BlueSky Prototype Model

Release v1.1

U.S. Energy Information Administration

Apr 01, 2025

CONTENTS

CHAPTER ONE

SPHINX DOCUMENTATION STRUCTURE

Sphinx organizes the documentation into the following sections:

1.1 Package Overview

The documentation starts with a general overview of the main package structure. In this project, the top-level package is *src*. Inside the *src* package, you will find the following main packages:

- **electricity**: Contains modules related to electricity modeling and calculations.
- **hydrogen**: Contains modules for hydrogen energy production, storage, and consumption.
- **residential**: Contains models and utilities related to residential energy use.
- **integrator**: Integrates components from various energy sources (electricity, hydrogen, etc.) into a cohesive system.

1.2 Submodules and Subpackages

Each package may contain additional submodules and subpackages, which are organized in the documentation as follows:

- **Package**: Each package (e.g., electricity, hydrogen) is documented with a high-level description of its purpose and contents.
- **Submodules**: The individual Python modules within each package are listed and documented. For example, the *integrator* package may contain submodules like *runner.py*, *utilites.py*, and *progress_plot.py*. Each of these modules will have its own section.
- **Subpackages**: If a package contains nested subpackages, these are also documented. For instance, if *electricity* has a subpackage *scripts*, it will have its own subsection with corresponding submodules.

Each module's docstrings are captured to provide detailed information about functions, classes, methods, and attributes.

1.3 Example Structure

For the *src* package, Sphinx might organize the contents as follows:

```
src (package)/
├── integrator (subpackage)/
│   ├── input
│   └── runner.py (module)
└── models (subpackage)/
```

(continues on next page)

(continued from previous page)

```
├── electricity (package)/
│   └── scripts (subpackage)/
│       ├── electricity_model.py (module)
│       └── preprocessor.py
├── hydrogen/
│   ├── utilities/
│   │   └── h2_functions.py
│   ├── model/
│   │   └── h2_model.py
│   └── etc.
└── residential/
    └── scripts/
        ├── residential.py
        └── utilites.py
```

In the HTML and Markdown outputs, each package and subpackage is represented with links **below** to the respective modules, making it easy to navigate between different sections of the documentation.

Use the **Search bar** on the top left to search for a specific function or module.

1.4 Model Structure

src.models	
src	This directory contains the subdirectory for the integrator module and the subdirectories for the sectoral modules.

1.4.1 src.models

Modules

electricity
hydrogen
residential

src.models.electricity

Modules

scripts

src.models.electricity.scripts

Modules

electricity_model	Electricity Model, a pyomo optimization model of the electric power sector.
postprocessor	This file is the main postprocessor for the electricity model.
preprocessor	This file is the main preprocessor for the electricity model.
runner	This file is a collection of functions that are used to build, run, and solve the electricity model.
utilities	This file is a collection of functions that are used in support of the electricity model.

src.models.electricity.scripts.electricity_model

Electricity Model, a pyomo optimization model of the electric power sector.

The class is organized by sections: settings, sets, parameters, variables, objective function, constraints, plus additional misc support functions.

Functions

getLogger([name])	Return a logger with the specified name, creating it if necessary.

Classes

HI(region, year)	(region, year)
Model(*args, **kwds)	This is the base model class for the models.
PowerModel(*args, **kwds)	A PowerModel instance.
defaultdict	defaultdict(default_factory=None, /, [...]) --> dict with default factory
em	alias of *ElectricityMethods*

class src.models.electricity.scripts.electricity_model.**PowerModel**(**args*, ***kwds*)

A PowerModel instance. Builds electricity pyomo model.

Parameters

- **all_frames** (*dictionary of pd.DataFrames*) – Contains all dataframes of inputs
- **setA** (Sets) – Contains all other non-dataframe inputs

_active

src.models.electricity.scripts.postprocessor

This file is the main postprocessor for the electricity model.

It writes out all relevant model outputs (e.g., variables, parameters, constraints). It contains:

- A function that converts pyomo component objects to dataframes
- A function that writes the dataframes to output directories
- A function to make the electricity output sub-directories
- The postprocessor function, which loops through the model component objects and applies the

functions to convert and write out the data to dfs to the electricity output sub-directories

Functions

`create_obj_df`(mod_object)	takes pyomo component objects (e.g., variables, parameters, constraints) and processes the pyomo data and converts it to a dataframe and then writes the dataframe out to an output dir.
`getLogger`([name])	Return a logger with the specified name, creating it if necessary.
`make_elec_output_dir`(output_dir)	generates an output subdirectory to write electricity model results.
`postprocessor`(instance)	master postprocessor function that writes out the final dataframes from to the electricity model.
`report_obj_df`(mod_object, instance, dir_out, ...)	Creates a df of the component object within the pyomo model, separates the key data into different columns and then names the columns if the names are included in the cols_dict.

Classes

`Path`(*args, **kwargs)	PurePath subclass that can make system calls.

`src.models.electricity.scripts.postprocessor.`**`make_elec_output_dir`**(*output_dir*)

generates an output subdirectory to write electricity model results. It includes subdirs for vars, params, constraints.

Returns
the name of the output directory

Return type
string

`src.models.electricity.scripts.postprocessor.`**`postprocessor`**(*instance*)

master postprocessor function that writes out the final dataframes from to the electricity model. Creates the output directories and writes out dataframes for variables, parameters, and constraints. Gets the correct columns names for each dataframe using the cols_dict.

Parameters
`instance` (*`pyomo model`*) – electricity concrete model

Returns
output directory name

Return type
string

`src.models.electricity.scripts.postprocessor.`**`report_obj_df`**(*mod_object*, *instance*, *dir_out*, *sub_dir*)

Creates a df of the component object within the pyomo model, separates the key data into different columns and then names the columns if the names are included in the cols_dict. Writes the df out to the output directory.

Parameters

- **obj** (*pyomo component object*) – e.g., pyo.Var, pyo.Set, pyo.Param, pyo.Constraint
- **instance** (*pyomo model*) – electricity concrete model
- **dir_out** (*str*) – output electricity directory
- **sub_dir** (*str*) – output electricity sub-directory

src.models.electricity.scripts.preprocessor

This file is the main preprocessor for the electricity model.

It established the parameters and sets that will be used in the model. It contains:

- A class that contains all sets used in the model
- A collection of support functions to read in and setup parameter data
- The preprocessor function, which produces an instance of the Set class and a dict of params
- A collection of support functions to write out the inputs to the output directory

Functions

add_season_index(cw_temporal, df, pos)	adds a season index to the input dataframe
avg_by_group(df, set_name, map_frame)	takes in a dataframe and groups it by the set specified and then averages the data.
capacitycredit_df(all_frames, setin)	builds the capacity credit dataframe
create_hourly_params(all_frames, key, cols)	Expands params that are indexed by season to be indexed by hour
create_hourly_sets(all_frames, df)	expands sets that are indexed by season to be indexed by hour
create_other_sets(all_frames, setin)	creates other (non-supply curve) sets
create_sc_sets(all_frames, setin)	creates supply curve sets
create_subsets(df, col, subset)	Create subsets off of full sets
fill_values(row, subset_list)	Function to fill in the subset values, is used to assign all years within the year solve range to each year the model will solve for.
hourly_sc_subset(all_frames, subset)	Creates sets/subsets that are related to the supply curve
hr_sub_sc_subset(all_frames, T_subset, hr_subset)	creates supply curve subsets by hour
load_data(tablename, metadata, engine)	loads the data from the SQL database; used in readin_sql function.
makedir(dir_out)	creates a folder directory based on the path provided
output_inputs(OUTPUT_ROOT)	function developed initial for QA purposes, writes out to csv all of the dfs and sets passed to the electricity model to an output directory.
preprocessor(setin)	main preprocessor function that generates the final dataframes and sets sent over to the electricity model.

continues on next page

Table 9 – continued from previous page

`print_sets`(setin)	function developed initially for QA purposes, prints out all of the sets passed to the electricity model.
`readin_csvs`(all_frames)	Reads in all of the CSV files from the input dir and returns a dictionary of dataframes, where the key is the file name and the value is the table data.
`readin_sql`(all_frames)	Reads in all of the tables from a SQL databased and returns a dictionary of dataframes, where the key is the table name and the value is the table data.
`scale_load`(data_root)	Reads in BaseLoad.csv (load for all regions/hours for first year) and LoadScalar.csv (a multiplier for all model years).
`scale_load_with_enduses`(data_root)	Reads in BaseLoad.csv (load for all regions/hours for first year), EnduseBaseShares.csv (the shares of demand for each enduse in the base year) and EnduseScalar.csv (a multiplier for all model years by enduse category).
`step_sub_sc_subset`(all_frames, T_subset, ...)	creates supply curve subsets by step
`subset_dfs`(all_frames, setin, i)	filters dataframes based on the values within the set
`time_map`(cw_temporal, rename_cols)	create temporal mapping parameters

Classes

`Path`(*args, **kwargs)	PurePath subclass that can make system calls.
`Sets`(settings)	Generates an initial batch of sets that are used to solve electricity model.

class `src.models.electricity.scripts.preprocessor.`**`Sets`**(*settings*)

Generates an initial batch of sets that are used to solve electricity model. Sets include:

- Scenario descriptor and model switches
- Regional sets
- Temporal sets
- Technology type sets
- Supply curve step sets
- Other

`src.models.electricity.scripts.preprocessor.`**`add_season_index`**(*cw_temporal*, *df*, *pos*)

adds a season index to the input dataframe

Parameters

- **cw_temporal** (*dataframe*) – dataframe that includes the season index
- **df** (*dataframe*) – parameter data to be modified
- **pos** (*int*) – column position for the seasonal set

Returns

modified parameter data now indexed by season

Return type

dataframe

`src.models.electricity.scripts.preprocessor.avg_by_group`(*df*, *set_name*, *map_frame*)

takes in a dataframe and groups it by the set specified and then averages the data.

Parameters

- **df** (*dataframe*) – parameter data to be modified
- **set_name** (*str*) – name of the column/set to average the data by
- **map_frame** (*dataframe*) – data that maps the set name to the new grouping for that set

Returns
parameter data that is averaged by specified set mapping

Return type
dataframe

`src.models.electricity.scripts.preprocessor.capacitycredit_df`(*all_frames*, *setin*)

builds the capacity credit dataframe

Parameters

- **all_frames** (*dict of pd.DataFrame*) – dictionary of dataframes where the key is the file name and the value is the table data
- **setin** (Sets) – an initial batch of sets that are used to solve electricity model

Returns
formatted capacity credit data frame

Return type
pd.DataFrame

`src.models.electricity.scripts.preprocessor.create_hourly_params`(*all_frames*, *key*, *cols*)

Expands params that are indexed by season to be indexed by hour

Parameters

- **all_frames** (*dict of pd.DataFrame*) – dictionary of dataframes where the key is the file name and the value is the table data
- **key** (*str*) – name of data frame to access
- **cols** (*list[str]*) – column names to keep in data frame

Returns
data frame with name key with new hourly index

Return type
pd.DataFrame

`src.models.electricity.scripts.preprocessor.create_hourly_sets`(*all_frames*, *df*)

expands sets that are indexed by season to be indexed by hour

Parameters

- **all_frames** (*dict of pd.DataFrame*) – dictionary of dataframes where the key is the file name and the value is the table data
- **df** (*pd.DataFrame*) – data frame containing seasonal data

Returns
data frame containing updated hourly set

Return type
pd.DataFrame

`src.models.electricity.scripts.preprocessor.`**`create_other_sets`**(*all_frames*, *setin*)

creates other (non-supply curve) sets

Parameters

- **all_frames** (*dict of pd.DataFrame*) – dictionary of dataframes where the key is the file name and the value is the table data
- **setin** (Sets) – an initial batch of sets that are used to solve electricity model

Returns
updated Sets which has non-supply curve-related sets updated

Return type
Sets

`src.models.electricity.scripts.preprocessor.`**`create_sc_sets`**(*all_frames*, *setin*)

creates supply curve sets

Parameters

- **all_frames** (*dict of pd.DataFrame*) – dictionary of dataframes where the key is the file name and the value is the table data
- **setin** (Sets) – an initial batch of sets that are used to solve electricity model

Returns
updated Set containing all sets related to supply curve

Return type
Sets

`src.models.electricity.scripts.preprocessor.`**`create_subsets`**(*df*, *col*, *subset*)

Create subsets off of full sets

Parameters

- **df** (*pd.DataFrame*) – data frame of full data
- **col** (*str*) – column name
- **subset** (*list[str]*) – names of values to subset

Returns
data frame containing subset of full data

Return type
pd.DataFrame

`src.models.electricity.scripts.preprocessor.`**`fill_values`**(*row*, *subset_list*)

Function to fill in the subset values, is used to assign all years within the year solve range to each year the model will solve for.

Parameters

- **row** (*int*) – row number in df
- **subset_list** (*list*) – list of values to map

Returns
value from subset_list

Return type
int

`src.models.electricity.scripts.preprocessor.`**`hourly_sc_subset`**(*all_frames*, *subset*)

Creates sets/subsets that are related to the supply curve

Parameters

- **all_frames** (*dict of pd.DataFrame*) – dictionary of dataframes where the key is the file name and the value is the table data
- **subset** (*list*) – list of technologies to subset

Returns

data frame containing sets/subsets related to supply curve

Return type

pd.DataFrame

`src.models.electricity.scripts.preprocessor.`**`hr_sub_sc_subset`**(*all_frames*, *T_subset*, *hr_subset*)

creates supply curve subsets by hour

Parameters

- **all_frames** (*dict of pd.DataFrame*) – dictionary of dataframes where the key is the file name and the value is the table data
- **T_subset** (*list*) – list of technologies to subset
- **hr_subset** (*list*) – list of hours to subset

Returns

data frame containing supply curve related hourly subset

Return type

pd.DataFrame

`src.models.electricity.scripts.preprocessor.`**`load_data`**(*tablename*, *metadata*, *engine*)

loads the data from the SQL database; used in readin_sql function.

Parameters

- **tablename** (*string*) – table name
- **metadata** (*SQL metadata*) – SQL metadata
- **engine** (*SQL engine*) – SQL engine

Returns

table from SQL db as a dataframe

Return type

dataframe

`src.models.electricity.scripts.preprocessor.`**`makedir`**(*dir_out*)

creates a folder directory based on the path provided

Parameters

dir_out (*str*) – path of directory

`src.models.electricity.scripts.preprocessor.`**`output_inputs`**(*OUTPUT_ROOT*)

function developed initial for QA purposes, writes out to csv all of the dfs and sets passed to the electricity model to an output directory.

Parameters

OUTPUT_ROOT (*str*) – path of output directory

Returns

- **all_frames** (*dictionary*) – dictionary of dataframes where the key is the file name and the value is the table data
- **setin** (*Sets*) – an initial batch of sets that are used to solve electricity model

`src.models.electricity.scripts.preprocessor.`**`preprocessor`**(*setin*)

main preprocessor function that generates the final dataframes and sets sent over to the electricity model. This function reads in the input data, modifies it based on the temporal and regional mapping specified in the inputs, and gets it into the final formatting needed. Also adds some additional regional sets to the set class based on parameter inputs.

Parameters
`setin` (`Sets`) – an initial batch of sets that are used to solve electricity model

Returns

- **all_frames** (*dictionary*) – dictionary of dataframes where the key is the file name and the value is the table data
- **setin** (*Sets*) – an initial batch of sets that are used to solve electricity model

`src.models.electricity.scripts.preprocessor.`**`print_sets`**(*setin*)

function developed initially for QA purposes, prints out all of the sets passed to the electricity model.

Parameters
`setin` (`Sets`) – an initial batch of sets that are used to solve electricity model

`src.models.electricity.scripts.preprocessor.`**`readin_csvs`**(*all_frames*)

Reads in all of the CSV files from the input dir and returns a dictionary of dataframes, where the key is the file name and the value is the table data.

Parameters
`all_frames` (*`dictionary`*) – empty dictionary to be filled with dataframes

Returns
completed dictionary filled with dataframes from the input directory

Return type
dictionary

`src.models.electricity.scripts.preprocessor.`**`readin_sql`**(*all_frames*)

Reads in all of the tables from a SQL databased and returns a dictionary of dataframes, where the key is the table name and the value is the table data.

Parameters
`all_frames` (*`dictionary`*) – empty dictionary to be filled with dataframes

Returns
completed dictionary filled with dataframes from the input directory

Return type
dictionary

`src.models.electricity.scripts.preprocessor.`**`step_sub_sc_subset`**(*all_frames*, *T_subset*, *step_subset*)

creates supply curve subsets by step

Parameters

- `all_frames` (*`dict of pd.DataFrame`*) – dictionary of dataframes where the key is the file name and the value is the table data
- `T_subset` (*`list`*) – technologies to subset

- **step_subset** (*list*) – step numbers to subset

Returns
data frame containing supply curve subsets by step

Return type
pd.DataFrame

`src.models.electricity.scripts.preprocessor.`**`subset_dfs`**(*all_frames*, *setin*, *i*)

filters dataframes based on the values within the set

Parameters

- **all_frames** (*dictionary*) – dictionary of dataframes where the key is the file name and the value is the table data
- **setin** (Sets) – contains an initial batch of sets that are used to solve electricity model
- **i** (*string*) – name of the set contained within the sets class that the df will be filtered based on.

Returns
completed dictionary filled with dataframes filtered based on set inputs specified

Return type
dictionary

`src.models.electricity.scripts.preprocessor.`**`time_map`**(*cw_temporal*, *rename_cols*)

create temporal mapping parameters

Parameters

- **cw_temporal** (*pd.DataFrame*) – temporal crosswalks
- **rename_cols** (*dict*) – columns to rename from/to

Returns
data frame with temporal mapping parameters

Return type
pd.DataFrame

src.models.electricity.scripts.runner

This file is a collection of functions that are used to build, run, and solve the electricity model.

Functions

build_elec_model(all_frames, setin)	building pyomo electricity model
check_results(results, SolutionStatus, ...)	Check results for termination condition and solution status
cost_learning_func(instance, tech, y)	function for updating learning costs by technology and year
getLogger([name])	Return a logger with the specified name, creating it if necessary.
init_old_cap(instance)	initialize capacity for 0th iteration
log_infeasible_constraints(m[, tol, logger, ...])	Logs the infeasible constraints in the model.
run_elec_model(settings[, solve])	build electricity model (and solve if solve=True) after passing in settings

continues on next page

Table 11 – continued from previous page

`select_solver`(instance)	Select solver based on learning method
`set_new_cap`(instance)	calculate new capacity after solve iteration
`solve_elec_model`(instance)	solve electicity model
`update_cost`(instance)	update capital cost based on new capacity learning

Classes

`Config_settings`(config_path[, args, test])	Generates the model settings that are used to solve.
`Path`(*args, **kwargs)	PurePath subclass that can make system calls.
`PowerModel`(*args, **kwds)	A PowerModel instance.
`SolutionStatus`(*values)	
`SolverStatus`(*values)	
`TerminationCondition`(*values)	
`TicTocTimer`([ostream, logger])	A class to calculate and report elapsed time.
`datetime`(year, month, day[, hour[, minute[, ...)	The year, month and day arguments are required.

`src.models.electricity.scripts.runner.`**`build_elec_model`**(*all_frames*, *setin*) → *PowerModel*

building pyomo electricity model

Parameters

- **`all_frames`** (*`dict of pd.DataFrame`*) – input data frames
- **`setin`** (`Sets`) – input settings Sets

Returns

built (but unsolved) electricity model

Return type

PowerModel

`src.models.electricity.scripts.runner.`**`cost_learning_func`**(*instance*, *tech*, *y*)

function for updating learning costs by technology and year

Parameters

- **`instance`** (`PowerModel`) – electricity pyomo model
- **`tech`** (*`int`*) – technology type
- **`y`** (*`int`*) – year

Returns

updated capital cost based on learning calculation

Return type

int

`src.models.electricity.scripts.runner.`**`init_old_cap`**(*instance*)

initialize capacity for 0th iteration

Parameters

`instance` (`PowerModel`) – unsolved electricity model `src.models.electricity.scripts.runner.`**`run_elec_model`**`(`*settings: Config_settings*, *solve=True*`)` → *PowerModel*

build electricity model (and solve if solve=True) after passing in settings

Parameters

- **settings** (`Config_settings`) – Configuration settings
- **solve** (*bool, optional*) – solve electricity model?, by default True

Returns

electricity model

Return type

PowerModel

`src.models.electricity.scripts.runner.`**`set_new_cap`**`(`*instance*`)`

calculate new capacity after solve iteration

Parameters

instance (`PowerModel`) – solved electricity pyomo model

`src.models.electricity.scripts.runner.`**`solve_elec_model`**`(`*instance*`)`

solve electicity model

Parameters

instance (`PowerModel`) – built (but not solved) electricity pyomo model

`src.models.electricity.scripts.runner.`**`update_cost`**`(`*instance*`)`

update capital cost based on new capacity learning

Parameters

instance (`PowerModel`) – electricity pyomo model

src.models.electricity.scripts.utilities

This file is a collection of functions that are used in support of the electricity model.

Functions

annual_count(hour, m)	return the aggregate weight of this hour in the representative year we know the hour weight, and the hours are unique to days, so we can get the day weight
check_results(results, SolutionStatus, ...)	Check results for termination condition and solution status
create_obj_df(mod_object)	takes pyomo component objects (e.g., variables, parameters, constraints) and processes the pyomo data and converts it to a dataframe and then writes the dataframe out to an output dir.

Classes

ElectricityMethods(*args, **kwds)	a collection of functions used within the electricity model that aid in building the model.
Model(*args, **kwds)	This is the base model class for the models.

continues on next page

Table 14 – continued from previous page

Path(*args, **kwargs)	PurePath subclass that can make system calls.
defaultdict	defaultdict(default_factory=None, /, [...]) --> dict with default factory

class src.models.electricity.scripts.utilities.**ElectricityMethods**(**args*, ***kwds*)

a collection of functions used within the electricity model that aid in building the model.

Parameters
`Model` (*Class*) – generic model class

_active

populate_RM_sets_rule()

Creates new reindexed sets for reserve margin constraint

Parameters
`m` (PowerModel) – pyomo electricity model instance

populate_by_hour_sets_rule()

Creates new reindexed sets for dispatch_cost calculations

Parameters
`m` (PowerModel) – pyomo electricity model instance

populate_demand_balance_sets_rule()

Creates new reindexed sets for demand balance constraint

Parameters
`m` (PowerModel) – pyomo electricity model instance

populate_hydro_sets_rule()

Creates new reindexed sets for hydroelectric generation seasonal upper bound constraint

Parameters
`m` (PowerModel) – pyomo electricity model instance

populate_reserves_sets_rule()

Creates new reindexed sets for operating reserves constraints

Parameters
`m` (PowerModel) – pyomo electricity model instance

populate_trade_sets_rule()

Creates new reindexed sets for trade constraints

Parameters
`m` (PowerModel) – pyomo electricity model instance

src.models.electricity.scripts.utilities.**annual_count**(*hour*, *m*) → int

return the aggregate weight of this hour in the representative year we know the hour weight, and the hours are unique to days, so we can get the day weight

Parameters
`hour` (*int*) – the rep_hour

Returns
the aggregate weight (count) of this hour in the rep_year. NOT the hour weight!

Return type
int

`src.models.electricity.scripts.utilities.`**`check_results`**(*results*, *SolutionStatus*, *TerminationCondition*)

Check results for termination condition and solution status

Parameters

- **`results`** (*`str`*) – Results from pyomo
- **`SolutionStatus`** (*`str`*) – Solution Status from pyomo
- **`TerminationCondition`** (*`str`*) – Termination Condition from pyomo

Return type
results

`src.models.electricity.scripts.utilities.`**`create_obj_df`**(*mod_object*)

takes pyomo component objects (e.g., variables, parameters, constraints) and processes the pyomo data and converts it to a dataframe and then writes the dataframe out to an output dir. The dataframe contains a key column which is the original way the pyomo data is structured, as well as columns broken out for each set and the final values.

Parameters
`mod_object` (*`pyomo component object`*) – pyomo component object

Returns
contains the pyomo model results for the component object

Return type
pd.DataFrame

src.models.hydrogen

Modules

model
network
utilities

src.models.hydrogen.model

Modules

actions	A sequencer for actions in the model.
h2_model	The Hydrogen Model takes in a Grid object and uses it to populate a Pyomo model that solves for the least cost to produce and distribute Hydrogen by electrolysis across the grid to satisfy a given demand, returning the duals as shadow prices.
validators	set of validator functions for use in model

src.models.hydrogen.model.actions

A sequencer for actions in the model. This may change up a bit, but it is a place to assert control of the execution sequence for now

Functions

build_grid(grid_data)	build a grid from grid_data
build_model(grid, **kwds)	build model from grd
check_optimal_termination(results)	This function returns True if the termination condition for the solver is 'optimal', 'locallyOptimal', or 'globallyOptimal', and the status is 'ok'
getLogger([name])	Return a logger with the specified name, creating it if necessary.
load_data(path_to_input, **kwds)	load data for model
make_h2_outputs(output_path, model)	save model outputs
quick_summary(solved_hm)	print and return summary of solve
run_hydrogen_model(settings)	run hydrogen model in standalone
solve(hm)	_summary_
solve_it(hm)	solve hm
value(obj[, exception])	A utility function that returns the value of a Pyomo object or expression.

Classes

Grid([data])	
GridData(data_folder[, regions_of_interest])	
H2Model(*args, **kwds)	
Path(*args, **kwargs)	PurePath subclass that can make system calls.
SolverResults(*args, **kwargs)	

src.models.hydrogen.model.actions.**build_grid**(*grid_data:* GridData) → *Grid*

build a grid from grid_data

Parameters
grid_data (*obj*) – GridData object to build grid from

Returns
Grid – Grid object

Return type
obj

src.models.hydrogen.model.actions.**build_model**(*grid:* Grid, ***kwds*) → *H2Model*

build model from grd

Parameters
grid (*obj*) – Grid object to build model from

Returns
H2Model – H2Model object

Return type
obj

`src.models.hydrogen.model.actions.`**`load_data`**(*path_to_input: Path*, ***kwds*) → *GridData*

load data for model

Parameters
`path_to_input` (*`Path`*) – Data folder path

Returns
GridData – Grid Data object from path

Return type
obj

`src.models.hydrogen.model.actions.`**`make_h2_outputs`**(*output_path*, *model*)

save model outputs

Parameters
`model` (*`obj`*) – Solved H2Model

`src.models.hydrogen.model.actions.`**`quick_summary`**(*solved_hm:* H2Model) → None

print and return summary of solve

Parameters
`solved_hm` (*`obj`*) – Solved H2Model

Returns
res – Printed summary

Return type
str

`src.models.hydrogen.model.actions.`**`run_hydrogen_model`**(*settings*)

run hydrogen model in standalone

Parameters
`settings` (*`obj`*) – Config_setup instance

`src.models.hydrogen.model.actions.`**`solve_it`**(*hm:* H2Model) → SolverResults

solve hm

Parameters
`hm` (*`objH2Model`*) – H2Model to solve

Returns
SolverResults – results of solve

Return type
obj

src.models.hydrogen.model.h2_model

The Hydrogen Model takes in a Grid object and uses it to populate a Pyomo model that solves for the least cost to produce and distribute Hydrogen by electrolysis across the grid to satisfy a given demand, returning the duals as shadow prices. It can be run in stand-alone or integrated runs. If stand-alone, a function for generated temporally varying data must be supplied. By default it simply projects geometric growth for electricity price and demand.

Functions

`check_optimal_termination`(results)	This function returns True if the termination condition for the solver is 'optimal', 'locallyOptimal', or 'globallyOptimal', and the status is 'ok'
`getLogger`([name])	Return a logger with the specified name, creating it if necessary.
`resolve`(hm[, new_demand, ...])	For convenience: After building and solving the model initially:
`solve`(hm)	_summary_
`value`(obj[, exception])	A utility function that returns the value of a Pyomo object or expression.

Classes

`Block`(*args, **kwds)	Blocks are indexed components that contain other components (including blocks).
`ConcreteModel`(*args, **kwds)	A concrete optimization model that does not defer construction of components.
`Constraint`(*args, **kwds)	This modeling component defines a constraint expression using a rule function.
`Grid`([data])	
`H2Model`(*args, **kwds)	
`HI`(region, year)	(region, year)
`LinearExpression`([args, constant, ...])	An expression object for linear polynomials.
`Objective`(*args, **kwds)	This modeling component defines an objective expression.
`Param`(*args, **kwds)	A parameter value, which may be defined over an index.
`RangeSet`(*args, **kwds)	A set object that represents a set of numeric values
`Set`(*args, **kwds)	A component used to index other Pyomo components.
`SolverResults`(*args, **kwargs)	
`Suffix`(*args, **kwargs)	A model suffix, representing extraneous model data
`Var`(*args, **kwargs)	A numeric variable, which may be defined over an index.
`defaultdict`	defaultdict(default_factory=None, /, [...]) --> dict with default factory

class `src.models.hydrogen.model.h2_model.`**`H2Model`**(**args*, ***kwds*)

`_active`

`_filter_update_info`(*data: dict[HI, float]*) → dict[*HI*, float]

quick filter to remove regions that don't exist in the model

It is possible (right now) that the H2 network is unaware of particular regions

because no baseline data for them was ever provided.... so it is possible to recieve and "unkown" region here, even though it was selected, due to lack of data

Parameters

- **hm** (H2Model) – self
- **data** (*dict[HI, float]*) – hydrogen index : value

Returns
regions index: value with missing data removed

Return type
dict[*HI*, float]

_update_demand(*new_demand*)

update the demand parameter with new demand data

insert new demand as a dict in the format: new_demand[region, year]

Parameters

- **hm** (H2Model) – self
- **new_demand** (*dict*) – new demand values

_update_electricity_price(*new_electricity_price*)

update electricity price parameter

Parameters

- **hm** (H2Model) – self
- **new_electricity_price** (*dict*) – region, year : electricity price

poll_electric_demand() → dict[*HI*, float]

compute the electrical demand by region-year after solve

Note: we will use production * 1/eff to compute electrical demand

Parameters
hm (H2Model) – self

Returns
electricity demand by region, year. (region, year):demand

Return type
dict[*HI*, float]

update_exchange_params(*new_demand=None*, *new_electricity_price=None*)

update exchange parameters in integrated mode

Parameters

- **hm** (H2Model) – model
- **new_demand** (*dict, optional*) – new demand (region, year):value. Defaults to None.
- **new_electricity_price** (*dict, optional*) – new electricity prices (region,year):value . Defaults to None.

src.models.hydrogen.model.h2_model.**resolve**(*hm: H2Model*, *new_demand=None*, *new_electricity_price=None*, *test=False*)

For convenience: After building and solving the model initially:

if you want to solve without annual data by applying a geometric growth rate to exhcange params

Parameters

- **hm** (H2Model) – model
- **new_demand** (*dict, optional*) – new_demand[region,year] for H2demand in (region,year). Defaults to None.
- **new_electricity_price** (*dict, optional*) – new_electricity_price[region,year]. Defaults to None.
- **test** (*bool, optional*) – is this just a test? Defaults to False.

`src.models.hydrogen.model.h2_model.`**`solve`**`(`*hm:* H2Model`)`

summary

Parameters
hm (H2Model) – self

Raises
RuntimeError – no optimal solution to problem

src.models.hydrogen.model.validators

set of validator functions for use in model

Functions

region_validator(hm, region)	checks if region name is string or numeric

`src.models.hydrogen.model.validators.`**`region_validator`**`(`*hm:* H2Model, *region*`)`

checks if region name is string or numeric

hm
[H2Model] model

region
[any] region name

Raises
ValueError – region wrong type

Returns
is correct type

Return type
bool

src.models.hydrogen.network

Modules

grid	GRID CLASS
grid_data	GRIDDATA CLASS
hub	HUB CLASS
region	REGION CLASS
registry	REGISTRY CLASS
transportation_arc	TRANSPORTATION ARC CLASS

src.models.hydrogen.network.grid

GRID CLASS

This is the central class that binds all the other classes together. No class instance exists in a reference that isn't fundamentally contained in a grid. The grid is used to instantiate a model, read data, create the regionality and hub / arc network within that regionality, assign data to objects and more.

notably, the grid is used to coordinate internal methods in various classes to make sure that their combined actions keep the model consistent and accomplish the desired task.

Classes

`Grid`([data])	
`GridData`(data_folder[, regions_of_interest])	
`Hub`(name, region[, data])	
`Region`(name[, grid, kind, data, parent])	
`Registry`()	
`TransportationArc`(origin, destination, capacity)	

class `src.models.hydrogen.network.grid.`**`Grid`**(*data:* GridData | *None* = *None*)

`aggregate_hubs`(*hublist*, *region*)

combine all hubs in hublist into a single hub, and place them in region. Arcs that connect to any of these hubs also get aggegated into arcs that connect to the new hub and their original origin / destination that's not in hublist.

Parameters

- **`hublist`** (*`list`*) – list of hubs to aggregate
- **`region`** (`Region`) – region to place them in

`arc_generation`(*df*)

generate arcs from the arc data

Parameters
`df` (*`DataFrame`*) – arc data

`build_grid`(*vis=True*)

builds a grid fom the GridData by recursively adding regions starting at top-level region 'world'.

Parameters
`vis` (*`bool, optional`*) – if True, will generate an image of the hub-network with regional color-coding. Defaults to True.

`collapse`(*region_name*)

make a region absorb all it's sub-regions and combine all its and its childrens hubs into one

Parameters
`region_name` (*`str`*) – region to collapse

collapse_level(*level*)

collapse all regions at a specific level of depth in the regional hierarchy, with world = 0

Parameters

level (*int*) – level to collapse

combine_arcs(*arclist*, *origin*, *destination*)

combine a set of arcs into a single arc with given origin and destination

Parameters

- **arclist** (*list*) – list of arcs to aggregate
- **origin** (*str*) – new origin hub
- **destination** (*str*) – new destination hub

connect_subregions()

create an arc for all hubs in bottom-level regions to whatever hub is located in their parent region

create_arc(*origin*, *destination*, *capacity*, *cost=0.0*)

Creates and arc from origin to destination with given capacity and cost

Parameters

- **origin** (*str*) – origin hub name
- **destination** (*str*) – destination hub name
- **capacity** (*float*) – capacity of arc
- **cost** (*float, optional*) – cost of transporting 1kg H2 along arc. Defaults to 0.

create_hub(*name*, *region*, *data=None*)

creates a hub in a given region

Parameters

- **name** (*str*) – hub name
- **region** (Region) – Region hub is placed in
- **data** (*DataFrame, optional*) – dataframe of hub data to append. Defaults to None.

create_region(*name*, *parent=None*, *data=None*)

creates a region with a given name, parent region, and data

Parameters

- **name** (*str*) – name of region
- **parent** (Region*, optional*) – parent region. Defaults to None.
- **data** (*DataFrame, optional*) – region data. Defaults to None.

delete(*thing*)

deletes a hub, arc, or region

Parameters

thing (Hub*, Arc, or* Region) – thing to delete

load_hubs()

load hubs from data

recursive_region_generation(*df*, *parent*)

cycle through a region dataframe, left column to right until it hits data column, adding new regions and subregions according to how it is hierarchically structured. Future versions should implement this with a graph structure for the data instead of a dataframe, which would be more natural.

Parameters

- **df** (*DataFrame*) – hierarchically structured dataframe of regions and their data.
- **parent** (Region) – Parent region

test()

test run

visualize()

visualize the grid network using graphx

write_data()

_write data to file

src.models.hydrogen.network.grid_data

GRIDDATA CLASS

grid_data is the the data object that grids are generated from. It reads in raw data with a region grid_data is the the data object that grids are generated from. It reads in raw data with a region filter, and holds it in one structure for easy access

Classes

GridData(data_folder[, regions_of_interest])	
Path(*args, **kwargs)	PurePath subclass that can make system calls.

class src.models.hydrogen.network.grid_data.**GridData**(*data_folder: Path*, *regions_of_interest: list[str] | None = None*)

src.models.hydrogen.network.hub

HUB CLASS

class objects are individual hubs, which are fundamental units of production in the model. Hubs belong to regions, and connect to each other with transportation arcs.

Classes

Hub(name, region[, data])	

class src.models.hydrogen.network.hub.**Hub**(*name*, *region*, *data=None*) `add_inbound`(*arc*)

add an inbound arc to hub

Parameters
`arc` (*Arc*) – add an inbound arc to hub

`add_outbound`(*arc*)

add an outbound arc to hub

Parameters
`arc` (*Arc*) – arc to add

`change_region`(*new_region*)

move hub to new region

Parameters
`new_region` (`Region`) – region hub should be moved to

`cost`(*technology*, *year*)

return a cost value in terms of data fields

Parameters

- `technology` (*str*) – technology type
- `year` (*int*) – year

Returns
a cost value

Return type
float

`display_outbound`()

print all outbound arcs from hub

`get_data`(*quantity*)

fetch quantity from hub data

Parameters
`quantity` (*str*) – name of data field to fetch

Returns
quantity to be fetched

Return type
float or str

`remove_inbound`(*arc*)

remove an inbound arc from hub

Parameters
`arc` (*Arc*) – arc to remove

`remove_outbound`(*arc*)

remove an outbound arc from hub

Parameters
`arc` (*Arc*) – arc to remove

src.models.hydrogen.network.region

REGION CLASS

Class objects are regions, which have a natural tree-structure. Each region can have a parent region and child regions (subregions), a data object, and a set of hubs.

Functions

`getLogger`([name])	Return a logger with the specified name, creating it if necessary.

Classes

`Region`(name[, grid, kind, data, parent])	

class `src.models.hydrogen.network.region.`**`Region`**(*name*, *grid=None*, *kind=None*, *data=None*, *parent=None*)

`absorb_subregions`()

delete subregions, acquire their hubs and subregions

`absorb_subregions_deep`()

absorb subregions recursively so that region becomes to the deepest level in the hierarchy

`add_hub`(*hub*)

add a hub to region

Parameters
hub (`Hub`) – hub to add

`add_subregion`(*subregion*)

make a region a subregion of self

Parameters
subregion (`Region`) – new subregion

`aggregate_subregion_data`(*subregions*)

combine the data from subregions and assign it to self

Parameters
subregions (*`list`*) – list of subregions

`assigned_names = {}`

`create_subregion`(*name*, *data=None*)

create a subregion

Parameters

- **name** (*`str`*) – subregion name
- **data** (*`DataFrame, optional`*) – subregion data. Defaults to None.

delete()
  delete self, reassign hubs to parent, reassign children to parent

display_children()
  display child regions

display_hubs()
  display hubs

get_data(*quantity*)
  pull data from region data

  Parameters
  quantity (*str*) – name of data field in region data

  Returns
  value of data

  Return type
  str, float

remove_hub(*hub*)
  remove hub from region

  Parameters
  hub (Hub) – hub to remove

remove_subregion(*subregion*)
  remove a subregion from self

  Parameters
  subregion (Region) – subregion to remove

update_data(*df*)
  change region data

  Parameters
  df (*DataFrame*) – new data

update_parent(*new_parent*)
  change parent region

  Parameters
  new_parent (Region) – new parent region

src.models.hydrogen.network.registry

REGISTRY CLASS

This class is the central registry of all objects in a grid. It preserves them in dicts of object-name:object so that they can be looked up by name. it also should serve as a place to save data in different configurations for faster parsing - for example, depth is a dict that organizes regions according to their depth in the region nesting tree.

Classes

Hub(name, region[, data])

continues on next page

Table 28 – continued from previous page

`Region`(name[, grid, kind, data, parent])	
`Registry`()	
`TransportationArc`(origin, destination, capacity)	

class `src.models.hydrogen.network.registry.`**`Registry`**

`add`(*thing*)

add a thing to the registry. Thing can be Hub,Arc, or Region

Parameters
`thing` (*`Arc, Region, or Hub`*) – thing to add to registry

Returns
thing being added gets returned

Return type
Arc, *Region*, or *Hub*

`remove`(*thing*)

remove thing from registry

Parameters
`thing` (*`Arc, Hub, or Region`*) – thing to remove

`update_levels`()

update dictionary of regions by level

src.models.hydrogen.network.transportation_arc

TRANSPORTATION ARC CLASS

objects in this class represent individual transportation arcs. An arc can exist with zero capacity, so they only represent *possible* arcs.

Classes

`TransportationArc`(origin, destination, capacity)	

class `src.models.hydrogen.network.transportation_arc.`**`TransportationArc`**(*origin*, *destination*, *capacity*, *cost=0*)

`change_destination`(*new_destination*)

change the destination hub of arc

Parameters
`new_destination` (`Hub`) – new destination hub

`change_origin`(*new_origin*)

change the origin hub of arc

Parameters
new_origin (Hub) – new origin hub

disconnect()

disconnect arc from it's origin and destination

src.models.hydrogen.utilities

Modules

h2_functions	This file is a collection of functions that are used in support of the hydrogen model.

src.models.hydrogen.utilities.h2_functions

This file is a collection of functions that are used in support of the hydrogen model.

Functions

get_demand(hm, region, time)	get demand for region at time.
get_elec_price(hm, region, year)	get electricity price in region, year
get_electricity_consumption_rate(hm, tech)	the electricity consumption rate for technology type tech
get_electricty_consumption(hm, region, year)	get electricity consumption for region, year
get_gas_price(hm, region, year)	get gas price for region, year
get_production_cost(hm, hub, tech, year)	return production cost for tech at hub in year

src.models.hydrogen.utilities.h2_functions.**get_demand**(*hm: H2Model*, *region*, *time*)

get demand for region at time. If mode not standard, just increase demand by 5% per year

Parameters

- **hm** (H2Model) – model
- **region** (*str*) – region
- **time** (*int*) – year

Returns
demand

Return type
float

src.models.hydrogen.utilities.h2_functions.**get_elec_price**(*hm: H2Model*, *region*, *year*)

get electricity price in region, year

Parameters

- **hm** (H2Model) – _model
- **region** (*str*) – region
- **year** (*int*) – year

Returns
electricity price in region and year

Return type
float

`src.models.hydrogen.utilities.h2_functions.`**`get_electricity_consumption_rate`**(*hm: H2Model*, *tech*)

the electricity consumption rate for technology type tech

Parameters

- **hm** (`H2Model`) – model
- **tech** (*str*) – technology type

Returns
GWh per kg H2

Return type
float

`src.models.hydrogen.utilities.h2_functions.`**`get_electricty_consumption`**(*hm: H2Model*, *region*, *year*)

get electricity consumption for region, year

Parameters

- **hm** (`H2Model`) – model
- **region** (*str*) – region
- **year** (*int*) – year

Returns
the elecctricity consumption for a region and year in the model

Return type
float

`src.models.hydrogen.utilities.h2_functions.`**`get_gas_price`**(*hm: H2Model*, *region*, *year*)

get gas price for region, year

Parameters

- **hm** (`H2Model`) – model
- **region** (*str*) – region
- **year** (*int*) – year

Returns
gas price in region and year

Return type
float

`src.models.hydrogen.utilities.h2_functions.`**`get_production_cost`**(*hm: H2Model*, *hub*, *tech*, *year*)

return production cost for tech at hub in year

Parameters

- **hm** (`H2Model`) – model
- **hub** (*str*) – hub
- **tech** (*str*) – technology type
- **year** (*int*) – year

Returns
production cost of H2 for tech at hub in year

Return type
float

src.models.residential

Modules

preprocessor	
scripts	

src.models.residential.preprocessor

Modules

generate_inputs	This file contains the options to re-create the input files. It creates:

src.models.residential.preprocessor.generate_inputs

This file contains the options to re-create the input files. It creates:

- Load.csv: electricity demand for all model years (used in residential and electricity)
- BaseElecPrice.csv: electricity prices for initial model year (used in residential only)

Uncomment out the functions at the end of this file in the "if __name__ == '__main__'" statement in order to generate new load or base electricity prices.

Functions

base_price()	Runs the electricity model with base price configuration settings and then merges the electricity prices and temporal crosswalk data produced from the run to generate base year electricity prices.
main([settings])	Runs model as defined in settings

Classes

Path(*args, **kwargs)	PurePath subclass that can make system calls.

src.models.residential.preprocessor.generate_inputs.**base_price**()

Runs the electricity model with base price configuration settings and then merges the electricity prices and temporal crosswalk data produced from the run to generate base year electricity prices.

Returns
dataframe that contains base year electricity prices for all regions/hours

Return type
pandas.core.frame.DataFrame

src.models.residential.scripts

Modules

residential	Residential Model.

src.models.residential.scripts.residential

Residential Model. This file contains the residentialModule class which contains a representation of residential electricity prices and demands.

Functions

`getLogger`([name])	Return a logger with the specified name, creating it if necessary.
run_residential(settings)	This runs the residential model in stand-alone mode.
`scale_load`(data_root)	Reads in BaseLoad.csv (load for all regions/hours for first year) and LoadScalar.csv (a multiplier for all model years).
`scale_load_with_enduses`(data_root)	Reads in BaseLoad.csv (load for all regions/hours for first year), EnduseBaseShares.csv (the shares of demand for each enduse in the base year) and EnduseScalar.csv (a multiplier for all model years by enduse category).

Classes

`Config_settings`(config_path[, args, test])	Generates the model settings that are used to solve.
`Path`(*args, **kwargs)	PurePath subclass that can make system calls.
residentialModule([settings, loadFile, ...])	This contains the Residential model and its associated functions.

class `src.models.residential.scripts.residential.`**`residentialModule`**(*settings: Config_settings | None = None*, *loadFile: str | None = None*, *load_df: DataFrame | None = None*, *calibrate: bool | None = False*)

This contains the Residential model and its associated functions. Once an object is instantiated, it can calculate new Load values for updated prices. It can also calculate estimated changes to the Load if one of the input variables is changed by a specified percent. The model will be created in a symbolic form to be easily manipulated, and then values can be filled in for calculations.

`baseYear = 0`

`complex_step_sensitivity`(*prices*, *change_var*, *percent*)

This estimates how much the output Load will change due to a change in one of the input variables. It can calculate these values for changes in price, price elasticity, income, income elasticity, or long term trend. The Load calculation requires input prices, so this function requires that as well for the base output Load.

Then, an estimate for Load is calculated for the case where the named 'change_var' is changed by 'percent' %.

Parameters

- **prices** (*dataframe or Pyomo Indexed Parameter*) – Price values used to calculate the Load value
- **change_var** (*string*) –

  Name of variable of interest for sensitivity. This can be:
  'income', 'i_elas', 'price', 'p_elas', 'trendGR'
- **percent** (*float*) – A value 0 - 100 for the percent that the variable of interest can change.

Returns
Indexed values for the calculated Load at the given prices, the Load if the variable of interest is increased by 'percent'%, and the Load if the variable of interest is decreased by 'percent'%

Return type
dataframe

demandF(*price*, *load*, *year*, *basePrice=1*, *p_elas=-0.1*, *baseYear=None*, *baseIncome=1*, *income=1*, *i_elas=1*, *trend=0*, *priceIndex=1*, *incomeIndex=1*, *p_lag=1*, *i_lag=1*)

The demand function. Wraps the sympy demand function with some defaults

Parameters

- **price** (*_type_*) – _description_
- **load** (*_type_*) – _description_
- **year** (*_type_*) – _description_
- **basePrice** (*int, optional*) – _description_, by default 1
- **p_elas** (*float, optional*) – _description_, by default -0.10
- **baseYear** (*_type_, optional*) – _description_, by default None
- **baseIncome** (*int, optional*) – _description_, by default 1
- **income** (*int, optional*) – _description_, by default 1
- **i_elas** (*int, optional*) – _description_, by default 1
- **trend** (*int, optional*) – _description_, by default 0
- **priceIndex** (*int, optional*) – _description_, by default 1
- **incomeIndex** (*int, optional*) – _description_, by default 1
- **p_lag** (*int, optional*) – _description_, by default 1
- **i_lag** (*int, optional*) – _description_, by default 1

Returns
description

Return type
type

hr_map = Empty DataFrame Columns: [] Index: []

loads = {}

make_block(*prices*, *pricesindex*)

Updates the value of 'Load' based on the new prices given. The new prices are fed into the equations from the residential model. The new calculated Loads are used to constrain 'Load' in pyomo blocks.

Parameters

- **prices** (*pyo.Param*) – Pyomo Parameter of newly updated prices
- **pricesindex** (*pyo.Set*) – Pyomo Set of indexes that matches the prices given

Returns

Block containing constraints that set 'Load' variable equal to the updated load values

Return type

pyo.Block

prices = {}

sensitivity(*prices*, *change_var*, *percent*)

This estimates how much the output Load will change due to a change in one of the input variables. It can calculate these values for changes in price, price elasticity, income, income elasticity, or long term trend. The Load calculation requires input prices, so this function requires that as well for the base output Load. Then, an estimate for Load is calculated for the case where the named 'change_var' is changed by 'percent' %.

Parameters

- **prices** (*dataframe or Pyomo Indexed Parameter*) – Price values used to calculate the Load value
- **change_var** (*string*) –

  Name of variable of interest for sensitivity. This can be:
  'income', 'i_elas', 'price', 'p_elas', 'trendGR'
- **percent** (*float*) – A value 0 - 100 for the percent that the variable of interest can change.

Returns

Indexed values for the calculated Load at the given prices, the Load if the variable of interest is increased by 'percent'%, and the Load if the variable of interest is decreased by 'percent'%

Return type

dataframe

update_load(*p*)

Takes in Dual pyomo Parameters or dataframes to update Load values

Parameters

p (*pyo.Param*) – Pyomo Parameter or dataframe of newly updated prices from Duals

Returns

Load values indexed by region, year, and hour

Return type

pandas DataFrame

view_output_load(*values: DataFrame*, *regions: list[int] = [1]*, *years: list[int] = [2023]*)

This is used to display the updated Load values after calculation. It will create a graph for each region and year combination.

Parameters

- **values** (*pd.DataFrame*) – The Load values calculated in update_load

- **regions** (*list[int], optional*) – The regions to be displayed
- **years** (*list[int], optional*) – The years to be displayed

view_sensitivity(*values: DataFrame*, *regions: list[int] = [1]*, *years: list[int] = [2023]*)

This is used by the sensitivity method to display graphs of the calculated values

Parameters

- **values** (*pd.DataFrame*) – indexed values for the Load, upper change, and lower change
- **regions** (*list[int], optional*) – regions to be graphed
- **years** (*list[int], optional*) – years to be graphed

src.models.residential.scripts.residential.**run_residential**(*settings: Config_settings*)

This runs the residential model in stand-alone mode. It can run update_load to calculate new Load values based on prices, or it can calculate the new Load value along with estimates for the Load if one of the input variables changes.

Parameters

settings (Config_settings) – information given from run_config to set several values

1.4.2 src

This directory contains the subdirectory for the integrator module and the subdirectories for the sectoral modules. The integrator module contains code for the different solve methods (standalone, iterative, or unified). The sectoral modules currently represented in this prototype include the electricity model, hydrogen model, and residential model. For more details on each of these, please see the READMEs located within each subdirectory.

Modules

common
integrator
models
sensitivity

src.common

Modules

config_setup	This file contains Config_settings class.
model	Establish a base model class for the sectoral modules to inherit.
utilities	A gathering of utility functions for dealing with model interconnectivity

src.common.config_setup

This file contains Config_settings class. It establishes the main settings used when running the model. It takes these settings from the run_config.toml file. It has universal configurations (e.g., configs that cut across modules and/or solve options) and module specific configs.

Functions

`create_temporal_mapping`(sw_temporal)	Combines the input mapping files within the electricity model to create a master temporal mapping dataframe.
`make_dir`(dir_name)	generates an output directory to write model results, output directory is the date/time at the time this function executes.

Classes

`Config_settings`(config_path[, args, test])	Generates the model settings that are used to solve.
`Path`(*args, **kwargs)	PurePath subclass that can make system calls.
`datetime`(year, month, day[, hour[, minute[, ...)	The year, month and day arguments are required.

`class src.common.config_setup.`**`Config_settings`**(*config_path: Path*, *args: Namespace | None = None*, *test=False*)

Generates the model settings that are used to solve. Settings include:

- Iterative Solve Config Settings
- Spatial Config Settings
- Temporal Config Settings
- Electricity Config Settings
- Other

`_additional_year_settings`(*name*, *value*)

Checks year related settings to see if values are within expected ranges and updates other settings linked to years if years is changed.

Parameters

- **name** (*str*) – attribute name
- **value** (*_type_*) – attribute value

Raises

TypeError – Error

`_check_elec_expansion_settings`(*name*, *value*)

Checks that switches for reserve margin and learning are on only if expansion is on.

Parameters

- **name** (*str*) – attribute name
- **value** (*_type_*) – attribute value

Raises
`TypeError` – Error

`_check_int`(*name*, *value*)

Checks if attribute is an integer

Parameters

- `name` (*str*) – attribute name
- `value` (*_type_*) – attribute value

Raises
`TypeError` – Error

`_check_regions`(*name*, *value*)

Checks to see if region is between the current default values of 1 and 25.

Parameters

- `name` (*str*) – attribute name
- `value` (*_type_*) – attribute value

Raises
`TypeError` – Error

`_check_res_settings`(*name*, *value*)

Checks if view year or region settings are subsets of year or region

Parameters

- `name` (*str*) – attribute name
- `value` (*_type_*) – attribute value

Raises
`TypeError` – Error

`_check_true_false`(*name*, *value*)

Checks if attribute is either true or false

Parameters

- `name` (*str*) – attribute name
- `value` (*_type_*) – attribute value

Raises
`TypeError` – Error

`_check_zero_one`(*name*, *value*)

Checks if attribute is either zero or one

Parameters

- `name` (*str*) – attribute name
- `value` (*_type_*) – attribute value

Raises
`TypeError` – Error

`_has_all_attributes`(*attrs: set*)

Determines if all attributes within the set exist or not

Parameters
`attrs` (*set*) – set of setting attributes

Returns
True or False

Return type
bool

src.common.model

Establish a base model class for the sectoral modules to inherit.

Functions

`getLogger`([name])	Return a logger with the specified name, creating it if necessary.

Classes

Model(*args, **kwds)	This is the base model class for the models.
`defaultdict`	defaultdict(default_factory=None, /, [...]) --> dict with default factory

class `src.common.model.`**Model**(**args*, ***kwds*)

This is the base model class for the models.

This class contains methods for declaring pyomo components, extracting duals, and decorating expressions. The model class methods and attributes provide functionality for keeping track of index labels and ordering for all pyomo components; this is essential for integration tasks without the use of hard-coded indices and allows for easy post-processing tasks.

class ConstraintExpression(**args*, ***kwargs*)

Constraint Expression decorator that works the same as pyomo decorators, while keeping column dictionary updated for any indexed parameters given.

class DefaultDecorator(*model*, **args*, ***kwargs*)

Default decorator class that handles assignment of model scope/pointer in order to use pyomo-style parameter and constraint decorators.

Upon initialization, the decorator handles model assignment at class level to ensure inheriting classes have access to the models within local scope.

classmethod assign_model(*model*)

Class-method that assigns a model instance to DefaultDecorator

Parameters
`model` (*pyo.ConcreteModel*) – A pyo model instance

class ParameterExpression(**args*, ***kwargs*)

Parameter Expression decorator that works the same as pyomo decorators, while keeping column dictionary updated for any indexed parameters given.

`_active`

`_declare_set_with_dict`(*sname: str*, *sdata: Dict | DefaultDict*, *scols: MutableSequence*, *return_set: bool | None = False*, *switch: bool | None = True*, *create_indexed_set: bool | None = True*, *use_values: bool | None = False*) → Set

Declares a pyomo Set object named 'sname' using input index values and labels.

Function takes a dictionary argument and creates pyomo set object from keys, values, or both.

If an indexed set is desired, set create_indexed_set to True; the function will create an indexed set with its own indices set as keys. Otherwise, an Ordered Scalar Set will be created, either from the keys or the values of 'sdata' depending on the value for 'use_values' (False for keys, True for values).

Names for the indices handled by scols; user must provide.

Parameters

- **sname** (*str*) – Name of set
- **sdata** (*Dict*) – Data object that contains set values
- **scols** (*Sequence | None, optional*) – List of column names corresponding to index labels and position, by default None
- **return_set** (*bool | None, optional*) – Return the set rather than assign within function, by default False
- **switch** (*bool | None, optional*) – Return None if False, by default True
- **create_indexed_set** (*bool | None, optional*) – Indicator for whether output set should include values as well as new index (IndexedSets), by default True
- **use_values** (*bool | None, optional*) – If create_indexed_set is False, use the values of sdata rather than keys for pyo Set members, by default False

Returns
Pyomo Set Object

Return type
pyo.Set

`_declare_set_with_iterable`(*sname: str*, *sdata: Sequence | Set | array*, *scols: Sequence[str] | None = None*, *return_set: bool | None = False*, *switch: bool | None = True*) → Set

Declares a pyomo Set object named 'sname' using input index values and labels.

Function can take iterable objects such as tuples, lists, etc as data inputs. Note that if the dimension of the index is larger than 1, user needs to provide a list of names for each set dimension.

Parameters

- **sname** (*str*) – Name of set
- **sdata** (*Sequence*) – Data object that contains set values
- **scols** (*Sequence | None, optional*) – List of column names corresponding to index labels and position, by default None
- **return_set** (*bool | None, optional*) – Return the set rather than assign within function, by default False
- **switch** (*bool | None, optional*) – Return None if False, by default True

Returns
Pyomo Set Object

Return type
pyo.Set

`_declare_set_with_pandas`(*sname: str*, *sdata: DataFrame | Series*, *return_set: bool | None = False*, *switch: bool | None = True*, *use_columns: bool | None = False*)

Declares a pyomo Set object named 'sname' using input index values and labels from a Pandas object.

Function assumes that the index values are the desired data to construct set object. User can specify whether to create set with column values instead

Parameters

- **sname** (*str*) – Name of set
- **sdata** (*MutableSequence | dict*) – Data object that contains set indices
- **return_set** (*bool | None, optional*) – Return the set rather than assign within function, by default False
- **switch** (*bool | None, optional*) – Return None if False, by default True
- **use_columns** (*bool | None, optional*) – Use columns as indices for pyo set rather than row index, by default False

Returns
Pyomo Set Object

Return type
pyo.Set

`classmethod build()`

Default build command; class-level build to create and return an instance of Model.

This will work for any class inheriting the method, but it is recommended to replace this with model-specific build instructions if this functionality is desired.

Returns
Instance of Model object

Return type
Object

`declare_ordered_time_set`(*sname: str*, **sets: Set*, *return_set: bool | None = False*)

Unnest the time sets into a single, unnested ordered, synchronous time set, an IndexedSet object keyed by the values in the time set, and an IndexedSet object keyed by the combined, original input sets.

These three set outputs are directly set as attributes of the model instance:

sname: (1,) , (2,), … ,(N) sname_time_to_index: (1,):[set1, set2, set3] , (2,):[set1, set2, set3] sname_index_to_time: (set1, set2, set3): [1] , (set1, set2, set3): [2]

In summary, this function creates three sets, creating a unique, ordered set from input sets with the assumption that they are given to the function in hierarchical order. For example, for a desired time set that orders Year, Month, Day values, the args for the function should be provided as:

m.Year, m.Month, m.Day

Pyomo set products are used to unpack and create new set values that are ordered by the hierarchy provided:

(year1, month1, day1) , (year1, month1, day2) , … , (year2, month1, day1) , … (yearY, monthM, dayD)

Parameters

- **sname** (*str*) – Desired root name for the new sets

- **sets** (*pyo.Set*) – A series of unnamed arguments assumed to contain pyo.Set in order of temporal hierarchy
- **return_set** (*bool | None, optional*) – Return the set rather than assign within function, by default False

Returns
No return object; all sets assigned to model internally

Return type
None

Raises
ValueError – "No sets provided in args; provide pyo.Set objects to use this function"

declare_param(*pname: str*, *p_set: Set | None*, *data: dict | DataFrame | Series | int | float*, *return_param: bool | None = False*, *default: int | None = 0*, *mutable: bool | None = False*) → Param

Declares a pyo Parameter component named 'pname' with the input data and index set.

Unpacks column dictionary of index set for param instance and creates pyo.Param; either assigns the value internally or returns the object based on return_param.

Parameters

- **pname** (*str*) – Desired name of new pyo.Param instance
- **p_set** (*pyo.Set*) – Pyomo Set instance to index new Param
- **data** (*dict | pd.DataFrame | pd.Series*) – Data to initialize Param instance
- **return_param** (*bool | None, optional*) – Return the param after function call rather than assign to self, by default False
- **default** (*int | None, optional*) – pyo.Param keyword argument, by default 0
- **mutable** (*bool | None, optional*) – pyo.Param keyword argument, by default False

Returns
A pyomo Parameter instance

Return type
pyo.Param

Raises
ValueError – Raises error if input data not in format supported by function

declare_set(*sname: str*, *sdata: MutableSequence | DataFrame | Series | Dict*, *scols: MutableSequence | None = None*, *return_set: bool | None = False*, *switch: bool | None = True*, *create_indexed_set: bool | None = True*, *use_values: bool | None = False*, *use_columns: bool | None = False*)

Declares a pyomo Set object named 'sname' using input index values and labels.

Function handles input values and calls appropriate declare_set methods based on data type of sdata

Parameters

- **sname** (*str*) – Name of set
- **sdata** (*Dict*) – Data object that contains set values
- **scols** (*Sequence | None, optional*) – List of column names corresponding to index labels and position, by default None
- **return_set** (*bool | None, optional*) – Return the set rather than assign within function, by default False

- **switch** (*bool | None, optional*) – Return None if False, by default True
- **create_indexed_set** (*bool | None, optional*) – If dict, indicator for whether output set should include values as well as new index (IndexedSets), by default True
- **use_values** (*bool | None, optional*) – If dict and create_indexed_set is False, use the values of sdata rather than keys for pyo Set members, by default False
- **use_columns** (*bool | None, optional*) – If Pandas, use columns as indices for pyo set rather than row index, by default False

Returns
Pyomo Set Object

Return type
pyo.Set

declare_set_with_sets(*sname: str*, **sets: Set*, *return_set: bool | None = False*, *switch: bool | None = True*) → Set

Declares a new set object using input sets as arguments.

Function creates a set product with set arguments to create a new set. This is how pyomo handles set creation with multiple existing sets as arguments.

However, this function finds each pyomo set in column dictionary and unpacks the names, so that the new set can be logged in the column dictionary too.

Parameters

- **sname** (*str*) – Desired name of new set
- ***sets** (*tuple of pyo.Set*) – Unnamed arguments assumed to be pyomo sets
- **return_set** (*bool | None, optional*) – Return the set rather than assign within function, by default False
- **switch** (*bool | None, optional*) – Return None if False, by default True

Returns
Pyomo Set Object

Return type
pyo.Set

declare_shifted_time_set(*sname: str*, *shift_size: int*, *shift_type: Literal['lag', 'lead']*, **sets: Set*, *return_set: bool | None = False*, *shift_sets: List | None = None*)

A generalize shifting function that creates sets compatible with leads or lags in pyomo components.

For example, with a storage constraint where the current value is contrained to be equal to the value of storage in the previous period:

model.storage[t] == model.storage[t-1] + …

The indexing set must be consistent with the storage variable, but not include elements that are undefined for this constraint. In this example, the set containing values for t must not include t = 1 (e.g. the lagged value must be defined). This function creates a shifted time set by removing values from the input sets to comply with the lags or leads.

Function inputs require a shift size (in the example above, this would be 1), a shift type (lead or lag), and the sets used to construct the new, shifted set (model.timestep). If a lag or lead is required on a single dimension of the new set, the 'shift_sets' argument can include a list of pyo.set names (included in the arguments) to shift by the other args.

For example…

model.storage[hub, season] == model.storage[hub, season - 1]

In this case, season = 1 is always invalid due to the lag; so index (1, 2) or the value for hub = 1 and season = 2 is valid, but (2, 1) remains an invalid argument as there is no season = 0. A new set composed of hub and season, with shift_sets = ["season"] and sets = model.hub, model.season, is created to lag on one index value while leaving others unchanged.

Default is to create set product of all input sets and lag/lead w/ resulting elements.

Parameters

- **sname** (*str*) – Desired name for new set
- **shift_size** (*int*) – Size of shift in set
- **shift_type** (*str in ["lag", "lead"]*) – Type of shift (e.g. t-1 or t+1)
- ***sets** (*Unnamed arguments*) – A series of unnamed arguments assumed to contain pyo.Set in order of temporal hierarchy
- **return_set** (*bool | None, optional*) – Return the set rather than assign within function, by default False
- **shift_sets** (*List | None, optional*) – List of pyo.Set (by name) in *sets to shift, by default None

Returns
A pyomo Set

Return type
pyo.Set

Raises

- **ValueError** – Shift sets don't align with *sets names
- **ValueError** – Type argument is neither lead nor lag

declare_var(*vname: str, v_set: Set, return_var: bool | None = False, within: Literal['NonNegativeReals', 'Binary', 'Reals', 'NonNegativeIntegers'] | None = 'NonNegativeReals', bound: tuple | None = (0, 1000000000), switch: bool | None = True*) → Var

Declares a pyo Variable component named 'vname' with index set 'v_set'.

Creates variable indexed by previously defined pyo Set instance 'v_set' and assigns to self; function will return the component if return_var is set to True. Other keywords passed to pyo.Var are within and bound.

Parameters

- **vname** (*str*) – Desired name of new pyo Variable
- **v_set** (*pyo.Set*) – Index set for new pyo Variable
- **return_var** (*bool | None, optional*) – Return component rather than assign internally, by default False
- **within** (*str in ["NonNegativeReals", "Binary", "Reals", "NonNegativeIntegers"] | None, optional*) – pyo.Var keyword argument, by default "NonNegativeReals"
- **bound** (*tuple | None, optional*) – pyo.Var keyword argument, by default (0, 1000000000)

Returns
A pyomo Variable instance

Return type
pyo.Var

get_duals(*component_name: str*) → defaultdict

Extract duals from a solved model instance

Parameters
component_name (*str*) – Name of constraint

Returns
Dual values w/ index values

Return type
defaultdict

populate_sets_rule(*sname*, *set_base_name=None*, *set_base2=None*) → Set

Generic function to create a new re-indexed set for a pyomo ConcreteModel instance which should speed up build time. Must pass non-empty (either) set_base_name or set_base2

Parameters

- **m1** (*pyo.ConcreteModel*) – pyomo model instance
- **sname** (*str*) – name of input pyomo set to base reindexing
- **set_base_name** (*str, optional*) – the name of the set to be the base of the reindexing, if left blank, uses set_base2, by default ''
- **set_base2** (*list, optional*) – the list of names of set columns to be the base of the reindexing, if left blank, should use set_base_name, by default [] these will form the index set of the indexed set structure

Returns
reindexed set to be added to model

Return type
pyomo set

reorganize_index_set(*sname: str*, *new_sname: str*, *return_set: bool | None = False*, *create_indexed_set: bool | None = False*, *reorg_set_cols: List[str] | None = None*, *reorg_set_sname: str | None = None*)

Creates new pyomo sets based on an input set and a desired set of indices for an output set. User should provide either names of columns desired for reorganized output set OR the name of a set that mirrors the desired indexing.

For instance, an input set indexed by (yr, region, month, day) can be reorganized into an output set:

(yr, region):[(month,day), (month,day), (month,day)]

when ["yr", "region"] is provided for reorg_set_cols.

If only the set keys are desired, without creating an indexed set object as illustrated above, the user can set 'create_indexed_set' to false. If true, the output is a pyo.IndexedSet, with each element of the IndexedSet containing the values of other indices

Parameters

- **sname** (*str*) – Name of input set
- **new_sname** (*str*) – Name of output set or IndexedSet
- **create_indexed_set** (*bool | None, optional*) – Indicator for whether output set should include values as well as new index (IndexedSets), by default False

- **return_set** (*bool | None, optional*) – Indicator for whether to return the constructed set
- **reorg_set_cols** (*List[str] | None, optional*) – List of columns to index output set contained in 'sname', by default None
- **reorg_set_sname** (*str | None, optional*) – Name of set to use for identifying output set indices, by default None

Returns
Pyomo Set or IndexedSet object reorganized based on input set

Return type
Pyo.Set

Raises

- **ValueError** – Populate function is either-or for reorg_set_cols and reorg_set_sname, received both
- **ValueError** – Populate function is either-or for reorg_set_cols and reorg_set_sname, received neither
- **ValueError** – Elements missing from input set desired in new set

unpack_set_arguments(*sname: str*, *sets: Tuple[Set]*, *return_set_product: bool* | *None = True*) → Set

Handles unnamed pyo.Set arguments for multiple declaration functions.

For an arbitrarily large number of set inputs, this function unpacks the names for each set stored in the column dictionary, creates a new list of the index labels and ordering, and then provides the pyo.Set product result as an output.

Parameters

- **sname** (*str*) – Name of new set
- **sets** (*tuple of pyo.Set*) – Tuple of pyo.Set arguments to be used to generate new set
- **return_set_product** (*bool*) – If True, return the unpacked set product

Returns
new_set – Set product result from input sets, by order of sets arguments

Return type
pyo.Set

src.common.utilities

A gathering of utility functions for dealing with model interconnectivity

Functions

getLogger([name])	Return a logger with the specified name, creating it if necessary.
get_args()	Parses args
make_dir(dir_name)	generates an output directory to write model results, output directory is the date/time at the time this function executes.

continues on next page

Table 45 – continued from previous page

`scale_load`(data_root)	Reads in BaseLoad.csv (load for all regions/hours for first year) and LoadScalar.csv (a multiplier for all model years).
`scale_load_with_enduses`(data_root)	Reads in BaseLoad.csv (load for all regions/hours for first year), EnduseBaseShares.csv (the shares of demand for each enduse in the base year) and EnduseScalar.csv (a multiplier for all model years by enduse category).
`setup_logger`(settings)	initiates logging, sets up logger in the output directory specified

Classes

`Path`(*args, **kwargs)	PurePath subclass that can make system calls.

`src.common.utilities.get_args()`

Parses args

Returns
args – Contains arguments pass to main.py executable

Return type
Namespace

`src.common.utilities.make_dir(`*dir_name*`)`

generates an output directory to write model results, output directory is the date/time at the time this function executes. It includes subdirs for vars, params, constraints.

Returns
the name of the output directory

Return type
string

`src.common.utilities.scale_load(`*data_root*`)`

Reads in BaseLoad.csv (load for all regions/hours for first year) and LoadScalar.csv (a multiplier for all model years). Merges the data and multiplies the load by the scalar to generate new load estimates for all model years.

Returns
dataframe that contains load for all regions/years/hours

Return type
pandas.core.frame.DataFrame

`src.common.utilities.scale_load_with_enduses(`*data_root*`)`

Reads in BaseLoad.csv (load for all regions/hours for first year), EnduseBaseShares.csv (the shares of demand for each enduse in the base year) and EnduseScalar.csv (a multiplier for all model years by enduse category). Merges the data and multiplies the load by the adjusted enduse scalar and then sums up to new load estimates for all model years.

Returns
dataframe that contains load for all regions/years/hours

Return type
pandas.core.frame.DataFrame

`src.common.utilities.setup_logger(settings)`

initiates logging, sets up logger in the output directory specified

Parameters

output_dir (*path*) – output directory path

src.integrator

Modules

gaussseidel	Iteratively solve 2 models with GS methodology
progress_plot	A plotter that can be used for combined solves
runner	A gathering of functions for running models solo
unified	Unifying the solve of both H2 and Elec and Res
utilities	A gathering of utility functions for dealing with model interconnectivity

src.integrator.gaussseidel

Iteratively solve 2 models with GS methodology

see README for process explanation

Functions

convert_elec_price_to_lut(prices)	convert electricity prices to dictionary, look up table
convert_h2_price_records(records)	simple coversion from list of records to a dictionary LUT repeat entries should not occur and will generate an error
getLogger([name])	Return a logger with the specified name, creating it if necessary.
get_elec_price(instance[, block])	pulls hourly electricity prices from completed PowerModel and de-weights them.
init_old_cap(instance)	initialize capacity for 0th iteration
namedtuple(typename, field_names, *[, ...])	Returns a new subclass of tuple with named fields.
plot_it(OUTPUT_ROOT[, h2_price_records, ...])	cheap plotter of iterative progress
poll_h2_demand(model)	Get the hydrogen demand by rep_year and region
poll_h2_prices_from_elec(model, tech, regions)	poll the step-1 H2 price currently in the model for region/year, averaged over any steps
poll_hydrogen_price(model[, block])	Retrieve the price of H2 from the H2 model
regional_annual_prices(m[, block])	pulls all regional annual weighted electricity prices
run_elec_model(settings[, solve])	build electricity model (and solve if solve=True) after passing in settings
run_gs(settings)	Start the iterative GS process
select_solver(instance)	Select solver based on learning method
set_new_cap(instance)	calculate new capacity after solve iteration
simple_solve(m)	a simple solve routine
simple_solve_no_opt(m, opt)	Solve concrete model using solver factory object
update_cost(instance)	update capital cost based on new capacity learning
update_h2_prices(model, h2_prices)	Update the H2 prices held in the model

Classes

`EI`(region, year, hour)	(region, year, hour)
`Path`(*args, **kwargs)	PurePath subclass that can make system calls.
`residentialModule`([settings, loadFile, ...])	This contains the Residential model and its associated functions.

`src.integrator.gaussseidel.`**`run_gs`**(*settings*)

Start the iterative GS process

Parameters

`settings` (*obj*) – Config_settings object that holds module choices and settings

src.integrator.progress_plot

A plotter that can be used for combined solves

Functions

`plot_it`(OUTPUT_ROOT[, h2_price_records, ...])	cheap plotter of iterative progress
`plot_price_distro`(OUTPUT_ROOT, price_records)	cheap/quick analyisis and plot of the price records

Classes

`Path`(*args, **kwargs)	PurePath subclass that can make system calls.
`datetime`(year, month, day[, hour[, minute[, ...)	The year, month and day arguments are required.

`src.integrator.progress_plot.`**`plot_it`**(*OUTPUT_ROOT*, *h2_price_records=[]*, *elec_price_records=[]*, *h2_obj_records=[]*, *elec_obj_records=[]*, *h2_demand_records=[]*, *elec_demand_records=[]*, *load_records=[]*, *elec_price_to_res_records=[]*)

cheap plotter of iterative progress

`src.integrator.progress_plot.`**`plot_price_distro`**(*OUTPUT_ROOT*, *price_records: list[float]*)

cheap/quick analyisis and plot of the price records

src.integrator.runner

A gathering of functions for running models solo

Functions

`getLogger`([name])	Return a logger with the specified name, creating it if necessary.

continues on next page

Table 52 – continued from previous page

plot_price_distro(OUTPUT_ROOT, price_records)	cheap/quick analyisis and plot of the price records
run_elec_model(settings[, solve])	build electricity model (and solve if solve=True) after passing in settings
run_elec_solo([settings])	Runs electricity model by itself as defined in settings
run_h2_solo([settings])	Runs hydrogen model by itself as defined in settings
run_hydrogen_model(settings)	run hydrogen model in standalone
run_residential(settings)	This runs the residential model in stand-alone mode.
run_residential_solo([settings])	Runs residential model by itself as defined in settings
run_standalone(settings)	Runs standalone methods based on settings selections; running 1 or more modules
value(obj[, exception])	A utility function that returns the value of a Pyomo object or expression.

Classes

Config_settings(config_path[, args, test])	Generates the model settings that are used to solve.
Path(*args, **kwargs)	PurePath subclass that can make system calls.

src.integrator.runner.**run_elec_solo**(*settings: Config_settings | None = None*)

Runs electricity model by itself as defined in settings

Parameters
settings (Config_settings) – Contains configuration settings for which regions, years, and switches to run

src.integrator.runner.**run_h2_solo**(*settings: Config_settings | None = None*)

Runs hydrogen model by itself as defined in settings

Parameters
settings (Config_settings) – Contains configuration settings for which regions and years to run

src.integrator.runner.**run_residential_solo**(*settings: Config_settings | None = None*)

Runs residential model by itself as defined in settings

Parameters
settings (Config_settings) – Contains configuration settings for which regions and years to run

src.integrator.runner.**run_standalone**(*settings: Config_settings*)

Runs standalone methods based on settings selections; running 1 or more modules

Parameters
settings (Config_settings) – Instance of config_settings containing run options, mode and settings

src.integrator.unified

Unifying the solve of both H2 and Elec and Res

Dev Notes:

1. The "annual demand" constraint that is present and INACTIVE is omitted here for clarity. It may likely be needed - in some form - at a later time. Recall, the key linkages to share the electrical demand primary variable are:

- an annual level demand constraint
- an accurate price-pulling function that can consider weighted duals from both constraints

2. This model has a 2-solve update cycle as commented on near the termination check

- elec_prices gleaned from cycle[n] results -> solve cycle[n+1]
- new_load gleaned from cycle[n+1] results -> solve cycle[n+2]
- elec_pices gleaned from cycle[n+2]

Functions

`convert_elec_price_to_lut`(prices)	convert electricity prices to dictionary, look up table
`convert_h2_price_records`(records)	simple coversion from list of records to a dictionary LUT repeat entries should not occur and will generate an error
`getLogger`([name])	Return a logger with the specified name, creating it if necessary.
`get_elec_price`(instance[, block])	pulls hourly electricity prices from completed PowerModel and de-weights them.
`init_old_cap`(instance)	initialize capacity for 0th iteration
`poll_h2_demand`(model)	Get the hydrogen demand by rep_year and region
`poll_hydrogen_price`(model[, block])	Retrieve the price of H2 from the H2 model
`regional_annual_prices`(m[, block])	pulls all regional annual weighted electricity prices
`run_elec_model`(settings[, solve])	build electricity model (and solve if solve=True) after passing in settings
`run_unified`(settings)	Runs unified solve method based on
`select_solver`(instance)	Select solver based on learning method
`set_new_cap`(instance)	calculate new capacity after solve iteration
`simple_solve`(m)	a simple solve routine
`simple_solve_no_opt`(m, opt)	Solve concrete model using solver factory object
`update_cost`(instance)	update capital cost based on new capacity learning
`update_h2_prices`(model, h2_prices)	Update the H2 prices held in the model

Classes

`Config_settings`(config_path[, args, test])	Generates the model settings that are used to solve.
`EI`(region, year, hour)	(region, year, hour)
`HI`(region, year)	(region, year)
`defaultdict`	defaultdict(default_factory=None, /, [...]) --> dict with default factory
`deque`	deque([iterable[, maxlen]]) --> deque object
`residentialModule`([settings, loadFile, ...])	This contains the Residential model and its associated functions.

`src.integrator.unified.`**`run_unified`**(*settings:* Config_settings)

Runs unified solve method based on

Parameters

`settings` (`Config_settings`) – Instance of config_settings containing run options, mode and settings

src.integrator.utilities

A gathering of utility functions for dealing with model interconnectivity

Dev Note: At some review point, some decisions may move these back & forth with parent models after it is decided if it is a utility job to do . . . . or a class method.

Additionally, there is probably some renaming due here for consistency

Module Attributes

EI(region, year, hour)	(region, year, hour)
HI(region, year)	(region, year)

Functions

convert_elec_price_to_lut(prices)	convert electricity prices to dictionary, look up table
convert_h2_price_records(records)	simple coversion from list of records to a dictionary LUT repeat entries should not occur and will generate an error
create_temporal_mapping(sw_temporal)	Combines the input mapping files within the electricity model to create a master temporal mapping dataframe.
getLogger([name])	Return a logger with the specified name, creating it if necessary.
get_annual_wt_avg(elec_price)	takes annual weighted average of hourly electricity prices
get_elec_price(instance[, block])	pulls hourly electricity prices from completed Power-Model and de-weights them.
namedtuple(typename, field_names, *[, ...])	Returns a new subclass of tuple with named fields.
poll_h2_demand(model)	Get the hydrogen demand by rep_year and region
poll_h2_prices_from_elec(model, tech, regions)	poll the step-1 H2 price currently in the model for region/year, averaged over any steps
poll_hydrogen_price(model[, block])	Retrieve the price of H2 from the H2 model
poll_year_avg_elec_price(price_list)	retrieve a REPRESENTATIVE price at the annual level from a listing of prices
regional_annual_prices(m[, block])	pulls all regional annual weighted electricity prices
select_solver(instance)	Select solver based on learning method
simple_solve(m)	a simple solve routine
simple_solve_no_opt(m, opt)	Solve concrete model using solver factory object
update_elec_demand(self, elec_demand)	Update the external electical demand parameter with demands from the H2 model
update_h2_prices(model, h2_prices)	Update the H2 prices held in the model
value(obj[, exception])	A utility function that returns the value of a Pyomo object or expression.

Classes

ConcreteModel(*args, **kwds)	A concrete optimization model that does not defer construction of components.
EI(region, year, hour)	(region, year, hour)
HI(region, year)	(region, year)
Path(*args, **kwargs)	PurePath subclass that can make system calls.

continues on next page

Table 58 – continued from previous page

`defaultdict`	defaultdict(default_factory=None, /, [...]) --> dict with default factory

class `src.integrator.utilities.`**`EI`**(*region*, *year*, *hour*)

(region, year, hour)

`_asdict`()

Return a new dict which maps field names to their values.

`_field_defaults = {}`

`_fields = ('region', 'year', 'hour')`

classmethod **`_make`**(*iterable*)

Make a new EI object from a sequence or iterable

`_replace`(***kwds*)

Return a new EI object replacing specified fields with new values

`hour`

Alias for field number 2

`region`

Alias for field number 0

`year`

Alias for field number 1

class `src.integrator.utilities.`**`HI`**(*region*, *year*)

(region, year)

`_asdict`()

Return a new dict which maps field names to their values.

`_field_defaults = {}`

`_fields = ('region', 'year')`

classmethod **`_make`**(*iterable*)

Make a new HI object from a sequence or iterable

`_replace`(***kwds*)

Return a new HI object replacing specified fields with new values

`region`

Alias for field number 0

`year`

Alias for field number 1

`src.integrator.utilities.`**`convert_elec_price_to_lut`**(*prices: list[tuple[EI, float]]*) → dict[*EI*, float]

convert electricity prices to dictionary, look up table

Parameters

prices (*list[tuple[EI, float]]*) – list of prices

Returns

dict of prices

Return type
dict[*EI*, float]

`src.integrator.utilities.`**`convert_h2_price_records`**(*records: list[tuple[HI, float]]*) → dict[*HI*, float]

simple coversion from list of records to a dictionary LUT repeat entries should not occur and will generate an error

`src.integrator.utilities.`**`create_temporal_mapping`**(*sw_temporal*)

Combines the input mapping files within the electricity model to create a master temporal mapping dataframe. The df is used to build multiple temporal parameters used within the model. It creates a single dataframe that has 8760 rows for each hour in the year. Each hour in the year is assigned a season type, day type, and hour type used in the model. This defines the number of time periods the model will use based on cw_s_day and cw_hr inputs.

Returns
a dataframe with 8760 rows that include each hour, hour type, day, day type, and season. It also includes the weights for each day type and hour type.

Return type
dataframe

`src.integrator.utilities.`**`get_annual_wt_avg`**(*elec_price: DataFrame*) → dict[*HI*, float]

takes annual weighted average of hourly electricity prices

Parameters
elec_price (*pd.DataFrame*) – hourly electricity prices

Returns
annual weighted average electricity prices

Return type
dict[*HI*, float]

`src.integrator.utilities.`**`get_elec_price`**(*instance: PowerModel | ConcreteModel*, *block=None*) → DataFrame

pulls hourly electricity prices from completed PowerModel and de-weights them.

Prices from the duals are weighted by the day and year weights applied in the OBJ function This function retrieves the prices for all hours and removes the day and annual weights to return raw prices (and the day weights to use as needed)

Parameters

- **instance** (`PowerModel`) – solved electricity model
- **block** (*ConcreteModel*) – reference to the block if the electricity model is a block within a larger model

Returns
df of raw prices and the day weights to re-apply (if needed) columns: [r, y, hour, day_weight, raw_price]

Return type
pd.DataFrame

`src.integrator.utilities.`**`poll_h2_demand`**(*model: PowerModel*) → dict[*HI*, float]

Get the hydrogen demand by rep_year and region

Use the Generation variable for h2 techs

NOTE: Not sure about day weighting calculation here!!

Returns
dictionary of prices by H2 Index: price

Return type
dict[*HI*, float]

`src.integrator.utilities.`**`poll_h2_prices_from_elec`**(*model: PowerModel*, *tech*, *regions: Iterable*) → dict[Any, float]

poll the step-1 H2 price currently in the model for region/year, averaged over any steps

Parameters

- **`model`** (`PowerModel`) – solved PowerModel
- **`tech`** (*`str`*) – h2 tech
- **`regions`** (*`Iterable`*)

Returns
a dictionary of (region, seasons, year): price

Return type
dict[*Any*, float]

`src.integrator.utilities.`**`poll_hydrogen_price`**(*model: H2Model | ConcreteModel*, *block=None*) → list[tuple[*HI*, float]]

Retrieve the price of H2 from the H2 model

Parameters

- **`model`** (`H2Model`) – the model to poll
- **`block`** (*`optional`*) – block model to poll

Returns
list of H2 Index, price tuples

Return type
list[tuple[*HI*, float]]

`src.integrator.utilities.`**`poll_year_avg_elec_price`**(*price_list: list[tuple[EI, float]]*) → dict[*HI*, float]

retrieve a REPRESENTATIVE price at the annual level from a listing of prices

This function computes the AVERAGE elec price for each region-year combo

Parameters
`price_list` (*`list[tuple[EI, float]]`*) – input price list

Returns
a dictionary of (region, year): price

Return type
dict[*HI*, float]

`src.integrator.utilities.`**`regional_annual_prices`**(*m: PowerModel | ConcreteModel*, *block=None*) → dict[*HI*, float]

pulls all regional annual weighted electricity prices

Parameters

- **`m`** (*`Union['PowerModel', ConcreteModel]`*) – solved PowerModel
- **`block`** (*`optional`*) – solved block model if applicable, by default None

Returns
dict with regional annual electricity prices

Return type
dict[*HI*, float]

`src.integrator.utilities.select_solver(instance: ConcreteModel)`

Select solver based on learning method

Parameters
instance (PowerModel) – electricity pyomo model

Returns
The pyomo solver

Return type
solver type (?)

`src.integrator.utilities.simple_solve(m: ConcreteModel)`

a simple solve routine

`src.integrator.utilities.simple_solve_no_opt(m: ~pyomo.core.base.PyomoModel.ConcreteModel, opt: <pyomo.opt.base.solvers.SolverFactoryClass object at 0x00000241DC23D880>)`

Solve concrete model using solver factory object

Parameters

- **m** (*ConcreteModel*) – Pyomo model
- **opt** (*SolverFactory*) – Solver object initiated prior to solve

`src.integrator.utilities.update_elec_demand(self, elec_demand: dict[HI, float]) → None`

Update the external electical demand parameter with demands from the H2 model

Parameters
elec_demand (*dict[HI, float]*) – the new demands broken out by hyd index (region, year)

`src.integrator.utilities.update_h2_prices(model: PowerModel, h2_prices: dict[HI, float]) → None`

Update the H2 prices held in the model

Parameters
h2_prices (*list[tuple[HI, float]]*) – new prices

src.sensitivity

Modules

babymodel	Baby Model This file contains the TestBabyModel class, which is a subclass of ConcreteModel, along with scripts that help generate the model parameters and structure.
faster_sensitivity	faster_sensitivity
sensitivity_tools	Sensitivity Tools This file contains the AutoSympy, SensitivityMatrix, CoordMap, and DifferentialMapping classes.
speed_test	Speed Test This is a script with some functions to run speed and accuracy tests on test models constructed with babymodel.

src.sensitivity.babymodel

Baby Model This file contains the TestBabyModel class, which is a subclass of ConcreteModel, along with scripts that help generate the model parameters and structure.

The model structure is randomly generated through the functions generate and connect_subregions, with model parameters as input.

Functions

`E1`(z)	Classical case of the generalized exponential integral.
`Eijk`(*args, **kwargs)	Represent the Levi-Civita symbol.
`GramSchmidt`(vlist[, orthonormal])	Apply the Gram-Schmidt process to a set of vectors.
`LC`(f, *gens, **args)	Return the leading coefficient of `f`.
`LM`(f, *gens, **args)	Return the leading monomial of `f`.
`LT`(f, *gens, **args)	Return the leading term of `f`.
`N`(x[, n])	Calls x.evalf(n, **options).
`POSform`(variables, minterms[, dontcares])	The POSform function uses simplified_pairs and a redundant-group eliminating algorithm to convert the list of all input combinations that generate '1' (the minterms) into the smallest product-of-sums form.
`SOPform`(variables, minterms[, dontcares])	The SOPform function uses simplified_pairs and a redundant group- eliminating algorithm to convert the list of all input combos that generate '1' (the minterms) into the smallest sum-of-products form.
`Ynm_c`(n, m, theta, phi)	Conjugate spherical harmonics defined as
`abundance`(n)	Returns the difference between the sum of the positive proper divisors of a number and the number.
`apart`(f[, x, full])	Compute partial fraction decomposition of a rational function.
`apart_list`(f[, x, dummies])	Compute partial fraction decomposition of a rational function and return the result in structured form.
`apply_finite_diff`(order, x_list, y_list[, x0])	Calculates the finite difference approximation of the derivative of requested order at `x0` from points provided in `x_list` and `y_list`.
`approximants`(l[, X, simplify])	Return a generator for consecutive Pade approximants for a series.
`are_similar`(e1, e2)	Are two geometrical entities similar.
`arity`(cls)	Return the arity of the function if it is known, else None.
`ask`(proposition[, assumptions, context])	Function to evaluate the proposition with assumptions.
`assemble_partfrac_list`(partial_list)	Reassemble a full partial fraction decomposition from a structured result obtained by the function `apart_list`.
`assuming`(*assumptions)	Context manager for assumptions.
`banded`(*args, **kwargs)	Returns a SparseMatrix from the given dictionary describing the diagonals of the matrix.
`besselsimp`(expr)	Simplify bessel-type functions.
`binomial_coefficients`(n)	Return a dictionary containing pairs $(k1, k2) : C_kn$ where C_kn are binomial coefficients and $n = k1 + k2$.
`binomial_coefficients_list`(n)	Return a list of binomial coefficients as rows of the Pascal's triangle.
`block_collapse`(expr)	Evaluates a block matrix expression
`blockcut`(expr, rowsizes, colsizes)	Cut a matrix expression into Blocks

continues on next page

Table 60 – continued from previous page

`bool_map`(bool1, bool2)	Return the simplified version of *bool1*, and the mapping of variables that makes the two expressions *bool1* and *bool2* represent the same logical behaviour for some correspondence between the variables of each.
`bottom_up`(rv, F[, atoms, nonbasic])	Apply `F` to all expressions in an expression tree from the bottom up.
`bspline_basis`(d, knots, n, x)	The n-th B-spline at x of degree d with knots.
`bspline_basis_set`(d, knots, x)	Return the `len(knots)-d-1` B-splines at *x* of degree *d* with *knots*.
`cacheit`(func)	
`cancel`(f, *gens[, _signsimp])	Cancel common factors in a rational function `f`.
`capture`(func)	Return the printed output of func().
`casoratian`(seqs, n[, zero])	Given linear difference operator L of order 'k' and homogeneous equation Ly = 0 we want to compute kernel of L, which is a set of 'k' sequences: a(n), b(n), .
`cbrt`(arg[, evaluate])	Returns the principal cube root.
`ccode`(expr[, assign_to, standard])	Converts an expr to a string of c code
`centroid`(*args)	Find the centroid (center of mass) of the collection containing only Points, Segments or Polygons.
`chebyshevt_poly`(n[, x, polys])	Generates the Chebyshev polynomial of the first kind *T_n(x)*.
`chebyshevu_poly`(n[, x, polys])	Generates the Chebyshev polynomial of the second kind *U_n(x)*.
`check_assumptions`(expr[, against])	Checks whether assumptions of `expr` match the T/F assumptions given (or possessed by `against`).
`checkodesol`(ode, sol[, func, order, ...])	Substitutes `sol` into `ode` and checks that the result is `0`.
`checkpdesol`(pde, sol[, func, solve_for_func])	Checks if the given solution satisfies the partial differential equation.
`checksol`(f, symbol[, sol])	Checks whether sol is a solution of equation f == 0.
`classify_ode`(eq[, func, dict, ics, prep, ...])	Returns a tuple of possible `dsolve()` classifications for an ODE.
`classify_pde`(eq[, func, dict, prep])	Returns a tuple of possible pdsolve() classifications for a PDE.
`closest_points`(*args)	Return the subset of points from a set of points that were the closest to each other in the 2D plane.
`cofactors`(f, g, *gens, **args)	Compute GCD and cofactors of `f` and `g`.
`collect`(expr, syms[, func, evaluate, exact, ...])	Collect additive terms of an expression.
`collect_const`(expr, *vars[, Numbers])	A non-greedy collection of terms with similar number coefficients in an Add expr.
`combsimp`(expr)	Simplify combinatorial expressions.
`comp`(z1, z2[, tol])	Return a bool indicating whether the error between z1 and z2 is le `tol`.
`compose`(f, g, *gens, **args)	Compute functional composition `f(g)`.
`composite`(nth)	Return the nth composite number, with the composite numbers indexed as composite(1) = 4, composite(2) = 6, etc....
`compositepi`(n)	Return the number of positive composite numbers less than or equal to n.
`connect_regions`(region_map, hubmap, ...)	given a mapping of regions to lists of hubs, and hubs to regions, creates a set of arcs between hubs such that:
`construct_domain`(obj, **args)	Construct a minimal domain for a list of expressions.

continues on next page

Table 60 – continued from previous page

`content`(f, *gens, **args)	Compute GCD of coefficients of `f`.
`continued_fraction`(a)	Return the continued fraction representation of a Rational or quadratic irrational.
`continued_fraction_convergents`(cf)	Return an iterator over the convergents of a continued fraction (cf).
`continued_fraction_iterator`(x)	Return continued fraction expansion of x as iterator.
`continued_fraction_periodic`(p, q[, d, s])	Find the periodic continued fraction expansion of a quadratic irrational.
`continued_fraction_reduce`(cf)	Reduce a continued fraction to a rational or quadratic irrational.
`convex_hull`(*args[, polygon])	The convex hull surrounding the Points contained in the list of entities.
`convolution`(a, b[, cycle, dps, prime, ...])	Performs convolution by determining the type of desired convolution using hints.
`cosine_transform`(f, x, k, **hints)	Compute the unitary, ordinary-frequency cosine transform of *f*, defined as
`count_ops`(expr[, visual])	Return a representation (integer or expression) of the operations in expr.
`count_roots`(f[, inf, sup])	Return the number of roots of `f` in `[inf, sup]` interval.
`covering_product`(a, b)	Returns the covering product of given sequences.
`cse`(exprs[, symbols, optimizations, ...])	Perform common subexpression elimination on an expression.
`cxxcode`(expr[, assign_to, standard])	C++ equivalent of `ccode()`.
`cycle_length`(f, x0[, nmax, values])	For a given iterated sequence, return a generator that gives the length of the iterated cycle (lambda) and the length of terms before the cycle begins (mu); if `values` is True then the terms of the sequence will be returned instead.
`cyclotomic_poly`(n[, x, polys])	Generates cyclotomic polynomial of order *n* in *x*.
`decompogen`(f, symbol)	Computes General functional decomposition of `f`. Given an expression `f`, returns a list `[f_1, f_2, ..., f_n]`, where:: f = f_1 o f_2 o ... f_n = f_1(f_2(... f_n)).
`decompose`(f, *gens, **args)	Compute functional decomposition of `f`.
`default_sort_key`(item[, order])	Return a key that can be used for sorting.
`deg`(r)	Return the degree value for the given radians (pi = 180 degrees).
`degree`(f[, gen])	Return the degree of `f` in the given variable.
`degree_list`(f, *gens, **args)	Return a list of degrees of `f` in all variables.
`denom`(expr)	
`derive_by_array`(expr, dx)	Derivative by arrays.
`det`(matexpr)	Matrix Determinant
`det_quick`(M[, method])	Return `det(M)` assuming that either there are lots of zeros or the size of the matrix is small.
`diag`(*values[, strict, unpack])	Returns a matrix with the provided values placed on the diagonal.
`diagonalize_vector`(vector)	
`dict_merge`(*dicts)	Merge dictionaries into a single dictionary.
`diff`(f, *symbols, **kwargs)	Differentiate f with respect to symbols.
`difference_delta`(expr[, n, step])	Difference Operator.

continues on next page

Table 60 – continued from previous page

`differentiate_finite`(expr, *symbols[, ...])	Differentiate expr and replace Derivatives with finite differences.
`diophantine`(eq[, param, syms, permute])	Simplify the solution procedure of diophantine equation `eq` by converting it into a product of terms which should equal zero.
`discrete_log`(n, a, b[, order, prime_order])	Compute the discrete logarithm of `a` to the base `b` modulo `n`.
`discriminant`(f, *gens, **args)	Compute discriminant of `f`.
`div`(f, g, *gens, **args)	Compute polynomial division of `f` and `g`.
`divisor_count`(n[, modulus, proper])	Return the number of divisors of `n`.
`divisors`(n[, generator, proper])	Return all divisors of n sorted from 1..n by default.
`dotprint`(expr[, styles, atom, maxdepth, ...])	DOT description of a SymPy expression tree
`dsolve`(eq[, func, hint, simplify, ics, xi, ...])	Solves any (supported) kind of ordinary differential equation and system of ordinary differential equations.
`egyptian_fraction`(r[, algorithm])	Return the list of denominators of an Egyptian fraction expansion **[1]_** of the said rational *r*.
`epath`(path[, expr, func, args, kwargs])	Manipulate parts of an expression selected by a path.
`euler_equations`(L[, funcs, vars])	Find the Euler-Lagrange equations **[1]_** for a given Lagrangian.
`evaluate`(x)	Control automatic evaluation
`expand`(e[, deep, modulus, power_base, ...])	Expand an expression using methods given as hints.
`expand_complex`(expr[, deep])	Wrapper around expand that only uses the complex hint.
`expand_func`(expr[, deep])	Wrapper around expand that only uses the func hint.
`expand_log`(expr[, deep, force, factor])	Wrapper around expand that only uses the log hint.
`expand_mul`(expr[, deep])	Wrapper around expand that only uses the mul hint.
`expand_multinomial`(expr[, deep])	Wrapper around expand that only uses the multinomial hint.
`expand_power_base`(expr[, deep, force])	Wrapper around expand that only uses the power_base hint.
`expand_power_exp`(expr[, deep])	Wrapper around expand that only uses the power_exp hint.
`expand_trig`(expr[, deep])	Wrapper around expand that only uses the trig hint.
`exptrigsimp`(expr)	Simplifies exponential / trigonometric / hyperbolic functions.
`exquo`(f, g, *gens, **args)	Compute polynomial exact quotient of `f` and `g`.
`eye`(*args, **kwargs)	Create square identity matrix n x n
`factor`(f, *gens[, deep])	Compute the factorization of expression, `f`, into irreducibles.
`factor_list`(f, *gens, **args)	Compute a list of irreducible factors of `f`.
`factor_nc`(expr)	Return the factored form of `expr` while handling non-commutative expressions.
`factor_terms`(expr[, radical, clear, ...])	Remove common factors from terms in all arguments without changing the underlying structure of the expr.
`factorint`(n[, limit, use_trial, use_rho, ...])	Given a positive integer `n`, `factorint(n)` returns a dict containing the prime factors of `n` as keys and their respective multiplicities as values.
`factorrat`(rat[, limit, use_trial, use_rho, ...])	Given a Rational `r`, `factorrat(r)` returns a dict containing the prime factors of `r` as keys and their respective multiplicities as values.
`failing_assumptions`(expr, **assumptions)	Return a dictionary containing assumptions with values not matching those of the passed assumptions.

continues on next page

Table 60 – continued from previous page

`farthest_points`(*args)	Return the subset of points from a set of points that were the furthest apart from each other in the 2D plane.
`fcode`(expr[, assign_to])	Converts an expr to a string of fortran code
`fft`(seq[, dps])	Performs the Discrete Fourier Transform (**DFT**) in the complex domain.
`field`(symbols, domain[, order])	Construct new rational function field returning (field, x1, ..., xn).
`field_isomorphism`(a, b, *[, fast])	Find an embedding of one number field into another.
`filldedent`(s[, w])	Strips leading and trailing empty lines from a copy of `s`, then dedents, fills and returns it.
`finite_diff_weights`(order, x_list[, x0])	Calculates the finite difference weights for an arbitrarily spaced one-dimensional grid (`x_list`) for derivatives at `x0` of order 0, 1, ..., up to `order` using a recursive formula.
`flatten`(iterable[, levels, cls])	Recursively denest iterable containers.
`fourier_series`(f[, limits, finite])	Computes the Fourier trigonometric series expansion.
`fourier_transform`(f, x, k, **hints)	Compute the unitary, ordinary-frequency Fourier transform of `f`, defined as
`fps`(f[, x, x0, dir, hyper, order, rational, ...])	Generates Formal Power Series of `f`.
`fraction`(expr[, exact])	Returns a pair with expression's numerator and denominator.
`fu`(rv[, measure])	Attempt to simplify expression by using transformation rules given in the algorithm by Fu et al.
`fwht`(seq)	Performs the Walsh Hadamard Transform (**WHT**), and uses Hadamard ordering for the sequence.
`galois_group`(f, *gens[, by_name, max_tries, ...])	Compute the Galois group for polynomials f up to degree 6.
`gammasimp`(expr)	Simplify expressions with gamma functions.
`gcd`(f[, g])	Compute GCD of `f` and `g`.
`gcd_list`(seq, *gens, **args)	Compute GCD of a list of polynomials.
`gcd_terms`(terms[, isprimitive, clear, fraction])	Compute the GCD of `terms` and put them together.
`gcdex`(f, g, *gens, **args)	Extended Euclidean algorithm of `f` and `g`.
`generate`([num_regions, hubs_per_region, ...])	generates a random network with all parameters required to initialize a ToyBabyModel
`getLogger`([name])	Return a logger with the specified name, creating it if necessary.
`get_contraction_structure`(expr)	Determine dummy indices of `expr` and describe its structure
`get_indices`(expr)	Determine the outer indices of expression `expr`
`gff`(f, *gens, **args)	Compute greatest factorial factorization of `f`.
`gff_list`(f, *gens, **args)	Compute a list of greatest factorial factors of `f`.
`glsl_code`(expr[, assign_to])	Converts an expr to a string of GLSL code
`groebner`(F, *gens, **args)	Computes the reduced Groebner basis for a set of polynomials.
`ground_roots`(f, *gens, **args)	Compute roots of `f` by factorization in the ground domain.
`group`(seq[, multiple])	Splits a sequence into a list of lists of equal, adjacent elements.
`gruntz`(e, z, z0[, dir])	Compute the limit of e(z) at the point z0 using the Gruntz algorithm.
`hadamard_product`(*matrices)	Return the elementwise (aka Hadamard) product of matrices.

continues on next page

Table 60 – continued from previous page

`half_gcdex`(f, g, *gens, **args)	Half extended Euclidean algorithm of `f` and `g`.
`hankel_transform`(f, r, k, nu, **hints)	Compute the Hankel transform of *f*, defined as
`has_dups`(seq)	Return True if there are any duplicate elements in `seq`.
`has_variety`(seq)	Return True if there are any different elements in `seq`.
`hermite_poly`(n[, x, polys])	Generates the Hermite polynomial *H_n(x)*.
`hermite_prob_poly`(n[, x, polys])	Generates the probabilist's Hermite polynomial *He_n(x)*.
`hessian`(f, varlist[, constraints])	Compute Hessian matrix for a function f wrt parameters in varlist which may be given as a sequence or a row/column vector.
`homogeneous_order`(eq, *symbols)	Returns the order *n* if *g* is homogeneous and `None` if it is not homogeneous.
`horner`(f, *gens, **args)	Rewrite a polynomial in Horner form.
`hyperexpand`(f[, allow_hyper, rewrite, place])	Expand hypergeometric functions.
`hypersimilar`(f, g, k)	Returns True if `f` and `g` are hyper-similar.
`hypersimp`(f, k)	Given combinatorial term f(k) simplify its consecutive term ratio i.e. f(k+1)/f(k).
`idiff`(eq, y, x[, n])	Return `dy/dx` assuming that `eq == 0`.
`ifft`(seq[, dps])	Performs the Discrete Fourier Transform (**DFT**) in the complex domain.
`ifwht`(seq)	Performs the Walsh Hadamard Transform (**WHT**), and uses Hadamard ordering for the sequence.
`igcd`(*args)	Computes nonnegative integer greatest common divisor.
`ilcm`(*args)	Computes integer least common multiple.
`imageset`(*args)	Return an image of the set under transformation `f`.
`init_printing`([pretty_print, order, ...])	Initializes pretty-printer depending on the environment.
`init_session`([ipython, pretty_print, order, ...])	Initialize an embedded IPython or Python session.
`integer_log`(y, x)	Returns `(e, bool)` where e is the largest nonnegative integer such that $\|y\| \geq \|x^e\|$ and `bool` is True if $y = x^e$.
`integer_nthroot`(y, n)	Return a tuple containing x = floor(y**(1/n)) and a boolean indicating whether the result is exact (that is, whether x**n == y).
`integrate`(f, var, ...)	
`interactive_traversal`(expr)	Traverse a tree asking a user which branch to choose.
`interpolate`(data, x)	Construct an interpolating polynomial for the data points evaluated at point x (which can be symbolic or numeric).
`interpolating_poly`(n, x[, X, Y])	Construct Lagrange interpolating polynomial for `n` data points.
`interpolating_spline`(d, x, X, Y)	Return spline of degree *d*, passing through the given *X* and *Y* values.
`intersecting_product`(a, b)	Returns the intersecting product of given sequences.
`intersection`(*entities[, pairwise])	The intersection of a collection of GeometryEntity instances.
`intervals`(F[, all, eps, inf, sup, strict, ...])	Compute isolating intervals for roots of `f`.
`intt`(seq, prime)	Performs the Number Theoretic Transform (**NTT**), which specializes the Discrete Fourier Transform (**DFT**) over quotient ring *Z/pZ* for prime *p* instead of complex numbers *C*.
`inv_quick`(M)	Return the inverse of `M`, assuming that either there are lots of zeros or the size of the matrix is small.

continues on next page

Table 60 – continued from previous page

`inverse_cosine_transform`(F, k, x, **hints)	Compute the unitary, ordinary-frequency inverse cosine transform of *F*, defined as
`inverse_fourier_transform`(F, k, x, **hints)	Compute the unitary, ordinary-frequency inverse Fourier transform of *F*, defined as
`inverse_hankel_transform`(F, k, r, nu, **hints)	Compute the inverse Hankel transform of *F* defined as
`inverse_laplace_transform`(F, s, t[, plane])	Compute the inverse Laplace transform of *F(s)*, defined as
`inverse_mellin_transform`(F, s, x, strip, **hints)	Compute the inverse Mellin transform of *F(s)* over the fundamental strip given by `strip=(a, b)`.
`inverse_mobius_transform`(seq[, subset])	Performs the Mobius Transform for subset lattice with indices of sequence as bitmasks.
`inverse_sine_transform`(F, k, x, **hints)	Compute the unitary, ordinary-frequency inverse sine transform of *F*, defined as
`invert`(f, g, *gens, **args)	Invert `f` modulo `g` when possible.
`is_abundant`(n)	Returns True if `n` is an abundant number, else False.
`is_amicable`(m, n)	Returns True if the numbers *m* and *n* are "amicable", else False.
`is_convex`(f, *syms[, domain])	Determines the convexity of the function passed in the argument.
`is_decreasing`(expression[, interval, symbol])	Return whether the function is decreasing in the given interval.
`is_deficient`(n)	Returns True if `n` is a deficient number, else False.
`is_increasing`(expression[, interval, symbol])	Return whether the function is increasing in the given interval.
`is_mersenne_prime`(n)	Returns True if `n` is a Mersenne prime, else False.
`is_monotonic`(expression[, interval, symbol])	Return whether the function is monotonic in the given interval.
`is_nthpow_residue`(a, n, m)	Returns True if `x**n == a (mod m)` has solutions.
`is_perfect`(n)	Returns True if `n` is a perfect number, else False.
`is_primitive_root`(a, p)	Returns True if `a` is a primitive root of `p`.
`is_quad_residue`(a, p)	Returns True if `a` (mod `p`) is in the set of squares mod `p`, i.e a % p in set([i**2 % p for i in range(p)]).
`is_strictly_decreasing`(expression[, ...])	Return whether the function is strictly decreasing in the given interval.
`is_strictly_increasing`(expression[, ...])	Return whether the function is strictly increasing in the given interval.
`is_zero_dimensional`(F, *gens, **args)	Checks if the ideal generated by a Groebner basis is zero-dimensional.
`isolate`(alg[, eps, fast])	Find a rational isolating interval for a real algebraic number.
`isprime`(n)	Test if n is a prime number (True) or not (False).
`itermonomials`(variables, max_degrees[, ...])	`max_degrees` and `min_degrees` are either both integers or both lists.
`jacobi_normalized`(n, a, b, x)	Jacobi polynomial $P_n^{left(alpha, betaright)}(x)$.
`jacobi_poly`(n, a, b[, x, polys])	Generates the Jacobi polynomial *P_n^{(a,b)}(x)*.
`jacobi_symbol`(m, n)	Returns the Jacobi symbol *(m / n)*.
`jn_zeros`(n, k[, method, dps])	Zeros of the spherical Bessel function of the first kind.
`jordan_cell`(eigenval, n)	Create a Jordan block:
`jscode`(expr[, assign_to])	Converts an expr to a string of javascript code
`julia_code`(expr[, assign_to])	Converts *expr* to a string of Julia code.
`kronecker_product`(*matrices)	The Kronecker product of two or more arguments.
`kroneckersimp`(expr)	Simplify expressions with KroneckerDelta.

continues on next page

Table 60 – continued from previous page

`laguerre_poly`(n[, x, alpha, polys])	Generates the Laguerre polynomial *L_n^{(alpha)}(x)*.
`lambdify`(args, expr[, modules, printer, ...])	Convert a SymPy expression into a function that allows for fast numeric evaluation.
`laplace_transform`(f, t, s[, legacy_matrix])	Compute the Laplace Transform *F(s)* of *f(t)*,
`lcm`(f[, g])	Compute LCM of `f` and `g`.
`lcm_list`(seq, *gens, **args)	Compute LCM of a list of polynomials.
`legendre_poly`(n[, x, polys])	Generates the Legendre polynomial *P_n(x)*.
`legendre_symbol`(a, p)	Returns the Legendre symbol *(a / p)*.
`limit`(e, z, z0[, dir])	Computes the limit of `e(z)` at the point `z0`.
`limit_seq`(expr[, n, trials])	Finds the limit of a sequence as index `n` tends to infinity.
`line_integrate`(field, Curve, variables)	Compute the line integral.
`linear_eq_to_matrix`(equations, *symbols)	Converts a given System of Equations into Matrix form.
`linsolve`(system, *symbols)	Solve system of N linear equations with M variables; both underdetermined and overdetermined systems are supported.
`list2numpy`(l[, dtype])	Converts Python list of SymPy expressions to a NumPy array.
`logcombine`(expr[, force])	Takes logarithms and combines them using the following rules:
`maple_code`(expr[, assign_to])	Converts `expr` to a string of Maple code.
`mathematica_code`(expr, **settings)	Converts an expr to a string of the Wolfram Mathematica code
`matrix2numpy`(m[, dtype])	Converts SymPy's matrix to a NumPy array.
`matrix_multiply_elementwise`(A, B)	Return the Hadamard product (elementwise product) of A and B
`matrix_symbols`(expr)	
`maximum`(f, symbol[, domain])	Returns the maximum value of a function in the given domain.
`mellin_transform`(f, x, s, **hints)	Compute the Mellin transform *F(s)* of *f(x)*,
`memoize_property`(propfunc)	Property decorator that caches the value of potentially expensive *propfunc* after the first evaluation.
`mersenne_prime_exponent`(nth)	Returns the exponent `i` for the nth Mersenne prime (which has the form *2^i - 1*).
`minimal_polynomial`(ex[, x, compose, polys, ...])	Computes the minimal polynomial of an algebraic element.
`minimum`(f, symbol[, domain])	Returns the minimum value of a function in the given domain.
`minpoly`(ex[, x, compose, polys, domain])	This is a synonym for `minimal_polynomial()`.
`mobius_transform`(seq[, subset])	Performs the Mobius Transform for subset lattice with indices of sequence as bitmasks.
`mod_inverse`(a, m)	Return the number c such that, $a times c = 1 pmod{m}$ where c has the same sign as m.
`monic`(f, *gens, **args)	Divide all coefficients of `f` by `LC(f)`.
`multiline_latex`(lhs, rhs[, terms_per_line, ...])	This function generates a LaTeX equation with a multiline right-hand side in an `align*`, `eqnarray` or `IEEEeqnarray` environment.
`multinomial_coefficients`(m, n)	Return a dictionary containing pairs `{(k1,k2,..,km) : C_kn}` where `C_kn` are multinomial coefficients such that `n=k1+k2+..+km`.
`multiplicity`(p, n)	Find the greatest integer m such that p**m divides n.
`n_order`(a, n)	Returns the order of `a` modulo `n`.

continues on next page

Table 60 – continued from previous page

`nextprime`(n[, ith])	Return the ith prime greater than n.
`nfloat`(expr[, n, exponent, dkeys])	Make all Rationals in expr Floats except those in exponents (unless the exponents flag is set to True) and those in undefined functions.
`nonlinsolve`(system, *symbols)	Solve system of N nonlinear equations with M variables, which means both under and overdetermined systems are supported.
`not_empty_in`(finset_intersection, *syms)	Finds the domain of the functions in `finset_intersection` in which the `finite_set` is not-empty.
`npartitions`(n[, verbose])	Calculate the partition function P(n), i.e. the number of ways that n can be written as a sum of positive integers.
`nroots`(f[, n, maxsteps, cleanup])	Compute numerical approximations of roots of `f`.
`nsimplify`(expr[, constants, tolerance, ...])	Find a simple representation for a number or, if there are free symbols or if `rational=True`, then replace Floats with their Rational equivalents.
`nsolve`(*args[, dict])	Solve a nonlinear equation system numerically: `nsolve(f, [args,] x0, modules=['mpmath'], **kwargs)`.
`nth_power_roots_poly`(f, n, *gens, **args)	Construct a polynomial with n-th powers of roots of `f`.
`nthroot_mod`(a, n, p[, all_roots])	Find the solutions to `x**n = a mod p`.
`ntt`(seq, prime)	Performs the Number Theoretic Transform (**NTT**), which specializes the Discrete Fourier Transform (**DFT**) over quotient ring *Z/pZ* for prime *p* instead of complex numbers *C*.
`numbered_symbols`([prefix, cls, start, exclude])	Generate an infinite stream of Symbols consisting of a prefix and increasing subscripts provided that they do not occur in `exclude`.
`numer`(expr)	
`octave_code`(expr[, assign_to])	Converts *expr* to a string of Octave (or Matlab) code.
`ode_order`(expr, func)	Returns the order of a given differential equation with respect to func.
`ones`(*args, **kwargs)	Returns a matrix of ones with `rows` rows and `cols` columns; if `cols` is omitted a square matrix will be returned.
`ordered`(seq[, keys, default, warn])	Return an iterator of the seq where keys are used to break ties in a conservative fashion: if, after applying a key, there are no ties then no other keys will be computed.
`pager_print`(expr, **settings)	Prints expr using the pager, in pretty form.
`parallel_poly_from_expr`(exprs, *gens, **args)	Construct polynomials from expressions.
`parse_expr`(s[, local_dict, transformations, ...])	Converts the string `s` to a SymPy expression, in `local_dict`.
`pde_separate`(eq, fun, sep[, strategy])	Separate variables in partial differential equation either by additive or multiplicative separation approach.
`pde_separate_add`(eq, fun, sep)	Helper function for searching additive separable solutions.
`pde_separate_mul`(eq, fun, sep)	Helper function for searching multiplicative separable solutions.
`pdiv`(f, g, *gens, **args)	Compute polynomial pseudo-division of `f` and `g`.
`pdsolve`(eq[, func, hint, dict, solvefun])	Solves any (supported) kind of partial differential equation.

continues on next page

Table 60 – continued from previous page

`per`(matexpr)	Matrix Permanent
`perfect_power`(n[, candidates, big, factor])	Return `(b, e)` such that `n` == `b**e` if `n` is a unique perfect power with `e > 1`, else `False` (e.g. 1 is not a perfect power).
`periodicity`(f, symbol[, check])	Tests the given function for periodicity in the given symbol.
`permutedims`(expr[, perm, index_order_old, ...])	Permutes the indices of an array.
`pexquo`(f, g, *gens, **args)	Compute polynomial exact pseudo-quotient of `f` and `g`.
`piecewise_exclusive`(expr, *[, skip_nan, deep])	Rewrite `Piecewise` with mutually exclusive conditions.
`piecewise_fold`(expr[, evaluate])	Takes an expression containing a piecewise function and returns the expression in piecewise form.
`plot`(*args[, show])	Plots a function of a single variable as a curve.
`plot_implicit`(expr[, x_var, y_var, ...])	A plot function to plot implicit equations / inequalities.
`plot_parametric`(*args[, show])	Plots a 2D parametric curve.
`polarify`(eq[, subs, lift])	Turn all numbers in eq into their polar equivalents (under the standard choice of argument).
`pollard_pm1`(n[, B, a, retries, seed])	Use Pollard's p-1 method to try to extract a nontrivial factor of `n`.
`pollard_rho`(n[, s, a, retries, seed, ...])	Use Pollard's rho method to try to extract a nontrivial factor of `n`.
`poly`(expr, *gens, **args)	Efficiently transform an expression into a polynomial.
`poly_from_expr`(expr, *gens, **args)	Construct a polynomial from an expression.
`posify`(eq)	Return `eq` (with generic symbols made positive) and a dictionary containing the mapping between the old and new symbols.
`postfixes`(seq)	Generate all postfixes of a sequence.
`postorder_traversal`(node[, keys])	Do a postorder traversal of a tree.
`powdenest`(eq[, force, polar])	Collect exponents on powers as assumptions allow.
`powsimp`(expr[, deep, combine, force, measure])	Reduce expression by combining powers with similar bases and exponents.
`pprint`(expr, **kwargs)	Prints expr in pretty form.
`pprint_try_use_unicode`()	See if unicode output is available and leverage it if possible
`pprint_use_unicode`([flag])	Set whether pretty-printer should use unicode by default
`pquo`(f, g, *gens, **args)	Compute polynomial pseudo-quotient of `f` and `g`.
`prefixes`(seq)	Generate all prefixes of a sequence.
`prem`(f, g, *gens, **args)	Compute polynomial pseudo-remainder of `f` and `g`.
`pretty_print`(expr, **kwargs)	Prints expr in pretty form.
`preview`(expr[, output, viewer, euler, ...])	View expression or LaTeX markup in PNG, DVI, PostScript or PDF form.
`prevprime`(n)	Return the largest prime smaller than n.
`prime`(nth)	Return the nth prime, with the primes indexed as prime(1) = 2, prime(2) = 3, etc.
`prime_decomp`(p[, T, ZK, dK, radical])	Compute the decomposition of rational prime p in a number field.
`prime_valuation`(I, P)	Compute the P-adic valuation for an integral ideal I.
`primefactors`(n[, limit, verbose])	Return a sorted list of n's prime factors, ignoring multiplicity and any composite factor that remains if the limit was set too low for complete factorization.
`primerange`(a[, b])	Generate a list of all prime numbers in the range [2, a), or [a, b).
`primitive`(f, *gens, **args)	Compute content and the primitive form of `f`.

continues on next page

Table 60 – continued from previous page

`primitive_element`(extension[, x, ex, polys])	Find a single generator for a number field given by several generators.
`primitive_root`(p)	Returns the smallest primitive root or None.
`primorial`(n[, nth])	Returns the product of the first n primes (default) or the primes less than or equal to n (when `nth=False`).
`print_ccode`(expr, **settings)	Prints C representation of the given expression.
`print_fcode`(expr, **settings)	Prints the Fortran representation of the given expression.
`print_glsl`(expr, **settings)	Prints the GLSL representation of the given expression.
`print_gtk`(x[, start_viewer])	Print to Gtkmathview, a gtk widget capable of rendering MathML.
`print_jscode`(expr, **settings)	Prints the Javascript representation of the given expression.
`print_latex`(expr, **settings)	Prints LaTeX representation of the given expression.
`print_maple_code`(expr, **settings)	Prints the Maple representation of the given expression.
`print_mathml`(expr[, printer])	Prints a pretty representation of the MathML code for expr.
`print_python`(expr, **settings)	Print output of python() function
`print_rcode`(expr, **settings)	Prints R representation of the given expression.
`print_tree`(node[, assumptions])	Prints a tree representation of "node".
`prod`(a[, start])	Return product of elements of a. Start with int 1 so if only
`product`(*args, **kwargs)	Compute the product.
`proper_divisor_count`(n[, modulus])	Return the number of proper divisors of `n`.
`proper_divisors`(n[, generator])	Return all divisors of n except n, sorted by default.
`public`(obj)	Append `obj`'s name to global `__all__` variable (call site).
`pycode`(expr, **settings)	Converts an expr to a string of Python code
`python`(expr, **settings)	Return Python interpretation of passed expression (can be passed to the exec() function without any modifications)
`quadratic_congruence`(a, b, c, p)	Find the solutions to ``a x**2 + b x + c = 0 mod p.
`quadratic_residues`(p)	Returns the list of quadratic residues.
`quo`(f, g, *gens, **args)	Compute polynomial quotient of `f` and `g`.
`rad`(d)	Return the radian value for the given degrees (pi = 180 degrees).
`radsimp`(expr[, symbolic, max_terms])	Rationalize the denominator by removing square roots.
`randMatrix`(r[, c, min, max, seed, ...])	Create random matrix with dimensions `r` x `c`.
`random_poly`(x, n, inf, sup[, domain, polys])	Generates a polynomial of degree `n` with coefficients in `[inf, sup]`.
`randprime`(a, b)	Return a random prime number in the range [a, b).
`rational_interpolate`(data, degnum[, X])	Returns a rational interpolation, where the data points are element of any integral domain.
`ratsimp`(expr)	Put an expression over a common denominator, cancel and reduce.
`ratsimpmodprime`(expr, G, *gens[, quick, ...])	Simplifies a rational expression `expr` modulo the prime ideal generated by `G`.
`rcode`(expr[, assign_to])	Converts an expr to a string of r code
`rcollect`(expr, *vars)	Recursively collect sums in an expression.
`real_root`(arg[, n, evaluate])	Return the real *n*'th-root of *arg* if possible.
`real_roots`(f[, multiple])	Return a list of real roots with multiplicities of `f`.
`reduce_abs_inequalities`(exprs, gen)	Reduce a system of inequalities with nested absolute values.

continues on next page

Table 60 – continued from previous page

`reduce_abs_inequality`(expr, rel, gen)	Reduce an inequality with nested absolute values.
`reduce_inequalities`(inequalities[, symbols])	Reduce a system of inequalities with rational coefficients.
`reduced`(f, G, *gens, **args)	Reduces a polynomial `f` modulo a set of polynomials `G`.
`refine`(expr[, assumptions])	Simplify an expression using assumptions.
`refine_root`(f, s, t[, eps, steps, fast, ...])	Refine an isolating interval of a root to the given precision.
`register_handler`(key, handler)	Register a handler in the ask system.
`rem`(f, g, *gens, **args)	Compute polynomial remainder of `f` and `g`.
`remove_handler`(key, handler)	Removes a handler from the ask system.
`reshape`(seq, how)	Reshape the sequence according to the template in `how`.
`residue`(expr, x, x0)	Finds the residue of `expr` at the point x=x0.
`resultant`(f, g, *gens[, includePRS])	Compute resultant of `f` and `g`.
`ring`(symbols, domain[, order])	Construct a polynomial ring returning `(ring, x_1, ..., x_n)`.
`root`(arg, n[, k, evaluate])	Returns the k-th n-th root of `arg`.
`rootof`(f, x[, index, radicals, expand])	An indexed root of a univariate polynomial.
`roots`(f, *gens[, auto, cubics, trig, ...])	Computes symbolic roots of a univariate polynomial.
`rot_axis1`(theta)	Returns a rotation matrix for a rotation of theta (in radians) about the 1-axis.
`rot_axis2`(theta)	Returns a rotation matrix for a rotation of theta (in radians) about the 2-axis.
`rot_axis3`(theta)	Returns a rotation matrix for a rotation of theta (in radians) about the 3-axis.
`rot_ccw_axis1`(theta)	Returns a rotation matrix for a rotation of theta (in radians) about the 1-axis.
`rot_ccw_axis2`(theta)	Returns a rotation matrix for a rotation of theta (in radians) about the 2-axis.
`rot_ccw_axis3`(theta)	Returns a rotation matrix for a rotation of theta (in radians) about the 3-axis.
`rot_givens`(i, j, theta[, dim])	Returns a a Givens rotation matrix, a a rotation in the plane spanned by two coordinates axes.
`rotations`(s[, dir])	Return a generator giving the items in s as list where each subsequent list has the items rotated to the left (default) or right (`dir=-1`) relative to the previous list.
`round_two`(T[, radicals])	Zassenhaus's "Round 2" algorithm.
`rsolve`(f, y[, init])	Solve univariate recurrence with rational coefficients.
`rsolve_hyper`(coeffs, f, n, **hints)	Given linear recurrence operator *operatorname{L}* of order k with polynomial coefficients and inhomogeneous equation *operatorname{L}* $y = f$ we seek for all hypergeometric solutions over field K of characteristic zero.
`rsolve_poly`(coeffs, f, n[, shift])	Given linear recurrence operator *operatorname{L}* of order k with polynomial coefficients and inhomogeneous equation *operatorname{L}* $y = f$, where f is a polynomial, we seek for all polynomial solutions over field K of characteristic zero.
`rsolve_ratio`(coeffs, f, n, **hints)	Given linear recurrence operator *operatorname{L}* of order k with polynomial coefficients and inhomogeneous equation *operatorname{L}* $y = f$, where f is a polynomial, we seek for all rational solutions over field K of characteristic zero.
`rust_code`(expr[, assign_to])	Converts an expr to a string of Rust code

continues on next page

Table 60 – continued from previous page

`satisfiable`(expr[, algorithm, all_models, ...])	Check satisfiability of a propositional sentence.
`separatevars`(expr[, symbols, dict, force])	Separates variables in an expression, if possible.
`sequence`(seq[, limits])	Returns appropriate sequence object.
`series`(expr[, x, x0, n, dir])	Series expansion of expr around point $x = x0$.
`seterr`([divide])	Should SymPy raise an exception on 0/0 or return a nan?
`sfield`(exprs, *symbols, **options)	Construct a field deriving generators and domain from options and input expressions.
`shape`()	Return the shape of the *expr* as a tuple.
`sift`(seq, keyfunc[, binary])	Sift the sequence, `seq` according to `keyfunc`.
`signsimp`(expr[, evaluate])	Make all Add sub-expressions canonical wrt sign.
`simplify`(expr[, ratio, measure, rational, ...])	Simplifies the given expression.
`simplify_logic`(expr[, form, deep, force, ...])	This function simplifies a boolean function to its simplified version in SOP or POS form.
`sine_transform`(f, x, k, **hints)	Compute the unitary, ordinary-frequency sine transform of *f*, defined as
`singularities`(expression, symbol[, domain])	Find singularities of a given function.
`singularityintegrate`(f, x)	This function handles the indefinite integrations of Singularity functions.
`smtlib_code`(expr[, auto_assert, ...])	Converts `expr` to a string of smtlib code.
`solve`(f, *symbols, **flags)	Algebraically solves equations and systems of equations.
`solve_linear`(lhs[, rhs, symbols, exclude])	Return a tuple derived from `f = lhs - rhs` that is one of the following: `(0, 1)`, `(0, 0)`, `(symbol, solution)`, `(n, d)`.
`solve_linear_system`(system, *symbols, **flags)	Solve system of N linear equations with M variables, which means both under- and overdetermined systems are supported.
`solve_linear_system_LU`(matrix, syms)	Solves the augmented matrix system using `LUsolve` and returns a dictionary in which solutions are keyed to the symbols of *syms* as ordered.
`solve_poly_inequality`(poly, rel)	Solve a polynomial inequality with rational coefficients.
`solve_poly_system`(seq, *gens[, strict])	Return a list of solutions for the system of polynomial equations or else None.
`solve_rational_inequalities`(eqs)	Solve a system of rational inequalities with rational coefficients.
`solve_triangulated`(polys, *gens, **args)	Solve a polynomial system using Gianni-Kalkbrenner algorithm.
`solve_undetermined_coeffs`(equ, coeffs, ...)	Solve a system of equations in k parameters that is formed by matching coefficients in variables `coeffs` that are on factors dependent on the remaining variables (or those given explicitly by `syms`.
`solve_univariate_inequality`(expr, gen[, ...])	Solves a real univariate inequality.
`solveset`(f[, symbol, domain])	Solves a given inequality or equation with set as output
`sqf`(f, *gens, **args)	Compute square-free factorization of `f`.
`sqf_list`(f, *gens, **args)	Compute a list of square-free factors of `f`.
`sqf_norm`(f, *gens, **args)	Compute square-free norm of `f`.
`sqf_part`(f, *gens, **args)	Compute square-free part of `f`.
`sqrt`(arg[, evaluate])	Returns the principal square root.
`sqrt_mod`(a, p[, all_roots])	Find a root of `x**2 = a mod p`.
`sqrt_mod_iter`(a, p[, domain])	Iterate over solutions to `x**2 = a mod p`.
`sqrtdenest`(expr[, max_iter])	Denests sqrts in an expression that contain other square roots if possible, otherwise returns the expr unchanged.

continues on next page

Table 60 – continued from previous page

`sring`(exprs, *symbols, **options)	Construct a ring deriving generators and domain from options and input expressions.
`stationary_points`(f, symbol[, domain])	Returns the stationary points of a function (where derivative of the function is 0) in the given domain.
`sturm`(f, *gens, **args)	Compute Sturm sequence of `f`.
`subresultants`(f, g, *gens, **args)	Compute subresultant PRS of `f` and `g`.
`subsets`(seq[, k, repetition])	Generates all k-subsets (combinations) from an n-element set, `seq`.
`substitution`(system, symbols[, result, ...])	Solves the *system* using substitution method.
`summation`(f, *symbols, **kwargs)	Compute the summation of f with respect to symbols.
`swinnerton_dyer_poly`(n[, x, polys])	Generates n-th Swinnerton-Dyer polynomial in x.
`symarray`(prefix, shape, **kwargs)	Create a numpy ndarray of symbols (as an object array).
`symbols`(names, *[, cls])	Transform strings into instances of `Symbol` class.
`symmetric_poly`(n, *gens[, polys])	Generates symmetric polynomial of order n.
`symmetrize`(F, *gens, **args)	Rewrite a polynomial in terms of elementary symmetric polynomials.
`sympify`(a[, locals, convert_xor, strict, ...])	Converts an arbitrary expression to a type that can be used inside SymPy.
`take`(iter, n)	Return `n` items from `iter` iterator.
`tensorcontraction`(array, *contraction_axes)	Contraction of an array-like object on the specified axes.
`tensordiagonal`(array, *diagonal_axes)	Diagonalization of an array-like object on the specified axes.
`tensorproduct`(*args)	Tensor product among scalars or array-like objects.
`terms_gcd`(f, *gens, **args)	Remove GCD of terms from `f`.
`textplot`(expr, a, b[, W, H])	Print a crude ASCII art plot of the SymPy expression 'expr' (which should contain a single symbol, e.g. x or something else) over the interval [a, b].
`threaded`(func)	Apply `func` to sub--elements of an object, including `Add`.
`timed`(func[, setup, limit])	Adaptively measure execution time of a function.
`to_cnf`(expr[, simplify, force])	Convert a propositional logical sentence `expr` to conjunctive normal form: `((A | ~B | ...) & (B | C | ...) & ...)`.
`to_dnf`(expr[, simplify, force])	Convert a propositional logical sentence `expr` to disjunctive normal form: `((A & ~B & ...) | (B & C & ...) | ...)`.
`to_nnf`(expr[, simplify])	Converts `expr` to Negation Normal Form (NNF).
`to_number_field`(extension[, theta, gen, alias])	Express one algebraic number in the field generated by another.
`together`(expr[, deep, fraction])	Denest and combine rational expressions using symbolic methods.
`topological_sort`(graph[, key])	Topological sort of graph's vertices.
`total_degree`(f, *gens)	Return the total_degree of `f` in the given variables.
`trace`(expr)	Trace of a Matrix.
`trailing`(n)	Count the number of trailing zero digits in the binary representation of n, i.e. determine the largest power of 2 that divides n.
`trigsimp`(expr[, inverse])	Returns a reduced expression by using known trig identities.
`trunc`(f, p, *gens, **args)	Reduce `f` modulo a constant `p`.
`unbranched_argument`(arg)	Returns periodic argument of arg with period as infinity.
`unflatten`(iter[, n])	Group `iter` into tuples of length `n`.

continues on next page

Table 60 – continued from previous page

`unpolarify`(eq[, subs, exponents_only])	If p denotes the projection from the Riemann surface of the logarithm to the complex line, return a simplified version eq' of eq such that $p(eq') = p(eq)$.
`use`(expr, func[, level, args, kwargs])	Use `func` to transform `expr` at the given level.
`var`(names, **args)	Create symbols and inject them into the global namespace.
`variations`(seq, n[, repetition])	Returns an iterator over the n-sized variations of `seq` (size N).
`vfield`(symbols, domain[, order])	Construct new rational function field and inject generators into global namespace.
`viete`(f[, roots])	Generate Viete's formulas for `f`.
`vring`(symbols, domain[, order])	Construct a polynomial ring and inject `x_1, ..., x_n` into the global namespace.
`wronskian`(functions, var[, method])	Compute Wronskian for [] of functions
`xfield`(symbols, domain[, order])	Construct new rational function field returning (field, (x1, ..., xn)).
`xring`(symbols, domain[, order])	Construct a polynomial ring returning `(ring, (x_1, ..., x_n))`.
`xthreaded`(func)	Apply `func` to sub--elements of an object, excluding `Add`.
`zeros`(*args, **kwargs)	Returns a matrix of zeros with `rows` rows and `cols` columns; if `cols` is omitted a square matrix will be returned.

Classes

`Abs`(arg)	Return the absolute value of the argument.
`AccumBounds`	alias of `AccumulationBounds`
`Add`(*args[, evaluate, _sympify])	Expression representing addition operation for algebraic group.
`Adjoint`(*args, **kwargs)	The Hermitian adjoint of a matrix expression.
`AlgebraicField`(dom, *ext[, alias])	Algebraic number field QQ(a)
`AlgebraicNumber`(expr[, coeffs, alias])	Class for representing algebraic numbers in SymPy.
`And`(*args)	Logical AND function.
`AppliedPredicate`(predicate, *args)	The class of expressions resulting from applying `Predicate` to the arguments.
`Array`	alias of `ImmutableDenseNDimArray`
`AssumptionsContext`	Set containing default assumptions which are applied to the `ask()` function.
`Atom`(*args)	A parent class for atomic things.
`AtomicExpr`(*args)	A parent class for object which are both atoms and Exprs.
`AutoSympy`(model)	
`Basic`(*args)	Base class for all SymPy objects.
`BlockDiagMatrix`(*mats)	A sparse matrix with block matrices along its diagonals
`BlockMatrix`(*args, **kwargs)	A BlockMatrix is a Matrix comprised of other matrices.
`CRootOf`	alias of `ComplexRootOf`
`Chi`(z)	Cosh integral.
`Ci`(z)	Cosine integral.
`Circle`(*args, **kwargs)	A circle in space.

continues on next page

Table 61 – continued from previous page

`Complement`(a, b[, evaluate])	Represents the set difference or relative complement of a set with another set.
`ComplexField`([prec, dps, tol])	Complex numbers up to the given precision.
`ComplexRegion`(sets[, polar])	Represents the Set of all Complex Numbers.
`ComplexRootOf`(f, x[, index, radicals, expand])	Represents an indexed complex root of a polynomial.
`ConditionSet`(sym, condition[, base_set])	Set of elements which satisfies a given condition.
`Contains`(x, s)	Asserts that x is an element of the set S.
`CoordMap`(var_vector, eq_duals, ineq_duals, ...)	
`CosineTransform`(*args)	Class representing unevaluated cosine transforms.
`Curve`(function, limits)	A curve in space.
`DeferredVector`(name, **assumptions)	A vector whose components are deferred (e.g. for use with lambdify).
`DenseNDimArray`(*args, **kwargs)	
`Derivative`(expr, *variables, **kwargs)	Carries out differentiation of the given expression with respect to symbols.
`Determinant`(mat)	Matrix Determinant
`DiagMatrix`(vector)	Turn a vector into a diagonal matrix.
`DiagonalMatrix`(*args, **kwargs)	DiagonalMatrix(M) will create a matrix expression that behaves as though all off-diagonal elements, *M[i, j]* where *i != j*, are zero.
`DiagonalOf`(*args, **kwargs)	DiagonalOf(M) will create a matrix expression that is equivalent to the diagonal of *M*, represented as a single column matrix.
`Dict`(*args)	Wrapper around the builtin dict object.
`DifferentialMapping`(US, coord2item, ...)	
`DiracDelta`(arg[, k])	The DiracDelta function and its derivatives.
`DisjointUnion`(*sets)	Represents the disjoint union (also known as the external disjoint union) of a finite number of sets.
`Domain`()	Superclass for all domains in the polys domains system.
`DotProduct`(arg1, arg2)	Dot product of vector matrices
`Dummy`([name, dummy_index])	Dummy symbols are each unique, even if they have the same name:
`EPath`(path)	Manipulate expressions using paths.
`Ei`(z)	The classical exponential integral.
`Ellipse`([center, hradius, vradius, eccentricity])	An elliptical GeometryEntity.
`Eq`	alias of `Equality`
`Equality`(lhs, rhs, **options)	An equal relation between two objects.
`Equivalent`(*args)	Equivalence relation.
`Expr`(*args)	Base class for algebraic expressions.
`ExpressionDomain`()	A class for arbitrary expressions.
`FF`	alias of `FiniteField`
`FF_gmpy`	alias of `GMPYFiniteField`
`FF_python`	alias of `PythonFiniteField`
`FallingFactorial`(x, k)	Falling factorial (related to rising factorial) is a double valued function arising in concrete mathematics, hypergeometric functions and series expansions.
`FiniteField`(mod[, symmetric])	Finite field of prime order GF(p)
`FiniteSet`(*args, **kwargs)	Represents a finite set of Sympy expressions.
`Float`(num[, dps, precision])	Represent a floating-point number of arbitrary precision.

continues on next page

Table 61 – continued from previous page

FourierTransform(*args)	Class representing unevaluated Fourier transforms.
FractionField(domain_or_field[, symbols, order])	A class for representing multivariate rational function fields.
Function(*args)	Base class for applied mathematical functions.
FunctionClass(*args, **kwargs)	Base class for function classes.
FunctionMatrix(rows, cols, lamda)	Represents a matrix using a function (Lambda) which gives outputs according to the coordinates of each matrix entries.
GF	alias of FiniteField
GMPYFiniteField(mod[, symmetric])	Finite field based on GMPY integers.
GMPYIntegerRing()	Integer ring based on GMPY's mpz type.
GMPYRationalField()	Rational field based on GMPY's mpq type.
Ge	alias of GreaterThan
GreaterThan(lhs, rhs, **options)	Class representations of inequalities.
GroebnerBasis(F, *gens, **args)	Represents a reduced Groebner basis.
Gt	alias of StrictGreaterThan
HadamardPower(base, exp)	Elementwise power of matrix expressions
HadamardProduct(*args[, evaluate, check])	Elementwise product of matrix expressions
HankelTransform(*args)	Class representing unevaluated Hankel transforms.
Heaviside(arg[, H0])	Heaviside step function.
ITE(*args)	If-then-else clause.
Identity(n)	The Matrix Identity I - multiplicative identity
Idx(label[, range])	Represents an integer index as an Integer or integer expression.
ImageSet(flambda, *sets)	Image of a set under a mathematical function.
ImmutableDenseMatrix(*args, **kwargs)	Create an immutable version of a matrix.
ImmutableDenseNDimArray(iterable[, shape])	
ImmutableMatrix	alias of ImmutableDenseMatrix
ImmutableSparseMatrix(*args, **kwargs)	Create an immutable version of a sparse matrix.
ImmutableSparseNDimArray([iterable, shape])	
Implies(*args)	Logical implication.
Indexed(base, *args, **kw_args)	Represents a mathematical object with indices.
IndexedBase(label[, shape, offset, strides])	Represent the base or stem of an indexed object
Integer(i)	Represents integer numbers of any size.
IntegerRing()	The domain ZZ representing the integers *mathbb{Z}*.
Integral(function, *symbols, **assumptions)	Represents unevaluated integral.
Intersection(*args, **kwargs)	Represents an intersection of sets as a Set.
Interval(start, end[, left_open, right_open])	Represents a real interval as a Set.
Inverse(mat[, exp])	The multiplicative inverse of a matrix expression
InverseCosineTransform(*args)	Class representing unevaluated inverse cosine transforms.
InverseFourierTransform(*args)	Class representing unevaluated inverse Fourier transforms.
InverseHankelTransform(*args)	Class representing unevaluated inverse Hankel transforms.
InverseLaplaceTransform(*args)	Class representing unevaluated inverse Laplace transforms.
InverseMellinTransform(*args)	Class representing unevaluated inverse Mellin transforms.
InverseSineTransform(*args)	Class representing unevaluated inverse sine transforms.

continues on next page

Table 61 – continued from previous page

`KroneckerDelta`(i, j[, delta_range])	The discrete, or Kronecker, delta function.
`KroneckerProduct`(*args[, check])	The Kronecker product of two or more arguments.
`Lambda`(signature, expr)	Lambda(x, expr) represents a lambda function similar to Python's 'lambda x: expr'.
`LambertW`(x[, k])	The Lambert W function $W(z)$ is defined as the inverse function of $w exp(w)$ [1]_.
`LaplaceTransform`(*args)	Class representing unevaluated Laplace transforms.
`Le`	alias of `LessThan`
`LessThan`(lhs, rhs, **options)	Class representations of inequalities.
`LeviCivita`(*args)	Represent the Levi-Civita symbol.
`Li`(z)	The offset logarithmic integral.
`Limit`(e, z, z0[, dir])	Represents an unevaluated limit.
`Line`(*args, **kwargs)	An infinite line in space.
`Line2D`(p1[, pt, slope])	An infinite line in space 2D.
`Line3D`(p1[, pt, direction_ratio])	An infinite 3D line in space.
`Lt`	alias of `StrictLessThan`
`MatAdd`(*args[, evaluate, check, _sympify])	A Sum of Matrix Expressions
`MatMul`(*args[, evaluate, check, _sympify])	A product of matrix expressions
`MatPow`(base, exp[, evaluate])	
`Matrix`	alias of `MutableDenseMatrix`
`MatrixBase`()	Base class for matrix objects.
`MatrixExpr`(*args, **kwargs)	Superclass for Matrix Expressions
`MatrixPermute`(mat, perm[, axis])	Symbolic representation for permuting matrix rows or columns.
`MatrixSlice`(parent, rowslice, colslice)	A MatrixSlice of a Matrix Expression
`MatrixSymbol`(name, n, m)	Symbolic representation of a Matrix object
`Max`(*args)	Return, if possible, the maximum value of the list.
`MellinTransform`(*args)	Class representing unevaluated Mellin transforms.
`Min`(*args)	Return, if possible, the minimum value of the list.
`Mod`(p, q)	Represents a modulo operation on symbolic expressions.
`Monomial`(monom[, gens])	Class representing a monomial, i.e. a product of powers.
`Mul`(*args[, evaluate, _sympify])	Expression representing multiplication operation for algebraic field.
`MutableDenseMatrix`(*args, **kwargs)	
`MutableDenseNDimArray`([iterable, shape])	
`MutableMatrix`	alias of `MutableDenseMatrix`
`MutableSparseMatrix`(*args, **kwargs)	
`MutableSparseNDimArray`([iterable, shape])	
`NDimArray`(iterable[, shape])	N-dimensional array.
`Nand`(*args)	Logical NAND function.
`Ne`	alias of `Unequality`
`Nor`(*args)	Logical NOR function.
`Not`(arg)	Logical Not function (negation)
`Number`(*obj)	Represents atomic numbers in SymPy.
`NumberSymbol`()	
`O`	alias of `Order`

continues on next page

Table 61 – continued from previous page

OmegaPower(a, b)	Represents ordinal exponential and multiplication terms one of the building blocks of the Ordinal class.
OneMatrix(m, n[, evaluate])	Matrix whose all entries are ones.
Options(gens, args[, flags, strict])	Options manager for polynomial manipulation module.
Or(*args)	Logical OR function
Order(expr, *args, **kwargs)	Represents the limiting behavior of some function.
Ordinal(*terms)	Represents ordinals in Cantor normal form.
Parabola([focus, directrix])	A parabolic GeometryEntity.
Permanent(mat)	Matrix Permanent
PermutationMatrix(perm)	A Permutation Matrix
Piecewise(*_args)	Represents a piecewise function.
Plane(p1[, a, b])	A plane is a flat, two-dimensional surface.
Point(*args, **kwargs)	A point in a n-dimensional Euclidean space.
Point2D(*args[, _nocheck])	A point in a 2-dimensional Euclidean space.
Point3D(*args[, _nocheck])	A point in a 3-dimensional Euclidean space.
Poly(rep, *gens, **args)	Generic class for representing and operating on polynomial expressions.
Polygon(*args[, n])	A two-dimensional polygon.
PolynomialRing(domain_or_ring[, symbols, order])	A class for representing multivariate polynomial rings.
Pow(b, e[, evaluate])	Defines the expression x**y as "x raised to a power y"
PowerSet(arg[, evaluate])	A symbolic object representing a power set.
Predicate(*args, **kwargs)	Base class for mathematical predicates.
Product(function, *symbols, **assumptions)	Represents unevaluated products.
ProductSet(*sets, **assumptions)	Represents a Cartesian Product of Sets.
PurePoly(rep, *gens, **args)	Class for representing pure polynomials.
PythonFiniteField(mod[, symmetric])	Finite field based on Python's integers.
PythonIntegerRing()	Integer ring based on Python's int type.
PythonRational	alias of PythonMPQ
QQ_gmpy	alias of GMPYRationalField
QQ_python	alias of PythonRationalField
Quaternion([a, b, c, d, real_field, norm])	Provides basic quaternion operations.
Range(*args)	Represents a range of integers.
Rational(p[, q, gcd])	Represents rational numbers (p/q) of any size.
RationalField()	Abstract base class for the domain QQ.
Ray(p1[, p2])	A Ray is a semi-line in the space with a source point and a direction.
Ray2D(p1[, pt, angle])	A Ray is a semi-line in the space with a source point and a direction.
Ray3D(p1[, pt, direction_ratio])	A Ray is a semi-line in the space with a source point and a direction.
RealField([prec, dps, tol])	Real numbers up to the given precision.
RealNumber	alias of Float
RegularPolygon(c, r, n[, rot])	A regular polygon.
Rel	alias of Relational
Rem(p, q)	Returns the remainder when p is divided by q where p is finite and q is not equal to zero.
RisingFactorial(x, k)	Rising factorial (also called Pochhammer symbol [1]_) is a double valued function arising in concrete mathematics, hypergeometric functions and series expansions.
RootOf(f, x[, index, radicals, expand])	Represents a root of a univariate polynomial.
RootSum(expr[, func, x, auto, quadratic])	Represents a sum of all roots of a univariate polynomial.
Segment(p1, p2, **kwargs)	A line segment in space.

continues on next page

Table 61 – continued from previous page

Segment2D(p1, p2, **kwargs)	A line segment in 2D space.
Segment3D(p1, p2, **kwargs)	A line segment in a 3D space.
SensitivityMatrix(sympification, duals, ...)	
SeqAdd(*args, **kwargs)	Represents term-wise addition of sequences.
SeqFormula(formula[, limits])	Represents sequence based on a formula.
SeqMul(*args, **kwargs)	Represents term-wise multiplication of sequences.
SeqPer(periodical[, limits])	Represents a periodic sequence.
Set(*args)	The base class for any kind of set.
Shi(z)	Sinh integral.
Si(z)	Sine integral.
Sieve()	An infinite list of prime numbers, implemented as a dynamically growing sieve of Eratosthenes.
SineTransform(*args)	Class representing unevaluated sine transforms.
SingularityFunction(variable, offset, exponent)	Singularity functions are a class of discontinuous functions.
SparseMatrix	alias of MutableSparseMatrix
SparseNDimArray(*args, **kwargs)	
StrPrinter([settings])	
StrictGreaterThan(lhs, rhs, **options)	Class representations of inequalities.
StrictLessThan(lhs, rhs, **options)	Class representations of inequalities.
Subs(expr, variables, point, **assumptions)	Represents unevaluated substitutions of an expression.
Sum(function, *symbols, **assumptions)	Represents unevaluated summation.
Symbol(name, **assumptions)	Assumptions:
SymmetricDifference(a, b[, evaluate])	Represents the set of elements which are in either of the sets and not in their intersection.
TableForm(data, **kwarg)	Create a nice table representation of data.
TestBabyModel(*args, **kwds)	
Trace(mat)	Matrix Trace
Transpose(*args, **kwargs)	The transpose of a matrix expression.
Triangle(*args, **kwargs)	A polygon with three vertices and three sides.
Tuple(*args, **kwargs)	Wrapper around the builtin tuple object.
Unequality(lhs, rhs, **options)	An unequal relation between two objects.
UnevaluatedExpr(arg, **kwargs)	Expression that is not evaluated unless released.
Union(*args, **kwargs)	Represents a union of sets as a Set.
Wild(name[, exclude, properties])	A Wild symbol matches anything, or anything without whatever is explicitly excluded.
WildFunction(*args)	A WildFunction function matches any function (with its arguments).
Xor(*args)	Logical XOR (exclusive OR) function.
Ynm(n, m, theta, phi)	Spherical harmonics defined as
ZZ_gmpy	alias of GMPYIntegerRing
ZZ_python	alias of PythonIntegerRing
ZeroMatrix(m, n)	The Matrix Zero 0 - additive identity
Znm(n, m, theta, phi)	Real spherical harmonics defined as
acos(arg)	The inverse cosine function.
acosh(arg)	acosh(x) is the inverse hyperbolic cosine of x.
acot(arg)	The inverse cotangent function.
acoth(arg)	acoth(x) is the inverse hyperbolic cotangent of x.

continues on next page

Table 61 – continued from previous page

acsc(arg)	The inverse cosecant function.
acsch(arg)	acsch(x) is the inverse hyperbolic cosecant of x.
adjoint(arg)	Conjugate transpose or Hermite conjugation.
airyai(arg)	The Airy function $operatorname{Ai}$ of the first kind.
airyaiprime(arg)	The derivative $operatorname{Ai}^prime$ of the Airy function of the first kind.
airybi(arg)	The Airy function $operatorname{Bi}$ of the second kind.
airybiprime(arg)	The derivative $operatorname{Bi}^prime$ of the Airy function of the first kind.
andre(n)	Andre numbers / Andre function
appellf1(a, b1, b2, c, x, y)	This is the Appell hypergeometric function of two variables as:
arg(arg)	Returns the argument (in radians) of a complex number.
asec(arg)	The inverse secant function.
asech(arg)	asech(x) is the inverse hyperbolic secant of x.
asin(arg)	The inverse sine function.
asinh(arg)	asinh(x) is the inverse hyperbolic sine of x.
assoc_laguerre(n, alpha, x)	Returns the nth generalized Laguerre polynomial in x, $L_n(x)$.
assoc_legendre(n, m, x)	assoc_legendre(n, m, x) gives $P_n^m(x)$, where n and m are the degree and order or an expression which is related to the nth order Legendre polynomial, $P_n(x)$ in the following manner:
atan(arg)	The inverse tangent function.
atan2(y, x)	The function atan2(y, x) computes *operatorname{atan}(y/x)* taking two arguments *y* and *x*.
atanh(arg)	atanh(x) is the inverse hyperbolic tangent of x.
bell(n[, k_sym, symbols])	Bell numbers / Bell polynomials
bernoulli(n[, x])	Bernoulli numbers / Bernoulli polynomials / Bernoulli function
besseli(nu, z)	Modified Bessel function of the first kind.
besselj(nu, z)	Bessel function of the first kind.
besselk(nu, z)	Modified Bessel function of the second kind.
bessely(nu, z)	Bessel function of the second kind.
beta(x[, y])	The beta integral is called the Eulerian integral of the first kind by Legendre:
betainc(*args)	The Generalized Incomplete Beta function is defined as
betainc_regularized(*args)	The Generalized Regularized Incomplete Beta function is given by
binomial(n, k)	Implementation of the binomial coefficient.
carmichael(*args)	Carmichael Numbers:
cartes	alias of product
catalan(n)	Catalan numbers
ceiling(arg)	Ceiling is a univariate function which returns the smallest integer value not less than its argument.
chebyshevt(n, x)	Chebyshev polynomial of the first kind, $T_n(x)$.
chebyshevt_root(n, k)	chebyshev_root(n, k) returns the kth root (indexed from zero) of the nth Chebyshev polynomial of the first kind; that is, if $0 le k < n$, chebyshevt(n, chebyshevt_root(n, k)) == 0.
chebyshevu(n, x)	Chebyshev polynomial of the second kind, $U_n(x)$.

continues on next page

Table 61 – continued from previous page

`chebyshevu_root`(n, k)	`chebyshevu_root(n, k)` returns the kth root (indexed from zero) of the nth Chebyshev polynomial of the second kind; that is, if $0 le k < n$, `chebyshevu(n, chebyshevu_root(n, k)) == 0`.
`conjugate`(arg)	Returns the *complex conjugate* [1]_ of an argument.
`cos`(arg)	The cosine function.
`cosh`(arg)	`cosh(x)` is the hyperbolic cosine of `x`.
`cot`(arg)	The cotangent function.
`coth`(arg)	`coth(x)` is the hyperbolic cotangent of `x`.
`csc`(arg)	The cosecant function.
`csch`(arg)	`csch(x)` is the hyperbolic cosecant of `x`.
`defaultdict`	defaultdict(default_factory=None, /, [...]) --> dict with default factory
`digamma`(z)	The `digamma` function is the first derivative of the `loggamma` function
`dirichlet_eta`(s[, a])	Dirichlet eta function.
`divisor_sigma`(n[, k])	Calculate the divisor function *sigma_k(n)* for positive integer n
`elliptic_e`(m[, z])	Called with two arguments z and m, evaluates the incomplete elliptic integral of the second kind, defined by
`elliptic_f`(z, m)	The Legendre incomplete elliptic integral of the first kind, defined by
`elliptic_k`(m)	The complete elliptic integral of the first kind, defined by
`elliptic_pi`(n, m[, z])	Called with three arguments n, z and m, evaluates the Legendre incomplete elliptic integral of the third kind, defined by
`erf`(arg)	The Gauss error function.
`erf2`(x, y)	Two-argument error function.
`erf2inv`(x, y)	Two-argument Inverse error function.
`erfc`(arg)	Complementary Error Function.
`erfcinv`(z)	Inverse Complementary Error Function.
`erfi`(z)	Imaginary error function.
`erfinv`(z)	Inverse Error Function.
`euler`(n[, x])	Euler numbers / Euler polynomials / Euler function
`exp`(arg)	The exponential function, e^x.
`exp_polar`(*args)	Represent a *polar number* (see g-function Sphinx documentation).
`expint`(nu, z)	Generalized exponential integral.
`factorial`(n)	Implementation of factorial function over nonnegative integers.
`factorial2`(arg)	The double factorial *n!!*, not to be confused with *(n!)!*
`ff`	alias of `FallingFactorial`
`fibonacci`(n[, sym])	Fibonacci numbers / Fibonacci polynomials
`floor`(arg)	Floor is a univariate function which returns the largest integer value not greater than its argument.
`frac`(arg)	Represents the fractional part of x
`fresnelc`(z)	Fresnel integral C.
`fresnels`(z)	Fresnel integral S.
`gamma`(arg)	The gamma function
`gegenbauer`(n, a, x)	Gegenbauer polynomial $C_n^{left(alpharight)}(x)$.

continues on next page

Table 61 – continued from previous page

genocchi(n[, x])	Genocchi numbers / Genocchi polynomials / Genocchi function
hankel1(nu, z)	Hankel function of the first kind.
hankel2(nu, z)	Hankel function of the second kind.
harmonic(n[, m])	Harmonic numbers
hermite(n, x)	`hermite(n, x)` gives the nth Hermite polynomial in x, $H_n(x)$.
hermite_prob(n, x)	`hermite_prob(n, x)` gives the nth probabilist's Hermite polynomial in x, $He_n(x)$.
hn1(nu, z)	Spherical Hankel function of the first kind.
hn2(nu, z)	Spherical Hankel function of the second kind.
hyper(ap, bq, z)	The generalized hypergeometric function is defined by a series where the ratios of successive terms are a rational function of the summation index.
im(arg)	Returns imaginary part of expression.
jacobi(n, a, b, x)	Jacobi polynomial $P_n^{left(alpha, betaright)}(x)$.
jn(nu, z)	Spherical Bessel function of the first kind.
laguerre(n, x)	Returns the nth Laguerre polynomial in x, $L_n(x)$.
legendre(n, x)	`legendre(n, x)` gives the nth Legendre polynomial of x, $P_n(x)$
lerchphi(*args)	Lerch transcendent (Lerch phi function).
li(z)	The classical logarithmic integral.
ln	alias of `log`
log(arg[, base])	The natural logarithm function *ln(x)* or *log(x)*.
loggamma(z)	The `loggamma` function implements the logarithm of the gamma function (i.e., $logGamma(x)$).
lowergamma(a, x)	The lower incomplete gamma function.
lucas(n)	Lucas numbers
marcumq(m, a, b)	The Marcum Q-function.
mathieuc(a, q, z)	The Mathieu Cosine function $C(a,q,z)$.
mathieucprime(a, q, z)	The derivative $C^{prime}(a,q,z)$ of the Mathieu Cosine function.
mathieus(a, q, z)	The Mathieu Sine function $S(a,q,z)$.
mathieusprime(a, q, z)	The derivative $S^{prime}(a,q,z)$ of the Mathieu Sine function.
meijerg(*args)	The Meijer G-function is defined by a Mellin-Barnes type integral that resembles an inverse Mellin transform.
mobius(n)	Mobius function maps natural number to {-1, 0, 1}
motzkin(n)	The nth Motzkin number is the number
multigamma(x, p)	The multivariate gamma function is a generalization of the gamma function
partition(n)	Partition numbers
periodic_argument(ar, period)	Represent the argument on a quotient of the Riemann surface of the logarithm.
polar_lift(arg)	Lift argument to the Riemann surface of the logarithm, using the standard branch.
polygamma(n, z)	The function `polygamma(n, z)` returns `log(gamma(z)).diff(n + 1)`.
polylog(s, z)	Polylogarithm function.
preorder_traversal(node[, keys])	Do a pre-order traversal of a tree.

continues on next page

Table 61 – continued from previous page

`primenu`(n)	Calculate the number of distinct prime factors for a positive integer n.
`primeomega`(n)	Calculate the number of prime factors counting multiplicities for a positive integer n.
`primepi`(n)	Represents the prime counting function pi(n) = the number of prime numbers less than or equal to n.
`principal_branch`(x, period)	Represent a polar number reduced to its principal branch on a quotient of the Riemann surface of the logarithm.
`re`(arg)	Returns real part of expression.
`reduced_totient`(n)	Calculate the Carmichael reduced totient function lambda(n)
`rf`	alias of `RisingFactorial`
`riemann_xi`(s)	Riemann Xi function.
`sec`(arg)	The secant function.
`sech`(arg)	`sech(x)` is the hyperbolic secant of `x`.
`sign`(arg)	Returns the complex sign of an expression:
`sin`(arg)	The sine function.
`sinc`(arg)	Represents an unnormalized sinc function:
`sinh`(arg)	`sinh(x)` is the hyperbolic sine of `x`.
`stieltjes`(n[, a])	Represents Stieltjes constants, $gamma_{k}$ that occur in Laurent Series expansion of the Riemann zeta function.
`subfactorial`(arg)	The subfactorial counts the derangements of n items and is defined for non-negative integers as:
`tan`(arg)	The tangent function.
`tanh`(arg)	`tanh(x)` is the hyperbolic tangent of `x`.
`totient`(n)	Calculate the Euler totient function phi(n)
`transpose`(arg)	Linear map transposition.
`tribonacci`(n[, sym])	Tribonacci numbers / Tribonacci polynomials
`trigamma`(z)	The `trigamma` function is the second derivative of the `loggamma` function
`uppergamma`(a, z)	The upper incomplete gamma function.
`vectorize`(*mdargs)	Generalizes a function taking scalars to accept multidimensional arguments.
`yn`(nu, z)	Spherical Bessel function of the second kind.
`zeta`(s[, a])	Hurwitz zeta function (or Riemann zeta function).

Exceptions

`BasePolynomialError`	Base class for polynomial related exceptions.
`CoercionFailed`	
`ComputationFailed`(func, nargs, exc)	
`DomainError`	
`EvaluationFailed`	
`ExactQuotientFailed`(f, g[, dom])	

continues on next page

Table 62 – continued from previous page

`ExtraneousFactors`	
`FlagError`	
`GeneratorsError`	
`GeneratorsNeeded`	
`GeometryError`	An exception raised by classes in the geometry module.
`HeuristicGCDFailed`	
`HomomorphismFailed`	
`IsomorphismFailed`	
`MultivariatePolynomialError`	
`NonSquareMatrixError`	
`NotAlgebraic`	
`NotInvertible`	
`NotReversible`	
`OperationNotSupported`(poly, func)	
`OptionError`	
`PoleError`	
`PolificationFailed`(opt, origs, exprs[, seq])	
`PolynomialDivisionFailed`(f, g, domain)	
`PolynomialError`	
`PrecisionExhausted`	
`RefinementFailed`	
`ShapeError`	Wrong matrix shape
`SympifyError`(expr[, base_exc])	
`UnificationFailed`	
`UnivariatePolynomialError`	

class `src.sensitivity.babymodel.`**`TestBabyModel`**(**args*, ***kwds*)

`_active`

`solve()`

`src.sensitivity.babymodel.`**`connect_regions`**`(`*region_map*, *hubmap*, *base_trans_cap*`)`

given a mapping of regions to lists of hubs, and hubs to regions, creates a set of arcs between hubs such that:

1. the grid is connected
2. each region has a main hub that all other hubs in the region are connected to

Parameters

- **`region_map`** (*`dict`*) – dictionary of region names to lists of hubs
- **`hubmap`** (*`dict`*) – dictionary of hub names to their parent region
- **`base_trans_cap`** (*`float`*) – base transportation capacity for arcs

Returns

- **arcs** (*list*) – list of tuples of hubs, representing start and endpoints
- **outbound** (*dict*) – dictionary of hub:list of arcs originating from hub
- **inbound** (*dict*) – dictionary of hub:list of arcs terminating at hub
- **trans_capacity** (*dict*) – dictionary of arcs:transportation capacity of arc

`src.sensitivity.babymodel.`**`generate`**`(`*num_regions=3*, *hubs_per_region=2*, *base_elec_price=5.0*, *base_prod_capacity=5000*, *demand_fraction=0.7*, *base_transport_cost=22.3*, *base_elec_consumption=9.8*`)`

generates a random network with all parameters required to initialize a ToyBabyModel

Parameters

- **`num_regions`** (*`int, optional`*) – number of regions. Defaults to 3.
- **`hubs_per_region`** (*`int, optional`*) – number of hubs per region. Defaults to 2.
- **`base_elec_price`** (*`float, optional`*) – the average electricity price in all regions. Defaults to 5.0.
- **`base_prod_capacity`** (*`int, optional`*) – the average production capacity for all hubs. Defaults to 5000.
- **`demand_fraction`** (*`float, optional`*) – the average fraction of capacity initial demand is set to. Defaults to 0.7.
- **`base_transport_cost`** (*`float, optional`*) – the base transportation cost for all arcs. Defaults to 22.3.
- **`base_elec_consumption`** (*`float, optional`*) – the electricity consumption rate for production. Defaults to 9.8.

Returns

- **hublist** (*list*)
- **region_list** (*list*)
- **hub_map** (*dict*)
- **region_map** (*dict*)
- **elec_price** (*dict*)
- **prod_capacity** (*dict*)

- **demand** (*dict*)
- **base_elec_consumption** (*float*)
- **base_transport_cost** (*float*)

src.sensitivity.faster_sensitivity

faster_sensitivity

This file contains the class SensitivityMatrix which takes in sympy objects that have been converted from pyomo. It builds the matrix of partials to be used in sensitivity analysis.

It also contains class AutoSympy which takes in pyomo models and converts the objects into sympy.

Finally, it contains class toy_model, the sensitivity method in action and then runs toy_model with input n=5.

The file babymodel.py can be also use this method by importing this file instead of sensitivity_tools.py by:
from faster_sensitivity import *

Classes

`AutoSympy`(model)	This class take in pyomo models and converts the objects into sympy.
`SensitivityMatrix`(sympification, duals, ...)	This class takes in sympy objects that have been converted from pyomo.
`date`	date(year, month, day) --> date object
`datetime`(year, month, day[, hour[, minute[, ...)	The year, month and day arguments are required.
`time`	time([hour[, minute[, second[, microsecond[, tzinfo]]]]]) --> a time object
`timedelta`	Difference between two datetime values.
`timezone`	Fixed offset from UTC implementation of tzinfo.
`toy_model`(n)	An example of the method in action that scales by the given 'n' value
`tzinfo`	Abstract base class for time zone info objects.

`class src.sensitivity.faster_sensitivity.AutoSympy`(*model*)

This class take in pyomo models and converts the objects into sympy. This is useful for problems that needs methods such as derivatives to be calculated on the equations. We use these derivatives to calculate a sensitivity matrix that estimates the changes in variables due to changes in parameters

`check_complimentarity_all`()

`generate_duals`(*constraints*, *duals*)

Uses dual values and slack values to classify each constraint. It also stores the dual values for substitution later.

Parameters

- **constraints** (*model.component_objects(pyo.Constraint)*) – All of the constraint objects from the pyomo model
- **duals** (*model.dual*) – All of the dual (or Suffix) objects from the pyomo model

Returns
description

Return type
dict of lists and dicts

`get_constraints()`

This function converts all of the constraints in the pyomo object and converts the pyomo expressions into sympy expressions.

Returns
Returns 2 dictionaries: equality_constraints: keys are tuples (constraint_name, index) and values are sympy expressions inequality_constraints: keys are tuples (constraint_name, index) and values are sympy expressions

Return type
dict, dict

`get_objective()`

This converts the pyomo objective function into a sympy function.

Returns
The pyomo objective function converted into sympy

Return type
sympy equation

`get_parameters()`

Convert pyomo parameters into sympy objects. This procedure creates sympy IndexedBase objects and sympy Symbol objects of similar names. The IndexedBase datatype is necessary to parse the equations, but it does not work well with derivatives. We will substitute in Symbols when the equations are all created, so they need to map to each other. To keep the columns in order through all procedures, all parameters are given a unique column number by the variable "position" This position is stored in the class dict param_position_map

Returns
Returns 4 dictionaries: parameters: keys are pyomo parameter names and values are sympy IndexedBase objects with the same name parameter_values: keys are sympy symbols and values are the numerical values of the pyomo objects parameter_index_sets: keys are pyomo parameter names and values are lists of that parameters indices symbol_map: keys are pyomo parameters with an index and values are sympy symbols with a similarly styled name and index

Return type
dict, dict, dict, dict

`get_sensitivity_matrix`(*parameters_of_interest=None*)

This function gathers all of the new sympy objects and creates a SensitivityMatrix object.

Parameters
parameters_of_interest (*dict, optional*) – Specified subset of the parameters if more information is known about needless parameters, by default None

Returns
a SensitivityMatrix object that contains the sensitivity matrix and commands to use it.

Return type
SensitivityMatrix

`get_sets()`

Convert pyomo sets into sympy indexes

Returns
The first dictionary has the pyomo objects' names as keys and newly created sympy indexes as values. The second dictionary has the new sympy indexes as keys and the pyomo sets' values as the dict values.

Return type
dict, dict

`get_variables()`

Convert pyomo variables into sympy objects. This procedure creates sympy IndexedBase objects and sympy Symbol objects of similar names. The IndexedBase datatype is necessary to parse the equations, but it does not work well with derivatives. We will substitute in Symbols when the equations are all created, so they need to map to each other. To keep the columns in order through all procedures, all parameters are given a unique column number by the variable "position" This position is stored in the class dict param_position_map

Returns
Returns 2 dictionaries: variables: keys are pyomo variable names and values are sympy IndexedBase objects with the same name variable values: keys are sympy symbols and values are the numerical values of the pyomo objects It is also worth mentioning that this adds entries to the self.symbol_map in the same way parameters do. Symbol map entries have keys of IndexedBase objects and the values are their associated sympy Symbol

Return type
dict, dict

class `src.sensitivity.faster_sensitivity.SensitivityMatrix`(*sympification*, *duals*, *parameters_of_interest*)

This class takes in sympy objects that have been converted from pyomo. It builds the matrix of partials to be used in sensitivity analysis.

`generate_matrix()`

This creates all of the matrices that will be combined into the U and S matrices.

Returns
Returns 2 dictionaries. The first is the dictionary of matrix components with their names as keys. The second dictionary is a map from the symbols to their values.

Return type
dict, dict

`get_partial`(*x*, *a*)

Retrieve the value of a particular partial derivative. The value retrieved will be dx/da.

Parameters

- **x** (*sp.Symbol*) – The symbol for the variable that you wish to know the change effect
- **a** (*sp.Symbol*) – The symbol for the parameter that you wish to change to cause an effect on a variable

Returns
The value of the partial derivative dx/da

Return type
float

`get_partials_matrix()`

Calculate the matrix of all partials as U^(-1) * S Thus far, this is found to run the fastest when U^(-1) and S are numpy arrays

Returns
Full partials matrix

Return type
np.ndarray

get_sensitivity_range(*x*, *a*, *percent*)

The estimated values for "x" if the parameter "a" changes by percent% (as number 0% to 100%). It will return values for an increase and decrease of the percent given.

Parameters

- **x** (*sp.Symbol*) – The symbol for the variable that you wish to know the change effect
- **a** (*sp.Symbol*) – The symbol for the parameter that you wish to change to cause an effect on a variable
- **percent** (*float*) – A number 0-100 for the percent change in "a"

Returns
Returns the estimated value for "x" if "a" is increased by percent% and decreased by percent%

Return type
float, float

invert_U()

Calculates the inverse of the U matrix. The fastest method found for this so far has been to convert to numpy and use its inverse function

Returns
Calculated matrix for the inverse of U

Return type
np.ndarray

matrix_assembly(*components*, *subs_dict*)

Combines matrix components to create U and S matrices from the literature.

Parameters

- **components** (*dict*) – Dictionary of all precalculated matrix components
- **subs_dict** (*dict*) – Dictionary that maps symbols to their values

Returns
Returns the U and S matrices respectively with all symbols replaced by corresponding values

Return type
sp.Matrix, sp.Matrix

matrix_sub(*M*, *subs*)

A function that substitutes values into a matrix. This is the same result as sp.Matrix().subs(subs). This speeds up runtime by only attempting to substitute values into symbols that actually exist in each cell.

Parameters

- **M** (*sp.Matrix*) – The matrix to have symbols substituted for values
- **subs** (*dict*) – Dictionary of values for sympy symbols

Returns
The original matrix with all given values substituted into their symbols

Return type
sp.Matrix

new_jacobian(*f*, *values*, *map*)

A function that returns the same result as Matrix.jacobian(values). This speeds up runtime by only taking derivatives of symbols that exist. The original function takes the derivative wrt everything in values.

Parameters

- **f** (*sp.Matrix*) – Matrix of equations
- **values** (*list*) – List of symbols that the function will take derivatives with respect to
- **map** (*dict*) – Dictionary of column locations for each symbol

Returns
Returns jacobian of given f matrix

Return type
sp.Matrix

`class src.sensitivity.faster_sensitivity.toy_model(n)`

An example of the method in action that scales by the given 'n' value

`create_model()`

src.sensitivity.sensitivity_tools

Sensitivity Tools This file contains the AutoSympy, SensitivityMatrix, CoordMap, and DifferentialMapping classes. These classes serve as the data structures and containers of methods to get from a Pyomo ConcreteModel to easily accessible sensitivities.

The flow goes:

Pyomo model -> AutoSympy -> SensitivityMatrix -> DifferentialMapping

sensitivities and sensitivity-based calculations can be done through the DifferentialMapping object.

Functions

`E1(z)`	Classical case of the generalized exponential integral.
`Eijk(*args, **kwargs)`	Represent the Levi-Civita symbol.
`GramSchmidt(vlist[, orthonormal])`	Apply the Gram-Schmidt process to a set of vectors.
`LC(f, *gens, **args)`	Return the leading coefficient of `f`.
`LM(f, *gens, **args)`	Return the leading monomial of `f`.
`LT(f, *gens, **args)`	Return the leading term of `f`.
`N(x[, n])`	Calls x.evalf(n, **options).
`POSform(variables, minterms[, dontcares])`	The POSform function uses simplified_pairs and a redundant-group eliminating algorithm to convert the list of all input combinations that generate '1' (the minterms) into the smallest product-of-sums form.
`SOPform(variables, minterms[, dontcares])`	The SOPform function uses simplified_pairs and a redundant group- eliminating algorithm to convert the list of all input combos that generate '1' (the minterms) into the smallest sum-of-products form.
`Ynm_c(n, m, theta, phi)`	Conjugate spherical harmonics defined as
`abundance(n)`	Returns the difference between the sum of the positive proper divisors of a number and the number.
`apart(f[, x, full])`	Compute partial fraction decomposition of a rational function.
`apart_list(f[, x, dummies])`	Compute partial fraction decomposition of a rational function and return the result in structured form.

continues on next page

Table 64 – continued from previous page

`apply_finite_diff`(order, x_list, y_list[, x0])	Calculates the finite difference approximation of the derivative of requested order at `x0` from points provided in `x_list` and `y_list`.
`approximants`(l[, X, simplify])	Return a generator for consecutive Pade approximants for a series.
`are_similar`(e1, e2)	Are two geometrical entities similar.
`arity`(cls)	Return the arity of the function if it is known, else None.
`ask`(proposition[, assumptions, context])	Function to evaluate the proposition with assumptions.
`assemble_partfrac_list`(partial_list)	Reassemble a full partial fraction decomposition from a structured result obtained by the function `apart_list`.
`assuming`(*assumptions)	Context manager for assumptions.
`banded`(*args, **kwargs)	Returns a SparseMatrix from the given dictionary describing the diagonals of the matrix.
`besselsimp`(expr)	Simplify bessel-type functions.
`binomial_coefficients`(n)	Return a dictionary containing pairs $(k1, k2) : C_k n$ where $C_k n$ are binomial coefficients and $n = k1 + k2$.
`binomial_coefficients_list`(n)	Return a list of binomial coefficients as rows of the Pascal's triangle.
`block_collapse`(expr)	Evaluates a block matrix expression
`blockcut`(expr, rowsizes, colsizes)	Cut a matrix expression into Blocks
`bool_map`(bool1, bool2)	Return the simplified version of *bool1*, and the mapping of variables that makes the two expressions *bool1* and *bool2* represent the same logical behaviour for some correspondence between the variables of each.
`bottom_up`(rv, F[, atoms, nonbasic])	Apply `F` to all expressions in an expression tree from the bottom up.
`bspline_basis`(d, knots, n, x)	The n-th B-spline at x of degree d with knots.
`bspline_basis_set`(d, knots, x)	Return the `len(knots)-d-1` B-splines at *x* of degree *d* with *knots*.
`cacheit`(func)	
`cancel`(f, *gens[, _signsimp])	Cancel common factors in a rational function `f`.
`capture`(func)	Return the printed output of func().
`casoratian`(seqs, n[, zero])	Given linear difference operator L of order 'k' and homogeneous equation Ly = 0 we want to compute kernel of L, which is a set of 'k' sequences: a(n), b(n), .
`cbrt`(arg[, evaluate])	Returns the principal cube root.
`ccode`(expr[, assign_to, standard])	Converts an expr to a string of c code
`centroid`(*args)	Find the centroid (center of mass) of the collection containing only Points, Segments or Polygons.
`chebyshevt_poly`(n[, x, polys])	Generates the Chebyshev polynomial of the first kind *T_n(x)*.
`chebyshevu_poly`(n[, x, polys])	Generates the Chebyshev polynomial of the second kind *U_n(x)*.
`check_assumptions`(expr[, against])	Checks whether assumptions of `expr` match the T/F assumptions given (or possessed by `against`).
`checkodesol`(ode, sol[, func, order, ...])	Substitutes `sol` into `ode` and checks that the result is `0`.
`checkpdesol`(pde, sol[, func, solve_for_func])	Checks if the given solution satisfies the partial differential equation.
`checksol`(f, symbol[, sol])	Checks whether sol is a solution of equation f == 0.
`classify_ode`(eq[, func, dict, ics, prep, ...])	Returns a tuple of possible `dsolve()` classifications for an ODE.

continues on next page

Table 64 – continued from previous page

`classify_pde`(eq[, func, dict, prep])	Returns a tuple of possible pdsolve() classifications for a PDE.
`closest_points`(*args)	Return the subset of points from a set of points that were the closest to each other in the 2D plane.
`cofactors`(f, g, *gens, **args)	Compute GCD and cofactors of `f` and `g`.
`collect`(expr, syms[, func, evaluate, exact, ...])	Collect additive terms of an expression.
`collect_const`(expr, *vars[, Numbers])	A non-greedy collection of terms with similar number coefficients in an Add expr.
`combsimp`(expr)	Simplify combinatorial expressions.
`comp`(z1, z2[, tol])	Return a bool indicating whether the error between z1 and z2 is le `tol`.
`compose`(f, g, *gens, **args)	Compute functional composition `f(g)`.
`composite`(nth)	Return the nth composite number, with the composite numbers indexed as composite(1) = 4, composite(2) = 6, etc....
`compositepi`(n)	Return the number of positive composite numbers less than or equal to n.
`construct_domain`(obj, **args)	Construct a minimal domain for a list of expressions.
`content`(f, *gens, **args)	Compute GCD of coefficients of `f`.
`continued_fraction`(a)	Return the continued fraction representation of a Rational or quadratic irrational.
`continued_fraction_convergents`(cf)	Return an iterator over the convergents of a continued fraction (cf).
`continued_fraction_iterator`(x)	Return continued fraction expansion of x as iterator.
`continued_fraction_periodic`(p, q[, d, s])	Find the periodic continued fraction expansion of a quadratic irrational.
`continued_fraction_reduce`(cf)	Reduce a continued fraction to a rational or quadratic irrational.
`convex_hull`(*args[, polygon])	The convex hull surrounding the Points contained in the list of entities.
`convolution`(a, b[, cycle, dps, prime, ...])	Performs convolution by determining the type of desired convolution using hints.
`cosine_transform`(f, x, k, **hints)	Compute the unitary, ordinary-frequency cosine transform of *f*, defined as
`count_ops`(expr[, visual])	Return a representation (integer or expression) of the operations in expr.
`count_roots`(f[, inf, sup])	Return the number of roots of `f` in `[inf, sup]` interval.
`covering_product`(a, b)	Returns the covering product of given sequences.
`cse`(exprs[, symbols, optimizations, ...])	Perform common subexpression elimination on an expression.
`cxxcode`(expr[, assign_to, standard])	C++ equivalent of `ccode()`.
`cycle_length`(f, x0[, nmax, values])	For a given iterated sequence, return a generator that gives the length of the iterated cycle (lambda) and the length of terms before the cycle begins (mu); if `values` is True then the terms of the sequence will be returned instead.
`cyclotomic_poly`(n[, x, polys])	Generates cyclotomic polynomial of order *n* in *x*.
`decompogen`(f, symbol)	Computes General functional decomposition of `f`. Given an expression `f`, returns a list `[f_1, f_2, ..., f_n]`, where:: f = f_1 o f_2 o ... f_n = f_1(f_2(... f_n)).
`decompose`(f, *gens, **args)	Compute functional decomposition of `f`.
`default_sort_key`(item[, order])	Return a key that can be used for sorting.

continues on next page

Table 64 – continued from previous page

`deg`(r)	Return the degree value for the given radians (pi = 180 degrees).
`degree`(f[, gen])	Return the degree of `f` in the given variable.
`degree_list`(f, *gens, **args)	Return a list of degrees of `f` in all variables.
`denom`(expr)	
`derive_by_array`(expr, dx)	Derivative by arrays.
`det`(matexpr)	Matrix Determinant
`det_quick`(M[, method])	Return `det(M)` assuming that either there are lots of zeros or the size of the matrix is small.
`diag`(*values[, strict, unpack])	Returns a matrix with the provided values placed on the diagonal.
`diagonalize_vector`(vector)	
`dict_merge`(*dicts)	Merge dictionaries into a single dictionary.
`diff`(f, *symbols, **kwargs)	Differentiate f with respect to symbols.
`difference_delta`(expr[, n, step])	Difference Operator.
`differentiate_finite`(expr, *symbols[, ...])	Differentiate expr and replace Derivatives with finite differences.
`diophantine`(eq[, param, syms, permute])	Simplify the solution procedure of diophantine equation `eq` by converting it into a product of terms which should equal zero.
`discrete_log`(n, a, b[, order, prime_order])	Compute the discrete logarithm of `a` to the base `b` modulo `n`.
`discriminant`(f, *gens, **args)	Compute discriminant of `f`.
`div`(f, g, *gens, **args)	Compute polynomial division of `f` and `g`.
`divisor_count`(n[, modulus, proper])	Return the number of divisors of `n`.
`divisors`(n[, generator, proper])	Return all divisors of n sorted from 1..n by default.
`dotprint`(expr[, styles, atom, maxdepth, ...])	DOT description of a SymPy expression tree
`dsolve`(eq[, func, hint, simplify, ics, xi, ...])	Solves any (supported) kind of ordinary differential equation and system of ordinary differential equations.
`egyptian_fraction`(r[, algorithm])	Return the list of denominators of an Egyptian fraction expansion **[1]_** of the said rational *r*.
`epath`(path[, expr, func, args, kwargs])	Manipulate parts of an expression selected by a path.
`euler_equations`(L[, funcs, vars])	Find the Euler-Lagrange equations **[1]_** for a given Lagrangian.
`evaluate`(x)	Control automatic evaluation
`expand`(e[, deep, modulus, power_base, ...])	Expand an expression using methods given as hints.
`expand_complex`(expr[, deep])	Wrapper around expand that only uses the complex hint.
`expand_func`(expr[, deep])	Wrapper around expand that only uses the func hint.
`expand_log`(expr[, deep, force, factor])	Wrapper around expand that only uses the log hint.
`expand_mul`(expr[, deep])	Wrapper around expand that only uses the mul hint.
`expand_multinomial`(expr[, deep])	Wrapper around expand that only uses the multinomial hint.
`expand_power_base`(expr[, deep, force])	Wrapper around expand that only uses the power_base hint.
`expand_power_exp`(expr[, deep])	Wrapper around expand that only uses the power_exp hint.
`expand_trig`(expr[, deep])	Wrapper around expand that only uses the trig hint.
`exptrigsimp`(expr)	Simplifies exponential / trigonometric / hyperbolic functions.
`exquo`(f, g, *gens, **args)	Compute polynomial exact quotient of `f` and `g`.

continues on next page

Table 64 – continued from previous page

`eye`(*args, **kwargs)	Create square identity matrix n x n
`factor`(f, *gens[, deep])	Compute the factorization of expression, `f`, into irreducibles.
`factor_list`(f, *gens, **args)	Compute a list of irreducible factors of `f`.
`factor_nc`(expr)	Return the factored form of `expr` while handling non-commutative expressions.
`factor_terms`(expr[, radical, clear, ...])	Remove common factors from terms in all arguments without changing the underlying structure of the expr.
`factorint`(n[, limit, use_trial, use_rho, ...])	Given a positive integer `n`, `factorint(n)` returns a dict containing the prime factors of `n` as keys and their respective multiplicities as values.
`factorrat`(rat[, limit, use_trial, use_rho, ...])	Given a Rational `r`, `factorrat(r)` returns a dict containing the prime factors of `r` as keys and their respective multiplicities as values.
`failing_assumptions`(expr, **assumptions)	Return a dictionary containing assumptions with values not matching those of the passed assumptions.
`farthest_points`(*args)	Return the subset of points from a set of points that were the furthest apart from each other in the 2D plane.
`fcode`(expr[, assign_to])	Converts an expr to a string of fortran code
`fft`(seq[, dps])	Performs the Discrete Fourier Transform (**DFT**) in the complex domain.
`field`(symbols, domain[, order])	Construct new rational function field returning (field, x1, ..., xn).
`field_isomorphism`(a, b, *[, fast])	Find an embedding of one number field into another.
`filldedent`(s[, w])	Strips leading and trailing empty lines from a copy of `s`, then dedents, fills and returns it.
`finite_diff_weights`(order, x_list[, x0])	Calculates the finite difference weights for an arbitrarily spaced one-dimensional grid (`x_list`) for derivatives at `x0` of order 0, 1, ..., up to `order` using a recursive formula.
`flatten`(iterable[, levels, cls])	Recursively denest iterable containers.
`fourier_series`(f[, limits, finite])	Computes the Fourier trigonometric series expansion.
`fourier_transform`(f, x, k, **hints)	Compute the unitary, ordinary-frequency Fourier transform of `f`, defined as
`fps`(f[, x, x0, dir, hyper, order, rational, ...])	Generates Formal Power Series of `f`.
`fraction`(expr[, exact])	Returns a pair with expression's numerator and denominator.
`fu`(rv[, measure])	Attempt to simplify expression by using transformation rules given in the algorithm by Fu et al.
`fwht`(seq)	Performs the Walsh Hadamard Transform (**WHT**), and uses Hadamard ordering for the sequence.
`galois_group`(f, *gens[, by_name, max_tries, ...])	Compute the Galois group for polynomials f up to degree 6.
`gammasimp`(expr)	Simplify expressions with gamma functions.
`gcd`(f[, g])	Compute GCD of `f` and `g`.
`gcd_list`(seq, *gens, **args)	Compute GCD of a list of polynomials.
`gcd_terms`(terms[, isprimitive, clear, fraction])	Compute the GCD of `terms` and put them together.
`gcdex`(f, g, *gens, **args)	Extended Euclidean algorithm of `f` and `g`.
`getLogger`([name])	Return a logger with the specified name, creating it if necessary.
`get_contraction_structure`(expr)	Determine dummy indices of `expr` and describe its structure

continues on next page

Table 64 – continued from previous page

`get_indices`(expr)	Determine the outer indices of expression `expr`
`gff`(f, *gens, **args)	Compute greatest factorial factorization of `f`.
`gff_list`(f, *gens, **args)	Compute a list of greatest factorial factors of `f`.
`glsl_code`(expr[, assign_to])	Converts an expr to a string of GLSL code
`groebner`(F, *gens, **args)	Computes the reduced Groebner basis for a set of polynomials.
`ground_roots`(f, *gens, **args)	Compute roots of `f` by factorization in the ground domain.
`group`(seq[, multiple])	Splits a sequence into a list of lists of equal, adjacent elements.
`gruntz`(e, z, z0[, dir])	Compute the limit of e(z) at the point z0 using the Gruntz algorithm.
`hadamard_product`(*matrices)	Return the elementwise (aka Hadamard) product of matrices.
`half_gcdex`(f, g, *gens, **args)	Half extended Euclidean algorithm of `f` and `g`.
`hankel_transform`(f, r, k, nu, **hints)	Compute the Hankel transform of *f*, defined as
`has_dups`(seq)	Return True if there are any duplicate elements in `seq`.
`has_variety`(seq)	Return True if there are any different elements in `seq`.
`hermite_poly`(n[, x, polys])	Generates the Hermite polynomial *H_n(x)*.
`hermite_prob_poly`(n[, x, polys])	Generates the probabilist's Hermite polynomial *He_n(x)*.
`hessian`(f, varlist[, constraints])	Compute Hessian matrix for a function f wrt parameters in varlist which may be given as a sequence or a row/column vector.
`homogeneous_order`(eq, *symbols)	Returns the order *n* if *g* is homogeneous and `None` if it is not homogeneous.
`horner`(f, *gens, **args)	Rewrite a polynomial in Horner form.
`hyperexpand`(f[, allow_hyper, rewrite, place])	Expand hypergeometric functions.
`hypersimilar`(f, g, k)	Returns True if `f` and `g` are hyper-similar.
`hypersimp`(f, k)	Given combinatorial term f(k) simplify its consecutive term ratio i.e. f(k+1)/f(k).
`idiff`(eq, y, x[, n])	Return `dy/dx` assuming that `eq == 0`.
`ifft`(seq[, dps])	Performs the Discrete Fourier Transform (**DFT**) in the complex domain.
`ifwht`(seq)	Performs the Walsh Hadamard Transform (**WHT**), and uses Hadamard ordering for the sequence.
`igcd`(*args)	Computes nonnegative integer greatest common divisor.
`ilcm`(*args)	Computes integer least common multiple.
`imageset`(*args)	Return an image of the set under transformation `f`.
`init_printing`([pretty_print, order, ...])	Initializes pretty-printer depending on the environment.
`init_session`([ipython, pretty_print, order, ...])	Initialize an embedded IPython or Python session.
`integer_log`(y, x)	Returns `(e, bool)` where e is the largest nonnegative integer such that $\|y\| \geq \|x^e\|$ and `bool` is True if $y = x^e$.
`integer_nthroot`(y, n)	Return a tuple containing x = floor(y**(1/n)) and a boolean indicating whether the result is exact (that is, whether x**n == y).
`integrate`(f, var, ...)	
`interactive_traversal`(expr)	Traverse a tree asking a user which branch to choose.
`interpolate`(data, x)	Construct an interpolating polynomial for the data points evaluated at point x (which can be symbolic or numeric).

continues on next page

Table 64 – continued from previous page

`interpolating_poly`(n, x[, X, Y])	Construct Lagrange interpolating polynomial for `n` data points.
`interpolating_spline`(d, x, X, Y)	Return spline of degree *d*, passing through the given *X* and *Y* values.
`intersecting_product`(a, b)	Returns the intersecting product of given sequences.
`intersection`(*entities[, pairwise])	The intersection of a collection of GeometryEntity instances.
`intervals`(F[, all, eps, inf, sup, strict, ...])	Compute isolating intervals for roots of `f`.
`intt`(seq, prime)	Performs the Number Theoretic Transform (**NTT**), which specializes the Discrete Fourier Transform (**DFT**) over quotient ring *Z/pZ* for prime *p* instead of complex numbers *C*.
`inv_quick`(M)	Return the inverse of `M`, assuming that either there are lots of zeros or the size of the matrix is small.
`inverse_cosine_transform`(F, k, x, **hints)	Compute the unitary, ordinary-frequency inverse cosine transform of *F*, defined as
`inverse_fourier_transform`(F, k, x, **hints)	Compute the unitary, ordinary-frequency inverse Fourier transform of *F*, defined as
`inverse_hankel_transform`(F, k, r, nu, **hints)	Compute the inverse Hankel transform of *F* defined as
`inverse_laplace_transform`(F, s, t[, plane])	Compute the inverse Laplace transform of *F(s)*, defined as
`inverse_mellin_transform`(F, s, x, strip, **hints)	Compute the inverse Mellin transform of *F(s)* over the fundamental strip given by `strip=(a, b)`.
`inverse_mobius_transform`(seq[, subset])	Performs the Mobius Transform for subset lattice with indices of sequence as bitmasks.
`inverse_sine_transform`(F, k, x, **hints)	Compute the unitary, ordinary-frequency inverse sine transform of *F*, defined as
`invert`(f, g, *gens, **args)	Invert `f` modulo `g` when possible.
`is_abundant`(n)	Returns True if `n` is an abundant number, else False.
`is_amicable`(m, n)	Returns True if the numbers *m* and *n* are "amicable", else False.
`is_convex`(f, *syms[, domain])	Determines the convexity of the function passed in the argument.
`is_decreasing`(expression[, interval, symbol])	Return whether the function is decreasing in the given interval.
`is_deficient`(n)	Returns True if `n` is a deficient number, else False.
`is_increasing`(expression[, interval, symbol])	Return whether the function is increasing in the given interval.
`is_mersenne_prime`(n)	Returns True if `n` is a Mersenne prime, else False.
`is_monotonic`(expression[, interval, symbol])	Return whether the function is monotonic in the given interval.
`is_nthpow_residue`(a, n, m)	Returns True if `x**n == a (mod m)` has solutions.
`is_perfect`(n)	Returns True if `n` is a perfect number, else False.
`is_primitive_root`(a, p)	Returns True if `a` is a primitive root of `p`.
`is_quad_residue`(a, p)	Returns True if `a` (mod `p`) is in the set of squares mod `p`, i.e a % p in set([i**2 % p for i in range(p)]).
`is_strictly_decreasing`(expression[, ...])	Return whether the function is strictly decreasing in the given interval.
`is_strictly_increasing`(expression[, ...])	Return whether the function is strictly increasing in the given interval.
`is_zero_dimensional`(F, *gens, **args)	Checks if the ideal generated by a Groebner basis is zero-dimensional.

continues on next page

Table 64 – continued from previous page

`isolate`(alg[, eps, fast])	Find a rational isolating interval for a real algebraic number.
`isprime`(n)	Test if n is a prime number (True) or not (False).
`itermonomials`(variables, max_degrees[, ...])	`max_degrees` and `min_degrees` are either both integers or both lists.
`jacobi_normalized`(n, a, b, x)	Jacobi polynomial $P_n^{left(alpha, betaright)}(x)$.
`jacobi_poly`(n, a, b[, x, polys])	Generates the Jacobi polynomial *P_n^{(a,b)}(x)*.
`jacobi_symbol`(m, n)	Returns the Jacobi symbol *(m / n)*.
`jn_zeros`(n, k[, method, dps])	Zeros of the spherical Bessel function of the first kind.
`jordan_cell`(eigenval, n)	Create a Jordan block:
`jscode`(expr[, assign_to])	Converts an expr to a string of javascript code
`julia_code`(expr[, assign_to])	Converts *expr* to a string of Julia code.
`kronecker_product`(*matrices)	The Kronecker product of two or more arguments.
`kroneckersimp`(expr)	Simplify expressions with KroneckerDelta.
`laguerre_poly`(n[, x, alpha, polys])	Generates the Laguerre polynomial *L_n^{(alpha)}(x)*.
`lambdify`(args, expr[, modules, printer, ...])	Convert a SymPy expression into a function that allows for fast numeric evaluation.
`laplace_transform`(f, t, s[, legacy_matrix])	Compute the Laplace Transform *F(s)* of *f(t)*,
`lcm`(f[, g])	Compute LCM of `f` and `g`.
`lcm_list`(seq, *gens, **args)	Compute LCM of a list of polynomials.
`legendre_poly`(n[, x, polys])	Generates the Legendre polynomial *P_n(x)*.
`legendre_symbol`(a, p)	Returns the Legendre symbol *(a / p)*.
`limit`(e, z, z0[, dir])	Computes the limit of `e(z)` at the point `z0`.
`limit_seq`(expr[, n, trials])	Finds the limit of a sequence as index `n` tends to infinity.
`line_integrate`(field, Curve, variables)	Compute the line integral.
`linear_eq_to_matrix`(equations, *symbols)	Converts a given System of Equations into Matrix form.
`linsolve`(system, *symbols)	Solve system of N linear equations with M variables; both underdetermined and overdetermined systems are supported.
`list2numpy`(l[, dtype])	Converts Python list of SymPy expressions to a NumPy array.
`logcombine`(expr[, force])	Takes logarithms and combines them using the following rules:
`maple_code`(expr[, assign_to])	Converts `expr` to a string of Maple code.
`mathematica_code`(expr, **settings)	Converts an expr to a string of the Wolfram Mathematica code
`matrix2numpy`(m[, dtype])	Converts SymPy's matrix to a NumPy array.
`matrix_multiply_elementwise`(A, B)	Return the Hadamard product (elementwise product) of A and B
`matrix_symbols`(expr)	
`maximum`(f, symbol[, domain])	Returns the maximum value of a function in the given domain.
`mellin_transform`(f, x, s, **hints)	Compute the Mellin transform *F(s)* of *f(x)*,
`memoize_property`(propfunc)	Property decorator that caches the value of potentially expensive *propfunc* after the first evaluation.
`mersenne_prime_exponent`(nth)	Returns the exponent `i` for the nth Mersenne prime (which has the form *2^i - 1*).
`minimal_polynomial`(ex[, x, compose, polys, ...])	Computes the minimal polynomial of an algebraic element.
`minimum`(f, symbol[, domain])	Returns the minimum value of a function in the given domain.

continues on next page

Table 64 – continued from previous page

minpoly(ex[, x, compose, polys, domain])	This is a synonym for minimal_polynomial().
mobius_transform(seq[, subset])	Performs the Mobius Transform for subset lattice with indices of sequence as bitmasks.
mod_inverse(a, m)	Return the number c such that, $a times c = 1 pmod{m}$ where c has the same sign as m.
monic(f, *gens, **args)	Divide all coefficients of f by LC(f).
multiline_latex(lhs, rhs[, terms_per_line, ...])	This function generates a LaTeX equation with a multiline right-hand side in an align*, eqnarray or IEEEeqnarray environment.
multinomial_coefficients(m, n)	Return a dictionary containing pairs {(k1,k2,..,km) : C_kn} where C_kn are multinomial coefficients such that n=k1+k2+..+km.
multiplicity(p, n)	Find the greatest integer m such that p**m divides n.
n_order(a, n)	Returns the order of a modulo n.
nextprime(n[, ith])	Return the ith prime greater than n.
nfloat(expr[, n, exponent, dkeys])	Make all Rationals in expr Floats except those in exponents (unless the exponents flag is set to True) and those in undefined functions.
nonlinsolve(system, *symbols)	Solve system of N nonlinear equations with M variables, which means both under and overdetermined systems are supported.
not_empty_in(finset_intersection, *syms)	Finds the domain of the functions in finset_intersection in which the finite_set is not-empty.
npartitions(n[, verbose])	Calculate the partition function P(n), i.e. the number of ways that n can be written as a sum of positive integers.
nroots(f[, n, maxsteps, cleanup])	Compute numerical approximations of roots of f.
nsimplify(expr[, constants, tolerance, ...])	Find a simple representation for a number or, if there are free symbols or if rational=True, then replace Floats with their Rational equivalents.
nsolve(*args[, dict])	Solve a nonlinear equation system numerically: nsolve(f, [args,] x0, modules=['mpmath'], **kwargs).
nth_power_roots_poly(f, n, *gens, **args)	Construct a polynomial with n-th powers of roots of f.
nthroot_mod(a, n, p[, all_roots])	Find the solutions to x**n = a mod p.
ntt(seq, prime)	Performs the Number Theoretic Transform (**NTT**), which specializes the Discrete Fourier Transform (**DFT**) over quotient ring *Z/pZ* for prime *p* instead of complex numbers *C*.
numbered_symbols([prefix, cls, start, exclude])	Generate an infinite stream of Symbols consisting of a prefix and increasing subscripts provided that they do not occur in exclude.
numer(expr)	
octave_code(expr[, assign_to])	Converts *expr* to a string of Octave (or Matlab) code.
ode_order(expr, func)	Returns the order of a given differential equation with respect to func.
ones(*args, **kwargs)	Returns a matrix of ones with rows rows and cols columns; if cols is omitted a square matrix will be returned.

continues on next page

Table 64 – continued from previous page

`ordered`(seq[, keys, default, warn])	Return an iterator of the seq where keys are used to break ties in a conservative fashion: if, after applying a key, there are no ties then no other keys will be computed.
`pager_print`(expr, **settings)	Prints expr using the pager, in pretty form.
`parallel_poly_from_expr`(exprs, *gens, **args)	Construct polynomials from expressions.
`parse_expr`(s[, local_dict, transformations, ...])	Converts the string `s` to a SymPy expression, in `local_dict`.
`pde_separate`(eq, fun, sep[, strategy])	Separate variables in partial differential equation either by additive or multiplicative separation approach.
`pde_separate_add`(eq, fun, sep)	Helper function for searching additive separable solutions.
`pde_separate_mul`(eq, fun, sep)	Helper function for searching multiplicative separable solutions.
`pdiv`(f, g, *gens, **args)	Compute polynomial pseudo-division of `f` and `g`.
`pdsolve`(eq[, func, hint, dict, solvefun])	Solves any (supported) kind of partial differential equation.
`per`(matexpr)	Matrix Permanent
`perfect_power`(n[, candidates, big, factor])	Return `(b, e)` such that `n` == `b**e` if `n` is a unique perfect power with `e > 1`, else `False` (e.g. 1 is not a perfect power).
`periodicity`(f, symbol[, check])	Tests the given function for periodicity in the given symbol.
`permutedims`(expr[, perm, index_order_old, ...])	Permutes the indices of an array.
`pexquo`(f, g, *gens, **args)	Compute polynomial exact pseudo-quotient of `f` and `g`.
`piecewise_exclusive`(expr, *[, skip_nan, deep])	Rewrite `Piecewise` with mutually exclusive conditions.
`piecewise_fold`(expr[, evaluate])	Takes an expression containing a piecewise function and returns the expression in piecewise form.
`plot`(*args[, show])	Plots a function of a single variable as a curve.
`plot_implicit`(expr[, x_var, y_var, ...])	A plot function to plot implicit equations / inequalities.
`plot_parametric`(*args[, show])	Plots a 2D parametric curve.
`polarify`(eq[, subs, lift])	Turn all numbers in eq into their polar equivalents (under the standard choice of argument).
`pollard_pm1`(n[, B, a, retries, seed])	Use Pollard's p-1 method to try to extract a nontrivial factor of `n`.
`pollard_rho`(n[, s, a, retries, seed, ...])	Use Pollard's rho method to try to extract a nontrivial factor of `n`.
`poly`(expr, *gens, **args)	Efficiently transform an expression into a polynomial.
`poly_from_expr`(expr, *gens, **args)	Construct a polynomial from an expression.
`posify`(eq)	Return `eq` (with generic symbols made positive) and a dictionary containing the mapping between the old and new symbols.
`postfixes`(seq)	Generate all postfixes of a sequence.
`postorder_traversal`(node[, keys])	Do a postorder traversal of a tree.
`powdenest`(eq[, force, polar])	Collect exponents on powers as assumptions allow.
`powsimp`(expr[, deep, combine, force, measure])	Reduce expression by combining powers with similar bases and exponents.
`pprint`(expr, **kwargs)	Prints expr in pretty form.
`pprint_try_use_unicode`()	See if unicode output is available and leverage it if possible
`pprint_use_unicode`([flag])	Set whether pretty-printer should use unicode by default
`pquo`(f, g, *gens, **args)	Compute polynomial pseudo-quotient of `f` and `g`.
`prefixes`(seq)	Generate all prefixes of a sequence.

continues on next page

Table 64 – continued from previous page

prem(f, g, *gens, **args)	Compute polynomial pseudo-remainder of f and g.
pretty_print(expr, **kwargs)	Prints expr in pretty form.
preview(expr[, output, viewer, euler, ...])	View expression or LaTeX markup in PNG, DVI, PostScript or PDF form.
prevprime(n)	Return the largest prime smaller than n.
prime(nth)	Return the nth prime, with the primes indexed as prime(1) = 2, prime(2) = 3, etc.
prime_decomp(p[, T, ZK, dK, radical])	Compute the decomposition of rational prime p in a number field.
prime_valuation(I, P)	Compute the P-adic valuation for an integral ideal I.
primefactors(n[, limit, verbose])	Return a sorted list of n's prime factors, ignoring multiplicity and any composite factor that remains if the limit was set too low for complete factorization.
primerange(a[, b])	Generate a list of all prime numbers in the range [2, a), or [a, b).
primitive(f, *gens, **args)	Compute content and the primitive form of f.
primitive_element(extension[, x, ex, polys])	Find a single generator for a number field given by several generators.
primitive_root(p)	Returns the smallest primitive root or None.
primorial(n[, nth])	Returns the product of the first n primes (default) or the primes less than or equal to n (when nth=False).
print_ccode(expr, **settings)	Prints C representation of the given expression.
print_fcode(expr, **settings)	Prints the Fortran representation of the given expression.
print_glsl(expr, **settings)	Prints the GLSL representation of the given expression.
print_gtk(x[, start_viewer])	Print to Gtkmathview, a gtk widget capable of rendering MathML.
print_jscode(expr, **settings)	Prints the Javascript representation of the given expression.
print_latex(expr, **settings)	Prints LaTeX representation of the given expression.
print_maple_code(expr, **settings)	Prints the Maple representation of the given expression.
print_mathml(expr[, printer])	Prints a pretty representation of the MathML code for expr.
print_python(expr, **settings)	Print output of python() function
print_rcode(expr, **settings)	Prints R representation of the given expression.
print_tree(node[, assumptions])	Prints a tree representation of "node".
prod(a[, start])	Return product of elements of a. Start with int 1 so if only
product(*args, **kwargs)	Compute the product.
proper_divisor_count(n[, modulus])	Return the number of proper divisors of n.
proper_divisors(n[, generator])	Return all divisors of n except n, sorted by default.
public(obj)	Append obj's name to global __all__ variable (call site).
pycode(expr, **settings)	Converts an expr to a string of Python code
python(expr, **settings)	Return Python interpretation of passed expression (can be passed to the exec() function without any modifications)
quadratic_congruence(a, b, c, p)	Find the solutions to ``a x**2 + b x + c = 0 mod p.
quadratic_residues(p)	Returns the list of quadratic residues.
quo(f, g, *gens, **args)	Compute polynomial quotient of f and g.
rad(d)	Return the radian value for the given degrees (pi = 180 degrees).
radsimp(expr[, symbolic, max_terms])	Rationalize the denominator by removing square roots.

continues on next page

Table 64 – continued from previous page

`randMatrix`(r[, c, min, max, seed, ...])	Create random matrix with dimensions `r` x `c`.
`random_poly`(x, n, inf, sup[, domain, polys])	Generates a polynomial of degree `n` with coefficients in `[inf, sup]`.
`randprime`(a, b)	Return a random prime number in the range [a, b).
`rational_interpolate`(data, degnum[, X])	Returns a rational interpolation, where the data points are element of any integral domain.
`ratsimp`(expr)	Put an expression over a common denominator, cancel and reduce.
`ratsimpmodprime`(expr, G, *gens[, quick, ...])	Simplifies a rational expression `expr` modulo the prime ideal generated by `G`.
`rcode`(expr[, assign_to])	Converts an expr to a string of r code
`rcollect`(expr, *vars)	Recursively collect sums in an expression.
`real_root`(arg[, n, evaluate])	Return the real *n*'th-root of *arg* if possible.
`real_roots`(f[, multiple])	Return a list of real roots with multiplicities of `f`.
`reduce_abs_inequalities`(exprs, gen)	Reduce a system of inequalities with nested absolute values.
`reduce_abs_inequality`(expr, rel, gen)	Reduce an inequality with nested absolute values.
`reduce_inequalities`(inequalities[, symbols])	Reduce a system of inequalities with rational coefficients.
`reduced`(f, G, *gens, **args)	Reduces a polynomial `f` modulo a set of polynomials `G`.
`refine`(expr[, assumptions])	Simplify an expression using assumptions.
`refine_root`(f, s, t[, eps, steps, fast, ...])	Refine an isolating interval of a root to the given precision.
`register_handler`(key, handler)	Register a handler in the ask system.
`rem`(f, g, *gens, **args)	Compute polynomial remainder of `f` and `g`.
`remove_handler`(key, handler)	Removes a handler from the ask system.
`reshape`(seq, how)	Reshape the sequence according to the template in `how`.
`residue`(expr, x, x0)	Finds the residue of `expr` at the point x=x0.
`resultant`(f, g, *gens[, includePRS])	Compute resultant of `f` and `g`.
`ring`(symbols, domain[, order])	Construct a polynomial ring returning `(ring, x_1, ..., x_n)`.
`root`(arg, n[, k, evaluate])	Returns the *k*-th *n*-th root of `arg`.
`rootof`(f, x[, index, radicals, expand])	An indexed root of a univariate polynomial.
`roots`(f, *gens[, auto, cubics, trig, ...])	Computes symbolic roots of a univariate polynomial.
`rot_axis1`(theta)	Returns a rotation matrix for a rotation of theta (in radians) about the 1-axis.
`rot_axis2`(theta)	Returns a rotation matrix for a rotation of theta (in radians) about the 2-axis.
`rot_axis3`(theta)	Returns a rotation matrix for a rotation of theta (in radians) about the 3-axis.
`rot_ccw_axis1`(theta)	Returns a rotation matrix for a rotation of theta (in radians) about the 1-axis.
`rot_ccw_axis2`(theta)	Returns a rotation matrix for a rotation of theta (in radians) about the 2-axis.
`rot_ccw_axis3`(theta)	Returns a rotation matrix for a rotation of theta (in radians) about the 3-axis.
`rot_givens`(i, j, theta[, dim])	Returns a a Givens rotation matrix, a a rotation in the plane spanned by two coordinates axes.
`rotations`(s[, dir])	Return a generator giving the items in s as list where each subsequent list has the items rotated to the left (default) or right (`dir=-1`) relative to the previous list.
`round_two`(T[, radicals])	Zassenhaus's "Round 2" algorithm.

continues on next page

Table 64 – continued from previous page

`rsolve`(f, y[, init])	Solve univariate recurrence with rational coefficients.
`rsolve_hyper`(coeffs, f, n, **hints)	Given linear recurrence operator *operatorname{L}* of order k with polynomial coefficients and inhomogeneous equation *operatorname{L}* $y = f$ we seek for all hypergeometric solutions over field K of characteristic zero.
`rsolve_poly`(coeffs, f, n[, shift])	Given linear recurrence operator *operatorname{L}* of order k with polynomial coefficients and inhomogeneous equation *operatorname{L}* $y = f$, where f is a polynomial, we seek for all polynomial solutions over field K of characteristic zero.
`rsolve_ratio`(coeffs, f, n, **hints)	Given linear recurrence operator *operatorname{L}* of order k with polynomial coefficients and inhomogeneous equation *operatorname{L}* $y = f$, where f is a polynomial, we seek for all rational solutions over field K of characteristic zero.
`rust_code`(expr[, assign_to])	Converts an expr to a string of Rust code
`satisfiable`(expr[, algorithm, all_models, ...])	Check satisfiability of a propositional sentence.
`separatevars`(expr[, symbols, dict, force])	Separates variables in an expression, if possible.
`sequence`(seq[, limits])	Returns appropriate sequence object.
`series`(expr[, x, x0, n, dir])	Series expansion of expr around point $x = x0$.
`seterr`([divide])	Should SymPy raise an exception on 0/0 or return a nan?
`sfield`(exprs, *symbols, **options)	Construct a field deriving generators and domain from options and input expressions.
`shape`()	Return the shape of the *expr* as a tuple.
`sift`(seq, keyfunc[, binary])	Sift the sequence, `seq` according to `keyfunc`.
`signsimp`(expr[, evaluate])	Make all Add sub-expressions canonical wrt sign.
`simplify`(expr[, ratio, measure, rational, ...])	Simplifies the given expression.
`simplify_logic`(expr[, form, deep, force, ...])	This function simplifies a boolean function to its simplified version in SOP or POS form.
`sine_transform`(f, x, k, **hints)	Compute the unitary, ordinary-frequency sine transform of *f*, defined as
`singularities`(expression, symbol[, domain])	Find singularities of a given function.
`singularityintegrate`(f, x)	This function handles the indefinite integrations of Singularity functions.
`smtlib_code`(expr[, auto_assert, ...])	Converts `expr` to a string of smtlib code.
`solve`(f, *symbols, **flags)	Algebraically solves equations and systems of equations.
`solve_linear`(lhs[, rhs, symbols, exclude])	Return a tuple derived from `f = lhs - rhs` that is one of the following: `(0, 1)`, `(0, 0)`, `(symbol, solution)`, `(n, d)`.
`solve_linear_system`(system, *symbols, **flags)	Solve system of N linear equations with M variables, which means both under- and overdetermined systems are supported.
`solve_linear_system_LU`(matrix, syms)	Solves the augmented matrix system using `LUsolve` and returns a dictionary in which solutions are keyed to the symbols of *syms* as ordered.
`solve_poly_inequality`(poly, rel)	Solve a polynomial inequality with rational coefficients.
`solve_poly_system`(seq, *gens[, strict])	Return a list of solutions for the system of polynomial equations or else None.
`solve_rational_inequalities`(eqs)	Solve a system of rational inequalities with rational coefficients.
`solve_triangulated`(polys, *gens, **args)	Solve a polynomial system using Gianni-Kalkbrenner algorithm.

continues on next page

Table 64 – continued from previous page

`solve_undetermined_coeffs`(equ, coeffs, ...)	Solve a system of equations in k parameters that is formed by matching coefficients in variables `coeffs` that are on factors dependent on the remaining variables (or those given explicitly by `syms`.
`solve_univariate_inequality`(expr, gen[, ...])	Solves a real univariate inequality.
`solveset`(f[, symbol, domain])	Solves a given inequality or equation with set as output
`sqf`(f, *gens, **args)	Compute square-free factorization of `f`.
`sqf_list`(f, *gens, **args)	Compute a list of square-free factors of `f`.
`sqf_norm`(f, *gens, **args)	Compute square-free norm of `f`.
`sqf_part`(f, *gens, **args)	Compute square-free part of `f`.
`sqrt`(arg[, evaluate])	Returns the principal square root.
`sqrt_mod`(a, p[, all_roots])	Find a root of `x**2 = a mod p`.
`sqrt_mod_iter`(a, p[, domain])	Iterate over solutions to `x**2 = a mod p`.
`sqrtdenest`(expr[, max_iter])	Denests sqrts in an expression that contain other square roots if possible, otherwise returns the expr unchanged.
`sring`(exprs, *symbols, **options)	Construct a ring deriving generators and domain from options and input expressions.
`stationary_points`(f, symbol[, domain])	Returns the stationary points of a function (where derivative of the function is 0) in the given domain.
`sturm`(f, *gens, **args)	Compute Sturm sequence of `f`.
`subresultants`(f, g, *gens, **args)	Compute subresultant PRS of `f` and `g`.
`subsets`(seq[, k, repetition])	Generates all *k*-subsets (combinations) from an *n*-element set, `seq`.
`substitution`(system, symbols[, result, ...])	Solves the *system* using substitution method.
`summation`(f, *symbols, **kwargs)	Compute the summation of f with respect to symbols.
`swinnerton_dyer_poly`(n[, x, polys])	Generates n-th Swinnerton-Dyer polynomial in *x*.
`symarray`(prefix, shape, **kwargs)	Create a numpy ndarray of symbols (as an object array).
`symbols`(names, *[, cls])	Transform strings into instances of `Symbol` class.
`symmetric_poly`(n, *gens[, polys])	Generates symmetric polynomial of order *n*.
`symmetrize`(F, *gens, **args)	Rewrite a polynomial in terms of elementary symmetric polynomials.
`sympify`(a[, locals, convert_xor, strict, ...])	Converts an arbitrary expression to a type that can be used inside SymPy.
`take`(iter, n)	Return `n` items from `iter` iterator.
`tensorcontraction`(array, *contraction_axes)	Contraction of an array-like object on the specified axes.
`tensordiagonal`(array, *diagonal_axes)	Diagonalization of an array-like object on the specified axes.
`tensorproduct`(*args)	Tensor product among scalars or array-like objects.
`terms_gcd`(f, *gens, **args)	Remove GCD of terms from `f`.
`textplot`(expr, a, b[, W, H])	Print a crude ASCII art plot of the SymPy expression 'expr' (which should contain a single symbol, e.g. x or something else) over the interval [a, b].
`threaded`(func)	Apply `func` to sub--elements of an object, including `Add`.
`timed`(func[, setup, limit])	Adaptively measure execution time of a function.
`to_cnf`(expr[, simplify, force])	Convert a propositional logical sentence `expr` to conjunctive normal form: `((A | ~B | ...) & (B | C | ...) & ...)`.
`to_dnf`(expr[, simplify, force])	Convert a propositional logical sentence `expr` to disjunctive normal form: `((A & ~B & ...) | (B & C & ...) | ...)`.
`to_nnf`(expr[, simplify])	Converts `expr` to Negation Normal Form (NNF).

continues on next page

Table 64 – continued from previous page

`to_number_field`(extension[, theta, gen, alias])	Express one algebraic number in the field generated by another.
`together`(expr[, deep, fraction])	Denest and combine rational expressions using symbolic methods.
`topological_sort`(graph[, key])	Topological sort of graph's vertices.
`total_degree`(f, *gens)	Return the total_degree of `f` in the given variables.
`trace`(expr)	Trace of a Matrix.
`trailing`(n)	Count the number of trailing zero digits in the binary representation of n, i.e. determine the largest power of 2 that divides n.
`trigsimp`(expr[, inverse])	Returns a reduced expression by using known trig identities.
`trunc`(f, p, *gens, **args)	Reduce `f` modulo a constant `p`.
`unbranched_argument`(arg)	Returns periodic argument of arg with period as infinity.
`unflatten`(iter[, n])	Group `iter` into tuples of length `n`.
`unpolarify`(eq[, subs, exponents_only])	If *p* denotes the projection from the Riemann surface of the logarithm to the complex line, return a simplified version *eq'* of *eq* such that *p(eq') = p(eq)*.
`use`(expr, func[, level, args, kwargs])	Use `func` to transform `expr` at the given level.
`var`(names, **args)	Create symbols and inject them into the global namespace.
`variations`(seq, n[, repetition])	Returns an iterator over the n-sized variations of `seq` (size N).
`vfield`(symbols, domain[, order])	Construct new rational function field and inject generators into global namespace.
`viete`(f[, roots])	Generate Viete's formulas for `f`.
`vring`(symbols, domain[, order])	Construct a polynomial ring and inject `x_1, ..., x_n` into the global namespace.
`wronskian`(functions, var[, method])	Compute Wronskian for [] of functions
`xfield`(symbols, domain[, order])	Construct new rational function field returning (field, (x1, ..., xn)).
`xring`(symbols, domain[, order])	Construct a polynomial ring returning `(ring, (x_1, ..., x_n))`.
`xthreaded`(func)	Apply `func` to sub--elements of an object, excluding `Add`.
`zeros`(*args, **kwargs)	Returns a matrix of zeros with `rows` rows and `cols` columns; if `cols` is omitted a square matrix will be returned.

Classes

`Abs`(arg)	Return the absolute value of the argument.
`AccumBounds`	alias of `AccumulationBounds`
`Add`(*args[, evaluate, _sympify])	Expression representing addition operation for algebraic group.
`Adjoint`(*args, **kwargs)	The Hermitian adjoint of a matrix expression.
`AlgebraicField`(dom, *ext[, alias])	Algebraic number field QQ(a)
`AlgebraicNumber`(expr[, coeffs, alias])	Class for representing algebraic numbers in SymPy.
`And`(*args)	Logical AND function.
`AppliedPredicate`(predicate, *args)	The class of expressions resulting from applying `Predicate` to the arguments.

continues on next page

Table 65 – continued from previous page

`Array`	alias of `ImmutableDenseNDimArray`
`AssumptionsContext`	Set containing default assumptions which are applied to the `ask()` function.
`Atom`(*args)	A parent class for atomic things.
`AtomicExpr`(*args)	A parent class for object which are both atoms and Exprs.
`AutoSympy`(model)	
`Basic`(*args)	Base class for all SymPy objects.
`BlockDiagMatrix`(*mats)	A sparse matrix with block matrices along its diagonals
`BlockMatrix`(*args, **kwargs)	A BlockMatrix is a Matrix comprised of other matrices.
`CRootOf`	alias of `ComplexRootOf`
`Chi`(z)	Cosh integral.
`Ci`(z)	Cosine integral.
`Circle`(*args, **kwargs)	A circle in space.
`Complement`(a, b[, evaluate])	Represents the set difference or relative complement of a set with another set.
`ComplexField`([prec, dps, tol])	Complex numbers up to the given precision.
`ComplexRegion`(sets[, polar])	Represents the Set of all Complex Numbers.
`ComplexRootOf`(f, x[, index, radicals, expand])	Represents an indexed complex root of a polynomial.
`ConditionSet`(sym, condition[, base_set])	Set of elements which satisfies a given condition.
`Contains`(x, s)	Asserts that x is an element of the set S.
`CoordMap`(var_vector, eq_duals, ineq_duals, ...)	
`CosineTransform`(*args)	Class representing unevaluated cosine transforms.
`Curve`(function, limits)	A curve in space.
`DeferredVector`(name, **assumptions)	A vector whose components are deferred (e.g. for use with lambdify).
`DenseNDimArray`(*args, **kwargs)	
`Derivative`(expr, *variables, **kwargs)	Carries out differentiation of the given expression with respect to symbols.
`Determinant`(mat)	Matrix Determinant
`DiagMatrix`(vector)	Turn a vector into a diagonal matrix.
`DiagonalMatrix`(*args, **kwargs)	DiagonalMatrix(M) will create a matrix expression that behaves as though all off-diagonal elements, *M[i, j]* where $i \mathrel{!}= j$, are zero.
`DiagonalOf`(*args, **kwargs)	DiagonalOf(M) will create a matrix expression that is equivalent to the diagonal of M, represented as a single column matrix.
`Dict`(*args)	Wrapper around the builtin dict object.
`DifferentialMapping`(US, coord2item, ...)	
`DiracDelta`(arg[, k])	The DiracDelta function and its derivatives.
`DisjointUnion`(*sets)	Represents the disjoint union (also known as the external disjoint union) of a finite number of sets.
`Domain`()	Superclass for all domains in the polys domains system.
`DotProduct`(arg1, arg2)	Dot product of vector matrices
`Dummy`([name, dummy_index])	Dummy symbols are each unique, even if they have the same name:
`EPath`(path)	Manipulate expressions using paths.
`Ei`(z)	The classical exponential integral.
`Ellipse`([center, hradius, vradius, eccentricity])	An elliptical GeometryEntity.

continues on next page

Table 65 – continued from previous page

Eq	alias of Equality
Equality(lhs, rhs, **options)	An equal relation between two objects.
Equivalent(*args)	Equivalence relation.
Expr(*args)	Base class for algebraic expressions.
ExpressionDomain()	A class for arbitrary expressions.
FF	alias of FiniteField
FF_gmpy	alias of GMPYFiniteField
FF_python	alias of PythonFiniteField
FallingFactorial(x, k)	Falling factorial (related to rising factorial) is a double valued function arising in concrete mathematics, hypergeometric functions and series expansions.
FiniteField(mod[, symmetric])	Finite field of prime order GF(p)
FiniteSet(*args, **kwargs)	Represents a finite set of Sympy expressions.
Float(num[, dps, precision])	Represent a floating-point number of arbitrary precision.
FourierTransform(*args)	Class representing unevaluated Fourier transforms.
FractionField(domain_or_field[, symbols, order])	A class for representing multivariate rational function fields.
Function(*args)	Base class for applied mathematical functions.
FunctionClass(*args, **kwargs)	Base class for function classes.
FunctionMatrix(rows, cols, lamda)	Represents a matrix using a function (Lambda) which gives outputs according to the coordinates of each matrix entries.
GF	alias of FiniteField
GMPYFiniteField(mod[, symmetric])	Finite field based on GMPY integers.
GMPYIntegerRing()	Integer ring based on GMPY's mpz type.
GMPYRationalField()	Rational field based on GMPY's mpq type.
Ge	alias of GreaterThan
GreaterThan(lhs, rhs, **options)	Class representations of inequalities.
GroebnerBasis(F, *gens, **args)	Represents a reduced Groebner basis.
Gt	alias of StrictGreaterThan
HadamardPower(base, exp)	Elementwise power of matrix expressions
HadamardProduct(*args[, evaluate, check])	Elementwise product of matrix expressions
HankelTransform(*args)	Class representing unevaluated Hankel transforms.
Heaviside(arg[, H0])	Heaviside step function.
ITE(*args)	If-then-else clause.
Identity(n)	The Matrix Identity I - multiplicative identity
Idx(label[, range])	Represents an integer index as an Integer or integer expression.
ImageSet(flambda, *sets)	Image of a set under a mathematical function.
ImmutableDenseMatrix(*args, **kwargs)	Create an immutable version of a matrix.
ImmutableDenseNDimArray(iterable[, shape])	
ImmutableMatrix	alias of ImmutableDenseMatrix
ImmutableSparseMatrix(*args, **kwargs)	Create an immutable version of a sparse matrix.
ImmutableSparseNDimArray([iterable, shape])	
Implies(*args)	Logical implication.
Indexed(base, *args, **kw_args)	Represents a mathematical object with indices.
IndexedBase(label[, shape, offset, strides])	Represent the base or stem of an indexed object
Integer(i)	Represents integer numbers of any size.
IntegerRing()	The domain ZZ representing the integers *mathbb{Z}*.
Integral(function, *symbols, **assumptions)	Represents unevaluated integral.

continues on next page

Table 65 – continued from previous page

Intersection(*args, **kwargs)	Represents an intersection of sets as a Set.
Interval(start, end[, left_open, right_open])	Represents a real interval as a Set.
Inverse(mat[, exp])	The multiplicative inverse of a matrix expression
InverseCosineTransform(*args)	Class representing unevaluated inverse cosine transforms.
InverseFourierTransform(*args)	Class representing unevaluated inverse Fourier transforms.
InverseHankelTransform(*args)	Class representing unevaluated inverse Hankel transforms.
InverseLaplaceTransform(*args)	Class representing unevaluated inverse Laplace transforms.
InverseMellinTransform(*args)	Class representing unevaluated inverse Mellin transforms.
InverseSineTransform(*args)	Class representing unevaluated inverse sine transforms.
KroneckerDelta(i, j[, delta_range])	The discrete, or Kronecker, delta function.
KroneckerProduct(*args[, check])	The Kronecker product of two or more arguments.
Lambda(signature, expr)	Lambda(x, expr) represents a lambda function similar to Python's 'lambda x: expr'.
LambertW(x[, k])	The Lambert W function $W(z)$ is defined as the inverse function of $w exp(w)$ [1]_.
LaplaceTransform(*args)	Class representing unevaluated Laplace transforms.
Le	alias of LessThan
LessThan(lhs, rhs, **options)	Class representations of inequalities.
LeviCivita(*args)	Represent the Levi-Civita symbol.
Li(z)	The offset logarithmic integral.
Limit(e, z, z0[, dir])	Represents an unevaluated limit.
Line(*args, **kwargs)	An infinite line in space.
Line2D(p1[, pt, slope])	An infinite line in space 2D.
Line3D(p1[, pt, direction_ratio])	An infinite 3D line in space.
Lt	alias of StrictLessThan
MatAdd(*args[, evaluate, check, _sympify])	A Sum of Matrix Expressions
MatMul(*args[, evaluate, check, _sympify])	A product of matrix expressions
MatPow(base, exp[, evaluate])	
Matrix	alias of MutableDenseMatrix
MatrixBase()	Base class for matrix objects.
MatrixExpr(*args, **kwargs)	Superclass for Matrix Expressions
MatrixPermute(mat, perm[, axis])	Symbolic representation for permuting matrix rows or columns.
MatrixSlice(parent, rowslice, colslice)	A MatrixSlice of a Matrix Expression
MatrixSymbol(name, n, m)	Symbolic representation of a Matrix object
Max(*args)	Return, if possible, the maximum value of the list.
MellinTransform(*args)	Class representing unevaluated Mellin transforms.
Min(*args)	Return, if possible, the minimum value of the list.
Mod(p, q)	Represents a modulo operation on symbolic expressions.
Monomial(monom[, gens])	Class representing a monomial, i.e. a product of powers.
Mul(*args[, evaluate, _sympify])	Expression representing multiplication operation for algebraic field.
MutableDenseMatrix(*args, **kwargs)	
MutableDenseNDimArray([iterable, shape])	

continues on next page

Table 65 – continued from previous page

MutableMatrix	alias of MutableDenseMatrix
MutableSparseMatrix(*args, **kwargs)	
MutableSparseNDimArray([iterable, shape])	
NDimArray(iterable[, shape])	N-dimensional array.
Nand(*args)	Logical NAND function.
Ne	alias of Unequality
Nor(*args)	Logical NOR function.
Not(arg)	Logical Not function (negation)
Number(*obj)	Represents atomic numbers in SymPy.
NumberSymbol()	
O	alias of Order
OmegaPower(a, b)	Represents ordinal exponential and multiplication terms one of the building blocks of the Ordinal class.
OneMatrix(m, n[, evaluate])	Matrix whose all entries are ones.
Options(gens, args[, flags, strict])	Options manager for polynomial manipulation module.
Or(*args)	Logical OR function
Order(expr, *args, **kwargs)	Represents the limiting behavior of some function.
Ordinal(*terms)	Represents ordinals in Cantor normal form.
Parabola([focus, directrix])	A parabolic GeometryEntity.
Permanent(mat)	Matrix Permanent
PermutationMatrix(perm)	A Permutation Matrix
Piecewise(*_args)	Represents a piecewise function.
Plane(p1[, a, b])	A plane is a flat, two-dimensional surface.
Point(*args, **kwargs)	A point in a n-dimensional Euclidean space.
Point2D(*args[, _nocheck])	A point in a 2-dimensional Euclidean space.
Point3D(*args[, _nocheck])	A point in a 3-dimensional Euclidean space.
Poly(rep, *gens, **args)	Generic class for representing and operating on polynomial expressions.
Polygon(*args[, n])	A two-dimensional polygon.
PolynomialRing(domain_or_ring[, symbols, order])	A class for representing multivariate polynomial rings.
Pow(b, e[, evaluate])	Defines the expression x**y as "x raised to a power y"
PowerSet(arg[, evaluate])	A symbolic object representing a power set.
Predicate(*args, **kwargs)	Base class for mathematical predicates.
Product(function, *symbols, **assumptions)	Represents unevaluated products.
ProductSet(*sets, **assumptions)	Represents a Cartesian Product of Sets.
PurePoly(rep, *gens, **args)	Class for representing pure polynomials.
PythonFiniteField(mod[, symmetric])	Finite field based on Python's integers.
PythonIntegerRing()	Integer ring based on Python's int type.
PythonRational	alias of PythonMPQ
QQ_gmpy	alias of GMPYRationalField
QQ_python	alias of PythonRationalField
Quaternion([a, b, c, d, real_field, norm])	Provides basic quaternion operations.
Range(*args)	Represents a range of integers.
Rational(p[, q, gcd])	Represents rational numbers (p/q) of any size.
RationalField()	Abstract base class for the domain QQ.
Ray(p1[, p2])	A Ray is a semi-line in the space with a source point and a direction.
Ray2D(p1[, pt, angle])	A Ray is a semi-line in the space with a source point and a direction.

continues on next page

Table 65 – continued from previous page

Ray3D(p1[, pt, direction_ratio])	A Ray is a semi-line in the space with a source point and a direction.
RealField([prec, dps, tol])	Real numbers up to the given precision.
RealNumber	alias of Float
RegularPolygon(c, r, n[, rot])	A regular polygon.
Rel	alias of Relational
Rem(p, q)	Returns the remainder when p is divided by q where p is finite and q is not equal to zero.
RisingFactorial(x, k)	Rising factorial (also called Pochhammer symbol [1]_) is a double valued function arising in concrete mathematics, hypergeometric functions and series expansions.
RootOf(f, x[, index, radicals, expand])	Represents a root of a univariate polynomial.
RootSum(expr[, func, x, auto, quadratic])	Represents a sum of all roots of a univariate polynomial.
Segment(p1, p2, **kwargs)	A line segment in space.
Segment2D(p1, p2, **kwargs)	A line segment in 2D space.
Segment3D(p1, p2, **kwargs)	A line segment in a 3D space.
SensitivityMatrix(sympification, duals, ...)	
SeqAdd(*args, **kwargs)	Represents term-wise addition of sequences.
SeqFormula(formula[, limits])	Represents sequence based on a formula.
SeqMul(*args, **kwargs)	Represents term-wise multiplication of sequences.
SeqPer(periodical[, limits])	Represents a periodic sequence.
Set(*args)	The base class for any kind of set.
Shi(z)	Sinh integral.
Si(z)	Sine integral.
Sieve()	An infinite list of prime numbers, implemented as a dynamically growing sieve of Eratosthenes.
SineTransform(*args)	Class representing unevaluated sine transforms.
SingularityFunction(variable, offset, exponent)	Singularity functions are a class of discontinuous functions.
SparseMatrix	alias of MutableSparseMatrix
SparseNDimArray(*args, **kwargs)	
StrPrinter([settings])	
StrictGreaterThan(lhs, rhs, **options)	Class representations of inequalities.
StrictLessThan(lhs, rhs, **options)	Class representations of inequalities.
Subs(expr, variables, point, **assumptions)	Represents unevaluated substitutions of an expression.
Sum(function, *symbols, **assumptions)	Represents unevaluated summation.
Symbol(name, **assumptions)	Assumptions:
SymmetricDifference(a, b[, evaluate])	Represents the set of elements which are in either of the sets and not in their intersection.
TableForm(data, **kwarg)	Create a nice table representation of data.
Trace(mat)	Matrix Trace
Transpose(*args, **kwargs)	The transpose of a matrix expression.
Triangle(*args, **kwargs)	A polygon with three vertices and three sides.
Tuple(*args, **kwargs)	Wrapper around the builtin tuple object.
Unequality(lhs, rhs, **options)	An unequal relation between two objects.
UnevaluatedExpr(arg, **kwargs)	Expression that is not evaluated unless released.
Union(*args, **kwargs)	Represents a union of sets as a Set.
Wild(name[, exclude, properties])	A Wild symbol matches anything, or anything without whatever is explicitly excluded.

continues on next page

Table 65 – continued from previous page

`WildFunction`(*args)	A WildFunction function matches any function (with its arguments).
`Xor`(*args)	Logical XOR (exclusive OR) function.
`Ynm`(n, m, theta, phi)	Spherical harmonics defined as
`ZZ_gmpy`	alias of `GMPYIntegerRing`
`ZZ_python`	alias of `PythonIntegerRing`
`ZeroMatrix`(m, n)	The Matrix Zero 0 - additive identity
`Znm`(n, m, theta, phi)	Real spherical harmonics defined as
`acos`(arg)	The inverse cosine function.
`acosh`(arg)	`acosh(x)` is the inverse hyperbolic cosine of `x`.
`acot`(arg)	The inverse cotangent function.
`acoth`(arg)	`acoth(x)` is the inverse hyperbolic cotangent of `x`.
`acsc`(arg)	The inverse cosecant function.
`acsch`(arg)	`acsch(x)` is the inverse hyperbolic cosecant of `x`.
`adjoint`(arg)	Conjugate transpose or Hermite conjugation.
`airyai`(arg)	The Airy function $operatorname{Ai}$ of the first kind.
`airyaiprime`(arg)	The derivative $operatorname{Ai}^prime$ of the Airy function of the first kind.
`airybi`(arg)	The Airy function $operatorname{Bi}$ of the second kind.
`airybiprime`(arg)	The derivative $operatorname{Bi}^prime$ of the Airy function of the first kind.
`andre`(n)	Andre numbers / Andre function
`appellf1`(a, b1, b2, c, x, y)	This is the Appell hypergeometric function of two variables as:
`arg`(arg)	Returns the argument (in radians) of a complex number.
`asec`(arg)	The inverse secant function.
`asech`(arg)	`asech(x)` is the inverse hyperbolic secant of `x`.
`asin`(arg)	The inverse sine function.
`asinh`(arg)	`asinh(x)` is the inverse hyperbolic sine of `x`.
`assoc_laguerre`(n, alpha, x)	Returns the nth generalized Laguerre polynomial in x, $L_n(x)$.
`assoc_legendre`(n, m, x)	`assoc_legendre(n, m, x)` gives $P_n^m(x)$, where n and m are the degree and order or an expression which is related to the nth order Legendre polynomial, $P_n(x)$ in the following manner:
`atan`(arg)	The inverse tangent function.
`atan2`(y, x)	The function `atan2(y, x)` computes *operatorname{atan}(y/x)* taking two arguments *y* and *x*.
`atanh`(arg)	`atanh(x)` is the inverse hyperbolic tangent of `x`.
`bell`(n[, k_sym, symbols])	Bell numbers / Bell polynomials
`bernoulli`(n[, x])	Bernoulli numbers / Bernoulli polynomials / Bernoulli function
`besseli`(nu, z)	Modified Bessel function of the first kind.
`besselj`(nu, z)	Bessel function of the first kind.
`besselk`(nu, z)	Modified Bessel function of the second kind.
`bessely`(nu, z)	Bessel function of the second kind.
`beta`(x[, y])	The beta integral is called the Eulerian integral of the first kind by Legendre:
`betainc`(*args)	The Generalized Incomplete Beta function is defined as
`betainc_regularized`(*args)	The Generalized Regularized Incomplete Beta function is given by

continues on next page

Table 65 – continued from previous page

binomial(n, k)	Implementation of the binomial coefficient.
carmichael(*args)	Carmichael Numbers:
cartes	alias of product
catalan(n)	Catalan numbers
ceiling(arg)	Ceiling is a univariate function which returns the smallest integer value not less than its argument.
chebyshevt(n, x)	Chebyshev polynomial of the first kind, $T_n(x)$.
chebyshevt_root(n, k)	chebyshev_root(n, k) returns the kth root (indexed from zero) of the nth Chebyshev polynomial of the first kind; that is, if $0 le k < n$, chebyshevt(n, chebyshevt_root(n, k)) == 0.
chebyshevu(n, x)	Chebyshev polynomial of the second kind, $U_n(x)$.
chebyshevu_root(n, k)	chebyshevu_root(n, k) returns the kth root (indexed from zero) of the nth Chebyshev polynomial of the second kind; that is, if $0 le k < n$, chebyshevu(n, chebyshevu_root(n, k)) == 0.
conjugate(arg)	Returns the *complex conjugate* [1]_ of an argument.
cos(arg)	The cosine function.
cosh(arg)	cosh(x) is the hyperbolic cosine of x.
cot(arg)	The cotangent function.
coth(arg)	coth(x) is the hyperbolic cotangent of x.
csc(arg)	The cosecant function.
csch(arg)	csch(x) is the hyperbolic cosecant of x.
defaultdict	defaultdict(default_factory=None, /, [...]) --> dict with default factory
digamma(z)	The digamma function is the first derivative of the loggamma function
dirichlet_eta(s[, a])	Dirichlet eta function.
divisor_sigma(n[, k])	Calculate the divisor function *sigma_k(n)* for positive integer n
elliptic_e(m[, z])	Called with two arguments z and m, evaluates the incomplete elliptic integral of the second kind, defined by
elliptic_f(z, m)	The Legendre incomplete elliptic integral of the first kind, defined by
elliptic_k(m)	The complete elliptic integral of the first kind, defined by
elliptic_pi(n, m[, z])	Called with three arguments n, z and m, evaluates the Legendre incomplete elliptic integral of the third kind, defined by
erf(arg)	The Gauss error function.
erf2(x, y)	Two-argument error function.
erf2inv(x, y)	Two-argument Inverse error function.
erfc(arg)	Complementary Error Function.
erfcinv(z)	Inverse Complementary Error Function.
erfi(z)	Imaginary error function.
erfinv(z)	Inverse Error Function.
euler(n[, x])	Euler numbers / Euler polynomials / Euler function
exp(arg)	The exponential function, e^x.
exp_polar(*args)	Represent a *polar number* (see g-function Sphinx documentation).
expint(nu, z)	Generalized exponential integral.

continues on next page

Table 65 – continued from previous page

factorial(n)	Implementation of factorial function over nonnegative integers.
factorial2(arg)	The double factorial *n!!*, not to be confused with *(n!)!*
ff	alias of FallingFactorial
fibonacci(n[, sym])	Fibonacci numbers / Fibonacci polynomials
floor(arg)	Floor is a univariate function which returns the largest integer value not greater than its argument.
frac(arg)	Represents the fractional part of x
fresnelc(z)	Fresnel integral C.
fresnels(z)	Fresnel integral S.
gamma(arg)	The gamma function
gegenbauer(n, a, x)	Gegenbauer polynomial \$C_n^{left(alpharight)}(x)\$.
genocchi(n[, x])	Genocchi numbers / Genocchi polynomials / Genocchi function
hankel1(nu, z)	Hankel function of the first kind.
hankel2(nu, z)	Hankel function of the second kind.
harmonic(n[, m])	Harmonic numbers
hermite(n, x)	hermite(n, x) gives the \$n\$th Hermite polynomial in \$x\$, \$H_n(x)\$.
hermite_prob(n, x)	hermite_prob(n, x) gives the \$n\$th probabilist's Hermite polynomial in \$x\$, \$He_n(x)\$.
hn1(nu, z)	Spherical Hankel function of the first kind.
hn2(nu, z)	Spherical Hankel function of the second kind.
hyper(ap, bq, z)	The generalized hypergeometric function is defined by a series where the ratios of successive terms are a rational function of the summation index.
im(arg)	Returns imaginary part of expression.
jacobi(n, a, b, x)	Jacobi polynomial \$P_n^{left(alpha, betaright)}(x)\$.
jn(nu, z)	Spherical Bessel function of the first kind.
laguerre(n, x)	Returns the \$n\$th Laguerre polynomial in \$x\$, \$L_n(x)\$.
legendre(n, x)	legendre(n, x) gives the \$n\$th Legendre polynomial of \$x\$, \$P_n(x)\$
lerchphi(*args)	Lerch transcendent (Lerch phi function).
li(z)	The classical logarithmic integral.
ln	alias of log
log(arg[, base])	The natural logarithm function *ln(x)* or *log(x)*.
loggamma(z)	The loggamma function implements the logarithm of the gamma function (i.e., \$logGamma(x)\$).
lowergamma(a, x)	The lower incomplete gamma function.
lucas(n)	Lucas numbers
marcumq(m, a, b)	The Marcum Q-function.
mathieuc(a, q, z)	The Mathieu Cosine function \$C(a,q,z)\$.
mathieucprime(a, q, z)	The derivative \$C^{prime}(a,q,z)\$ of the Mathieu Cosine function.
mathieus(a, q, z)	The Mathieu Sine function \$S(a,q,z)\$.
mathieusprime(a, q, z)	The derivative \$S^{prime}(a,q,z)\$ of the Mathieu Sine function.
meijerg(*args)	The Meijer G-function is defined by a Mellin-Barnes type integral that resembles an inverse Mellin transform.
mobius(n)	Mobius function maps natural number to {-1, 0, 1}
motzkin(n)	The nth Motzkin number is the number

continues on next page

Table 65 – continued from previous page

`multigamma`(x, p)	The multivariate gamma function is a generalization of the gamma function
`partition`(n)	Partition numbers
`periodic_argument`(ar, period)	Represent the argument on a quotient of the Riemann surface of the logarithm.
`polar_lift`(arg)	Lift argument to the Riemann surface of the logarithm, using the standard branch.
`polygamma`(n, z)	The function `polygamma(n, z)` returns `log(gamma(z)).diff(n + 1)`.
`polylog`(s, z)	Polylogarithm function.
`preorder_traversal`(node[, keys])	Do a pre-order traversal of a tree.
`primenu`(n)	Calculate the number of distinct prime factors for a positive integer n.
`primeomega`(n)	Calculate the number of prime factors counting multiplicities for a positive integer n.
`primepi`(n)	Represents the prime counting function pi(n) = the number of prime numbers less than or equal to n.
`principal_branch`(x, period)	Represent a polar number reduced to its principal branch on a quotient of the Riemann surface of the logarithm.
`re`(arg)	Returns real part of expression.
`reduced_totient`(n)	Calculate the Carmichael reduced totient function lambda(n)
`rf`	alias of `RisingFactorial`
`riemann_xi`(s)	Riemann Xi function.
`sec`(arg)	The secant function.
`sech`(arg)	`sech(x)` is the hyperbolic secant of `x`.
`sign`(arg)	Returns the complex sign of an expression:
`sin`(arg)	The sine function.
`sinc`(arg)	Represents an unnormalized sinc function:
`sinh`(arg)	`sinh(x)` is the hyperbolic sine of `x`.
`stieltjes`(n[, a])	Represents Stieltjes constants, $gamma_{k}$ that occur in Laurent Series expansion of the Riemann zeta function.
`subfactorial`(arg)	The subfactorial counts the derangements of n items and is defined for non-negative integers as:
`tan`(arg)	The tangent function.
`tanh`(arg)	`tanh(x)` is the hyperbolic tangent of `x`.
`totient`(n)	Calculate the Euler totient function phi(n)
`transpose`(arg)	Linear map transposition.
`tribonacci`(n[, sym])	Tribonacci numbers / Tribonacci polynomials
`trigamma`(z)	The `trigamma` function is the second derivative of the `loggamma` function
`uppergamma`(a, z)	The upper incomplete gamma function.
`vectorize`(*mdargs)	Generalizes a function taking scalars to accept multidimensional arguments.
`yn`(nu, z)	Spherical Bessel function of the second kind.
`zeta`(s[, a])	Hurwitz zeta function (or Riemann zeta function).

Exceptions

`BasePolynomialError`	Base class for polynomial related exceptions.
`CoercionFailed`	
`ComputationFailed`(func, nargs, exc)	
`DomainError`	
`EvaluationFailed`	
`ExactQuotientFailed`(f, g[, dom])	
`ExtraneousFactors`	
`FlagError`	
`GeneratorsError`	
`GeneratorsNeeded`	
`GeometryError`	An exception raised by classes in the geometry module.
`HeuristicGCDFailed`	
`HomomorphismFailed`	
`IsomorphismFailed`	
`MultivariatePolynomialError`	
`NonSquareMatrixError`	
`NotAlgebraic`	
`NotInvertible`	
`NotReversible`	
`OperationNotSupported`(poly, func)	
`OptionError`	
`PoleError`	
`PolificationFailed`(opt, origs, exprs[, seq])	
`PolynomialDivisionFailed`(f, g, domain)	
`PolynomialError`	
`PrecisionExhausted`	

continues on next page

Table 66 – continued from previous page

RefinementFailed	
ShapeError	Wrong matrix shape
SympifyError(expr[, base_exc])	
UnificationFailed	
UnivariatePolynomialError	

class src.sensitivity.sensitivity_tools.**AutoSympy**(*model*)

check_complimentarity_all()

quick check of complementarity conditions

generate_duals(*constraints*, *duals*)

cycles through constraints, extracts duals, categorizes by complementary slackness conditions

Parameters

- **constraints** (*dict*) – iterator of pyomo constraints
- **duals** (*obj*) – duals object from pyomo model

Returns

dictionary of duals sorts by type

Return type

dict

get_constraints()

extract constraint expressions from self.model and convert to Sympy expressions in terms of the extracted Sympy Symbols for variables and parameters

Returns

- **equality_constraints** (*dict*) – dictionary of names and indices of equality constraints:Sympy expression for lhs of constraint
- **inequality_constraints** (*dict*) – dictionary of names and indices of inequality constraints:Sympy expression for lhs of constraint

get_objective()

extract objective expression from self.model

Returns

objective expression

Return type

expr

get_parameters()

extracts parameters from self.model and converts to Sympy Symbol or IndexedBase + index objects depending on whether they are indexed components or not.

Returns

- **parameters** (*dict*) – dictionary of parameter names:IndexedBase objects with that name
- **parameters_values** (*dict*) – dictionary of Sympy objects:their numeric value

- **parameter_index_sets** (*dict*) – dictionary of parameter name:list of indices for the name

`get_sensitivity_matrix`(*parameters_of_interest=None*)

generates a SensitivityMatrix object based on the sympy representation, keeping parameters in parameters_of_interest and substituting numeric values for the rest in all expressions.

Parameters
`parameters_of_interest` (*list, optional*) – list of parameters to keep symbolic. These are the parameters to evaluate sensitivity for. Defaults to None.

Returns
SensitivityMatrix object for the model, with parameters_of_interest considered.

Return type
obj

`get_sets`()

extracts sets from self.model and converts them to Sympy Idx objects

Returns

- **sets** (*dict*) – dictionary of set names:Idx objects
- **set_values** (*dict*) – dictionary of Idx objects:corresponding element of the Pyomo set

`get_variables`()

extracts variables from self.model and converts them to Sympy Symbol or IndexedBase + index objects

Returns

- **variables** (*dict*) – dictionary of variable names:corresponding Sympy object (Symbol or IndexedBase)
- **variable_values** (*dict*) – dictionary of Sympy objects (Symbol or IndexedBase):numeric value

`substitute_values`(*substitution_dict*)

subsitutes numeric values for sympy symbols according to substitution_dict

Parameters
`substitution_dict` (*dict*) – dictionary of symbols:values to substitute

`class src.sensitivity.sensitivity_tools.CoordMap`(*var_vector*, *eq_duals*, *ineq_duals*, *params*)

`class src.sensitivity.sensitivity_tools.DifferentialMapping`(*US*, *coord2item*, *item2coord*, *param2coord*, *coord2param*)

`extrapolate`(*current_values*, *param_delta*)

given the current value of all varying quantities (variables, duals, objective) as a vector,
and a delta of parameter values as a vector, calculates the effect of perturbing the parameters by the parameter delta.

Parameters

- `current_values` (*Matrix*) – vector of [variables values, dual values, objective value]
- `param_delta` (*Matrix*) – vector of [parameter values]

Returns

- **item_delta** (*obj*) – column Matrix of items
- **item_delta_map** (*dict*) – map between items and their changes

- **item_new_value** (*Matrix*) – column Matrix of new values after change
- **item_new_value_map** (*dict*) – map between items and their new values

`sensitivity`(*item*, *parameter*)

picks out the sensitivity of item with respect to parameters

Parameters

- `item` (`symbol, str`) – however the particular item was stored, usually symbol
- `parameter` (`symbol, str`) – however the particular parameter was stored, usually symbol

Returns
the sensitivity of item with respect to parameter.

Return type
float

`class src.sensitivity.sensitivity_tools.SensitivityMatrix`(*sympification*, *duals*, *parameters_of_interest*)

`create_substitution_dictionary`(*values_of_interest=None*)

creates a single dictionary with all values to be substituted into the sensitivity matrix.

Parameters
`values_of_interest` (`dict, optional`) – Subset of all possible substitutions you'd like to perform. Defaults to None.

Returns
dictionary of substitution values

Return type
dict

`generate_matrix`()

generates the submatrix components of the sensitivity matrix that will be used to express sensitivities and extrapolate from a given solution point

Returns
dictionary of submatrix name:Matrix object.

Return type
dict

`get_sensitivities`()

creates a set of dictionaries to keep track of the relationships between symbol names, symbol objects, and their position in the respective vectors of parameters and variable quantities. Calculates U^-1*S and Creates a DifferentialMapping object. Sensitivities of quantities with respect to parameters will be corresponding entries in the matrix generated, which can be queried directly in the DifferentialMapping object

Returns
an DifferentialMapping object that stores the sensitivity matrix and dictionaries matching coordinates to symbols

Return type
obj

matrix_assembly(*components*)

assemble the submatrix components into U,S

Parameters

components (*dict*) – dictionary of submatrix names:Matrix objects

Returns

- *obj* – Matrix object U
- *obj* – Matrix object S

resolve_kronecker(*expr*)

evaluates KroneckerDelta terms

Parameters

obj – sympy expression

Returns

sympy expr with KroneckerDelta terms evaluated to 0 or 1

Return type

obj

substitute_values(*values_dict=None*)

substitute values into self.U and self.S according to a given substitution dictionary, or a substition dictionary for all values

Parameters

values_dict (*dict, optional*) – dictionary of symbols:values

Returns

- *obj* – the stored Matrix object U after substituting numeric values
- *obj* – the stored matrix S after substituting numeric values

src.sensitivity.speed_test

Speed Test This is a script with some functions to run speed and accuracy tests on test models constructed with baby-model.

It can be run directly. Parameter values are at the top of the file and any desired value can be entered into the declarations. The script will then run a sequence of functions to time the build, sympification, and sensitivity calculation for the TestBabyModel constructed with those input values.

Functions

`E1`(z)	Classical case of the generalized exponential integral.
`Eijk`(*args, **kwargs)	Represent the Levi-Civita symbol.
`GramSchmidt`(vlist[, orthonormal])	Apply the Gram-Schmidt process to a set of vectors.
`LC`(f, *gens, **args)	Return the leading coefficient of `f`.
`LM`(f, *gens, **args)	Return the leading monomial of `f`.
`LT`(f, *gens, **args)	Return the leading term of `f`.
`N`(x[, n])	Calls x.evalf(n, **options).
`POSform`(variables, minterms[, dontcares])	The POSform function uses simplified_pairs and a redundant-group eliminating algorithm to convert the list of all input combinations that generate '1' (the minterms) into the smallest product-of-sums form.

continues on next page

Table 67 – continued from previous page

SOPform(variables, minterms[, dontcares])	The SOPform function uses simplified_pairs and a redundant group- eliminating algorithm to convert the list of all input combos that generate '1' (the minterms) into the smallest sum-of-products form.
Ynm_c(n, m, theta, phi)	Conjugate spherical harmonics defined as
abundance(n)	Returns the difference between the sum of the positive proper divisors of a number and the number.
apart(f[, x, full])	Compute partial fraction decomposition of a rational function.
apart_list(f[, x, dummies])	Compute partial fraction decomposition of a rational function and return the result in structured form.
apply_finite_diff(order, x_list, y_list[, x0])	Calculates the finite difference approximation of the derivative of requested order at `x0` from points provided in `x_list` and `y_list`.
approximants(l[, X, simplify])	Return a generator for consecutive Pade approximants for a series.
are_similar(e1, e2)	Are two geometrical entities similar.
arity(cls)	Return the arity of the function if it is known, else None.
ask(proposition[, assumptions, context])	Function to evaluate the proposition with assumptions.
assemble_partfrac_list(partial_list)	Reassemble a full partial fraction decomposition from a structured result obtained by the function `apart_list`.
assuming(*assumptions)	Context manager for assumptions.
banded(*args, **kwargs)	Returns a SparseMatrix from the given dictionary describing the diagonals of the matrix.
besselsimp(expr)	Simplify bessel-type functions.
binomial_coefficients(n)	Return a dictionary containing pairs $(k1, k2) : C_k n$ where $C_k n$ are binomial coefficients and $n = k1 + k2$.
binomial_coefficients_list(n)	Return a list of binomial coefficients as rows of the Pascal's triangle.
block_collapse(expr)	Evaluates a block matrix expression
blockcut(expr, rowsizes, colsizes)	Cut a matrix expression into Blocks
bool_map(bool1, bool2)	Return the simplified version of *bool1*, and the mapping of variables that makes the two expressions *bool1* and *bool2* represent the same logical behaviour for some correspondence between the variables of each.
bottom_up(rv, F[, atoms, nonbasic])	Apply `F` to all expressions in an expression tree from the bottom up.
bspline_basis(d, knots, n, x)	The n-th B-spline at x of degree d with knots.
bspline_basis_set(d, knots, x)	Return the `len(knots)-d-1` B-splines at *x* of degree *d* with *knots*.
cacheit(func)	
cancel(f, *gens[, _signsimp])	Cancel common factors in a rational function `f`.
capture(func)	Return the printed output of func().
casoratian(seqs, n[, zero])	Given linear difference operator L of order 'k' and homogeneous equation Ly = 0 we want to compute kernel of L, which is a set of 'k' sequences: a(n), b(n), .
cbrt(arg[, evaluate])	Returns the principal cube root.
ccode(expr[, assign_to, standard])	Converts an expr to a string of c code
centroid(*args)	Find the centroid (center of mass) of the collection containing only Points, Segments or Polygons.

continues on next page

Table 67 – continued from previous page

`chebyshevt_poly`(n[, x, polys])	Generates the Chebyshev polynomial of the first kind *T_n(x)*.
`chebyshevu_poly`(n[, x, polys])	Generates the Chebyshev polynomial of the second kind *U_n(x)*.
`check_assumptions`(expr[, against])	Checks whether assumptions of `expr` match the T/F assumptions given (or possessed by `against`).
`checkodesol`(ode, sol[, func, order, ...])	Substitutes `sol` into `ode` and checks that the result is `0`.
`checkpdesol`(pde, sol[, func, solve_for_func])	Checks if the given solution satisfies the partial differential equation.
`checksol`(f, symbol[, sol])	Checks whether sol is a solution of equation f == 0.
`classify_ode`(eq[, func, dict, ics, prep, ...])	Returns a tuple of possible `dsolve()` classifications for an ODE.
`classify_pde`(eq[, func, dict, prep])	Returns a tuple of possible pdsolve() classifications for a PDE.
`closest_points`(*args)	Return the subset of points from a set of points that were the closest to each other in the 2D plane.
`cofactors`(f, g, *gens, **args)	Compute GCD and cofactors of `f` and `g`.
`collect`(expr, syms[, func, evaluate, exact, ...])	Collect additive terms of an expression.
`collect_const`(expr, *vars[, Numbers])	A non-greedy collection of terms with similar number coefficients in an Add expr.
`combsimp`(expr)	Simplify combinatorial expressions.
`comp`(z1, z2[, tol])	Return a bool indicating whether the error between z1 and z2 is le `tol`.
`compose`(f, g, *gens, **args)	Compute functional composition `f(g)`.
`composite`(nth)	Return the nth composite number, with the composite numbers indexed as composite(1) = 4, composite(2) = 6, etc....
`compositepi`(n)	Return the number of positive composite numbers less than or equal to n.
`connect_regions`(region_map, hubmap, ...)	given a mapping of regions to lists of hubs, and hubs to regions, creates a set of arcs between hubs such that:
`construct_domain`(obj, **args)	Construct a minimal domain for a list of expressions.
`content`(f, *gens, **args)	Compute GCD of coefficients of `f`.
`continued_fraction`(a)	Return the continued fraction representation of a Rational or quadratic irrational.
`continued_fraction_convergents`(cf)	Return an iterator over the convergents of a continued fraction (cf).
`continued_fraction_iterator`(x)	Return continued fraction expansion of x as iterator.
`continued_fraction_periodic`(p, q[, d, s])	Find the periodic continued fraction expansion of a quadratic irrational.
`continued_fraction_reduce`(cf)	Reduce a continued fraction to a rational or quadratic irrational.
`convex_hull`(*args[, polygon])	The convex hull surrounding the Points contained in the list of entities.
`convolution`(a, b[, cycle, dps, prime, ...])	Performs convolution by determining the type of desired convolution using hints.
`cosine_transform`(f, x, k, **hints)	Compute the unitary, ordinary-frequency cosine transform of *f*, defined as
`count_ops`(expr[, visual])	Return a representation (integer or expression) of the operations in expr.
`count_roots`(f[, inf, sup])	Return the number of roots of `f` in `[inf, sup]` interval.
`covering_product`(a, b)	Returns the covering product of given sequences.

continues on next page

Table 67 – continued from previous page

cse(exprs[, symbols, optimizations, ...])	Perform common subexpression elimination on an expression.
cxxcode(expr[, assign_to, standard])	C++ equivalent of ccode().
cycle_length(f, x0[, nmax, values])	For a given iterated sequence, return a generator that gives the length of the iterated cycle (lambda) and the length of terms before the cycle begins (mu); if values is True then the terms of the sequence will be returned instead.
cyclotomic_poly(n[, x, polys])	Generates cyclotomic polynomial of order *n* in *x*.
decompogen(f, symbol)	Computes General functional decomposition of f. Given an expression f, returns a list [f_1, f_2, ..., f_n], where:: f = f_1 o f_2 o ... f_n = f_1(f_2(... f_n)).
decompose(f, *gens, **args)	Compute functional decomposition of f.
default_sort_key(item[, order])	Return a key that can be used for sorting.
deg(r)	Return the degree value for the given radians (pi = 180 degrees).
degree(f[, gen])	Return the degree of f in the given variable.
degree_list(f, *gens, **args)	Return a list of degrees of f in all variables.
denom(expr)	
derive_by_array(expr, dx)	Derivative by arrays.
det(matexpr)	Matrix Determinant
det_quick(M[, method])	Return det(M) assuming that either there are lots of zeros or the size of the matrix is small.
diag(*values[, strict, unpack])	Returns a matrix with the provided values placed on the diagonal.
diagonalize_vector(vector)	
dict_merge(*dicts)	Merge dictionaries into a single dictionary.
diff(f, *symbols, **kwargs)	Differentiate f with respect to symbols.
difference_delta(expr[, n, step])	Difference Operator.
differentiate_finite(expr, *symbols[, ...])	Differentiate expr and replace Derivatives with finite differences.
diophantine(eq[, param, syms, permute])	Simplify the solution procedure of diophantine equation eq by converting it into a product of terms which should equal zero.
discrete_log(n, a, b[, order, prime_order])	Compute the discrete logarithm of a to the base b modulo n.
discriminant(f, *gens, **args)	Compute discriminant of f.
div(f, g, *gens, **args)	Compute polynomial division of f and g.
divisor_count(n[, modulus, proper])	Return the number of divisors of n.
divisors(n[, generator, proper])	Return all divisors of n sorted from 1..n by default.
dotprint(expr[, styles, atom, maxdepth, ...])	DOT description of a SymPy expression tree
dsolve(eq[, func, hint, simplify, ics, xi, ...])	Solves any (supported) kind of ordinary differential equation and system of ordinary differential equations.
egyptian_fraction(r[, algorithm])	Return the list of denominators of an Egyptian fraction expansion [1]_ of the said rational *r*.
epath(path[, expr, func, args, kwargs])	Manipulate parts of an expression selected by a path.
euler_equations(L[, funcs, vars])	Find the Euler-Lagrange equations [1]_ for a given Lagrangian.
evaluate(x)	Control automatic evaluation
expand(e[, deep, modulus, power_base, ...])	Expand an expression using methods given as hints.

continues on next page

Table 67 – continued from previous page

`expand_complex`(expr[, deep])	Wrapper around expand that only uses the complex hint.
`expand_func`(expr[, deep])	Wrapper around expand that only uses the func hint.
`expand_log`(expr[, deep, force, factor])	Wrapper around expand that only uses the log hint.
`expand_mul`(expr[, deep])	Wrapper around expand that only uses the mul hint.
`expand_multinomial`(expr[, deep])	Wrapper around expand that only uses the multinomial hint.
`expand_power_base`(expr[, deep, force])	Wrapper around expand that only uses the power_base hint.
`expand_power_exp`(expr[, deep])	Wrapper around expand that only uses the power_exp hint.
`expand_trig`(expr[, deep])	Wrapper around expand that only uses the trig hint.
`exptrigsimp`(expr)	Simplifies exponential / trigonometric / hyperbolic functions.
`exquo`(f, g, *gens, **args)	Compute polynomial exact quotient of `f` and `g`.
`eye`(*args, **kwargs)	Create square identity matrix n x n
`factor`(f, *gens[, deep])	Compute the factorization of expression, `f`, into irreducibles.
`factor_list`(f, *gens, **args)	Compute a list of irreducible factors of `f`.
`factor_nc`(expr)	Return the factored form of `expr` while handling non-commutative expressions.
`factor_terms`(expr[, radical, clear, ...])	Remove common factors from terms in all arguments without changing the underlying structure of the expr.
`factorint`(n[, limit, use_trial, use_rho, ...])	Given a positive integer `n`, `factorint(n)` returns a dict containing the prime factors of `n` as keys and their respective multiplicities as values.
`factorrat`(rat[, limit, use_trial, use_rho, ...])	Given a Rational `r`, `factorrat(r)` returns a dict containing the prime factors of `r` as keys and their respective multiplicities as values.
`failing_assumptions`(expr, **assumptions)	Return a dictionary containing assumptions with values not matching those of the passed assumptions.
`farthest_points`(*args)	Return the subset of points from a set of points that were the furthest apart from each other in the 2D plane.
`fcode`(expr[, assign_to])	Converts an expr to a string of fortran code
`fft`(seq[, dps])	Performs the Discrete Fourier Transform (**DFT**) in the complex domain.
`field`(symbols, domain[, order])	Construct new rational function field returning (field, x1, ..., xn).
`field_isomorphism`(a, b, *[, fast])	Find an embedding of one number field into another.
`filldedent`(s[, w])	Strips leading and trailing empty lines from a copy of `s`, then dedents, fills and returns it.
`finite_diff_weights`(order, x_list[, x0])	Calculates the finite difference weights for an arbitrarily spaced one-dimensional grid (`x_list`) for derivatives at `x0` of order 0, 1, ..., up to `order` using a recursive formula.
`flatten`(iterable[, levels, cls])	Recursively denest iterable containers.
`fourier_series`(f[, limits, finite])	Computes the Fourier trigonometric series expansion.
`fourier_transform`(f, x, k, **hints)	Compute the unitary, ordinary-frequency Fourier transform of `f`, defined as
`fps`(f[, x, x0, dir, hyper, order, rational, ...])	Generates Formal Power Series of `f`.
`fraction`(expr[, exact])	Returns a pair with expression's numerator and denominator.

continues on next page

Table 67 – continued from previous page

`fu`(rv[, measure])	Attempt to simplify expression by using transformation rules given in the algorithm by Fu et al.
`fwht`(seq)	Performs the Walsh Hadamard Transform (**WHT**), and uses Hadamard ordering for the sequence.
`galois_group`(f, *gens[, by_name, max_tries, ...])	Compute the Galois group for polynomials *f* up to degree 6.
`gammasimp`(expr)	Simplify expressions with gamma functions.
`gcd`(f[, g])	Compute GCD of `f` and `g`.
`gcd_list`(seq, *gens, **args)	Compute GCD of a list of polynomials.
`gcd_terms`(terms[, isprimitive, clear, fraction])	Compute the GCD of `terms` and put them together.
`gcdex`(f, g, *gens, **args)	Extended Euclidean algorithm of `f` and `g`.
`generate`([num_regions, hubs_per_region, ...])	generates a random network with all parameters required to initialize a ToyBabyModel
`getLogger`([name])	Return a logger with the specified name, creating it if necessary.
`get_contraction_structure`(expr)	Determine dummy indices of `expr` and describe its structure
`get_indices`(expr)	Determine the outer indices of expression `expr`
`gff`(f, *gens, **args)	Compute greatest factorial factorization of `f`.
`gff_list`(f, *gens, **args)	Compute a list of greatest factorial factors of `f`.
`glsl_code`(expr[, assign_to])	Converts an expr to a string of GLSL code
`groebner`(F, *gens, **args)	Computes the reduced Groebner basis for a set of polynomials.
`ground_roots`(f, *gens, **args)	Compute roots of `f` by factorization in the ground domain.
`group`(seq[, multiple])	Splits a sequence into a list of lists of equal, adjacent elements.
`gruntz`(e, z, z0[, dir])	Compute the limit of e(z) at the point z0 using the Gruntz algorithm.
`hadamard_product`(*matrices)	Return the elementwise (aka Hadamard) product of matrices.
`half_gcdex`(f, g, *gens, **args)	Half extended Euclidean algorithm of `f` and `g`.
`hankel_transform`(f, r, k, nu, **hints)	Compute the Hankel transform of *f*, defined as
`has_dups`(seq)	Return True if there are any duplicate elements in `seq`.
`has_variety`(seq)	Return True if there are any different elements in `seq`.
`hermite_poly`(n[, x, polys])	Generates the Hermite polynomial *H_n(x)*.
`hermite_prob_poly`(n[, x, polys])	Generates the probabilist's Hermite polynomial *He_n(x)*.
`hessian`(f, varlist[, constraints])	Compute Hessian matrix for a function f wrt parameters in varlist which may be given as a sequence or a row/column vector.
`homogeneous_order`(eq, *symbols)	Returns the order *n* if *g* is homogeneous and `None` if it is not homogeneous.
`horner`(f, *gens, **args)	Rewrite a polynomial in Horner form.
`hyperexpand`(f[, allow_hyper, rewrite, place])	Expand hypergeometric functions.
`hypersimilar`(f, g, k)	Returns True if `f` and `g` are hyper-similar.
`hypersimp`(f, k)	Given combinatorial term f(k) simplify its consecutive term ratio i.e. f(k+1)/f(k).
`idiff`(eq, y, x[, n])	Return `dy/dx` assuming that `eq == 0`.
`ifft`(seq[, dps])	Performs the Discrete Fourier Transform (**DFT**) in the complex domain.
`ifwht`(seq)	Performs the Walsh Hadamard Transform (**WHT**), and uses Hadamard ordering for the sequence.

continues on next page

Table 67 – continued from previous page

igcd(*args)	Computes nonnegative integer greatest common divisor.
ilcm(*args)	Computes integer least common multiple.
imageset(*args)	Return an image of the set under transformation f.
init_printing([pretty_print, order, ...])	Initializes pretty-printer depending on the environment.
init_session([ipython, pretty_print, order, ...])	Initialize an embedded IPython or Python session.
integer_log(y, x)	Returns (e, bool) where e is the largest nonnegative integer such that $\|y\| \geq \|x^e\|$ and bool is True if $y = x^e$.
integer_nthroot(y, n)	Return a tuple containing x = floor(y**(1/n)) and a boolean indicating whether the result is exact (that is, whether x**n == y).
integrate(f, var, ...)	
interactive_traversal(expr)	Traverse a tree asking a user which branch to choose.
interpolate(data, x)	Construct an interpolating polynomial for the data points evaluated at point x (which can be symbolic or numeric).
interpolating_poly(n, x[, X, Y])	Construct Lagrange interpolating polynomial for n data points.
interpolating_spline(d, x, X, Y)	Return spline of degree *d*, passing through the given *X* and *Y* values.
intersecting_product(a, b)	Returns the intersecting product of given sequences.
intersection(*entities[, pairwise])	The intersection of a collection of GeometryEntity instances.
intervals(F[, all, eps, inf, sup, strict, ...])	Compute isolating intervals for roots of f.
intt(seq, prime)	Performs the Number Theoretic Transform (**NTT**), which specializes the Discrete Fourier Transform (**DFT**) over quotient ring *Z/pZ* for prime *p* instead of complex numbers *C*.
inv_quick(M)	Return the inverse of M, assuming that either there are lots of zeros or the size of the matrix is small.
inverse_cosine_transform(F, k, x, **hints)	Compute the unitary, ordinary-frequency inverse cosine transform of *F*, defined as
inverse_fourier_transform(F, k, x, **hints)	Compute the unitary, ordinary-frequency inverse Fourier transform of *F*, defined as
inverse_hankel_transform(F, k, r, nu, **hints)	Compute the inverse Hankel transform of *F* defined as
inverse_laplace_transform(F, s, t[, plane])	Compute the inverse Laplace transform of *F(s)*, defined as
inverse_mellin_transform(F, s, x, strip, **hints)	Compute the inverse Mellin transform of *F(s)* over the fundamental strip given by strip=(a, b).
inverse_mobius_transform(seq[, subset])	Performs the Mobius Transform for subset lattice with indices of sequence as bitmasks.
inverse_sine_transform(F, k, x, **hints)	Compute the unitary, ordinary-frequency inverse sine transform of *F*, defined as
invert(f, g, *gens, **args)	Invert f modulo g when possible.
is_abundant(n)	Returns True if n is an abundant number, else False.
is_amicable(m, n)	Returns True if the numbers *m* and *n* are "amicable", else False.
is_convex(f, *syms[, domain])	Determines the convexity of the function passed in the argument.
is_decreasing(expression[, interval, symbol])	Return whether the function is decreasing in the given interval.
is_deficient(n)	Returns True if n is a deficient number, else False.

continues on next page

Table 67 – continued from previous page

is_increasing(expression[, interval, symbol])	Return whether the function is increasing in the given interval.
is_mersenne_prime(n)	Returns True if n is a Mersenne prime, else False.
is_monotonic(expression[, interval, symbol])	Return whether the function is monotonic in the given interval.
is_nthpow_residue(a, n, m)	Returns True if x**n == a (mod m) has solutions.
is_perfect(n)	Returns True if n is a perfect number, else False.
is_primitive_root(a, p)	Returns True if a is a primitive root of p.
is_quad_residue(a, p)	Returns True if a (mod p) is in the set of squares mod p, i.e a % p in set([i**2 % p for i in range(p)]).
is_strictly_decreasing(expression[, ...])	Return whether the function is strictly decreasing in the given interval.
is_strictly_increasing(expression[, ...])	Return whether the function is strictly increasing in the given interval.
is_zero_dimensional(F, *gens, **args)	Checks if the ideal generated by a Groebner basis is zero-dimensional.
isolate(alg[, eps, fast])	Find a rational isolating interval for a real algebraic number.
isprime(n)	Test if n is a prime number (True) or not (False).
itermonomials(variables, max_degrees[, ...])	max_degrees and min_degrees are either both integers or both lists.
jacobi_normalized(n, a, b, x)	Jacobi polynomial $P_n^{left(alpha, betaright)}(x)$.
jacobi_poly(n, a, b[, x, polys])	Generates the Jacobi polynomial *P_n^{(a,b)}(x)*.
jacobi_symbol(m, n)	Returns the Jacobi symbol *(m / n)*.
jn_zeros(n, k[, method, dps])	Zeros of the spherical Bessel function of the first kind.
jordan_cell(eigenval, n)	Create a Jordan block:
jscode(expr[, assign_to])	Converts an expr to a string of javascript code
julia_code(expr[, assign_to])	Converts *expr* to a string of Julia code.
kronecker_product(*matrices)	The Kronecker product of two or more arguments.
kroneckersimp(expr)	Simplify expressions with KroneckerDelta.
laguerre_poly(n[, x, alpha, polys])	Generates the Laguerre polynomial *L_n^{(alpha)}(x)*.
lambdify(args, expr[, modules, printer, ...])	Convert a SymPy expression into a function that allows for fast numeric evaluation.
laplace_transform(f, t, s[, legacy_matrix])	Compute the Laplace Transform *F(s)* of *f(t)*,
lcm(f[, g])	Compute LCM of f and g.
lcm_list(seq, *gens, **args)	Compute LCM of a list of polynomials.
legendre_poly(n[, x, polys])	Generates the Legendre polynomial *P_n(x)*.
legendre_symbol(a, p)	Returns the Legendre symbol *(a / p)*.
limit(e, z, z0[, dir])	Computes the limit of e(z) at the point z0.
limit_seq(expr[, n, trials])	Finds the limit of a sequence as index n tends to infinity.
line_integrate(field, Curve, variables)	Compute the line integral.
linear_eq_to_matrix(equations, *symbols)	Converts a given System of Equations into Matrix form.
linsolve(system, *symbols)	Solve system of N linear equations with M variables; both underdetermined and overdetermined systems are supported.
list2numpy(l[, dtype])	Converts Python list of SymPy expressions to a NumPy array.
logcombine(expr[, force])	Takes logarithms and combines them using the following rules:
maple_code(expr[, assign_to])	Converts expr to a string of Maple code.
mathematica_code(expr, **settings)	Converts an expr to a string of the Wolfram Mathematica code

continues on next page

Table 67 – continued from previous page

`matrix2numpy`(m[, dtype])	Converts SymPy's matrix to a NumPy array.
`matrix_multiply_elementwise`(A, B)	Return the Hadamard product (elementwise product) of A and B
`matrix_symbols`(expr)	
`maximum`(f, symbol[, domain])	Returns the maximum value of a function in the given domain.
`mellin_transform`(f, x, s, **hints)	Compute the Mellin transform *F(s)* of *f(x)*,
`memoize_property`(propfunc)	Property decorator that caches the value of potentially expensive *propfunc* after the first evaluation.
`mersenne_prime_exponent`(nth)	Returns the exponent `i` for the nth Mersenne prime (which has the form *2^i - 1*).
`minimal_polynomial`(ex[, x, compose, polys, ...])	Computes the minimal polynomial of an algebraic element.
`minimum`(f, symbol[, domain])	Returns the minimum value of a function in the given domain.
`minpoly`(ex[, x, compose, polys, domain])	This is a synonym for `minimal_polynomial()`.
`mobius_transform`(seq[, subset])	Performs the Mobius Transform for subset lattice with indices of sequence as bitmasks.
`mod_inverse`(a, m)	Return the number c such that, $a times c = 1 pmod{m}$ where c has the same sign as m.
`monic`(f, *gens, **args)	Divide all coefficients of `f` by `LC(f)`.
`multiline_latex`(lhs, rhs[, terms_per_line, ...])	This function generates a LaTeX equation with a multiline right-hand side in an `align*`, `eqnarray` or `IEEEeqnarray` environment.
`multinomial_coefficients`(m, n)	Return a dictionary containing pairs `{(k1,k2,..,km) : C_kn}` where `C_kn` are multinomial coefficients such that `n=k1+k2+..+km`.
`multiplicity`(p, n)	Find the greatest integer m such that p**m divides n.
`n_order`(a, n)	Returns the order of `a` modulo `n`.
`nextprime`(n[, ith])	Return the ith prime greater than n.
`nfloat`(expr[, n, exponent, dkeys])	Make all Rationals in expr Floats except those in exponents (unless the exponents flag is set to True) and those in undefined functions.
`nonlinsolve`(system, *symbols)	Solve system of N nonlinear equations with M variables, which means both under and overdetermined systems are supported.
`not_empty_in`(finset_intersection, *syms)	Finds the domain of the functions in `finset_intersection` in which the `finite_set` is not-empty.
`npartitions`(n[, verbose])	Calculate the partition function P(n), i.e. the number of ways that n can be written as a sum of positive integers.
`nroots`(f[, n, maxsteps, cleanup])	Compute numerical approximations of roots of `f`.
`nsimplify`(expr[, constants, tolerance, ...])	Find a simple representation for a number or, if there are free symbols or if `rational=True`, then replace Floats with their Rational equivalents.
`nsolve`(*args[, dict])	Solve a nonlinear equation system numerically: `nsolve(f, [args,] x0, modules=['mpmath'], **kwargs)`.
`nth_power_roots_poly`(f, n, *gens, **args)	Construct a polynomial with n-th powers of roots of `f`.
`nthroot_mod`(a, n, p[, all_roots])	Find the solutions to `x**n = a mod p`.

continues on next page

Table 67 – continued from previous page

ntt(seq, prime)	Performs the Number Theoretic Transform (**NTT**), which specializes the Discrete Fourier Transform (**DFT**) over quotient ring *Z/pZ* for prime *p* instead of complex numbers *C*.
numbered_symbols([prefix, cls, start, exclude])	Generate an infinite stream of Symbols consisting of a prefix and increasing subscripts provided that they do not occur in `exclude`.
numer(expr)	
octave_code(expr[, assign_to])	Converts *expr* to a string of Octave (or Matlab) code.
ode_order(expr, func)	Returns the order of a given differential equation with respect to func.
ones(*args, **kwargs)	Returns a matrix of ones with `rows` rows and `cols` columns; if `cols` is omitted a square matrix will be returned.
ordered(seq[, keys, default, warn])	Return an iterator of the seq where keys are used to break ties in a conservative fashion: if, after applying a key, there are no ties then no other keys will be computed.
pager_print(expr, **settings)	Prints expr using the pager, in pretty form.
parallel_poly_from_expr(exprs, *gens, **args)	Construct polynomials from expressions.
parse_expr(s[, local_dict, transformations, ...])	Converts the string `s` to a SymPy expression, in `local_dict`.
pde_separate(eq, fun, sep[, strategy])	Separate variables in partial differential equation either by additive or multiplicative separation approach.
pde_separate_add(eq, fun, sep)	Helper function for searching additive separable solutions.
pde_separate_mul(eq, fun, sep)	Helper function for searching multiplicative separable solutions.
pdiv(f, g, *gens, **args)	Compute polynomial pseudo-division of `f` and `g`.
pdsolve(eq[, func, hint, dict, solvefun])	Solves any (supported) kind of partial differential equation.
per(matexpr)	Matrix Permanent
perfect_power(n[, candidates, big, factor])	Return `(b, e)` such that `n` == `b**e` if `n` is a unique perfect power with `e > 1`, else `False` (e.g. 1 is not a perfect power).
periodicity(f, symbol[, check])	Tests the given function for periodicity in the given symbol.
permutedims(expr[, perm, index_order_old, ...])	Permutes the indices of an array.
pexquo(f, g, *gens, **args)	Compute polynomial exact pseudo-quotient of `f` and `g`.
piecewise_exclusive(expr, *[, skip_nan, deep])	Rewrite `Piecewise` with mutually exclusive conditions.
piecewise_fold(expr[, evaluate])	Takes an expression containing a piecewise function and returns the expression in piecewise form.
plot(*args[, show])	Plots a function of a single variable as a curve.
plot_implicit(expr[, x_var, y_var, ...])	A plot function to plot implicit equations / inequalities.
plot_parametric(*args[, show])	Plots a 2D parametric curve.
polarify(eq[, subs, lift])	Turn all numbers in eq into their polar equivalents (under the standard choice of argument).
pollard_pm1(n[, B, a, retries, seed])	Use Pollard's p-1 method to try to extract a nontrivial factor of `n`.
pollard_rho(n[, s, a, retries, seed, ...])	Use Pollard's rho method to try to extract a nontrivial factor of `n`.
poly(expr, *gens, **args)	Efficiently transform an expression into a polynomial.

continues on next page

Table 67 – continued from previous page

poly_from_expr(expr, *gens, **args)	Construct a polynomial from an expression.
posify(eq)	Return eq (with generic symbols made positive) and a dictionary containing the mapping between the old and new symbols.
postfixes(seq)	Generate all postfixes of a sequence.
postorder_traversal(node[, keys])	Do a postorder traversal of a tree.
powdenest(eq[, force, polar])	Collect exponents on powers as assumptions allow.
powsimp(expr[, deep, combine, force, measure])	Reduce expression by combining powers with similar bases and exponents.
pprint(expr, **kwargs)	Prints expr in pretty form.
pprint_try_use_unicode()	See if unicode output is available and leverage it if possible
pprint_use_unicode([flag])	Set whether pretty-printer should use unicode by default
pquo(f, g, *gens, **args)	Compute polynomial pseudo-quotient of f and g.
prefixes(seq)	Generate all prefixes of a sequence.
prem(f, g, *gens, **args)	Compute polynomial pseudo-remainder of f and g.
pretty_print(expr, **kwargs)	Prints expr in pretty form.
preview(expr[, output, viewer, euler, ...])	View expression or LaTeX markup in PNG, DVI, PostScript or PDF form.
prevprime(n)	Return the largest prime smaller than n.
prime(nth)	Return the nth prime, with the primes indexed as prime(1) = 2, prime(2) = 3, etc.
prime_decomp(p[, T, ZK, dK, radical])	Compute the decomposition of rational prime p in a number field.
prime_valuation(I, P)	Compute the P-adic valuation for an integral ideal I.
primefactors(n[, limit, verbose])	Return a sorted list of n's prime factors, ignoring multiplicity and any composite factor that remains if the limit was set too low for complete factorization.
primerange(a[, b])	Generate a list of all prime numbers in the range [2, a), or [a, b).
primitive(f, *gens, **args)	Compute content and the primitive form of f.
primitive_element(extension[, x, ex, polys])	Find a single generator for a number field given by several generators.
primitive_root(p)	Returns the smallest primitive root or None.
primorial(n[, nth])	Returns the product of the first n primes (default) or the primes less than or equal to n (when nth=False).
print_ccode(expr, **settings)	Prints C representation of the given expression.
print_fcode(expr, **settings)	Prints the Fortran representation of the given expression.
print_glsl(expr, **settings)	Prints the GLSL representation of the given expression.
print_gtk(x[, start_viewer])	Print to Gtkmathview, a gtk widget capable of rendering MathML.
print_jscode(expr, **settings)	Prints the Javascript representation of the given expression.
print_latex(expr, **settings)	Prints LaTeX representation of the given expression.
print_maple_code(expr, **settings)	Prints the Maple representation of the given expression.
print_mathml(expr[, printer])	Prints a pretty representation of the MathML code for expr.
print_python(expr, **settings)	Print output of python() function
print_rcode(expr, **settings)	Prints R representation of the given expression.
print_tree(node[, assumptions])	Prints a tree representation of "node".
prod(a[, start])	Return product of elements of a. Start with int 1 so if only

continues on next page

Table 67 – continued from previous page

product(*args, **kwargs)	Compute the product.
proper_divisor_count(n[, modulus])	Return the number of proper divisors of n.
proper_divisors(n[, generator])	Return all divisors of n except n, sorted by default.
public(obj)	Append obj's name to global __all__ variable (call site).
pycode(expr, **settings)	Converts an expr to a string of Python code
python(expr, **settings)	Return Python interpretation of passed expression (can be passed to the exec() function without any modifications)
quadratic_congruence(a, b, c, p)	Find the solutions to ``a x**2 + b x + c = 0 mod p.
quadratic_residues(p)	Returns the list of quadratic residues.
quo(f, g, *gens, **args)	Compute polynomial quotient of f and g.
rad(d)	Return the radian value for the given degrees (pi = 180 degrees).
radsimp(expr[, symbolic, max_terms])	Rationalize the denominator by removing square roots.
randMatrix(r[, c, min, max, seed, ...])	Create random matrix with dimensions r x c.
random_poly(x, n, inf, sup[, domain, polys])	Generates a polynomial of degree n with coefficients in [inf, sup].
randprime(a, b)	Return a random prime number in the range [a, b).
rational_interpolate(data, degnum[, X])	Returns a rational interpolation, where the data points are element of any integral domain.
ratsimp(expr)	Put an expression over a common denominator, cancel and reduce.
ratsimpmodprime(expr, G, *gens[, quick, ...])	Simplifies a rational expression expr modulo the prime ideal generated by G.
rcode(expr[, assign_to])	Converts an expr to a string of r code
rcollect(expr, *vars)	Recursively collect sums in an expression.
real_root(arg[, n, evaluate])	Return the real *n*'th-root of *arg* if possible.
real_roots(f[, multiple])	Return a list of real roots with multiplicities of f.
reduce_abs_inequalities(exprs, gen)	Reduce a system of inequalities with nested absolute values.
reduce_abs_inequality(expr, rel, gen)	Reduce an inequality with nested absolute values.
reduce_inequalities(inequalities[, symbols])	Reduce a system of inequalities with rational coefficients.
reduced(f, G, *gens, **args)	Reduces a polynomial f modulo a set of polynomials G.
refine(expr[, assumptions])	Simplify an expression using assumptions.
refine_root(f, s, t[, eps, steps, fast, ...])	Refine an isolating interval of a root to the given precision.
register_handler(key, handler)	Register a handler in the ask system.
rem(f, g, *gens, **args)	Compute polynomial remainder of f and g.
remove_handler(key, handler)	Removes a handler from the ask system.
reshape(seq, how)	Reshape the sequence according to the template in how.
residue(expr, x, x0)	Finds the residue of expr at the point x=x0.
resultant(f, g, *gens[, includePRS])	Compute resultant of f and g.
ring(symbols, domain[, order])	Construct a polynomial ring returning (ring, x_1, ..., x_n).
root(arg, n[, k, evaluate])	Returns the *k*-th *n*-th root of arg.
rootof(f, x[, index, radicals, expand])	An indexed root of a univariate polynomial.
roots(f, *gens[, auto, cubics, trig, ...])	Computes symbolic roots of a univariate polynomial.
rot_axis1(theta)	Returns a rotation matrix for a rotation of theta (in radians) about the 1-axis.

continues on next page

Table 67 – continued from previous page

rot_axis2(theta)	Returns a rotation matrix for a rotation of theta (in radians) about the 2-axis.
rot_axis3(theta)	Returns a rotation matrix for a rotation of theta (in radians) about the 3-axis.
rot_ccw_axis1(theta)	Returns a rotation matrix for a rotation of theta (in radians) about the 1-axis.
rot_ccw_axis2(theta)	Returns a rotation matrix for a rotation of theta (in radians) about the 2-axis.
rot_ccw_axis3(theta)	Returns a rotation matrix for a rotation of theta (in radians) about the 3-axis.
rot_givens(i, j, theta[, dim])	Returns a a Givens rotation matrix, a a rotation in the plane spanned by two coordinates axes.
rotations(s[, dir])	Return a generator giving the items in s as list where each subsequent list has the items rotated to the left (default) or right (dir=-1) relative to the previous list.
round_two(T[, radicals])	Zassenhaus's "Round 2" algorithm.
rsolve(f, y[, init])	Solve univariate recurrence with rational coefficients.
rsolve_hyper(coeffs, f, n, **hints)	Given linear recurrence operator *operatorname{L}* of order k with polynomial coefficients and inhomogeneous equation *operatorname{L}* $y = f$ we seek for all hypergeometric solutions over field K of characteristic zero.
rsolve_poly(coeffs, f, n[, shift])	Given linear recurrence operator *operatorname{L}* of order k with polynomial coefficients and inhomogeneous equation *operatorname{L}* $y = f$, where f is a polynomial, we seek for all polynomial solutions over field K of characteristic zero.
rsolve_ratio(coeffs, f, n, **hints)	Given linear recurrence operator *operatorname{L}* of order k with polynomial coefficients and inhomogeneous equation *operatorname{L}* $y = f$, where f is a polynomial, we seek for all rational solutions over field K of characteristic zero.
run_timed(params, parameter_name)	generates an instance of TestBabyModel with given parameters, sympifies it, and follows the sequence of steps
rust_code(expr[, assign_to])	Converts an expr to a string of Rust code
satisfiable(expr[, algorithm, all_models, ...])	Check satisfiability of a propositional sentence.
separatevars(expr[, symbols, dict, force])	Separates variables in an expression, if possible.
sequence(seq[, limits])	Returns appropriate sequence object.
series(expr[, x, x0, n, dir])	Series expansion of expr around point $x = x0$.
seterr([divide])	Should SymPy raise an exception on 0/0 or return a nan?
sfield(exprs, *symbols, **options)	Construct a field deriving generators and domain from options and input expressions.
shape()	Return the shape of the *expr* as a tuple.
sift(seq, keyfunc[, binary])	Sift the sequence, seq according to keyfunc.
signsimp(expr[, evaluate])	Make all Add sub-expressions canonical wrt sign.
simplify(expr[, ratio, measure, rational, ...])	Simplifies the given expression.
simplify_logic(expr[, form, deep, force, ...])	This function simplifies a boolean function to its simplified version in SOP or POS form.
sine_transform(f, x, k, **hints)	Compute the unitary, ordinary-frequency sine transform of f, defined as
singularities(expression, symbol[, domain])	Find singularities of a given function.
singularityintegrate(f, x)	This function handles the indefinite integrations of Singularity functions.

continues on next page

Table 67 – continued from previous page

`smtlib_code`(expr[, auto_assert, ...])	Converts `expr` to a string of smtlib code.
`solve`(f, *symbols, **flags)	Algebraically solves equations and systems of equations.
`solve_linear`(lhs[, rhs, symbols, exclude])	Return a tuple derived from `f = lhs - rhs` that is one of the following: `(0, 1)`, `(0, 0)`, `(symbol, solution)`, `(n, d)`.
`solve_linear_system`(system, *symbols, **flags)	Solve system of N linear equations with M variables, which means both under- and overdetermined systems are supported.
`solve_linear_system_LU`(matrix, syms)	Solves the augmented matrix system using `LUsolve` and returns a dictionary in which solutions are keyed to the symbols of *syms* as ordered.
`solve_poly_inequality`(poly, rel)	Solve a polynomial inequality with rational coefficients.
`solve_poly_system`(seq, *gens[, strict])	Return a list of solutions for the system of polynomial equations or else None.
`solve_rational_inequalities`(eqs)	Solve a system of rational inequalities with rational coefficients.
`solve_triangulated`(polys, *gens, **args)	Solve a polynomial system using Gianni-Kalkbrenner algorithm.
`solve_undetermined_coeffs`(equ, coeffs, ...)	Solve a system of equations in k parameters that is formed by matching coefficients in variables `coeffs` that are on factors dependent on the remaining variables (or those given explicitly by `syms`.
`solve_univariate_inequality`(expr, gen[, ...])	Solves a real univariate inequality.
`solveset`(f[, symbol, domain])	Solves a given inequality or equation with set as output
`speed_accuracy_test`(params, parameter_name, p)	performs a run_timed execution with given parameters, takes the returned values, and measures the effect of increasing and
`sqf`(f, *gens, **args)	Compute square-free factorization of `f`.
`sqf_list`(f, *gens, **args)	Compute a list of square-free factors of `f`.
`sqf_norm`(f, *gens, **args)	Compute square-free norm of `f`.
`sqf_part`(f, *gens, **args)	Compute square-free part of `f`.
`sqrt`(arg[, evaluate])	Returns the principal square root.
`sqrt_mod`(a, p[, all_roots])	Find a root of `x**2 = a mod p`.
`sqrt_mod_iter`(a, p[, domain])	Iterate over solutions to `x**2 = a mod p`.
`sqrtdenest`(expr[, max_iter])	Denests sqrts in an expression that contain other square roots if possible, otherwise returns the expr unchanged.
`sring`(exprs, *symbols, **options)	Construct a ring deriving generators and domain from options and input expressions.
`stationary_points`(f, symbol[, domain])	Returns the stationary points of a function (where derivative of the function is 0) in the given domain.
`sturm`(f, *gens, **args)	Compute Sturm sequence of `f`.
`subresultants`(f, g, *gens, **args)	Compute subresultant PRS of `f` and `g`.
`subsets`(seq[, k, repetition])	Generates all *k*-subsets (combinations) from an *n*-element set, `seq`.
`substitution`(system, symbols[, result, ...])	Solves the *system* using substitution method.
`summation`(f, *symbols, **kwargs)	Compute the summation of f with respect to symbols.
`swinnerton_dyer_poly`(n[, x, polys])	Generates n-th Swinnerton-Dyer polynomial in *x*.
`symarray`(prefix, shape, **kwargs)	Create a numpy ndarray of symbols (as an object array).
`symbols`(names, *[, cls])	Transform strings into instances of `Symbol` class.
`symmetric_poly`(n, *gens[, polys])	Generates symmetric polynomial of order *n*.
`symmetrize`(F, *gens, **args)	Rewrite a polynomial in terms of elementary symmetric polynomials.

continues on next page

Table 67 – continued from previous page

`sympify`(a[, locals, convert_xor, strict, ...])	Converts an arbitrary expression to a type that can be used inside SymPy.
`take`(iter, n)	Return `n` items from `iter` iterator.
`tensorcontraction`(array, *contraction_axes)	Contraction of an array-like object on the specified axes.
`tensordiagonal`(array, *diagonal_axes)	Diagonalization of an array-like object on the specified axes.
`tensorproduct`(*args)	Tensor product among scalars or array-like objects.
`terms_gcd`(f, *gens, **args)	Remove GCD of terms from `f`.
`textplot`(expr, a, b[, W, H])	Print a crude ASCII art plot of the SymPy expression 'expr' (which should contain a single symbol, e.g. x or something else) over the interval [a, b].
`threaded`(func)	Apply `func` to sub--elements of an object, including `Add`.
`timed`(func[, setup, limit])	Adaptively measure execution time of a function.
`to_cnf`(expr[, simplify, force])	Convert a propositional logical sentence `expr` to conjunctive normal form: `((A | ~B | ...) & (B | C | ...) & ...)`.
`to_dnf`(expr[, simplify, force])	Convert a propositional logical sentence `expr` to disjunctive normal form: `((A & ~B & ...) | (B & C & ...) | ...)`.
`to_nnf`(expr[, simplify])	Converts `expr` to Negation Normal Form (NNF).
`to_number_field`(extension[, theta, gen, alias])	Express one algebraic number in the field generated by another.
`together`(expr[, deep, fraction])	Denest and combine rational expressions using symbolic methods.
`topological_sort`(graph[, key])	Topological sort of graph's vertices.
`total_degree`(f, *gens)	Return the total_degree of `f` in the given variables.
`trace`(expr)	Trace of a Matrix.
`trailing`(n)	Count the number of trailing zero digits in the binary representation of n, i.e. determine the largest power of 2 that divides n.
`trigsimp`(expr[, inverse])	Returns a reduced expression by using known trig identities.
`trunc`(f, p, *gens, **args)	Reduce `f` modulo a constant `p`.
`unbranched_argument`(arg)	Returns periodic argument of arg with period as infinity.
`unflatten`(iter[, n])	Group `iter` into tuples of length `n`.
`unpolarify`(eq[, subs, exponents_only])	If *p* denotes the projection from the Riemann surface of the logarithm to the complex line, return a simplified version *eq'* of *eq* such that *p(eq') = p(eq)*.
`use`(expr, func[, level, args, kwargs])	Use `func` to transform `expr` at the given level.
`var`(names, **args)	Create symbols and inject them into the global namespace.
`variations`(seq, n[, repetition])	Returns an iterator over the n-sized variations of `seq` (size N).
`vfield`(symbols, domain[, order])	Construct new rational function field and inject generators into global namespace.
`viete`(f[, roots])	Generate Viete's formulas for `f`.
`vring`(symbols, domain[, order])	Construct a polynomial ring and inject `x_1, ..., x_n` into the global namespace.
`wronskian`(functions, var[, method])	Compute Wronskian for [] of functions
`xfield`(symbols, domain[, order])	Construct new rational function field returning (field, (x1, ..., xn)).

continues on next page

Table 67 – continued from previous page

`xring`(symbols, domain[, order])	Construct a polynomial ring returning `(ring, (x_1, ..., x_n))`.
`xthreaded`(func)	Apply `func` to sub--elements of an object, excluding `Add`.
`zeros`(*args, **kwargs)	Returns a matrix of zeros with `rows` rows and `cols` columns; if `cols` is omitted a square matrix will be returned.

Classes

`Abs`(arg)	Return the absolute value of the argument.
`AccumBounds`	alias of `AccumulationBounds`
`Add`(*args[, evaluate, _sympify])	Expression representing addition operation for algebraic group.
`Adjoint`(*args, **kwargs)	The Hermitian adjoint of a matrix expression.
`AlgebraicField`(dom, *ext[, alias])	Algebraic number field QQ(a)
`AlgebraicNumber`(expr[, coeffs, alias])	Class for representing algebraic numbers in SymPy.
`And`(*args)	Logical AND function.
`AppliedPredicate`(predicate, *args)	The class of expressions resulting from applying `Predicate` to the arguments.
`Array`	alias of `ImmutableDenseNDimArray`
`AssumptionsContext`	Set containing default assumptions which are applied to the `ask()` function.
`Atom`(*args)	A parent class for atomic things.
`AtomicExpr`(*args)	A parent class for object which are both atoms and Exprs.
`AutoSympy`(model)	
`Basic`(*args)	Base class for all SymPy objects.
`BlockDiagMatrix`(*mats)	A sparse matrix with block matrices along its diagonals
`BlockMatrix`(*args, **kwargs)	A BlockMatrix is a Matrix comprised of other matrices.
`CRootOf`	alias of `ComplexRootOf`
`Chi`(z)	Cosh integral.
`Ci`(z)	Cosine integral.
`Circle`(*args, **kwargs)	A circle in space.
`Complement`(a, b[, evaluate])	Represents the set difference or relative complement of a set with another set.
`ComplexField`([prec, dps, tol])	Complex numbers up to the given precision.
`ComplexRegion`(sets[, polar])	Represents the Set of all Complex Numbers.
`ComplexRootOf`(f, x[, index, radicals, expand])	Represents an indexed complex root of a polynomial.
`ConditionSet`(sym, condition[, base_set])	Set of elements which satisfies a given condition.
`Contains`(x, s)	Asserts that x is an element of the set S.
`CoordMap`(var_vector, eq_duals, ineq_duals, ...)	
`CosineTransform`(*args)	Class representing unevaluated cosine transforms.
`Curve`(function, limits)	A curve in space.
`DeferredVector`(name, **assumptions)	A vector whose components are deferred (e.g. for use with lambdify).
`DenseNDimArray`(*args, **kwargs)	
`Derivative`(expr, *variables, **kwargs)	Carries out differentiation of the given expression with respect to symbols.

continues on next page

Table 68 – continued from previous page

`Determinant`(mat)	Matrix Determinant
`DiagMatrix`(vector)	Turn a vector into a diagonal matrix.
`DiagonalMatrix`(*args, **kwargs)	DiagonalMatrix(M) will create a matrix expression that behaves as though all off-diagonal elements, *M[i, j]* where *i != j*, are zero.
`DiagonalOf`(*args, **kwargs)	DiagonalOf(M) will create a matrix expression that is equivalent to the diagonal of *M*, represented as a single column matrix.
`Dict`(*args)	Wrapper around the builtin dict object.
`DifferentialMapping`(US, coord2item, ...)	
`DiracDelta`(arg[, k])	The DiracDelta function and its derivatives.
`DisjointUnion`(*sets)	Represents the disjoint union (also known as the external disjoint union) of a finite number of sets.
`Domain`()	Superclass for all domains in the polys domains system.
`DotProduct`(arg1, arg2)	Dot product of vector matrices
`Dummy`([name, dummy_index])	Dummy symbols are each unique, even if they have the same name:
`EPath`(path)	Manipulate expressions using paths.
`Ei`(z)	The classical exponential integral.
`Ellipse`([center, hradius, vradius, eccentricity])	An elliptical GeometryEntity.
`Eq`	alias of `Equality`
`Equality`(lhs, rhs, **options)	An equal relation between two objects.
`Equivalent`(*args)	Equivalence relation.
`Expr`(*args)	Base class for algebraic expressions.
`ExpressionDomain`()	A class for arbitrary expressions.
`FF`	alias of `FiniteField`
`FF_gmpy`	alias of `GMPYFiniteField`
`FF_python`	alias of `PythonFiniteField`
`FallingFactorial`(x, k)	Falling factorial (related to rising factorial) is a double valued function arising in concrete mathematics, hypergeometric functions and series expansions.
`FiniteField`(mod[, symmetric])	Finite field of prime order GF(p)
`FiniteSet`(*args, **kwargs)	Represents a finite set of Sympy expressions.
`Float`(num[, dps, precision])	Represent a floating-point number of arbitrary precision.
`FourierTransform`(*args)	Class representing unevaluated Fourier transforms.
`FractionField`(domain_or_field[, symbols, order])	A class for representing multivariate rational function fields.
`Function`(*args)	Base class for applied mathematical functions.
`FunctionClass`(*args, **kwargs)	Base class for function classes.
`FunctionMatrix`(rows, cols, lamda)	Represents a matrix using a function (`Lambda`) which gives outputs according to the coordinates of each matrix entries.
`GF`	alias of `FiniteField`
`GMPYFiniteField`(mod[, symmetric])	Finite field based on GMPY integers.
`GMPYIntegerRing`()	Integer ring based on GMPY's `mpz` type.
`GMPYRationalField`()	Rational field based on GMPY's `mpq` type.
`Ge`	alias of `GreaterThan`
`GreaterThan`(lhs, rhs, **options)	Class representations of inequalities.
`GroebnerBasis`(F, *gens, **args)	Represents a reduced Groebner basis.
`Gt`	alias of `StrictGreaterThan`
`HadamardPower`(base, exp)	Elementwise power of matrix expressions

continues on next page

Table 68 – continued from previous page

HadamardProduct(*args[, evaluate, check])	Elementwise product of matrix expressions
HankelTransform(*args)	Class representing unevaluated Hankel transforms.
Heaviside(arg[, H0])	Heaviside step function.
ITE(*args)	If-then-else clause.
Identity(n)	The Matrix Identity I - multiplicative identity
Idx(label[, range])	Represents an integer index as an Integer or integer expression.
ImageSet(flambda, *sets)	Image of a set under a mathematical function.
ImmutableDenseMatrix(*args, **kwargs)	Create an immutable version of a matrix.
ImmutableDenseNDimArray(iterable[, shape])	
ImmutableMatrix	alias of ImmutableDenseMatrix
ImmutableSparseMatrix(*args, **kwargs)	Create an immutable version of a sparse matrix.
ImmutableSparseNDimArray([iterable, shape])	
Implies(*args)	Logical implication.
Indexed(base, *args, **kw_args)	Represents a mathematical object with indices.
IndexedBase(label[, shape, offset, strides])	Represent the base or stem of an indexed object
Integer(i)	Represents integer numbers of any size.
IntegerRing()	The domain ZZ representing the integers *mathbb{Z}*.
Integral(function, *symbols, **assumptions)	Represents unevaluated integral.
Intersection(*args, **kwargs)	Represents an intersection of sets as a Set.
Interval(start, end[, left_open, right_open])	Represents a real interval as a Set.
Inverse(mat[, exp])	The multiplicative inverse of a matrix expression
InverseCosineTransform(*args)	Class representing unevaluated inverse cosine transforms.
InverseFourierTransform(*args)	Class representing unevaluated inverse Fourier transforms.
InverseHankelTransform(*args)	Class representing unevaluated inverse Hankel transforms.
InverseLaplaceTransform(*args)	Class representing unevaluated inverse Laplace transforms.
InverseMellinTransform(*args)	Class representing unevaluated inverse Mellin transforms.
InverseSineTransform(*args)	Class representing unevaluated inverse sine transforms.
KroneckerDelta(i, j[, delta_range])	The discrete, or Kronecker, delta function.
KroneckerProduct(*args[, check])	The Kronecker product of two or more arguments.
Lambda(signature, expr)	Lambda(x, expr) represents a lambda function similar to Python's 'lambda x: expr'.
LambertW(x[, k])	The Lambert W function $W(z)$ is defined as the inverse function of $w exp(w)$ [1]_.
LaplaceTransform(*args)	Class representing unevaluated Laplace transforms.
Le	alias of LessThan
LessThan(lhs, rhs, **options)	Class representations of inequalities.
LeviCivita(*args)	Represent the Levi-Civita symbol.
Li(z)	The offset logarithmic integral.
Limit(e, z, z0[, dir])	Represents an unevaluated limit.
Line(*args, **kwargs)	An infinite line in space.
Line2D(p1[, pt, slope])	An infinite line in space 2D.
Line3D(p1[, pt, direction_ratio])	An infinite 3D line in space.
Lt	alias of StrictLessThan
MatAdd(*args[, evaluate, check, _sympify])	A Sum of Matrix Expressions

continues on next page

Table 68 – continued from previous page

`MatMul`(*args[, evaluate, check, _sympify])	A product of matrix expressions
`MatPow`(base, exp[, evaluate])	
`Matrix`	alias of `MutableDenseMatrix`
`MatrixBase`()	Base class for matrix objects.
`MatrixExpr`(*args, **kwargs)	Superclass for Matrix Expressions
`MatrixPermute`(mat, perm[, axis])	Symbolic representation for permuting matrix rows or columns.
`MatrixSlice`(parent, rowslice, colslice)	A MatrixSlice of a Matrix Expression
`MatrixSymbol`(name, n, m)	Symbolic representation of a Matrix object
`Max`(*args)	Return, if possible, the maximum value of the list.
`MellinTransform`(*args)	Class representing unevaluated Mellin transforms.
`Min`(*args)	Return, if possible, the minimum value of the list.
`Mod`(p, q)	Represents a modulo operation on symbolic expressions.
`Monomial`(monom[, gens])	Class representing a monomial, i.e. a product of powers.
`Mul`(*args[, evaluate, _sympify])	Expression representing multiplication operation for algebraic field.
`MutableDenseMatrix`(*args, **kwargs)	
`MutableDenseNDimArray`([iterable, shape])	
`MutableMatrix`	alias of `MutableDenseMatrix`
`MutableSparseMatrix`(*args, **kwargs)	
`MutableSparseNDimArray`([iterable, shape])	
`NDimArray`(iterable[, shape])	N-dimensional array.
`Nand`(*args)	Logical NAND function.
`Ne`	alias of `Unequality`
`Nor`(*args)	Logical NOR function.
`Not`(arg)	Logical Not function (negation)
`Number`(*obj)	Represents atomic numbers in SymPy.
`NumberSymbol`()	
`O`	alias of `Order`
`OmegaPower`(a, b)	Represents ordinal exponential and multiplication terms one of the building blocks of the `Ordinal` class.
`OneMatrix`(m, n[, evaluate])	Matrix whose all entries are ones.
`Options`(gens, args[, flags, strict])	Options manager for polynomial manipulation module.
`Or`(*args)	Logical OR function
`Order`(expr, *args, **kwargs)	Represents the limiting behavior of some function.
`Ordinal`(*terms)	Represents ordinals in Cantor normal form.
`Parabola`([focus, directrix])	A parabolic GeometryEntity.
`Permanent`(mat)	Matrix Permanent
`PermutationMatrix`(perm)	A Permutation Matrix
`Piecewise`(*_args)	Represents a piecewise function.
`Plane`(p1[, a, b])	A plane is a flat, two-dimensional surface.
`Point`(*args, **kwargs)	A point in a n-dimensional Euclidean space.
`Point2D`(*args[, _nocheck])	A point in a 2-dimensional Euclidean space.
`Point3D`(*args[, _nocheck])	A point in a 3-dimensional Euclidean space.
`Poly`(rep, *gens, **args)	Generic class for representing and operating on polynomial expressions.

continues on next page

Table 68 – continued from previous page

Polygon(*args[, n])	A two-dimensional polygon.
PolynomialRing(domain_or_ring[, symbols, order])	A class for representing multivariate polynomial rings.
Pow(b, e[, evaluate])	Defines the expression x**y as "x raised to a power y"
PowerSet(arg[, evaluate])	A symbolic object representing a power set.
Predicate(*args, **kwargs)	Base class for mathematical predicates.
Product(function, *symbols, **assumptions)	Represents unevaluated products.
ProductSet(*sets, **assumptions)	Represents a Cartesian Product of Sets.
PurePoly(rep, *gens, **args)	Class for representing pure polynomials.
PythonFiniteField(mod[, symmetric])	Finite field based on Python's integers.
PythonIntegerRing()	Integer ring based on Python's int type.
PythonRational	alias of PythonMPQ
QQ_gmpy	alias of GMPYRationalField
QQ_python	alias of PythonRationalField
Quaternion([a, b, c, d, real_field, norm])	Provides basic quaternion operations.
Range(*args)	Represents a range of integers.
Rational(p[, q, gcd])	Represents rational numbers (p/q) of any size.
RationalField()	Abstract base class for the domain QQ.
Ray(p1[, p2])	A Ray is a semi-line in the space with a source point and a direction.
Ray2D(p1[, pt, angle])	A Ray is a semi-line in the space with a source point and a direction.
Ray3D(p1[, pt, direction_ratio])	A Ray is a semi-line in the space with a source point and a direction.
RealField([prec, dps, tol])	Real numbers up to the given precision.
RealNumber	alias of Float
RegularPolygon(c, r, n[, rot])	A regular polygon.
Rel	alias of Relational
Rem(p, q)	Returns the remainder when p is divided by q where p is finite and q is not equal to zero.
RisingFactorial(x, k)	Rising factorial (also called Pochhammer symbol [1]_) is a double valued function arising in concrete mathematics, hypergeometric functions and series expansions.
RootOf(f, x[, index, radicals, expand])	Represents a root of a univariate polynomial.
RootSum(expr[, func, x, auto, quadratic])	Represents a sum of all roots of a univariate polynomial.
Segment(p1, p2, **kwargs)	A line segment in space.
Segment2D(p1, p2, **kwargs)	A line segment in 2D space.
Segment3D(p1, p2, **kwargs)	A line segment in a 3D space.
SensitivityMatrix(sympification, duals, ...)	
SeqAdd(*args, **kwargs)	Represents term-wise addition of sequences.
SeqFormula(formula[, limits])	Represents sequence based on a formula.
SeqMul(*args, **kwargs)	Represents term-wise multiplication of sequences.
SeqPer(periodical[, limits])	Represents a periodic sequence.
Set(*args)	The base class for any kind of set.
Shi(z)	Sinh integral.
Si(z)	Sine integral.
Sieve()	An infinite list of prime numbers, implemented as a dynamically growing sieve of Eratosthenes.
SineTransform(*args)	Class representing unevaluated sine transforms.
SingularityFunction(variable, offset, exponent)	Singularity functions are a class of discontinuous functions.
SparseMatrix	alias of MutableSparseMatrix

continues on next page

Table 68 – continued from previous page

SparseNDimArray(*args, **kwargs)	
StrPrinter([settings])	
StrictGreaterThan(lhs, rhs, **options)	Class representations of inequalities.
StrictLessThan(lhs, rhs, **options)	Class representations of inequalities.
Subs(expr, variables, point, **assumptions)	Represents unevaluated substitutions of an expression.
Sum(function, *symbols, **assumptions)	Represents unevaluated summation.
Symbol(name, **assumptions)	Assumptions:
SymmetricDifference(a, b[, evaluate])	Represents the set of elements which are in either of the sets and not in their intersection.
TableForm(data, **kwarg)	Create a nice table representation of data.
TestBabyModel(*args, **kwds)	
Trace(mat)	Matrix Trace
Transpose(*args, **kwargs)	The transpose of a matrix expression.
Triangle(*args, **kwargs)	A polygon with three vertices and three sides.
Tuple(*args, **kwargs)	Wrapper around the builtin tuple object.
Unequality(lhs, rhs, **options)	An unequal relation between two objects.
UnevaluatedExpr(arg, **kwargs)	Expression that is not evaluated unless released.
Union(*args, **kwargs)	Represents a union of sets as a `Set`.
Wild(name[, exclude, properties])	A Wild symbol matches anything, or anything without whatever is explicitly excluded.
WildFunction(*args)	A WildFunction function matches any function (with its arguments).
Xor(*args)	Logical XOR (exclusive OR) function.
Ynm(n, m, theta, phi)	Spherical harmonics defined as
ZZ_gmpy	alias of `GMPYIntegerRing`
ZZ_python	alias of `PythonIntegerRing`
ZeroMatrix(m, n)	The Matrix Zero 0 - additive identity
Znm(n, m, theta, phi)	Real spherical harmonics defined as
acos(arg)	The inverse cosine function.
acosh(arg)	`acosh(x)` is the inverse hyperbolic cosine of `x`.
acot(arg)	The inverse cotangent function.
acoth(arg)	`acoth(x)` is the inverse hyperbolic cotangent of `x`.
acsc(arg)	The inverse cosecant function.
acsch(arg)	`acsch(x)` is the inverse hyperbolic cosecant of `x`.
adjoint(arg)	Conjugate transpose or Hermite conjugation.
airyai(arg)	The Airy function $operatorname{Ai}$ of the first kind.
airyaiprime(arg)	The derivative $operatorname{Ai}^prime$ of the Airy function of the first kind.
airybi(arg)	The Airy function $operatorname{Bi}$ of the second kind.
airybiprime(arg)	The derivative $operatorname{Bi}^prime$ of the Airy function of the first kind.
andre(n)	Andre numbers / Andre function
appellf1(a, b1, b2, c, x, y)	This is the Appell hypergeometric function of two variables as:
arg(arg)	Returns the argument (in radians) of a complex number.
asec(arg)	The inverse secant function.
asech(arg)	`asech(x)` is the inverse hyperbolic secant of `x`.
asin(arg)	The inverse sine function.

continues on next page

Table 68 – continued from previous page

`asinh`(arg)	`asinh(x)` is the inverse hyperbolic sine of `x`.
`assoc_laguerre`(n, alpha, x)	Returns the nth generalized Laguerre polynomial in x, $L_n(x)$.
`assoc_legendre`(n, m, x)	`assoc_legendre(n, m, x)` gives $P_n^m(x)$, where n and m are the degree and order or an expression which is related to the nth order Legendre polynomial, $P_n(x)$ in the following manner:
`atan`(arg)	The inverse tangent function.
`atan2`(y, x)	The function `atan2(y, x)` computes *operatorname{atan}(y/x)* taking two arguments *y* and *x*.
`atanh`(arg)	`atanh(x)` is the inverse hyperbolic tangent of `x`.
`bell`(n[, k_sym, symbols])	Bell numbers / Bell polynomials
`bernoulli`(n[, x])	Bernoulli numbers / Bernoulli polynomials / Bernoulli function
`besseli`(nu, z)	Modified Bessel function of the first kind.
`besselj`(nu, z)	Bessel function of the first kind.
`besselk`(nu, z)	Modified Bessel function of the second kind.
`bessely`(nu, z)	Bessel function of the second kind.
`beta`(x[, y])	The beta integral is called the Eulerian integral of the first kind by Legendre:
`betainc`(*args)	The Generalized Incomplete Beta function is defined as
`betainc_regularized`(*args)	The Generalized Regularized Incomplete Beta function is given by
`binomial`(n, k)	Implementation of the binomial coefficient.
`carmichael`(*args)	Carmichael Numbers:
`cartes`	alias of `product`
`catalan`(n)	Catalan numbers
`ceiling`(arg)	Ceiling is a univariate function which returns the smallest integer value not less than its argument.
`chebyshevt`(n, x)	Chebyshev polynomial of the first kind, $T_n(x)$.
`chebyshevt_root`(n, k)	`chebyshev_root(n, k)` returns the kth root (indexed from zero) of the nth Chebyshev polynomial of the first kind; that is, if $0 le k < n$, `chebyshevt(n, chebyshevt_root(n, k)) == 0`.
`chebyshevu`(n, x)	Chebyshev polynomial of the second kind, $U_n(x)$.
`chebyshevu_root`(n, k)	`chebyshevu_root(n, k)` returns the kth root (indexed from zero) of the nth Chebyshev polynomial of the second kind; that is, if $0 le k < n$, `chebyshevu(n, chebyshevu_root(n, k)) == 0`.
`conjugate`(arg)	Returns the *complex conjugate* **[1]_** of an argument.
`cos`(arg)	The cosine function.
`cosh`(arg)	`cosh(x)` is the hyperbolic cosine of `x`.
`cot`(arg)	The cotangent function.
`coth`(arg)	`coth(x)` is the hyperbolic cotangent of `x`.
`csc`(arg)	The cosecant function.
`csch`(arg)	`csch(x)` is the hyperbolic cosecant of `x`.
`defaultdict`	defaultdict(default_factory=None, /, [...]) --> dict with default factory
`digamma`(z)	The `digamma` function is the first derivative of the `loggamma` function
`dirichlet_eta`(s[, a])	Dirichlet eta function.

continues on next page

Table 68 – continued from previous page

`divisor_sigma`(n[, k])	Calculate the divisor function *sigma_k(n)* for positive integer n
`elliptic_e`(m[, z])	Called with two arguments \$z\$ and \$m\$, evaluates the incomplete elliptic integral of the second kind, defined by
`elliptic_f`(z, m)	The Legendre incomplete elliptic integral of the first kind, defined by
`elliptic_k`(m)	The complete elliptic integral of the first kind, defined by
`elliptic_pi`(n, m[, z])	Called with three arguments \$n\$, \$z\$ and \$m\$, evaluates the Legendre incomplete elliptic integral of the third kind, defined by
`erf`(arg)	The Gauss error function.
`erf2`(x, y)	Two-argument error function.
`erf2inv`(x, y)	Two-argument Inverse error function.
`erfc`(arg)	Complementary Error Function.
`erfcinv`(z)	Inverse Complementary Error Function.
`erfi`(z)	Imaginary error function.
`erfinv`(z)	Inverse Error Function.
`euler`(n[, x])	Euler numbers / Euler polynomials / Euler function
`exp`(arg)	The exponential function, e^x.
`exp_polar`(*args)	Represent a *polar number* (see g-function Sphinx documentation).
`expint`(nu, z)	Generalized exponential integral.
`factorial`(n)	Implementation of factorial function over nonnegative integers.
`factorial2`(arg)	The double factorial *n!!*, not to be confused with *(n!)!*
`ff`	alias of `FallingFactorial`
`fibonacci`(n[, sym])	Fibonacci numbers / Fibonacci polynomials
`floor`(arg)	Floor is a univariate function which returns the largest integer value not greater than its argument.
`frac`(arg)	Represents the fractional part of x
`fresnelc`(z)	Fresnel integral C.
`fresnels`(z)	Fresnel integral S.
`gamma`(arg)	The gamma function
`gegenbauer`(n, a, x)	Gegenbauer polynomial \$C_n^{left(alpharight)}(x)\$.
`genocchi`(n[, x])	Genocchi numbers / Genocchi polynomials / Genocchi function
`hankel1`(nu, z)	Hankel function of the first kind.
`hankel2`(nu, z)	Hankel function of the second kind.
`harmonic`(n[, m])	Harmonic numbers
`hermite`(n, x)	`hermite(n, x)` gives the \$n\$th Hermite polynomial in \$x\$, \$H_n(x)\$.
`hermite_prob`(n, x)	`hermite_prob(n, x)` gives the \$n\$th probabilist's Hermite polynomial in \$x\$, \$He_n(x)\$.
`hn1`(nu, z)	Spherical Hankel function of the first kind.
`hn2`(nu, z)	Spherical Hankel function of the second kind.
`hyper`(ap, bq, z)	The generalized hypergeometric function is defined by a series where the ratios of successive terms are a rational function of the summation index.
`im`(arg)	Returns imaginary part of expression.
`jacobi`(n, a, b, x)	Jacobi polynomial \$P_n^{left(alpha, betaright)}(x)\$.

continues on next page

Table 68 – continued from previous page

jn(nu, z)	Spherical Bessel function of the first kind.
laguerre(n, x)	Returns the nth Laguerre polynomial in x, $L_n(x)$.
legendre(n, x)	legendre(n, x) gives the nth Legendre polynomial of x, $P_n(x)$
lerchphi(*args)	Lerch transcendent (Lerch phi function).
li(z)	The classical logarithmic integral.
ln	alias of log
log(arg[, base])	The natural logarithm function *ln(x)* or *log(x)*.
loggamma(z)	The loggamma function implements the logarithm of the gamma function (i.e., $logGamma(x)$).
lowergamma(a, x)	The lower incomplete gamma function.
lucas(n)	Lucas numbers
marcumq(m, a, b)	The Marcum Q-function.
mathieuc(a, q, z)	The Mathieu Cosine function $C(a,q,z)$.
mathieucprime(a, q, z)	The derivative $C^{prime}(a,q,z)$ of the Mathieu Cosine function.
mathieus(a, q, z)	The Mathieu Sine function $S(a,q,z)$.
mathieusprime(a, q, z)	The derivative $S^{prime}(a,q,z)$ of the Mathieu Sine function.
meijerg(*args)	The Meijer G-function is defined by a Mellin-Barnes type integral that resembles an inverse Mellin transform.
mobius(n)	Mobius function maps natural number to {-1, 0, 1}
motzkin(n)	The nth Motzkin number is the number
multigamma(x, p)	The multivariate gamma function is a generalization of the gamma function
partition(n)	Partition numbers
periodic_argument(ar, period)	Represent the argument on a quotient of the Riemann surface of the logarithm.
polar_lift(arg)	Lift argument to the Riemann surface of the logarithm, using the standard branch.
polygamma(n, z)	The function polygamma(n, z) returns log(gamma(z)).diff(n + 1).
polylog(s, z)	Polylogarithm function.
preorder_traversal(node[, keys])	Do a pre-order traversal of a tree.
primenu(n)	Calculate the number of distinct prime factors for a positive integer n.
primeomega(n)	Calculate the number of prime factors counting multiplicities for a positive integer n.
primepi(n)	Represents the prime counting function pi(n) = the number of prime numbers less than or equal to n.
principal_branch(x, period)	Represent a polar number reduced to its principal branch on a quotient of the Riemann surface of the logarithm.
re(arg)	Returns real part of expression.
reduced_totient(n)	Calculate the Carmichael reduced totient function lambda(n)
rf	alias of RisingFactorial
riemann_xi(s)	Riemann Xi function.
sec(arg)	The secant function.
sech(arg)	sech(x) is the hyperbolic secant of x.
sign(arg)	Returns the complex sign of an expression:
sin(arg)	The sine function.

continues on next page

Table 68 – continued from previous page

`sinc`(arg)	Represents an unnormalized sinc function:
`sinh`(arg)	`sinh(x)` is the hyperbolic sine of `x`.
`stieltjes`(n[, a])	Represents Stieltjes constants, $gamma_{k}$ that occur in Laurent Series expansion of the Riemann zeta function.
`subfactorial`(arg)	The subfactorial counts the derangements of n items and is defined for non-negative integers as:
`tan`(arg)	The tangent function.
`tanh`(arg)	`tanh(x)` is the hyperbolic tangent of `x`.
`totient`(n)	Calculate the Euler totient function phi(n)
`transpose`(arg)	Linear map transposition.
`tribonacci`(n[, sym])	Tribonacci numbers / Tribonacci polynomials
`trigamma`(z)	The `trigamma` function is the second derivative of the `loggamma` function
`uppergamma`(a, z)	The upper incomplete gamma function.
`vectorize`(*mdargs)	Generalizes a function taking scalars to accept multidimensional arguments.
`yn`(nu, z)	Spherical Bessel function of the second kind.
`zeta`(s[, a])	Hurwitz zeta function (or Riemann zeta function).

Exceptions

`BasePolynomialError`	Base class for polynomial related exceptions.
`CoercionFailed`	
`ComputationFailed`(func, nargs, exc)	
`DomainError`	
`EvaluationFailed`	
`ExactQuotientFailed`(f, g[, dom])	
`ExtraneousFactors`	
`FlagError`	
`GeneratorsError`	
`GeneratorsNeeded`	
`GeometryError`	An exception raised by classes in the geometry module.
`HeuristicGCDFailed`	
`HomomorphismFailed`	
`IsomorphismFailed`	
`MultivariatePolynomialError`	

continues on next page

Table 69 – continued from previous page

`NonSquareMatrixError`	
`NotAlgebraic`	
`NotInvertible`	
`NotReversible`	
`OperationNotSupported`(poly, func)	
`OptionError`	
`PoleError`	
`PolificationFailed`(opt, origs, exprs[, seq])	
`PolynomialDivisionFailed`(f, g, domain)	
`PolynomialError`	
`PrecisionExhausted`	
`RefinementFailed`	
`ShapeError`	Wrong matrix shape
`SympifyError`(expr[, base_exc])	
`UnificationFailed`	
`UnivariatePolynomialError`	

`src.sensitivity.speed_test.`**`run_timed`**(*params*, *parameter_name*)

generates an instance of TestBabyModel with given parameters, sympifies it, and follows the sequence of steps
to get the sensitivity matrix. Prints times for each step.

Parameters

- **`params`** (*`dict`*) – dictionary of parameter values for the TestBabyModel instance generated. These are set in the beginning of the script. Values can be changed in the declaration statements.
- **`parameter_name`** (*`str`*) – the name of the scalar parameter to evaluate sensitivities with respect to.

Returns
tuple of model to be solved, the sympification of the model, the SensitivityMatrix, DifferentialMapping associated with parameter and model solve values

Return type
tuple

`src.sensitivity.speed_test.`**`speed_accuracy_test`**(*params*, *parameter_name*, *p*)

performs a run_timed execution with given parameters, takes the returned values, and measures the effect of increasing and

decreasing the named parameter by p, and compares the result to resolving the model with that same perturbation.

Returns

- **params** (*dict*) – dictionary of parameter values for the TestBabyModel instance generated. These are set in the beginning of the script. Values can be changed in the declaration statements.
- **parameter_name** (*str*) – the name of the scalar parameter to evaluate sensitivities with respect to.
- **p** (*float*) – percentage change up and down to be measured, as a decimal.

PYTHON MODULE INDEX

S

INDEX