`. This structure with default values is created in constructor of GEP object and updated at very beginning of method `solve` with changes made by user.

6.5.1 General

`showStatusWindow`

`Logical: {true}`

Show status window during calculation of solver.

`verbosity`

`Double: 0 | {1} | 2`

Write progress into command line.

`verbosity == 0`: No information in command line.

`verbosity == 1`: Basic information in command line.

`verbosity == 2`: Detailed information in command line.

6.5.2 Pre-processing

`delNegValuesOfInputMatrices`

`Logical: {false}`

Decompose input matrices with `eig` and discard negative eigen-numbers.

`symmetrizeInputMatrices`

`Logical: {false}`

Symmetrize input matrices as `M = 1/2 * (M + M.')`.

6.5.3 GEP

innerSolver

String: 'empty'| 'custom'| 'CMs (MoM1D)'| {'CMs (MoM2D)'} | 'CMs (MoM2D + Smatrix)'

Name of predefined inner solver or custom.

innerSolverHndl

String: {'MoM2D'}

Name of other AToM solver or string with handle function which return object of custom inner solver. See Section 6.7 for more details.

innerSolverSolve

String: {'@(objInnerSolver, frequencyList, waitBar)objInnerSolver.solve(frequencyList, ... waitBar)'}

String with handle function which define how to solve inner solver at given frequencies.

innerSolverGetA

String: {'@(objInnerSolver)imag(objInnerSolver.results.zMat.data)'}

String with handle function which define how to get matrix `A` from inner solver.

innerSolverGetB

String: {'@(objInnerSolver)real(objInnerSolver.results.zMat.data)'}

String with handle function which define how to get matrix `B` from inner solver.

innerSolverGetN

String: {'@(objInnerSolver)NaN'}

String with handle function which define how to get matrix `N` from inner solver. When `N` is empty or `NaN`, `N = B` is used for normalization, it is not necessary to duplicate same matrices in `N` and `B`.

matricesStorage

String: {'ram'} | 'hdd'

System how to input matrices for GEP are computed.

`matricesStorage == 'ram'`: All matrices are stored in memory.

`matricesStorage == 'hdd'`: Each frequency sample is computed separately and stored in `.mat` file in folder which is created by solver. Note that only `corrType = 'II'` is allowed in this mode mode.

nModes

Double: {10}

Number of modes computed by `eigs`. When `nModes == 0`, `eig` is computed.

normConst

Double: {0.5}

Normalized constant `c` used in (6.2) as `c * I * N * I = 1`.

eigRunPreAndPostprocessing

Logical: {false}

Determine if functions defined in `eigPreprocessing` and `eigPostprocessing` are executed before/after `eigs` is called.

eigPreprocessing

String: {'@(data, iFreq, objGEP)myPreprocFcn(data, iFreq, objGEP)'}

String with handle function which is executed before `eigs` is called. See Section 6.7 for more details.

eigPostprocessing

String: {'@(eigVec, eigNum, iFreq, objGEP, dataFromPreproc)myPostprocFcn(eigVec, ... eigNum, iFreq, objGEP, dataFromPreproc)'}

String with handle function which is executed after `eigs` is called. See Section 6.7 for more details.

6.5.4 Post-processing

`discardNegativeINI`
: `Logical: {false}`
Discard modes with `INI < 0`.

`discardComplexLambda`
: `Logical: {true}`
Discard modes with complex eigen-number.

`maxMagnEigVal`
: `Double: {1e6}`
Maximal value of eigen-number, `abs(eigNum)>` `maxMagnEigVal` is considered as `inf`.

6.5.5 Tracking

`corrType`
: `String: {'II'} | 'IRI'| 'FF'| 'custom'| 'none'`
Type of used computation of correlation table.

`corrDataHndl`
: `String: {'@(objGEP)myCorrInputData(objGEP)'}`
String with handle function which return correlation input data for custom computation of correlation table. See Section 6.7 for more details.

`corrTableHndl`
: `String: {'@(eigVec, corrInputData, gepOptions, corrTable, ... statusWindow)myCorrTable(eigVec, corrInputData, gepOptions, corrTable, ... statusWindow)'}`
String with handle function which return custom correlation table. See Section 6.7 for more details.

`minCorrValue`
: `Double: {0.8}`
Minimal correlation value for modes connecting.

`minModeLength`
: `Double: {1}`
Minimal length of mode after tracking. Shorter modes are discarded.

`maxLengthOfGap`
: `Double: {5}`
Maximal length of gap in mode (in number of freq. samples). Modes with larger gap are splitted into two modes.

`charAngleBoundaryTop`
: `Double: {270}`
Modes with eigen-angle running above this level are discarded.

`charAngleBoundaryBottom`
: `Double: {90}`
Modes with eigen-angle running under this level are discarded.

`charAngleBoundarySwitch`
: `Logical: {true}`
Discard modes with eigen-angle switching between values above `charAngleBoundaryTop` and below `charAngleBoundaryBottom`.

6.5.6 AFS

`AFSnIterations`
: `Double: {1}`

Number of iterations of solver. When `AFSnIterations = 1`, no adaptive features are used.

AFSnInsertedSamples
`Double: {3}`
Number of inserted samples between two freq. samples.

AFSinsertedSamplesHandle
`String: {'@(f1, f2, n)f1+(1:n)* (1/(n+1))* (f2 - f1)'}`
String with handle function where are defined distribution of `n` inserted samples between frequencies `f1 < f2`.

AFSgoal
`String: {''} | 'resonance'| 'crossing'| 'startOfMode'| 'endOfMode'| 'gapInMode'`
Rules, where are inserted new samples. More goals can be defined as comma separated string.

AFSregionBoundaryTop
`Double: {270}`
Top level for defining of region, where AFS goals conditions are decided.

AFSregionBoundaryBottom
`Double: {90}`
Bottom level for defining of region, where AFS goals conditions are decided.

AFSresultsAtGivenSamples
`Logical: {true}`
Final results are returned only at given samples. Inserted samples are discarded, but the tracking is preserved.

6.6 How to use GEP

GEP Solver can be controlled as part of AToM package (using GUI or commands in Command Line) or GEP can be created as standalone object without need to have AToM object (see Section 6.8 further). The GEP Solver requires frequency and mesh to be set before. Next lines are about using GEP with AToM.

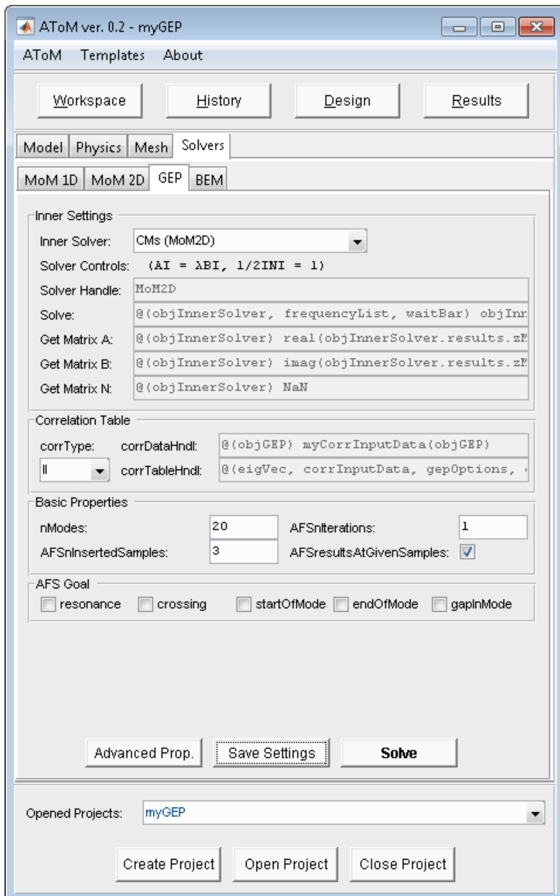
All followed settings can be set using AToM GUI, interface of the GEP Solver is shown in Fig. 6.3. How to prepare simple task with a dipole for the GEP Solver is stated in Listing 6.1.

Listing 6.1: Prepare task with dipole for GEP Solver.

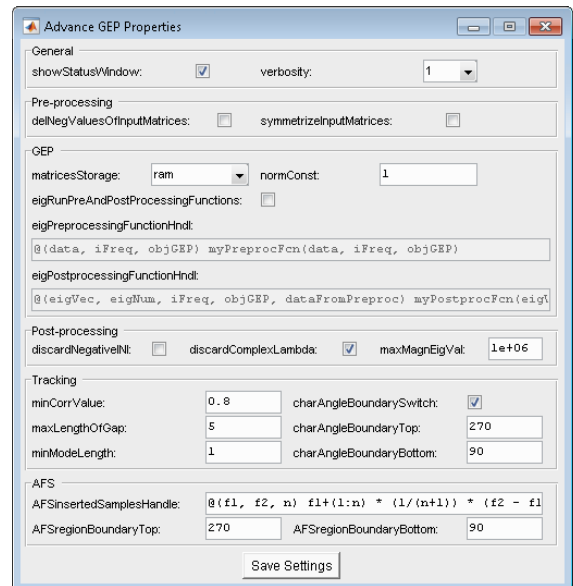
```
% start AToM and create project
atom = Atom.start(false);
atom.createProject('atomStart');
% set frequency list
frequencyList = (1:0.2:8)*1e8;
atom.selectedProject.physics.setFrequencyList(frequencyList);
% create dipole
atom.selectedProject.geom.addRectangle('[0 0 0]', '1', '1/20', 'z', 'Dipole1');
% get mesh
atom.selectedProject.mesh.getMesh;
```

Now it is possible to run GEP Solver. In default state is defined to compute characteristic modes with MoM2D as inner solver thus GEP can be started by clicking on Solve button or by command `atom.selectedProject.solver.GEP.solve`.

The Status window (SW), see Fig. 6.4, shows the progress of GEP computations. Actual results are displayed in the axes of SW at the end of each iteration. One can switch between showing eigen-angles or eigen-numbers and current axes can be separated in new window.



(a) Main settings and controls.



(b) Advance properties settings.

Figure 6.3: GUI of GEP Solver. Basic properties mentioned in Section 6.5 can be set in main window, the rest in Advance properties window.

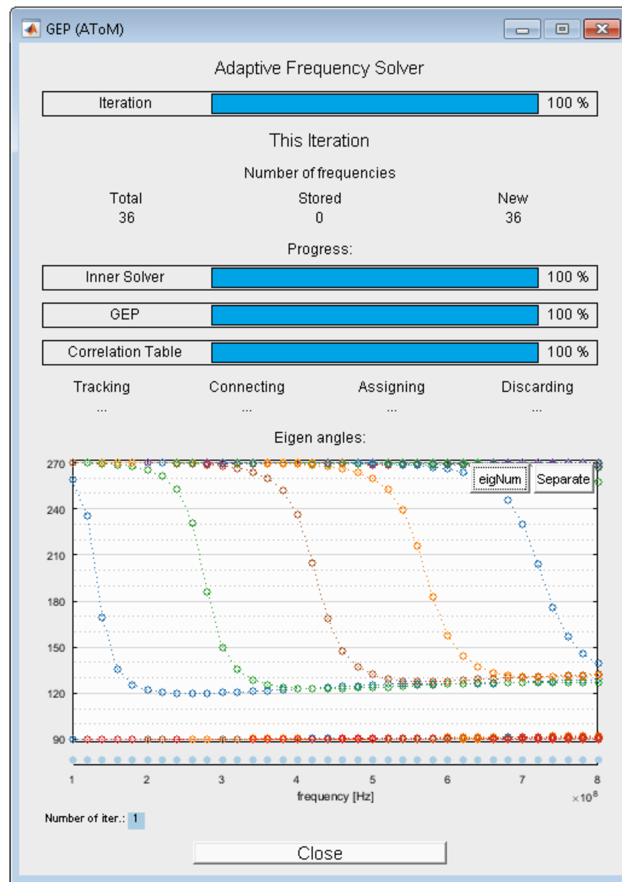


Figure 6.4: GEP Status Window

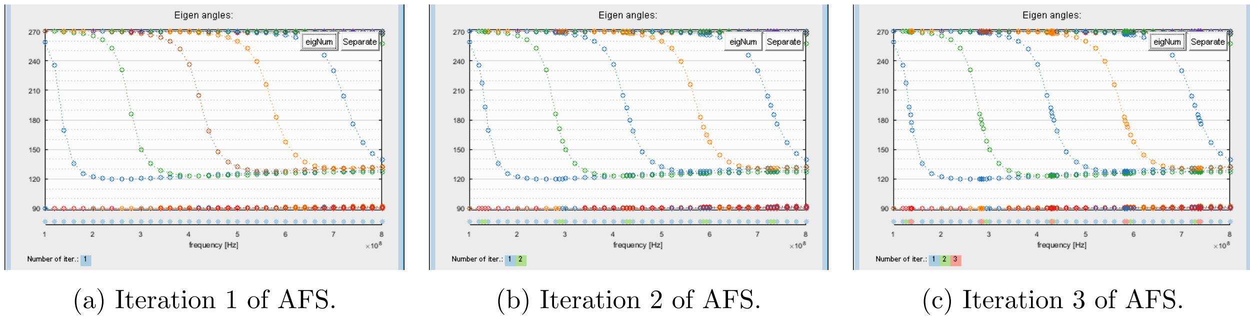
6.6.1 Results

Results of GEP are provided in `results = atom.selectedProject.solver.GEP.results;` as a structure with this fields:

- `eigNum`: Eigen-numbers are broken down to two fields:
  - `sorted`: sorted eigen-numbers after tracking,
  - `unsorted`: untracked raw eigen-numbers computed by `eig/eigs`.
- `eigVec` Eigen-vectors are broken down to two fields:
  - `sorted`: sorted eigen-vectors after tracking,
  - `unsorted`: untracked raw eigen-numbers computed by `eig/eigs`.
- `iterNumberAFS`: Vector of numbers denoting in which AFS iteration the sample were computed.
- `modesTrack`: It denotes on which position the sorted modes were originally computed in unsorted data.
- `corrTable`: Correlation table serving to track the modes.

6.6.2 Setting of properties

All properties listed above in Section 6.5 can be set using GEP Solver's method `setProperties`. How to set, for example, number of computed modes to 20 and use far-field correlations (Section 6.1.1) is demonstrated in Listing 6.2.



(a) Iteration 1 of AFS. (b) Iteration 2 of AFS. (c) Iteration 3 of AFS.

Figure 6.5: Cutouts of GEP status Window showing gradual refinement around resonance using AFS option of GEP.

Listing 6.2: Set properties to GEP Solver.

```
atom.selectedProject.solver.GEP.setProperties(...
    'nModes', 20,...
    'corrType', 'FF');
atom.selectedProject.solver.GEP.solve;
```

6.6.3 Run AFS

The AFS (Section 6.2) can be run easily by set property `AFSnIterations > 1`. Concurrently it is necessary to set a goal evaluated by AFS. It is appropriate to set resonance goal for the dipole from Listing 6.1. Number of inserted samples was changed to two.

Listing 6.3: Run AFS in GEP Solver.

```
atom.selectedProject.solver.GEP.setProperties(...
    'AFSnIterations', 3,...
    'AFSgoal', 'resonance',...
    'AFSnInsertedSamples', 2);
atom.selectedProject.solver.GEP.solve;
```

Sequence of figures in Fig. 6.5 show gradual refinement around resonance ($\delta_n = 180°$).

6.6.4 Using S-matrix decomposition

Using **S** matrix [?] (more in Section 8.1) leads to more accuracy and it allows to compute modes of higher order. To apply this method in GEP is necessary to implement additional function into solver. How to do that is described in Section 6.7, however for standard decomposition with **X** matrix as input `A` and **S** matrix as input `B` is pre-deffined setting of property `innerSolver = ... 'CMs (MoM2D + Smatrix)'`.

6.7 Customization

The GEP Solver can be customized by user-defined functions thus GEP Solver can be used for arbitrary problem. Only limitation is to keep up given inputs and outputs.

6.7.1 Customize Inner Solver

When property `innerSolver == 'custom'`, another five properties have to be set properly:

- `innerSolverHndl` (go to →),
- `innerSolverSolve` (go to →),
- `innerSolverGetA` (go to →),

- innerSolverGetB (go to →),
- innerSolverGetN (go to →).

innerSolverHndl

Property innerSolverHndl has to contain a string with handle function which create inner solver or name of other solver in AToM. This function is called once at beginning of GEP Solver.

Inputs

- `objGEP`: object of GEP class. GEP has references for Workspace, Solvers, etc. when it is used with AToM.

Outputs

- `objInnerSolver`: created object of inner solver. Can be structure too.

Examples

- `innerSolverHndl = 'MoM2D'`: Name of AToM solver.
- `innerSolverHndl = '@(objGEP)objGEP.solver.MoM2D'`: Alternative enrollment of command above.
- `innerSolverHndl = ['@(objGEP)models.solvers.GEP.customFunctions.'... 'solverSMatrixDecomposition(objGEP)']`: Static function is used to create inner solver. Example of that function is in Listing 6.4.

Listing 6.4: Custom static function is used to create inner solver.

```
function mySolver = solverSMatrixDecomposition(objGEP)
%% solverSMatrixDecomposition create solver for SMatrix decomposition
% This function create solver when S matrix is used for
% computing characteristic modes.

% MoM2D of AToM is used
mySolver.solver = objGEP.solver.MoM2D;
% alocate field for results
mySolver.S = [];
mySolver.X = [];
mySolve.mesh = [];
end
```

innerSolverSolve

Property innerSolverSolve has to contain a string with handle function which run the inner solver on given frequencies. This function is called once at every iteration of AFS.

Inputs

- `objSolver`: object of inner solver.
- `frequencyList`: list of frequencies.
- `waitbar`: reference to waitbar in GEP Status Window.

Outputs

- `objSolver`: Function can have one output – inner solver or the function can be without any output.

Examples

- `innerSolverSolve = ['@(objInnerSolver, frequencyList,waitBar)'... 'objInnerSolver.solve(frequencyList, waitBar)']`: For example, MoM2D can be called this way.
- `innerSolverSolve = ['@(objInnerSolver, frequencyList, waitbar)'... 'models.solvers.GEP.customFunctions.solveSMatrixDecomposition('... 'objInnerSolver, frequencyList, waitBar)']`: Static function is used to solve inner solver. Example of that function is in Listing 6.5.

Listing 6.5: Custom static function is used to solve inner solver.

```
function objSolver = solveSMatrixDecomposition(objSolver, frequencyList, waitBar)
%% solveSMatrixDecomposition run solver for SMatrix decomposition
% This function run custom inner solver in GEP when S matrix is used for
% computing characteristic modes.

%% solve MoM2D
objSolver.solver.solve(frequencyList, waitBar);

%% constants
nFrequencies = length(frequencyList);
mesh = objSolver.solver.results.mesh;
basisFcns = objSolver.solver.results.basisFcns;
maxDegreeL = 20;
maxAlpha = 2 * maxDegreeL * (maxDegreeL + 2);

%% compute S
S = nan(maxAlpha, basisFcns.nUnknowns, nFrequencies);
for iF = 1:nFrequencies
   S(:, :, iF) = models.utilities.matrixOperators.SMatrix.computeS(mesh,...
      basisFcns, frequencyList(iF), maxDegreeL);
end

%% save for outputs
objSolver.S = S;
objSolver.X = imag(objSolver.solver.results.zMat.data);
objSolver.mesh = mesh;
end
```

innerSolverGetA

Property `innerSolverGetA` has to contain a string with handle function to get input matrix `A` from inner solver to input for GEP.

Inputs

- `objSolver`: object of inner solver.

Outpus

- `A`: Matrix `A`.

Examples

- `innerSolverGetA = '@(objSolver)imag(objSolver.results.zMat.data)'`: Imaginary part of impedance matrix. Used when characteristic modes are computed.
- `innerSolverGetA = '@(objSolver)objSolver.X'`: Used when S-matrix is used in decomposition.

innerSolverGetB

Property `innerSolverGetB` has to contain a string with handle function to get input matrix `B` from inner solver to input for GEP.

Inputs

- `objSolver`: object of inner solver.

Outpus

- `B`: Matrix `B`.

Examples

- `innerSolverGetB = '@(objSolver)real(objSolver.results.zMat.data)'`: Real part of impedance matrix. Used when characteristic modes are computed.
- `innerSolverGetB = '@(objSolver)objSolver.S'`: Used when S-matrix is used in decomposition.

innerSolverGetN

Property `innerSolverGetN` has to contain a string with handle function to get normalized matrix `N` from inner solver to input for GEP. Can return `NaN`, when `N = B` to avoid duplication of matrix `B` in memory.

Inputs

- `objSolver`: object of inner solver.

Outpus

- `N`: Matrix `N`.

Examples

- `innerSolverGetN = '@(objSolver)real(objSolver.results.zMat.data)'`: Real part of impedance matrix is duplicated in memory.
- `innerSolverGetN = NaN`: matrix `B` is used for normalization but it is not stored in memory twice.

6.7.2 User-defined `eig`/`eigs` preprocessing and postprocessing

When are input matrices `A` and `B` computed, modal decomposition by `eig` or `eigs` follows. In some cases can be useful to include intermediate step. Setting of property `eigRunPreAndPostprocessing ... = true` allows to specify two functions – `eigPreprocessing` (go to →) and `eigPostprocessing` (go to →) – which are executed before and after modal decomposition, respectively.

eigPreprocessing

Property `eigPreprocessing` has to contain a string with handle function which is executed before `eig`/`eigs` on each frequency sample.

Inputs

- `data`: structure with fields `A`, `B` and `N` containing input matrices.
- `iFreq`: number of actual frequency sample.
- `objGEP`: object of GEP Solver.

Outpus

- `data`: structure with fields `A`, `B` and `N` containing modified input matrices.
- `dataForPostproc`: arbitrary variable which allows to pass any data to post-processing function.

Examples

- `eigPreprocessing = ['@(data, iFreq, objGEP)models.solvers.GEP.'... 'customFunctions.preEigSMatrixDecomposition(data, iFreq, objGEP)']`: Pre-processing function, example can be seen in Listing 6.6.

Listing 6.6: Pre-processing function used by GEP with S-matrix decomposition.

```
function [data, dataForPostproc] = preEigSMatrixDecomposition(data, ~, ~)
%% 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.

%% computed data from inner solvers
X = data.A;
S = data.B;
nCMmax = min(size(S));

%% svd decompostion
[SvecU, Ssvd, SvecV] = svd(S);
Ssvd_square = Ssvd(1:nCMmax,1:nCMmax);

%% SVD-representation of the problem
Xq  = SvecV.'*X*SvecV;

%% Model-order reduction:
X11 = Xq(1:nCMmax, 1:nCMmax);
X12 = Xq(1:nCMmax, nCMmax+1:end);
X21 = Xq(nCMmax+1:end, 1:nCMmax);
X22 = Xq(nCMmax+1:end, nCMmax+1:end);

%%
RU1 = Ssvd_square*Ssvd_square.';
XU1 = X11 - X12*(X22\X21);

%% save data
data.A = XU1;
data.B = RU1;
dataForPostproc.X21 = X21;
dataForPostproc.X22 = X22;
dataForPostproc.SvecV = SvecV;
end
```

eigPostprocessing

Property eigPostprocessing has to contain a string with handle function which is executed after eig/eigs on each frequency sample.

Inputs

- eigVec: computed eigen-vectors.
- eigNum: computed eigen-numbers.
- iFreq: number of actual frequency sample.
- objGEP: object of GEP Solver.
- dataFromPreproc: data computed in pre-processing function.

Outpus

- eigVec: modified eigen-vectors.
- eigNum: modified eigen-numbers.

Examples

- eigPreprocessing = ['@(eigVec, eigNum, iFreq, objGEP, dataFromPreproc)'... 'models.solvers.GEP.customFunctions.postEigSMatrixDecomposition(eigVec,'... 'eigNum, iFreq, objGEP, dataFromPreproc)']: Post-processing function, example can be seen in Listing 6.7.

Listing 6.7: Post-processing function used by GEP with S-matrix decomposition.

```
function [eigVec, eigNum] = postEigSMatrixDecomposition(eigVec, eigNum, ~, ~, ...
    dataFromPreproc)
%% 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.
%

% computed data from inner solver
X21 = dataFromPreproc.X21;
X22 = dataFromPreproc.X22;
SvecV = dataFromPreproc.SvecV;

% recalculate eigVec
eigVec = SvecV * [eye(size(eigVec, 1)); - X22\X21] * eigVec;
end
```

6.7.3 Customize Correlation Table

Two handle function for customization of correlation table can be set to properties corrDataHndl and corrTableHndl if is set corrType == 'custom'.

corrDataHndl

Property corrDataHndl has to contain a string with handle function which is executed during inner solver processing. Arbitrary data for correlation table ocmputation can be stored.

Inputs

- `objGEP`: object of GEP Solver.

Outpus

- `corrInputData`: modified eigen-vectors.

Examples

- `corrDataHndl = '@(objGEP)myCorrInputData(objGEP)'`: Arbitrary data can be obtained from GEP object and stored in variable `corrInputData` (typically in structure form). This variable is not processed, it is only handed down into custom function for correlation table.

corrTableHndl

Property `corrTableHndl` has to contain a string with handle function which is computing correlation table between eigenvectors.

Inputs

- `eigVec`: computed eigen-vectors in GEP.
- `corrInputData`: data computed in inner solver.
- `gepOptions`: structure wit GEP settings.
- `corrTable`: correlation table (correlation table computed in previous AFS iteration.).
- `statusWindow`: GEP Status window.

Outpus

- `corrTable`: correlation table.

Examples

- `corrTableHndl = '@(eigVec, corrInputData, gepOptions, corrTable, '... 'statusWindow)myCorrTable(eigVec, corrInputData, gepOptions, '... 'corrTable, statusWindow)'`: Handle function which compute custom correlation table.

6.8 GEP without AToM

GEP can be created with some limitations without need of AToM. Because without existing AToM object GEP can not have reference to physics, workspace, mesh and other solvers, only values `'custom'` or `'empty'` are allowed for property `innerSolver`. All settings of mesh and other properties is up tu user in custom functions described in Section 6.7. An example setting follows in Listing 6.10. GEP can be also used when an impedance matrix is already computed and inner solver is not needed. Such example is in Listing 6.11.

Listing 6.8: An example how to use GEP without AToM and with MoM2D as inner solver.

```
myGEP = models.solvers.GEP.GEP;
myGEP.setProperties(...
   'innerSolver', 'custom',...
   'innerSolverHndl', '@(objGEP) myMoM2D(objGEP)',...
   'innerSolverSolve', '@(innerSolver, frequencyList, waitbar) ...
      myMoM2DSolve(innerSolver, frequencyList, waitbar)',...
```

```
    'innerSolverGetA', '@(innerSolver) innerSolver.X',...
    'innerSolverGetB', '@(innerSolver) innerSolver.R',...
    'nModes', 30,...
    'AFSnIterations', 3,...
    'AFSgoal', 'crossing');
frequencyList = linspace(1, 3, 21) .* 1e8;
myGEP.solve(frequencyList);
```

Listing 6.9: Function `myMoM2D` for example in Listing 6.10

```
function mySolver = myMoM2D(objGEP)

% Run AToM for mesh:
atom = Atom.start(false);
delete('myEllipse.atom')
atom.createProject('myEllipse');
atom.selectedProject.geom.addEllipse('[0 0 0]', '[1 0 0]', '[0 1/2 0]', '0', ...
    '2*pi', 'Ellipse1');
atom.selectedProject.mesh.setElementSizeFromFrequency(false);
atom.selectedProject.mesh.setGlobalMeshDensity('0.1');
atom.selectedProject.mesh.getMesh();
mesh = atom.selectedProject.mesh.getMeshData2D();
atom.quit;

% prepare fields
mySolver.mesh = mesh;
mySolver.R = [];
mySolver.X = [];
end
```

Listing 6.10: Function `myMoM2DSolve` for example in Listing 6.10

```
function mySolver = myMoM2DSolve(mySolver, frequencyList, waitbar)
%% inputs for MoM
gaps = [];
waves = [];
symmetries = [];
settings = struct(...
    'quadOrder', 1, ...
    'nBatchMax', 5e3, ...
    'nWorkers', 0, ...
    'resultPath', pwd, ...
    'resultsInRAM', true, ...
    'resultsInFile', false, ...
    'verbosity', 2);
requests = {'mesh', 'basisFcns', 'zMat'};

% solve MoM
Res = models.solvers.MoM2D.solve(mySolver.mesh, frequencyList, gaps, waves, ...
    symmetries, settings, requests, waitbar);

% set outputs
mySolver.X = imag(Res.zMat.data);
mySolver.R = real(Res.zMat.data);
```

Listing 6.11: An example how to use GEP without AToM and without inner solver when input matrices are already computed.

```
%% frequency list - at this frequencies is computed zMatrix
frequencyList = linspace(0.5e9, 1.5e9, 21);
```

```
%% precomputing input matrices
% can be loaded from file, here are computed by MoM2D in AToM
% - create dipole - compute zMat - close AToM
atom = Atom.start(false);
atom.createProject('zMatComputation');
atom.selectedProject.geom.addRectangle('[0 0 0]', '1', '0.05', 'z', 'dipole');
atom.selectedProject.mesh.getMesh();
atom.selectedProject.physics.setFrequencyList(frequencyList);
atom.selectedProject.solver.MoM2D.solve;
zMat = atom.selectedProject.solver.MoM2D.results.zMat.data;
atom.quit; % quit AToM

%% create and set GEP object
myGEP = models.solvers.GEP.GEP();
myGEP.setProperties(...
   'innerSolver', 'empty',... % set innerSolver to empty
   'nModes', 20); %increase number of modes

%% set input matrices (3 ways)
% way 1:
myGEP.setMatrixA(imag(zMat));
myGEP.setMatrixB(real(zMat));
% way 2:
% myGEP.setMatrix('A',imag(zMat));
% myGEP.setMatrix('B',real(zMat));
% % way3:
myGEP.setMatrices(imag(zMat),real(zMat),[]);

%% solve
% solve with frequency list as input
myGEP.solve(frequencyList);
% OR
% myGEP.setFrequencyList(frequencyList);
% myGEP.solve;
```

Chapter 7

Results

7.1 Overview

Results is part of the AToM which is intended to work with outcomes coming from AToM solvers. This package contains several tools for their post-processing and following visualisation which is an important part of the design process, it provides you informations about your project and also physical insight into your design.

Results processing in AToM can be done in two completely different ways. First, low-level approach, which puts demands on the user is based on set functions (see Section 7.2) which can be used. Second, high-level approach, is about special standalone results browser application (see Section 7.3) which can be operated by its graphical user interface.

Listing 7.1: Basic example of usage AToM and current visualisation by Results.

```
% init AToM, create project
atom = Atom.start(false);
atom.createProject('basicExample');

% Physics
frequencyList = 150e6;
atom.selectedProject.physics.setFrequencyList(frequencyList);

% Geometry
atom.selectedProject.geom.addParallelogram('[-0.5, -0.02, 0]', ...
'[0.5, -0.02, 0]', '[-0.5, 0.02, 0]', 'dipole');

% Feeding
atom.selectedProject.physics.feeding.add2DDiscretePort(1, ...
'baseObjName', 'dipole', ...
'positionPar', '[0, -0.02, 0; 0, 0.02, 0]', ...
'description', 'port1');
atom.selectedProject.physics.feeding.editFeeder(1, 'isEnabled', true, ...
'type', 'voltage', 'value', '1');

% Mesh
atom.selectedProject.mesh.setGlobalMeshDensity(20);
atom.selectedProject.mesh.getMesh();

% MoM2D solver setup
atom.selectedProject.solver.MoM2D.setProperties('resultRequests', ...
'basisFcns, mesh, iVec', ...
'verbosity', 2, 'nBatchMax', 2000, ...
'nWorkers', 0);
atom.selectedProject.solver.MoM2D.setProperties('quadOrder', 1);
atom.selectedProject.solver.MoM2D.solve();

% collect results
mesh = atom.selectedProject.solver.MoM2D.results.mesh;
```

```
basisFcns = atom.selectedProject.solver.MoM2D.results.basisFcns;
iVec = atom.selectedProject.solver.MoM2D.results.iVec.data;

% quit AToM
atom.quit;

% plot current
results.plotCurrent(mesh, 'basisFcns', basisFcns, 'iVec', iVec);
```

7.2 Public functions

This section is a list of all low-level functions that can be used to process or display outcomes from AToM solvers. Notice that only basic examples of use are listed here, for further details look into the list of functions related to this document or check corresponding files in examples folder.

`calculateCharacteristicAngle` more

`characteristicAngle=results.calculateCharacteristicAngle(eigennumber)`

Computes characteristic angles from eigennumbers.

`calculateCharge` more

`divJ=results.calculateCharge(mesh, basisFcns, iVec)`

Computes charge distribution on the structure with respect to the inserted vector of expansion coefficients.

`calculateCurrent` more

`[Jx, Jy, Jz]=results.calculateCurrent(mesh, basisFcns, iVec)`

Computes current distribution on the structure with respect to the inserted vector of expansion coefficients.

`calculateCurrentDecomposition` more

```
decomposition=results.calculateCurrentDecomposition(mesh, basisFcns, iVec, ...
frequency)
```

Computes current decomposition using projection of spherical harmonics using **S** matrix (see Sec. 8.1).

`calculateEigennumber` more

`eigennumber=results.calculateEigennumber(characteristicAngle)`

Computes eigennumbers from characteristic angles.

`calculateFarField` more

```
farFieldStructure=results.calculateFarField(mesh, frequency, 'basisFcns', ...
basisFcns, 'iVec', iVec)
```

Calculates the far-field radiation pattern and other related quantities.

`calculateNearField` more

```
nearFieldStructure=results.calculateNearField(mesh, basisFcns, iVec, ...
frequency, plane, distance, uPoints, vPoints)
```

Computes the near electric and magnetic field.

`calculateQFBW` more

`QFBW=results.calculateQFBW(zIn, frequency, alpha)`

Computes quality factor Q_{FBW} [?].

`calculateQZ` more

`[QZ, QZtuned]=results.calculateQZ(zIn, frequency)`

Computes quality factor Q_{Z} [?].

`calculateRCS` more

`RCS=results.calculateRCS(mesh, basisFcns, iVec, frequency)`

Computes radar cross section (RCS). Depending on the incident wave and position of observation point can be computed monostatic or bistatic radar cross section.

`calculateS` more

```
s = results.calculateS(zIn)
```

Computes s-parameters from z-parameters.

`plotBasisFcns` more

```
handles=results.plotBasisFcns(mesh, basisFcns)
```

Creates plot of basis functions.

`plotCharacteristicAngle` more

```
handles=results.plotCharacteristicAngle('characteristicAngle', charAngle, ...
'frequency', frequency)
```

Creates plot of characteristic angles.

`plotCharge` more

```
handles=results.plotCharge(mesh, 'divJ', divJ)
```

Creates plot of charge distribution on the structure.

`plotCurrent` more

```
handles=results.plotCurrent(mesh, 'J', J)
```

Creates plot of current distribution on the structure.

`plotCurrentDecomposition` more

```
handles = results.plotCurrentDecomposition('frequency', frequencyList, ...
'decomposition', decomposition)
```

Creates plot of decomposition of given current into projection of spherical harmonics.

`plotEigennumber` more

```
handles=results.plotEigennumber('frequency', frequency, 'eigennumber', ...
eigennumber)
```

Creates plot of eigennumbers.

`plotFarField` more

```
handles=results.plotFarField('farField', farFieldMatrix, 'phi', phiVector, ...
'theta', thetaVector)
```

Creates plot of far-field radiation pattern.

`plotFarFieldCut` more

```
handles=results.plotFarFieldCut('theta', thetaVector, 'phi', phiVector, ...
'farField', farFieldMatrix, 'thetaCut', thetaCutValue)
```

Creates plot of far-field radiation pattern cut.

`plotMesh` more

```
handles=results.plotMesh(mesh)
```

Creates plot of inserted mesh structure and its remarkable attributes.

`plotNearField` more

```
handles=results.plotNearField('nearField', nearField, 'uPoints', uPoints, ...
'vPoints', vPoints, 'plane', 'x', 'distance', distance)
```

Creates plot of near-field.

`plotQ` more

```
handles=results.plotQ(Q, frequency)
```

Creates plot of given quality factor.

`plotRCS` more

```
handles=results.plotRCS('RCS', RCS, 'independentVariable', 'theta', 'phi', ...
phi, 'theta', theta, 'fixedDimensionPhi', fixedValue)
```

Creates plot of radar cross section.

`plotS` more

```
handles = results.plotS('frequency', frequency, 's', s)
```

Creates plot of s-parameters.

All plotting functions return structure containing references to all graphical objects used in specified plot. These references can be used to modify plot appearance.

Listing 7.2: Example of usage all public functions.

```
% init AToM, create project
atom = Atom.start(false);
atom.createProject('basicExample');

% Physics
nFrequencies = 10;
frequencyList = linspace(100e6, 300e6, nFrequencies);
atom.selectedProject.physics.setFrequencyList(frequencyList);

% Geometry
atom.selectedProject.geom.addParallelogram('[-0.5, -0.02, 0]', ...
'[0.5, -0.02, 0]', '[-0.5, 0.02, 0]', 'dipole');

% Feeding
atom.selectedProject.physics.feeding.add2DDiscretePort(1, ...
'baseObjName', 'dipole', ...
'positionPar', '[0, -0.02, 0; 0, 0.02, 0]', ...
'description', 'port1');
atom.selectedProject.physics.feeding.editFeeder(1, 'isEnabled', true, ...
'type', 'voltage', 'value', '1');

% Mesh
atom.selectedProject.mesh.setGlobalMeshDensity(20);
atom.selectedProject.mesh.getMesh();

% MoM2D solver setup
atom.selectedProject.solver.MoM2D.setProperties('resultRequests', ...
'basisFcns, mesh, iVec, zInActive, zMat', ...
'verbosity', 2, 'nBatchMax', 2000, ...
'nWorkers', 0);
atom.selectedProject.solver.MoM2D.setProperties('quadOrder', 1);
atom.selectedProject.solver.MoM2D.solve();

% GEP solver
atom.selectedProject.solver.GEP.solve();

% collect results
% MoM2D
mesh = atom.selectedProject.solver.MoM2D.results.mesh;
basisFcns = atom.selectedProject.solver.MoM2D.results.basisFcns;
iVec = atom.selectedProject.solver.MoM2D.results.iVec.data;
zInActive = atom.selectedProject.solver.MoM2D.results.zInActive.data;

%GEP
eigNum = atom.selectedProject.solver.GEP.results.eigNum.sorted.data;

% characteristicAngle
characteristicAngle = results.calculateCharacteristicAngle(eigNum);
handles1 = results.plotCharacteristicAngle('frequency', frequencyList, ...
'characteristicAngle', characteristicAngle);

% charge
divJ = results.calculateCharge(mesh, basisFcns, iVec(:, 2));
handles2 = results.plotCharge(mesh, 'divJ', divJ);

% current
[J(:,1), J(:,2), J(:,3)] = results.calculateCurrent(mesh, basisFcns, ...
iVec(:, 2));
handles3 = results.plotCurrent(mesh, 'J', J);

% currentDecomposition
decomposition = results.calculateCurrentDecomposition(mesh, basisFcns, ...
iVec, frequencyList);
handles4 = results.plotCurrentDecomposition('frequency', frequencyList, ...
'decomposition', decomposition);
```

```
% eigennumber
eigennumber = results.calculateEigennumber(characteristicAngle);
handles5 = results.plotEigennumber('frequency', frequencyList, ...
'eigennumber', eigennumber);

% far-field
farFieldStructure = results.calculateFarField(mesh, frequencyList(2), ...
'basisFcns', basisFcns, 'iVec', iVec(:, 2));
handles6 = results.plotFarField('farField', farFieldStructure.D, ...
'theta', farFieldStructure.theta, 'phi', farFieldStructure.phi);

% far-field cut
handles7 = results.plotFarFieldCut('theta', farFieldStructure.theta, ...
'phi', farFieldStructure.phi, 'farField', farFieldStructure.D, ...
'thetaCut', pi/2);

% near-field
plane = 'y';
distance = 0;
uPoints = linspace(-1, 1, 100);
vPoints = uPoints;
nearFieldStructure = results.calculateNearField(mesh, basisFcns, ...
iVec(:, 2), frequencyList(2), plane, distance, uPoints, vPoints);
handles8 = results.plotNearField('nearField', nearFieldStructure.E, ...
'uPoints', uPoints, 'vPoints', vPoints, 'plane', 'x', ...
'distance', distance);

% QFBW
QFBW = results.calculateQFBW(zInActive, frequencyList, 0.1);
handles9 = results.plotQ(QFBW, frequencyList);

% QZ
QZ = results.calculateQZ(zInActive, frequencyList);
handles10 = results.plotQ(real(QZ), frequencyList);

% disable feeder, set plane wave
atom.selectedProject.physics.feeding.editFeeder(1, 'isEnabled', false);
atom.selectedProject.physics.feeding.addPlaneWave('Plane_Wave', ...
'propagationVector', '[0 0 1]', 'initElectricField', '[1 0 0]', ...
'axialRatio', 'Inf', 'direction', 'right', 'isEnabled', true);

% MoM2D solver setup
atom.selectedProject.solver.MoM2D.setProperties('resultRequests', ...
'basisFcns, mesh, iVec', ...
'verbosity', 2, 'nBatchMax', 2000, ...
'nWorkers', 0);
atom.selectedProject.solver.MoM2D.setProperties('quadOrder', 1);
atom.selectedProject.solver.MoM2D.solve();

% collect results
% MoM2D
iVecPW = atom.selectedProject.solver.MoM2D.results.iVec.data;

% RCS
[RCS, theta, phi] = results.calculateRCS(mesh, basisFcns, iVec(:, 2), ...
frequencyList(2));
handles11 = results.plotRCS('RCS', RCS, 'independentVariable', 'theta', ...
'fixedDimensionPhi', pi/2, 'theta', theta, 'phi', phi);

% s-parameters
s = results.calculateS(reshape(zInActive, [1 1 length(zInActive)]));
handles12 = results.plotS('frequency', frequencyList, 's', s);

% mesh
handles13 = results.plotMesh(mesh);
```

```
% basisFcns
handles14 = results.plotMesh(mesh);
handles14 = results.plotBasisFcns(mesh, basisFcns, 'handles', handles14);

% quit AToM
atom.quit;
```

7.3 Results class

`Results` class is a standalone application which is in AToM responsible for post-processing and visualisation of results. This application can be used from a graphical user interface (GUI) which is recommended or from classical scripts or command line. Usage of this application has at least two primary advantages. First, `Results` has its own storage called Repository, which manages all loaded as well as computed data (for more details see Section 7.3.1). Second, it is possible to control everything from GUI (see Section 7.3.2).

Listing 7.3 shows the basic commands how to start Results. Second option how to start `Results` is clicking the corresponding button in upper right corner of AToM `MainViewer` (see Fig. 9.1), which starts Results and also connects them with AToM.

Listing 7.3: Example of basic usage of Results class, which consists of starting, interconnectioning with AToM and closing.

```
% start Results class; true - opens GUI, false - without opening GUI
atomresults = results.Results.start(true);

% interconnection with AToM
atomresults.connectAtom(atomReference);

% close Results class
atomresults.exit;
```

7.3.1 Structure of repository and data types

Structure of `Results` is hierarchically divided into four separated layers with straight rules. Layers and its hierarchy is shown in Fig. 7.1.

Layers and their purpose are as follows:

- Results - container which packs everything together.
- Repository - internal memory controller which manages all data.
- Slot - container for logically related data, i.e. results of one MoM calculation. Slot stores all common informations about structure in form of mesh and basis functions.
- ListPoint - container for data with the same value of frequency. Data are stored here.

Results

`repository`

`object of class Repository`

Reference to internal data storage.

`template`

`struct`

Structure containing all graphical pre-sets.

`options`
: `struct`
Structure containing some values influencing the computations.

`version`
: `char`
Version of Results.

Repository

`slot`
: `object of class Slot`
List of references to all stored slots.

Slot

`name`
: `char`
Unique name of the Slot.

`listValues`
: `double`
Contains list or frequencies.

`domain`
: `char`
Specifies computational domain.

`mesh1D`
: `struct`
Contains mesh for results from MoM1D.

`mesh2D`
: `struct`
Contains mesh for results from MoM2D.

`meshBEM`
: `struct`
Contains mesh for results from BEM.

`basisFcns`
: `struct`
Contains basis functions for results from MoM2D.

`listPoint`
: `object of class ListPoint`
Contains all ListPoint which are located in this Slot.

ListPoint

`name`
: `char`
Unique name of the ListPoint.

`listValue`
: `double`
Frequency value specific for this ListPoint.

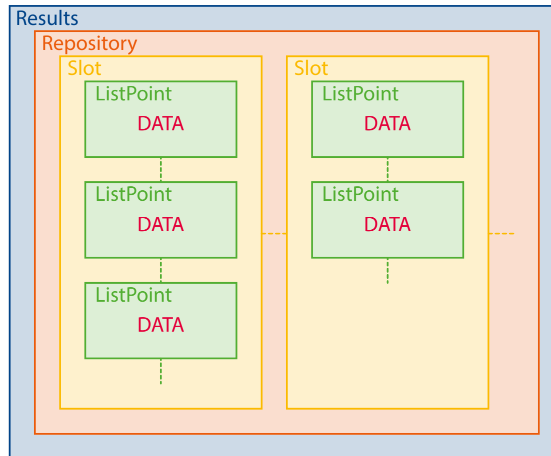


Figure 7.1: Structure of the inner layers in Results

`data`
: `cell`
: Contains loaded as well as computed data. List of all data types which can by stored is shown below.

`col`
: `cell`
: Contains unique data names which are used for data identification.

Slots and ListPoints are unambiguously specified by name, specifically, it is required that all Slots and all ListPoints in one Slot have unique names.

Components of the structure can be created directly from command line. Listing 7.4 shows commands made for Results'structure management.

Listing 7.4: Example of commands for Results'structure management.

```
% creation of structure components
atomresults.createSlot('SlotName');
atomresults.createListPoint('ListPointName', 'SlotName', 'value of the ListPoint ...
    (frequency value)');

% creation of a whole list of ListPoints
atomresults.createListPoint('ListPointName', 'SlotName', 'vector of values ...
    (frequency list)');

% removal of structure components
atomresults.removeSlot('SlotName');
atomresults.removeListPoint('ListPointName', 'SlotName');
```

Repository is open to read which allows to extract all calculated data for users' own operations (see Listing 7.5).

Listing 7.5: Example of extraction current densities data from Results.

```
% extraction of data for current density from Results
J = atomresults.repository.slot(1).listPoint(1).data.J
```

Following list contains all data types which can by stored in Results

- `iVec` - vector of expansion coefficients,

- `points` - position of computed current density,
- `J` - matrix of current densities computed in triangles' centres,
- `J_nodes` - current density computed in nodes of mesh,
- `divJ` - surface charge density,
- `divJ_nodes` - surface charge density in nodes,
- `theta` - theta angle for far-field calculation,
- `phi` - phi angle for faf-field calculation,
- `FTheta` - theta component of radiation vector in (ϑ, φ),
- `FPhi` - phi component of radiation vector in (ϑ, φ),
- `FHor` - horizontal component of radiation vector in (ϑ, φ),
- `FVer` - vertical component of radiation vector in (ϑ, φ),
- `U` - total radiation intensity,
- `UTheta` - theta component of radiation intensity,
- `UPhi` - phi component of radiation intensity,
- `UHor` - horizontal component of radiation intensity,
- `UVer` - vertical component of radiation intensity,
- `D` - total directivity,
- `DTheta` - theta component of directivity,
- `DPhi` - phi component of directivity,
- `DHor` - horizontal component of directivity,
- `DVer` - vertical component of directivity,
- `Prad` - radiated power,
- `plane` - plane, in which is near-field computed,
- `uPoints` - vector of the first axis coordinates for near-field computation,
- `vPoints` - vector of the second axis coordinates for near-field computation,
- `distance` - distance of near-field computational plane from the origin,
- `E` - electric near-field computed in points defined by plane, distance, uPoints and vPoints,
- `H` - magnetic near-field computed in points defined by plane, distance, uPoints and vPoints,
- `eigNum` - eigennumbers,
- `charAngle` - characteristic angles,
- `zInActive` - active input impedance,
- `S11` - reflection coefficient,
- `QZ` - quality factor Q_{Z},
- `QFBW` - quality factor Q_{FBW},
- `RCS` - radar cross section,
- `RCStheta` - theta angle for RCS computation,
- `RCSphi` - phi angle for RCS computatio,
- `decomposition` - decomposition of given current into spherical harmonics using **S** matrix,

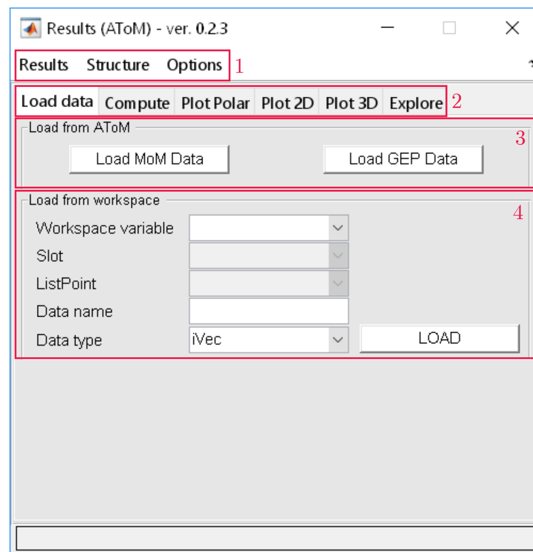


Figure 7.2: Graphical user interface of Resultg - Load tab

- `indexMatrix` - matrix used to identify spherical harmonics participating in current composition,
- `V` - voltage distribution over edge (only for `BEM` see Section 5.1),
- `BEMRadPattern` - radiation patter computed from data given by `BEM`,

7.3.2 GUI

Fig. 7.2 shows the basic layout of `Results`' GUI. Menu Results contains entries `Save`, `Load` and `Quit`, which allows to save and load data or quit Results respectively. Structure menu allows to manage internal repository and for this reason contains entries `Create Slot`, `Create ListPoint`, `Remove Slot` and `Remove ListPoint`. Last menu Options contains items `Load options` and `Load template` which allows to load user pre-sets of graphical behaviour or important user values.

Box 2 of Fig. 7.2 contains six special tabs which are used for following actions:

- Load data - data loading,
- Compute - data pre-calculating,
- Plot Polar - plotting results into polar plot,
- Plot 2D - plotting data into two dimensional Cartesian coordinates,
- Plot 3D - plotting data into three dimensional Cartesian coordinates,
- Explore - going through the results.

Listing 7.6: Commands which can be used to open and close GUI.

```
% open GUI
atomresults.open;

% close GUI
atomresults.close;
```

Load data tab

Load tab, see Fig. 7.2, gives two possibilities how to load data into Results, directly (automatically) from AToM (see Box 3) or manually from Matlab workspace (see Box 4). Commands for data

loading directly are shown in Listing 7.7.

Automatic loading proceeds in the following way:

1. creation of Slot which is named '\AToMProjectName_\SolverAbbreviation\' (Solver abbreviations are: MoM1D, MoM2D, GEP or BEM),
2. creation of all ListPoints inside before-created Slot, these ListPoints are named 'f_\frequency number\'
3. creation of data fields inside each ListPoint, there can be one or more data fields depending on solver (one for MoMs and BEM, more for GEP - depending on number of modes),
4. inserting data into ListPoints.

Listing 7.7: Example of loading data to Results from command line.

```
% Loading data with pre-connected atom
atomresults.loadMoM();
atomresults.loadGEP();
atomresults.loadBEM();

% Loading data without pre-connected atom
atomresults.loadMoM(atomReference);
atomresults.loadGEP(atomReference);
atomresults.loadBEM(atomReference);
```

On the contrary, manual loading procedure is completely user-based, it means the user is responsible for all steps of loading.

- creation repository structure (Slot and ListPoints) and insert all informations about object under explore, mesh eventually basis functions,
- load data via panel "Load from workspace"(see Fig. 7.2 Box 4) or directly from MATLAB command line.

Listing 7.4 shows all commands intended for repository structure management and Listing 7.8 shows all functions which can be used for importing data into Results.

Listing 7.8: List of commands intended for data loading.

```
% insert mesh and basis functions
atomresults.insertMesh('SlotName', meshStructure);
atomresults.insertBasisFcns('SlotName', BasisFunctionsStructure);

% insert data
AToM.insertData('dataName', 'dataType', data, 'ListPointName', 'SlotName');
```

Compute tab

This second tab (see Fig. 7.3) is intended for pre-calculation of results but for next visualisation is not necessary to use it. However, there are at least three cases in which is this tab important:

- User wants only compute data without visualisation.
- User wants to speed up next visualisation (without pre-calculation must be data calculated just before visualisation).
- User wants to change some of the parameters which affects the calculation (see. Fig. 7.3 Box 2).

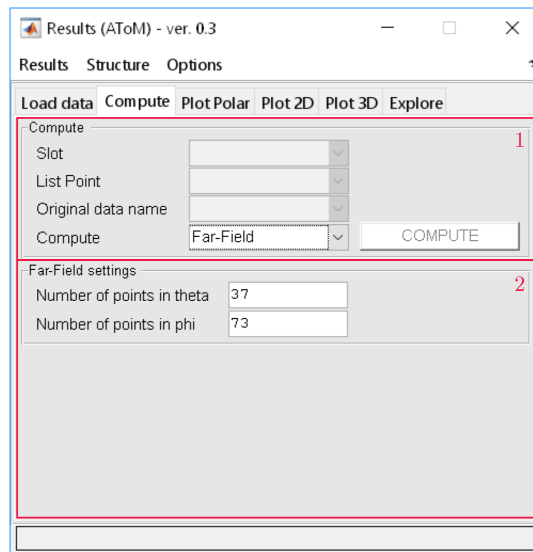


Figure 7.3: Results GUI - Compute tab

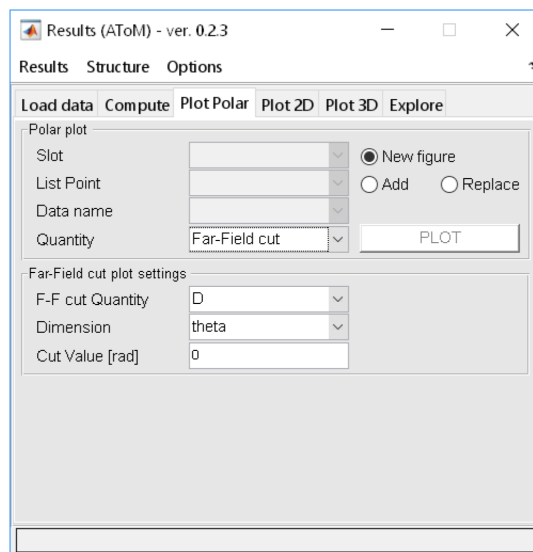


Figure 7.4: Results GUI - Plot polar tab

Compute tab consists of two main parts, first part (see Fig. 7.3 Box 1) is used to select the data for which should be new quantity computed. Second part of the window (see Box 2) contains special panel which is based on requested quantity. Computational parameters can be changed using this panel.

Plot tabs

Third, fourth and fifth tab, specifically Plot Polar, Plot 2D and Plot 3D, are responsible for data visualisation. Structure of these tabs is similar with Compute tab. it is divided into two parts, upper part is used to specify data which will be plotted and lower part contains special panel which can modify resultant figure.

Plot polar tab (see Fig. 7.4) allows user generate figures with polar axes. These quantities can be plotted in polar plot:

- Far-field cut

Plot 2D tab (see Fig. 7.5) is used to generate figures with two dimensional Cartesian coordinates. These quantities can be plotted into 2D Cartesian axes:

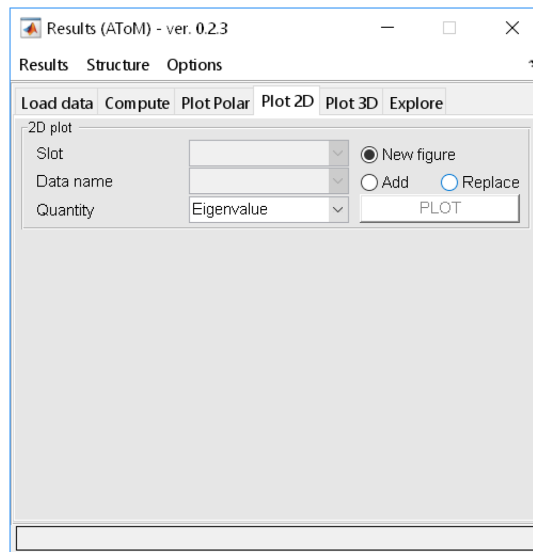


Figure 7.5: Results GUI - Plot 2D tab

- Characteristic angle
- Eigennumber
- Input impedance
- Reflection coefficient
- Q_z
- Q_{FBW}
- Radar cross section (RCS)
- Current decomposition

Plot 3D tab (see Fig. 7.6) is destined to generate figures with three dimensional Cartesian coordinates. These quantities can be plotted into 3D Cartesian axes:

- Current
- Charge
- Far-Field
- Near-Field
- Mesh

All these three tabs also allows to control where the new plot will be placed. There are following choices

- New figure – plot will be in new figure,
- Add – new plot will be added into current figure,
- Replace – new plot will replace present plot in current figure.

Using this feature it is possible to put more 'layers' into one figure and create for example figure as it is depicted in Fig. 7.7.

Explore tab

Explore tab (see Fig. 7.8) is special tool which serves to browse computed results and make it easier to analyse them. To use this tool the figure with analysed quantity must be created. After that the

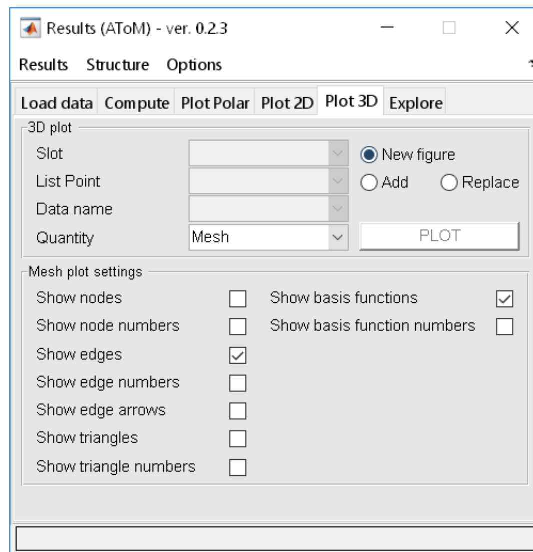


Figure 7.6: Results GUI - Plot 3D tab

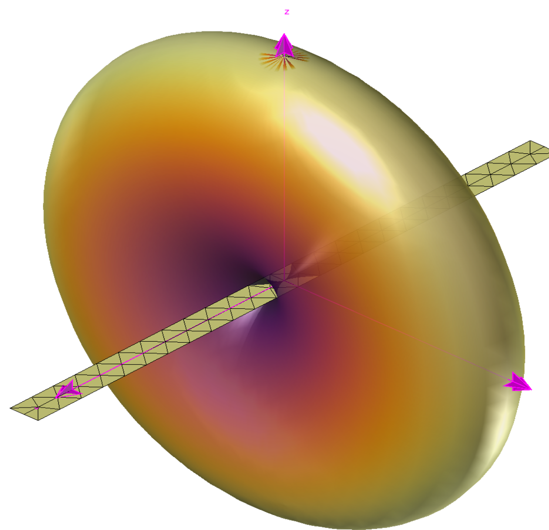


Figure 7.7: Example of combination more layers in one figure

Explore tab is activated and allows to browse the Slots, ListPoints and also different data defined by DataName and see corresponding results.

For comfortable use it is recommended to pre-calculate (see part Compute tab in this Section) all data which user plans to analyse. Otherwise the usage of the Explore tab can be limited by the time necessary to prepare new data.

7.4 Templates and other user preferences

7.4.1 Templates

The appearance of the displayed graphs can be affected by a personal set of graphical styles, which are stored in the template. This file contains all graphical presets which are used inside the plotting functions (colors, line styles, fonts, etc.). Templates must be stored in `results.templates`. In the following box is shown a short preview of the part of the default template.

Listing 7.9: Example of template definition.

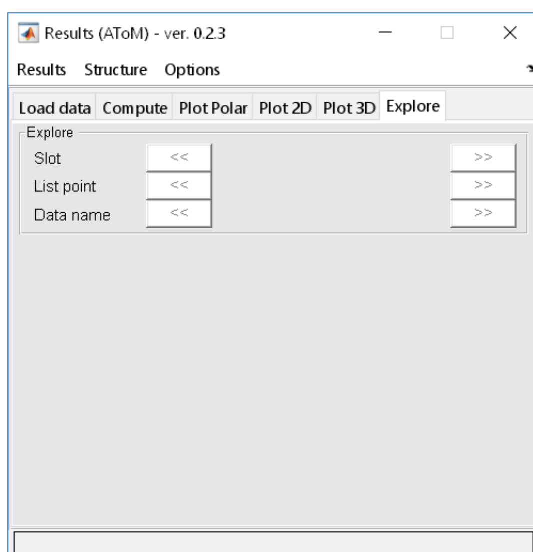


Figure 7.8: Results GUI - Explore tab

```
% triangles
template.triangles.faceColor                       = paired12(list.brownLight,:);
template.triangles.faceAlpha                       = 1;
template.triangles.lineColor                       = atom.b1;
template.triangles.lineStyle                       = '-';
template.triangles.lineWidth                       = 0.5;
template.triangles.lineAlpha                       = 1;
```

An important feature of templates is the necessity to define in the file only differences from the original (default) template.

7.4.2 Standardize figure and user profile

This feature is intended to normalise figure (size, position of legend, grid, etc.), especially to be used before saving or exporting pictures. The list of preferences which are used is saved in `results.figureProfiles`.

standardizeFigure

`standardizeFigure` **more**

`results.standardizeFigure(handles)`

Function standardizeFigure is used as standardization function for generated figure. User pre-defined profile can be used to normalize all figures to the same graphical appearance.

Chapter 8

Utilities

8.1 Matrix S

S matrix [?] is special way how to compute resistance matrix **R**

$$\mathbf{R} = \mathbf{S}^{\mathrm{T}}\mathbf{S}, \tag{8.1}$$

with the particular elements of the **S** matrix are

$$S_{\alpha p} = k\sqrt{Z_0} \int\limits_{\Omega} \boldsymbol{\psi}_p (\boldsymbol{r}) \cdot \mathbf{u}_{\alpha}^{(1)} (k\boldsymbol{r}) \, \mathrm{d}S. \tag{8.2}$$

For more details see [?].

Listing 8.1 shows basic example how to create **S** matrix. This matrix can be also used in GEP (see 6.6.4).

Listing 8.1: Example of the **S** matrix creation.

```
% create S matrix
S = models.matrixOperators.SMatrix.computeS(mesh, basisFcns, frequency)
```

8.2 Subregion matrix C

Subregion matrix **C** with properties

$$\mathbf{Z}^{\mathrm{S}} = \mathbf{C}^{\mathrm{T}}\mathbf{Z}\mathbf{C}, \tag{8.3}$$

$$\boldsymbol{I}^{\mathrm{S}} = \mathbf{C}^{\mathrm{T}}\boldsymbol{I} \tag{8.4}$$

$$\boldsymbol{I} = \mathbf{C}\boldsymbol{I}^{\mathrm{S}} \tag{8.5}$$

where $\mathbf{Z}$ is impedance matrix of full system, $\mathbf{Z}^{\mathrm{S}}$ is impedance matrix of reduced system, $\boldsymbol{I}$ are the current density coefficients related to $\mathbf{Z}$ and $\boldsymbol{I}^{\mathrm{S}}$ are the current density coefficients related to $\mathbf{Z}^{\mathrm{S}}$, provides possibility to reduce the system and create virtual holes into structure without its modification.

The idea of this matrix is based on elimination of specific rows and columns of the impedance matrix which leads to fictive removal of part of the structure. Fig. 8.1 illustrates influence of subregion matrix to the computation and Listing 8.2 shows how can be this matrix created.

Listing 8.2: Example of the **C** matrix creation.

```
% define polygons bounding removed parts
polygon1 = [x1 y1; x2 y2; x3 y3; x1 y1];
polygon1 = [x4 y4; x5 y5; x6 y6; x4 y4];
polygons = {polygon1, polygon2};
```

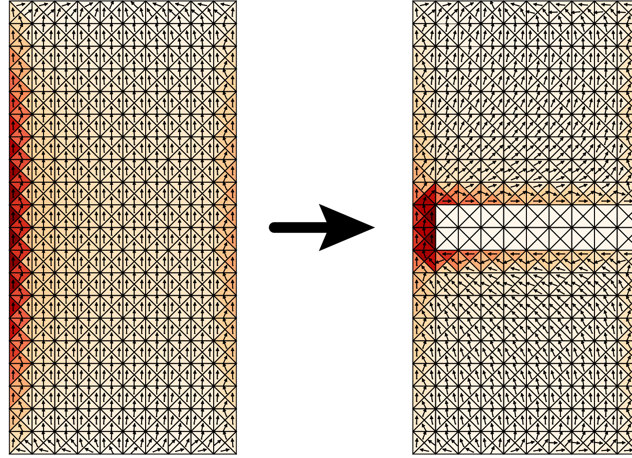


Figure 8.1: Example how the **C** matrix affects the structure.

```
% create subregion matrix C
[C, basisFcnsOrder, newBasisFcns] = ...
    models.utilities.subregionMatrices.computeCMat(mesh, basisFcns, polygons)
```

8.3 Symmetry matrices

If the structure has a symmetry, the symmetry reflection can be used to reduce size of symmetric quantities. There exist rectangular matrix $\mathbf{C}_\mathrm{r}$ which reduce, for example,

$$\hat{\mathbf{Z}} = \mathbf{C}_\mathrm{r}^\mathrm{T} \mathbf{Z} \mathbf{C}_\mathrm{r}, \tag{8.6}$$

$$\hat{\mathbf{I}} = \mathbf{C}_\mathrm{r}^\mathrm{T} \mathbf{I}, \tag{8.7}$$

$$\mathbf{I} = \mathbf{C}_\mathrm{r} \hat{\mathbf{I}}, \tag{8.8}$$

where $\mathbf{Z}$ is full impedance matrix and $\hat{\mathbf{Z}}$ is reduced impedance matrix.

8.3.1 Nomenclature

- `Cr` – Rectangular matrix $\mathbf{C}_r$ for given transform matrix $\mathbf{T}$.
- `C` – Square matrix denoting on which basis function is reflected to other basis function after applying transform matrix $\mathbf{T}$. Each row can contain only one non-zero element ± 1.
- `eBF` – Vector of logical value where is marked if each basis function have to be computed or it can be reached by applying symmetry matrix $\mathbf{C}_r$. (Note, that for pair of mirroring basis functions is one of them marked as "elementary", *i.e.* `true` and the second one is marked as "mirrored", *i.e.* `false`). One can see `Z == Cr.^T * Z(eBF, eBF)* Cr` in case `Z` has symmetry the `Cr` was computed for.
- `T` – 3D transform matrix of size `[3 x 3]`.
- `parityMatrix` – A diagonal matrix which distinguish PEC and PMC type of mirroring.
- `symmetry` – Symmetry selection:
  - `'E'` – Identity
  - `'I'` – Inverse
  - `'C:aaa:b'` – Rotation by `2pi/b` around plane with normal vector defined by `aaa`:

  * aaa = 'XY' or aaa = 'Z' for basic symmetry plane with normal vector [0 0 1].
  * aaa = 'XZ' or aaa = 'Y' for basic symmetry plane with normal vector [0 1 0].
  * aaa = 'YZ' or aaa = 'X' for basic symmetry plane with normal vector [1 0 0].
  * aaa = 'A,B,C', where vector n = [A B C] is a normal vector.

- 'Sig:aaa:ccc' – reflection, where:

  * aaa is the same as above.
  * ccc = 'PEC' or ccc = 'PMC' for select type of symmetry plane.

- 'S:aaa:b' – rotary-reflection by 2pi/b around plane with normal vector defined by aaa. It is defined as T = T1 * T2, where T1 is computed by rotation matrix 'C:aaa:b' and T2 is computed by reflection matrix Sig:aaa:PMC.
- Examples:

  * symmetry = 'C:Z:2' – Rotation about π around z-axis.
  * symmetry = 'Sig:XY:PEC' – PEC reflection in z-axis.
  * symmetry = 'Sig:1,-1,1:PMC' – PMC reflection in given plane.

8.3.2 Public functions

All followed functions are located in namespace +models/+utilities/+symmetryMatrices. Complete syntax is outputs = models.utilities.symmetryMatrices(inputs).

aggregateTwoC
: [Cr_tot, eBF_tot] = aggregateTwoC(Cr1, eBF1, Cr2, eBF2)
: Aggregate two Cr matrices. Matrix C fulfill transform operation $\mathbf{T} = \mathbf{T}_1\mathbf{T}_2$.

applyPEC
: C = applyPEC(C, parityMatrix)
: Apply PEC reflection to matrix C.

applyPMC
: C = applyPMC(C, parityMatrix)
: Apply PMC reflection to matrix C.

CMatrixSymmetryPlanes
: [Cr_tot, eBF_tot] = CMatrixSymmetryPlanes(mesh, symmetriesMoM, basisFcns)
: Used in MoM Solver, where symmetriesMoM is a vector [3 x 1] corresponding to [XY, ... XZ, XY] symmetry planes with values 1 for none symmetry, 2 for PEC and 3 for PMC.

computeCMat
: [isSymmetric, C, parityMatrix] = computeCMat(mesh, symmetry, basisFcns)
: Compute C matrix for given symmetry.

computeCMatTotal
: [Cr_total, eBF_tot] = computeCMatTotal(mesh, symmetries, basisFcns)
: Compute total coupling matrices for list of symmetries given as symmetries = 'symmetry1; ... symmetry2; symmetry3'.

identityMatrix
: E = identityMatrix()
: Create transform matrix for identity.

inverseMatrix
: I = inverseMatrix()
: Create transform matrix for inversion.

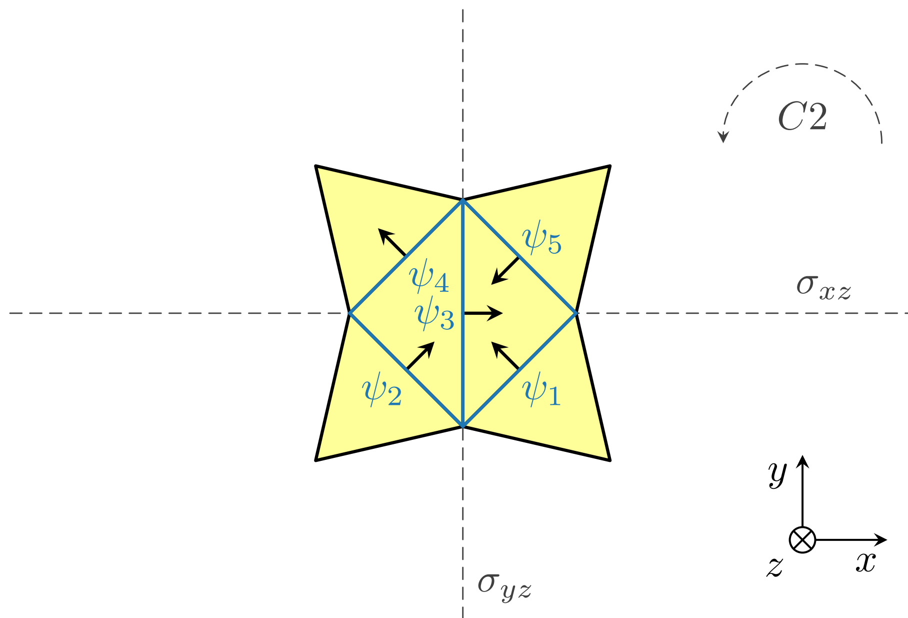


Figure 8.2: Simple symmetric structure.

normalizeCr

`Cr = normalizeCr()`

Normalize `Cr`

reduceC

`[Cr, eBF] = reduceC(C)`

Reduce square matrix `C` to rectangular matrix `Cr` and provide vector `eBF`.

reflectionMatrix

`P = reflectionMatrix(n)`

Create transform matrix for reflection. Input `n` is normal to mirror plane.

rotationMatrix

`R = rotationMatrix(u, theta)`

Create transform matrix for rotation around unit vector `u` by angle `theta`.

symmetryCheck

`[isSymmetric, C, parityMatrix] = symmetryCheck(mesh, basisFcns, T)`

Check if given mesh has symmetry described by transform matrix `T`. Output is logical value `isSymmetric` and in positive case corresponding matrices `C` and `parityMatrix`.

8.3.3 Example

Let us demonstrate a principle of symmetry matrices on simply structure shown in Fig. 8.2. This structure can be generated by code in Listing 8.3, this structure has three types of symmetry: rotation about π (C2) and two reflections (σ_{xz} and σ_{yz}).

Code in Listing 8.4 shows two ways, how to get matrices $\mathbf{C}_\mathrm{r}$ for each symmetry type separately, Listing 8.5 shows how to combine two symmetry or all symmetry types.

Listing 8.3: Generate simple symmetric structure.

```
% start atom, create project
atom = Atom.start(false);
atom.createProject('symmetryMatrices');
% create polygon, generate mesh
atom.selectedProject.geom.addPolygon(['[1 0 0; 1.5 1.5 0; 0 1 0; -1.5 1.5 0;'...
   '-1 0 0; -1.5 -1.5 0; 0 -1 0; 1.5 -1.5 0]'], 'Polygon1');
atom.selectedProject.mesh.setElementSizeFromFrequency(false);
```

```
atom.selectedProject.mesh.setGlobalMeshDensity('2');
atom.selectedProject.mesh.getMesh();
% get mesh and basis functions
mesh = atom.selectedProject.mesh.getMeshData2D;
basisFcns = models.solvers.MoM2D.basisFcns.getBasisFcns(mesh);
```

Listing 8.4: Generate symmetry matrices for each symmetry type.

```
%% Reflection YZ
% prepare and compute C matrix
T = models.utilities.symmetryMatrices.reflectionMatrix([1 0 0]);
[isSymmetric1, C1, parityMatrix1] = ...
    models.utilities.symmetryMatrices.symmetryCheck(mesh, basisFcns, T);
C1 = models.utilities.symmetryMatrices.applyPMC(C1, parityMatrix1);
% OR do it in one command
[isSymmetric1a, C1a, eBF1a] = ...
    models.utilities.symmetryMatrices.computeCMat(mesh,'Sig:X:PMC');

% reduce Cr and normalize Cr matrix
[Cr1, eBF1] = models.utilities.symmetryMatrices.reduceC(C1);
Cr1 = models.utilities.symmetryMatrices.normalizeCr(Cr1);

% OR do all commands above in one line
[Cr1a, eBF1a] = models.utilities.symmetryMatrices.computeCMatTotal(mesh, ...
    'Sig:X:PMC', basisFcns);

%% Reflection XY
[Cr2, eBF2] = models.utilities.symmetryMatrices.computeCMatTotal(mesh, ...
    'Sig:Y:PMC', basisFcns);

%% Rotation C2 arout Z axis
[Cr3, eBF3] = models.utilities.symmetryMatrices.computeCMatTotal(mesh, 'C:Z:2', ...
    basisFcns);
```

Listing 8.5: Generate symmetry matrices fulfilling all symmetries of structure.

```
[Cr12, eBF12] = models.utilities.symmetryMatrices.aggregateTwoC(Cr1, eBF1, Cr2, eBF2);
[CrALL, eBFALL] = models.utilities.symmetryMatrices.aggregateTwoC(Cr12, eBF12, ...
    Cr3, eBF3)

%% OR do everything in one command
[CrALLa, eBFALLa] = models.utilities.symmetryMatrices.computeCMatTotal(mesh, ...
    'Sig:X:PMC; Sig:Y:PMC; C:Z:2', basisFcns)
```

Chapter 9

Graphical User Interface (GUI)

First of all it is necessary to note, that ATоM is originally meant to be controlled via command line and GUI is just superstructure, which provide graphical interface to some (not all) functions of ATоM. GUI consists of several individual viewers, which control one or more ATоM models. When starting ATоM with command `atom = Atom.start` or `atom = Atom.start(true)`, ATоM will be started with opened GUI. To start ATоM without GUI is performed by `atom ... = Atom.start(false)`.

9.1 Viewers with separated figure

These viewers shows the most important parts of ATоM projects and it is suitable to have them in separated figures to be possible show them in all cases. Viewers with separated figure are

- Main Viewer - main figure of ATоM GUI which contains several tabbed panels integrating another viewers. See Fig. 9.1. This viewer is shown right after creating ATоM with GUI.
- Workspace Viewer - GUI for `Workspace` model. See Fig. 1.1. This viewer is possible to open from Main Viewer by clicking on "Workspace" button.
- History Viewer - GUI for `History` model. This viewer is possible to open from Main Viewer by clicking on "History" button.
- Design Viewer - GUI for `Geom`, `Mesh`, `Physics` and `Feeding` models. Section 9.5. This viewer is possible to open from Main Viewer by clicking on "Desing" button.

These viewers is possible to simply close and open also from Command Window by calling method `open` and `close`. For example when ATоM is started without GUI (`atom = ... Atom.start(false);`) it is possible to open, *e.g.*, Main Viewer by calling `atom.gui.mainViewer.open`. On the other hand, when all viewers are opened it is possible to close, *e.g.*, Workspace Viewer by calling `atom.gui.workspaceViewer.close`.

9.1.1 Public Methods

`close`

```
gui.designViewer.close; gui.historyViewer.close; gui.mainViewer.close; ...
gui.workspaceViewer.close;
```

Close figure of relevant viewer.

`open`

```
gui.designViewer.open; gui.historyViewer.open; gui.mainViewer.open; ...
gui.workspaceViewer.open;
```

Open figure of relevant viewer.

`setPosition`

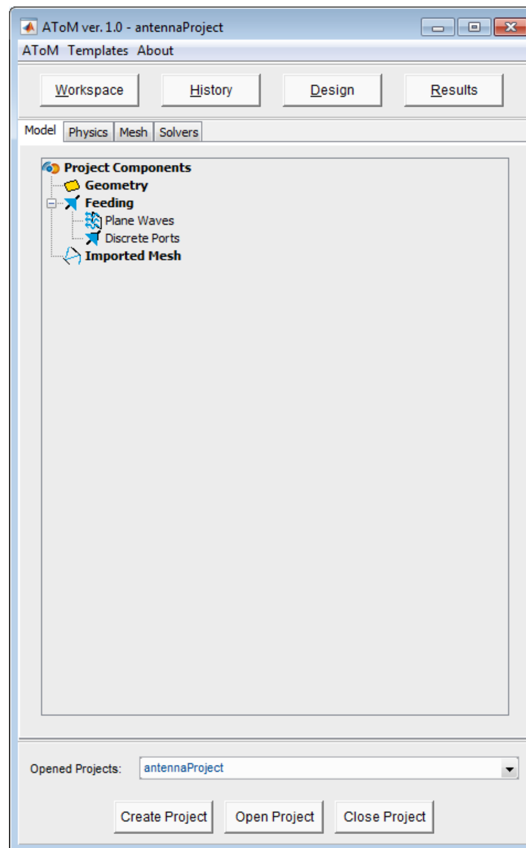


Figure 9.1: Main Viewer. Model List Viewer is the tree representing simulation model, Project Viewer is on the bottom part of figure.

```
gui.designViewer.setPosition(viewerPos); gui.historyViewer.setPosition(viewerPos); ...
gui.mainViewer.setPosition(viewerPos); gui.workspaceViewer.setPosition(viewerPos);
```

Allows to set precise position of relevant viewer on screen. Input argument `viewerPos` has meaning `[left bottom width height]`, which is usual in MATLAB graphics.

9.2 Viewers integrated in Main Viewer

Viewers directly integrated in Main Viewer are

- Project Viewer - on the bottom of Main Viewer. Allows selection, creation, opening and closing of projects. This viewer is accessible whatever tab in Main Viewer is selected.
- Model List Viewer - on tab "Model". Contain tree representing simulation model including created geometry entities, feeding and imported meshes.
- Physics Viewer - on tab "Physics". Allows controlling of `Physics` model.
- Mesh Viewer - on tab "Mesh". Allows controlling of `Mesh` model.
- Solvers Viewer - on tab "Solvers". Include tabs "MoM1D", "MoM2D", "GEP" and "BEM" where is possible to define properties of individual solvers and perform simulation.

These viewers has no accessible references because its appearance is fully managed by Main Viewer itself.

9.3 Color Templates

AToM GUI allows to use color templates and hence adjust appearance of whole GUI. AToM is distributed with two templates: `defaultTemplate.m` and `darkTemplate.m`. These templates are

placed in `+interface\+templates\` folder and user defined template can be simply defined using these default templates. Switching between templates is possible via Main Viewer (Fig. 9.1) in menu "Templates". To reset list of templates in menu "Templates" it is necessary to close Main Viewer and open it again. By default, during the start of GUI `interface.templates.defaultTemplate` is always loaded. If change of default template is needed, original file `defaultTemplate.m` has to be rewritten (modified).

9.4 Class GUI

All viewers are managed by class `GUI` and it is possible to access its public methods for general viewers treatment. In singleton `atom` instance GUI exists just once and is common to all projects. Individual projects does not contain any information about way of displaying its content in GUI. Instance of GUI class is accessible via Command Window by `atom.gui`.

9.4.1 Public Methods

`applyTemplate`

`gui.applyTemplate('templateName')`

Apply color template located in `+interface\+templates\` to all ATOM viewers. Template file has to be function `templateName.m` returning `struct` variable in correct form with correct data (Section 9.3).

`close`

`gui.close`

Close all opened viewers. Even without GUI ATOM is still running and it is possible to control it via Command Line. If reference to ATOM object does not exists (ATOM was created by `Atom.start;`) it can be created by `atom = Atom.start;`. Returned `atom` variable is reference to already running ATOM session.

`open`

`gui.open`

Opens all ATOM viewers. When no project active, just Main Viewer is opened.

9.4.2 Public Properties

`designViewer`

Reference to Design Viewer.

`historyViewer`

Reference to History Viewer.

`mainViewer`

Reference to Main Viewer.

`workspaceViewer`

Reference to Workspace Viewer.

9.4.3 Preferences

These preferences can be set using method `atom.gui.setProperties('prop1Name', prop1Value, ... 'prop2Name', prop2Value, ..)`. Default values are shown in brackets {}.

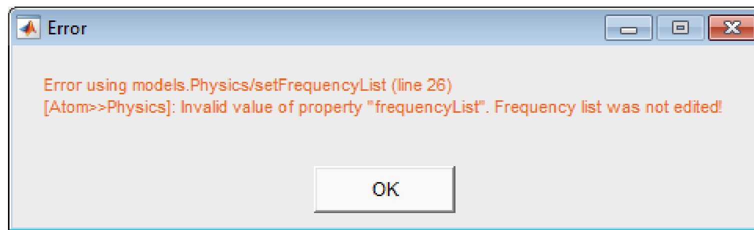


Figure 9.2: Example of error dialogue which is shown every time when invalid step is made when working from GUI. Throwing error to Command Window depends on property `throwErrorToCommandWindow` of GUI.

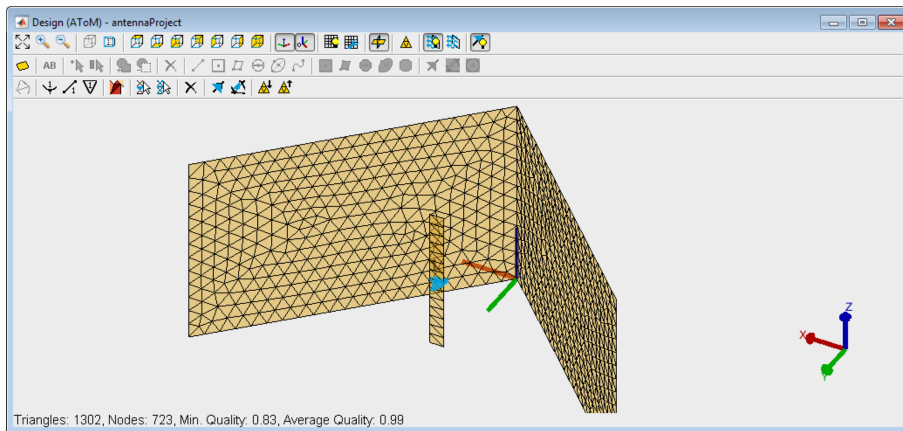


Figure 9.3: Design Viewer (DW) with mesh structure and one discrete port.

`throwErrorToCommandWindow`

`Logical: {true} | false`

Determine if thrown error because of invalid step when working with GUI will be shown also in Command Window. Small window with error stack is always shown (Fig. 9.3).

9.5 Design Viewer

Design Viewer is the most complex viewer in AToM and allows interactively create simulation task. Design Viewer is shown in Fig. 9.5. In the lower part of DW is status bar, where are shown some basic instructions for selected interactive tool, or statistics of mesh. In the upper part are buttons with miscellaneous functions. Every button has tool tip string with function description and its functionality should be clear for people from Computational Electromagnetics discipline. These buttons are logically separated into three rows. The upper row is mention to be for DW display settings like zoom, projection, view direction, coordinate systems, working plane, showing of symmetry planes, plane waves, discrete ports, adding plane waves and computing mesh from geometry. The middle row is mentioned to be for geometry appearance, creating, editing and creating ports. The lower row is mentioned to be for mesh appearance, editing, loading and saving.

9.5.1 Public Methods

`close`

`gui.designViewer.close`

Close figure with DesignViewer.

`copy`

`gui.designViewer.copy`

Copy axes of Design Viewer with all child objects into separate figure for advanced user defined processing.

fitWiew

`gui.designViewer.fitWiew`

Fit whole shown scene to fit into Design Viewer axes. There exist also keyboard short cut Space for this method.

hideCoordinates

`gui.designViewer.hideCoordinates`

Hide coordinate cross in the lower-right corner of Design Viewer.

hideDiscretePorts

`gui.designViewer.hideDiscretePorts`

Hide all discrete ports in shown scene. Even newly defined ports will be hidden.

hideLabelsOfGeometryObjects

`gui.designViewer.hideLabelsOfGeometryObjects`

Hide labels of all geometry objects.

hideMeshQuality

`gui.designViewer.hideMeshQuality`

Hide colormap over mesh representing quality of mesh.

hideNumbersOfMeshEdges

`gui.designViewer.hideNumbersOfMeshEdges`

Hide numbers of edges of mesh.

hideNumbersOfMeshNodes

`gui.designViewer.hideNumbersOfMeshNodes`

Hide numbers of nodes of mesh.

hideNumbersOfMeshTriangles

`gui.designViewer.hideNumbersOfMeshTriangles`

Hide numbers of triangles of 2D mesh.

hideOrigin

`gui.designViewer.hideOrigin`

Hide origin of coordinate system.

hidePlaneWaves

`gui.designViewer.hidePlaneWaves`

Hide all plane plane waves. Even newly defined waves will be hidden.

hideSymmetryPlanes

`gui.designViewer.hideSymmetryPlanes`

Hide active symmetry planes. Even newly defined planes will be hidden.

hideWorkingPlane

`gui.designViewer.hideWorkingPlane`

Hide working plane.

open

`gui.designViewer.open`

Open figure with Design Viewer.

setPosition

`gui.designViewer.setPosition(newFigurePosition)`

Allows to set precise position of Design Viewer on screen. Input argument `newFigurePosition` has meaning `[left bottom width height]`, which is usual in MATLAB graphics.

setProjection

`gui.designViewer.setProjection(newProjection)`

Set projection to axes of Design Viewer. `newProjection` can be `'orthographic'` or `'perspective'`.

setView

`gui.designViewer.setView(cameraTarget, cameraPosition, cameraViewAngle)`

This method set properties of camera in Design Viewer. `cameraTarget` is target (center of view) of camera as numeric vector $[1 \times 3]$ with $[x, y, z]$ coordinates, `cameraPosition` is global position of camera also as numeric vector $[1 \times 3]$ and `cameraViewAngle` is angle of view of camera in degrees.

setViewDirection

`gui.designViewer.setViewDirection(newDirection)`

Change direction of view. `newDirection` can be `'perspective'`, `'front'`, `'back'`, `'top'`, `'bottom'`, `'left'` and `'right'`.

showCoordinates

`gui.designViewer.showCoordinates`

Show coordinate cross in the lower-right corner of Design Viewer.

showDiscretePorts

`gui.designViewer.showDiscretePorts`

Show all discrete ports in shown scene.

showLabelsOfGeometryObjects

`gui.designViewer.showLabelsOfGeometryObjects`

Show labels of all geometry objects.

showMeshQuality

`gui.designViewer.showMeshQuality`

Show colormap over mesh representing quality of mesh.

showNumbersOfMeshEdges

`gui.designViewer.showNumbersOfMeshEdges`

Show numbers of edges of mesh.

showNumbersOfMeshNodes

`gui.designViewer.showNumbersOfMeshNodes`

Show numbers of nodes of mesh.

hideNumbersOfMeshTriangles

`gui.designViewer.hideNumbersOfMeshTriangles`

Hide numbers of triangles of 2D mesh.

showOrigin

`gui.designViewer.showOrigin`

Show origin of coordinate system.

showPlaneWaves

`gui.designViewer.showPlaneWaves`

Show all plane plane waves.

showSymmetryPlanes

`gui.designViewer.showSymmetryPlanes`

Show active symmetry planes.

showWorkingPlane

`gui.designViewer.showWorkingPlane`

Show working plane.

zoom

`gui.designViewer.zoom(direction)`

Perform zoom to the center of screen. Input `direction` can be numeric scalar. If is scalar positive, zoom in is performed, if negative zoom out is performed.

Index

ATToM
- Quit, 1
- Start, 1

GEP
- AFSgoal, 54
- AFSinsertedSamplesHandle, 54
- AFSnInsertedSamples, 54
- AFSnIterations, 53
- AFSregionBoundaryBottom, 54
- AFSregionBoundaryTop, 54
- AFSresultsAtGivenSamples, 54
- charAngleBoundaryBottom, 53
- charAngleBoundarySwitch, 53
- charAngleBoundaryTop, 53
- corrDataHndl, 53
- corrTableHndl, 53
- corrType, 53
- delNegValuesOfInputMatrices, 51
- discardComplexLambda, 53
- discardNegativeINI, 53
- eigPostprocessing, 52
- eigPreprocessing, 52
- eigRunPreAndPostprocessing, 52
- innerSolverGetA, 52
- innerSolverGetB, 52
- innerSolverGetN, 52
- innerSolverHndl, 52
- innerSolverSolve, 52
- innerSolver, 52
- matricesStorage, 52
- maxLengthOfGap, 53
- maxMagnEigVal, 53
- minCorrValue, 53
- minModeLength, 53
- nModes, 52
- normConst, 52
- showStatusWindow, 51
- symmetrizeInputMatrices, 51
- verbosity, 51

GUI
- designViewer, 91
- historyViewer, 91
- mainViewer, 91
- throwErrorToCommandWindow, 92
- workspaceViewer, 91

ListPoint
- col, 74
- listValue, 73
- name, 73

MoM1D
- nBatchMax, 40
- nWorkers, 40
- quadOrder, 40
- resultRequests, 40
- resultsInFile, 40
- resultsInRAM, 41
- verbosity, 41
- wireRadius, 41

MoM2D
- nBatchMax, 43
- nWorkers, 43
- quadOrder, 43
- resultRequests, 43
- resultsInFile, 44
- resultsInRAM, 44
- verbosity, 44

Physics
- c0, 29
- ep0, 29
- gamma, 29
- mu0, 29
- symmetryPlanes.(plane).type, 30

Repository
- slot, 73

Results
- options, 73
- repository, 72
- template, 72
- version, 73

Slot
- basisFcns, 73
- data, 74
- domain, 73
- listPoint, 73
- listValues, 73
- mesh1D, 73

mesh2D, 73
meshBEM, 73
name, 73
WorkspaceViewer
columnWidth, 4
Workspace
nSignificantDigitsView, 4
table, 4, 36