PowerGAMA Documentation

Release 1.1.3

Harald G Svendsen

Sep 11, 2020

Contents

CHAPTER 1

GridData Module

Module containing PowerGAMA GridData class and sub-classes

Grid data and time-dependent profiles

class powergama.GridData.**GridData**
Class for grid data storage and import

Methods

branchDistances([R])	computes branch distance from node coordinates, resuls in km
branchFromNodeIdx()	get node indices for branch FROM node
branchToNodeIdx()	get node indices for branch TO node
computePowerFlowMatrices([baseZ])	Compute and return dc power flow matrices B' and DA
dcBranchFromNodeIdx()	get node indices for dc branch FROM node
dcBranchToNodeIdx()	get node indices for dc branch TO node
getAllAreas()	Return list of areas included in the grid model
getAllGeneratorTypes([sort])	Return list of generator types included in the grid model
getConsumerAreas()	List of areas for each consumer
getConsumersPerArea()	Returns dictionary with indices of loads within each area
getDcBranches()	Returns a list with DC branches in the format [index,from area,to area]
getDcBranchesAtNode(nodeIdx, direction)	Indices of all DC branches attached to a particular node
getFlexibleLoadStorageCapacity(consumer_index)	Flexible load storage capacity in MWh
getGeneratorAreas()	List of areas for each generator
getGeneratorsAtNode(nodeIdx)	Indices of all generators attached to a particular node

Continued on next page

Table 1 – continued from previous page

getGeneratorsPerAreaAndType()	Returns dictionary with indices of generators within each area
getGeneratorsPerType()	Returns dictionary with indices of generators per type
getGeneratorsWithPumpAtNode(nodeIdx)	Indices of all pumps attached to a particular node
getGeneratorsWithPumpByArea()	Returns dictionary with indices of generators with pumps within each area
getIdxBranchesWithFlowConstraints()	Indices of branches with less than infinite branch capacity
getIdxBranchesWithLength()	Indices of branches with specified length
getIdxConsumersWithFlexibleLoad()	Indices of all consumers with flexible load
getIdxDcBranchesWithFlowConstraints()	Indices of DC branches with less than infinite branch capacity
getIdxGeneratorsWithNonzeroInflow()	Indices of all generators with nonzero inflow
getIdxGeneratorsWithPumping()	Indices of all generators with pumping capacity
getIdxGeneratorsWithStorage()	Indices of all generators with nonzero and non-infinite storage
getIdxNodesWithLoad()	Indices of nodes that have load (consumer) attached to them
getInterAreaBranches([area_from, area_to, acdc])	Get indices of branches from and/or to specified area(s)
getLoadsAtNode(nodeIdx)	Indices of all loads (consumers) attached to a particular node
getLoadsFlexibleAtNode(nodeIdx)	Indices of all flexible nodes attached to a particular node
readGridData(nodes, ac_branches, ... [, ...])	Read grid data from files into data variables
readProfileData(filename, timerange[, ...])	Read profile (timeseries) into numpy arrays
readSipData(nodes, branches, generators, ...)	Read grid data for investment analysis from files (PowerGIM)
spreadNodeCoordinates([radius, inplace])	Spread nodes with identical coordinates in a small circle with radius r
writeGridDataToFiles(prefix)	Save data to new input files

getBranchAreas	
getDcBranchAreas	
numBranches	
numConsumers	
numDcBranches	
numGenerators	
numNodes	

branchDistances (*R=6373.0*)
computes branch distance from node coordinates, resuls in km

Uses haversine formula

Parameters

R [radius of the Earth]

branchFromNodeIdx ( )
get node indices for branch FROM node

branchToNodeIdx()
get node indices for branch TO node

computePowerFlowMatrices(*baseZ=1*)
Compute and return dc power flow matrices B' and DA

Parameters

baseZ [float (impedance should already be in pu.)] base value for impedance

Returns

(Bprime, DA) [compressed sparse row matrix]

dcBranchFromNodeIdx()
get node indices for dc branch FROM node

dcBranchToNodeIdx()
get node indices for dc branch TO node

getAllAreas()
Return list of areas included in the grid model

getAllGeneratorTypes(*sort='fuelcost'*)
Return list of generator types included in the grid model

getBranchAreas()

getConsumerAreas()
List of areas for each consumer

getConsumersPerArea()
Returns dictionary with indices of loads within each area

getDcBranchAreas()

getDcBranches()
Returns a list with DC branches in the format [index,from area,to area]

getDcBranchesAtNode(*nodeIdx*, *direction*)
Indices of all DC branches attached to a particular node

getFlexibleLoadStorageCapacity(*consumer_indx*)
flexible load storage capacity in MWh

getGeneratorAreas()
List of areas for each generator

getGeneratorsAtNode(*nodeIdx*)
Indices of all generators attached to a particular node

getGeneratorsPerAreaAndType()
Returns dictionary with indices of generators within each area

getGeneratorsPerType()
Returns dictionary with indices of generators per type

getGeneratorsWithPumpAtNode(*nodeIdx*)
Indices of all pumps attached to a particular node

getGeneratorsWithPumpByArea()
Returns dictionary with indices of generators with pumps within each area

getIdxBranchesWithFlowConstraints()
Indices of branches with less than infinite branch capacity

getIdxBranchesWithLength()
Indices of branches with specified length

getIdxConsumersWithFlexibleLoad()
Indices of all consumers with flexible load

getIdxDcBranchesWithFlowConstraints()
Indices of DC branches with less than infinite branch capacity

getIdxGeneratorsWithNonzeroInflow()
Indices of all generators with nonzero inflow

getIdxGeneratorsWithPumping()
Indices of all generators with pumping capacity

getIdxGeneratorsWithStorage()
Indices of all generators with nonzero and non-infinite storage

getIdxNodesWithLoad()
Indices of nodes that have load (consumer) attached to them

getInterAreaBranches(*area_from=None*, *area_to=None*, *acdc='ac'*)
Get indices of branches from and/or to specified area(s)

area_from = area from. Use None (default) to leave unspecifie area_to= area to. Use None (default) to leave unspecified acdc = 'ac' (default) for ac branches, 'dc' for dc branches

getLoadsAtNode(*nodeIdx*)
Indices of all loads (consumers) attached to a particular node

getLoadsFlexibleAtNode(*nodeIdx*)
Indices of all flexible nodes attached to a particular node

keys_powergama = {'branch': {'capacity': None, 'node_from': None, 'node_to': None,

keys_sipdata = {'branch': {'capacity': None, 'capacity2': 0, 'cost_scaling': None,

numBranches()

numConsumers()

numDcBranches()

numGenerators()

numNodes()

readGridData(*nodes*, *ac_branches*, *dc_branches*, *generators*, *consumers*, *remove_extra_columns=False*)
Read grid data from files into data variables

readProfileData(*filename*, *timerange*, *storagevalue_filling=None*, *storagevalue_time=None*, *timedelta=1.0*)
Read profile (timeseries) into numpy arrays

readSipData(*nodes*, *branches*, *generators*, *consumers*)
Read grid data for investment analysis from files (PowerGIM)

This is used with the grid investment module (PowerGIM)

time-series data may be used for consumer demand generator inflow (e.g. solar and wind) generator fuelcost (e.g. one generator with fuelcost = power price)

spreadNodeCoordinates(*radius=0.01*, *inplace=False*)
Spread nodes with identical coordinates in a small circle with radius r

Parameters

radius [float] radius in degrees for the size of the spread

inplace [boolean] if true, update GridData object

Returns

coords [array] lat,lon pandas array for nodes

writeGridDataToFiles(*prefix*)
Save data to new input files

CHAPTER 2

LpProblem Module

Module containing PowerGAMA LpProblem class

class powergama.LpProblemPyomo.**LpProblem**(*grid*, *lossmethod=0*)

Class containing problem definition as a LP problem, and function calls to solve the problem

Methods

setProgressBar(value)	Specify how to show simulation progress
solve(results[, solver, solver_path, ...])	Solve LP problem for each time step in the time range

setProgressBar(*value*)

Specify how to show simulation progress

Parameters

value [string] 'fancy' or 'default'

solve(*results*, *solver='cbc'*, *solver_path=None*, *warmstart=False*, *savefiles=False*, *aclossmultiplier=1*, *dclossmultiplier=1*, *logfile='lpsolver_log.txt'*)

Solve LP problem for each time step in the time range

Parameters

results [Results] PowerGAMA Results object reference

solver [string (optional)] name of solver to use ("cbc" or "gurobi"). Gurobi uses python interface, whilst CBC uses command line executable

solver_path :string (optional, only relevant for cbc) path for solver executable

warmstart [Boolean] Use warmstart option (only some solvers, e.g. gurobi)

savefiles [Boolean] Save Pyomo model file and LP problem MPS file for each timestep This may be useful for debugging.

aclossmultiplier [float] Multiplier factor to scale computed AC losses, used with method 1

dclossmultiplier [float] Multiplier factor to scale computed DC losses, used with method 1

logfile [string] Name of log file for LP solver. Will keep only last iteration

Returns

results [Results] PowerGAMA Results object reference

CHAPTER 3

Results Module

Module containing the PowerGAMA Results class

class powergama.Results.**Results**(*grid*, *databasefile*, *replace=True*, *sip=False*)

Class for storing and analysing/presenting results from PowerGAMA

Parameters

grid [GridData] PowerGAMA GridData object

databasefile [string] name of sqlite3 file for storage of results

replace [boolean] whether to replace existing sqlite file (default=true). replace=false is useful to analyse previously generated results

Methods

addResultsFromTimestep(timestep, ...[, ...])	Store results from optimal power flow for a new timestep
getAllGeneratorProductionOBSOLETE([timeMaxMin])	Return production [MWh] for all generators
getAreaPrices(area[, timeMaxMin])	Weighted average nodal price timeseries for given area
getAreaPricesAverage([areas, timeMaxMin])	Time average of weighted average nodal price per area
getAverageBranchFlows([timeMaxMin, branchtype])	Average flow on branches over a given time period
getAverageBranchSensitivity([timeMaxMin, ...])	Average branch capacity sensitivity over a given time period
getAverageEnergyBalance([timeMaxMin])	Average energy balance (generation minus demand) over a time period
getAverageImportExport(area[, timeMaxMin])	Return average import and export for a specified area

Continued on next page

Table 1 – continued from previous page

getAverageInterareaBranchFlow([filename, ...])	Calculate average flow in each direction and total flow for inter-area branches.
getAverageNodalPrices([timeMaxMin])	Average nodal price over a given time period
getAverageUtilisation([timeMaxMin, branchtype])	Average branch utilisation over a given time period
getDemandPerArea(area[, timeMaxMin])	Returns demand timeseries for given area, as dictionary with fields "fixed", "flex", and "sum"
getEnergyBalanceInArea(area, spillageGen[, ...])	Print time series of energy balance in an area, including production, spillage, load shedding, storage, pump consumption and imports
getEnergyMix([timeMaxMin, relative, ...])	Get energy, generation capacity or spilled energy per area per type
getGeneratorOutputSumPerArea([timeMaxMin])	Description Sums up generation per area.
getGeneratorSpilled(generatorindx[, timeMaxMin])	Get spilled inflow time series for given generator
getGeneratorSpilledSums([timeMaxMin])	Get sum of spilled inflow for all generators
getGeneratorStorageAll(timestep)	Get stored energy for all storage generators at given time
getGeneratorStorageValues(timestep)	Get value of stored energy for given time
getImportExport([areas, timeMaxMin, acdc])	Return time series for import and export for a specified area
getLoadheddingSums([timeMaxMin])	get loadshedding sum per area
getLoadsheddingPerNode([timeMaxMin])	get loadshedding sum per node
getNetImport(area[, timeMaxMin])	Return time series for net import for a specified area
getStorageFillingInAreas(areas, generator_type)	Gets time-series with aggregated storage filling for specified area(s) for a specific generator type.
getSystemCost([timeMaxMin])	Calculates system cost for energy produced by using generator fuel cost.
getSystemCostOBSOLETE([timeMaxMin])	Calculates system cost for energy produced by using generator fuel cost.
plotAreaPrice(areas[, timeMaxMin, showTitle])	Show area price(s)
plotDemandAtLoad(consumer_index[, ...])	Make a time-series plot of consumption of a specified load
plotDemandPerArea(areas[, timeMaxMin, showTitle])	Show demand in area(s)
plotEnergyMix([areas, timeMaxMin, relative, ...])	Plot energy, generation capacity or spilled energy as stacked bars
plotFlexibleLoadStorageValues(consumerindx)	Plot storage valuesfor flexible loads
plotGenerationPerArea(area[, timeMaxMin, ...])	Show generation per area
plotGenerationScatter(area[, tech, dotsize, ...])	Scatter plot of generation capacity and correlation of inflow with load.
plotGeneratorOutput(generator_index[, ...])	Show output of a generator
plotMapGrid([nodetype, branchtype, ...])	Plot results to map
plotNodalPrice(nodeIndx[, timeMaxMin, showTitle])	Show nodal price in single node
plotRelativeGenerationCapacity(tech[, ...])	Plots the relative input generation capacity.
plotRelativeLoadDistribution([...])	Plots the relative input load distribution.

Continued on next page

Table 1 – continued from previous page

plotStorageFilling(generatorIndx[, ...])	Show storage filling level (MWh) for generators with storage
plotStoragePerArea(area[, absolute, ...])	Show generation storage accumulated per area
plotStorageValues(genindx[, timeMaxMin, ...])	Plot storage values (marginal prices) for generators with storage
plotTimeseriesColour(areas[, value, ...])	Plot timeseries values with days on x-axis and hour of day on y-axis
writeProductionOverview(areas, types[, ...])	Export production overview to CSV file

getLoadheddingInArea	
getNodalPrices	

addResultsFromTimestep(*timestep*, *objective_function*, *generator_power*, *generator_pumped*, *branch_power*, *dcbranch_power*, *node_angle*, *sensitivity_branch_capacity*, *sensitivity_dcbranch_capacity*, *sensitivity_node_power*, *storage*, *inflow_spilled*, *loadshed_power*, *marginalprice*, *flexload_power*, *flexload_storage*, *flexload_storagevalue*, *branch_ac_losses=None*, *branch_dc_losses=None*)

Store results from optimal power flow for a new timestep

timestep [int] timestamp of results

objective_function [float] value of objective function

generator_power [list] generator output (list of same length and order as generators)

generator_pumped [list] position according to grid.getIdxGeneratorsWithPumping()

branch_power [list] AC branch power flow (list of same length and order as branches)

dcbranch_power [list] DC branch power flow (list of same length and order as dcbranches)

node_angles [list] voltage angles (list of same length and order as nodes)

sensitivity_branch_capacity [list] dual, position according to grid.getIdxBranchesWithFlowConstraints()

sensitivity_dcbranch_capacity [list] dual, capacity (list of same length and order as dcbranches)

sensitivity_node_power [list] dual, node demand(list of same length and order as nodes)

storage [list] position accordint to grid.getIdxGeneratorsWithStorage()

inflow_spilled [list] spileld power (list of same length and order as generators)

loadshed_power [list] same length and order as nodes

marginalprice [list] position according to grid.getIdxGeneratorsWithStorage()

flexload_power [list] position according to grid.getIdxConsumersWithFlexibleLoad()

flexload_storage [list] position according to grid.getIdxConsumersWithFlexibleLoad()

flexload_storagevalue [list] position according to grid.getIdxConsumersWithFlexibleLoad()

branch_ac_losses [list] ac branch losses

branch_dc_losses [list] dc branch losses

getAllGeneratorProductionOBSOLETE(*timeMaxMin=None*)

Returns all production [MWh] for all generators

getAreaPrices(*area*, *timeMaxMin=None*)
Weighted average nodal price timeseries for given area

getAreaPricesAverage(*areas=None*, *timeMaxMin=None*)
Time average of weighted average nodal price per area

getAverageBranchFlows(*timeMaxMin=None*, *branchtype='ac'*)
Average flow on branches over a given time period

Parameters

timeMaxMin [list (default = None)] [min, max] - lower and upper time interval

branchtype [string] 'ac' (default) or 'dc'

Returns

List with values for each branch:

[flow from 1 to 2, flow from 2 to 1, average absolute flow]

getAverageBranchSensitivity(*timeMaxMin=None*, *branchtype='ac'*)
Average branch capacity sensitivity over a given time period

Parameters

timeMaxMin (list) (default = None) [min, max] - lower and upper time interval

branchtype [str] ac or dc branch type

Returns

1-dim Array of sensitivities (one per branch)

getAverageEnergyBalance(*timeMaxMin=None*)
Average energy balance (generation minus demand) over a time period

timeMaxMin (list) (default = None) [min, max] - lower and upper time interval

Returns

1-dim Array of nodal prices (one per node)

getAverageImportExport(*area*, *timeMaxMin=None*)
Return average import and export for a specified area

getAverageInterareaBranchFlow(*filename=None*, *timeMaxMin=None*)
Calculate average flow in each direction and total flow for inter-area branches. Requires sqlite version newer than 3.6

Parameters

filename [string, optional] if a filename is given then the information is stored to file.

timeMaxMin [list with two integer values, or None, optional] time interval for the calculation [start,end]

Returns

List with values for each inter-area branch:

[flow from 1 to 2, flow from 2 to 1, average absolute flow]

getAverageNodalPrices(*timeMaxMin=None*)
Average nodal price over a given time period

Parameters

timeMaxMin (list) (default = None) [min, max] - lower and upper time interval

Returns

1-dim Array of nodal prices (one per node)

getAverageUtilisation (*timeMaxMin=None*, *branchtype='ac'*)

Average branch utilisation over a given time period

Parameters

timeMaxMin [(list) (default = None)] [min, max] - lower and upper time interval

branchtype [str] ac or dc branch type

Returns

1-dim Array of branch utilisation (power flow/capacity)

getDemandPerArea (*area*, *timeMaxMin=None*)

Returns demand timeseries for given area, as dictionary with fields "fixed", "flex", and "sum"

Parameters

area (string) area to get demand for

timeMaxMin (list) (default = None) [min, max] - lower and upper time interval

getEnergyBalanceInArea (*area*, *spillageGen*, *resolution='H'*, *fileName=None*, *timeMaxMin=None*, *start_date='2014-01-01'*)

Print time series of energy balance in an area, including production, spillage, load shedding, storage, pump consumption and imports

Parameters

area [string] area code

spillageGen [list] generator types for which to show spillage (renewables)

resolution [string] resolution of output, see pandas:resample

fileName [string (default=None)] name of file to export results

timeMaxMin [list] time range to consider

start_date [date string] date when time series start

getEnergyMix (*timeMaxMin=None*, *relative=False*, *showTitle=True*, *variable='energy'*)

Get energy, generation capacity or spilled energy per area per type

Parameters

timeMaxMin [list of two integers] Time range, [min,max]

relative [boolean] Whether to plot absolute (false) or relative (true) values

variable [string ("energy","capacity","spilled")] Which variable to plot (default is energy production)

getGeneratorOutputSumPerArea (*timeMaxMin=None*)

Description Sums up generation per area.

Parameters

timeMaxMin (list) (default = None) [min, max] - lower and upper time interval

Returns

array of dictionary of generation sorted per area

getGeneratorSpilled(*generatorindx*, *timeMaxMin=None*)
Get spilled inflow time series for given generator

Parameters

generatorindx (int) index ofgenerator

timeMaxMin (list) (default = None) [min, max] - lower and upper time interval

getGeneratorSpilledSums(*timeMaxMin=None*)
Get sum of spilled inflow for all generators

Parameters

timeMaxMin (list) (default = None) [min, max] - lower and upper time interval

getGeneratorStorageAll(*timestep*)
Get stored energy for all storage generators at given time

Parameters

timestep [int] timestep when storage is requested

getGeneratorStorageValues(*timestep*)
Get value of stored energy for given time

Parameters

timestep [int] when to compute value

Returns

list of int Value of stored energy for all storage generators

The method uses the storage value absolute level (basecost) per

generator to compute total storage value

getImportExport(*areas=None, timeMaxMin=None, acdc=['ac', 'dc']*)
Return time series for import and export for a specified area

getLoadheddingInArea(*area*, *timeMaxMin=None*)

getLoadheddingSums(*timeMaxMin=None*)
get loadshedding sum per area

getLoadsheddingPerNode(*timeMaxMin=None*)
get loadshedding sum per node

getNetImport(*area*, *timeMaxMin=None*)
Return time series for net import for a specified area

getNodalPrices(*node*, *timeMaxMin=None*)

getStorageFillingInAreas(*areas*, *generator_type*, *relative_storage=True*, *timeMaxMin=None*)
Gets time-series with aggregated storage filling for specified area(s) for a specific generator type.

Parameters

areas [list] list of area codes (e.g. ['DE','FR'])

generator_type [string] generator type string (e.g. 'hydro')

relative_storage [boolean] show relative (True) or absolute (False) storage

timeMaxMin [list] time range to consider (e.g. [0,8760])

getSystemCost(*timeMaxMin=None*)
Calculates system cost for energy produced by using generator fuel cost.

Parameters

timeMaxMin (list) (default = None) [min, max] - lower and upper time interval

Returns

array of dictionary of cost of generation sorted per area

getSystemCostOBSOLETE(*timeMaxMin=None*)
Calculates system cost for energy produced by using generator fuel cost.

Parameters

timeMaxMin (list) (default = None) [min, max] - lower and upper time interval

Returns

array of tuples of total cost of energy per area for all areas

[(area, costs), . . .]

plotAreaPrice(*areas*, *timeMaxMin=None*, *showTitle=True*)
Show area price(s)

Parameters

areas (list) list of areas to show

timeMaxMin (list) (default = None) [min, max] - lower and upper time interval

plotDemandAtLoad(*consumer_index*, *timeMaxMin=None*, *relativestorage=True*, *showTitle=True*)
Make a time-series plot of consumption of a specified load

Parameters

consumer_index (int) index of consumer for which to make the plot

timeMaxMin [int,int] (default=None) time interval for the plot [start,end]

relativestorage (default=True) use filling fraction as y axis label for storage

plotDemandPerArea(*areas*, *timeMaxMin=None*, *showTitle=True*)
Show demand in area(s)

Parameters

areas (list?) list of areas to be plotted

timeMaxMin (list) (default = None) [min, max] - lower and upper time interval

plotEnergyMix(*areas=None*, *timeMaxMin=None*, *relative=False*, *showTitle=True*, *variable='energy'*, *gentypes=None*)
Plot energy, generation capacity or spilled energy as stacked bars

Parameters

areas [list of strings] Which areas to include, default=None means include all

timeMaxMin [list of two integers] Time range, [min,max]

relative [boolean] Whether to plot absolute (false) or relative (true) values

variable [string ("energy","capacity","spilled")] Which variable to plot (default is energy production)

gentypes [list] List of generator types to include. None gives all.

plotFlexibleLoadStorageValues(*consumerindx*, *timeMaxMin=None*, *showTitle=True*)
Plot storage valuesfor flexible loads

Parameters

consumerindx [int] index of consumer for which to make the plot

timeMaxMin [list, [int,int]] time interval for the plot [start,end], or None for entire range

plotGenerationPerArea(*area*, *timeMaxMin=None*, *fill=True*, *reversed_order=False*, *net_import=True*, *loadshed=True*, *showTitle=True*)
Show generation per area

Parameters

area (str)

timeMaxMin (list) (default = None) [min, max] - lower and upper time interval

fill (Boolean) - whether use filled plot

reversed_order - whether to reverse order of generator types

net_import - whether to include net import in graph

loadshed - whether to include unmet demand

plotGenerationScatter(*area*, *tech=[]*, *dotsize=300*, *annotations=True*)
Scatter plot of generation capacity and correlation of inflow with load.

Parameters

area [string] area to plot

tech [list of strings] production technologies to plot. Empty list = all

dotsize [integer] adjust the size of scatterplots

annotations: boolean whether to plot annotations

plotGeneratorOutput(*generator_index*, *timeMaxMin=None*, *relativestorage=True*, *showTitle=True*)
Show output of a generator

Parameters

generator_index (int) index of generator for which to make the plot

timeMaxMin [int,int] (default=None) time interval for the plot [start,end]

relativestorage (default=True) use filling fraction as y axis label for storage

plotMapGrid(*nodetype=None*, *branchtype=None*, *dcbranchtype=None*, *show_node_labels=False*, *branch_style='c'*, *latlon=None*, *timeMaxMin=None*, *dotsize=40*, *filter_node=None*, *filter_branch=None*, *draw_par_mer=False*, *showTitle=True*, *colors=True*)
Plot results to map

Parameters

nodetype [string] "", "area", "nodalprice", "energybalance", "loadshedding"

branchtype [string] "", "capacity", "area", "utilisation", "flow", "sensitivity"

dcbranchtype [string] "", "capacity"

show_node_labels [boolean] whether to show node names (true/false)

branch_style [string or list of strings (optional)] How branch capacity and flow should be visualised. "c" = colour, "t" = thickness. The two options may be combined.

dotsize [integer (optional)] set dot size for each plotted node

latlon: list of four floats (optional) map area [lat_min, lon_min, lat_max, lon_max]

filter_node [list of two floats (optional)] [min,max] - lower and upper cutoff for node value

filter_branch [list of two floats] [min,max] - lower and upper cutoff for branch value

draw_par_mer [boolean] whether to draw parallels and meridians on map

showTitle [boolean]

colors [boolean] Whether to use colours or not

plotNodalPrice(*nodeIndx*, *timeMaxMin=None*, *showTitle=True*)

Show nodal price in single node

Parameters

nodeIndx (int) index of node to plot from

timeMaxMin (list) (default = None) [min, max] - lower and upper time interval

plotRelativeGenerationCapacity(*tech*, *show_node_labels=False*, *latlon=None*, *dotsize=40*, *draw_par_mer=False*, *colours=True*, *showTitle=True*)

Plots the relative input generation capacity.

Parameters

tech [string] production technology to be plotted

show_node_labels [boolean] whether to show node names (true/false)

latlon [list of four floats] map area [lat_min, lon_min, lat_max, lon_max]

draw_par_mer [boolean] whether to draw parallels and meridians on map

colours [boolean] whether to draw the map in colours or black and white

plotRelativeLoadDistribution(*show_node_labels=False*, *latlon=None*, *dotsize=40*, *draw_par_mer=False*, *colours=True*, *showTitle=True*)

Plots the relative input load distribution.

Parameters

show_node_labels [boolean] whether to show node names (true/false)

latlon [list of four floats] map area [lat_min, lon_min, lat_max, lon_max]

draw_par_mer [boolean] whether to draw parallels and meridians on map

colours [boolean] whether to draw the map in colours or black and white

plotStorageFilling(*generatorIndx*, *timeMaxMin=None*, *showTitle=True*)

Show storage filling level (MWh) for generators with storage

Parameters

generatorIndx (int) index of generator to plot from

timeMaxMin (list) (default = None) [min, max] - lower and upper time interval

plotStoragePerArea(*area*, *absolute=False*, *timeMaxMin=None*, *showTitle=True*)

Show generation storage accumulated per area

Parameters

area (str)

absolute (bool)(default=False) plot storage value in absolute or relative to maximum

timeMaxMin (list) (default = None) [min, max] - lower and upper time interval

plotStorageValues(*genindx*, *timeMaxMin=None*, *showTitle=True*)

Plot storage values (marginal prices) for generators with storage

Parameters

genindx (int) index of generator for which to make the plot

timeMaxMin [int,int] (default=None) time interval for the plot [start,end]

plotTimeseriesColour(*areas*, *value='nodalprice'*, *filter_values=None*)

Plot timeseries values with days on x-axis and hour of day on y-axis

Parameters

areas [list of strings] which areas to include, default=None means include all

value ['nodalprice' (default),]

'demand', 'gen%<type1>%<type2>.. (where type=gentype)

which times series value to plot

Example: res.plotTimeseriescolour(['ES'],value='gen%solar_csp%wind')

writeProductionOverview(*areas*, *types*, *filename=None*, *timeMaxMin=None*, *TimeUnitCorrectionFactor=1*)

Export production overview to CSV file

Write a .csv overview of the production[MWh] in timespan 'timeMaxMin' with the different areas and types as headers. The vectors 'areas' and 'types' becomes headers (column- and row headers), but the different elements of 'types' and 'areas' are also the key words in the search function 'getAreaTypeProduction'. The vectors 'areas' and 'types' can be of any length.

CHAPTER 4

Plotting Module

PowerGAMA module containing plotting functions

powergama.plots.**plotMap**(*pg_data, pg_res=None, filename=None, nodetype=None, branchtype=None, filter_node=[0, 100], filter_branch=None, timeMaxMin=None, spread_nodes_r=None, **kwargs*)

Plot PowerGAMA data/results on map

Parameters

pg_data [powergama.GridData] powergama data object

pg_res [powergama.Results] powergama results object

filename [str] name of output file (html)

nodetype [str ('nodalprice','area') or None (default)] how to colour nodes

branchtype [str ('utilisation','sensitivity') or None (default)] how to colour branches

filter_node [list] max/min value used for colouring nodes (e.g. nodalprice)

filter_branch [list] max/min value used for colouring branches (e.g. utilisation)

timeMaxMin [[min,max]] time interval used when showing simulation results

spread_nodes_r [float (degrees)] radius (degrees) of circle on which overlapping nodes are spread (use eg 0.04)

kwargs [arguments passed on to folium.Map(...)]

CHAPTER 5

Scenarios Module

Module for creating different PowerGAMA scenarios by scaling grid model parameters according to specified input.

powergama.scenarios.**newScenario**(*base_grid_data*, *scenario_file*, *newfile_prefix=None*)

Create new dataset by modifying grid model according to scenario file

This method replaces generator and consumer data according to information given in scenario file. Information that should not be replaced should be omitted from or have an empty value in the scenario file; the default is to keep existing value.

Parameters

base_grid_data [GridData]

PowerGAMA grid model object used as basis for modifications

scenario_file [string] Name of scenario file (CSV)

newfiles_prefix [string] Prefix used when creating new files. New files will be the same as old files with this additional prefix

powergama.scenarios.**saveScenario**(*base_grid_data*, *scenario_file*, *verbose=True*)

Saves the data in the current grid model to a scenario file of the format used to create new scenarios

Parameters

base_grid_data [GridData]

PowerGAMA GridData object

scenario_file [string] name of new scenario (CSV) file

CHAPTER 6

Database Module

Module dealing with database IO

class powergama.database.**Database**(*filename*)
Class for storing results from PowerGAMA in sqlite databse

Methods

appendResults(timestep, objective_function, ...)	Store results from a given timestep to the database
createTables(data)	Create database for PowerGAMA results
getAverageInterareaBranchFlow(timeMaxMin)	Get average negative flow, positive flow and total flow of branches between different areas
getBranchesSumFlow(branches_pos, ...)	Return time series for aggregated flow along specified branches
getGridBranches()	Get branch indices as a list
getGridGeneratorFromArea(area)	Get indices of generators in given area as a list
getGridInterareaBranches()	Get indices of branches between different areas as a list
getGridNodeIndices()	Get node indices as a list
getResultAreaPrices(node_weight, timeMaxMin)	Get area price timeseries
getResultBranchFlow(branchindx, timeMaxMin)	Get branch flow at specified branch
getResultBranchFlowAll(timeMaxMin[, acdc])	Get branch flow at all branches (list of tuples)
getResultBranchFlowsMean(timeMaxMin[, ac])	Get average branch flow on branches in both direction
getResultBranchLossesSum(timeMaxMin[, acdc])	Sum of losses for each time-step time step

Continued on next page

Table 1 – continued from previous page

getResultBranchSens(branchindx, timeMaxMin)	Get branch capacity sensitivity at specified branch
getResultBranchSensAll(timeMaxMin)	Get branch capacity sensitivity at all branches
getResultBranchSensMean(timeMaxMin[, acdc])	Get average sensitivity of all branches acdc = 'ac' or 'dc'
getResultBranches(timeMaxMin[, br_indx, acdc])	Branch results for each time-step
getResultFlexloadPower(consumerindx, timeMaxMin)	Get flexible load for consumer with flexible load
getResultFlexloadStorageFilling(...)	Get storage filling level for flexible loads
getResultFlexloadStorageValue(consumerindx, ...)	Get storage value for flexible loads
getResultGeneratorPower(generatorindx, ...)	Get power output time series for specified generator
getResultGeneratorPowerInArea(area, timeMaxMin)	Get accumulated generation per type in given area
getResultGeneratorPowerSum(timeMaxMin)	Sum of generator power output per generator
getResultGeneratorSpilled(generatorindx, ...)	Get spilled power time series for specified generator
getResultGeneratorSpilledSums(timeMaxMin)	Get sum of spilled power for all generator
getResultLoadheddingInArea(area, timeMaxMin)	Aggregated loadshedding timeseries for specified area
getResultLoadheddingSum(timeMaxMin)	Sum of loadshedding timeseries per node
getResultNodalPrice(nodeindx, timeMaxMin)	Get nodal price at specified node
getResultNodalPricesAll(timeMaxMin)	Get nodal price at all nodes (list of tuples)
getResultNodalPricesMean(timeMaxMin)	Get average nodal price at all nodes
getResultPumpPower(genindx, timeMaxMin)	Get pumping for generators with pumping
getResultPumpPowerMultiple(genindx, timeMaxMin)	Get pumping for generators with pumping
getResultPumpingSum(timeMaxMin[, variable])	Sum of pumping per generator
getResultStorageFilling(genindx, timeMaxMin)	Get storage filling level for storage generators
getResultStorageFillingAll(timestep)	Get storage filling level for all storage generators
getResultStorageFillingMultiple(genindx, ...)	Get storage filling level for multiple storage generators
getResultStorageValue(storageindx, timeMaxMin)	Get storage value for storage generators
getResultStorageValueMultiple(storageindx, ...)	Get average storage value (marginal price) for multiple storage generators
getTimerange()	Get the timesteps

SQLITE_MAX_VARIABLE_NUMBER = 990

appendResults(*timestep*, *objective_function*, *generator_power*, *generator_pumped*, *branch_flow*, *dcbranch_flow*, *node_angle*, *sensitivity_branch_capacity*, *sensitivity_dcbranch_capacity*, *sensitivity_node_power*, *storage*, *inflow_spilled*, *loadshed_power*, *marginalprice*, *flexload_power*, *flexload_storage*, *flexload_storagevalue*, *idx_storagegen*, *idx_branchsens*, *idx_pumpgen*, *idx_flexload*, *branch_ac_losses*, *branch_dc_losses*)

Store results from a given timestep to the database

Parameters

timestep (int) timestep number

objective_function (float) value of objective function

generator_power (list of floats) power output of generators

generator_pumped (list of floats) pumped power for generators

branch_power (list of floats) power flow on branches

node_angle (list of floats) phase angle (relative to node 0) at nodes

sensitivity_branch_capacity (list of floats) sensitivity to branch capacity

sensitivity_dcbranch_capacity (list of floats) sensitivty to DC branch capacity

sensitivity_node_power (list of floats) sensitivity to node power (nodal price)

storage storage filling level of generators

inflow_spilled (list of floats) spilled power inflow of generators

loadshed_power (list of floats) unmet power demand at nodes

marginalprice price of generators with storage

flexload_power (list of floats) flexible load power consumption

flexload_storage storage filling level of flexible load

flexload_storagevalue storage value in flexible load energy storage

idx_storagegen index in generator list of generators with storage

idx_branchsens index in branch list of branches with limited capacity

idx_pumpgen index in generator list of generators with pumping

idx_flexload index in consumer list of flexible loads

branch_ac_losses [list] ac branch losses

branch_dc_losses [list] dc branch losses

createTables(*data*)
Create database for PowerGAMA results

getAverageInterareaBranchFlow(*timeMaxMin*)
Get average negative flow, positive flow and total flow of branches between different areas

Returns

List of tuples for inter-area branches with following values:

(indices, fromArea, toArea, average negative flow, average positive

flow, average flow)

getBranchesSumFlow(*branches_pos*, *branches_neg*, *timeMaxMin*, *acdc*)
Return time series for aggregated flow along specified branches

branches_pos = indices of branches with positive flow direction branches_neg = indices of branches with negative flow direction timeMaxMin = [start, end] acdc = 'ac' or 'dc'

Note: This function can be used to get net import, but not separate import/export values (requires summing positive and negative values separately)

getGridBranches()
Get branch indices as a list

getGridGeneratorFromArea(*area*)
Get indices of generators in given area as a list

Returns

(indice)

getGridInterareaBranches()
Get indices of branches between different areas as a list

Returns

(indice, fromArea, toArea)

getGridNodeIndices()
Get node indices as a list

getResultAreaPrices(*node_weight*, *timeMaxMin*)
Get area price timeseries

node_weight = list of weights for each node

getResultBranchFlow(*branchindx*, *timeMaxMin*, *ac=True*)
Get branch flow at specified branch

getResultBranchFlowAll(*timeMaxMin*, *acdc='ac'*)
Get branch flow at all branches (list of tuples)

Returns

List of tuples with values:

(timestep, branch index, flow)

getResultBranchFlowsMean(*timeMaxMin*, *ac=True*)
Get average branch flow on branches in both direction

Parameters

timeMaxMin (list of two elements) - time interval

ac (bool) - ac (true) or dc (false) branches

Returns

List with values for each branch:

[average flow 1->2, average flow 2->1, average absolute flow]

getResultBranchLossesSum(*timeMaxMin*, *acdc='ac'*)
Sum of losses for each time-step time step

getResultBranchSens(*branchindx*, *timeMaxMin*, *acdc='ac'*)
Get branch capacity sensitivity at specified branch

getResultBranchSensAll(*timeMaxMin*)
Get branch capacity sensitivity at all branches

getResultBranchSensMean(*timeMaxMin*, *acdc='ac'*)
Get average sensitivity of all branches acdc = 'ac' or 'dc'

getResultBranches(*timeMaxMin*, *br_indx=None*, *acdc='ac'*)
Branch results for each time-step

Parameters

timeMaxMin [[start,end]] tuple with time window start <= t < end

br_indx [list (optional)] list of branches to consider, None=include all

getResultFlexloadPower(*consumerindx*, *timeMaxMin*)
Get flexible load for consumer with flexible load

getResultFlexloadStorageFilling(*consumerindx*, *timeMaxMin*)
Get storage filling level for flexible loads

getResultFlexloadStorageValue(*consumerindx*, *timeMaxMin*)
Get storage value for flexible loads

getResultGeneratorPower(*generatorindx*, *timeMaxMin*)
Get power output time series for specified generator

getResultGeneratorPowerInArea(*area*, *timeMaxMin*)
Get accumulated generation per type in given area

getResultGeneratorPowerSum(*timeMaxMin*)
Sum of generator power output per generator

getResultGeneratorSpilled(*generatorindx*, *timeMaxMin*)
Get spilled power time series for specified generator

getResultGeneratorSpilledSums(*timeMaxMin*)
Get sum of spilled power for all generator

getResultLoadheddingInArea(*area*, *timeMaxMin*)
Aggregated loadshedding timeseries for specified area

getResultLoadheddingSum(*timeMaxMin*)
Sum of loadshedding timeseries per node

getResultNodalPrice(*nodeindx*, *timeMaxMin*)
Get nodal price at specified node

getResultNodalPricesAll(*timeMaxMin*)
Get nodal price at all nodes (list of tuples)

Returns

List of tuples with values:

(timestep, node index, nodal price)

getResultNodalPricesMean(*timeMaxMin*)
Get average nodal price at all nodes

getResultPumpPower(*genindx*, *timeMaxMin*)
Get pumping for generators with pumping

getResultPumpPowerMultiple(*genindx*, *timeMaxMin*, *negative=True*)
Get pumping for generators with pumping

getResultPumpingSum(*timeMaxMin*, *variable='output'*)
Sum of pumping per generator

getResultStorageFilling(*genindx*, *timeMaxMin*)
Get storage filling level for storage generators

getResultStorageFillingAll(*timestep*)
Get storage filling level for all storage generators

getResultStorageFillingMultiple(*genindx*, *timeMaxMin*, *capacity=None*)
Get storage filling level for multiple storage generators

getResultStorageValue(*storageindx*, *timeMaxMin*)
Get storage value for storage generators

getResultStorageValueMultiple(*storageindx*, *timeMaxMin*)
Get average storage value (marginal price) for multiple storage generators

getTimerange()
Get the timesteps

CHAPTER 7

Constants Module

Module for PowerGAMA constants

powergama.constants.**MWh_per_GWh = 1000.0**
: Conversion factor from GWh to MWh

powergama.constants.**baseAngle = 1**
: Base value for voltage angle

powergama.constants.**baseS = 100000000.0**
: Per unit base value for power in W (100 MW)

powergama.constants.**baseV = 400000.0**
: Per unit base value for voltage in V (400 kV)

powergama.constants.**flexload_outside_cost = 1000.0**
: (Very high) storage value for flexible demand outside flexibility range

powergama.constants.**hoursperyear = 8760.0**
: Hours per year (365*24 = 8760)

powergama.constants.**loadshedcost = 1000.0**
: Penalty (/MWh) for load shedding

CHAPTER 8

PowerGIM

8.1 PowerGIM

Module for power grid investment analyses

class powergama.powergim.**SipModel**(*M_const=1000*)
Power Grid Investment Module - Stochastic Investment Problem

Methods

computeAreaCostBranch(model, c, stage[, ...])	Investment cost for branches connected to an given area
computeAreaCostGen(model, c)	compute capital costs for new generator capacity
computeAreaEmissions(model, c[, stage, cost])	compute total emissions from a load/country
computeAreaPrice(model, area, t[, stage])	cumpute the approximate area price based on max marginal cost
computeAreaRES(model, j, shareof[, stage])	compute renewable share of demand or total generation capacity
computeAreaWelfare(model, c, t[, stage])	compute social welfare for a given area and time step
computeBranchCongestionRent(model, b[, stage])	Compute annual congestion rent for a given branch
computeCostBranch(model, b[, stage, include_om])	Investment cost of single branch NPV
computeCostGenerator(model, g[, stage, ...])	Investment cost of generator NPV
computeCostNode(model, n[, include_om])	Investment cost of single node
computeCurtailment(model, g, t[, stage])	compute curtailment [MWh] per generator per hour
computeDemand(model, c, t)	compute demand at specified load ant time
computeGenerationCost(model, g, stage)	compute NPV cost of generation (+ CO2 emissions)

Continued on next page

Table 1 – continued from previous page

costBranch(model, b, stage)	Expression for cost of branch, investment cost no discounting
costGen(model, g, stage)	Expression for cost of generator, investment cost no discounting
costInvestments(model, stage[, includeOM, ...])	Investment cost, including lifetime O&M costs (NPV)
costNode(model, n, stage)	Expression for cost of node, investment cost no discounting
costOperation(model, stage)	Operational costs: cost of gen, load shed (NPV)
costOperationSingleGen(model, g, stage)	Operational costs: cost of gen, load shed (NPV)
createConcreteModel(dict_data)	Create Concrete Pyomo model for PowerGIM
createModelData(grid_data, datafile, ...)	Create model data in dictionary format
createScenarioTreeModel(num_scenarios[, ...])	Generate model instance with data.
extractResultingGridData(grid_data[, model, ...])	Extract resulting optimal grid layout from simulation results
loadResults(filename, sheet)	load results from excel into pandas dataframe
npvInvestment(model, stage, investment[, ...])	NPV of investment cost including lifetime O&M and salvage value
plotAreaPrice(model[, boxplot, areas, ...])	Show area price(s) TODO: incoporate samplefactor
plotBranchData(model[, stage])	Plot branch data
plotEnergyMix(model[, areas, timeMaxMin, ...])	Plot energy, generation capacity or spilled energy as stacked bars
plotInvestments(filename, variable[, unit])	Plot investment bar plots
plotWelfare(model[, areas, timeMaxMin, ...])	Plot welfare
saveDeterministicResults(model, excel_file)	export results to excel file
writeStochasticProblem(path, dict_data)	create input files for solving stochastic problem

computeAreaCostBranch (*model*, *c*, *stage*, *include_om=False*)
Investment cost for branches connected to an given area

computeAreaCostGen (*model*, *c*)
compute capital costs for new generator capacity

computeAreaEmissions (*model*, *c*, *stage=2*, *cost=False*)
compute total emissions from a load/country

computeAreaPrice (*model*, *area*, *t*, *stage=2*)
cumpute the approximate area price based on max marginal cost

computeAreaRES (*model*, *j*, *shareof*, *stage=2*)
compute renewable share of demand or total generation capacity

computeAreaWelfare (*model*, *c*, *t*, *stage=2*)
compute social welfare for a given area and time step

Returns: Welfare, ProducerSurplus, ConsumerSurplus, CongestionRent, IMport, eXport

computeBranchCongestionRent (*model*, *b*, *stage=1*)
Compute annual congestion rent for a given branch

computeCostBranch (*model*, *b*, *stage=2*, *include_om=False*)
Investment cost of single branch NPV

corresponds to firstStageCost in abstract model

computeCostGenerator(*model*, *g*, *stage=2*, *include_om=False*)
Investment cost of generator NPV

computeCostNode(*model*, *n*, *include_om=False*)
Investment cost of single node

corresponds to cost in abstract model

computeCurtailment(*model*, *g*, *t*, *stage=2*)
compute curtailment [MWh] per generator per hour

computeDemand(*model*, *c*, *t*)
compute demand at specified load ant time

computeGenerationCost(*model*, *g*, *stage*)
compute NPV cost of generation (+ CO2 emissions)

This corresponds to secondStageCost in abstract model

costBranch(*model*, *b*, *stage*)
Expression for cost of branch, investment cost no discounting

costGen(*model*, *g*, *stage*)
Expression for cost of generator, investment cost no discounting

costInvestments(*model*, *stage*, *includeOM=True*, *subtractSalvage=True*)
Investment cost, including lifetime O&M costs (NPV)

costNode(*model*, *n*, *stage*)
Expression for cost of node, investment cost no discounting

costOperation(*model*, *stage*)
Operational costs: cost of gen, load shed (NPV)

costOperationSingleGen(*model*, *g*, *stage*)
Operational costs: cost of gen, load shed (NPV)

createConcreteModel(*dict_data*)
Create Concrete Pyomo model for PowerGIM

Parameters

dict_data [dictionary] dictionary containing the model data. This can be created with the createModelData(...) method

Returns

Concrete pyomo model

createModelData(*grid_data*, *datafile*, *maxNewBranchNum*, *maxNewBranchCap*)
Create model data in dictionary format

Parameters

grid_data [powergama.GridData object] contains grid model

datafile [string] name of XML file containing additional parameters

maxNewBranchNum [int] upper limit on parallel branches to consider (e.g. 10)

maxNewBranchCap [float (MW)] upper limit on new capacity to consider (e.g. 10000)

Returns

dictionary with pyomo data (in pyomo format)

createScenarioTreeModel(*num_scenarios, probabilities=None, stages=[1, 2]*)
Generate model instance with data. Alternative to .dat files

Parameters

num_scenarios [int] number of scenarios. Each with the same probability

probabilities [list of float] probabilities of each scenario (must sum to 1). Number of elements determine number of scenarios

stages [list of stage names] NOTE: Presently only works with default value=[1,2]

Returns

PySP 2-stage scenario tree model

This method may be called by "pysp_scenario_tree_model_callback()" in

the model input file instead of using input .dat files

extractResultingGridData(*grid_data*, *model=None*, *file_ph=None*, *stage=1*, *scenario=None*, *newData=False*)
Extract resulting optimal grid layout from simulation results

Parameters

grid_data [powergama.GridData] grid data class

model [Pyomo model] concrete instance of optimisation model containing det. results

file_ph [string] CSV file containing results from stochastic solution

stage [int] Which stage to extract data for (1 or 2). 1: only stage one investments included (default) 2: both stage one and stage two investments included

scenario [int] which stage 2 scenario to get data for (only relevant when stage=2)

newData [Boolean] Choose whether to use only new data (True) or add new data to existing data (False)

Use either model or file_ph parameter

Returns

GridData object reflecting optimal solution

loadResults(*filename*, *sheet*)
load results from excel into pandas dataframe

npvInvestment(*model*, *stage*, *investment*, *includeOM=True*, *subtractSalvage=True*)
NPV of investment cost including lifetime O&M and salvage value

Parameters

model [object] Pyomo model

stage [int] Investment or operation stage (1 or 2)

investment : cost of e.g. node, branch or gen

plotAreaPrice(*model*, *boxplot=False*, *areas=None*, *timeMaxMin=None*, *showTitle=False*, *stage=1*)
Show area price(s) TODO: incoporate samplefactor

Parameters

areas (list) list of areas to show

timeMaxMin (list) (default = None) [min, max] - lower and upper time interval

plotBranchData(*model*, *stage=2*)
Plot branch data

plotEnergyMix(*model*, *areas=None*, *timeMaxMin=None*, *relative=False*, *showTitle=True*, *variable='energy'*, *gentypes=None*, *stage=1*)
Plot energy, generation capacity or spilled energy as stacked bars

Parameters

areas [list of sting] Which areas to include, default=None means include all

timeMaxMin [list of two integers] Time range, [min,max]

relative [boolean] Whether to plot absolute (false) or relative (true) values

variable [string ("energy","capacity","spilled")] Which variable to plot (default is energy production)

gentypes [list] List of generator types to include. None gives all.

plotInvestments(*filename*, *variable*, *unit='capacity'*)
Plot investment bar plots

filename: string excel-file generated by 'saveDeterministicResults'

variable: string dcbranch, acbranch, node, generator

unit: string capacity, monetary

plotWelfare(*model*, *areas=None*, *timeMaxMin=None*, *relative=False*, *showTitle=False*, *variable='energy'*, *gentypes=None*, *stage=2*)
Plot welfare

Parameters

areas [list of sting] Which areas to include, default=None means include all

timeMaxMin [list of two integers] Time range, [min,max]

relative [boolean] Whether to plot absolute (false) or relative (true) values

variable [string ("energy","capacity","spilled")] Which variable to plot (default is energy production)

gentypes [list] List of generator types to include. None gives all.

saveDeterministicResults(*model*, *excel_file*)
export results to excel file

Parameters

model [Pyomo model] concrete instance of optimisation model

excel_file [string] name of Excel file to create

writeStochasticProblem(*path*, *dict_data*)
create input files for solving stochastic problem

Parameters

path [string] Where to put generated files

dict_data [dictionary] Pyomo data model in dictionary format. Output from createModelData method

Returns

string that can be written to .dat file (reference model data)

powergama.powergim.**annuityfactor**(*rate*, *years*)
Net present value factor for fixed payments per year at fixed rate

powergama.powergim.**computeSTOcosts**(*grid_data*, *dict_data*, *generation=None*, *include_om=True*)
Compute costs as in objective function of the optimisation This function is used to analyse optimisation results.

Parameters

grid_data [powergama.grid_data] grid object

dict_data [dict] dictionary holding the optimisation input data (as dictionary)

generation [list of dataframes, one per stage] generator operational costs, dataframe with columns ['gen','time','value']

8.2 Sampling Module

powergama.sampling.**sampleProfileData**(*data*, *samplesize*, *sampling_method*)
Sample data from full-year time series

Parameters

data [GridData object] Sample from data.profiles

samplesize [int] size of sample

sampling_method [str] 'kmeans', 'uniform', EXPERIMENTAL: 'kmeans_scale', 'lhs', ('mmatching', 'meanshift')

Returns

reduced data matrix according to sample size and method

8.3 Costbenefit Module

class powergama.costbenefit.**CostBenefit**
Experimental class for cost-benefit calculations.

Currently including allocation schemes based on "cooperative game theory"

Methods

gameIsMonotone(values)	Returns true if the game/valueFunction is monotonic.
gameIsSuperadditive(values)	Returns true if the game/valueFunction is superadditive.
gamePayoffHasNullplayer(player_list, values, ...)	Return true if the payoff vector possesses the nullplayer property.
gamePayoffIsEfficient(player_list, values, ...)	Return 'true if the payoff vector is efficient.A payoff vector v is efficient if the sum of payments equal the total value provided by a set of players.sum_{i=1}^N lambda_i = v(Omega);.
gamePayoffIsSymmetric(values, payoff_vector)	Returns true if the resulting payoff vector possesses the symmetry property.

Continued on next page

Table 2 – continued from previous page

gameShapleyValue(player_list, values)	compute the Shapley Value from cooperative game theory
getBinaryCombinations(num)	Returns a sequence of different combinations 1/0 for a number of decision variables.
nCr(n, r)	calculate the binomial coefficient, i.e.
power_set(List)	function to return the powerset of a list, i.e.

gameSetup

gameIsMonotone (*values*)
Returns true if the game/valueFunction is monotonic. A game G = (N, v) is monotonic if it satisfies the value function of a subset is less or equal then the value function from its union set: v(C_2) geq v(C_1) for all C_1 subseteq C_2

gameIsSuperadditive (*values*)
Returns true if the game/valueFunction is superadditive. A characteristic function game G = (N, v) is superadditive if it the sum of two coalitions/subsets gives a larger value than the individual sum: v(C_1 cup C_2) geq v(C_1) + v(C_2) for all C_1, C_2 subseteq 2^{Omega} such that C_1 cap C_2 = emptyset.

gamePayoffHasNullplayer (*player_list*, *values*, *payoff_vector*)
Return true if the payoff vector possesses the nullplayer property. A payoff vector v has the nullplayer property if there exists an i such that v(C cup i) = v(C) for all C in 2^{Omega} then, lambda_i = 0. In other words: if a player does not contribute to any coalition then that player should receive no payoff.

gamePayoffIsEfficient (*player_list*, *values*, *payoff_vector*)
Return ‘true if the payoff vector is efficient. A payoff vector v is efficient if the sum of payments equal the total value provided by a set of players. sum_{i=1}^N lambda_i = v(Omega);

gamePayoffIsSymmetric (*values*, *payoff_vector*)
Returns true if the resulting payoff vector possesses the symmetry property. A payoff vector possesses the symmetry property if players with equal marginal contribution receives the same payoff: v(C cup i) = v(C cup j) for all C in 2^{Omega} setminus {i,j}, then x_i = x_j.

gameSetup (*grid_data*)

gameShapleyValue (*player_list*, *values*)
compute the Shapley Value from cooperative game theory

getBinaryCombinations (*num*)
Returns a sequence of different combinations 1/0 for a number of decision variables. E.g. three cable investments; (0,0,0), (1,0,0), (0,1,0), and so on.

nCr (*n*, *r*)
calculate the binomial coefficient, i.e. how many different possible subsets can be made from the larger set n

power_set (*List*)
function to return the powerset of a list, i.e. all possible subsets ranging from length of one, to the length of the larger list

CHAPTER 9

Indices

- modindex
- genindex
- search

Python Module Index

p

Index

E

F

G

H

K

L

M

N

P

R

S

W