pyEPR

Release 0.9.0

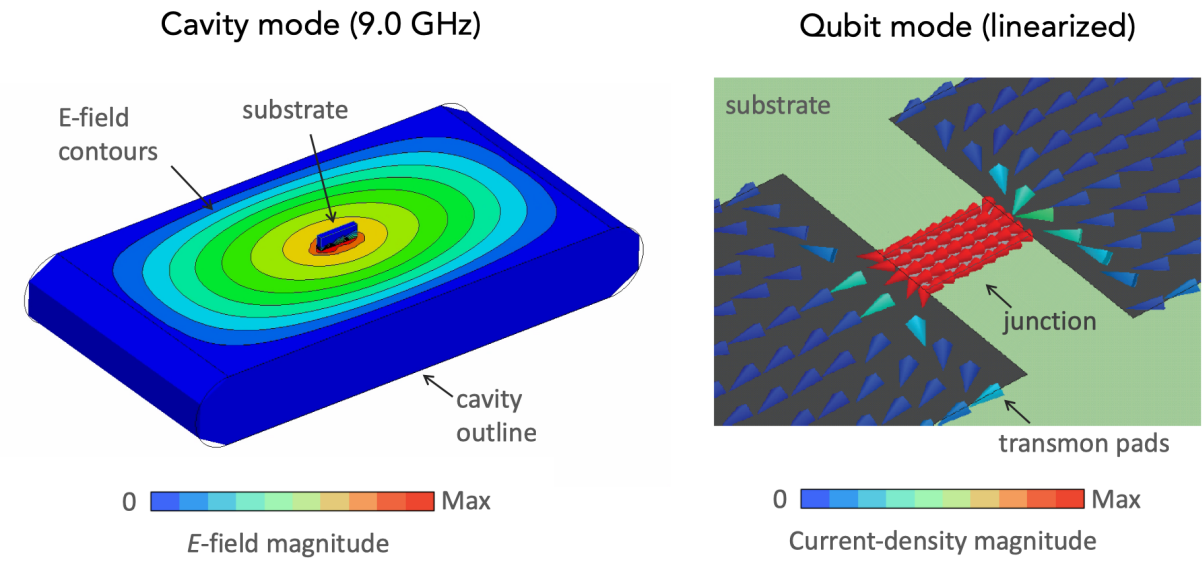
Jun 14, 2023

Contents

CHAPTER 1

Powerful, automated analysis and design of quantum microwave devices

Version: 0.9.0



pyEPR is an open source, BSD-licensed library providing high-efficiency, easy-to-use analysis functions and automation for the design of quantum chips based on superconducting quantum circuits, both distributed and lumped. pyEPR interfaces the classical distributed microwave analysis with that of quantum structures and Hamiltonians. It is chiefly based on the energy participation ratio approach; however, it has since v0.4 extended to cover a broad range of design approaches. pyEPR stradels the analysis from Maxwell's to Schrodinger's equations, and converts the solutions of distributed microwave (typically eigenmode simulations) to a fully diagonalized spectrum of the energy levels, couplings, and key parameters of a many-body quantum Hamiltonian.

pyEPR contains both analytic and numeric solutions.

1.1 Contents

1.1.1 About

Welcome to pyEPR: about :beers:!

Automated Python module for the design and quantization of Josephson quantum circuits

Abstract: Superconducting circuits incorporating non-linear devices, such as Josephson junctions and nanowires, are among the leading platforms for emerging quantum technologies. Promising applications require designing and optimizing circuits with ever-increasing complexity and controlling their dissipative and Hamiltonian parameters to several significant digits. Therefore, there is a growing need for a systematic, simple, and robust approach for precise circuit design, extensible to increased complexity. The energy-participation ratio (EPR) approach presents such an approach to unify the design of dissipation and Hamiltonians around a single concept — the energy participation, a number between zero and one — in a single-step electromagnetic simulation. This markedly reduces the required number of simulations and allows for robust extension to complex systems. The approach is general purpose, derived ab initio, and valid for arbitrary non-linear devices and circuit architectures. Experimental results on a variety of circuit quantum electrodynamics (cQED) devices and architectures, 3D and flip-chip (2.5D), have been demonstrated to exhibit ten percent to percent-level agreement for non-linear coupling and modal Hamiltonian parameters over five-orders of magnitude and across a dozen samples. Here, in this package, all routines of the EPR approach are fully automated.

References

- Z.K. Minev, Z. Leghtas, *et al.* (2020) (arXiv:2010.00620).
- Z.K. Minev, Ph.D. Dissertation, Yale University (2018), see Chapter 4. (arXiv:1902.10355)

1.1.2 Installation

Main installation method

1. **Fork** the **‘‘‘pyEPR top-level repository‘‘‘**_ on GitHub. (How to fork a GitHub repo?). Share some love by **staring** :star: pyEPR.
2. **Clone** your forked repository locally. (How to clone a GitHub repo?). Setup the `pyEPR` python code by following *Installation and Python Setup*.
3. **Tutorials** Learn how to use using the jupyter notebook tutorials
4. **Stay up to date** Enjoy and make sure to git add the master remote branch `git remote add MASTER_MINEV git://github.com/zlatko-minev/pyEPR.git` (help?).
5. **Cite ‘‘pyEPR‘‘** arXiv:2010.00620 and arXiv:1902.10355 enjoy!

Installing locally via pip

In the future, `pyEPR` can be installed using the Python package manager pip.

However, for the moment, we recommend a local developer installation, which allows for fast upgrades. We are still in active development. Perform the steps in the *Main installation method* section. What you could do, once you have the local clone git, is to install pyEPR locally. Navigate to the local root folder of the repo.

First, in bash, upgrade python `pip`

```
python -m pip install -U pip
```

Now we can locally install the pyEPR module.

```
python -m pip install -r requirements.txt -e .
```

Installing via conda

For Python 3.6+, installation via conda is supported since `pyEPR` v.0.8.03, through the `conda-forge` channel. You can download and install `pyEPR` typing in from bash:

```
conda install -c conda-forge pyepr-quantum
```

The prefix `-c conda-forge` is required to activate the optional `conda-forge` channel.

Installing via pip from PyPI

For Python 3.6+, installation via PyPI is supported since `pyEPR` v.0.8. You can download and install `pyEPR` typing in from bash:

```
pip install pyEPR-quantum
```

Note: Note that the name of the recipe on the `conda-forge` channel is `pyepr-quantum`, and on PyPI is `pyEPR-quantum`, as the name *pyepr* was already taken by another project. This does not change anything in the way the library is imported in Python as documented in the guide and examples.

1.1.3 Examples

Start-up quick example

The following code illustrates how to perform a complete analysis of a simple two-qubit, one-cavity device in just a few lines of code with `pyEPR`. In the HFSS file, before running the script, first specify the non-linear junction rectangles and variables (see Sec. pyEPR Project Setup in HFSS). All operations in the eigen analysis and Hamiltonian computation are fully automated. The results are saved, printed, and succinctly plotted.

```
# Load pyEPR. See the tutorial notebooks!
import pyEPR as epr

# 1. Connect to your Ansys, and load your design
pinfo = epr.ProjectInfo(project_path = r'C:\sim_folder',
                        project_name = r'cavity_with_two_qubits',
                        design_name  = r'Alice_Bob')


# 2a. Non-linear (Josephson) junctions
```

(continues on next page)

(continued from previous page)

```
pinfo.junctions['jAlice'] = {'Lj_variable':'Lj_alice', 'rect':'rect_alice', 'line':␣
↪'line_alice', 'Cj_variable':'Cj_alice'}
pinfo.junctions['jBob']   = {'Lj_variable':'Lj_bob',   'rect':'rect_bob',   'line':␣
↪'line_bob', 'Cj_variable':'Cj_bob'}
pinfo.validate_junction_info() # Check that valid names of variables and objects have␣
↪been supplied.

# 2b. Dissipative elements: specify
pinfo.dissipative['dielectrics_bulk']    = ['si_substrate', 'dielectric_object2'] #␣
↪supply names of hfss objects
pinfo.dissipative['dielectric_surfaces'] = ['interface1', 'interface2']
# Alternatively, these could be specified in ProjectInfo with
# pinfo = epr.ProjectInfo(..., dielectrics_bulk = ['si_substrate', 'dielectric_object2
↪'])

# 3.  Perform microwave analysis on eigenmode solutions
eprd = epr.DistributedAnalysis(pinfo)
swp_var = 'Lj_alice' # Sweep variable from optimetric analysis that should be used on␣
↪the x axis for the frequency plot
eprd.quick_plot_frequencies(swp_var) # plot the solved frequencies before the analysis
eprd.hfss_report_full_convergence() # report convergence
eprd.do_EPR_analysis()

# 4a.  Perform Hamiltonian spectrum post-analysis, building on mw solutions using EPR
epra = epr.QuantumAnalysis(eprd.data_filename)
epra.analyze_all_variations(cos_trunc = 8, fock_trunc = 7)

# 4b. Report solved results
swp_variable = 'Lj_alice' # suppose we swept an optimetric analysis vs. inductance Lj_
↪alice
epra.plot_hamiltonian_results(swp_variable=swp_variable)
epra.report_results(swp_variable=swp_variable, numeric=True)
epra.quick_plot_mode(0,0,1,numeric=True, swp_variable=swp_variable)
```

Video tutorials

Jupyter notebooks and example Ansys files

The most extensive way to learn pyEPR is to look over the pyEPR example notebooks and example files contained in the package repo.

The Jupyter notebooks can be viewed in a web browser directly here.

1.1.4 Main classes

The first main class of pyEPR is *ProjectInfo*, which instantiates and stores the Ansys interfaces classes and user-defined parameters related to the design, such as junction names and properties.

The second main class of pyEPR is *DistributedAnalysis*, which performs the EPR analysis on the ansys eigenfield solutions from the fields. It saves the calculated energy participation ratios (EPRs) and related convergences, and other paramete results. It does not calculate the Hamiltonian. This is left for the third class.

The third main class of pyEPR is *QuantumAnalysis*, which uses the EPRs and other save quantities to create and diagonalize the Hamiltonian.

ProjectInfo

class pyEPR.project_info.**ProjectInfo**(*project_path: str = None*, *project_name: str = None*, *design_name: str = None*, *setup_name: str = None*, *dielectrics_bulk: list = None*, *dielectric_surfaces: list = None*, *resistive_surfaces: list = None*, *seams: list = None*, *do_connect: bool = True*)

Bases: `object`

Primary class to store interface information between `pyEPR` and `Ansys`.

- **Ansys:** stores and provides easy access to the ansys interface classes *pyEPR.ansys.HfssApp*, *pyEPR.ansys.HfssDesktop*, *pyEPR.ansys.HfssProject*, *pyEPR.ansys.HfssDesign*, *pyEPR.ansys.HfssSetup* (which, if present could nbe a subclass, such as a driven modal setup *pyEPR.ansys.HfssDMSetup*, eigenmode *pyEPR.ansys.HfssEMSetup*, or Q3D *pyEPR.ansys.AnsysQ3DSetup*), the 3D modeler to design geometry *pyEPR.ansys.HfssModeler*.
- **Junctions:** The class stores params about the design that the user puts will use, such as the names and properties of the junctions, such as which rectangle and line is associated with which junction.

Note: Junction parameters. The junction parameters are stored in the `self.junctions` ordered dictionary

A Josephson tunnel junction has to have its parameters specified here for the analysis. Each junction is given a name and is specified by a dictionary. It has the following properties:

- **`Lj_variable` (str):** Name of HFSS variable that specifies junction inductance Lj defined on the boundary condition in HFSS. WARNING: DO NOT USE Global names that start with $.
- **`rect` (str):** String of Ansys name of the rectangle on which the lumped boundary condition is defined.
- **`line` (str):** Name of HFSS polyline which spans the length of the rectangle. Used to define the voltage across the junction. Used to define the current orientation for each junction. Used to define sign of ZPF.
- **`length` (str):** Length in HFSS of the junction rectangle and line (specified in meters). To create, you can use `epr.parse_units('100um')`.
- **`Cj_variable` (str, optional) [experimental]:** Name of HFSS variable that specifies junction inductance Cj defined on the boundary condition in HFSS. DO NOT USE Global names that start with `$`.

Warning: To define junctions, do **NOT** use global names! I.e., do not use names in ansys that start with `$`.

Note: Junction parameters example . To define junction parameters, see the following example

```
# Create project infor class
pinfo = ProjectInfo()

# Now, let us add a junction called `j1`, with the following properties
pinfo.junctions['j1'] = {
            'Lj_variable' : 'Lj_1', # name of Lj variable in Ansys
            'rect'        : 'jj_rect_1',
            'line'        : 'jj_line_1',
            #'Cj'          : 'Cj_1' # name of Cj variable in Ansys - optional
            }
```

To extend to define 5 junctions in bulk, we could use the following script

```
n_junctions = 5
for i in range(1, n_junctions + 1):
    pinfo.junctions[f'j{i}'] = {'Lj_variable' : f'Lj_{i}',
                                'rect'        : f'jj_rect_{i}',
                                'line'        : f'jj_line_{i}'}
```

---

check_connected()
Checks if fully connected including setup.

connect()
Do establish connection to Ansys desktop. Connects to project and then get design and setup

connect_design(*design_name: str = None*)
Sets self.design self.design_name

connect_project()
Sets self.app self.desktop self.project self.project_name self.project_path

connect_setup()
Connect to the first available setup or create a new in eigenmode and driven modal

Raises `Exception` – [description]

disconnect()
Disconnect from existing Ansys Desktop API.

get_all_object_names()
Returns array of strings

get_all_variables_names()
Returns array of all project and local design names.

get_dm()
Utility shortcut function to get the design and modeler.

```
oDesign, oModeler = pinfo.get_dm()
```

get_setup(*name: str*)
Connects to a specific setup for the design. Sets self.setup and self.setup_name.

Parameters

- **name** (*str*) – Name of the setup.
- **the setup does not exist, then throws a logger error.** (*If*) –
- **to None, in which case returns None** (*Defaults*) –

save()
Return all the data in a dictionary form that can be used to be saved

validate_junction_info()
Validate that the user has put in the junction info correctly. Do not also forget to check the length of the rectangles/line of the junction if you change it.

DistributedAnalysis

class pyEPR.core_distributed_analysis.**DistributedAnalysis**(**args*, ***kwargs*)
Bases: `object`

DISTRIBUTED ANALYSIS of layout and microwave results.

Main computation class & interface with HFSS.

This class defines a DistributedAnalysis object which calculates and saves Hamiltonian parameters from an HFSS simulation.

Further, it allows one to calculate dissipation, etc.

calc_Q_external(*variation*, *freq_GHz*, *U_E=None*)

Calculate the coupling Q of mode m with each port p Expected that you have specified the mode before calling this

Parameters

- **variation** (*str*) – A string identifier of the variation,
- **as '0', '1', ..** (*such*) –

calc_avg_current_J_surf_mag(*variation: str*, *junc_rect: str*, *junc_line*)

Peak current I_max for mode J in junction J The avg. is over the surface of the junction. I.e., spatial.

Parameters

- **variation** (*str*) – A string identifier of the variation, such as '0', '1', ...
- **junc_rect** (*str*) – name of rectangle to integrate over
- **junc_line** (*str*) – name of junction line to integrate over

Returns Value of peak current

calc_current(*fields*, *line: str*)

Function to calculate Current based on line. Not in use.

Parameters **line** (*str*) – integration line between plates - name

calc_current_using_line_voltage(*variation: str*, *junc_line_name: str*, *junc_L_Henries: float*, *Cj_Farads: float = None*)

Peak current I_max for prespecified mode calculating line voltage across junction.

Make sure that you have set the correct variation in HFSS before running this

Parameters

- **variation** – variation number
- **junc_line_name** – name of the HFSS line spanning the junction
- **junc_L_Henries** – junction inductance in henries
- **Cj_Farads** – junction cap in Farads
- **TODO** – Smooth?

calc_energy_electric(*variation: str = None*, *obj: str = 'AllObjects'*, *volume: str = 'Deprecated'*, *smooth: bool = False*, *obj_dims: int = 3*)

Calculates two times the peak electric energy, or 4 times the RMS, $4 * \mathcal{E}_{\mathrm{elec}}$ (since we do not divide by 2 and use the peak phasors).

$$\mathcal{E}_{\mathrm{elec}} = \frac{1}{4}\mathrm{Re}\int_V \mathrm{d}v \vec{E}^*_{\mathrm{max}} \overleftrightarrow{\epsilon} \vec{E}_{\mathrm{max}}$$

Parameters

- **variation** (*str*) – A string identifier of the variation, such as '0', '1', ...

- **obj** (*string | 'AllObjects'*) – Name of the object to integrate over
- **smooth** (*bool | False*) – Smooth the electric field or not when performing calculation
- **obj_dims** (*int | 3*) – 1 - line, 2 - surface, 3 - volume. Default volume

Example

Example use to calculate the energy participation ratio (EPR) of a substrate

```
_total  = epr_hfss.calc_energy_electric(obj='AllObjects')
_substr = epr_hfss.calc_energy_electric(obj='Box1')
print(f'Energy in substrate = {100*_substr/_total:.1f}%')
```

calc_energy_magnetic(*variation: str = None*, *obj: str = 'AllObjects'*, *volume: str = 'Deprecated'*, *smooth: bool = False*, *obj_dims: int = 3*)

See calc_energy_electric.

Parameters

- **variation** (*str*) – A string identifier of the variation, such as '0', '1', ...
- **volume** (*string | 'AllObjects'*) – Name of the volume to integrate over
- **smooth** (*bool | False*) – Smooth the electric field or not when performing calculation
- **obj_dims** (*int | 3*) – 1 - line, 2 - surface, 3 - volume. Default volume

calc_line_current(*variation*, *junc_line_name*)

calc_p_electric_volume(*name_dielectric3D*, *relative_to='AllObjects'*, *variation=None*, *E_total=None*)

Calculate the dielectric energy-participation ratio of a 3D object (one that has volume) relative to the dielectric energy of a list of objects.

This is as a function relative to another object or all objects.

When all objects are specified, this does not include any energy that might be stored in any lumped elements or lumped capacitors.

Returns _object/_total, (_object, _total)

calc_p_junction(*variation*, *U_H*, *U_E*, *Ljs*, *Cjs*)

For a single specific mode. Expected that you have specified the mode before calling this, *set_mode()*.

Expected to precalc U_H and U_E for mode, will return pandas pd.Series object:

- junc_rect = ['junc_rect1', 'junc_rect2'] name of junc rectangles to integrate H over
- junc_len = [0.0001] specify in SI units; i.e., meters
- LJs = [8e-09, 8e-09] SI units
- calc_sign = ['junc_line1', 'junc_line2']

WARNING: Cjs is experimental.

This function assumes there are no lumped capacitors in model.

Parameters

- **variation** (*str*) – A string identifier of the variation,
- **as '0', '1', ..** (*such*) –

Note: U_E and U_H are the total peak energy. (NOT twice as in U_ and U_H other places)

Warning: Potential errors: If you dont have a line or rect by the right name you will prob get an error of the type: com_error: (-2147352567, 'Exception occurred.', (0, None, None, None, 0, -2147024365), None)

calc_p_junction_single (*mode*, *variation*, *U_E=None*, *U_H=None*)
This function is used in the case of a single junction only. For multiple junctions, see *calc_p_junction()*.

Assumes no lumped capacitive elements.

design
Ansys design class handle

do_EPR_analysis (*variations: list = None*, *modes=None*, *append_analysis=True*)
Main analysis routine

Parameters **variation** (*str*) – A string identifier of the variation, such as '0', '1', ...

variations [list | None] Example list of variations is ['0', '1'] A variation is a combination of project/design variables in an optimetric sweep

modes [list | None] Modes to analyze for example modes = [0, 2, 3]

append_analysis (bool) : When we run the Ansys analysis, should we redo any variations that we have already done?

Assumptions: Low dissipation (high-Q). It is easier to assume no lumped capacitors to simply calculations, but we have recently added Cj_variable as a new feature that is begin tested to handle capacitors.

See the paper.

Load results with epr.QuantumAnalysis class

```
eprd = epr.DistributedAnalysis(pinfo)
eprd.do_EPR_analysis(append_analysis=False)
```

get_Qdielectric (*dielectric*, *mode*, *variation*, *U_E=None*)

get_Qseam (*seam*, *mode*, *variation*, *U_H=None*)
Calculate the contribution to Q of a seam, by integrating the current in the seam with finite conductance: set in the config file ref: http://arxiv.org/pdf/1509.01119.pdf

get_Qseam_sweep (*seam*, *mode*, *variation*, *variable*, *values*, *unit*, *U_H=None*, *pltresult=True*)
Q due to seam loss.

values = ['5mm','6mm','7mm'] ref: http://arxiv.org/pdf/1509.01119.pdf

get_Qsurface (*mode*, *variation*, *name*, *U_E=None*, *material_properties=None*)
Calculate the contribution to Q of a dielectric layer of dirt on a given surface. Set the dirt thickness and loss tangent in the config file ref: http://arxiv.org/pdf/1509.01854.pdf

get_Qsurface_all(*mode*, *variation*, *U_E=None*)
Calculate the contribution to Q of a dielectric layer of dirt on all surfaces. Set the dirt thickness and loss tangent in the config file ref: http://arxiv.org/pdf/1509.01854.pdf

get_ansys_frequencies_all(*vs='variation'*)
Return all ansys frequencies and quality factors vs a variation

Returns a multi-index pandas DataFrame

get_ansys_variables()
Get ansys variables for all variations

Returns Return a dataframe of variables as index and columns as the variations

get_ansys_variations()
Will update ansys information and result the list of variations.

Returns

```
("Cj='2fF' Lj='12nH'",
"Cj='2fF' Lj='12.5nH'",
"Cj='2fF' Lj='13nH'",
"Cj='2fF' Lj='13.5nH'",
"Cj='2fF' Lj='14nH'")
```

Return type For example

get_convergence(*variation='0'*)

Parameters

- **variation** (*str*) – A string identifier of the variation,
- **as '0', '1', ..** (*such*) –

Returns

A pandas DataFrame object

```
    Solved Elements     Max Delta Freq. % Pass Number
1           128955              NaN
2               167607          11.745000
3               192746          3.208600
4               199244          1.524000
```

get_convergence_vs_pass(*variation='0'*)
Makes a plot in HFSS that return a pandas dataframe

Parameters

- **variation** (*str*) – A string identifier of the variation,
- **as '0', '1', ..** (*such*) –

Returns

Returns a convergence vs pass number of the eignemode freqs.

```
    re(Mode(1)) [g]     re(Mode(2)) [g] re(Mode(3)) [g]
Pass []
1       4.643101        4.944204        5.586289
2       5.114490        5.505828        6.242423
3       5.278594        5.604426        6.296777
```

get_freqs_bare(*variation: str*)

> **Warning:** Outdated. Do not use. To be deprecated

Parameters **variation** (*str*) – A string identifier of the variation, such as '0', '1', ...

Returns [type] – [description]

get_freqs_bare_pd(*variation: str*, *frame=True*)

Return the freq and Qs of the solved modes for a variation. I.e., the Ansys solved frequencies.

Parameters

- **variation** (*str*) – A string identifier of the variation, such as '0', '1', ...
- **{bool} -- if True returns dataframe, else tuple of series.** (*frame*) –

Returns

If frame = True, then a multi-index Dataframe that looks something like this

		Freq. (GHz)	Quality Factor
variation	mode		
0	0	5.436892	1020
	1	7.030932	50200
1	0	5.490328	2010
	1	7.032116	104500

If frame = False, then a tuple of two Series, such as (Fs, Qs) – Tuple of pandas.Series objects; the row index is the mode number

get_junc_len_dir(*variation: str*, *junc_line*)

Return the length and direction of a junction defined by a line

Parameters

- **variation** (*str*) – simulation variation
- **junc_line** (*str*) – polyline object

Returns

junction length uj (list of 3 floats): x,y,z coordinates of the unit vector

tangent to the junction line

Return type jl (float)

get_junctions_L_and_C(*variation: str*)

Returns a pandas Series with the index being the junction name as specified in the project_info.

The values in the series are numeric and in SI base units, i.e., not nH but Henries, and not fF but Farads.

Parameters

- **variation** (*str*) – label such as '0' or 'all', in which case return
- **table for all variations** (*pandas*) –

get_mesh_statistics(*variation='0'*)

Parameters

- **variation** (*str*) – A string identifier of the variation,
- **as '0', '1', ..** (*such*) –

Returns: A pandas dataframe, such as

```
    Name        Num Tets    Min edge    length          Max edge length RMS edge length Min tet vol     Max tet vol     Mean tet vol    Std Devn (vol)
0   Region      909451          0.000243    0.860488        0.037048        6.006260e-13        0.037352        0.000029        6.268190e-04
1   substrate       1490356     0.000270    0.893770        0.023639        1.160090e-12        0.031253        0.000007        2.309920e-04
```

get_nominal_variation_index()

Returns A string identifies, such as '0' or '1', that labels the nominal variation index number.

This may not be in the solved list!s

get_previously_analyzed()
Return previously analyzed data.

Does not yet handle data that was previously saved in a filename.

get_variable_vs_variations(*variable: str*, *convert: bool = True*)
Get ansys variables

Return HFSS variable from `self.get_ansys_variables()` as a pandas series vs variations.

Parameters **convert** (*bool*) – Convert to a numeric quantity if possible using the ureg

get_variables(*variation=None*)
Get ansys variables.

Parameters **variation** (*str*) – A string identifier of the variation, such as '0', '1', ...

get_variation_string(*variation=None*)
Solved variation string identifier.

Parameters **variation** (*str*) – A string identifier of the variation, such as '0', '1', ...

Returns

Return the list variation string of parameters in ansys used to identify the variation.

```
"$test='0.25mm' Cj='2fF' Lj='12.5nH'"
```

get_variations()
An array of strings corresponding to **solved** variations corresponding to the selected Setup.

Returns

Returns a list of strings that give the variation labels for HFSS.

```
OrderedDict([
    ('0', "Cj='2fF' Lj='12nH'"),
    ('1', "Cj='2fF' Lj='12.5nH'"),
    ('2', "Cj='2fF' Lj='13nH'"),
    ('3', "Cj='2fF' Lj='13.5nH'"),
    ('4', "Cj='2fF' Lj='14nH'")])
```

has_fields(*variation: str = None*)
Determine if fields exist for a particular solution. Just calls *self.solutions.has_fields(variation_string)*

Parameters **variation** (*str*) – String of variation label, such as '0' or '1'. If None, gets the nominal variation

hfss_report_f_convergence (*variation='0'*, *save_csv=True*)
Create a report inside HFSS to plot the converge of freq and style it.

Saves report to csv file.

Returns a convergence vs pass number of the eignemode freqs. Returns a pandas dataframe:

```
    re(Mode(1)) [g]  re(Mode(2)) [g]  re(Mode(3)) [g]
Pass []
1      4.643101         4.944204         5.586289
2      5.114490         5.505828         6.242423
3      5.278594         5.604426         6.296777
```

hfss_report_full_convergence (*fig=None*, *_display=True*)
Plot a full report of teh convergences of an eigenmode analysis for a a given variation. Makes a plot inside hfss too.

Keyword Arguments

- **{matplotlib figure} -- Optional figure (default** (*fig*) – {None})
- **{bool} -- Force display or not. (default** (*_display*) – {True})

Returns [type] – [description]

load (*filepath=None*)
Utility function to load results file

Keyword Arguments **{[type]} -- [description] (default** (*filepath*) – {None})

n_variations
Number of **solved** variations, corresponding to the selected Setup.

options
Project info options

project
Ansys project class handle

quick_plot_frequencies (*swp_variable='variations'*, *ax=None*)
Quick plot of frequencies from HFSS

static results_variations_on_inside (*results: dict*)
Switches the order on result of variations. Reverse dict.

save (*project_info: dict = None*)
Save results to self.data_filename

Keyword Arguments **{dict} -- [description] (default** (project_info) – {None})

set_mode (*mode_num*, *phase=0*)
Set source excitations should be used for fields post processing. Counting modes from 0 onward

set_variation (*variation: str*)
Set the ansys design to a solved variation. This will change all local variables!

Warning: not tested with global variables.

setup
Ansys setup class handle. Could be None.

setup_data()
Set up folder paths for saving data to.

Sets the save filename with the current time.

Saves to Path(config.root_dir) / self.project.name / self.design.name

update_ansys_info()
‘ Updates all information about the Ansys solved variations and variables.

```
n_modes, _list_variations, nominal_variation, n_variations
```

variations = None
List of variation indices, which are strings of ints, such as [‘0’, ‘1’]

QuantumAnalysis

class pyEPR.core_quantum_analysis.**QuantumAnalysis**(*data_filename*, *variations: list = None*, *do_print_info=True*, *Res_hamil_filename=None*)

Bases: object

Defines an analysis object which loads and plots data from a h5 file This data is obtained using DistributedAnalysis

analyze_all_variations(*variations: List[str] = None*, *analyze_previous=False*, ***kwargs*)
See analyze_variation for full documentation

Parameters

- **variations** – None returns all_variations otherwise this is a list with number as strings [‘0’, ‘1’]
- **analyze_previous** – set to true if you wish to overwrite previous analysis
- ****kwargs** – Keyword arguments passed to *analyze_variation()*.

analyze_variation(*variation: str*, *cos_trunc: int = None*, *fock_trunc: int = None*, *print_result: bool = True*, *junctions: List[T] = None*, *modes: List[T] = None*)
Core analysis function to call!

Parameters

- **junctions** – list or slice of junctions to include in the analysis. None defaults to analysing all junctions
- **modes** – list or slice of modes to include in the analysis. None defaults to analysing all modes

Returns

Dictionary containing at least the following:

- f_0 [MHz]: Eigenmode frequencies computed by HFSS; i.e., linear freq returned in GHz
- f_1 [MHz]: Dressed mode frequencies (by the non-linearity; e.g., Lamb shift, etc.). Result based on 1st order perturbation theory on the 4th order expansion of the cosine.
- f_ND [MHz]: Numerical diagonalization result of dressed mode frequencies. only available if *cos_trunc* and *fock_trunc* are set (non None).

- chi_O1 [MHz]: Analytic expression for the chis based on a cos trunc to 4th order, and using 1st order perturbation theory. Diag is anharmonicity, off diag is full cross-Kerr.
- chi_ND [MHz]: Numerically diagonalized chi matrix. Diag is anharmonicity, off diag is full cross-Kerr.

Return type dict

full_report_variations(*var_list: list = None*)
see full_variation_report

full_variation_report(*variation*)
prints the results and parameters of a specific variation

Parameters **variation** (*int or str*) – the variation to be printed .

Returns

Return type None.

get_Ecs(*variation*)
ECs in GHz Returns as pandas series

get_Ejs(*variation*)
EJs in GHz See calcs.convert

get_ansys_energies(*swp_var='variation'*)
Return a multi-index dataframe of ansys energies vs swep_variable

Parameters **swp_var** (*str*) –

get_chis(*swp_variable='variation'*, *numeric=True*, *variations: list = None*, *m=None*, *n=None*)
return as multiindex data table

If you provide m and n as integers or mode labels, then the chi between these modes will be returned as a pandas Series.

get_convergences_max_delta_freq_vs_pass(*as_dataframe=True*)
Index([u'Pass Number', u'Solved Elements', u'Max Delta Freq. %'])

get_convergences_max_tets()
Index([u'Pass Number', u'Solved Elements', u'Max Delta Freq. %'])

get_convergences_tets_vs_pass(*as_dataframe=True*)
Index([u'Pass Number', u'Solved Elements', u'Max Delta Freq. %'])

get_epr_base_matrices(*variation*, *_renorm_pj=None*, *print_=False*)
Return the key matrices used in the EPR method for analytic calculations.

All as matrices

PJ Participation matrix, p_mj

SJ Sign matrix, s_mj

Om Omega_mm matrix (in GHz) (hbar = 1) Not radians.

EJ E_jj matrix of Josephson energies (in same units as hbar omega matrix)

PHI_zpf ZPFs in units of phi_0 reduced flux quantum

PJ_cap capacitive participation matrix

Return all as *np.array* PM, SIGN, Om, EJ, Phi_ZPF

get_frequencies (*swp_variable='variation'*, *numeric=True*, *variations: list = None*)
return as multiindex data table index: eigenmode label columns: variation label

get_mesh_tot ()

get_participations (*swp_variable='variation'*, *variations: list = None*, *inductive=True*, *_normed=True*)

inductive (bool): EPR for junction inductance when True, else for capacitors

Returns a multiindex dataframe: index 0: sweep variable index 1: mode number column: junction number

Plot the participation ratio of all junctions for a given mode vs a sweep of Lj.

get_quality_factors (*swp_variable='variation'*, *variations: list = None*)
return as pd.Series index: eigenmode label columns: variation label

get_variable_value (*swpvar*, *lv=None*)

get_variable_vs (*swpvar*, *lv=None*)
lv is list of variations (example ['0', '1']), if None it takes all variations swpvar is the variable by which to organize

return: ordered dictionary of key which is the variation number and the magnitude of swaver as the item

get_variation_of_multiple_variables_value (*Var_dic*, *lv=None*)

SEE get_variations_of_variable_value

A function to return all the variations in which one of the variables has a specific value lv is list of variations (example ['0', '1']), if None it takes all variations Var_dic is a dic with the name of the variable as key and the value to filter as item

get_variations_of_variable_value (*swpvar*, *value*, *lv=None*)
A function to return all the variations in which one of the variables has a specific value lv is list of variations (example ['0', '1']), if None it takes all variations swpvar is a string and the name of the variable we wish to filter value is the value of swapvr in which we are interested

returns lv - a list of the variations for which swavr==value

get_vs_variable (*swp_var*, *attr: str*)
Convert the index of a dictionary that is stored here from variation number to variable value.

Parameters

- **swp_var** (*str*) – name of sweep variable in ansys
- **attr** – name of local attribute, eg.., 'ansys_energies'

plot_hamiltonian_results (*swp_variable: str = 'variation'*, *variations: list = None*, *fig=None*, *x_label: str = None*)
Plot results versus variation

Keyword Arguments

- **{str} -- Variable against which we swept. If none, then just** (*swp_variable*) – take the variation index (default: {None})
- **{list} -- [description] (default** (variations) – {None})
- **{[type]} -- [description] (default** (*fig*) – {None})

Returns fig, axs

plot_results (*result*, *Y_label*, *variable*, *X_label*, *variations: list = None*)

plotting_dic_x(*Var_dic*, *var_name*)

print_info()

print_result(*result*)
Utility reporting function

print_variation(*variation*)
Utility reporting function

project_info

quick_plot_chi_alpha(*mode1*, *mode2*, *swp_variable='variation'*, *ax=None*, *kw=None*, *numeric=False*)
Quick plot chi between mode 1 and mode 2.

If you select mode1=mode2, then you will plot the alpha

kw : extra plot arguments

quick_plot_convergence(*ax=None*)
Plot a report of the Ansys convergence vs pass number ona twin axis for the number of tets and the max delta frequency of the eignemode.

quick_plot_frequencies(*mode*, *swp_variable='variation'*, *ax=None*, *kw=None*, *numeric=False*)
Quick plot freq for one mode

kw : extra plot arguments

quick_plot_mode(*mode*, *junction*, *mode1=None*, *swp_variable='variation'*, *numeric=False*, *sharex=True*)
Create a quick report to see mode parameters for only a single mode and a cross-kerr coupling to another mode. Plots the participation and cross participation Plots the frequencie plots the anharmonicity

The values are either for the numeric or the non-numeric results, set by *numeric*

quick_plot_participation(*mode*, *junction*, *swp_variable='variation'*, *ax=None*, *kw=None*)
Quick plot participation for one mode

kw : extra plot arguments

report_results(*swp_variable='variation'*, *numeric=True*)
Report in table form the results in a markdown friendly way in Jupyter notebook using the pandas interface.

1.1.5 pyEPR package

pyEPR

Automated Python module for the design and quantization of Josephson quantum circuits

Abstract: Superconducting circuits incorporating non-linear devices, such as Josephson junctions and nanowires, are among the leading platforms for emerging quantum technologies. Promising applications require designing and optimizing circuits with ever-increasing complexity and controlling their dissipative and Hamiltonian parameters to several significant digits. Therefore, there is a growing need for a systematic, simple, and robust approach for precise circuit design, extensible to increased complexity. The energy-participation ratio (EPR) approach presents such an approach to unify the design of dissipation and Hamiltonians around a single concept — the energy participation, a number between zero and one — in a single-step electromagnetic simulation. This markedly reduces the required number of simulations and allows for robust extension to complex systems. The approach is general purpose, derived ab initio, and valid for arbitrary non-linear devices and circuit architectures. Experimental results on a variety of circuit quantum electrodynamics (cQED) devices and architectures, 3D and flip-chip (2.5D), have been demonstrated

to exhibit ten percent to percent-level agreement for non-linear coupling and modal Hamiltonian parameters over five-orders of magnitude and across a dozen samples.

Here, in this package, all routines of the EPR approach are fully automated. An interface with ansys is provided. Automated analysis of lumped and distributed circuits is provided.

@author: Zlatko Minev, Zaki Leghas, ... and the pyEPR team @site: https://github.com/zlatko-minev/pyEPR @license: "BSD-3-Clause" @version: 0.9.0 @maintainer: Zlatko K. Minev and Asaf Diringer @email: zlatko.minev@aya.yale.edu @url: https://github.com/zlatko-minev/pyEPR @status: "Dev-Production"

pyEPR.**config**

class pyEPR.**ProjectInfo**(*project_path: str = None*, *project_name: str = None*, *design_name: str = None*, *setup_name: str = None*, *dielectrics_bulk: list = None*, *dielectric_surfaces: list = None*, *resistive_surfaces: list = None*, *seams: list = None*, *do_connect: bool = True*)

Bases: object

Primary class to store interface information between pyEPR and Ansys.

- **Ansys:** stores and provides easy access to the ansys interface classes *pyEPR.ansys.HfssApp*, *pyEPR.ansys.HfssDesktop*, *pyEPR.ansys.HfssProject*, *pyEPR.ansys.HfssDesign*, *pyEPR.ansys.HfssSetup* (which, if present could nbe a subclass, such as a driven modal setup *pyEPR.ansys.HfssDMSetup*, eigenmode *pyEPR.ansys.HfssEMSetup*, or Q3D *pyEPR.ansys.AnsysQ3DSetup*), the 3D modeler to design geometry *pyEPR.ansys.HfssModeler*.
- **Junctions:** The class stores params about the design that the user puts will use, such as the names and properties of the junctions, such as which rectangle and line is associated with which junction.

Note: Junction parameters. The junction parameters are stored in the self.junctions ordered dictionary

A Josephson tunnel junction has to have its parameters specified here for the analysis. Each junction is given a name and is specified by a dictionary. It has the following properties:

- **Lj_variable (str):** Name of HFSS variable that specifies junction inductance Lj defined on the boundary condition in HFSS. WARNING: DO NOT USE Global names that start with $.
- **rect (str):** String of Ansys name of the rectangle on which the lumped boundary condition is defined.
- **line (str):** Name of HFSS polyline which spans the length of the rectangle. Used to define the voltage across the junction. Used to define the current orientation for each junction. Used to define sign of ZPF.
- **length (str):** Length in HFSS of the junction rectangle and line (specified in meters). To create, you can use epr.parse_units('100um').
- **Cj_variable (str, optional) [experimental]:** Name of HFSS variable that specifies junction inductance Cj defined on the boundary condition in HFSS. DO NOT USE Global names that start with $.

Warning: To define junctions, do **NOT** use global names! I.e., do not use names in ansys that start with $.

Note: Junction parameters example . To define junction parameters, see the following example

```
# Create project infor class
pinfo = ProjectInfo()
```

(continues on next page)

(continued from previous page)

```
3
4  # Now, let us add a junction called `j1`, with the following properties
5  pinfo.junctions['j1'] = {
6              'Lj_variable' : 'Lj_1', # name of Lj variable in Ansys
7              'rect'        : 'jj_rect_1',
8              'line'        : 'jj_line_1',
9              #'Cj'          : 'Cj_1' # name of Cj variable in Ansys - optional
10             }
```

To extend to define 5 junctions in bulk, we could use the following script

```
n_junctions = 5
for i in range(1, n_junctions + 1):
    pinfo.junctions[f'j{i}'] = {'Lj_variable' : f'Lj_{i}',
                                'rect'        : f'jj_rect_{i}',
                                'line'        : f'jj_line_{i}'}
```

---

check_connected()
Checks if fully connected including setup.

connect()
Do establish connection to Ansys desktop. Connects to project and then get design and setup

connect_design(*design_name: str = None*)
Sets self.design self.design_name

connect_project()
Sets self.app self.desktop self.project self.project_name self.project_path

connect_setup()
Connect to the first available setup or create a new in eigenmode and driven modal

Raises `Exception` – [description]

disconnect()
Disconnect from existing Ansys Desktop API.

get_all_object_names()
Returns array of strings

get_all_variables_names()
Returns array of all project and local design names.

get_dm()
Utility shortcut function to get the design and modeler.

```
oDesign, oModeler = pinfo.get_dm()
```

get_setup(*name: str*)
Connects to a specific setup for the design. Sets self.setup and self.setup_name.

Parameters

- **name** (*str*) – Name of the setup.
- **the setup does not exist, then throws a logger error.** (*If*) –
- **to None, in which case returns None** (*Defaults*) –

save()
Return all the data in a dictionary form that can be used to be saved

validate_junction_info()
Validate that the user has put in the junction info correctly. Do not also forget to check the length of the rectangles/line of the junction if you change it.

class pyEPR.**DistributedAnalysis**(**args*, ***kwargs*)
Bases: object

DISTRIBUTED ANALYSIS of layout and microwave results.

Main computation class & interface with HFSS.

This class defines a DistributedAnalysis object which calculates and saves Hamiltonian parameters from an HFSS simulation.

Further, it allows one to calculate dissipation, etc.

calc_Q_external(*variation*, *freq_GHz*, *U_E=None*)
Calculate the coupling Q of mode m with each port p Expected that you have specified the mode before calling this

Parameters

- **variation** (*str*) – A string identifier of the variation,
- **as '0', '1', ..** (*such*) –

calc_avg_current_J_surf_mag(*variation: str*, *junc_rect: str*, *junc_line*)

Peak current I_max for mode J in junction J The avg. is over the surface of the junction. I.e., spatial.

Parameters

- **variation** (*str*) – A string identifier of the variation, such as '0', '1', ...
- **junc_rect** (*str*) – name of rectangle to integrate over
- **junc_line** (*str*) – name of junction line to integrate over

Returns Value of peak current

calc_current(*fields*, *line: str*)
Function to calculate Current based on line. Not in use.

Parameters **line** (*str*) – integration line between plates - name

calc_current_using_line_voltage(*variation: str*, *junc_line_name: str*, *junc_L_Henries: float*, *Cj_Farads: float = None*)
Peak current I_max for prespecified mode calculating line voltage across junction.

Make sure that you have set the correct variation in HFSS before running this

Parameters

- **variation** – variation number
- **junc_line_name** – name of the HFSS line spanning the junction
- **junc_L_Henries** – junction inductance in henries
- **Cj_Farads** – junction cap in Farads
- **TODO** – Smooth?

calc_energy_electric(*variation: str = None*, *obj: str = 'AllObjects'*, *volume: str = 'Deprecated'*, *smooth: bool = False*, *obj_dims: int = 3*)

Calculates two times the peak electric energy, or 4 times the RMS, $4 * \mathcal{E}_{\text{elec}}$ (since we do not divide by 2 and use the peak phasors).

$$\mathcal{E}_{\text{elec}} = \frac{1}{4}\text{Re}\int_V \mathrm{d}v \vec{E}^*_{\text{max}} \overleftrightarrow{\epsilon} \vec{E}_{\text{max}}$$

Parameters

- **variation** (*str*) – A string identifier of the variation, such as '0', '1', ...
- **obj** (*string | 'AllObjects'*) – Name of the object to integrate over
- **smooth** (*bool | False*) – Smooth the electric field or not when performing calculation
- **obj_dims** (*int | 3*) – 1 - line, 2 - surface, 3 - volume. Default volume

Example

Example use to calculate the energy participation ratio (EPR) of a substrate

```
_total  = epr_hfss.calc_energy_electric(obj='AllObjects')
_substr = epr_hfss.calc_energy_electric(obj='Box1')
print(f'Energy in substrate = {100*_substr/_total:.1f}%')
```

calc_energy_magnetic(*variation: str = None*, *obj: str = 'AllObjects'*, *volume: str = 'Deprecated'*, *smooth: bool = False*, *obj_dims: int = 3*)

See calc_energy_electric.

Parameters

- **variation** (*str*) – A string identifier of the variation, such as '0', '1', ...
- **volume** (*string | 'AllObjects'*) – Name of the volume to integrate over
- **smooth** (*bool | False*) – Smooth the electric field or not when performing calculation
- **obj_dims** (*int | 3*) – 1 - line, 2 - surface, 3 - volume. Default volume

calc_line_current(*variation*, *junc_line_name*)

calc_p_electric_volume(*name_dielectric3D*, *relative_to='AllObjects'*, *variation=None*, *E_total=None*)

Calculate the dielectric energy-participation ratio of a 3D object (one that has volume) relative to the dielectric energy of a list of objects.

This is as a function relative to another object or all objects.

When all objects are specified, this does not include any energy that might be stored in any lumped elements or lumped capacitors.

Returns _object/_total, (_object, _total)

calc_p_junction(*variation*, *U_H*, *U_E*, *Ljs*, *Cjs*)

For a single specific mode. Expected that you have specified the mode before calling this, *set_mode()*.

Expected to precalc U_H and U_E for mode, will return pandas pd.Series object:

- junc_rect = ['junc_rect1', 'junc_rect2'] name of junc rectangles to integrate H over
- junc_len = [0.0001] specify in SI units; i.e., meters

- LJs = [8e-09, 8e-09] SI units
- calc_sign = ['junc_line1', 'junc_line2']

WARNING: Cjs is experimental.

This function assumes there are no lumped capacitors in model.

Parameters

- **variation** (*str*) – A string identifier of the variation,
- **as '0', '1', ..** (*such*) –

Note: U_E and U_H are the total peak energy. (NOT twice as in U_ and U_H other places)

Warning: Potential errors: If you dont have a line or rect by the right name you will prob get an error of the type: com_error: (-2147352567, 'Exception occurred.', (0, None, None, None, 0, -2147024365), None)

calc_p_junction_single (*mode*, *variation*, *U_E=None*, *U_H=None*)
This function is used in the case of a single junction only. For multiple junctions, see *calc_p_junction()*.

Assumes no lumped capacitive elements.

design
Ansys design class handle

do_EPR_analysis (*variations: list = None*, *modes=None*, *append_analysis=True*)
Main analysis routine

Parameters **variation** (*str*) – A string identifier of the variation, such as '0', '1', ...

variations [list | None] Example list of variations is ['0', '1'] A variation is a combination of project/design variables in an optimetric sweep

modes [list | None] Modes to analyze for example modes = [0, 2, 3]

append_analysis (bool) : When we run the Ansys analysis, should we redo any variations that we have already done?

Assumptions: Low dissipation (high-Q). It is easier to assume no lumped capacitors to simply calculations, but we have recently added Cj_variable as a new feature that is begin tested to handle capacitors.

See the paper.

Load results with epr.QuantumAnalysis class

```
eprd = epr.DistributedAnalysis(pinfo)
eprd.do_EPR_analysis(append_analysis=False)
```

get_Qdielectric (*dielectric*, *mode*, *variation*, *U_E=None*)

get_Qseam (*seam*, *mode*, *variation*, *U_H=None*)
Calculate the contribution to Q of a seam, by integrating the current in the seam with finite conductance: set in the config file ref: http://arxiv.org/pdf/1509.01119.pdf

get_Qseam_sweep(*seam*, *mode*, *variation*, *variable*, *values*, *unit*, *U_H=None*, *pltresult=True*)
Q due to seam loss.

values = ['5mm','6mm','7mm'] ref: http://arxiv.org/pdf/1509.01119.pdf

get_Qsurface(*mode*, *variation*, *name*, *U_E=None*, *material_properties=None*)
Calculate the contribution to Q of a dielectric layer of dirt on a given surface. Set the dirt thickness and loss tangent in the config file ref: http://arxiv.org/pdf/1509.01854.pdf

get_Qsurface_all(*mode*, *variation*, *U_E=None*)
Calculate the contribution to Q of a dielectric layer of dirt on all surfaces. Set the dirt thickness and loss tangent in the config file ref: http://arxiv.org/pdf/1509.01854.pdf

get_ansys_frequencies_all(*vs='variation'*)
Return all ansys frequencies and quality factors vs a variation

Returns a multi-index pandas DataFrame

get_ansys_variables()
Get ansys variables for all variations

Returns Return a dataframe of variables as index and columns as the variations

get_ansys_variations()
Will update ansys information and result the list of variations.

Returns

```
("Cj='2fF' Lj='12nH'",
"Cj='2fF' Lj='12.5nH'",
"Cj='2fF' Lj='13nH'",
"Cj='2fF' Lj='13.5nH'",
"Cj='2fF' Lj='14nH'")
```

Return type For example

get_convergence(*variation='0'*)

Parameters

- **variation** (*str*) – A string identifier of the variation,
- **as '0', '1', ..** (*such*) –

Returns

A pandas DataFrame object

```
    Solved Elements     Max Delta Freq. % Pass Number
1           128955              NaN
2              167607          11.745000
3              192746          3.208600
4              199244          1.524000
```

get_convergence_vs_pass(*variation='0'*)
Makes a plot in HFSS that return a pandas dataframe

Parameters

- **variation** (*str*) – A string identifier of the variation,
- **as '0', '1', ..** (*such*) –

Returns

Returns a convergence vs pass number of the eignemode freqs.

```
    re(Mode(1)) [g]     re(Mode(2)) [g] re(Mode(3)) [g]
Pass []
1       4.643101        4.944204        5.586289
2       5.114490        5.505828        6.242423
3       5.278594        5.604426        6.296777
```

get_freqs_bare (*variation: str*)

Warning: Outdated. Do not use. To be deprecated

Parameters **variation** (*str*) – A string identifier of the variation, such as '0', '1', ...

Returns [type] – [description]

get_freqs_bare_pd (*variation: str*, *frame=True*)

Return the freq and Qs of the solved modes for a variation. I.e., the Ansys solved frequencies.

Parameters

- **variation** (*str*) – A string identifier of the variation, such as '0', '1', ...
- **{bool} -- if True returns dataframe, else tuple of series.** (*frame*) –

Returns

If frame = True, then a multi-index Dataframe that looks something like this

```
                Freq. (GHz)  Quality Factor
variation mode
0         0        5.436892            1020
          1        7.030932           50200
1         0        5.490328            2010
          1        7.032116          104500
```

If frame = False, then a tuple of two Series, such as (Fs, Qs) – Tuple of pandas.Series objects; the row index is the mode number

get_junc_len_dir (*variation: str*, *junc_line*)

Return the length and direction of a junction defined by a line

Parameters

- **variation** (*str*) – simulation variation
- **junc_line** (*str*) – polyline object

Returns

junction length uj (list of 3 floats): x,y,z coordinates of the unit vector

tangent to the junction line

Return type jl (float)

get_junctions_L_and_C(*variation: str*)
Returns a pandas Series with the index being the junction name as specified in the project_info.

The values in the series are numeric and in SI base units, i.e., not nH but Henries, and not fF but Farads.

Parameters

- **variation** (*str*) – label such as '0' or 'all', in which case return
- **table for all variations** (*pandas*) –

get_mesh_statistics(*variation='0'*)

Parameters

- **variation** (*str*) – A string identifier of the variation,
- **as '0', '1', ..** (*such*) –

Returns: A pandas dataframe, such as

```
   Name        Num Tets    Min edge    length          Max edge length
↪RMS edge length Min tet vol     Max tet vol     Mean tet vol    Std Devn
↪(vol)
0   Region      909451          0.000243    0.860488        0.037048
↪  6.006260e-13        0.037352        0.000029        6.268190e-04
1   substrate       1490356     0.000270    0.893770        0.023639
↪  1.160090e-12        0.031253        0.000007        2.309920e-04
```

get_nominal_variation_index()

Returns A string identifies, such as '0' or '1', that labels the nominal variation index number.

This may not be in the solved list!s

get_previously_analyzed()
Return previously analyzed data.

Does not yet handle data that was previously saved in a filename.

get_variable_vs_variations(*variable: str*, *convert: bool = True*)
Get ansys variables

Return HFSS variable from `self.get_ansys_variables()` as a pandas series vs variations.

Parameters **convert** (*bool*) – Convert to a numeric quantity if possible using the ureg

get_variables(*variation=None*)
Get ansys variables.

Parameters **variation** (*str*) – A string identifier of the variation, such as '0', '1', ...

get_variation_string(*variation=None*)
Solved variation string identifier.

Parameters **variation** (*str*) – A string identifier of the variation, such as '0', '1', ...

Returns

Return the list variation string of parameters in ansys used to identify the variation.

```
"$test='0.25mm' Cj='2fF' Lj='12.5nH'"
```

get_variations()
An array of strings corresponding to **solved** variations corresponding to the selected Setup.

Returns

Returns a list of strings that give the variation labels for HFSS.

```
OrderedDict([
    ('0', "Cj='2fF' Lj='12nH'"),
    ('1', "Cj='2fF' Lj='12.5nH'"),
    ('2', "Cj='2fF' Lj='13nH'"),
    ('3', "Cj='2fF' Lj='13.5nH'"),
    ('4', "Cj='2fF' Lj='14nH'")])
```

has_fields (*variation: str = None*)
Determine if fields exist for a particular solution. Just calls *self.solutions.has_fields(variation_string)*

Parameters **variation** (*str*) – String of variation label, such as '0' or '1'. If None, gets the nominal variation

hfss_report_f_convergence (*variation='0'*, *save_csv=True*)
Create a report inside HFSS to plot the converge of freq and style it.

Saves report to csv file.

Returns a convergence vs pass number of the eignemode freqs. Returns a pandas dataframe:

```
    re(Mode(1)) [g]  re(Mode(2)) [g]  re(Mode(3)) [g]
Pass []
1    4.643101         4.944204         5.586289
2    5.114490         5.505828         6.242423
3    5.278594         5.604426         6.296777
```

hfss_report_full_convergence (*fig=None*, *_display=True*)
Plot a full report of teh convergences of an eigenmode analysis for a a given variation. Makes a plot inside hfss too.

Keyword Arguments

- **{matplotlib figure} -- Optional figure (default** (*fig*) – {None})
- **{bool} -- Force display or not. (default** (*_display*) – {True})

Returns [type] – [description]

load (*filepath=None*)
Utility function to load results file

Keyword Arguments **{[type]} -- [description] (default** (*filepath*) – {None})

n_variations
Number of **solved** variations, corresponding to the selected Setup.

options
Project info options

project
Ansys project class handle

quick_plot_frequencies (*swp_variable='variations'*, *ax=None*)
Quick plot of frequencies from HFSS

static results_variations_on_inside (*results: dict*)
Switches the order on result of variations. Reverse dict.

save (*project_info: dict = None*)
Save results to self.data_filename

Keyword Arguments {dict} -- [description] (default (project_info) – {None})

set_mode (*mode_num*, *phase=0*)
Set source excitations should be used for fields post processing. Counting modes from 0 onward

set_variation (*variation: str*)
Set the ansys design to a solved variation. This will change all local variables!

Warning: not tested with global variables.

setup
Ansys setup class handle. Could be None.

setup_data ()
Set up folder paths for saving data to.

Sets the save filename with the current time.

Saves to Path(config.root_dir) / self.project.name / self.design.name

update_ansys_info ()
‘ Updates all information about the Ansys solved variations and variables.

```
n_modes, _list_variations, nominal_variation, n_variations
```

variations = None
List of variation indices, which are strings of ints, such as [‘0’, ‘1’]

class pyEPR.**QuantumAnalysis** (*data_filename*, *variations: list = None*, *do_print_info=True*, *Res_hamil_filename=None*)
Bases: object

Defines an analysis object which loads and plots data from a h5 file This data is obtained using DistributedAnalysis

analyze_all_variations (*variations: List[str] = None*, *analyze_previous=False*, ***kwargs*)
See analyze_variation for full documentation

Parameters

- **variations** – None returns all_variations otherwise this is a list with number as strings [‘0’, ‘1’]
- **analyze_previous** – set to true if you wish to overwrite previous analysis
- ****kwargs** – Keyword arguments passed to *analyze_variation()*.

analyze_variation (*variation: str*, *cos_trunc: int = None*, *fock_trunc: int = None*, *print_result: bool = True*, *junctions: List[T] = None*, *modes: List[T] = None*)
Core analysis function to call!

Parameters

- **junctions** – list or slice of junctions to include in the analysis. None defaults to analysing all junctions
- **modes** – list or slice of modes to include in the analysis. None defaults to analysing all modes

Returns

Dictionary containing at least the following:

- f_0 [MHz]: Eigenmode frequencies computed by HFSS; i.e., linear freq returned in GHz
- f_1 [MHz]: Dressed mode frequencies (by the non-linearity; e.g., Lamb shift, etc.). Result based on 1st order perturbation theory on the 4th order expansion of the cosine.
- f_ND [MHz]: Numerical diagonalization result of dressed mode frequencies. only available if *cos_trunc* and *fock_trunc* are set (non None).
- chi_O1 [MHz]: Analytic expression for the chis based on a cos trunc to 4th order, and using 1st order perturbation theory. Diag is anharmonicity, off diag is full cross-Kerr.
- chi_ND [MHz]: Numerically diagonalized chi matrix. Diag is anharmonicity, off diag is full cross-Kerr.

Return type dict

full_report_variations(*var_list: list = None*)
see full_variation_report

full_variation_report(*variation*)
prints the results and parameters of a specific variation

Parameters **variation** (*int or str*) – the variation to be printed .

Returns

Return type None.

get_Ecs(*variation*)
ECs in GHz Returns as pandas series

get_Ejs(*variation*)
EJs in GHz See calcs.convert

get_ansys_energies(*swp_var='variation'*)
Return a multi-index dataframe of ansys energies vs swep_variable

Parameters **swp_var** (*str*) –

get_chis(*swp_variable='variation'*, *numeric=True*, *variations: list = None*, *m=None*, *n=None*)
return as multiindex data table

If you provide m and n as integers or mode labels, then the chi between these modes will be returned as a pandas Series.

get_convergences_max_delta_freq_vs_pass(*as_dataframe=True*)
Index([u'Pass Number', u'Solved Elements', u'Max Delta Freq. %'])

get_convergences_max_tets()
Index([u'Pass Number', u'Solved Elements', u'Max Delta Freq. %'])

get_convergences_tets_vs_pass(*as_dataframe=True*)
Index([u'Pass Number', u'Solved Elements', u'Max Delta Freq. %'])

get_epr_base_matrices(*variation*, *_renorm_pj=None*, *print_=False*)
Return the key matrices used in the EPR method for analytic calculations.

All as matrices

PJ Participation matrix, p_mj

SJ Sign matrix, s_mj

Om Omega_mm matrix (in GHz) (hbar = 1) Not radians.

EJ E_jj matrix of Josephson energies (in same units as hbar omega matrix)

PHI_zpf ZPFs in units of phi_0 reduced flux quantum

PJ_cap capacitive participation matrix

Return all as *np.array* PM, SIGN, Om, EJ, Phi_ZPF

get_frequencies(*swp_variable='variation'*, *numeric=True*, *variations: list = None*)
return as multiindex data table index: eigenmode label columns: variation label

get_mesh_tot()

get_participations(*swp_variable='variation'*, *variations: list = None*, *inductive=True*, *_normed=True*)
inductive (bool): EPR for junction inductance when True, else for capacitors

Returns a multiindex dataframe: index 0: sweep variable index 1: mode number column: junction number

Plot the participation ratio of all junctions for a given mode vs a sweep of Lj.

get_quality_factors(*swp_variable='variation'*, *variations: list = None*)
return as pd.Series index: eigenmode label columns: variation label

get_variable_value(*swpvar*, *lv=None*)

get_variable_vs(*swpvar*, *lv=None*)
lv is list of variations (example ['0', '1']), if None it takes all variations swpvar is the variable by which to organize

return: ordered dictionary of key which is the variation number and the magnitude of swaver as the item

get_variation_of_multiple_variables_value(*Var_dic*, *lv=None*)

SEE get_variations_of_variable_value

A function to return all the variations in which one of the variables has a specific value lv is list of variations (example ['0', '1']), if None it takes all variations Var_dic is a dic with the name of the variable as key and the value to filter as item

get_variations_of_variable_value(*swpvar*, *value*, *lv=None*)
A function to return all the variations in which one of the variables has a specific value lv is list of variations (example ['0', '1']), if None it takes all variations swpvar is a string and the name of the variable we wish to filter value is the value of swapvr in which we are interested

returns lv - a list of the variations for which swavr==value

get_vs_variable(*swp_var*, *attr: str*)
Convert the index of a dictionary that is stored here from variation number to variable value.

Parameters

- **swp_var** (*str*) – name of sweep variable in ansys
- **attr** – name of local attribute, eg.., 'ansys_energies'

plot_hamiltonian_results(*swp_variable: str = 'variation'*, *variations: list = None*, *fig=None*, *x_label: str = None*)
Plot results versus variation

Keyword Arguments

- **{str} -- Variable against which we swept. If none, then just** (*swp_variable*) – take the variation index (default: {None})
- **{list} -- [description] (default** (*variations*) – {None})
- **{[type]} -- [description] (default** (*fig*) – {None})

Returns fig, axs

plot_results(*result*, *Y_label*, *variable*, *X_label*, *variations: list = None*)

plotting_dic_x(*Var_dic*, *var_name*)

print_info()

print_result(*result*)
Utility reporting function

print_variation(*variation*)
Utility reporting function

project_info

quick_plot_chi_alpha(*mode1*, *mode2*, *swp_variable='variation'*, *ax=None*, *kw=None*, *numeric=False*)
Quick plot chi between mode 1 and mode 2.

If you select mode1=mode2, then you will plot the alpha

kw : extra plot arguments

quick_plot_convergence(*ax=None*)
Plot a report of the Ansys convergence vs pass number ona twin axis for the number of tets and the max delta frequency of the eignemode.

quick_plot_frequencies(*mode*, *swp_variable='variation'*, *ax=None*, *kw=None*, *numeric=False*)
Quick plot freq for one mode

kw : extra plot arguments

quick_plot_mode(*mode*, *junction*, *mode1=None*, *swp_variable='variation'*, *numeric=False*, *sharex=True*)
Create a quick report to see mode parameters for only a single mode and a cross-kerr coupling to another mode. Plots the participation and cross participation Plots the frequencie plots the anharmonicity

The values are either for the numeric or the non-numeric results, set by *numeric*

quick_plot_participation(*mode*, *junction*, *swp_variable='variation'*, *ax=None*, *kw=None*)
Quick plot participation for one mode

kw : extra plot arguments

report_results(*swp_variable='variation'*, *numeric=True*)
Report in table form the results in a markdown friendly way in Jupyter notebook using the pandas interface.

pyEPR.**Project_Info**
alias of *pyEPR.project_info.ProjectInfo*

pyEPR.**pyEPR_HFSSAnalysis**
alias of *pyEPR.core_distributed_analysis.DistributedAnalysis*

pyEPR.**pyEPR_Analysis**
alias of *pyEPR.core_quantum_analysis.QuantumAnalysis*

pyEPR.**parse_units**(*x*)
Convert number, string, and lists/arrays/tuples to numbers scaled in HFSS units.

Converts to LENGTH_UNIT = meters [HFSS UNITS] Assumes input units LENGTH_UNIT_ASSUMED = mm [USER UNITS]

[USER UNITS] —-> [HFSS UNITS]

pyEPR.**parse_units_user**(*x*)
Convert from user assumed units to user assumed units [USER UNITS] —-> [USER UNITS]

pyEPR.**parse_entry**(*entry*, *convert_to_unit='meter'*)
Should take a list of tuple of list... of int, float or str... For iterables, returns lists

Subpackages

pyEPR.calcs package

Main calculation module

Submodules

pyEPR.calcs.back_box_numeric module

Numerical diagonalization of quantum Hamiltonian and parameter extraction.

@author: Phil Reinhold, Zlatko Minev, Lysander Christakis

Original code on black_box_hamiltonian and make_dispersive functions by Phil Reinhold Revisions and updates by Zlatko Minev & Lysander Christakis

pyEPR.calcs.back_box_numeric.**epr_numerical_diagonalization**(*freqs*, *Ljs*, *ϕzpf*, *cos_trunc=8*, *fock_trunc=9*, *use_1st_order=False*, *return_H=False*, *non_linear_potential=None*)
Numerical diagonalization for pyEPR. Ask Zlatko for details.

Parameters

- **fs** – (GHz, not radians) Linearized model, H_lin, normal mode frequencies in Hz, length M
- **ljs** – (Henries) junction linearized inductances in Henries, length J
- **fzpfs** – (reduced) Reduced Zero-point fluctuation of the junction fluxes for each mode across each junction, shape MxJ

Returns Hamiltonian mode freq and dispersive shifts. Shifts are in MHz. Shifts have flipped sign so that down shift is positive.

pyEPR.calcs.back_box_numeric.**make_dispersive**(*H*, *fock_trunc*, *fzpfs=None*, *f0s=None*, *chi_prime=False*, *use_1st_order=False*)
Input: Hamiltonian Matrix. Optional: phi_zpfs and normal mode frequencies, f0s. use_1st_order : deprecated

Output: Return dressed mode frequencies, chis, chi prime, phi_zpf flux (not reduced), and linear frequencies

Description: Takes the Hamiltonian matrix H from bbq_hmt. It them finds the eigenvalues/eigenvectors and assigns quantum numbers to them — i.e., mode excitations, such as, for instance, for three mode, $|0,0,0\rangle$ or $|0,0,1\rangle$, which correspond to no excitations in any of the modes or one excitation in the 3rd mode, resp. The assignment is performed based on the maximum overlap between the eigenvectors of H_full and H_lin. If this crude explanation is confusing, let me know, I will write a more detailed one |:slightly_smiling_face:| Based on the assignment of the excitations, the function returns the dressed mode frequencies ω'_m, and the cross-Kerr matrix (including anharmonicities) extracted from the numerical diagonalization, as well as from 1st order perturbation theory. Note, the diagonal of the CHI matrix is directly the anharmonicity term.

pyEPR.calcs.back_box_numeric.**black_box_hamiltonian**(*fs*, *ljs*, *fzpfs*, *cos_trunc=5*, *fock_trunc=8*, *individual=False*, *non_linear_potential=None*)

Parameters

- **fs** – Linearized model, H_lin, normal mode frequencies in Hz, length N
- **ljs** – junction linearized inductances in Henries, length M
- **fzpfs** – Zero-point fluctuation of the junction fluxes for each mode across each junction, shape MxJ

Returns Hamiltonian in units of Hz (i.e H / h)

All in SI units. The ZPF fed in are the generalized, not reduced, flux.

Description: Takes the linear mode frequencies, ω_m, and the zero-point fluctuations, ZPFs, and builds the Hamiltonian matrix of H_{full}, assuming cos potential.

pyEPR.calcs.back_box_numeric.**black_box_hamiltonian_nq**(*freqs*, *zmat*, *ljs*, *cos_trunc=6*, *fock_trunc=8*, *show_fit=False*)

N-Qubit version of bbq, based on the full Z-matrix Currently reproduces 1-qubit data, untested on n-qubit data Assume: Solve the model without loss in HFSS.

pyEPR.calcs.basic module

Basic calculations that apply in general .

class pyEPR.calcs.basic.**CalcsBasic**

Bases: object

static epr_cap_to_nzpf(*Pmj_cap*, *SJ*, Ω, *Ec*)

Experimental. To be tested

static epr_to_zpf(*Pmj*, *SJ*, Ω, *EJ*)

Arguments, All as matrices (numpy arrays):

Pnj MxJ energy-participation-ratio matrix, p_mj

SJ MxJ sign matrix, s_mj

Ω MxM diagonal matrix of frequencies (GHz, not radians, diagonal)

EJ JxJ diagonal matrix matrix of Josephson energies (in same units as Om)

Returns reduced zpf (in units of ϕ_0)

pyEPR.calcs.constants module

pyEPR constants and convenience definitions.

@author: Zlatko Minev

pyEPR.calcs.convert module

Created on Tue Mar 19 18:14:08 2019

Unit and variable conversions.

@author: Zlatko Minev

class pyEPR.calcs.convert.**Convert**
Bases: object

Static container class for conversions of units and variables.

TEST CONVERSION:

```
from pyEPR.toolbox.conversions import Convert

Lj_nH, Cs_fF = 11, 60
Convert.transmon_print_all_params(Lj_nH, Cs_fF);
```

static Cs_from_Ec(*Ec*, *units_in='MHz'*, *units_out='fF'*)
Charging energy $4E_c n^2$, where $n = Q/2e$

Returns in SI units, in Farads.

$E_C = \frac{e^2}{2C} J$

static Ec_from_Cs(*Cs*, *units_in='fF'*, *units_out='MHz'*)
Charging energy $4E_c n^2$, where $n = Q/2e$ Returns in MHz

$E_C = \frac{e^2}{2C} J$

static Ej_from_Lj(*Lj*, *units_in='nH'*, *units_out='MHz'*)
Josephson Junction energy from Josephson inductance. Returns in MHz

$E_j = \phi_0^2 / L_J$

static Ic_from_Lj(*Lj*, *units_in='nH'*, *units_out='nA'*)
Josephson Junction crit. curr from Josephson inductance.

$E_j = \phi_0^2 / L_J = \phi_0 I_C$

static Lj_from_Ej(*Ej*, *units_in='MHz'*, *units_out='nH'*)
Josephson Junction ind from Josephson energy in MHZ. Returns in units of nano Henries by default

$E_j = \phi_0^2 / L_J$

static Lj_from_Ic(*Lj*, *units_in='nA'*, *units_out='nH'*)
Josephson Junction crit. curr from Josephson inductance.

$E_j = \phi_0^2 / L_J = \phi_0 I_C$

static Omega_from_LC(*L*, *C*)
Calculate the resonant *angular* frequency

static ZPF_from_EPR(*hfss_freqs*, *hfss_epr_*, *hfss_signs*, *hfss_Ljs*, *Lj_units_in='H'*, *to_df=False*)

Parameters

- **be either Pandas or numpy arrays.** (*Can*) –
- **hfss_freqs** – HFSS Freqs. (standard units: GHz, but these will cancel with Ejs) (list/Series)
- **hfss_epr** – EPR ratio matrix, dim = M x J (2D array/DataFrame)
- **hfss_signs** – Sign matrix, dim = M x J (2D array/DataFrame)
- **hfss_Ljs** – Assumed in Henries (see Lj_units_in). (list/Series)
- **Lj_units_in** – Default 'H' for Henries. Can change here.

Returns M x J matrix of reduced ZPF; i.e., scaled by reduced flux quantum. type: np.array and a tuple of matrices.

Example use: zpf, (Ωm, Ej, Pmj, Smj) = Convert.ZPF_from_EPR(hfss_freqs, hfss_epr, hfss_signs, hfss_Ljs, to_df=True)

static ZPF_from_LC(*L*, *C*)
Input units assumed to be identical

Returns Phi ZPF in and Q_ZPF in NOT reduced units, but SI

static fromSI(*number*, *from_units: str*)
Convert a number with SI units, such as fF to F.

Parameters

- **{[numeric]} -- number**(*number*) –
- **{str} -- string**(*from_units*) –

Returns numeric number, with units expanded

static toSI(*number*, *from_units: str*)
Convert from SI unit prefix to regular SI units If the from_units is ' ' or not in the prefix list, then the unit is assumed to be

pyEPR.calcs.hamiltonian module

Hamiltonian and Matrix Operations. Hamiltonian operations heavily draw on qutip package. This package must be installed for them to work.

class pyEPR.calcs.hamiltonian.**HamOps**
Bases: object

static closest_state_to(*s: qutip.qobj.Qobj*, *energyMHz*, *evecs*)
Returns the energy of the closest state to s

static closest_state_to_idx(*s: qutip.qobj.Qobj*, *evecs*)
Returns the index

static fock_state_on(*d: dict*, *fock_trunc: int*, *N_modes: int*)
d={mode number: # of photons} In the bare eigen basis

static identify_Fock_levels(*fock_trunc: int*, *evecs*, *N_modes=2*, *Fock_max=4*)
Return quantum numbers in terms of the undiagonalized eigenbasis.

class pyEPR.calcs.hamiltonian.**MatrixOps**
Bases: object

static cos(*op_cos_arg: qutip.qobj.Qobj*)
Make cosine operator matrix from argument op_cos_arg

op_cos_arg (qutip.Qobj) : argument of the cosine

static cos_approx(*x*, *cos_trunc=5*)
Create a Taylor series matrix approximation of the cosine, up to some order.

static dot(*ais*, *bis*)
Dot product

pyEPR.calcs.transmon module

Transmon calculations

class pyEPR.calcs.transmon.**CalcsTransmon**
Bases: object

Common calculations and parameter reporting used for transmon qubits.

static charge_dispersion_approx(*m*, *Ec*, *Ej*)
Use Eq. (2.5) of Koch's paper.

static dispersiveH_params_PT_O1(*Pmj*, *Ωm*, *Ej*)
First order PT on the 4th power of the JJ cosine.

This function applied to an unfrustrated Josephson junction.

Pmj : Matrix MxJ Ωm : GHz Matrix MxM Ej : GHz Matrix JxJ

returns f_O1, χ_O1 χ_O1 has diagonal divided by 2 so as to give true anharmonicity.

Example use: ..codeblock python

PT_01: Calculate 1st order PT results f_O1, χ_O1 = Calc_basic.dispersiveH_params_PT_O1(Pmj, Ωm, Ej)

static transmon_get_all_params(*Ej_MHz*, *Ec_MHz*)
Linear harmonic oscillator approximation of transmon. Convenience func

static transmon_print_all_params(*Lj_nH*, *Cs_fF*)
Linear harmonic oscillator approximation of transmon. Convenience func

pyEPR.toolbox package

Submodules

pyEPR.toolbox.plotting module

Created on Fri Aug 25 19:30:12 2017

Plotting snippets and useful functions

@author: Zlatko K. Minev

pyEPR.toolbox.plotting.**legend_translucent**(*ax: matplotlib.axes._axes.Axes*, *values=[]*, *loc=0*, *alpha=0.5*, *leg_kw={}*)
values = [["%.2f" %k for k in RES]]

Also, you can use the following: leg_kw = dict(fancybox =True, fontsize = 9,

framealpha =0.5, ncol = 1)

blah.plot().legend(**leg_kw)

pyEPR.toolbox.plotting.**cmap_discrete**(*n*, *cmap_kw={}*)
Discrete colormap. cmap_kw = dict(colormap = plt.cm.gist_earth, start = 0.05, stop = .95)

helix = True, Allows us to instead call helix from here

pyEPR.toolbox.plotting.**get_color_cycle**(*n*, *colormap=None*, *start=0.0*, *stop=1.0*, *format='hex'*)
See also get_next_color

pyEPR.toolbox.plotting.**xarr_heatmap**(*fg*, *title=None*, *kwheat={}*, *fmt=('%.3f', '%.2f')*, *fig=None*)
Needs seaborn and xarray

pyEPR.toolbox.pythonic module

Created on Sat Feb 04 09:32:46 2017

@author: Zlatko K. Minev, pyEPR ream

pyEPR.toolbox.pythonic.**fact**(*n*)
Factorial

pyEPR.toolbox.pythonic.**nck**(*n*, *k*)
choose

pyEPR.toolbox.pythonic.**combinekw**(*kw1*, *kw2*)
Copy kw1, update with kw2, return result

pyEPR.toolbox.pythonic.**divide_diagonal_by_2**(*CHI0*, *div_fact=2.0*)

pyEPR.toolbox.pythonic.**df_find_index**(*s: pandas.core.series.Series*, *find*, *degree=2*, *ax=False*)
Given a Pandas Series such as of freq with index Lj, find the Lj that would give the right frequency

pyEPR.toolbox.pythonic.**sort_df_col**(*df*)
sort by numerical int order

pyEPR.toolbox.pythonic.**sort_Series_idx**(*sr*)
sort by numerical int order

pyEPR.toolbox.pythonic.**print_matrix**(*M*, *frmt='{:7.2f}'*, *append_row=''*)

pyEPR.toolbox.pythonic.**print_NoNewLine**(*text*)

pyEPR.toolbox.pythonic.**DataFrame_col_diff**(*PS*, *indx=0*)
check weather the columns of a dataframe are equal, returns a T/F series of the row index that specifies which rows are different USE:

PS[DataFrame_col_diff(PS)]

pyEPR.toolbox.pythonic.**xarray_unravel_levels**(*arr*, *names*, *my_convert=<function <lambda>>*)
Takes in nested dict of dict of dataframes names : names of lists; you dont have to include the last two dataframe columns & rows, but you can to override them requires xarray

pyEPR.toolbox.pythonic.**robust_percentile**(*calc_data*, *ROBUST_PERCENTILE=2.0*)
analysis helper function

Submodules

pyEPR.ansys module

pyEPR.ansys 2014-present

Purpose: Handles Ansys interaction and control from version 2014 onward. Tested most extensively with V2016 and V2019R3.

@authors: Originally contributed by Phil Reinhold. Developed further by Zlatko Minev, Zaki Leghtas, and the pyEPR team. For the base version of hfss.py, see https://github.com/PhilReinhold/pyHFSS

class pyEPR.ansys.**AnsysQ3DSetup**(*design*, *setup: str*)
Bases: *pyEPR.ansys.HfssSetup*

Q3D setup

add_fields_convergence_expr(*expr*, *pct_delta*, *phase=0*)
note: because of hfss idiocy, you must call "commit_convergence_exprs" after adding all exprs

analyze(*name=None*)
Use: Solves a single solution setup and all of its frequency sweeps. Command: Right-click a solution setup in the project tree, and then click Analyze

on the shortcut menu.

Syntax: Analyze(<SetupName>) Parameters: <setupName> Return Value: None

---

Will block the until the analysis is completely done. Will raise a com_error if analysis is aborted in HFSS.

basis_order

commit_convergence_exprs()
note: this will eliminate any convergence expressions not added through this interface

delete_sweep(*name*)

delta_f

frequency

get_convergence(*variation=''*)

Returns df

Triangle Delta %

Pass 1 164 NaN

get_fields()

get_frequency_Hz()

get_matrix(*variation=''*, *pass_number=0*, *frequency=None*, *MatrixType='Maxwell'*, *solution_kind='LastAdaptive'*, *ACPlusDCResistance=False*, *soln_type='C'*)

variation: an empty string returns nominal variation. Otherwise need the list

frequency: in Hz soln_type = "C", "AC RL" and "DC RL" solution_kind = 'LastAdaptive' # AdaptivePass

Uses self.solution_name = Setup1 : LastAdaptive

df_cmat, user_units, (df_cond, units_cond), design_variation

get_mesh_stats(*variation=''*)
variation should be in the form variation = "scale_factor='1.2001'" ...

get_profile(*variation=''*)

get_solutions()

get_sweep(*name=None*)

get_sweep_names()

insert_sweep(*start_ghz*, *stop_ghz*, *count=None*, *step_ghz=None*, *name='Sweep'*, *type='Fast'*, *save_fields=False*)

static load_q3d_matrix(*path*, *user_units='fF'*)
Load Q3D capacitance file exported as Maxwell matrix. Exports also conductance conductance. Units are read in automatically and converted to user units.

Parameters **{[str or Path]} -- [path to file text with matrix]** (*path*) –

Returns

df_cmat, user_units, (df_cond, units_cond), design_variation

dataframes: df_cmat, df_cond

max_pass

min_freq

min_pass

n_modes = 0

passes

pct_error

pct_refinement

prop_holder = None

prop_server = None

prop_tab = 'CG'

release()

solve(*name=None*)

Use: Performs a blocking simulation. The next script command will not be executed until the simulation is complete.

Command: HFSS>Analyze Syntax: Solve <SetupNameArray> Return Value: Type: <int>

-1: simulation error 0: normal completion

Parameters: <SetupNameArray>: Array(<SetupName>, <SetupName>, ...) <SetupName>

Type: <string> Name of the solution setup to solve. .. rubric:: Example

return_status = oDesign.Solve Array("Setup1", "Setup2")

HFSS abort: still returns 0 , since termination by user.

class pyEPR.ansys.**Box**(*name*, *modeler*, *corner*, *size*)
Bases: *pyEPR.ansys.ModelEntity*

capitalize()
Return a capitalized version of the string.

More specifically, make the first character have upper case and the rest lower case.

casefold()
Return a version of the string suitable for caseless comparisons.

center()
Return a centered string of length width.

Padding is done using the specified fill character (default is a space).

coordinate_system

count(*sub*[, *start*[, *end*]]) → int
Return the number of non-overlapping occurrences of substring sub in string S[start:end]. Optional arguments start and end are interpreted as in slice notation.

encode()
Encode the string using the codec registered for encoding.

encoding The encoding in which to encode the string.

errors The error handling scheme to use for encoding errors. The default is 'strict' meaning that encoding errors raise a UnicodeEncodeError. Other possible values are 'ignore', 'replace' and 'xmlcharrefreplace' as well as any other name registered with codecs.register_error that can handle UnicodeEncodeErrors.

endswith(*suffix*[, *start*[, *end*]]) → bool
Return True if S ends with the specified suffix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. suffix can also be a tuple of strings to try.

expandtabs()
Return a copy where all tab characters are expanded using spaces.

If tabsize is not given, a tab size of 8 characters is assumed.

find(*sub*[, *start*[, *end*]]) → int
Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

format(**args*, ***kwargs*) → str
Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

format_map(*mapping*) → str
Return a formatted version of S, using substitutions from mapping. The substitutions are identified by braces ('{' and '}').

index(*sub*[, *start*[, *end*]]) → int
Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the substring is not found.

isalnum()
Return True if the string is an alpha-numeric string, False otherwise.

A string is alpha-numeric if all characters in the string are alpha-numeric and there is at least one character in the string.

isalpha()
Return True if the string is an alphabetic string, False otherwise.

A string is alphabetic if all characters in the string are alphabetic and there is at least one character in the string.

isascii()
Return True if all characters in the string are ASCII, False otherwise.

ASCII characters have code points in the range U+0000-U+007F. Empty string is ASCII too.

isdecimal()
Return True if the string is a decimal string, False otherwise.

A string is a decimal string if all characters in the string are decimal and there is at least one character in the string.

isdigit()
Return True if the string is a digit string, False otherwise.

A string is a digit string if all characters in the string are digits and there is at least one character in the string.

isidentifier()
Return True if the string is a valid Python identifier, False otherwise.

Use keyword.iskeyword() to test for reserved identifiers such as “def” and “class”.

islower()
Return True if the string is a lowercase string, False otherwise.

A string is lowercase if all cased characters in the string are lowercase and there is at least one cased character in the string.

isnumeric()
Return True if the string is a numeric string, False otherwise.

A string is numeric if all characters in the string are numeric and there is at least one character in the string.

isprintable()
Return True if the string is printable, False otherwise.

A string is printable if all of its characters are considered printable in repr() or if it is empty.

isspace()
Return True if the string is a whitespace string, False otherwise.

A string is whitespace if all characters in the string are whitespace and there is at least one character in the string.

istitle()
Return True if the string is a title-cased string, False otherwise.

In a title-cased string, upper- and title-case characters may only follow uncased characters and lowercase characters only cased ones.

isupper()
Return True if the string is an uppercase string, False otherwise.

A string is uppercase if all cased characters in the string are uppercase and there is at least one cased character in the string.

join()
Concatenate any number of strings.

The string whose method is called is inserted in between each given string. The result is returned as a new string.

Example: '.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'

ljust()
Return a left-justified string of length width.

Padding is done using the specified fill character (default is a space).

lower()
Return a copy of the string converted to lowercase.

lstrip()
Return a copy of the string with leading whitespace removed.

If chars is given and not None, remove characters in chars instead.

static maketrans()
Return a translation table usable for str.translate().

If there is only one argument, it must be a dictionary mapping Unicode ordinals (integers) or characters to Unicode ordinals, strings or None. Character keys will be then converted to ordinals. If there are two arguments, they must be strings of equal length, and in the resulting dictionary, each character in x will be mapped to the character at the same position in y. If there is a third argument, it must be a string, whose characters will be mapped to None in the result.

material

model_command = 'CreateBox'

partition()
Partition the string into three parts using the given separator.

This will search for the separator in the string. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing the original string and two empty strings.

position

prop_holder = None

prop_server = None

prop_tab = 'Geometry3DCmdTab'

release()

replace()
Return a copy with all occurrences of substring old replaced by new.

count Maximum number of occurrences to replace. -1 (the default value) means replace all occurrences.

If the optional argument count is given, only the first count occurrences are replaced.

rfind(*sub*[, *start*[, *end*]]) → int
Return the highest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

rindex(*sub*[, *start*[, *end*]]) → int
Return the highest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the substring is not found.

rjust()
Return a right-justified string of length width.

Padding is done using the specified fill character (default is a space).

rpartition()
Partition the string into three parts using the given separator.

This will search for the separator in the string, starting at the end. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing two empty strings and the original string.

rsplit()
Return a list of the words in the string, using sep as the delimiter string.

sep The delimiter according which to split the string. None (the default value) means split according to any whitespace, and discard empty strings from the result.

maxsplit Maximum number of splits to do. -1 (the default value) means no limit.

Splits are done starting at the end of the string and working to the front.

rstrip()
Return a copy of the string with trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

split()
Return a list of the words in the string, using sep as the delimiter string.

sep The delimiter according which to split the string. None (the default value) means split according to any whitespace, and discard empty strings from the result.

maxsplit Maximum number of splits to do. -1 (the default value) means no limit.

splitlines()
Return a list of the lines in the string, breaking at line boundaries.

Line breaks are not included in the resulting list unless keepends is given and true.

startswith(*prefix*[, *start*[, *end*]]) → bool
Return True if S starts with the specified prefix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. prefix can also be a tuple of strings to try.

strip()
Return a copy of the string with leading and trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

swapcase()
Convert uppercase characters to lowercase and lowercase characters to uppercase.

title()
Return a version of the string where each word is titlecased.

More specifically, words start with uppercased characters and all remaining cased characters have lower case.

translate()
Replace each character in the string using the given translation table.

table Translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, strings, or None.

The table must implement lookup/indexing via __getitem__, for instance a dictionary or list. If this operation raises LookupError, the character is left untouched. Characters mapped to None are deleted.

transparency

upper()
Return a copy of the string converted to uppercase.

wireframe

x_size

y_size

z_size

zfill()
Pad a numeric string with zeros on the left, to fill a field of the given width.

The string is never truncated.

class pyEPR.ansys.**COMWrapper**
Bases: object

release()

class pyEPR.ansys.**CalcObject**(*stack*, *setup*)
Bases: *pyEPR.ansys.COMWrapper*

complexmag()

conj()

dot(*other*)

evaluate(*phase=0*, *lv=None*, *print_debug=False*)

getQty(*name*)

imag()

integrate_line(*name*)

integrate_line_tangent(*name*)
integrate line tangent to vector expression

name = of line to integrate over

integrate_surf(*name='AllObjects'*)

integrate_vol(*name='AllObjects'*)

line_tangent_coor(*name*, *coordinate*)
integrate line tangent to vector expression

name = of line to integrate over

mag()

maximum_vol(*name='AllObjects'*)

norm_2()

normal2surface(*name*)
return the part normal to surface. Complex Vector.

real()

release()

save_as(*name*)
if the object already exists, try clearing your named expressions first with fields.clear_named_expressions

scalar_x()

scalar_y()

scalar_z()

smooth()

tangent2surface(*name*)
return the part tangent to surface. Complex Vector.

times_eps()

times_mu()

write_stack()

class pyEPR.ansys.**ConstantCalcObject**(*num*, *setup*)
Bases: *pyEPR.ansys.CalcObject*

complexmag()

conj()

dot(*other*)

evaluate(*phase=0*, *lv=None*, *print_debug=False*)

getQty(*name*)

imag()

integrate_line(*name*)

integrate_line_tangent(*name*)
integrate line tangent to vector expression

name = of line to integrate over

integrate_surf(*name='AllObjects'*)

integrate_vol(*name='AllObjects'*)

line_tangent_coor(*name*, *coordinate*)
integrate line tangent to vector expression

name = of line to integrate over

mag()

maximum_vol(*name='AllObjects'*)

norm_2()

normal2surface(*name*)
return the part normal to surface. Complex Vector.

real()

release()

save_as(*name*)
if the object already exists, try clearing your named expressions first with fields.clear_named_expressions

scalar_x()

scalar_y()

scalar_z()

smooth()

tangent2surface(*name*)
return the part tangent to surface. Complex Vector.

times_eps()

times_mu()

write_stack()

class pyEPR.ansys.**ConstantVecCalcObject**(*vec*, *setup*)
Bases: *pyEPR.ansys.CalcObject*

complexmag()

conj()

dot(*other*)

evaluate(*phase=0*, *lv=None*, *print_debug=False*)

getQty(*name*)

imag()

integrate_line(*name*)

integrate_line_tangent(*name*)
integrate line tangent to vector expression

name = of line to integrate over

integrate_surf(*name='AllObjects'*)

integrate_vol(*name='AllObjects'*)

line_tangent_coor(*name*, *coordinate*)
integrate line tangent to vector expression

name = of line to integrate over

mag()

maximum_vol(*name='AllObjects'*)

norm_2()

normal2surface(*name*)
return the part normal to surface. Complex Vector.

real()

release()

save_as(*name*)
if the object already exists, try clearing your named expressions first with fields.clear_named_expressions

scalar_x()

scalar_y()

scalar_z()

smooth()

tangent2surface(*name*)
return the part tangent to surface. Complex Vector.

times_eps()

times_mu()

write_stack()

class pyEPR.ansys.**HfssApp**(*ProgID='AnsoftHfss.HfssScriptInterface'*)
Bases: *pyEPR.ansys.COMWrapper*

get_app_desktop()

release()

class pyEPR.ansys.**HfssDMDesignSolutions**(*setup*, *solutions*)
Bases: *pyEPR.ansys.HfssDesignSolutions*

get_valid_solution_list()
Gets all available solution names that exist in a design. Return example:

('Setup1 : AdaptivePass', 'Setup1 : LastAdaptive')

list_variations(*setup_name: str = None*)
Get a list of solved variations.

Parameters **setup_name** (*str*) – Example name ("Setup1 : LastAdaptive") Defaults to None.

Returns

An array of strings corresponding to solved variations.

```
("Cj='2fF' Lj='12nH'",
"Cj='2fF' Lj='12.5nH'",
"Cj='2fF' Lj='13nH'",
"Cj='2fF' Lj='13.5nH'",
"Cj='2fF' Lj='14nH'")
```

release()

class pyEPR.ansys.**HfssDMSetup**(*design*, *setup: str*)
Bases: *pyEPR.ansys.HfssSetup*

Driven modal setup

add_fields_convergence_expr(*expr*, *pct_delta*, *phase=0*)
note: because of hfss idiocy, you must call "commit_convergence_exprs" after adding all exprs

analyze(*name=None*)
Use: Solves a single solution setup and all of its frequency sweeps. Command: Right-click a solution setup in the project tree, and then click Analyze

on the shortcut menu.

Syntax: Analyze(<SetupName>) Parameters: <setupName> Return Value: None

---

Will block the until the analysis is completely done. Will raise a com_error if analysis is aborted in HFSS.

basis_order

commit_convergence_exprs()
note: this will eliminate any convergence expressions not added through this interface

delete_sweep(*name*)

delta_f

delta_s

get_convergence(*variation=''*, *pre_fn_args=[]*, *overwrite=True*)

Returns converge as a dataframe Variation should be in the form variation = "scale_factor='1.2001'" ...

get_fields()

get_mesh_stats(*variation=''*)
variation should be in the form variation = "scale_factor='1.2001'" ...

get_profile(*variation=''*)

get_solutions()

get_sweep(*name=None*)

get_sweep_names()

insert_sweep(*start_ghz*, *stop_ghz*, *count=None*, *step_ghz=None*, *name='Sweep'*, *type='Fast'*, *save_fields=False*)

min_freq

n_modes

passes

pct_refinement

prop_holder = None

prop_server = None

prop_tab = 'HfssTab'

release()

setup_link(*linked_setup*)
type: linked_setup <HfssSetup>

solution_freq

solve(*name=None*)

Use: Performs a blocking simulation. The next script command will not be executed until the simulation is complete.

Command: HFSS>Analyze Syntax: Solve <SetupNameArray> Return Value: Type: <int>

-1: simulation error 0: normal completion

Parameters: <SetupNameArray>: Array(<SetupName>, <SetupName>, ...) <SetupName>

Type: <string> Name of the solution setup to solve. .. rubric:: Example

return_status = oDesign.Solve Array("Setup1", "Setup2")

HFSS abort: still returns 0 , since termination by user.

solver_type

class pyEPR.ansys.**HfssDTDesignSolutions**(*setup*, *solutions*)
Bases: *pyEPR.ansys.HfssDesignSolutions*

get_valid_solution_list()
Gets all available solution names that exist in a design. Return example:

('Setup1 : AdaptivePass', 'Setup1 : LastAdaptive')

list_variations(*setup_name: str = None*)
Get a list of solved variations.

Parameters **setup_name** (*str*) – Example name ("Setup1 : LastAdaptive") Defaults to None.

Returns

An array of strings corresponding to solved variations.

```
("Cj='2fF' Lj='12nH'",
"Cj='2fF' Lj='12.5nH'",
"Cj='2fF' Lj='13nH'",
"Cj='2fF' Lj='13.5nH'",
"Cj='2fF' Lj='14nH'")
```

release()

class pyEPR.ansys.**HfssDTSetup**(*design*, *setup: str*)
Bases: *pyEPR.ansys.HfssDMSetup*

add_fields_convergence_expr(*expr*, *pct_delta*, *phase=0*)
note: because of hfss idiocy, you must call "commit_convergence_exprs" after adding all exprs

analyze(*name=None*)
Use: Solves a single solution setup and all of its frequency sweeps. Command: Right-click a solution setup in the project tree, and then click Analyze

on the shortcut menu.

Syntax: Analyze(<SetupName>) Parameters: <setupName> Return Value: None

---

Will block the until the analysis is completely done. Will raise a com_error if analysis is aborted in HFSS.

basis_order

commit_convergence_exprs()
note: this will eliminate any convergence expressions not added through this interface

delete_sweep(*name*)

delta_f

delta_s

get_convergence(*variation=''*, *pre_fn_args=[]*, *overwrite=True*)

Returns converge as a dataframe Variation should be in the form variation = "scale_factor='1.2001'" ...

get_fields()

get_mesh_stats(*variation=''*)
variation should be in the form variation = "scale_factor='1.2001'" ...

get_profile(*variation=''*)

get_solutions()

get_sweep(*name=None*)

get_sweep_names()

insert_sweep(*start_ghz*, *stop_ghz*, *count=None*, *step_ghz=None*, *name='Sweep'*, *type='Fast'*, *save_fields=False*)

min_freq

n_modes

passes

pct_refinement

prop_holder = None

prop_server = None

prop_tab = 'HfssTab'

release()

setup_link(*linked_setup*)
type: linked_setup <HfssSetup>

solution_freq

solve(*name=None*)

Use: Performs a blocking simulation. The next script command will not be executed until the simulation is complete.

Command: HFSS>Analyze Syntax: Solve <SetupNameArray> Return Value: Type: <int>

-1: simulation error 0: normal completion

Parameters: <SetupNameArray>: Array(<SetupName>, <SetupName>, ...) <SetupName>

Type: <string> Name of the solution setup to solve. .. rubric:: Example

return_status = oDesign.Solve Array("Setup1", "Setup2")

HFSS abort: still returns 0 , since termination by user.

solver_type

class pyEPR.ansys.**HfssDesign**(*project*, *design*)
Bases: *pyEPR.ansys.COMWrapper*

Clear_Field_Clac_Stack()

add_message(*message: str*, *severity: int = 0*)
Add a message to HFSS log with severity and context to message window.

Keyword Arguments **severity** (*int*) – 0 = Informational, 1 = Warning, 2 = Error, 3 = Fatal..

clean_up_solutions()

copy_design_variables(*source_design*)
does not check that variables are all present

copy_to_project(*project*)

create_dm_setup(*freq_ghz=1*, *name='Setup'*, *max_delta_s=0.1*, *max_passes=10*, *min_passes=1*, *min_converged=1*, *pct_refinement=30*, *basis_order=-1*)

create_dt_setup(*freq_ghz=1*, *name='Setup'*, *max_delta_s=0.1*, *max_passes=10*, *min_passes=1*, *min_converged=1*, *pct_refinement=30*, *basis_order=-1*)

create_em_setup(*name='Setup'*, *min_freq_ghz=1*, *n_modes=1*, *max_delta_f=0.1*, *max_passes=10*, *min_passes=1*, *min_converged=1*, *pct_refinement=30*, *basis_order=-1*)

create_q3d_setup(*freq_ghz=5.0*, *name='Setup'*, *save_fields=False*, *enabled=True*, *max_passes=15*, *min_passes=2*, *min_converged_passes=2*, *percent_error=0.5*, *percent_refinement=30*, *auto_increase_solution_order=True*, *solution_order='High'*, *solver_type='Iterative'*)

create_variable(*name*, *value*, *postprocessing=False*)

delete_full_variation(*DesignVariationKey='All'*, *del_linked_data=False*)

DeleteFullVariation Use: Use to selectively make deletions or delete all solution data. Command: HFSS>Results>Clean Up Solutions... Syntax: DeleteFullVariation Array(<parameters>), boolean Parameters: All | <DataSpecifierArray>

If, All, all data of existing variations is deleted. Array(<DesignVariationKey>,) <DesignVariationKey>

Type: <string> Design variation string.

<Boolean> Type: boolean Whether to also delete linked data.

delete_setup(*name*)

duplicate(*name=None*)

eval_expr(*expr*, *units='mm'*)

get_excitations()

get_nominal_variation()

Use: Gets the nominal variation string Return Value: Returns a string representing the nominal variation Returns string such as "Height='0.06mm' Lj='13.5nH'"

get_setup(*name=None*)

Return type *HfssSetup*

get_setup_names()

get_variable_names()

Returns the local design variables. Does not return the project (global) variables, which start with $.

get_variable_value(*name*)

Can only access the design variables, i.e., the local ones Cannot access the project (global) variables, which start with $.

get_variables()

Returns dictionary of local design variables and their values. Does not return the project (global) variables and their values, whose names start with $.

release()

rename_design(*name*)

save_screenshot(*path: str = None*, *show: bool = True*)

set_variable(*name: str*, *value: str*, *postprocessing=False*)

Warning: THis is case sensitive,

Parameters

- **{str} -- Name of variable to set, such as 'Lj_1'.** (*name*) – This is not the same as as 'LJ_1'. You must use the same casing.
- **{str} -- Value, such as '10nH'** (*value*) –

Keyword Arguments **{bool} -- Postprocessing variable only or not.** (*postprocessing*) – (default: {False})

Returns VariableString

set_variables(*variation_string: str*)
Set all variables to match a solved variation string.

Parameters **variation_string** (*str*) – Variation string such as "Cj='2fF' Lj='13.5nH'"

class pyEPR.ansys.**HfssDesignSolutions**(*setup*, *solutions*)
Bases: *pyEPR.ansys.COMWrapper*

get_valid_solution_list()
Gets all available solution names that exist in a design. Return example:

('Setup1 : AdaptivePass', 'Setup1 : LastAdaptive')

list_variations(*setup_name: str = None*)
Get a list of solved variations.

Parameters **setup_name** (*str*) – Example name ("Setup1 : LastAdaptive") Defaults to None.

Returns

An array of strings corresponding to solved variations.

```
("Cj='2fF' Lj='12nH'",
"Cj='2fF' Lj='12.5nH'",
"Cj='2fF' Lj='13nH'",
"Cj='2fF' Lj='13.5nH'",
"Cj='2fF' Lj='14nH'")
```

release()

class pyEPR.ansys.**HfssDesktop**(*app*, *desktop*)
Bases: *pyEPR.ansys.COMWrapper*

close_all_windows()

get_active_project()

get_messages(*project_name=''*, *design_name=''*, *level=0*)
Use: Collects the messages from a specified project and design. Syntax: GetMessages <ProjectName>, <DesignName>, <SeverityName> Return Value: A simple array of strings.

Parameters: <ProjectName>

Type:<string> Name of the project for which to collect messages. An incorrect project name results in no messages (design is ignored) An empty project name results in all messages (design is ignored)

<DesignName> Type: <string> Name of the design in the named project for which to collect messages An incorrect design name results in no messages for the named project An empty design name results in all messages for the named project

<SeverityName> Type: <integer> Severity is 0-3, and is tied in to info/warning/error/fatal types as follows:

0 is info and above 1 is warning and above 2 is error and fatal 3 is fatal only (rarely used)

get_project_names()

get_projects()

get_version()

library_directory

new_project()

open_project(*path*)
returns error if already open

project_count()

project_directory

release()

set_active_project(*name*)

temp_directory

class pyEPR.ansys.**HfssEMDesignSolutions**(*setup*, *solutions*)
Bases: *pyEPR.ansys.HfssDesignSolutions*

create_report(*plot_name*, *xcomp*, *ycomp*, *params*, *pass_name='LastAdaptive'*)
pass_name: AdaptivePass, LastAdaptive

Example

Example plot for a single variation all pass converge of mode freq

```
ycomp = [f"re(Mode({i}))" for i in range(1,1+epr_hfss.n_modes)]
params = ["Pass:=", ["All"]]+variation
setup.create_report("Freq. vs. pass", "Pass", ycomp, params, pass_
↪name='AdaptivePass')
```

eigenmodes(*lv=''*)
Returns the eigenmode data of freq and kappa/2p

get_valid_solution_list()
Gets all available solution names that exist in a design. Return example:

('Setup1 : AdaptivePass', 'Setup1 : LastAdaptive')

has_fields(*variation_string=None*)
Determine if fields exist for a particular solution.

variation_string [str | None] This must the string that describes the variation in hFSS, not 0 or 1, but the string of variables, such as

"Cj='2fF' Lj='12.75nH'"

If None, gets the nominal variation

list_variations(*setup_name: str = None*)
Get a list of solved variations.

Parameters **setup_name** (*str*) – Example name ("Setup1 : LastAdaptive") Defaults to None.

Returns

An array of strings corresponding to solved variations.

```
("Cj='2fF' Lj='12nH'",
"Cj='2fF' Lj='12.5nH'",
"Cj='2fF' Lj='13nH'",
"Cj='2fF' Lj='13.5nH'",
"Cj='2fF' Lj='14nH'")
```

release()

set_mode(*n*, *phase=0*, *FieldType='EigenStoredEnergy'*)

Indicates which source excitations should be used for fields post processing. HFSS>Fields>Edit Sources

Mode count starts at 1

Amplitude is set to 1

No error is thrown if a number exceeding number of modes is set

FieldType – EigenStoredEnergy or EigenPeakElecticField

class pyEPR.ansys.**HfssEMSetup**(*design*, *setup: str*)

Bases: *pyEPR.ansys.HfssSetup*

Eigenmode setup

add_fields_convergence_expr(*expr*, *pct_delta*, *phase=0*)

note: because of hfss idiocy, you must call "commit_convergence_exprs" after adding all exprs

analyze(*name=None*)

Use: Solves a single solution setup and all of its frequency sweeps. Command: Right-click a solution setup in the project tree, and then click Analyze

on the shortcut menu.

Syntax: Analyze(<SetupName>) Parameters: <setupName> Return Value: None

———————————————

Will block the until the analysis is completely done. Will raise a com_error if analysis is aborted in HFSS.

basis_order

commit_convergence_exprs()

note: this will eliminate any convergence expressions not added through this interface

delete_sweep(*name*)

delta_f

get_convergence(*variation=''*, *pre_fn_args=[]*, *overwrite=True*)

Returns converge as a dataframe Variation should be in the form variation = "scale_factor='1.2001'" ...

get_fields()

get_mesh_stats(*variation=''*)

variation should be in the form variation = "scale_factor='1.2001'" ...

get_profile(*variation=''*)

get_solutions()

get_sweep(*name=None*)

get_sweep_names()

insert_sweep(*start_ghz*, *stop_ghz*, *count=None*, *step_ghz=None*, *name='Sweep'*, *type='Fast'*, *save_fields=False*)

min_freq

n_modes

passes

pct_refinement

prop_holder = None

prop_server = None

prop_tab = 'HfssTab'

release()

solve(*name=None*)

Use: Performs a blocking simulation. The next script command will not be executed until the simulation is complete.

Command: HFSS>Analyze Syntax: Solve <SetupNameArray> Return Value: Type: <int>

-1: simulation error 0: normal completion

Parameters: <SetupNameArray>: Array(<SetupName>, <SetupName>, ...) <SetupName>

Type: <string> Name of the solution setup to solve. .. rubric:: Example

return_status = oDesign.Solve Array("Setup1", "Setup2")

HFSS abort: still returns 0 , since termination by user.

class pyEPR.ansys.**HfssFieldsCalc**(*setup*)

Bases: *pyEPR.ansys.COMWrapper*

clear_named_expressions()

declare_named_expression(*name*)

" If a named expression has been created in the fields calculator, this function can be called to initialize the name to work with the fields object

release()

use_named_expression(*name*)

Expression can be used to access dictionary of named expressions, Alternately user can access dictionary directly via named_expression()

class pyEPR.ansys.**HfssFrequencySweep**(*setup*, *name*)

Bases: *pyEPR.ansys.COMWrapper*

analyze_sweep()

count

create_report(*name*, *expr*)

get_network_data(*formats*)

get_report_arrays(*expr*)

prop_tab = 'HfssTab'

release()

start_freq

step_size

stop_freq

sweep_type

class pyEPR.ansys.**HfssModeler**(*design*, *modeler*, *boundaries*, *mesh*)
Bases: *pyEPR.ansys.COMWrapper*

append_PerfE_assignment(*boundary_name: str*, *object_names: list*)
This will create a new boundary if need, and will otherwise append given names to an existing boundary

append_mesh(*mesh_name: str*, *object_names: list*, *old_objs: list*, ***kwargs*)
This will create a new boundary if need, and will otherwise append given names to an existing boundary old_obj = circ._mesh_assign

assign_perfect_E(*obj: List[str]*, *name: str = 'PerfE'*)
Assign a boundary condition to a list of objects.

Arg: objs (List[str]): Takes a name of an object or a list of object names. name(str): If *name* is not specified *PerfE* is appended to object name for the name.

create_relative_coorinate_system_both(*cs_name*, *origin=['0um', '0um', '0um']*, *XAxisVec=['1um', '0um', '0um']*, *YAxisVec=['0um', '1um', '0um']*)
Use: Creates a relative coordinate system. Only the Name attribute of the <AttributesArray> parameter is supported. Command: Modeler>Coordinate System>Create>Relative CS->Offset Modeler>Coordinate System>Create>Relative CS->Rotated Modeler>Coordinate System>Create>Relative CS->Both

Current coordinate system is set right after this.

cs_name [name of coord. sys] If the name already exists, then a new coordinate system with _1 is created.

origin, XAxisVec, YAxisVec: 3-vectors You can also pass in params such as origin = [0,1,0] rather than ["0um","1um","0um"], but these will be interpreted in default units, so it is safer to be explicit. Explicit over implicit.

draw_box_center(*pos*, *size*, ***kwargs*)
Creates a 3-D box centered at pos [x0, y0, z0], with width size [xwidth, ywidth, zwidth] along each respective direction.

Parameters

- **pos** (*list*) – Coordinates of center of box, [x0, y0, z0]
- **size** (*list*) – Width of box along each direction, [xwidth, ywidth, zwidth]

draw_box_corner(*pos*, *size*, ***kwargs*)

draw_cylinder(*pos*, *radius*, *height*, *axis*, ***kwargs*)

draw_cylinder_center(*pos*, *radius*, *height*, *axis*, ***kwargs*)

draw_polyline(*points*, *closed=True*, ***kwargs*)
Draws a closed or open polyline. If closed = True, then will make into a sheet. points : need to be in the correct units

For optional arguments, see _attributes_array; these include: ‘‘‘

nonmodel=False, wireframe=False, color=None, transparency=0.9, material=None, # str solve_inside=None, # bool coordinate_system="Global"

‘‘‘

draw_rect_center(*pos*, *x_size=0*, *y_size=0*, *z_size=0*, ***kwargs*)
Creates a rectangle centered at pos [x0, y0, z0]. It is assumed that the rectangle lies parallel to the xy, yz, or xz plane. User inputs 2 of 3 of the following: x_size, y_size, and z_size depending on how the rectangle is oriented.

Parameters

- **pos** (*list*) – Coordinates of rectangle center, [x0, y0, z0]
- **x_size** (*int, optional*) – Width along the x direction. Defaults to 0.
- **y_size** (*int, optional*) – Width along the y direction. Defaults to 0.
- **z_size** (*int, optional*) – Width along the z direction]. Defaults to 0.

draw_rect_corner(*pos*, *x_size=0*, *y_size=0*, *z_size=0*, ***kwargs*)

draw_region(*Padding*, *PaddingType='Percentage Offset'*, *name='Region'*, *material='"vacuum"'*)
PaddingType : ‘Absolute Offset’, “Percentage Offset”

draw_wirebond(*pos*, *ori*, *width*, *height='0.1mm'*, *z=0*, *wire_diameter='0.02mm'*, *NumSides=6*, ***kwargs*)

Parameters

- **pos** – 2D position vector (specify center point)
- **ori** – should be normed
- **z** – z position

TODO create Wirebond class position is the origin of one point ori is the orientation vector, which gets normalized

eval_expr(*expr*, *units='mm'*)

get_all_properties(*obj_name*, *PropTab='Geometry3DAttributeTab'*)
Get all properties for modeler PropTab, PropServer

get_boundary_assignment(*boundary_name: str*)

get_face_ids(*obj*)

get_object_name_by_face_id(*ID: str*)
Gets an object name corresponding to the input face id.

get_objects_in_group(*group*)
Use: Returns the objects for the specified group. Return Value: The objects in the group. Parameters: <groupName> Type: <string> One of <materialName>, <assignmentName>, “Non Model”,

“Solids”, “Unclassified”, “Sheets”, “Lines”

get_units()
Get the model units. Return Value: A string contains current model units.

get_vertex_ids(*obj*)
Get the vertex IDs of given an object name oVertexIDs = oEditor.GetVertexIDsFromObject(“Box1”)

intersect(*names*, *keep_originals=False*)

mesh_get_all_props(*mesh_name*)

mesh_get_names(*kind='Length Based'*)
“Length Based”, “Skin Depth Based”, . . .

mesh_length(*name_mesh*, *objects: list*, *MaxLength='0.1mm'*, ***kwargs*)
"RefineInside:=" , False, "Enabled:=" , True, "RestrictElem:=" , False, "NumMaxElem:=" , "1000", "RestrictLength:=" , True, "MaxLength:=" , "0.1mm"

Example use: modeler.assign_mesh_length('mesh2', ["Q1_mesh"], MaxLength=0.1)

mesh_reassign(*name_mesh*, *objects: list*)

release()

rename_obj(*obj*, *name*)

set_units(*units*, *rescale=True*)

set_working_coordinate_system(*cs_name='Global'*)
Use: Sets the working coordinate system. Command: Modeler>Coordinate System>Set Working CS

subtract(*blank_name*, *tool_names*, *keep_originals=False*)

sweep_along_vector(*names*, *vector*)

translate(*name*, *vector*)

unite(*names*, *keep_originals=False*)

class pyEPR.ansys.**HfssProject**(*desktop*, *project*)
Bases: *pyEPR.ansys.COMWrapper*

close()

create_variable(*name*, *value*)

duplicate_design(*target*, *source*)

get_active_design()

get_design(*name*)

get_design_names()

get_designs()

get_path()

get_variable_names()

get_variable_value(*name*)

get_variables()
Returns the project variables only, which start with $. These are global variables.

import_dataset(*path*)

make_active()

name

new_design(*design_name*, *solution_type*, *design_type='HFSS'*)

new_dm_design(*name: str*)
Create a new driven model design

Parameters **name** (*str*) – Name of driven modal design

new_em_design(*name: str*)
Create a new eigenmode design

Parameters **name** (*str*) – Name of eigenmode design

new_q3d_design(*name: str*)
Create a new Q3D design. :param name: Name of Q3D design :type name: str

release()

rename_design(*design*, *rename*)

save(*path=None*)

set_variable(*name*, *value*)

simulate_all()

class pyEPR.ansys.**HfssPropertyObject**
Bases: *pyEPR.ansys.COMWrapper*

prop_holder = None

prop_server = None

prop_tab = None

release()

class pyEPR.ansys.**HfssQ3DDesignSolutions**(*setup*, *solutions*)
Bases: *pyEPR.ansys.HfssDesignSolutions*

get_valid_solution_list()
Gets all available solution names that exist in a design. Return example:

('Setup1 : AdaptivePass', 'Setup1 : LastAdaptive')

list_variations(*setup_name: str = None*)
Get a list of solved variations.

Parameters **setup_name** (*str*) – Example name ("Setup1 : LastAdaptive") Defaults to None.

Returns

An array of strings corresponding to solved variations.

```
("Cj='2fF' Lj='12nH'",
"Cj='2fF' Lj='12.5nH'",
"Cj='2fF' Lj='13nH'",
"Cj='2fF' Lj='13.5nH'",
"Cj='2fF' Lj='14nH'")
```

release()

class pyEPR.ansys.**HfssReport**(*design*, *name*)
Bases: *pyEPR.ansys.COMWrapper*

export_to_file(*filename*)

get_arrays()

release()

class pyEPR.ansys.**HfssSetup**(*design*, *setup: str*)
Bases: *pyEPR.ansys.HfssPropertyObject*

add_fields_convergence_expr(*expr*, *pct_delta*, *phase=0*)
note: because of hfss idiocy, you must call "commit_convergence_exprs" after adding all exprs

analyze(*name=None*)
Use: Solves a single solution setup and all of its frequency sweeps. Command: Right-click a solution setup in the project tree, and then click Analyze

on the shortcut menu.

Syntax: Analyze(<SetupName>) Parameters: <setupName> Return Value: None

---

Will block the until the analysis is completely done. Will raise a com_error if analysis is aborted in HFSS.

basis_order

commit_convergence_exprs()
note: this will eliminate any convergence expressions not added through this interface

delete_sweep(*name*)

delta_f

get_convergence(*variation=''*, *pre_fn_args=[]*, *overwrite=True*)

Returns converge as a dataframe Variation should be in the form variation = "scale_factor='1.2001'" ...

get_fields()

get_mesh_stats(*variation=''*)
variation should be in the form variation = "scale_factor='1.2001'" ...

get_profile(*variation=''*)

get_sweep(*name=None*)

get_sweep_names()

insert_sweep(*start_ghz*, *stop_ghz*, *count=None*, *step_ghz=None*, *name='Sweep'*, *type='Fast'*, *save_fields=False*)

min_freq

n_modes

passes

pct_refinement

prop_holder = None

prop_server = None

prop_tab = 'HfssTab'

release()

solve(*name=None*)

Use: Performs a blocking simulation. The next script command will not be executed until the simulation is complete.

Command: HFSS>Analyze Syntax: Solve <SetupNameArray> Return Value: Type: <int>

-1: simulation error 0: normal completion

Parameters: <SetupNameArray>: Array(<SetupName>, <SetupName>, ...) <SetupName>

Type: <string> Name of the solution setup to solve. .. rubric:: Example

return_status = oDesign.Solve Array("Setup1", "Setup2")

HFSS abort: still returns 0 , since termination by user.

class pyEPR.ansys.**ModelEntity**(*val*, *modeler*)

Bases: str, *pyEPR.ansys.HfssPropertyObject*

capitalize()

Return a capitalized version of the string.

More specifically, make the first character have upper case and the rest lower case.

casefold()

Return a version of the string suitable for caseless comparisons.

center()

Return a centered string of length width.

Padding is done using the specified fill character (default is a space).

coordinate_system

count(*sub*[, *start*[, *end*]]) → int

Return the number of non-overlapping occurrences of substring sub in string S[start:end]. Optional arguments start and end are interpreted as in slice notation.

encode()

Encode the string using the codec registered for encoding.

encoding The encoding in which to encode the string.

errors The error handling scheme to use for encoding errors. The default is 'strict' meaning that encoding errors raise a UnicodeEncodeError. Other possible values are 'ignore', 'replace' and 'xmlcharrefreplace' as well as any other name registered with codecs.register_error that can handle UnicodeEncodeErrors.

endswith(*suffix*[, *start*[, *end*]]) → bool

Return True if S ends with the specified suffix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. suffix can also be a tuple of strings to try.

expandtabs()

Return a copy where all tab characters are expanded using spaces.

If tabsize is not given, a tab size of 8 characters is assumed.

find(*sub*[, *start*[, *end*]]) → int

Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

format(**args*, ***kwargs*) → str

Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

format_map(*mapping*) → str

Return a formatted version of S, using substitutions from mapping. The substitutions are identified by braces ('{' and '}').

index(*sub*[, *start*[, *end*]]) → int
Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the substring is not found.

isalnum()
Return True if the string is an alpha-numeric string, False otherwise.

A string is alpha-numeric if all characters in the string are alpha-numeric and there is at least one character in the string.

isalpha()
Return True if the string is an alphabetic string, False otherwise.

A string is alphabetic if all characters in the string are alphabetic and there is at least one character in the string.

isascii()
Return True if all characters in the string are ASCII, False otherwise.

ASCII characters have code points in the range U+0000-U+007F. Empty string is ASCII too.

isdecimal()
Return True if the string is a decimal string, False otherwise.

A string is a decimal string if all characters in the string are decimal and there is at least one character in the string.

isdigit()
Return True if the string is a digit string, False otherwise.

A string is a digit string if all characters in the string are digits and there is at least one character in the string.

isidentifier()
Return True if the string is a valid Python identifier, False otherwise.

Use keyword.iskeyword() to test for reserved identifiers such as “def” and “class”.

islower()
Return True if the string is a lowercase string, False otherwise.

A string is lowercase if all cased characters in the string are lowercase and there is at least one cased character in the string.

isnumeric()
Return True if the string is a numeric string, False otherwise.

A string is numeric if all characters in the string are numeric and there is at least one character in the string.

isprintable()
Return True if the string is printable, False otherwise.

A string is printable if all of its characters are considered printable in repr() or if it is empty.

isspace()
Return True if the string is a whitespace string, False otherwise.

A string is whitespace if all characters in the string are whitespace and there is at least one character in the string.

istitle()
Return True if the string is a title-cased string, False otherwise.

In a title-cased string, upper- and title-case characters may only follow uncased characters and lowercase characters only cased ones.

isupper()
Return True if the string is an uppercase string, False otherwise.

A string is uppercase if all cased characters in the string are uppercase and there is at least one cased character in the string.

join()
Concatenate any number of strings.

The string whose method is called is inserted in between each given string. The result is returned as a new string.

Example: '.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'

ljust()
Return a left-justified string of length width.

Padding is done using the specified fill character (default is a space).

lower()
Return a copy of the string converted to lowercase.

lstrip()
Return a copy of the string with leading whitespace removed.

If chars is given and not None, remove characters in chars instead.

static maketrans()
Return a translation table usable for str.translate().

If there is only one argument, it must be a dictionary mapping Unicode ordinals (integers) or characters to Unicode ordinals, strings or None. Character keys will be then converted to ordinals. If there are two arguments, they must be strings of equal length, and in the resulting dictionary, each character in x will be mapped to the character at the same position in y. If there is a third argument, it must be a string, whose characters will be mapped to None in the result.

material

model_command = None

partition()
Partition the string into three parts using the given separator.

This will search for the separator in the string. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing the original string and two empty strings.

prop_holder = None

prop_server = None

prop_tab = 'Geometry3DCmdTab'

release()

replace()
Return a copy with all occurrences of substring old replaced by new.

count Maximum number of occurrences to replace. -1 (the default value) means replace all occurrences.

If the optional argument count is given, only the first count occurrences are replaced.

rfind(*sub*[, *start*[, *end*]]) → int
Return the highest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

rindex(*sub*[, *start*[, *end*]]) → int
Return the highest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the substring is not found.

rjust()
Return a right-justified string of length width.

Padding is done using the specified fill character (default is a space).

rpartition()
Partition the string into three parts using the given separator.

This will search for the separator in the string, starting at the end. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing two empty strings and the original string.

rsplit()
Return a list of the words in the string, using sep as the delimiter string.

sep The delimiter according which to split the string. None (the default value) means split according to any whitespace, and discard empty strings from the result.

maxsplit Maximum number of splits to do. -1 (the default value) means no limit.

Splits are done starting at the end of the string and working to the front.

rstrip()
Return a copy of the string with trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

split()
Return a list of the words in the string, using sep as the delimiter string.

sep The delimiter according which to split the string. None (the default value) means split according to any whitespace, and discard empty strings from the result.

maxsplit Maximum number of splits to do. -1 (the default value) means no limit.

splitlines()
Return a list of the lines in the string, breaking at line boundaries.

Line breaks are not included in the resulting list unless keepends is given and true.

startswith(*prefix*[, *start*[, *end*]]) → bool
Return True if S starts with the specified prefix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. prefix can also be a tuple of strings to try.

strip()
Return a copy of the string with leading and trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

swapcase()
Convert uppercase characters to lowercase and lowercase characters to uppercase.

title()
Return a version of the string where each word is titlecased.

More specifically, words start with uppercased characters and all remaining cased characters have lower case.

translate()
Replace each character in the string using the given translation table.

table Translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, strings, or None.

The table must implement lookup/indexing via __getitem__, for instance a dictionary or list. If this operation raises LookupError, the character is left untouched. Characters mapped to None are deleted.

transparency

upper()
Return a copy of the string converted to uppercase.

wireframe

zfill()
Pad a numeric string with zeros on the left, to fill a field of the given width.

The string is never truncated.

class pyEPR.ansys.**NamedCalcObject**(*name*, *setup*)
Bases: *pyEPR.ansys.CalcObject*

complexmag()

conj()

dot(*other*)

evaluate(*phase=0*, *lv=None*, *print_debug=False*)

getQty(*name*)

imag()

integrate_line(*name*)

integrate_line_tangent(*name*)
integrate line tangent to vector expression

name = of line to integrate over

integrate_surf(*name='AllObjects'*)

integrate_vol(*name='AllObjects'*)

line_tangent_coor(*name*, *coordinate*)
integrate line tangent to vector expression

name = of line to integrate over

mag()

maximum_vol(*name='AllObjects'*)

norm_2()

normal2surface(*name*)
return the part normal to surface. Complex Vector.

real()

release()

save_as(*name*)
if the object already exists, try clearing your named expressions first with fields.clear_named_expressions

scalar_x()

scalar_y()

scalar_z()

smooth()

tangent2surface(*name*)
return the part tangent to surface. Complex Vector.

times_eps()

times_mu()

write_stack()

class pyEPR.ansys.**OpenPolyline**(*name*, *modeler*, *points=None*)
Bases: *pyEPR.ansys.ModelEntity*

capitalize()
Return a capitalized version of the string.

More specifically, make the first character have upper case and the rest lower case.

casefold()
Return a version of the string suitable for caseless comparisons.

center()
Return a centered string of length width.

Padding is done using the specified fill character (default is a space).

coordinate_system

copy(*new_name*)

count(*sub*[, *start*[, *end*]]) → int
Return the number of non-overlapping occurrences of substring sub in string S[start:end]. Optional arguments start and end are interpreted as in slice notation.

encode()
Encode the string using the codec registered for encoding.

encoding The encoding in which to encode the string.

errors The error handling scheme to use for encoding errors. The default is 'strict' meaning that encoding errors raise a UnicodeEncodeError. Other possible values are 'ignore', 'replace' and 'xmlcharrefreplace' as well as any other name registered with codecs.register_error that can handle UnicodeEncodeErrors.

endswith(*suffix*[, *start*[, *end*]]) → bool
Return True if S ends with the specified suffix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. suffix can also be a tuple of strings to try.

expandtabs()
Return a copy where all tab characters are expanded using spaces.

If tabsize is not given, a tab size of 8 characters is assumed.

fillet(*radius*, *vertex_index*)

fillets(*radius*, *do_not_fillet=[]*)
do_not_fillet : Index list of vertices to not fillete

find(*sub*[, *start*[, *end*]]) → int
Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

format(**args*, ***kwargs*) → str
Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

format_map(*mapping*) → str
Return a formatted version of S, using substitutions from mapping. The substitutions are identified by braces ('{' and '}').

index(*sub*[, *start*[, *end*]]) → int
Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the substring is not found.

isalnum()
Return True if the string is an alpha-numeric string, False otherwise.

A string is alpha-numeric if all characters in the string are alpha-numeric and there is at least one character in the string.

isalpha()
Return True if the string is an alphabetic string, False otherwise.

A string is alphabetic if all characters in the string are alphabetic and there is at least one character in the string.

isascii()
Return True if all characters in the string are ASCII, False otherwise.

ASCII characters have code points in the range U+0000-U+007F. Empty string is ASCII too.

isdecimal()
Return True if the string is a decimal string, False otherwise.

A string is a decimal string if all characters in the string are decimal and there is at least one character in the string.

isdigit()
Return True if the string is a digit string, False otherwise.

A string is a digit string if all characters in the string are digits and there is at least one character in the string.

isidentifier()
Return True if the string is a valid Python identifier, False otherwise.

Use keyword.iskeyword() to test for reserved identifiers such as "def" and "class".

islower()
Return True if the string is a lowercase string, False otherwise.

A string is lowercase if all cased characters in the string are lowercase and there is at least one cased character in the string.

isnumeric()
Return True if the string is a numeric string, False otherwise.

A string is numeric if all characters in the string are numeric and there is at least one character in the string.

isprintable()
Return True if the string is printable, False otherwise.

A string is printable if all of its characters are considered printable in repr() or if it is empty.

isspace()
Return True if the string is a whitespace string, False otherwise.

A string is whitespace if all characters in the string are whitespace and there is at least one character in the string.

istitle()
Return True if the string is a title-cased string, False otherwise.

In a title-cased string, upper- and title-case characters may only follow uncased characters and lowercase characters only cased ones.

isupper()
Return True if the string is an uppercase string, False otherwise.

A string is uppercase if all cased characters in the string are uppercase and there is at least one cased character in the string.

join()
Concatenate any number of strings.

The string whose method is called is inserted in between each given string. The result is returned as a new string.

Example: '.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'

ljust()
Return a left-justified string of length width.

Padding is done using the specified fill character (default is a space).

lower()
Return a copy of the string converted to lowercase.

lstrip()
Return a copy of the string with leading whitespace removed.

If chars is given and not None, remove characters in chars instead.

static maketrans()
Return a translation table usable for str.translate().

If there is only one argument, it must be a dictionary mapping Unicode ordinals (integers) or characters to Unicode ordinals, strings or None. Character keys will be then converted to ordinals. If there are two arguments, they must be strings of equal length, and in the resulting dictionary, each character in x will be mapped to the character at the same position in y. If there is a third argument, it must be a string, whose characters will be mapped to None in the result.

material

model_command = 'CreatePolyline'

partition()
Partition the string into three parts using the given separator.

This will search for the separator in the string. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing the original string and two empty strings.

prop_holder = None

prop_server = None

prop_tab = 'Geometry3DCmdTab'

release()

rename(*new_name*)
Warning: The increment_name only works if the sheet has not been stracted or used as a tool elsewher. These names are not checked - They require modifying get_objects_in_group

replace()
Return a copy with all occurrences of substring old replaced by new.

count Maximum number of occurrences to replace. -1 (the default value) means replace all occurrences.

If the optional argument count is given, only the first count occurrences are replaced.

rfind(*sub*[, *start*[, *end*]]) → int
Return the highest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

rindex(*sub*[, *start*[, *end*]]) → int
Return the highest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the substring is not found.

rjust()
Return a right-justified string of length width.

Padding is done using the specified fill character (default is a space).

rpartition()
Partition the string into three parts using the given separator.

This will search for the separator in the string, starting at the end. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing two empty strings and the original string.

rsplit()
Return a list of the words in the string, using sep as the delimiter string.

sep The delimiter according which to split the string. None (the default value) means split according to any whitespace, and discard empty strings from the result.

maxsplit Maximum number of splits to do. -1 (the default value) means no limit.

Splits are done starting at the end of the string and working to the front.

rstrip()
Return a copy of the string with trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

show_direction

split()
Return a list of the words in the string, using sep as the delimiter string.

sep The delimiter according which to split the string. None (the default value) means split according to any whitespace, and discard empty strings from the result.

maxsplit Maximum number of splits to do. -1 (the default value) means no limit.

splitlines()
Return a list of the lines in the string, breaking at line boundaries.

Line breaks are not included in the resulting list unless keepends is given and true.

startswith(*prefix*[, *start*[, *end*]]) → bool
Return True if S starts with the specified prefix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. prefix can also be a tuple of strings to try.

strip()
Return a copy of the string with leading and trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

swapcase()
Convert uppercase characters to lowercase and lowercase characters to uppercase.

sweep_along_path(*to_sweep*)

title()
Return a version of the string where each word is titlecased.

More specifically, words start with uppercased characters and all remaining cased characters have lower case.

translate()
Replace each character in the string using the given translation table.

table Translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, strings, or None.

The table must implement lookup/indexing via __getitem__, for instance a dictionary or list. If this operation raises LookupError, the character is left untouched. Characters mapped to None are deleted.

transparency

upper()
Return a copy of the string converted to uppercase.

vertices()

wireframe

zfill()
Pad a numeric string with zeros on the left, to fill a field of the given width.

The string is never truncated.

class pyEPR.ansys.**Optimetrics**(*design*)
Bases: *pyEPR.ansys.COMWrapper*

Optimetrics script commands executed by the "Optimetrics" module.

Example use:

```
opti = Optimetrics(pinfo.design)
names = opti.get_setup_names()
print('Names of optimetrics: ', names)
opti.solve_setup(names[0])
```

Note that running optimetrics requires the license for Optimetrics by Ansys.

create_setup(*variable*, *swp_params*, *name='ParametricSetup1'*, *swp_type='linear_step'*, *setup_name=None*, *save_fields=True*, *copy_mesh=True*, *solve_with_copied_mesh_only=True*, *setup_type='parametric'*)

Inserts a new parametric setup of one variable. Either with sweep definition or from file.

Synchronized sweeps (more than one variable changing at once) can be implemented by giving a list of variables to `variable` and corresponding lists to `swp_params` and `swp_type`. The lengths of the sweep types should match (excluding single value).

Corresponds to ui access: Right-click the Optimetrics folder in the project tree, and then click Add> Parametric on the shortcut menu.

Ansys provides six sweep definitions types specified using the swp_type variable.

Sweep type definitions:

- **'single_value'** Specify a single value for the sweep definition.
- **'linear_step'** Specify a linear range of values with a constant step size.
- **'linear_count'** Specify a linear range of values and the number, or count of points within this range.
- **'decade_count'** Specify a logarithmic (base 10) series of values, and the number of values to calculate in each decade.
- **'octave_count'** Specify a logarithmic (base 2) series of values, and the number of values to calculate in each octave.
- **'exponential_count'** Specify an exponential (base e) series of values, and the number of values to calculate.

For swp_type='single_value' swp_params is the single value.

For swp_type='linear_step' swp_params is start, stop, step: swp_params = ("12.8nH", "13.6nH", "0.2nH")

All other types swp_params is start, stop, count: swp_params = ("12.8nH", "13.6nH", 4) The definition of count varies amongst the available types.

For Decade count and Octave count, the Count value specifies the number of points to calculate in every decade or octave. For Exponential count, the Count value is the total number of points. The total number of points includes the start and stop values.

For parametric from file, setup_type='parametric_file', pass in a file name and path to swp_params like "C:test.csv" or "C:test.txt" for example.

Example csv formatting: *,Lj_qubit 1,12.2nH 2,9.7nH 3,10.2nH

See Ansys documentation for additional formatting instructions.

get_setup_names()

Return list of Optimetrics setup names

release()

solve_setup(*setup_name: str*)
Solves the specified Optimetrics setup. Corresponds to: Right-click the setup in the project tree, and then click Analyze on the shortcut menu.

setup_name (str) : name of setup, should be in get_setup_names

Blocks execution until ready to use.

Note that this requires the license for Optimetrics by Ansys.

class pyEPR.ansys.**Polyline**(*name*, *modeler*, *points=None*)
Bases: *pyEPR.ansys.ModelEntity*

Assume closed polyline, which creates a polygon.

capitalize()
Return a capitalized version of the string.

More specifically, make the first character have upper case and the rest lower case.

casefold()
Return a version of the string suitable for caseless comparisons.

center()
Return a centered string of length width.

Padding is done using the specified fill character (default is a space).

coordinate_system

count(*sub*[, *start*[, *end*]]) → int
Return the number of non-overlapping occurrences of substring sub in string S[start:end]. Optional arguments start and end are interpreted as in slice notation.

encode()
Encode the string using the codec registered for encoding.

encoding The encoding in which to encode the string.

errors The error handling scheme to use for encoding errors. The default is 'strict' meaning that encoding errors raise a UnicodeEncodeError. Other possible values are 'ignore', 'replace' and 'xmlcharrefreplace' as well as any other name registered with codecs.register_error that can handle UnicodeEncodeErrors.

endswith(*suffix*[, *start*[, *end*]]) → bool
Return True if S ends with the specified suffix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. suffix can also be a tuple of strings to try.

expandtabs()
Return a copy where all tab characters are expanded using spaces.

If tabsize is not given, a tab size of 8 characters is assumed.

fillet(*radius*, *vertex_index*)

find(*sub*[, *start*[, *end*]]) → int
Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

format(**args*, ***kwargs*) → str
Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

format_map(*mapping*) → str
Return a formatted version of S, using substitutions from mapping. The substitutions are identified by braces ('{' and '}').

index(*sub*[, *start*[, *end*]]) → int
Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the substring is not found.

isalnum()
Return True if the string is an alpha-numeric string, False otherwise.

A string is alpha-numeric if all characters in the string are alpha-numeric and there is at least one character in the string.

isalpha()
Return True if the string is an alphabetic string, False otherwise.

A string is alphabetic if all characters in the string are alphabetic and there is at least one character in the string.

isascii()
Return True if all characters in the string are ASCII, False otherwise.

ASCII characters have code points in the range U+0000-U+007F. Empty string is ASCII too.

isdecimal()
Return True if the string is a decimal string, False otherwise.

A string is a decimal string if all characters in the string are decimal and there is at least one character in the string.

isdigit()
Return True if the string is a digit string, False otherwise.

A string is a digit string if all characters in the string are digits and there is at least one character in the string.

isidentifier()
Return True if the string is a valid Python identifier, False otherwise.

Use keyword.iskeyword() to test for reserved identifiers such as "def" and "class".

islower()
Return True if the string is a lowercase string, False otherwise.

A string is lowercase if all cased characters in the string are lowercase and there is at least one cased character in the string.

isnumeric()
Return True if the string is a numeric string, False otherwise.

A string is numeric if all characters in the string are numeric and there is at least one character in the string.

isprintable()
Return True if the string is printable, False otherwise.

A string is printable if all of its characters are considered printable in repr() or if it is empty.

isspace()
Return True if the string is a whitespace string, False otherwise.

A string is whitespace if all characters in the string are whitespace and there is at least one character in the string.

istitle()
Return True if the string is a title-cased string, False otherwise.

In a title-cased string, upper- and title-case characters may only follow uncased characters and lowercase characters only cased ones.

isupper()
Return True if the string is an uppercase string, False otherwise.

A string is uppercase if all cased characters in the string are uppercase and there is at least one cased character in the string.

join()
Concatenate any number of strings.

The string whose method is called is inserted in between each given string. The result is returned as a new string.

Example: '.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'

ljust()
Return a left-justified string of length width.

Padding is done using the specified fill character (default is a space).

lower()
Return a copy of the string converted to lowercase.

lstrip()
Return a copy of the string with leading whitespace removed.

If chars is given and not None, remove characters in chars instead.

make_center_line(*axis*)

make_rlc_boundary(*axis*, *r=0*, *l=0*, *c=0*, *name='LumpRLC'*)

static maketrans()
Return a translation table usable for str.translate().

If there is only one argument, it must be a dictionary mapping Unicode ordinals (integers) or characters to Unicode ordinals, strings or None. Character keys will be then converted to ordinals. If there are two arguments, they must be strings of equal length, and in the resulting dictionary, each character in x will be mapped to the character at the same position in y. If there is a third argument, it must be a string, whose characters will be mapped to None in the result.

material

model_command = 'CreatePolyline'

partition()
Partition the string into three parts using the given separator.

This will search for the separator in the string. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing the original string and two empty strings.

prop_holder = None

prop_server = None

prop_tab = 'Geometry3DCmdTab'

release()

rename(*new_name*)
Warning: The increment_name only works if the sheet has not been stracted or used as a tool elsewhere. These names are not checked; they require modifying get_objects_in_group.

replace()
Return a copy with all occurrences of substring old replaced by new.

count Maximum number of occurrences to replace. -1 (the default value) means replace all occurrences.

If the optional argument count is given, only the first count occurrences are replaced.

rfind(*sub*[, *start*[, *end*]]) → int
Return the highest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

rindex(*sub*[, *start*[, *end*]]) → int
Return the highest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the substring is not found.

rjust()
Return a right-justified string of length width.

Padding is done using the specified fill character (default is a space).

rpartition()
Partition the string into three parts using the given separator.

This will search for the separator in the string, starting at the end. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing two empty strings and the original string.

rsplit()
Return a list of the words in the string, using sep as the delimiter string.

sep The delimiter according which to split the string. None (the default value) means split according to any whitespace, and discard empty strings from the result.

maxsplit Maximum number of splits to do. -1 (the default value) means no limit.

Splits are done starting at the end of the string and working to the front.

rstrip()
Return a copy of the string with trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

split()
Return a list of the words in the string, using sep as the delimiter string.

sep The delimiter according which to split the string. None (the default value) means split according to any whitespace, and discard empty strings from the result.

maxsplit Maximum number of splits to do. -1 (the default value) means no limit.

splitlines()
Return a list of the lines in the string, breaking at line boundaries.

Line breaks are not included in the resulting list unless keepends is given and true.

startswith(*prefix*[, *start*[, *end*]]) → bool
Return True if S starts with the specified prefix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. prefix can also be a tuple of strings to try.

strip()
Return a copy of the string with leading and trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

swapcase()
Convert uppercase characters to lowercase and lowercase characters to uppercase.

title()
Return a version of the string where each word is titlecased.

More specifically, words start with uppercased characters and all remaining cased characters have lower case.

translate()
Replace each character in the string using the given translation table.

table Translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, strings, or None.

The table must implement lookup/indexing via __getitem__, for instance a dictionary or list. If this operation raises LookupError, the character is left untouched. Characters mapped to None are deleted.

transparency

unite(*list_other*)

upper()
Return a copy of the string converted to uppercase.

vertices()

wireframe

zfill()
Pad a numeric string with zeros on the left, to fill a field of the given width.

The string is never truncated.

class pyEPR.ansys.**Rect**(*name*, *modeler*, *corner*, *size*)
Bases: *pyEPR.ansys.ModelEntity*

capitalize()
Return a capitalized version of the string.

More specifically, make the first character have upper case and the rest lower case.

casefold()
Return a version of the string suitable for caseless comparisons.

center()
Return a centered string of length width.

Padding is done using the specified fill character (default is a space).

coordinate_system

count(*sub*[, *start*[, *end*]]) → int
Return the number of non-overlapping occurrences of substring sub in string S[start:end]. Optional arguments start and end are interpreted as in slice notation.

encode()
Encode the string using the codec registered for encoding.

encoding The encoding in which to encode the string.

errors The error handling scheme to use for encoding errors. The default is 'strict' meaning that encoding errors raise a UnicodeEncodeError. Other possible values are 'ignore', 'replace' and 'xmlcharrefreplace' as well as any other name registered with codecs.register_error that can handle UnicodeEncodeErrors.

endswith(*suffix*[, *start*[, *end*]]) → bool
Return True if S ends with the specified suffix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. suffix can also be a tuple of strings to try.

expandtabs()
Return a copy where all tab characters are expanded using spaces.

If tabsize is not given, a tab size of 8 characters is assumed.

find(*sub*[, *start*[, *end*]]) → int
Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

format(**args*, ***kwargs*) → str
Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

format_map(*mapping*) → str
Return a formatted version of S, using substitutions from mapping. The substitutions are identified by braces ('{' and '}').

index(*sub*[, *start*[, *end*]]) → int
Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the substring is not found.

isalnum()
Return True if the string is an alpha-numeric string, False otherwise.

A string is alpha-numeric if all characters in the string are alpha-numeric and there is at least one character in the string.

isalpha()
Return True if the string is an alphabetic string, False otherwise.

A string is alphabetic if all characters in the string are alphabetic and there is at least one character in the string.

isascii()
Return True if all characters in the string are ASCII, False otherwise.

ASCII characters have code points in the range U+0000-U+007F. Empty string is ASCII too.

isdecimal()
Return True if the string is a decimal string, False otherwise.

A string is a decimal string if all characters in the string are decimal and there is at least one character in the string.

isdigit()
Return True if the string is a digit string, False otherwise.

A string is a digit string if all characters in the string are digits and there is at least one character in the string.

isidentifier()
Return True if the string is a valid Python identifier, False otherwise.

Use keyword.iskeyword() to test for reserved identifiers such as “def” and “class”.

islower()
Return True if the string is a lowercase string, False otherwise.

A string is lowercase if all cased characters in the string are lowercase and there is at least one cased character in the string.

isnumeric()
Return True if the string is a numeric string, False otherwise.

A string is numeric if all characters in the string are numeric and there is at least one character in the string.

isprintable()
Return True if the string is printable, False otherwise.

A string is printable if all of its characters are considered printable in repr() or if it is empty.

isspace()
Return True if the string is a whitespace string, False otherwise.

A string is whitespace if all characters in the string are whitespace and there is at least one character in the string.

istitle()
Return True if the string is a title-cased string, False otherwise.

In a title-cased string, upper- and title-case characters may only follow uncased characters and lowercase characters only cased ones.

isupper()
Return True if the string is an uppercase string, False otherwise.

A string is uppercase if all cased characters in the string are uppercase and there is at least one cased character in the string.

join()
Concatenate any number of strings.

The string whose method is called is inserted in between each given string. The result is returned as a new string.

Example: ‘.’.join([‘ab’, ‘pq’, ‘rs’]) -> ‘ab.pq.rs’

ljust()
Return a left-justified string of length width.

Padding is done using the specified fill character (default is a space).

lower()
Return a copy of the string converted to lowercase.

lstrip()
Return a copy of the string with leading whitespace removed.

If chars is given and not None, remove characters in chars instead.

make_center_line(*axis*)
Returns *start* and *end* list of 3 coordinates

make_lumped_port(*axis*, *z0='50ohm'*, *name='LumpPort'*)

make_rlc_boundary(*axis*, *r=0*, *l=0*, *c=0*, *name='LumpRLC'*)

static maketrans()
Return a translation table usable for str.translate().

If there is only one argument, it must be a dictionary mapping Unicode ordinals (integers) or characters to Unicode ordinals, strings or None. Character keys will be then converted to ordinals. If there are two arguments, they must be strings of equal length, and in the resulting dictionary, each character in x will be mapped to the character at the same position in y. If there is a third argument, it must be a string, whose characters will be mapped to None in the result.

material

model_command = 'CreateRectangle'

partition()
Partition the string into three parts using the given separator.

This will search for the separator in the string. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing the original string and two empty strings.

prop_holder = None

prop_server = None

prop_tab = 'Geometry3DCmdTab'

release()

replace()
Return a copy with all occurrences of substring old replaced by new.

count Maximum number of occurrences to replace. -1 (the default value) means replace all occurrences.

If the optional argument count is given, only the first count occurrences are replaced.

rfind(*sub*[, *start*[, *end*]]) → int
Return the highest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

rindex(*sub*[, *start*[, *end*]]) → int
Return the highest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the substring is not found.

rjust()
Return a right-justified string of length width.

Padding is done using the specified fill character (default is a space).

rpartition()
Partition the string into three parts using the given separator.

This will search for the separator in the string, starting at the end. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing two empty strings and the original string.

rsplit()
Return a list of the words in the string, using sep as the delimiter string.

sep The delimiter according which to split the string. None (the default value) means split according to any whitespace, and discard empty strings from the result.

maxsplit Maximum number of splits to do. -1 (the default value) means no limit.

Splits are done starting at the end of the string and working to the front.

rstrip()
Return a copy of the string with trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

split()
Return a list of the words in the string, using sep as the delimiter string.

sep The delimiter according which to split the string. None (the default value) means split according to any whitespace, and discard empty strings from the result.

maxsplit Maximum number of splits to do. -1 (the default value) means no limit.

splitlines()
Return a list of the lines in the string, breaking at line boundaries.

Line breaks are not included in the resulting list unless keepends is given and true.

startswith(*prefix*[, *start*[, *end*]]) → bool
Return True if S starts with the specified prefix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. prefix can also be a tuple of strings to try.

strip()
Return a copy of the string with leading and trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

swapcase()
Convert uppercase characters to lowercase and lowercase characters to uppercase.

title()
Return a version of the string where each word is titlecased.

More specifically, words start with uppercased characters and all remaining cased characters have lower case.

translate()
Replace each character in the string using the given translation table.

table Translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, strings, or None.

The table must implement lookup/indexing via __getitem__, for instance a dictionary or list. If this operation raises LookupError, the character is left untouched. Characters mapped to None are deleted.

transparency

upper()
Return a copy of the string converted to uppercase.

wireframe

zfill()
Pad a numeric string with zeros on the left, to fill a field of the given width.

The string is never truncated.

class pyEPR.ansys.**VariableString**
Bases: str

capitalize()
Return a capitalized version of the string.

More specifically, make the first character have upper case and the rest lower case.

casefold()
Return a version of the string suitable for caseless comparisons.

center()
Return a centered string of length width.

Padding is done using the specified fill character (default is a space).

count(*sub*[, *start*[, *end*]]) → int
Return the number of non-overlapping occurrences of substring sub in string S[start:end]. Optional arguments start and end are interpreted as in slice notation.

encode()
Encode the string using the codec registered for encoding.

encoding The encoding in which to encode the string.

errors The error handling scheme to use for encoding errors. The default is 'strict' meaning that encoding errors raise a UnicodeEncodeError. Other possible values are 'ignore', 'replace' and 'xmlcharrefreplace' as well as any other name registered with codecs.register_error that can handle UnicodeEncodeErrors.

endswith(*suffix*[, *start*[, *end*]]) → bool
Return True if S ends with the specified suffix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. suffix can also be a tuple of strings to try.

expandtabs()
Return a copy where all tab characters are expanded using spaces.

If tabsize is not given, a tab size of 8 characters is assumed.

find(*sub*[, *start*[, *end*]]) → int
Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

format(**args*, ***kwargs*) → str
Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

format_map(*mapping*) → str
Return a formatted version of S, using substitutions from mapping. The substitutions are identified by braces ('{' and '}').

index(*sub*[, *start*[, *end*]]) → int
Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the substring is not found.

isalnum()
Return True if the string is an alpha-numeric string, False otherwise.

A string is alpha-numeric if all characters in the string are alpha-numeric and there is at least one character in the string.

isalpha()
Return True if the string is an alphabetic string, False otherwise.

A string is alphabetic if all characters in the string are alphabetic and there is at least one character in the string.

isascii()
Return True if all characters in the string are ASCII, False otherwise.

ASCII characters have code points in the range U+0000-U+007F. Empty string is ASCII too.

isdecimal()
Return True if the string is a decimal string, False otherwise.

A string is a decimal string if all characters in the string are decimal and there is at least one character in the string.

isdigit()
Return True if the string is a digit string, False otherwise.

A string is a digit string if all characters in the string are digits and there is at least one character in the string.

isidentifier()
Return True if the string is a valid Python identifier, False otherwise.

Use keyword.iskeyword() to test for reserved identifiers such as “def” and “class”.

islower()
Return True if the string is a lowercase string, False otherwise.

A string is lowercase if all cased characters in the string are lowercase and there is at least one cased character in the string.

isnumeric()
Return True if the string is a numeric string, False otherwise.

A string is numeric if all characters in the string are numeric and there is at least one character in the string.

isprintable()
Return True if the string is printable, False otherwise.

A string is printable if all of its characters are considered printable in repr() or if it is empty.

isspace()
Return True if the string is a whitespace string, False otherwise.

A string is whitespace if all characters in the string are whitespace and there is at least one character in the string.

istitle()
Return True if the string is a title-cased string, False otherwise.

In a title-cased string, upper- and title-case characters may only follow uncased characters and lowercase characters only cased ones.

isupper()
Return True if the string is an uppercase string, False otherwise.

A string is uppercase if all cased characters in the string are uppercase and there is at least one cased character in the string.

join()
Concatenate any number of strings.

The string whose method is called is inserted in between each given string. The result is returned as a new string.

Example: '.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'

ljust()
Return a left-justified string of length width.

Padding is done using the specified fill character (default is a space).

lower()
Return a copy of the string converted to lowercase.

lstrip()
Return a copy of the string with leading whitespace removed.

If chars is given and not None, remove characters in chars instead.

static maketrans()
Return a translation table usable for str.translate().

If there is only one argument, it must be a dictionary mapping Unicode ordinals (integers) or characters to Unicode ordinals, strings or None. Character keys will be then converted to ordinals. If there are two arguments, they must be strings of equal length, and in the resulting dictionary, each character in x will be mapped to the character at the same position in y. If there is a third argument, it must be a string, whose characters will be mapped to None in the result.

partition()
Partition the string into three parts using the given separator.

This will search for the separator in the string. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing the original string and two empty strings.

replace()
Return a copy with all occurrences of substring old replaced by new.

count Maximum number of occurrences to replace. -1 (the default value) means replace all occurrences.

If the optional argument count is given, only the first count occurrences are replaced.

rfind(*sub*[, *start*[, *end*]]) → int
Return the highest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Return -1 on failure.

rindex(*sub*[, *start*[, *end*]]) → int
Return the highest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation.

Raises ValueError when the substring is not found.

rjust()
Return a right-justified string of length width.

Padding is done using the specified fill character (default is a space).

rpartition()
Partition the string into three parts using the given separator.

This will search for the separator in the string, starting at the end. If the separator is found, returns a 3-tuple containing the part before the separator, the separator itself, and the part after it.

If the separator is not found, returns a 3-tuple containing two empty strings and the original string.

rsplit()
Return a list of the words in the string, using sep as the delimiter string.

sep The delimiter according which to split the string. None (the default value) means split according to any whitespace, and discard empty strings from the result.

maxsplit Maximum number of splits to do. -1 (the default value) means no limit.

Splits are done starting at the end of the string and working to the front.

rstrip()
Return a copy of the string with trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

split()
Return a list of the words in the string, using sep as the delimiter string.

sep The delimiter according which to split the string. None (the default value) means split according to any whitespace, and discard empty strings from the result.

maxsplit Maximum number of splits to do. -1 (the default value) means no limit.

splitlines()
Return a list of the lines in the string, breaking at line boundaries.

Line breaks are not included in the resulting list unless keepends is given and true.

startswith(*prefix*[, *start*[, *end*]]) → bool
Return True if S starts with the specified prefix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. prefix can also be a tuple of strings to try.

strip()
Return a copy of the string with leading and trailing whitespace removed.

If chars is given and not None, remove characters in chars instead.

swapcase()
Convert uppercase characters to lowercase and lowercase characters to uppercase.

title()
Return a version of the string where each word is titlecased.

More specifically, words start with uppercased characters and all remaining cased characters have lower case.

translate()
Replace each character in the string using the given translation table.

table Translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, strings, or None.

The table must implement lookup/indexing via __getitem__, for instance a dictionary or list. If this operation raises LookupError, the character is left untouched. Characters mapped to None are deleted.

upper()
Return a copy of the string converted to uppercase.

zfill()
Pad a numeric string with zeros on the left, to fill a field of the given width.

The string is never truncated.

pyEPR.ansys.**extract_value_dim**(*expr*)
type expr: str

pyEPR.ansys.**extract_value_unit**(*expr*, *units*)

Returns float

pyEPR.ansys.**fix_units**(*x*, *unit_assumed=None*)
Convert all numbers to string and append the assumed units if needed. For an iterable, returns a list

pyEPR.ansys.**get_active_design**()

pyEPR.ansys.**get_active_project**()
If you see the error: "The requested operation requires elevation." then you need to run your python as an admin.

pyEPR.ansys.**get_report_arrays**(*name: str*)

pyEPR.ansys.**increment_name**(*base*, *existing*)

pyEPR.ansys.**load_ansys_project**(*proj_name: str*, *project_path: str = None*, *extension: str = '.aedt'*)
Utility function to load an Ansys project.

Parameters

- **proj_name** – None –> get active. (make sure 2 run as admin)
- **extension** – *aedt* is for 2016 version and newer

pyEPR.ansys.**make_float_prop**(*name*, *prop_tab=None*, *prop_server=None*)

pyEPR.ansys.**make_int_prop**(*name*, *prop_tab=None*, *prop_server=None*)

pyEPR.ansys.**make_prop**(*name*, *prop_tab=None*, *prop_server=None*, *prop_args=None*)

pyEPR.ansys.**make_str_prop**(*name*, *prop_tab=None*, *prop_server=None*)

pyEPR.ansys.**parse_entry**(*entry*, *convert_to_unit='meter'*)
Should take a list of tuple of list... of int, float or str... For iterables, returns lists

pyEPR.ansys.**parse_units**(*x*)
Convert number, string, and lists/arrays/tuples to numbers scaled in HFSS units.

Converts to LENGTH_UNIT = meters [HFSS UNITS] Assumes input units LENGTH_UNIT_ASSUMED = mm [USER UNITS]

[USER UNITS] —-> [HFSS UNITS]

pyEPR.ansys.**parse_units_user**(*x*)
Convert from user assumed units to user assumed units [USER UNITS] —-> [USER UNITS]

pyEPR.ansys.**release**()
Release COM connection to Ansys.

pyEPR.ansys.**set_property**(*prop_holder*, *prop_tab*, *prop_server*, *name*, *value*, *prop_args=None*)
More general non obj oriented, functional version prop_args = [] by default

pyEPR.ansys.**simplify_arith_expr**(*expr*)

pyEPR.ansys.**unparse_units**(*x*)
Undo effect of parse_unit.

Converts to LENGTH_UNIT_ASSUMED = mm [USER UNITS] Assumes input units LENGTH_UNIT = meters [HFSS UNITS]

[HFSS UNITS] —-> [USER UNITS]

pyEPR.ansys.**var**(*x*)

pyEPR.core module

Main interface module to use pyEPR.

Contains code to connect to Ansys and to analyze HFSS files using the EPR method.

This module handles the microwave part of the analysis and connection to

Further contains code to be able to do autogenerated reports,

pyEPR.core_distributed_analysis module

Main distributed analysis module to use pyEPR.

Contains code to connect to Ansys and to analyze HFSS files using the EPR method.

This module handles the microwave part of the analysis and connection to

Further contains code to be able to do autogenerated reports,

class pyEPR.core_distributed_analysis.**DistributedAnalysis**(**args*, ***kwargs*)
Bases: object

DISTRIBUTED ANALYSIS of layout and microwave results.

Main computation class & interface with HFSS.

This class defines a DistributedAnalysis object which calculates and saves Hamiltonian parameters from an HFSS simulation.

Further, it allows one to calculate dissipation, etc.

calc_Q_external(*variation*, *freq_GHz*, *U_E=None*)
Calculate the coupling Q of mode m with each port p Expected that you have specified the mode before calling this

Parameters

- **variation** (*str*) – A string identifier of the variation,
- **as '0', '1', ..** (*such*) –

calc_avg_current_J_surf_mag(*variation: str*, *junc_rect: str*, *junc_line*)

Peak current I_max for mode J in junction J The avg. is over the surface of the junction. I.e., spatial.

Parameters

- **variation** (*str*) – A string identifier of the variation, such as '0', '1', ...

- **junc_rect** (*str*) – name of rectangle to integrate over
- **junc_line** (*str*) – name of junction line to integrate over

Returns Value of peak current

calc_current (*fields*, *line: str*)
Function to calculate Current based on line. Not in use.

Parameters **line** (*str*) – integration line between plates - name

calc_current_using_line_voltage (*variation: str*, *junc_line_name: str*, *junc_L_Henries: float*, *Cj_Farads: float = None*)
Peak current I_max for prespecified mode calculating line voltage across junction.

Make sure that you have set the correct variation in HFSS before running this

Parameters

- **variation** – variation number
- **junc_line_name** – name of the HFSS line spanning the junction
- **junc_L_Henries** – junction inductance in henries
- **Cj_Farads** – junction cap in Farads
- **TODO** – Smooth?

calc_energy_electric (*variation: str = None*, *obj: str = 'AllObjects'*, *volume: str = 'Deprecated'*, *smooth: bool = False*, *obj_dims: int = 3*)
Calculates two times the peak electric energy, or 4 times the RMS, $4 * \mathcal{E}_{\text{elec}}$ (since we do not divide by 2 and use the peak phasors).

$$\mathcal{E}_{\text{elec}} = \frac{1}{4}\text{Re}\int_V \mathrm{d}v \vec{E}^*_{\text{max}} \overleftrightarrow{\epsilon} \vec{E}_{\text{max}}$$

Parameters

- **variation** (*str*) – A string identifier of the variation, such as '0', '1', ...
- **obj** (*string | 'AllObjects'*) – Name of the object to integrate over
- **smooth** (*bool | False*) – Smooth the electric field or not when performing calculation
- **obj_dims** (*int | 3*) – 1 - line, 2 - surface, 3 - volume. Default volume

Example

Example use to calculate the energy participation ratio (EPR) of a substrate

```
_total  = epr_hfss.calc_energy_electric(obj='AllObjects')
_substr = epr_hfss.calc_energy_electric(obj='Box1')
print(f'Energy in substrate = {100*_substr/_total:.1f}%')
```

calc_energy_magnetic (*variation: str = None*, *obj: str = 'AllObjects'*, *volume: str = 'Deprecated'*, *smooth: bool = False*, *obj_dims: int = 3*)
See calc_energy_electric.

Parameters

- **variation** (*str*) – A string identifier of the variation, such as '0', '1', ...

- **volume** (*string | 'AllObjects'*) – Name of the volume to integrate over
- **smooth** (*bool | False*) – Smooth the electric field or not when performing calculation
- **obj_dims** (*int | 3*) – 1 - line, 2 - surface, 3 - volume. Default volume

calc_line_current (*variation*, *junc_line_name*)

calc_p_electric_volume (*name_dielectric3D*, *relative_to='AllObjects'*, *variation=None*, *E_total=None*)

Calculate the dielectric energy-participation ratio of a 3D object (one that has volume) relative to the dielectric energy of a list of objects.

This is as a function relative to another object or all objects.

When all objects are specified, this does not include any energy that might be stored in any lumped elements or lumped capacitors.

Returns _object/_total, (_object, _total)

calc_p_junction (*variation*, *U_H*, *U_E*, *Ljs*, *Cjs*)

For a single specific mode. Expected that you have specified the mode before calling this, *set_mode()*.

Expected to precalc U_H and U_E for mode, will return pandas pd.Series object:

- junc_rect = ['junc_rect1', 'junc_rect2'] name of junc rectangles to integrate H over
- junc_len = [0.0001] specify in SI units; i.e., meters
- LJs = [8e-09, 8e-09] SI units
- calc_sign = ['junc_line1', 'junc_line2']

WARNING: Cjs is experimental.

This function assumes there are no lumped capacitors in model.

Parameters

- **variation** (*str*) – A string identifier of the variation,
- **as '0', '1', ..** (*such*) –

Note: U_E and U_H are the total peak energy. (NOT twice as in U_ and U_H other places)

Warning: Potential errors: If you dont have a line or rect by the right name you will prob get an error of the type: com_error: (-2147352567, 'Exception occurred.', (0, None, None, None, 0, -2147024365), None)

calc_p_junction_single (*mode*, *variation*, *U_E=None*, *U_H=None*)

This function is used in the case of a single junction only. For multiple junctions, see *calc_p_junction()*.

Assumes no lumped capacitive elements.

design

Ansys design class handle

do_EPR_analysis (*variations: list = None*, *modes=None*, *append_analysis=True*)

Main analysis routine

Parameters **variation** (*str*) – A string identifier of the variation, such as '0', '1', ...

variations [list | None] Example list of variations is ['0', '1'] A variation is a combination of project/design variables in an optimetric sweep

modes [list | None] Modes to analyze for example modes = [0, 2, 3]

append_analysis (bool) : When we run the Ansys analysis, should we redo any variations that we have already done?

Assumptions: Low dissipation (high-Q). It is easier to assume no lumped capacitors to simply calculations, but we have recently added Cj_variable as a new feature that is begin tested to handle capacitors.

See the paper.

Load results with epr.QuantumAnalysis class

```
eprd = epr.DistributedAnalysis(pinfo)
eprd.do_EPR_analysis(append_analysis=False)
```

get_Qdielectric(*dielectric*, *mode*, *variation*, *U_E=None*)

get_Qseam(*seam*, *mode*, *variation*, *U_H=None*)
Calculate the contribution to Q of a seam, by integrating the current in the seam with finite conductance: set in the config file ref: http://arxiv.org/pdf/1509.01119.pdf

get_Qseam_sweep(*seam*, *mode*, *variation*, *variable*, *values*, *unit*, *U_H=None*, *pltresult=True*)
Q due to seam loss.

values = ['5mm','6mm','7mm'] ref: http://arxiv.org/pdf/1509.01119.pdf

get_Qsurface(*mode*, *variation*, *name*, *U_E=None*, *material_properties=None*)
Calculate the contribution to Q of a dielectric layer of dirt on a given surface. Set the dirt thickness and loss tangent in the config file ref: http://arxiv.org/pdf/1509.01854.pdf

get_Qsurface_all(*mode*, *variation*, *U_E=None*)
Calculate the contribution to Q of a dielectric layer of dirt on all surfaces. Set the dirt thickness and loss tangent in the config file ref: http://arxiv.org/pdf/1509.01854.pdf

get_ansys_frequencies_all(*vs='variation'*)
Return all ansys frequencies and quality factors vs a variation

Returns a multi-index pandas DataFrame

get_ansys_variables()
Get ansys variables for all variations

Returns Return a dataframe of variables as index and columns as the variations

get_ansys_variations()
Will update ansys information and result the list of variations.

Returns

```
("Cj='2fF' Lj='12nH'",
"Cj='2fF' Lj='12.5nH'",
"Cj='2fF' Lj='13nH'",
"Cj='2fF' Lj='13.5nH'",
"Cj='2fF' Lj='14nH'")
```

Return type For example

get_convergence(*variation='0'*)

Parameters

- **variation** (*str*) – A string identifier of the variation,
- **as '0', '1', ..** (*such*) –

Returns

A pandas DataFrame object

```
    Solved Elements     Max Delta Freq. % Pass Number
1           128955              NaN
2               167607          11.745000
3               192746          3.208600
4               199244          1.524000
```

get_convergence_vs_pass(*variation='0'*)

Makes a plot in HFSS that return a pandas dataframe

Parameters

- **variation** (*str*) – A string identifier of the variation,
- **as '0', '1', ..** (*such*) –

Returns

Returns a convergence vs pass number of the eignemode freqs.

```
    re(Mode(1)) [g]     re(Mode(2)) [g] re(Mode(3)) [g]
Pass []
1       4.643101        4.944204        5.586289
2       5.114490        5.505828        6.242423
3       5.278594        5.604426        6.296777
```

get_freqs_bare(*variation: str*)

Warning: Outdated. Do not use. To be deprecated

Parameters **variation** (*str*) – A string identifier of the variation, such as '0', '1', ...

Returns [type] – [description]

get_freqs_bare_pd(*variation: str*, *frame=True*)

Return the freq and Qs of the solved modes for a variation. I.e., the Ansys solved frequencies.

Parameters

- **variation** (*str*) – A string identifier of the variation, such as '0', '1', ...
- **{bool} -- if True returns dataframe, else tuple of series.** (*frame*) –

Returns

If frame = True, then a multi-index Dataframe that looks something like this

```
                Freq. (GHz)  Quality Factor
variation mode
0         0        5.436892            1020
          1        7.030932           50200
1         0        5.490328            2010
          1        7.032116          104500
```

If frame = False, then a tuple of two Series, such as (Fs, Qs) – Tuple of pandas.Series objects; the row index is the mode number

get_junc_len_dir (*variation: str*, *junc_line*)

Return the length and direction of a junction defined by a line

Parameters

- **variation** (*str*) – simulation variation
- **junc_line** (*str*) – polyline object

Returns

junction length uj (list of 3 floats): x,y,z coordinates of the unit vector

tangent to the junction line

Return type jl (float)

get_junctions_L_and_C (*variation: str*)

Returns a pandas Series with the index being the junction name as specified in the project_info.

The values in the series are numeric and in SI base units, i.e., not nH but Henries, and not fF but Farads.

Parameters

- **variation** (*str*) – label such as '0' or 'all', in which case return
- **table for all variations** (*pandas*) –

get_mesh_statistics (*variation='0'*)

Parameters

- **variation** (*str*) – A string identifier of the variation,
- **as '0', '1', ..** (*such*) –

Returns: A pandas dataframe, such as

```
1  Name         Num Tets    Min edge    length          Max edge length
   ↪RMS edge length Min tet vol     Max tet vol     Mean tet vol    Std Devn
   ↪(vol)
2  0   Region      909451          0.000243    0.860488        0.037048
   ↪  6.006260e-13        0.037352        0.000029        6.268190e-04
3  1   substrate       1490356     0.000270    0.893770        0.023639
   ↪  1.160090e-12        0.031253        0.000007        2.309920e-04
```

get_nominal_variation_index ()

Returns A string identifies, such as '0' or '1', that labels the nominal variation index number.

This may not be in the solved list!s

get_previously_analyzed ()

Return previously analyzed data.

Does not yet handle data that was previously saved in a filename.

get_variable_vs_variations (*variable: str*, *convert: bool = True*)
Get ansys variables

Return HFSS variable from `self.get_ansys_variables()` as a pandas series vs variations.

Parameters **convert** (*bool*) – Convert to a numeric quantity if possible using the ureg

get_variables (*variation=None*)
Get ansys variables.

Parameters **variation** (*str*) – A string identifier of the variation, such as '0', '1', ...

get_variation_string (*variation=None*)
Solved variation string identifier.

Parameters **variation** (*str*) – A string identifier of the variation, such as '0', '1', ...

Returns

Return the list variation string of parameters in ansys used to identify the variation.

```
"$test='0.25mm' Cj='2fF' Lj='12.5nH'"
```

get_variations ()
An array of strings corresponding to **solved** variations corresponding to the selected Setup.

Returns

Returns a list of strings that give the variation labels for HFSS.

```
OrderedDict([
    ('0', "Cj='2fF' Lj='12nH'"),
    ('1', "Cj='2fF' Lj='12.5nH'"),
    ('2', "Cj='2fF' Lj='13nH'"),
    ('3', "Cj='2fF' Lj='13.5nH'"),
    ('4', "Cj='2fF' Lj='14nH'")])
```

has_fields (*variation: str = None*)
Determine if fields exist for a particular solution. Just calls *self.solutions.has_fields(variation_string)*

Parameters **variation** (*str*) – String of variation label, such as '0' or '1'. If None, gets the nominal variation

hfss_report_f_convergence (*variation='0'*, *save_csv=True*)
Create a report inside HFSS to plot the converge of freq and style it.

Saves report to csv file.

Returns a convergence vs pass number of the eignemode freqs. Returns a pandas dataframe:

```
    re(Mode(1)) [g]  re(Mode(2)) [g]  re(Mode(3)) [g]
Pass []
1     4.643101         4.944204         5.586289
2     5.114490         5.505828         6.242423
3     5.278594         5.604426         6.296777
```

hfss_report_full_convergence (*fig=None*, *_display=True*)
Plot a full report of teh convergences of an eigenmode analysis for a a given variation. Makes a plot inside hfss too.

Keyword Arguments

- **{matplotlib figure} -- Optional figure (default** (*fig*) – {None})

- **{bool} -- Force display or not. (default** (*_display*) – {True})

Returns [type] – [description]

load (*filepath=None*)
Utility function to load results file

Keyword Arguments **{[type]} -- [description] (default** (*filepath*) – {None})

n_variations
Number of **solved** variations, corresponding to the selected Setup.

options
Project info options

project
Ansys project class handle

quick_plot_frequencies (*swp_variable='variations'*, *ax=None*)
Quick plot of frequencies from HFSS

static results_variations_on_inside (*results: dict*)
Switches the order on result of variations. Reverse dict.

save (*project_info: dict = None*)
Save results to self.data_filename

Keyword Arguments **{dict} -- [description] (default** (*project_info*) – {None})

set_mode (*mode_num*, *phase=0*)
Set source excitations should be used for fields post processing. Counting modes from 0 onward

set_variation (*variation: str*)
Set the ansys design to a solved variation. This will change all local variables!

Warning: not tested with global variables.

setup
Ansys setup class handle. Could be None.

setup_data ()
Set up folder paths for saving data to.

Sets the save filename with the current time.

Saves to Path(config.root_dir) / self.project.name / self.design.name

update_ansys_info ()
‘ Updates all information about the Ansys solved variations and variables.

```
n_modes, _list_variations, nominal_variation, n_variations
```

variations = None
List of variation indices, which are strings of ints, such as [‘0’, ‘1’]

pyEPR.core_quantum_analysis module

Main interface module to use pyEPR.

Contains code that works on the analysis after hfss, ansys, etc. These can now be closed.

class pyEPR.core_quantum_analysis.**HamiltonianResultsContainer**(*dict_file=None*, *data_dir=None*)

Bases: collections.OrderedDict

The user should only use the QuantumAnalysis class interface.

This class is largely for internal use.

It is a dictionary based class to contain the results stored.

clear() → None. Remove all items from od.

copy() → a shallow copy of od

file_name_extra = ' HamiltonianResultsContainer.npz'

fromkeys()
Create a new ordered dictionary with keys from iterable and values set to value.

get()
Return the value for key if key is in the dictionary, else default.

get_chi_ND(*variations: list = None*, *vs='variation'*)

get_chi_O1(*variations: list = None*, *vs='variation'*)

get_frequencies_HFSS(*variations: list = None*, *vs='variation'*)
See help for *vs_variations*

get_frequencies_ND(*variations: list = None*, *vs='variation'*)
See help for *vs_variations*

get_frequencies_O1(*variations: list = None*, *vs='variation'*)
See help for *vs_variations*

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

load(*filename=None*)
Uses numpy npz file.

move_to_end()
Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

popitem()
Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

save(*filename: str = None*)
Uses numpy npz file.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

update(*[E]*, ***F*) → None. Update D from dict/iterable E and F.
If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

vs_variations(*quantity: str*, *variations: list = None*, *vs='variation'*, *to_dataframe=False*)

QUANTITIES: *f_0* : HFSS Frequencies *f_1* : Analytical first order PT on the p=4 term of the cosine *f_ND* : Numerically diagonalized *chi_O1*: chi matrix from 1st order PT

Parameters **{[type]} -- [description]** (*quantity*) –

Keyword Arguments

- **{list of strings} -- Variations (default** (variations) – {None} – means all)
- **{str} -- Swept against (default** (*vs*) – {'variation'})
- **{bool} -- convert or not the result to dataframe.** (*to_dataframe*) – Make sure to call only if it can be converted to a DataFrame or can be concatenated into a multi-index DataFrame

Returns [type] – [description]

class pyEPR.core_quantum_analysis.**QuantumAnalysis**(*data_filename*, *variations: list = None*, *do_print_info=True*, *Res_hamil_filename=None*)

Bases: object

Defines an analysis object which loads and plots data from a h5 file This data is obtained using DistributedAnalysis

analyze_all_variations(*variations: List[str] = None*, *analyze_previous=False*, ***kwargs*)
See analyze_variation for full documentation

Parameters

- **variations** – None returns all_variations otherwise this is a list with number as strings ['0', '1']
- **analyze_previous** – set to true if you wish to overwrite previous analysis
- ****kwargs** – Keyword arguments passed to *analyze_variation()*.

analyze_variation(*variation: str*, *cos_trunc: int = None*, *fock_trunc: int = None*, *print_result: bool = True*, *junctions: List[T] = None*, *modes: List[T] = None*)
Core analysis function to call!

Parameters

- **junctions** – list or slice of junctions to include in the analysis. None defaults to analysing all junctions
- **modes** – list or slice of modes to include in the analysis. None defaults to analysing all modes

Returns

Dictionary containing at least the following:

- f_0 [MHz]: Eigenmode frequencies computed by HFSS; i.e., linear freq returned in GHz

- f_1 [MHz]: Dressed mode frequencies (by the non-linearity; e.g., Lamb shift, etc.). Result based on 1st order perturbation theory on the 4th order expansion of the cosine.
- f_ND [MHz]: Numerical diagonalization result of dressed mode frequencies. only available if *cos_trunc* and *fock_trunc* are set (non None).
- chi_O1 [MHz]: Analytic expression for the chis based on a cos trunc to 4th order, and using 1st order perturbation theory. Diag is anharmonicity, off diag is full cross-Kerr.
- chi_ND [MHz]: Numerically diagonalized chi matrix. Diag is anharmonicity, off diag is full cross-Kerr.

Return type dict

full_report_variations(*var_list: list = None*)
see full_variation_report

full_variation_report(*variation*)
prints the results and parameters of a specific variation

Parameters **variation** (*int or str*) – the variation to be printed .

Returns

Return type None.

get_Ecs(*variation*)
ECs in GHz Returns as pandas series

get_Ejs(*variation*)
EJs in GHz See calcs.convert

get_ansys_energies(*swp_var='variation'*)
Return a multi-index dataframe of ansys energies vs swep_variable

Parameters **swp_var** (*str*) –

get_chis(*swp_variable='variation'*, *numeric=True*, *variations: list = None*, *m=None*, *n=None*)
return as multiindex data table

If you provide m and n as integers or mode labels, then the chi between these modes will be returned as a pandas Series.

get_convergences_max_delta_freq_vs_pass(*as_dataframe=True*)
Index([u'Pass Number', u'Solved Elements', u'Max Delta Freq. %'])

get_convergences_max_tets()
Index([u'Pass Number', u'Solved Elements', u'Max Delta Freq. %'])

get_convergences_tets_vs_pass(*as_dataframe=True*)
Index([u'Pass Number', u'Solved Elements', u'Max Delta Freq. %'])

get_epr_base_matrices(*variation*, *_renorm_pj=None*, *print_=False*)
Return the key matrices used in the EPR method for analytic calculations.

All as matrices

PJ Participation matrix, p_mj

SJ Sign matrix, s_mj

Om Omega_mm matrix (in GHz) (hbar = 1) Not radians.

EJ E_jj matrix of Josephson energies (in same units as hbar omega matrix)

PHI_zpf ZPFs in units of phi_0 reduced flux quantum

PJ_cap capacitive participation matrix

Return all as *np.array* PM, SIGN, Om, EJ, Phi_ZPF

get_frequencies(*swp_variable='variation'*, *numeric=True*, *variations: list = None*)
return as multiindex data table index: eigenmode label columns: variation label

get_mesh_tot()

get_participations(*swp_variable='variation'*, *variations: list = None*, *inductive=True*, *_normed=True*)

inductive (bool): EPR for junction inductance when True, else for capacitors

Returns a multiindex dataframe: index 0: sweep variable index 1: mode number column: junction number

Plot the participation ratio of all junctions for a given mode vs a sweep of Lj.

get_quality_factors(*swp_variable='variation'*, *variations: list = None*)
return as pd.Series index: eigenmode label columns: variation label

get_variable_value(*swpvar*, *lv=None*)

get_variable_vs(*swpvar*, *lv=None*)
lv is list of variations (example ['0', '1']), if None it takes all variations swpvar is the variable by which to organize

return: ordered dictionary of key which is the variation number and the magnitude of swaver as the item

get_variation_of_multiple_variables_value(*Var_dic*, *lv=None*)

SEE get_variations_of_variable_value

A function to return all the variations in which one of the variables has a specific value lv is list of variations (example ['0', '1']), if None it takes all variations Var_dic is a dic with the name of the variable as key and the value to filter as item

get_variations_of_variable_value(*swpvar*, *value*, *lv=None*)
A function to return all the variations in which one of the variables has a specific value lv is list of variations (example ['0', '1']), if None it takes all variations swpvar is a string and the name of the variable we wish to filter value is the value of swapvr in which we are interested

returns lv - a list of the variations for which swavr==value

get_vs_variable(*swp_var*, *attr: str*)
Convert the index of a dictionary that is stored here from variation number to variable value.

Parameters

- **swp_var** (*str*) – name of sweep variable in ansys
- **attr** – name of local attribute, eg.., 'ansys_energies'

plot_hamiltonian_results(*swp_variable: str = 'variation'*, *variations: list = None*, *fig=None*, *x_label: str = None*)
Plot results versus variation

Keyword Arguments

- **{str} -- Variable against which we swept. If none, then just** (*swp_variable*) – take the variation index (default: {None})
- **{list} -- [description] (default** (variations) – {None})

- **{[type]} -- [description] (default** (*fig*) – {None})

Returns fig, axs

plot_results(*result*, *Y_label*, *variable*, *X_label*, *variations: list = None*)

plotting_dic_x(*Var_dic*, *var_name*)

print_info()

print_result(*result*)
Utility reporting function

print_variation(*variation*)
Utility reporting function

project_info

quick_plot_chi_alpha(*mode1*, *mode2*, *swp_variable='variation'*, *ax=None*, *kw=None*, *numeric=False*)
Quick plot chi between mode 1 and mode 2.

If you select mode1=mode2, then you will plot the alpha

kw : extra plot arguments

quick_plot_convergence(*ax=None*)
Plot a report of the Ansys convergence vs pass number ona twin axis for the number of tets and the max delta frequency of the eignemode.

quick_plot_frequencies(*mode*, *swp_variable='variation'*, *ax=None*, *kw=None*, *numeric=False*)
Quick plot freq for one mode

kw : extra plot arguments

quick_plot_mode(*mode*, *junction*, *mode1=None*, *swp_variable='variation'*, *numeric=False*, *sharex=True*)
Create a quick report to see mode parameters for only a single mode and a cross-kerr coupling to another mode. Plots the participation and cross participation Plots the frequencie plots the anharmonicity

The values are either for the numeric or the non-numeric results, set by *numeric*

quick_plot_participation(*mode*, *junction*, *swp_variable='variation'*, *ax=None*, *kw=None*)
Quick plot participation for one mode

kw : extra plot arguments

report_results(*swp_variable='variation'*, *numeric=True*)
Report in table form the results in a markdown friendly way in Jupyter notebook using the pandas interface.

pyEPR.core_quantum_analysis.**extract_dic**(*name=None*, *file_name=None*)
#name is the name of the dictionary as saved in the npz file if it is None, the function will return a list of all dictionaries in the npz file file name is the name of the npz file

pyEPR.project_info module

Main interface module to use pyEPR.

Contains code to connect to Ansys and to analyze HFSS files using the EPR method.

This module handles the microwave part of the analysis and connection to

Further contains code to be able to do autogenerated reports,

class pyEPR.project_info.**ProjectInfo**(*project_path: str = None*, *project_name: str = None*, *design_name: str = None*, *setup_name: str = None*, *dielectrics_bulk: list = None*, *dielectric_surfaces: list = None*, *resistive_surfaces: list = None*, *seams: list = None*, *do_connect: bool = True*)

Bases: `object`

Primary class to store interface information between `pyEPR` and `Ansys`.

- **Ansys:** stores and provides easy access to the ansys interface classes *pyEPR.ansys.HfssApp*, *pyEPR.ansys.HfssDesktop*, *pyEPR.ansys.HfssProject*, *pyEPR.ansys.HfssDesign*, *pyEPR.ansys.HfssSetup* (which, if present could nbe a subclass, such as a driven modal setup *pyEPR.ansys.HfssDMSetup*, eigenmode *pyEPR.ansys.HfssEMSetup*, or Q3D *pyEPR.ansys.AnsysQ3DSetup*), the 3D modeler to design geometry *pyEPR.ansys.HfssModeler*.
- **Junctions:** The class stores params about the design that the user puts will use, such as the names and properties of the junctions, such as which rectangle and line is associated with which junction.

Note: Junction parameters. The junction parameters are stored in the `self.junctions` ordered dictionary

A Josephson tunnel junction has to have its parameters specified here for the analysis. Each junction is given a name and is specified by a dictionary. It has the following properties:

- **`Lj_variable` (str):** Name of HFSS variable that specifies junction inductance Lj defined on the boundary condition in HFSS. WARNING: DO NOT USE Global names that start with $.
- **`rect` (str):** String of Ansys name of the rectangle on which the lumped boundary condition is defined.
- **`line` (str):** Name of HFSS polyline which spans the length of the rectangle. Used to define the voltage across the junction. Used to define the current orientation for each junction. Used to define sign of ZPF.
- **`length` (str):** Length in HFSS of the junction rectangle and line (specified in meters). To create, you can use `epr.parse_units('100um')`.
- **`Cj_variable` (str, optional) [experimental]:** Name of HFSS variable that specifies junction inductance Cj defined on the boundary condition in HFSS. DO NOT USE Global names that start with `$`.

Warning: To define junctions, do **NOT** use global names! I.e., do not use names in ansys that start with `$`.

Note: Junction parameters example . To define junction parameters, see the following example

```
# Create project infor class
pinfo = ProjectInfo()

# Now, let us add a junction called `j1`, with the following properties
pinfo.junctions['j1'] = {
            'Lj_variable' : 'Lj_1', # name of Lj variable in Ansys
            'rect'        : 'jj_rect_1',
            'line'        : 'jj_line_1',
            #'Cj'          : 'Cj_1' # name of Cj variable in Ansys - optional
            }
```

To extend to define 5 junctions in bulk, we could use the following script

```
n_junctions = 5
for i in range(1, n_junctions + 1):
    pinfo.junctions[f'j{i}'] = {'Lj_variable' : f'Lj_{i}',
                                'rect'        : f'jj_rect_{i}',
                                'line'        : f'jj_line_{i}'}
```

---

check_connected()
Checks if fully connected including setup.

connect()
Do establish connection to Ansys desktop. Connects to project and then get design and setup

connect_design(*design_name: str = None*)
Sets self.design self.design_name

connect_project()
Sets self.app self.desktop self.project self.project_name self.project_path

connect_setup()
Connect to the first available setup or create a new in eigenmode and driven modal

Raises `Exception` – [description]

disconnect()
Disconnect from existing Ansys Desktop API.

get_all_object_names()
Returns array of strings

get_all_variables_names()
Returns array of all project and local design names.

get_dm()
Utility shortcut function to get the design and modeler.

```
oDesign, oModeler = pinfo.get_dm()
```

get_setup(*name: str*)
Connects to a specific setup for the design. Sets self.setup and self.setup_name.

Parameters

- **name** (*str*) – Name of the setup.
- **the setup does not exist, then throws a logger error.** (*If*) –
- **to None, in which case returns None** (*Defaults*) –

save()
Return all the data in a dictionary form that can be used to be saved

validate_junction_info()
Validate that the user has put in the junction info correctly. Do not also forget to check the length of the rectangles/line of the junction if you change it.

pyEPR.reports module

Module for reporting utility functions

@author: Zlatko K Minev

pyEPR.reports.**plot_convergence_f_vspass**(*ax*, *s*, *kw={}*)
For a single pass

pyEPR.reports.**plot_convergence_max_df**(*ax*, *s*, *kw={}*, *color='r'*)
For a single pass

pyEPR.reports.**plot_convergence_maxdf_vs_sol**(*ax*, *s*, *s2*, *kw={}*)
ax, 'Max Δf %', 'Solved elements', kw for plot

pyEPR.reports.**plot_convergence_solved_elem**(*ax*, *s*, *kw={}*, *color='b'*)
For a single pass

1.2 Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z