pypmj Documentation

Release 2.2.0

Carlo Barth

Apr 23, 2018

Contents

The pypmj (python project manager for JCMsuite; pronounce "*py pi m de*") package extends the python interface shipped with the finite element Maxwell solver JCMsuite, distributed by the JCMwave GmbH.

It simplifies the setup, execution and data storage of JCMsuite simulations. Some of the main advantages are:

- The JCMsuite installation directory, the preferred storage directories and computation resources can be set up using a configuration file.
- Projects can be collected in one place as a project library and used from there.
- Parameter scans can be efficiently executed and evaluated using the *SimulationSet* class. Different combinations of input parameter lists make nested loops unnecessary.
- User defined processing of post process results.
- Computational costs and user results are efficiently stored in an HDF5 data base.
- Automatic detection of known results in the database.

CHAPTER 1

Contents

1.1 pypmj package

1.1.1 Module contents

pypmj

The pypmj (python project manager for JCMsuite; pronounce "*py pi m de*") package extends the python interface shipped with the finite element Maxwell solver JCMsuite, distributed by the JCMwave GmbH.

It simplifies the setup, execution and data storage of JCMsuite simulations. Some of the main advantages are:

- The JCMsuite installation directory, the preferred storage directories and computation resources can be set up using a configuration file.
- Projects can be collected in one place as a project library and used from there.
- Parameter scans can be efficiently executed and evaluated using the *SimulationSet* class. Different combinations of input parameter lists make nested loops unnecessary.
- User defined processing of post process results.
- Computational costs and user results are efficiently stored in an HDF5 data base.
- Automatic detection of known results in the database.

pypmj.**import_jcmwave**(*jcm_install_path=None*)
Imports jcmwave as jcm and jcmwave.daemon as daemon into the pypmj namespace and sets the __jcm_version__ module attribute.

Parameters **jcm_install_path** (*str or NoneType, default None*) – Sets the path to the JCMsuite installation directory in the current configuration. If *None*, it is assumed that the path is already configured. Raises a *RuntimeError* in that case if the configuration is invalid.

pypmj.**jcm_license_info**(*log=True*, *return_output=False*)
Prints and/or returns the current JCMsuite license information. Returns None, if jcmwave is not yet imported.

pypmj.**jcm_version_info**(*log=True*, *return_output=False*)
Prints and/or returns the current JCMsuite version information. Returns None, if jcmwave is not yet imported.

pypmj.**load_config_file**(*filepath*)
Reset the current configuration and overwrite it with the configuration in the config file specified by *filepath*.

pypmj.**load_extension**(*ext_name*)
Loads the specified extension of pypmj.

See *pypmj.extensions* for a list of extensions.

pypmj.**set_log_file**(*directory='logs'*, *filename='from_date'*)
Sets up the logging to a log-file if this is not already configured.

Parameters

- **directory** (*str, default 'logs'*) – The directory in which the logging file should be created as an absolute or relative path. It will be created if does not exist.
- **filename** (*str, default 'from_date'*) – The name of the logging file. If 'from_date', a date string will be used (format: %y%m%d.log).

1.1.2 pypmj.core module

Defines the centerpiece class *SimulationSet* of pypmj and the abstraction layers for projects, single simulations. Also, more specialized simulation sets such as the *ConvergenceTest*-class are defined here.

Authors : Carlo Barth

class pypmj.core.**ConvergenceTest**(*project*, *keys_test*, *keys_ref*, *duplicate_path_levels=0*, *storage_folder='from_date'*, *storage_base='from_config'*, *transitional_storage_base=None*, *combination_mode='product'*, *check_version_match=True*, *resource_manager=None*)
Bases: object

Class to set up, run and analyze convergence tests for JCMsuite projects. A convergence test consists of a reference simulation and (a) test simulation(s). The reference simulation should be of much higher accuracy than any of the test simulations.

This class initializes two SimulationSet instances. All init arguments are the same as for SimulationSet, except that there are two sets of keys.

Parameters

- **project** (JCMProject, *str or tuple/list of the form (specifier,)* – working_dir) JCMProject to use for the simulations. If no JCMProject-instance is provided, it is created using the given specifier or, if project is of type tuple, using (specifier, working_dir) (i.e. JCMProject(project[0], project[1])).

- **keys_test/keys_ref** (*dict*) – These are keys-dicts as used to initialize a SimulationSet. The *keys_ref* must correspond to a single simulation. The syntax is the same as for SimulationSet, which we repeat here: There are two possible use cases:

  1. The keys are the normal keys as defined by JCMsuite, containing all the values that need to passed to parse the JCM-template files. In this case, a single computation is performed using these keys.

  2. The keys-dict contains at least one of the keys [*constants*, *geometry*, *parameters*] and no additional keys. The values of each of these keys must be of type dict again and contain the keys necessary to parse the JCM-template files. Depending on the *combination_mode*, loops are performed over any parameter-sequences provided in *geometry* or *parameters*. JCMgeo is only called if the keys in *geometry* change between consecutive runs. Keys in *constants* are not stored in the HDF5 store! Consequently, this information is lost, but also adds the flexibility to path arbitrary data types to JCMsuite that could not be stored in the HDF5 format.

- **duplicate_path_levels** (*int, default 0*) – For clearly arranged data storage, the folder structure of the current working directory can be replicated up to the level given here. I.e., if the current dir is /path/to/your/pypmj/ and duplicate_path_levels=2, the subfolders your/pypmj will be created in the storage base dir (which is controlled using the configuration file). This is not done if duplicate_path_levels=0.

- **storage_folder** (*str, default 'from_date'*) – Name of the subfolder inside the storage folder in which the final data is stored. If 'from_date' (default), the current date (%y%m%d) is used. Note: in contrast to a single SimulationSet, subfolders 'Test' and 'Reference' are created inside the storage folder for the two sets.

- **storage_base** (*str, default 'from_config'*) – Directory to use as the base storage folder. If 'from_config', the folder set by the configuration option Storage->base is used.

- **transitional_storage_base** (*str, default None*) – Use this directory as the "real" storage_base during the execution, and move all files to the path configured using *storage_base* and *storage_folder* afterwards. This is useful if you have a fast drive which you want to use to accelerate the simulations, but which you do not want to use as your global storage for simulation data, e.g. because it is to small.

- **combination_mode** (*{'product', 'list'}*) – Controls the way in which sequences in the *geometry* or *parameters* keys are treated.

  – If *product*, all possible combinations of the provided keys are used.

  – If *list*, all provided sequences need to be of the same length N, so that N simulations are performed, using the value of the i-th element of each sequence in simulation i.

- **check_version_match** (*bool, default True*) – Controls if the versions of JCMsuite and pypmj are compared to the versions that were used when the HDF5 store was used. This has no effect if no HDF5 is present, i.e. if you are starting with an empty working directory.

- **resource_manager** (*ResourceManager or NoneType, default None*) – You can pass your own *ResourceManager*-instance here, e.g. to configure the resources to use before the *ConvergenceTest* is initialized. The *resource_manager* will be used for both of the simulation sets. If *None*, a *ResourceManager*-instance will be created automatically.

add_resources(*n_shots=10*, *wait_seconds=5*, *ignore_fail=False*)

Tries to add all resources configured in the configuration using the JCMdaemon.

analyze_convergence_results(*dev_columns*, *sort_by=None*, *data_ref=None*)
Calculates the relative deviations to the reference data for the columns in the *dev_columns*. A new DataFrame containing the test simulation data and the relative deviations is created (as class attribute *analyzed_data*) and returned. It is sorted in ascending order by the first dev_column or by the one specified by *sort_by*. A list of all deviation column names is stored in the *deviation_columns* attribute.

If more than 1 *dev_columns* is given, the mean deviation is also calculated and stored in the DataFrame column 'deviation_mean'. It is used to sort the data if *sort_by* is None.

close_stores()
Closes all HDF5 stores.

get_current_resources()
Returns a list of the currently configured resources, i.e. the ones that will be added using *add_resources*.

make_simulation_schedule()
Same as for SimulationSet.

Calls the *make_simulation_schedule* method for both sets.

open_stores()
Opens all HDF5 stores.

reset_resources()
Resets the resources to the default configuration.

run(*run_ref_with_max_cores='AUTO'*, *save_run=False*, ***simuset_kwargs*)
Runs the reference and the test simulation sets using the simuset_kwargs, which are passed to the run-method of each SimulationSet-instance.

Parameters

- **run_ref_with_max_cores** (*str (DaemonResource nickname) or False,*) – default 'AUTO' If 'AUTO', the DaemonResource with the most cores is automatically determined and used for the reference simulation with a *multiplicity* of 1 and all configured cores as *n_threads*. If a nickname is given, all configured cores of this resource are used in the same way. If False, the currently active resource configuration is used. The configuration for the test simulation set remains untouched.
- **save_run** (*bool, default False*) – If True, the utility function *run_simusets_in_save_mode* is used for the run.

run_reference_simulation(*run_on_resource='AUTO'*, *save_run=False*, ***simuset_kwargs*)
Runs the reference simulation set using the *simuset_kwargs*, which are passed to the *run*-method.

Parameters

- **run_on_resource** (*str (DaemonResource.nickname) or False, default 'AUTO'*) – If 'AUTO', the DaemonResource with the most cores is automatically determined and used for the reference simulation with a *multiplicity* of 1 and all configured cores as *n_threads*. If a nickname is given, all configured cores of this resource are used in the same way. If False, the currently active resource configuration is used.
- **save_run** (*bool, default False*) – If True, the utility function *run_simusets_in_save_mode* is used for the run.

run_test_simulations(*save_run=False*, ***simuset_kwargs*)
Runs the test simulation set using the *simuset_kwargs*, which are passed to the *run*-method.

Parameters **save_run** (*bool, default False*) – If True, the utility function *run_simusets_in_save_mode* is used for the run.

use_only_resources(*names*)
Restrict the daemon resources to *names*. Only makes sense if the resources have not already been added.

Names that are unknown are ignored. If no valid name is present, the default configuration will remain untouched.

write_analyzed_data_to_file(*file_path=None*, *mode='CSV'*, ***kwargs*)
Writes the data calculated by *analyze_convergence_results* to a CSV or an Excel file.

mode must be either 'CSV' or 'Excel'. If *file_path* is None, the default name results.csv/xls in the storage folder is used. *kwargs* are passed to the corresponding pandas functions.

class pypmj.core.**JCMProject**(*specifier*, *working_dir=None*, *project_file_name=None*, *job_name=None*)
Bases: object

Represents a JCMsuite project, initialized using a path specifier (relative to the *projects* path specified in the configuration), checks its validity and provides functions to copy its content to a working directory, remove it afterwards, etc.

Parameters

- **specifier** (*str or list*) –

  Can be

  - a path relative to the *projects* path specified in the configuration, given as complete str to append or sequence of strings which are .joined by os.path.join(),
  - or an absolute path to the project directory.

- **working_dir** (*str or None, default None*) – The path to which the files in the project directory are copied. If None, a folder called *current_run* is created in the current working directory
- **project_file_name** (*str or None, default None*) – The name of the project file. If None, automatic detection is tried by looking for a .jcmp or .jcmpt file with a line that starts with the word *Project*. If this fails, an Exception is raised.
- **job_name** (*str or None, default None*) – Name to use for queuing system such as slurm. If None, a name is composed using the specifier.

copy_to(*path=None*, *overwrite=True*, *sys_append=True*)
Copies all files inside the project directory to path, overwriting it if overwrite=True, raising an Error otherwise if it already exists.

Note: Appends the path to sys.path if sys_append=True.

get_file_path(*file_name*)
Returns the full path to the file with *file_name* if present in the current project. If this project was already copied to a working directory, the path to this directory is used. Otherwise, the source directory is used.

get_project_file_path()
Returns the complete path to the project file.

merge_pp_files_to_project_file(*pp_files*)
Creates a backup of the project file and appends the contents of the *pp_files* (single file or list) to the project file. This is useful if additional post processes should be executed without modifying the original project file. The path to the backup file is stored in the *project_file_backup_path* attribute.

remove_working_dir()
Removes the working directory.

restore_original_project_file()
Overwrites the original project file with the backup version if it exists.

show_readme(*try_use_markdown=True*)
Returns the content of the README.md file, if present. If *try_use_markdown* is True, it is tried to display the mark down file in a parsed way, which might only work inside ipython/jupyter notebooks.

class pypmj.core.**QuantityMinimizer**(*project*, *fixed_keys*, *duplicate_path_levels=0*, *storage_folder='from_date'*, *storage_base='from_config'*, *combination_mode='product'*, *resource_manager=None*)
Bases: *pypmj.core.SimulationSet*

check_validity_of_input_args()
Checks if the provided *fixed_keys* describe a single simulation.

make_simulation_schedule()

minimize_quantity(*x*, *quantity_to_minimize*, *maximize_instead=False*, *processing_func=None*, *wdir_mode='keep'*, *jcm_geo_kwargs=None*, *jcm_solve_kwargs=None*, ***scipy_minimize_kwargs*)
TODO

Parameters

- **x** (*string type*) – Name of the input parameter which is the input argument to the function that will be minimized.
- **quantity_to_minimize** (*string type*) – The result quantity for which the minimium should be found. This must be calculated by the *processing_func*.
- **maximize_instead** (*bool, default False*) – Whether to search for the maximum instead of the minimum.
- **processing_func** (*callable or NoneType, default None*) – Function for result processing. If None, only a standard processing will be executed. See the docs of the Simulation.process_results-method for more info on how to use this parameter.
- **wdir_mode** (*{'keep', 'delete'}, default 'keep'*) – The way in which the working directories of the simulations are treated. If 'keep', they are left on disk. If 'delete', they are deleted.
- **jcm_solve_kwargs** (*jcm_geo_kwargs,*) – Keyword arguments which are directly passed to jcm.geo and jcm.solve, respectively.
- **will be passed to the scipy.optimize.minimize** (*scipy_minimize_kwargs*) –
- **function.** –

pickle_optimization_results(*file_name='optimization_results.pkl'*)

class pypmj.core.**ResourceManager**
Bases: object

Class for convenient management of resources in all objects that are able to provoke simulations, i.e. call jcmwave.solve.

add_resources(*n_shots=10*, *wait_seconds=5*, *ignore_fail=False*)
Tries to add all resources configured in the configuration using the JCMdaemon.

get_current_resources()
Returns a list of the currently configured resources, i.e. the ones that will be added using *add_resources*.

load_state()
Loads a previously saved state.

reset_daemon()
Resets the JCMdaemon, i.e. disconnects it and resets the queue.

reset_resources()
Resets the resources to the default configuration.

save_state()
Saves the current resource configuration internally, allowing to reset it to this state later.

use_only_resources(*names*)
Restrict the daemon resources to *names*. Only makes sense if the resources have not already been added.

Names that are unknown are ignored. If no valid name is present, the default configuration will remain untouched.

use_single_resource_with_max_threads(*resource_nick=None*, *n_threads=None*)
Changes the current resource configuration to only a single resource. This resource can be specified by its *nickname*. If *resource_nick* is None, the resource with the maximum available cores will be detected automatically from the current configuration. The multiplicity of this resource will be set to 1, and the number of threads to the maximum or the given number *n_threads*.

class pypmj.core.**Simulation**(*keys*, *project=None*, *number=0*, *stored_keys=None*, *storage_dir=None*, *rerun_JCMgeo=False*, *store_logs=True*, *resultbag=None*, ***kwargs*)

Bases: object

Describes a distinct JCMsuite simulation by its keys and path/filename specific attributes. Provides method to perform the simulation , i.e. run JCMsolve on the project and to process the returned results using a custom function. It then also holds all the results, logs, etc. and can return them as a pandas DataFrame.

Parameters

- **keys** (*dict*) – The keys dict passed as the *keys* argument of jcmwave.solve. Used to translate JCM template files (i.e. **.jcmt*-files).
- **project** (*JCMProject, default None*) – The JCMProject instance related to this simulation.
- **number** (*int*) – A simulation number to identify/order simulations in a series of multiple simulations. It is used as the row index of the returned pandas DataFrame (e.g. by _get_DataFrame()).
- **stored_keys** (*list or NoneType, default None*) – A list of keys (must be a subset of *keys.keys()*) which will be part of the data in the pandas DataFrame, i.e. columns in the DataFrame returned by _get_DataFrame(). These keys will be stored in the HDF5 store by the SimulationSet-instance. If None, a it is tried to generate an as complete list of storable keys as possible automatically.
- **storage_dir** (*str (path)*) – Path to the directory were simulation working directories will be stored. The Simulation itself will be in a subfolder containing its number in the folder name. If None, the subdirectory 'standalone_solves' in the current working directory is used.
- **rerun_JCMgeo** (*bool, default False*) – Controls if JCMgeo needs to be called before execution in a series of simulations.
- **store_logs** (*bool, default True*) – If True, the 'Error' and 'Out' data of the logs returned by JCMsuite will be added to the results *dict* returned by *process_results*, and consequently stored in the HDF5 store by the parent *SimulationSet* instance.

- **resultbag** (*jcmwave.Resultbag or None, default None*) – *Experimental!*

  Assign a resultbag (see jcmwave.resultbag for details).

compute_geometry (***jcm_kwargs*)
Computes the geometry (i.e. runs jcm.geo) for this simulation.

The jcm_kwargs are directly passed to jcm.geo, except for *project_dir*, *keys* and *working_dir*, which are set automatically (ignored if provided).

find_file (*pattern*)
Finds a file in the working directory (see method *working_dir()*) matching the given (*fnmatch.filer-*) *pattern*. The working directory is scanned recursively.

Returns *None* if no match is found, the file path if a single file is found, or raises a *RuntimeError* if multiple files are found.

find_files (*pattern*, *only_one=False*)
Finds files in the working directory (see method *working_dir()*) matching the given (*fnmatch.filer-*) *pattern*. The working directory is scanned recursively.

If *only_one* is False (default), returns a list with matching file paths. Else, returns *None* if no match is found, the file path if a single file is found, or raises a *RuntimeError* if multiple files are found.

forget_jcm_results_and_logs ()

process_results (*processing_func=None*, *overwrite=False*)
Process the raw results from JCMsolve with a function *processing_func* of one input argument. The input argument, which is the list of results as it was set in *_set_jcm_results_and_logs*, is automatically passed to this function.

If *processing_func* is None, the JCM results are not processed and nothing will be saved to the HDF5 store, except for the computational costs.

The *processing_func* must be a function of one or two input arguments. A list of all results returned by post processes in JCMsolve are passed as the first argument to this function. If a second input argument is present, it must be called 'keys'. Then, the simulation keys are passed (i.e. self.keys). This is useful to use parameters of the simulation, e.g. the wavelength, inside your processing function. It must return a dict with key-value pairs that should be saved to the HDF5 store. Consequently, the values must be of types that can be stored to HDF5, otherwise Exceptions will occur in the saving steps.

remove_working_directory ()
Removes the working directory.

set_pass_computational_costs (*val*)
Sets the value of *pass_computational_costs*.

solve (*pp_file=None*, *additional_keys=None*, ***jcm_kwargs*)
Starts the simulation (i.e. runs jcm.solve) and returns the job ID.

Parameters

- **pp_file** (*str or NoneType, default None*) – File path to a JCM post processing file (extension .jcmp(t)). If None, the *get_project_file_path* of the current project is used and the mode 'solve' is used for jcmwave.solve. If not None, the mode 'post_process' is used.
- **additional_keys** (*dict or NoneType, default None*) – dict which will be merged to the *keys*-dict before passing them to the jcmwave.solve-method. Only new keys are added, duplicates are ignored and not updated.
- **jcm_kwargs are directly passed to jcm.solve, except for** (*The*) –

- **keys and working_dir, which are set** (*project_dir,*) –
- **(ignored if provided)** (*automatically*) –

solve_standalone(*processing_func=None*, *wdir_mode='keep'*, *run_post_process_files=None*, *resource_manager=None*, *additional_keys_for_pps=None*, *jcm_solve_kwargs=None*)

Solves this simulation and returns the results and logs.

Parameters

- **processing_func** (*callable or NoneType, default None*) – Function for result processing. If None, only a standard processing will be executed. See the docs of the Simulation.process_results-method for more info on how to use this parameter.
- **wdir_mode** (*{'keep', 'delete'}, default 'keep'*) – The way in which the working directories of the simulations are treated. If 'keep', they are left on disk. If 'delete', they are deleted.
- **run_post_process_files** (*str, list or NoneType, default None*) – File path or list of file paths to post processing files (extension .jcmp(t)) which should be executed subsequent to the actual solve. This calls jcmwave.solve with mode *post_process* internally. The results are appended to the *jcm_results*-list of the *Simulation* instance.
- **resource_manager** (ResourceManager *or NoneType, default None*) – You can pass your own *ResourceManager*-instance here, e.g. to configure the resources to use before the *SimulationSet* is initialized. If *None*, a *ResourceManager*-instance will be created automatically.
- **additional_keys_for_pps** (*dict or NoneType, default None*) – dict which will be merged to the *keys*-dict of the *Simulation* instance before passing them to the jcmwave.solve-method in the post process run. This has no effect if *run_post_process_files* is None. Only new keys are added, duplicates are ignored and not updated.
- **jcm_solve_kwargs** (*dict or NoneType, default None*) – These keyword arguments are directly passed to jcm.solve, except for *project_dir*, *keys* and *working_dir*, which are set automatically (ignored if provided).

view_geometry()

Opens the grid.jcm file using JCMview if it exists.

working_dir()

Returns the name of the working directory, specified by the storage_dir and the simulation number.

It is constructed using the global SIM_DIR_FMT formatter.

class pypmj.core.**SimulationSet**(*project*, *keys*, *duplicate_path_levels=0*, *storage_folder='from_date'*, *storage_base='from_config'*, *use_resultbag=False*, *transitional_storage_base=None*, *combination_mode='product'*, *check_version_match=True*, *resource_manager=None*, *store_logs=False*, *minimize_memory_usage=False*)

Bases: object

Class for initializing, planning, running and processing multiple simulations.

Parameters

- **project** (JCMProject, *str or tuple/list of the form (specifier,*working_dir*)*) – JCMProject to use for the simulations. If no JCMProject-instance is provided, it is created using the given specifier or, if project is of type tuple, using (specifier, working_dir) (i.e. JCMProject(project[0], project[1])).

- **keys** (*dict*) – There are two possible use cases:

  1. The keys are the normal keys as defined by JCMsuite, containing all the values that need to passed to parse the JCM-template files. In this case, a single computation is performed using these keys.
  2. The keys-dict contains at least one of the keys [*constants*, *geometry*, *parameters*] and no additional keys. The values of each of these keys must be of type dict again and contain the keys necessary to parse the JCM-template files. Depending on the *combination_mode*, loops are performed over any parameter-sequences provided in *geometry* or *parameters*. JCMgeo is only called if the keys in *geometry* change between consecutive runs. Keys in *constants* are not stored in the HDF5 store! Consequently, this information is lost, but also adds the flexibility to path arbitrary data types to JCMsuite that could not be stored in the HDF5 format.

- **duplicate_path_levels** (*int, default 0*) – For clearly arranged data storage, the folder structure of the current working directory can be replicated up to the level given here. I.e., if the current dir is /path/to/your/pypmj/ and duplicate_path_levels=2, the subfolders your/pypmj will be created in the storage base dir (which is controlled using the configuration file). This is not done if duplicate_path_levels=0.

- **storage_folder** (*str, default 'from_date'*) – Name of the subfolder inside the storage folder in which the final data is stored. If 'from_date' (default), the current date (%y%m%d) is used.

- **storage_base** (*str, default 'from_config'*) – Directory to use as the base storage folder. If 'from_config', the folder set by the configuration option Storage->base is used.

- **use_resultbag** (*bool, str (file path) or jcmwave.Resultbag, default False*) – *Experimental!*

  Whether to use a resultbag (see jcmwave.resultbag for details). If a *str* is given, it is considered as the path to the resultbag-file. If a *False*, the standard saving process using directories and data files is used. If *True*, the standard resultbag file *'resultbag.db'* in the storage directory is used. You can also pass a *jcmwave.Resultbag*-instance. Use the *get_resultbag_path()*-method to get the path of the current resultbag. *resultbag()* returns the *jcmwave.Resultbag*-instance. Use the methods *rb_get_log_for_sim* and *rb_get_result_for_sim* to get logs and results from the resultbag for a particular simulation. Note: using a resultbag will ignore settings for *store_logs*.

- **transitional_storage_base** (*str, default None*) – Use this directory as the "real" storage_base during the execution, and move all files to the path configured using *storage_base* and *storage_folder* afterwards. This is useful if you have a fast drive which you want to use to accelerate the simulations, but which you do not want to use as your global storage for simulation data, e.g. because it is to small.

- **combination_mode** (*{'product', 'list'}*) – Controls the way in which sequences in the *geometry* or *parameters* keys are treated.

  – If *product*, all possible combinations of the provided keys are used.

  – If *list*, all provided sequences need to be of the same length N, so that N simulations are performed, using the value of the i-th element of each sequence in simulation i.

- **check_version_match** (*bool, default True*) – Controls whether the versions of JCMsuite and pypmj are compared to the versions that were used when the HDF5 store was created. This has no effect if no HDF5 store is present, i.e. if you are starting with an empty working directory.

- **resource_manager** (*ResourceManager or NoneType, default None*) – You can pass your own *ResourceManager*-instance here, e.g. to configure the resources to use before the *SimulationSet* is initialized. If *None*, a *ResourceManager*-instance will be created automatically.
- **store_logs** (*bool, default False*) – Whether to store the JCMsuite logs to the HDF5 file (these may be cropped in some cases).
- **minimize_memory_usage** (*bool, default False*) – Huge parameter scans can cause python to need massive memory because the results and logs are kept for each simulation. Set this parameter to true to minimize the memory usage. Caution: you will loose all the *jcm_results* and *logs* in the *Simulation*-instances.

STORE_META_GROUPS = ['parameters', 'geometry']

STORE_VERSION_GROUP = 'version_data'

add_resources(*n_shots=10*, *wait_seconds=5*, *ignore_fail=False*)
Tries to add all resources configured in the configuration using the JCMdaemon.

all_done()
Checks if all simulations are done, i.e. already in the HDF5 store.

append_store(*data*)
Appends a new row or multiple rows to the HDF5 store.

close_store()
Closes the HDF5 store.

compute_geometry(*simulation*, ***jcm_kwargs*)
Computes the geometry (i.e. runs jcm.geo) for a specific simulation of the simulation set.

Parameters

- **simulation** (*Simulation or int*) – The *Simulation*-instance for which the geometry should be computed. If the type is *int*, it is treated as the index of the simulation in the simulation list.
- **jcm_kwargs are directly passed to jcm.geo, except for**(*The*) –
- **keys and working_dir, which are set automatically** (*project_dir,*) –
- **if provided)** ((*ignored*) –

fix_h5_store(*try_restructure=True*, *brute_force=False*)
Tries to remove duplicate rows in the HDF5 store based on the stored keys. If *try_restructure* is True, the HDF5 store is also restructured using *ptrepack* to possibly free disc space and optimize the compression. If problems persist, set *brute_force=True* which will remove all rows with duplicate indices (warning: data gets lost!).

get_all_keys()
Returns a list of all keys that are passed to JCMsolve.

get_current_resources()
Returns a list of the currently configured resources, i.e. the ones that will be added using *add_resources*.

get_project_wdir()
Returns the path to the working directory of the current project.

get_resultbag_path()

get_store_data()
Returns the data currently in the store.

is_store_empty()
Checks if the HDF5 store is empty.

make_simulation_schedule(*fix_h5_duplicated_rows=False*)
Makes a schedule by getting a list of simulations that must be performed, reorders them to avoid unnecessary calls of JCMgeo, and checks the HDF5 store for simulation data which is already known. If duplicated rows are found, a *RuntimeError* is raised. In this case, you can rerun *make_simulation_schedule* with *fix_h5_duplicated_rows=True* to try to automatically fix it. Alternatively, you could call the *fix_h5_store*-method yourself.

num_sims_to_do()
Returns the number of simulations that still needs to be solved, i.e. which are not already in the store.

open_store()
Closes the HDF5 store.

rb_get_log_for_sim(*sim*)
Returns the logs for the simulation *sim* from the resultbag. *sim* must be simulation number or a *Simulation*-instance of the current *simulations*-list.

rb_get_result_for_sim(*sim*)
Returns the logs for the simulation *sim* from the resultbag. *sim* must be simulation number or a *Simulation*-instance of the current *simulations*-list.

reset_resources()
Resets the resources to the default configuration.

resultbag()
Returns the resultbag (*jcmwave.Resultbag*-instance) if configured using the class attribute *use_resultbag*. Else, raises RuntimeError.

run(*processing_func=None*, *N='all'*, *auto_rerun_failed=1*, *run_post_process_files=None*, *additional_keys=None*, *wdir_mode='keep'*, *zip_file_path=None*, *show_progress_bar=False*, *jcm_geo_kwargs=None*, *jcm_solve_kwargs=None*, *pass_ccosts_to_processing_func=False*)
Convenient function to add the resources, run all necessary simulations and save the results to the HDF5 store.

Parameters

- **processing_func** (*callable or NoneType, default None*) – Function for result processing. If None, only a standard processing will be executed. See the docs of the Simulation.process_results-method for more info on how to use this parameter.
- **N** (*int or 'all', default 'all'*) – Number of simulations that will be pushed to the jcm.daemon at a time. If 'all', all simulations will be pushed at once. If many simulations are pushed to the daemon, the number of files and the size on disk can grow dramatically. This can be avoided by using this parameter, while deleting or zipping the working directories at the same time using the *wdir_mode* parameter.
- **auto_rerun_failed** (*int or bool, default 1*) – Controls whether/how often a simulation which failed is automatically rerun. If False or 0, no automatic rerunning will be done.
- **run_post_process_files** (*str, list or NoneType, default None*) – File path or list of file paths to post processing files (extension .jcmp(t)) which should be executed subsequent to the actual solve. In contrast to the procedure in the *solve_single_simulation* method, a merged project file is created in this case, i.e. the content of the post processing files is appended to the actual project file. The original project file is backed up and restored after the run.

- **additional_keys** (*dict or NoneType, default None*) – dict which will be merged to the *keys*-dict of the *Simulation* instance before passing them to the jcmwave.solve-method. Only new keys are added, duplicates are ignored and not updated. These values are not stored in the HDF5 store!
- **wdir_mode** (*{'keep', 'zip', 'delete'}, default 'keep'*) – The way in which the working directories of the simulations are treated. If 'keep', they are left on disk. If 'zip', they are appended to the zip-archive controled by *zip_file_path*. If 'delete', they are deleted. Caution: if you zip the directories and extend your data later in a way that the simulation numbers change, problems may occur.
- **zip_file_path** (*str (file path) or None*) – Path to the zip file if *wdir_mode* is 'zip'. The file is created if it does not exist. If None, the default file name 'working_directories.zip' in the current *storage_dir* is used.
- **jcm_solve_kwargs** (*jcm_geo_kwargs,*) – Keyword arguments which are directly passed to jcm.geo and jcm.solve, respectively.
- **pass_ccosts_to_processing_func** (*bool, default False*) – Whether to pass the computational costs as the 0th list element to the processing_func.

solve_single_simulation(*simulation*, *compute_geometry=True*, *run_post_process_files=None*, *additional_keys_for_pps=None*, *jcm_geo_kwargs=None*, *jcm_solve_kwargs=None*)

Solves a specific simulation and returns the results and logs without any further processing and without saving of data to the HDF5 store. Recomputes the geometry before if compute_geometry is True.

Parameters

- **simulation** (Simulation *or int*) – The *Simulation*-instance for which the geometry should be computed. If the type is *int*, it is treated as the index of the simulation in the simulation list.
- **compute_geometry** (*bool, default True*) – Runs jcm.geo before the simulation if True.
- **run_post_process_files**(*str, list or NoneType, default None*) – File path or list of file paths to post processing files (extension .jcmp(t)) which should be executed subsequent to the actual solve. This calls jcmwave.solve with mode *post_process* internally. The results are appended to the *jcm_results*-list of the *Simulation* instance. Note: this feature is yet incompatible with *use_resultbag*!
- **additional_keys_for_pps** (*dict or NoneType, default None*) – dict which will be merged to the *keys*-dict of the *Simulation* instance before passing them to the jcmwave.solve-method in the post process run. This has no effect if *run_post_process_files* is None. Only new keys are added, duplicates are ignored and not updated.
- **jcm_geo_kwargs** (*dict or NoneType, default None*) – These keyword arguments are directly passed to jcm.geo, except for *project_dir*, *keys* and *working_dir*, which are set automatically (ignored if provided).
- **jcm_solve_kwargs** (*dict or NoneType, default None*) – These keyword arguments are directly passed to jcm.solve, except for *project_dir*, *keys* and *working_dir*, which are set automatically (ignored if provided).

use_only_resources(*names*)

Restrict the daemon resources to *names*. Only makes sense if the resources have not already been added.

Names that are unknown are ignored. If no valid name is present, the default configuration will remain untouched.

write_store_data_to_file(*file_path=None*, *mode='CSV'*, ***kwargs*)
Writes the data that is currently in the store to a CSV or an Excel file.

mode must be either 'CSV' or 'Excel'. If *file_path* is None, the default name results.csv/xls in the storage folder is used. *kwargs* are passed to the corresponding pandas functions.

1.1.3 pypmj.parallelization module

Definitions classes for convenient usage of the jcmwave.daemon to run jobs in parallel. The class *DaemonResource* gives eaccess to both, workstations and queues and eases their configuration. The *ResourceDict*-class serves as a set of such resources and provides methods to set their properties all at once.

Authors : Carlo Barth

exception pypmj.parallelization.**DaemonError**(*message*)
Bases: exceptions.Exception

Exception raised for errors in adding daemon resources.

expression
Input expression in which the error occurred.

message
str – Explanation of the error.

class pypmj.parallelization.**DaemonResource**(*daemon_*, *hostname*, *login*, *JCM_root*, *multiplicity_default*, *n_threads_default*, *stype*, *nickname*, ***kwargs*)
Bases: object

Computation resource that can be used by the daemon-module of the JCMsuite python interface. This can be a workstation or a queue.

Holds all properties which are necessary to call the *add_workstation* or *add_queue* methods of the *jcmwave.daemon*. Frequently changed attributes like the multiplicity and the number of threads can be changed by convenient methods. Default values for these properties can be restored, just as every other state can be saved and restored.

Parameters

- **daemon** (*module*) – The *daemon* submodule of the *jcmwave* package delivered with your JCMsuite installation.
- **hostname** (*str*) – Hostname of the server as it would be used for e.g. ssh. Use *localhost* for the local computer.
- **JCM_root** (*str (path), default None*) – Path to the JCMsuite root installation folder. If None, the same path as on the local computer is assumed.
- **login** (*str*) – The username used for login (a password-free login is required)
- **multiplicity_default** (*int*) – The default number of CPUs to use on this server.
- **n_threads_default** (*int*) – The default number of threads per CPU to use on this server.
- **stype** (*{'Workstation', 'Queue'}*) – Type of the resource to use in the JCMsuite daemon utility.
- **nickname** (*str, default None*) – Shorthand name to use for this server. If None, the *hostname* is used.

- ****kwargs** – Add additional key-value pairs to pass to the daemon functions (which are *add_workstation* and *add_queue*) on your own risk.

add()
Adds the resource to the current daemon configuration.

add_repeatedly(*n_shots=10*, *wait_seconds=5*, *ignore_fail=False*)
Tries to add the resource repeatedly for *n_shots* times.

get_available_cores()
Returns the total number of currently configured cores for this resource, i.e. multiplicity*n_threads.

maximize_multiplicity(*multiplicity=None*)
Changes n_threads to 1 and the multiplicity to the product of the currently configured *multiplicity* and *n_threads* or to the given number *multiplicity*.

maximize_n_threads(*n_threads=None*)
Changes the multiplicity to 1 and the number of threads to the product of the currently configured *multiplicity* and *n_threads* or to the given number *n_threads*.

restore_default_m_n()
Restores the default values for multiplicity and n_threads.

restore_previous_m_n()
Restores the default values for multiplicity and n_threads.

save_m_n()
Saves the currently active multiplicity and n_threads.

They can be restored using the *restore_previous_m_n*-method.

set_m_n(*m*, *n*)
Shorthand for setting multiplicity and n_threads both at a time.

set_multiplicity(*value*)
Set the number of CPUs to use.

set_n_threads(*value*)
Set the number of threads to use per CPU.

class pypmj.parallelization.**ResourceDict**(**args*, ***kwargs*)
Bases: dict

Subclass of dict for extended handling of DaemonResource instances.

add_all()
Calls the *add* method for all resources.

add_all_repeatedly(*n_shots=10*, *wait_seconds=5*, *ignore_fail=False*)
Calls the *add_repeatedly* method for all resources.

get_all_queues()
Returns a list of all resources with stype==’Queue’.

get_all_workstations()
Returns a list of all resources with stype==’Workstation’.

get_resource_names()
Just a more meaningful name for the keys()-method.

get_resource_with_most_cores()
Determines which of the resources has the most usable cores, i.e. multiplicity*n_threads, and returns its nickname and this number.

get_resources()
Just a more meaningful name for the values()-method.

set_m_n_for_all(*m*, *n*)
Shorthand for setting multiplicity and n_threads for all resources.

pypmj.parallelization.**read_resources_from_config**(*daemon_*)
Reads all server configurations from the configuration file.

It is assumed that each server is in a section starting with *Server:*. For convenience, use the function *addServer* provided in *write_config_file.py*.

pypmj.parallelization.**savely_convert_config_value**(*value*)
Tries to convert a configuration value from a string type to int. If *value* is not a string type, a *ConfigurationError* is raised. If *value* does not consist of digits only, the input string is returned.

1.1.4 pypmj.utils module

Defines functions and classes which are internally used in all parts of pypmj, but may also be relevant to the user. Most importantly, the functions *run_simusets_in_save_mode* and *send_status_email* are defined here.

Authors : Carlo Barth

class pypmj.utils.**Capturing**
Bases: list

Context manager to capture any output printed to stdout.

based on: http://stackoverflow.com/questions/16571150/ how-to-capture-stdout-output-from-a-python-function-call

class pypmj.utils.**DisableLogger**(*level=20*)
Bases: object

Context manager to disable all logging events below specific level.

pypmj.utils.**append_dir_to_zip**(*directory*, *zip_file_path*)
Appends a directory to a zip-archive.

Raises an exception if the directory is already inside the archive.

pypmj.utils.**assign_kwargs_to_functions**(*functions*, *kwargs*, *ignore_unmatched=True*)
Uses *inspect* to assign which argument in *kwargs* belongs to which of the functions in the *functions* list.

If functions have any common argument names, an Error is raised. If *ignore_unmatched* is True, unassigned arguments are ignored. Returns a list of kwargs-dictionaries, one for each function.

pypmj.utils.**check_type_consistency_in_sequence**(*sequence*)
Checks if all elements of a sequence have the same type.

pypmj.utils.**computational_costs_to_flat_dict**(*ccosts*, *_sub=False*)
Converts the computational costs dict as returned by JCMsolve to a flat dict with only scalar values (i.e. numbers or strings).

This is useful to store the computational costs in a pandas DataFrame. Keys which have sequence values with a length other than 1 are converted to single values, while appending an underscore plus index to the key.

pypmj.utils.**file_content**(*file_path*)
Returns the content of an existing file.

pypmj.utils.**get_folders_in_zip**(*zipf*)
Returns a list of all folders and files in the root level of an open ZipFile.

pypmj.utils.**get_len_of_parameter_dict**(*d*)
Given a dict, returns the length of the longest sequence in its values.

pypmj.utils.**infer_dtype**(*obj*)
Tries to infer the numpy.dtype (or equivalent) of the elements of a sequence, or the numpy.dtype (or equivalent) of the object intelf if it is no sequence.

pypmj.utils.**is_callable**(*obj*)
Return whether the object is callable (i.e., some kind of function).

Note that classes are callable, as are instances with a __call__() method.

pypmj.utils.**is_sequence**(*obj*)
Checks if a given object is a sequence by checking if it is not a string or dict, but has a __len__-method.

This might fail!

pypmj.utils.**lists_overlap**(*list_1*, *list_2*)
Checks if two lists have no common elements.

pypmj.utils.**obj_to_fixed_length_Series**(*obj*, *length*)
Generates a pandas Series with a fixed len of *length* with the best matching dtype for the object.

If the object is sequence, the rows of the Series are filled with its elements. Otherwise it will be the value of the first row.

pypmj.utils.**query_yes_no**(*question*, *default='yes'*)
Ask a yes/no question via raw_input() and return their answer.

"question" is a string that is presented to the user. "default" is the presumed answer if the user just hits <Enter>. It must be "yes" (the default), "no" or None (meaning an answer is required of the user).

pypmj.utils.**relative_deviation**(*sample*, *reference*)
Returns the relative deviation d=|A/B-1| of sample A and reference B.

A can be a (complex) number or a list/numpy.ndarray of (complex) numbers. In case of complex numbers, the average relative deviation of real and imaginary part (d_real+d_imag)/2 is returned.

pypmj.utils.**rename_directories**(*renaming_dict*)
Safely renames directories given as old_name:new_name pairs as keys and values in the renaming_dict.

It first renames all old names to unique temporary names, and renames these to the new_names in a second step. This produces some overhead, but circumvents the problem of overlapping names in the old and new names. Safely ignores missing directories.

pypmj.utils.**rm_empty_directory_tail**(*path*, *stop_at=None*)
Removes all empty directories of a path recursively, starting at the tail, until a non empty directory is found or *path* is the same directory given in *stop_at*.

pypmj.utils.**run_simusets_in_save_mode**(*simusets*, *Ntrials=5*, ***kwargs*)
Given a list of SimulationSets, tries to run each SimulationSet *Ntrials* times, starting at the point where it was terminated by an unwanted error.

The *kwargs* are passed to the run-method of each set or to the *send_status_email* utility function. They are automatically assigned. Status e-mails are sent if configured in the configuration file.

pypmj.utils.**send_status_email**(*text*, *subject='JCMwave Simulation Information'*, *subject_prefix=''*, *subject_suffix=''*)
Tries to send a status e-mail with the given *text* using the configured e-mail server and address.

pypmj.utils.**split_path_to_parts**(*path*)
Splits a path to its parts, so that *os.path.join(*parts)* gives the input path again.

pypmj.utils.**tForm**(*t1*)
Returns a well formated time string.

pypmj.utils.**wait_for_all_other_daemons**()
Waits for all other currently active JCMdaemon processes to finish on UNIX systems.

pypmj.utils.**walk_df**(*df*, *col_vals*, *keys=None*)
Recursively finds a row in a pandas DataFrame where all values match the values given in col_vals for the keys (i.e. column specifiers) in *keys*.

If no matching rows exist, None is returned. If multiple matching rows exist, a list of indices of the matching rows is returned.

Parameters

- **df** (*pandas.DataFrame*) – This is the DataFrame in which a matching row should be found. For efficiency, it is not checked if the keys are present in the columns of df, so this should be checked by the user.
- **col_vals** (*dict or OrderedDict*) – A dict that holds the (single) values the matching row of the DataFrame should have, so that df.loc[match_index, key) == col_vals[key] for all keys in the row with index *match_index*. If keys is only a subset of the keys in the dict, remaining key-value pairs are ignored.
- **keys** (*sequence (list/tuple/numpy.ndarray/etc.), default None*) – keys (i.e. columns in df) to use for the comparison. The keys must be present in col_vals. If keys is None, all keys of col_vals are used.

1.2 Extensions

1.2.1 pypmj.extension_antenna

TODO: Explanation

Authors: Niko Nikolay, Carlo Barth

class pypmj.extension_antenna.**FarFieldEvaluation**(*simulation=None*, *direction=None*, *resolution=25*, *geometry='2D'*, *subfolder='post_processes'*)

Bases: object

TODO: Explanation

Parameters

- **simulation** (pypmj.core.Simulation) – The simulation instance for which the far field evaluation should be performed.
- **direction** (*{'half_space_up', 'half_space_down', 'point_up',}*) – ‘point_down’, None} Direction specification for the far field evaluation. If None, the complete space will be considered. If ‘half_space_up’/ ‘half_space_down’, only the upper/lower half space will be considered. If ‘point_up’/’point_down’, a single evaluation point in upward/downward direction will be used. Note: If a point direction is used, the resolution parameter will be ignored.
- **resolution** (*int, default 25*) – ...
- **geometry** (*{'2D', '3D'}, default '2D'*) – ...

- **subfolder** (*str, default 'post_processes'*) – Folder name of the subfolder in the project working directory into which the post processing jcmp(t)-files should be written.

analyze_far_field(**simulation_solve_kwargs*)
Analyzes the far field of the current simulation. Checks if the expected .jcm-result files already exist and runs the simulation plus necessary post-processes if not. Afterwards, it executes the standard far field processing (using the *_process_far_field_data*-method).

load_far_field_data(*file_path*)
Loads far field data from the .npz-file located at *file_path*.

save_far_field_data(*file_path*, *compressed=True*)
Saves the far field data to the file at *file_path* using the numpy.savez (or numpy.savez_compressed method if *compressed* is True).

pypmj.extension_antenna.**far_field_processing_func**(*pps*)
This is the processing function for the far field evaluation as needed for the *core.Simulation.process_results*-method (which is also used be the *run*-methods). It reads the far field, refractive index and the evaluation points from the far field post-processes.

pypmj.extension_antenna.**read_jcm_far_field_tables**(*jcm_files*)
This is the processing function for the far field evaluation as needed for the *core.Simulation.process_results*-method (which is also used be the *run*-methods). It reads the far field, refractive index and the evaluation points from the far field post-processes.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

I

J

L

M

N

O

P

Q

R

S

T

U

V

W