Package: epluspar (via r-universe)

October 26, 2024

Title Conduct Parametric Analysis on 'EnergyPlus' Models

Version 0.0.0.9000

Description A toolkit for conducting parametric analysis on 'EnergyPlus'(<https://energyplus.net>) models in R, including sensitivity analysis using Morris method and Bayesian calibration using using 'Stan'(<https://mc-stan.org>). References: Chong (2018) <doi:10.1016/j.enbuild.2018.06.028>.

URL https://github.com/ideas-lab-nus/epluspar

BugReports https://github.com/ideas-lab-nus/epluspar

License MIT + file LICENSE

Depends R (>= 3.4.0)

Imports cli, checkmate, data.table, ecr (>= 2.1.0), eplusr (>= 0.12.0), future, future.apply, globals, lhs, lubridate, methods, Rcpp (>= 0.12.0), rstan (>= 2.18.1), rstantools (>= 1.4.0), sensitivity

LinkingTo BH (>= 1.66.0), Rcpp (>= 0.12.0), RcppEigen (>= 0.3.3.3.0), rstan (>= 2.18.1), StanHeaders (>= 2.18.0)

Suggests testthat, covr, pkgdown, pkgload, knitr, rmarkdown

SystemRequirements EnergyPlus (>= 8.3, optional) (<https://energyplus.net>); GNU make

Encoding UTF-8

LazyData true

Biarch true

Roxygen list(markdown = TRUE, roclets = c(``rd", ``namespace", ``collate"))

RoxygenNote 7.3.1

Remotes hongyuanjia/eplusr

Collate 'bayesian.R' 'epluspar.R' 'utils.R' 'gaoptim.R' 'sensitivity.R' 'stanmodels.R'

Repository https://hongyuanjia.r-universe.dev

RemoteUrl https://github.com/hongyuanjia/epluspar

RemoteRef HEAD

RemoteSha 8057945361d00b1b087808368819fddb0bb691f2

Contents

`epluspar-package` *The 'epluspar' package.*

Description

Conduct sensitivity analysis and Bayesian calibration of EnergyPlus models.

References

A. Chong and K. Menberg, "Guidelines for the Bayesian calibration of building energy models", Energy and Buildings, vol. 174, pp. 527–547. DOI: 10.1016/j.enbuild.2018.06.028

`BayesCalibJob` *Conduct Bayesian Calibration on An EnergyPlus Model*

Description

`BayesCalibJob` class provides a prototype of conducting Bayesian calibration of EnergyPlus model.

Details

The basic workflow is basically:

1. Setting input and output variables using `$input()` and `$output()`, respectively. Input variables should be variables listed in RDD while output variables should be variables listed in RDD and MDD.
2. Adding parameters to calibrate using `$param()` or `$apply_measure()`.
3. Check parameter sampled values and generated parametric models using `$samples()` and `$models()`, respectively.
4. Run EnergyPlus simulations in parallel using `$eplus_run()`,
5. Gather simulated data of input and output parameters using `$data_sim()`.
6. Specify field measured data of input and output parameters using `$data_field()`.
7. Specify input data for Stan for Bayesian calibration using `$data_bc()`.
8. Run bayesian calibration using stan using `$stan_run()`.

Super classes

`eplusr::EplusGroupJob` -> `eplusr::ParametricJob` -> `BayesCalibJob`

Methods

Public methods:

- `BayesCalibJob$new()`
- `BayesCalibJob$read_rdd()`
- `BayesCalibJob$read_mdd()`
- `BayesCalibJob$input()`
- `BayesCalibJob$output()`
- `BayesCalibJob$param()`
- `BayesCalibJob$apply_measure()`
- `BayesCalibJob$samples()`
- `BayesCalibJob$models()`
- `BayesCalibJob$data_sim()`
- `BayesCalibJob$data_field()`
- `BayesCalibJob$data_bc()`
- `BayesCalibJob$eplus_run()`

- BayesCalibJob$eplus_kill()
- BayesCalibJob$eplus_status()
- BayesCalibJob$eplus_output_dir()
- BayesCalibJob$eplus_locate_output()
- BayesCalibJob$eplus_errors()
- BayesCalibJob$eplus_report_data_dict()
- BayesCalibJob$eplus_report_data()
- BayesCalibJob$eplus_tabular_data()
- BayesCalibJob$eplus_save()
- BayesCalibJob$stan_run()
- BayesCalibJob$stan_file()
- BayesCalibJob$post_dist()
- BayesCalibJob$prediction()
- BayesCalibJob$evaluate()

Method `new()`**:** Create a `BayesCalibJob` object

Usage:

```
BayesCalibJob$new(idf, epw)
```

Arguments:

`idf` A path to an local EnergyPlus IDF file or an eplusr::Idf object.

`epw` A path to an local EnergyPlus EPW file or an eplusr::Epw object.

Details: When initialization, the objects of classes related in output variable reporting in the original eplusr::Idf will be deleted, in order to make sure all input and output variable specifications can be achieved using `Output:Variable` and `Output:Meter`. Classes to be deleted include:

- `Output:Variable`
- `Output:Meter`
- `Output:Meter:MeterFileOnly`
- `Output:Meter:Cumulative`
- `Output:Meter:Cumulative:MeterFileOnly`
- `Meter:Custom`
- `Meter:CustomDecrement`
- `Output:EnvironmentalImpactFactors`

Returns: An `BayesCalibJob` object.

Examples:

```
\dontrun{
if (eplusr::is_avail_eplus(8.8)) {
    idf_name <- "1ZoneUncontrolled.idf"
    epw_name <-  "USA_CA_San.Francisco.Intl.AP.724940_TMY3.epw"

  idf_path <- file.path(eplusr::eplus_config(8.8)$dir, "ExampleFiles", idf_name)
  epw_path <- file.path(eplusr::eplus_config(8.8)$dir, "WeatherData", epw_name)
```

```
    # create from local files
    BayesCalibJob$new(idf_path, epw_path)

    # create from an Idf and an Epw object
   bc <- BayesCalibJob$new(eplusr::read_idf(idf_path), eplusr::read_epw(epw_path))
}
}
```

Method `read_rdd()`**:** Read EnergyPlus Report Data Dictionary (RDD) file

Usage:

```
BayesCalibJob$read_rdd(update = FALSE)
```

Arguments:

`update` Whether to run the design-day-only simulation and parse `.rdd` and `.mdd` file again. Default: `FALSE`.

Details: `$read_rdd()` silently runs EnergyPlus using input seed model with design-day-only mode to create the `.rdd` file and returns the corresponding RddFile object.
The `RddFile` object is stored internally and will be directly returned whenever you call `$read_rdd()` again. You can force to rerun the design-day-only simulation again to update the contents by setting `update` to `TRUE`.
`$read_rdd()` and `$read_mdd()` are useful when adding input and output parameters using `$input()` and `$output()`, respectively.

Returns: An RddFile object.

Examples:

```
\dontrun{
bc$read_rdd()

# force to rerun
bc$read_rdd(update = TRUE)
}
```

Method `read_mdd()`**:** Read EnergyPlus Meter Data Dictionary (MDD) file

Usage:

```
BayesCalibJob$read_mdd(update = FALSE)
```

Arguments:

`update` Whether to run the design-day-only simulation and parse `.rdd` and `.mdd` file again. Default: `FALSE`.

Details: `$read_mdd()` silently runs EnergyPlus using input seed model with design-day-only mode to create the `.mdd` file and returns the corresponding MddFile object.
The `MddFile` object is stored internally and will be directly returned whenever you call `$read_mdd()` again. You can force to rerun the design-day-only simulation again to update the contents by setting `update` to `TRUE`.
`$read_rdd()` and `read_mdd()` are useful when adding input and output parameters using `$input()` and `$output()`, respectively.

Returns: An MddFile object.

Examples:

```
\dontrun{
bc$read_mdd()

# force to rerun
bc$read_mdd(update = TRUE)
}
```

Method `input()`**:** Set input parameters

Usage:

```
BayesCalibJob$input(
  key_value = NULL,
  name = NULL,
  reporting_frequency = NULL,
  append = FALSE
)
```

Arguments:

`key_value` Key value name for variables. If not specified, `"*"` are used for all variables. `key_value` can also be an `RddFile`, `MddFile` or a `data.frame()`. Please see description above.

`name` Variable names listed in RDD or MDD.

`reporting_frequency` Variable reporting frequency for **all** variables. If `NULL`, `"Timestep"` are used for all variables. All possible values: `"Detailed"`, `"Timestep"`, `"Hourly"`, `"Daily"`, `"Monthly"`, `"RunPeriod"`, `"Environment"`, and `"Annual"`. Default: `NULL`.

`append` Whether to append input variables at the end of existing ones. A special value `NULL` can be given to remove all existing parameters. Default: `FALSE`.

Details: `$input()` takes input parameter definitions in a similar pattern as you set output variables in `Output:Variable` and `Output:Meter` class and returns a `data.table::data.table()` containing the information of input parameters. Only variables in RDD are allowed.The returned `data.table::data.table()` has 5 columns:

- `index`: Indices of input or output parameters.
- `class`: The class that parameters belong to. Will be either `Output:Variable` or `Output:Meter`.
- `key_value`: Key value name for variables.
- `variable_name`: Variable names listed in RDD or MDD.
- `reporting_frequency`: Variable reporting frequency.

If calling without any argument, the existing input parameters are directly returned, e.g. `bc$input()`. You can remove all existing input parameters by setting `append` to `NULL`, e.g. `bc$input(append = NULL)`.

`key_value` accepts 3 different formats:

- A character vector.
- An RddFile object. It can be retrieved using `$read_rdd()`. In this case, `name` argument will be ignored, as its values are directly taken from variable names in input RddFile object. For example:

```
bc$input(bc$read_rdd()[1:5])
```

- A data.frame() with valid format for adding Output:Variable and Output:Meter objects using eplusr::Idf$load(). In this case, name argument will be ignored. For example:

```
bc$input(eplusr::rdd_to_load(bc$read_rdd()[1:5]))
```

Returns: A data.table::data.table().

Examples:

```
\dontrun{
# explicitly specify input variable name
bc$input(name = "fan air mass flow rate", reporting_frequency = "hourly")

# use an RddFile
bc$input(bc$read_rdd()[1:5])

# use a data.frame
bc$input(eplusr::rdd_to_load(bc$read_rdd()[1:5]))

# get existing input
bc$input()
}
```

Method `output()`**:** Set output parameters

Usage:

```
BayesCalibJob$output(
  key_value = NULL,
  name = NULL,
  reporting_frequency = NULL,
  append = FALSE
)
```

Arguments:

`key_value` Key value name for variables. If not specified, `"*"` are used for all variables. `key_value` can also be an `RddFile`, `MddFile` or a `data.frame()`. Please see description above.

`name` Variable names listed in RDD or MDD.

`reporting_frequency` Variable reporting frequency for **all** variables. If `NULL`, `"Timestep"` are used for all variables. All possible values: `"Detailed"`, `"Timestep"`, `"Hourly"`, `"Daily"`, `"Monthly"`, `"RunPeriod"`, `"Environment"`, and `"Annual"`. Default: `NULL`.

`append` Whether to append input variables at the end of existing ones. A special value `NULL` can be given to remove all existing parameters. Default: `FALSE`.

Details: `$output()` takes output parameter definitions in a similar pattern as you set output variables in `Output:Variable` and `Output:Meter` class and returns a `data.table::data.table()` containing the information of output parameters. Unlike `$input()` both variables in RDD and MDD are allowd. The returned data.table has 5 columns:

- `index`: Indices of input or output parameters.
- `class`: The class that parameters belong to. Will be either `Output:Variable` or `Output:Meter`.

- `key_value`: Key value name for variables.
- `variable_name`: Variable names listed in RDD or MDD.
- `reporting_frequency`: Variable reporting frequency.

If calling without any argument, the existing output parameters are directly returned, e.g. `bc$output()`. You can remove all existing parameter by setting append to `NULL`, e.g. `bc$output(append = NULL)`.

`key_value` accepts 3 different formats:

- A character vector.
- An RddFile object or an MddFile object. They can be retrieved using `$read_rdd()` and `$read_mdd()`, respectively. In this case, `name` argument will be ignored, as its values are directly taken from variable names in input RddFile object or MddFile object. For example: `bc$output(bc$read_mdd()[1:5])`
- A `data.frame()` with valid format for adding `Output:Variable` and `Output:Meter` objects using Idf$load(). In this case, `name` argument will be ignored. For example:

```
bc$output(eplusr::mdd_to_load(bc$read_mdd()[1:5]))
```

Returns: A `data.table::data.table()`.

Examples:

```
\dontrun{
# explicitly specify input variable name
bc$output(name = "fan electric power", reporting_frequency = "hourly")

# use an RddFile or MddFile
bc$output(bc$read_rdd()[6:10])
bc$output(bc$read_mdd()[6:10])

# use a data.frame
bc$output(eplusr::rdd_to_load(bc$read_mdd()[6:10]))

# get existing input
bc$output()
}
```

Method `param()`**:** Set parameters for Bayesian calibration

Usage:

```
BayesCalibJob$param(..., .names = NULL, .num_sim = 30L)
```

Arguments:

`...` Lists of paramter definitions. Please see above on the syntax.

`.names` A character vector of the parameter names. If `NULL`, the parameter will be named in format `t` + `number`, where `number` is the index of parameter. Default: `NULL`.

`.num_sim` An positive integer specifying the number of simulations to run for each combination of calibration parameter value. Default: `30L`.

Details: `$param()` takes parameter definitions in list format, which is similar to `$set()` in eplusr::Idf class except that each field is not assigned with a single value, but a numeric vector of length 2, indicating the minimum and maximum value of each parameter.

Similar like the way of modifying object field values in eplusr::Idf$set(), there are 3 different ways of defining a parameter in epluspar:

- `object = list(field = c(min, max))`: Where `object` is a valid object ID or name. Note object ID should be denoted with two periods `..`, e.g. `..10` indicates the object with ID `10`, It will set that specific field in that object as one parameter.
- `.(object, object) := list(field = c(min, max))`: Simimar like above, but note the use of `.()` in the left hand side. You can put multiple object ID or names in `.()`. It will set the field of all specified objects as one parameter.
- `class := list(field = c(min, max, levels))`: Note the use of `:=` instead of `=`. The main difference is that, unlike `=`, the left hand side of `:=` should be a valid class name in current eplusr::Idf. It will set that field of all objects in specified class as one parameter.

For example, the code block below defines 4 calibration parameters:

- Field `Fan Total Efficiency` in object named `Supply Fan 1` in class `Fan:VariableVolume` class, with minimum and maximum being 0.1 and 1.0, respectively.
- Field `Thickness` in all objects in class `Material`, with minimum and maximum being 0.01 and 1.0, respectively.
- Field `Conductivity` in all objects in class `Material`, with minimum and maximum being 0.1 and 0.6, respectively.
- Field `Watts per Zone Floor Area` in objects `Light1` and `Light2` in class `Lights`, with minimum and maximum being 10 and 30, respectively.

```
bc$param(
    `Supply Fan 1` = list(Fan_Total_Efficiency = c(min = 0.1, max = 1.0)),
    Material := list(Thickness = c(0.01, 1), Conductivity = c(0.1, 0.6)),
   .("Light1", "Light2") := list(Watts_per_Zone_Floor_Area = c(10, 30))
)
```

All models created using `$param()` will be named in the same pattern, i.e. `Case_ParameterName(ParamterValue)...`. Note that only paramter names will be abbreviated using `abbreviate()` with `minlength` being `5L` and `use.classes` being `TRUE`. If samples contain duplications, `make.unique()` will be called to make sure every model has a unique name.

Returns: The modified `BayesCalibJob` object itself.

Examples:

```
\dontrun{
bc$param(
    `Supply Fan 1` = list(Fan_Total_Efficiency = c(min = 0.1, max = 1.0)),
    Material := list(Thickness = c(0.01, 1), Conductivity = c(0.1, 0.6)),
   .("Light1", "Light2") := list(Watts_per_Zone_Floor_Area = c(10, 30))
)
}
```

Method `apply_measure()`**:** Set parameters for Bayesian calibration using function

Usage:

```
BayesCalibJob$apply_measure(measure, ..., .num_sim = 30L)
```

Arguments:

`measure` A function that takes an eplusr::Idf and other arguments as input and returns an eplusr::Idf object as output.

`...` Arguments **except first** `Idf` **argument** that are passed to that `measure`.

`.num_sim` An positive integer specifying the number of simulations to run taking into account of all parameter combinations. Default: `30L`.

Details: `$apply_measure()` works in a similar way as the `$apply_measure` in eplusr::ParametricJob class, with only exception that each argument supplied in `...` should be a numeric vector of length 2, indicating the minimum value and maximum value of each parameter.
Basically `$apply_measure()` allows to apply a measure to an eplusr::Idf. A measure here is just a function that takes an eplusr::Idf object and other arguments as input, and returns a modified eplusr::Idf object as output.
The names of function parameter will be used as the names of calibration parameter. For example, the equivalent version of specifying parameters described in `$param()` using `$apply_measure()` can be:

```
# set calibration parameters using $apply_measure()
# (a) first define a "measure"
measure <- function (idf, efficiency, thickness, conducitivy, lpd) {
    idf$set(
        `Supply Fan 1` = list(Fan_Total_Efficiency = efficiency),
        Material := list(Thickness = thickness, Conductivity = conducivity)
        .("Light1", "Light2") := list(Watts_per_Zone_Floor_Area = lpd)
    )

    idf
}

# (b) then apply that measure with parameter space definitions as
# function arguments
bc$apply_measure(measure,
    efficiency = c(min = 0.1, max = 1.0),
    thickness = c(0.01, 1), conductivity = c(0.1, 0.6),
    lpd = c(10, 30)
)
```

All models created using `$apply_measure()` will be named in the same pattern, i.e. `Case_ParameterName(ParamterVal`
Note that only paramter names will be abbreviated using `abbreviate()` with `minlength` being `5L` and `use.classes` being `TRUE`. If samples contain duplications, `make.unique()` will be called to make sure every model has a unique name.

Returns: The modified `BayesCalibJob` object itself.

Examples:

```
\dontrun{
# set calibration parameters using $apply_measure()
# (a) first define a "measure"
measure <- function (idf, efficiency, thickness, conducitivy, lpd) {
    idf$set(
        `Supply Fan 1` = list(Fan_Total_Efficiency = efficiency),
        Material := list(Thickness = thickness, Conductivity = conducivity)
        .("Light1", "Light2") := list(Watts_per_Zone_Floor_Area = lpd)
    )
```

```
    idf
}

# (b) then apply that measure with parameter space definitions as
# function arguments
bc$apply_measure(measure,
    efficiency = c(min = 0.1, max = 1.0),
    thickness = c(0.01, 1), conductivity = c(0.1, 0.6),
    lpd = c(10, 30)
)
}
```

Method `samples()`**:** Get sampled parameter values

Usage:

```
BayesCalibJob$samples()
```

Details: `$samples()` returns a `data.table::data.table()` which contains the sampled value for each parameter using Random Latin Hypercube Sampling method. The returned `data.table::data.table()` has `1 + n` columns, where `n` is the parameter number, and `1` indicates an extra column named `case` giving the index of each sample.
Note that if `$samples()` is called before input and output parameters being set using `$input()`, and `$output()`, only the sampling will be performed and no parametric models will be created. This is because information of input and output parameters are needed in order to make sure that corresponding variables will be reported during simulations. In this case, you can use `$models()`, to create those models.

Returns: A `data.table::data.table()`.

Examples:

```
\dontrun{
bc$samples()
}
```

Method `models()`**:** Get parametric models

Usage:

```
BayesCalibJob$models()
```

Details: `$models()` returns a list of parametric eplusr::Idf objects created using calibration parameter values genereated using Random Latin Hypercube Sampling. As stated above, parametric models can only be created after input, output and calibration parameters have all be set using `$input()`, `$output()` and `$param()` (or `$apply_measure()`), respectively.
All models will be named in the same pattern, i.e. `Case_ParameterName(ParamterValue)...`. Note that paramter names will be abbreviated using `abbreviate()` with `minlength` being `5L` and `use.classes` being `TRUE`.

Returns: A named list of eplusr::Idf objects.

Examples:

```
\dontrun{
bc$models()
}
```

Method `data_sim()`**:** Collect simulation data

Usage:

```
BayesCalibJob$data_sim(resolution = NULL, exclude_ddy = TRUE, all = FALSE)
```

Arguments:

`resolution` A character string specifying a time unit or a multiple of a unit to change the time resolution of returned simulation data. Valid base units are `min`, `hour`, `day`, `week`, `month`, and `year`. Example: `10 mins`, `2 hours`, `1 day`. If `NULL`, the variable reporting frequency is used. Default: `NULL`.

`exclude_ddy` Whether to exclude design day data. Default: `TRUE`. Default: `FALSE`.

`all` If `TRUE`, extra columns are also included in the returned `data.table::data.table()` describing the simulation case and datetime components. Default: `FALSE`.

Details: `$data_sim()` returns a list of 2 `data.table::data.table()` which contains the simulated data of input and output parameters. These data will be stored internally and used during Bayesian calibration using Stan.
The `resolution` parameter can be used to specify the time resolution of returned data. Note that input time resolution cannot be smaller than the reporting frequency, otherwise an error will be issued.
The parameter is named in the same way as standard EnergyPlus csv output file, i.e. `KeyValue:VariableName [Unit](Frequency)`.
By default, `$data_sim()` returns minimal columns, i.e. the `Date/Time` column together with all input and output parameters are returned.
You can retrieve extra columns by setting `all` to `TRUE`. Those column include:

- `case`: Integer type. Indices of parametric simulations.
- `environment_period_index`: Integer type. The indice of environment.
- `environment_name`: Character type. A text string identifying the simulation environment.
- `simulation_days`: Integer type. Day of simulation.
- `datetime`: DateTime type. The date time of simulation result. Note that the year valueas are automatically calculated to meets the start day of week restriction for each simulation environment.
- `month`: Integer type. The month of reported date time.
- `day`: Integer type. The day of month of reported date time.
- `hour`: Integer type. The hour of reported date time.
- `minute`: Integer type. The minute of reported date time.
- `day_type`: Character type. The type of day, e.g. `Monday`, `Tuesday` and etc. Note that `day_type` will always be `NA` if `resolution` is specified.

Returns: A list of 2 `data.table::data.table()`.

Examples:

```
\dontrun{
bc$data_sim()
}
```

Method `data_field()`**:** Specify field measured data

Usage:

```
BayesCalibJob$data_field(output, new_input = NULL, all = FALSE)
```

Arguments:

`output` A `data.frame()` containing measured value of output parameters.

`new_input` A `data.frame()` containing newly measured value of input parameters used for prediction. If `NULL`, values of the first case in `$data_sim()` will be used.

`all` If `TRUE`, extra columns are also included in the returned `data.table::data.table()` describing the simulation case and datetime components. For details, please see `$data_sim()`. Default: `FALSE`.

Details: `$data_field()` takes a `data.frame()` of measured value of output parameters and returns a list of `data.table::data.table()`s which contains the measured value of input and output parameters, and newly measured value of input if applicable.
The specified `output` `data.frame()` is validated using criteria below:

- The column number should be the same as the number of output specified in `$output()`.
- The row number should be the same as the number of simulated values for each case extracted using `$data_sim()`.

For input parameters, the values of simulation data for the first case are directly used as the measured values.
Parameter `new_input` can be used to give a `data.frame()` of newly measured value of input parameters. The column number of input `data.frame()` should be the same as the number of input parameters specified in `$input()`. If not specified, the measured values of input parameters will be used for predictions.
All the data will be stored internally and used during Bayesian calibration using Stan.
Note that as `$data_field()` relies on the output of `$data_sim()`. to perform validation on the specified data, `$data_field()` cannot be called before `$data_sim()`. and internally stored data will be removed whenever `$data_sim()`. is called. This aims to make sure that simulated data and field data can be matched whenever the calibration is performed.

Returns: A list of 3 elements:

- `input`: a `data.table::data.table()` which is basically the input variable values of the first case in `$data_sim()`.
- `output`: a `data.table::data.table()` of output variable values.
- `new_output`: `NULL` or a `data.table::data.table()` of newly measured input variable values.

For details on the meaning of each columns, see `$data_sim()`.

Method `data_bc()`**:** Combine simulation data and field measured data

Usage:

```
BayesCalibJob$data_bc(data_field = NULL, data_sim = NULL)
```

Arguments:

`data_field` A `data.frame()` specifying field measured data. Should have the same structure as the output from `$data_field()`. If `NULL`, the output from `$data_field()` will be used. Default: `NULL`.

`data_sim` A `data.frame()` specifying field measured data. Should have the same structure as the output from `$data_sim()`. If `NULL`, the output from `$data_sim()` will be used. Default: `NULL`.

Details: `$data_bc()` takes a list of field data and simulated data, and returns a list that contains data input for Bayesian calibration using the Stan model from Chong (2018):

- `n`: Number of measured parameter observations.
- `n_pred`: Number of newly design points for predictions.
- `m`: Number of simulated observations.
- `p`: Number of input parameters.
- `q`: Number of calibration parameters.
- `yf`: Data of measured output after z-score standardization using data of simulated output.
- `yc`: Data of simulated output after z-score standardization.
- `xf`: Data of measured input after min-max normalization.
- `xc`: Data of simulated input after min-max normalization.
- `x_pred`: Data of new design points for predictions after min-max normalization.
- `tc`: Data of calibration parameters after min-max normalization.

Input `data_field` and `data_sim` should have the same structure as the output from `$data_field()` and `$data_sim()`. If `data_field` and `data_sim` is not specified, the output from `$data_field()` and `$data_sim()` will be used.

Returns: A list of 11 elements.

Examples:

```
\dontrun{
bc$data_bc()
}
```

Method `eplus_run()`**:** Run parametric simulations

Usage:

```
BayesCalibJob$eplus_run(
  dir = NULL,
  run_period = NULL,
  wait = TRUE,
  force = FALSE,
  copy_external = FALSE,
  echo = wait
)
```

Arguments:

`dir` The parent output directory for specified simulations. Outputs of each simulation are placed in a separate folder under the parent directory. If `NULL`, directory of seed model will be used. Default: `NULL`.

`run_period` A list giving a new `RunPeriod` object definition. If not `NULL`, only this new RunPeriod will take effect with all existing RunPeriod objects in the seed model being commented out. If `NULL`, existing run period in the seed model will be used. Default: `NULL`.

`wait` If `TRUE`, R will hang on and wait all EnergyPlus simulations finish. If `FALSE`, all EnergyPlus simulations are run in the background. Default: `TRUE`.

`force` Only applicable when the last simulation runs with `wait` equals to `FALSE` and is still running. If `TRUE`, current running job is forced to stop and a new one will start. Default: `FALSE`.

`copy_external` If `TRUE`, the external files that every `Idf` object depends on will also be copied into the simulation output directory. The values of file paths in the Idf will be changed automatically. Currently, only `Schedule:File` class is supported. This ensures that the output directory will have all files needed for the model to run. Default is `FALSE`.

`echo` Only applicable when `wait` is `TRUE`. Whether to print simulation status. Default: same as the value of `wait`.

Details: `$eplus_run()` runs all parametric models in parallel. Parameter `run_period` can be given to insert a new `RunPeriod` object. In this case, all existing `RunPeriod` objects in the seed model will be commented out.
Note that when `run_period` is given, value of field `Run Simulation for Weather File Run Periods` in `SimulationControl` class will be reset to `Yes` to make sure input run period can take effect.

Returns: The modified `BayesCalibJob` object itself.

Examples:

```
\dontrun{
# specify output directory and run period
bc$eplus_run(dir = tempdir(), run_period = list("example", 1, 1, 1, 31))

# run in the background
bc$eplus_run(wait = TRUE)
# see job status
bc$status()

# force to kill background job before running the new one
bc$eplus_run(force = TRUE)

# do not show anything in the console
bc$eplus_run(echo = FALSE)

# copy external files used in the model to simulation output directory
bc$eplus_run(copy_external = TRUE)
}
```

Method `eplus_kill()`**:** Kill current running EnergyPlus simulations

Usage:

```
BayesCalibJob$eplus_kill()
```

Details: `$eplus_kill()` kills all background EnergyPlus processes that are current running if possible. It only works when simulations run in non-waiting mode.

Returns: A single logical value of `TRUE` or `FALSE`, invisibly.

Examples:

```
\dontrun{
bc$eplus_kill()
}
```

Method `eplus_status()`**:** Get the EnergyPlus simulation status

Usage:

```
BayesCalibJob$eplus_status()
```

Details: `$eplus_status()` returns a named list of values indicates the status of the job:

- `run_before`: `TRUE` if the job has been run before. `FALSE` otherwise.
- `alive`: `TRUE` if the job is still running in the background. `FALSE` otherwise.
- `terminated`: `TRUE` if the job was terminated during last simulation. `FALSE` otherwise. `NA` if the job has not been run yet.
- `successful`: `TRUE` if all simulations ended successfully. `FALSE` if there is any simulation failed. `NA` if the job has not been run yet.
- `changed_after`: `TRUE` if the *seed model* has been modified since last simulation. `FALSE` otherwise.
- `job_status`: A `data.table::data.table()` contains meta data for each simulation job. For details, please see `run_multi()`. If the job has not been run before, a `data.table::data.table()` with 4 columns is returned:
  - `index`: The index of simulation
  - `status`: The status of simulation. As the simulation has not been run, `status` will always be "idle".
  - `idf`: The path of input IDF file.
  - `epw`: The path of input EPW file. If not provided, `NA` will be assigned.

Returns: A named list of 6 elements.

Examples:

```
\dontrun{
bc$eplus_status()
}
```

Method `eplus_output_dir()`**:** Get EnergyPlus simulation output directory

Usage:

```
BayesCalibJob$eplus_output_dir(which = NULL)
```

Arguments:

`which` An integer vector of the indexes or a character vector or names of parametric simulations. If `NULL`, results of all parametric simulations are returned. Default: `NULL`.

Details: `$eplus_output_dir()` returns the output directory of EnergyPlus simulation results.

Returns: A character vector.

Examples:

```
\dontrun{
# get output directories of all simulations
bc$eplus_output_dir()

# get output directories of specified simulations
bc$eplus_output_dir(c(1, 4))
}
```

Method `eplus_locate_output()`**:** Get paths of EnergyPlus output file

Usage:

```
BayesCalibJob$eplus_locate_output(which = NULL, suffix = ".err", strict = TRUE)
```

Arguments:

`which` An integer vector of the indexes or a character vector or names of parametric simulations. If `NULL`, results of all parametric simulations are returned. Default: `NULL`.

`suffix` A string that indicates the file extension of simulation output. Default: `".err"`.

`strict` If `TRUE`, it will check if the simulation was terminated, is still running or the file exists or not. Default: `TRUE`.

Details: `$eplus_locate_output()` returns the path of a single output file of specified simulations.

Returns: A character vector.

Examples:

```
\dontrun{
# get the file path of the error file
bc$eplus_locate_output(c(1, 4), ".err", strict = FALSE)

# can use to detect if certain output file exists
bc$eplus_locate_output(c(1, 4), ".expidf", strict = TRUE)
}
```

Method `eplus_errors()`**:** Read EnergyPlus simulation errors

Usage:

```
BayesCalibJob$eplus_errors(which = NULL, info = FALSE)
```

Arguments:

`which` An integer vector of the indexes or a character vector or names of parametric simulations. If `NULL`, results of all parametric simulations are returned. Default: `NULL`.

`info` If `FALSE`, only warnings and errors are printed. Default: `FALSE`.

Details: $eplus_errors() returns a list of ErrFile objects which contain all contents of the simulation error files (`.err`). If `info` is `FALSE`, only warnings and errors are printed.

Returns: A list of ErrFile objects.

Examples:

```
\dontrun{
bc$errors()

# show all information
bc$errors(info = TRUE)
}
```

Method `eplus_report_data_dict()`**:** Read report data dictionary from EnergyPlus SQL outputs

Usage:

```
BayesCalibJob$eplus_report_data_dict(which = NULL)
```

Arguments:

`which` An integer vector of the indexes or a character vector or names of parametric simulations. If `NULL`, results of all parametric simulations are returned. Default: `NULL`.

Details: `$eplus_report_data_dict()` returns a `data.table::data.table()` which contains all information about report data.
For details on the meaning of each columns, please see "2.20.2.1 ReportDataDictionary Table" in EnergyPlus "Output Details and Examples" documentation.

Returns: A `data.table::data.table()` of 10 columns:

- `case`: The model name. This column can be used to distinguish output from different simulations
- `report_data_dictionary_index`: The integer used to link the dictionary data to the variable data. Mainly useful when joining diferent tables
- `is_meter`: Whether report data is a meter data. Possible values: `0` and `1`
- `timestep_type`: Type of data timestep. Possible values: `Zone` and `HVAC System`
- `key_value`: Key name of the data
- `name`: Actual report data name
- `reporting_frequency`:
- `schedule_name`: Name of the the schedule that controls reporting frequency.
- `units`: The data units

Examples:

```
\dontrun{
bc$eplus_report_data_dict(c(1, 4))
}
```

Method `eplus_report_data()`**:** Read EnergyPlus report data

Usage:

```
BayesCalibJob$eplus_report_data(
  which = NULL,
  key_value = NULL,
  name = NULL,
  year = NULL,
  tz = "UTC",
  all = FALSE,
  wide = FALSE,
  period = NULL,
  month = NULL,
  day = NULL,
  hour = NULL,
  minute = NULL,
  interval = NULL,
  simulation_days = NULL,
  day_type = NULL,
```

```
  environment_name = NULL
)
```

Arguments:

`which` An integer vector of the indexes or a character vector or names of parametric simulations. If `NULL`, results of all parametric simulations are returned. Default: `NULL`.

`key_value` A character vector to identify key values of the data. If `NULL`, all keys of that variable will be returned. `key_value` can also be data.frame that contains `key_value` and `name` columns. In this case, `name` argument in `$eplus_report_data()` is ignored. All available `key_value` for current simulation output can be obtained using `$eplus_report_data_dict()`. Default: `NULL`.

`name` A character vector to identify names of the data. If `NULL`, all names of that variable will be returned. If `key_value` is a data.frame, `name` is ignored. All available `name` for current simulation output can be obtained using `$eplus_report_data_dict()`. Default: `NULL`.

`year` Year of the date time in column `datetime`. If `NULL`, it will calculate a year value that meets the start day of week restriction for each environment. Default: `NULL`.

`tz` Time zone of date time in column `datetime`. Default: `"UTC"`.

`all` If `TRUE`, extra columns are also included in the returned `data.table::data.table()`.

`wide` If `TRUE`, the output is formated in the same way as standard EnergyPlus csv output file.

`period` A Date or POSIXt vector used to specify which time period to return. The year value does not matter and only month, day, hour and minute value will be used when subsetting. If `NULL`, all time period of data is returned. Default: `NULL`.

`month, day, hour, minute` Each is an integer vector for month, day, hour, minute subsetting of `datetime` column when querying on the SQL database. If `NULL`, no subsetting is performed on those components. All possible `month`, `day`, `hour` and `minute` can be obtained using `$eplus_report_data_dict()`. Default: `NULL`.

`interval` An integer vector used to specify which interval length of report to extract. If `NULL`, all interval will be used. Default: `NULL`.

`simulation_days` An integer vector to specify which simulation day data to extract. Note that this number resets after warmup and at the beginning of an environment period. All possible `simulation_days` can be obtained using `$eplus_report_data_dict()`. If `NULL`, all simulation days will be used. Default: `NULL`.

`day_type` A character vector to specify which day type of data to extract. All possible day types are: `Sunday`, `Monday`, `Tuesday`, `Wednesday`, `Thursday`, `Friday`, `Saturday`, `Holiday`, `SummerDesignDay`, `WinterDesignDay`, `CustomDay1`, and `CustomDay2`. All possible values for current simulation output can be obtained using `$eplus_report_data_dict()`.

`environment_name` A character vector to specify which environment data to extract. If `NULL`, all environment data are returned. Default: `NULL`. All possible `environment_name` for current simulation output can be obtained using:

```
$read_table(NULL, "EnvironmentPeriods")
```

`case` If not `NULL`, a character column will be added indicates the case of this simulation. If `"auto"`, the name of the IDF file without extension is used.

Details: `$eplus_report_data()` extracts the report data in a `data.table::data.table()` using key values, variable names and other specifications.
`$eplus_report_data()` can also directly take all or subset output from `$eplus_report_data_dict()` as input, and extract all data specified.
The returned column numbers varies depending on `all` argument.

- `all` is `FALSE`, the returned `data.table::data.table()` has 6 columns:
  - `case`: The model name. This column can be used to distinguish output from different simulations
  - `datetime`: The date time of simulation result
  - `key_value`: Key name of the data
  - `name`: Actual report data name
  - `units`: The data units
  - `value`: The data value
- `all` is `TRUE`, besides columns described above, extra columns are also included:
  - `month`: The month of reported date time
  - `day`: The day of month of reported date time
  - `hour`: The hour of reported date time
  - `minute`: The minute of reported date time
  - `dst`: Daylight saving time indicator. Possible values: `0` and `1`
  - `interval`: Length of reporting interval
  - `simulation_days`: Day of simulation
  - `day_type`: The type of day, e.g. `Monday`, `Tuesday` and etc.
  - `environment_period_index`: The indice of environment.
  - `environment_name`: A text string identifying the environment.
  - `is_meter`: Whether report data is a meter data. Possible values: `0` and `1`
  - `type`: Nature of data type with respect to state. Possible values: `Sum` and `Avg`
  - `index_group`: The report group, e.g. `Zone`, `System`
  - `timestep_type`: Type of data timestep. Possible values: `Zone` and `HVAC System`
  - `reporting_frequency`: The reporting frequency of the variable, e.g. `HVAC System Timestep`, `Zone Timestep`.
  - `schedule_name`: Name of the the schedule that controls reporting frequency.

With the `datetime` column, it is quite straightforward to apply time-series analysis on the simulation output. However, another painful thing is that every simulation run period has its own `Day of Week for Start Day`. Randomly setting the `year` may result in a date time series that does not have the same start day of week as specified in the RunPeriod objects.
eplusr provides a simple solution for this. By setting `year` to `NULL`, which is the default behavior, eplusr will calculate a year value (from current year backwards) for each run period that compliances with the start day of week restriction.
It is worth noting that EnergyPlus uses 24-hour clock system where 24 is only used to denote midnight at the end of a calendar day. In EnergyPlus output, "00:24:00" with a time interval being 15 mins represents a time period from "00:23:45" to "00:24:00", and similarly "00:15:00" represents a time period from "00:24:00" to "00:15:00" of the next day. This means that if current day is Friday, day of week rule applied in schedule time period "00:23:45" to "00:24:00" (presented as "00:24:00" in the output) is also Friday, but not Saturday. However, if you try to get the day of week of time "00:24:00" in R, you will get Saturday, but not Friday. This introduces inconsistency and may cause problems when doing data analysis considering day of week value.
With `wide` equals `TRUE`, `$eplus_report_data()` will format the simulation output in the same way as standard EnergyPlus csv output file. Sometimes this can be useful as there may be existing tools/workflows that depend on this format. When both `wide` and `all` are `TRUE`, columns of

runperiod environment names and date time components are also returned, including: `environment_period_index`, `"e`
`simulation_days`, `datetime`, `month`, `day`, `hour`, `minute`, `day_type`.
For convenience, input character arguments matching in `$eplus_report_data()` are **case-insensitive**.

Returns: A `data.table::data.table()`.

Examples:

```
\dontrun{
# read report data
bc$report_data(c(1, 4))

# specify output variables using report data dictionary
dict <- bc$report_data_dict(1)
bc$report_data(c(1, 4), dict[units == "C"])

# specify output variables using 'key_value' and 'name'
bc$report_data(c(1, 4), "environment", "site outdoor air drybulb temperature")

# explicitly specify year value and time zone
bc$report_data(c(1, 4), dict[1], year = 2020, tz = "Etc/GMT+8")

# get all possible columns
bc$report_data(c(1, 4), dict[1], all = TRUE)

# return in a format that is similar as EnergyPlus CSV output
bc$report_data(c(1, 4), dict[1], wide = TRUE)

# return in a format that is similar as EnergyPlus CSV output with
# extra columns
bc$report_data(c(1, 4), dict[1], wide = TRUE, all = TRUE)

# only get data at the working hour on the first Monday
bc$report_data(c(1, 4), dict[1], hour = 8:18, day_type = "monday", simulation_days = 1:7)
}
```

Method `eplus_tabular_data()`**:** Read EnergyPlus tabular data

Usage:

```
BayesCalibJob$eplus_tabular_data(
  which = NULL,
  report_name = NULL,
  report_for = NULL,
  table_name = NULL,
  column_name = NULL,
  row_name = NULL
)
```

Arguments:

`which` An integer vector of the indexes or a character vector or names of parametric simulations. If `NULL`, results of all parametric simulations are returned. Default: `NULL`.

`report_name`, `report_for`, `table_name`, `column_name`, `row_name` Each is a character vector for subsetting when querying the SQL database. For the meaning of each argument, please see the description above.

Details: `$eplus_tabular_data()` extracts the tabular data in a `data.table::data.table()` using report, table, column and row name specifications. The returned `data.table::data.table()` has 9 columns:

- `case`: The model name. This column can be used to distinguish output from different simulations
- `index`: Tabular data index
- `report_name`: The name of the report that the record belongs to
- `report_for`: The `For` text that is associated with the record
- `table_name`: The name of the table that the record belongs to
- `column_name`: The name of the column that the record belongs to
- `row_name`: The name of the row that the record belongs to
- `units`: The units of the record
- `value`: The value of the record **in string format**

For convenience, input character arguments matching in `$eplus_tabular_data()` are **case-insensitive**.

Returns: A `data.table::data.table()` with 8 columns.

Examples:

```
\dontrun{
# read all tabular data
bc$eplus_tabular_data(c(1, 4))

# explicitly specify data you want
str(bc$eplus_tabular_data(c(1, 4),
    report_name = "AnnualBuildingUtilityPerformanceSummary",
    table_name = "Site and Source Energy",
    column_name = "Total Energy",
    row_name = "Total Site Energy"
))
}
```

Method `eplus_save()`**:** Save EnergyPlus parametric models

Usage:

```
BayesCalibJob$eplus_save(dir = NULL, separate = TRUE, copy_external = FALSE)
```

Arguments:

`dir` The parent output directory for models to be saved. If `NULL`, the directory of the seed model will be used. Default: `NULL`.

`separate` If `TRUE`, all models are saved in a separate folder with each model's name under specified directory. If `FALSE`, all models are saved in the specified directory. Default: `TRUE`.

`copy_external` Only applicable when `separate` is `TRUE`. If `TRUE`, the external files that every `Idf` object depends on will also be copied into the saving directory. The values of file paths

in the Idf will be changed automatically. Currently, only `Schedule:File` class is supported. This ensures that the output directory will have all files needed for the model to run. Default: `FALSE`.

Details: `$eplus_save()` saves all parametric models in specified folder. An error will be issued if no measure has been applied.

Returns: A `data.table::data.table()` with two columns:

- model: The path of saved parametric model files.
- weather: The path of saved weather files.

Examples:

```
\dontrun{
# save all parametric models with each model in a separate folder
bc$save(tempdir())

# save all parametric models with all models in the same folder
bc$save(tempdir(), separate = FALSE)
}
```

Method `stan_run()`**:** Run Bayesian calibration using Stan

Usage:

```
BayesCalibJob$stan_run(
  file = NULL,
  data = NULL,
  iter = 2000L,
  chains = 4L,
  echo = TRUE,
  mc.cores = parallel::detectCores(),
  all = FALSE,
  merge = TRUE,
  ...
)
```

Arguments:

`file` The path to the Stan program to use. If `NULL`, the pre-compiled Stan code from Chong (2018) will be used. Default: `NULL`.

`data` Only applicable when `file` is not `NULL`. The data to be used for Bayesian calibration. If `NULL`, the data that `$data_bc()` returns is used. Default: `NULL`.

`iter` A positive integer specifying the number of iterations for each chain (including warmup). Default: `2000`.

`chains` A positive integer specifying the number of Markov chains. Default: `4`.

`echo` Only applicable when `file` is NULL. Whether to print the summary of Informational Messages to the screen after a chain is finished or a character string naming a path where the summary is stored. Default: `TRUE`.

`mc.cores` An integer specifying how many cores to be used for Stan. Default: `parallel::detectCores()`.

`all` If `FALSE`, among above meta data columns, only `index`, `type` and `Date/Time` will be returned. Default: `FALSE`.

merge If TRUE, y_pred in returned list will merge all $data_field(), and predicted output into one data.table::data.table() with all predicted values put in columns with a \\[prediction\\] prefix. If FALSE, similar like above, but combine rows of field measured output and predicted output together, with a new column type added giving field indicating field measured output and prediction indicating predicted output. Default: TRUE.

... Additional arguments to pass to rstan::sampling (when file is NULL) or rstan::stan (when file is not NULL).

Details: $stan_run() runs Bayesian calibration using Stan and returns a list of 2 elements:

- fit: An object of S4 class rstan::stanfit.
- y_pred: The output of $prediction()

Returns: A list of 2 elements.

Examples:

```
\dontrun{
bc$stan_run()
}
```

Method stan_file(): Extract Stan file for Bayesian calibration

Usage:

```
BayesCalibJob$stan_file(path = NULL)
```

Arguments:

path A path to save the Stan code. If NULL, a character vector of the Stan code is returned.

Details: $stan_file() saves the Stan file used internally for Bayesian calibration. If no path is given, a character vector of the Stan code is returned. If given, the code will be save to the path and the file path is returned.

Examples:

```
\dontrun{
bc$stan_file()
}
```

Method post_dist(): Extract posterior distributions of calibrated parameters

Usage:

```
BayesCalibJob$post_dist()
```

Details: $post_dist() extracted calibrated parameter posterior distributions based on the results of $stan_run() and returns a data.table::data.table() with each parameter values filling one column. The parameter names are defined by the .names arguments in the $param().

Returns: A data.table::data.table().

Examples:

```
\dontrun{
bc$post_dist()
}
```

Method `prediction()`**:** Extract predictions of output variables

Usage:

```
BayesCalibJob$prediction(all = FALSE, merge = TRUE)
```

Arguments:

`all` If `FALSE`, among above meta data columns, only `index`, `type` and `Date/Time` will be returned. Default: `FALSE`.

`merge` If `TRUE`, `y_pred` in returned list will merge all `$data_field()`, and predicted output into one `data.table::data.table()` with all predicted values put in columns with a `\\[prediction\\]` prefix. If `FALSE`, similar like above, but combine rows of field measured output and predicted output together, with a new column `type` added giving `field` indicating field measured output and `prediction` indicating predicted output. Default: `TRUE`.

Details: `$prediction()` calculates predicted output variable values based on the results of `$stan_run()` and returns a `data.table::data.table()` which combines the output of `$data_field()` and predicted output values.
Possible returned meta data columns:

- `index`: Integer type. Row indices of field input data in `$data_field()`
- `sample`: Integer type. Sample indices of the MCMC.
- `type`: Character type. Only exists when `merge` is `FALSE`. The type of output values. `field` indicates field measured output values while `prediction` means predicted output values.
- `Data/Time`: Character type. The date time in EnergyPlus-format.
- `environment_period_index`: Integer type. The indice of environment.
- `environment_name`: Character type. A text string identifying the simulation environment.
- `simulation_days`: Integer type. Day of simulation.
- `datetime`: DateTime type. The date time of simulation result. Note that the year valueas are automatically calculated to meets the start day of week restriction for each simulation environment.
- `month`: Integer type. The month of reported date time.
- `day`: Integer type. The day of month of reported date time.
- `hour`: Integer type. The hour of reported date time.
- `minute`: Integer type. The minute of reported date time.
- `day_type`: Character type. The type of day, e.g. `Monday`, `Tuesday` and etc. Note that `day_type` will always be `NA` if `resolution` is specified.

Returns: A `data.table::data.table()` with 1 column `sample` giving the sample indices from MCMC, plus the same number of columns as given calibrated parameters.

Examples:

```
\dontrun{
bc$prediction()
}
```

Method `evaluate()`**:** Calculate statistical indicators of output variable predictions

Usage:

```
BayesCalibJob$evaluate(funs = list(nmbe, cvrmse))
```

Arguments:

`funs` A list of functions that takes the simulation results as the first argument and the measured results as the second argument. Default: `list(cvrmse, nmbe)`.

Details: `$evalute()` quantify the uncertainty of output variable predictions from each MCMC sample gathered from `$prediction()` by calculating the statistical indicators.
The default behavior is to evaluate the principal uncertainty indices used in ASHRAE Guideline 14 are Normalized Mean Bias Error (NMBE) and Coefficient of Variation of the Root Mean Square Error (CVRMSE).

Returns: A `data.table::data.table()` with 1 column `sample` giving the sample indices from MCMC, plus the same number of columns as given evaluation functions.

Examples:

```
\dontrun{
bc$evaluate()
}
```

Note

Currently, when using builtin Bayesian calibration algorithm, only one prediction output variable is supported. An error will be issued if multiple output variables found in `data`.

Author(s)

Hongyuan Jia, Adrian Chong

References

A. Chong and K. Menberg, "Guidelines for the Bayesian calibration of building energy models", Energy and Buildings, vol. 174, pp. 527–547. DOI: 10.1016/j.enbuild.2018.06.028

Examples

```
## ------------------------------------------------
## Method `BayesCalibJob$new`
## ------------------------------------------------

## Not run:
if (eplusr::is_avail_eplus(8.8)) {
    idf_name <- "1ZoneUncontrolled.idf"
    epw_name <-  "USA_CA_San.Francisco.Intl.AP.724940_TMY3.epw"

    idf_path <- file.path(eplusr::eplus_config(8.8)$dir, "ExampleFiles", idf_name)
    epw_path <- file.path(eplusr::eplus_config(8.8)$dir, "WeatherData", epw_name)

    # create from local files
    BayesCalibJob$new(idf_path, epw_path)

    # create from an Idf and an Epw object
    bc <- BayesCalibJob$new(eplusr::read_idf(idf_path), eplusr::read_epw(epw_path))
```

```
}

## End(Not run)


## ------------------------------------------------
## Method `BayesCalibJob$read_rdd`
## ------------------------------------------------

## Not run:
bc$read_rdd()

# force to rerun
bc$read_rdd(update = TRUE)

## End(Not run)


## ------------------------------------------------
## Method `BayesCalibJob$read_mdd`
## ------------------------------------------------

## Not run:
bc$read_mdd()

# force to rerun
bc$read_mdd(update = TRUE)

## End(Not run)


## ------------------------------------------------
## Method `BayesCalibJob$input`
## ------------------------------------------------

## Not run:
# explicitly specify input variable name
bc$input(name = "fan air mass flow rate", reporting_frequency = "hourly")

# use an RddFile
bc$input(bc$read_rdd()[1:5])

# use a data.frame
bc$input(eplusr::rdd_to_load(bc$read_rdd()[1:5]))

# get existing input
bc$input()

## End(Not run)


## ------------------------------------------------
## Method `BayesCalibJob$output`
```

```
## ------------------------------------------------

## Not run:
# explicitly specify input variable name
bc$output(name = "fan electric power", reporting_frequency = "hourly")

# use an RddFile or MddFile
bc$output(bc$read_rdd()[6:10])
bc$output(bc$read_mdd()[6:10])

# use a data.frame
bc$output(eplusr::rdd_to_load(bc$read_mdd()[6:10]))

# get existing input
bc$output()

## End(Not run)


## ------------------------------------------------
## Method `BayesCalibJob$param`
## ------------------------------------------------

## Not run:
bc$param(
    `Supply Fan 1` = list(Fan_Total_Efficiency = c(min = 0.1, max = 1.0)),
    Material := list(Thickness = c(0.01, 1), Conductivity = c(0.1, 0.6)),
   .("Light1", "Light2") := list(Watts_per_Zone_Floor_Area = c(10, 30))
)

## End(Not run)


## ------------------------------------------------
## Method `BayesCalibJob$apply_measure`
## ------------------------------------------------

## Not run:
# set calibration parameters using $apply_measure()
# (a) first define a "measure"
measure <- function (idf, efficiency, thickness, conducitivy, lpd) {
    idf$set(
        `Supply Fan 1` = list(Fan_Total_Efficiency = efficiency),
        Material := list(Thickness = thickness, Conductivity = conducivity)
        .("Light1", "Light2") := list(Watts_per_Zone_Floor_Area = lpd)
    )

    idf
}

# (b) then apply that measure with parameter space definitions as
# function arguments
bc$apply_measure(measure,
```

```
    efficiency = c(min = 0.1, max = 1.0),
    thickness = c(0.01, 1), conductivity = c(0.1, 0.6),
    lpd = c(10, 30)
)

## End(Not run)


## ------------------------------------------------
## Method `BayesCalibJob$samples`
## ------------------------------------------------

## Not run:
bc$samples()

## End(Not run)


## ------------------------------------------------
## Method `BayesCalibJob$models`
## ------------------------------------------------

## Not run:
bc$models()

## End(Not run)


## ------------------------------------------------
## Method `BayesCalibJob$data_sim`
## ------------------------------------------------

## Not run:
bc$data_sim()

## End(Not run)


## ------------------------------------------------
## Method `BayesCalibJob$data_bc`
## ------------------------------------------------

## Not run:
bc$data_bc()

## End(Not run)


## ------------------------------------------------
## Method `BayesCalibJob$eplus_run`
## ------------------------------------------------

## Not run:
```

```
# specify output directory and run period
bc$eplus_run(dir = tempdir(), run_period = list("example", 1, 1, 1, 31))

# run in the background
bc$eplus_run(wait = TRUE)
# see job status
bc$status()

# force to kill background job before running the new one
bc$eplus_run(force = TRUE)

# do not show anything in the console
bc$eplus_run(echo = FALSE)

# copy external files used in the model to simulation output directory
bc$eplus_run(copy_external = TRUE)

## End(Not run)


## ------------------------------------------------
## Method `BayesCalibJob$eplus_kill`
## ------------------------------------------------

## Not run:
bc$eplus_kill()

## End(Not run)


## ------------------------------------------------
## Method `BayesCalibJob$eplus_status`
## ------------------------------------------------

## Not run:
bc$eplus_status()

## End(Not run)


## ------------------------------------------------
## Method `BayesCalibJob$eplus_output_dir`
## ------------------------------------------------

## Not run:
# get output directories of all simulations
bc$eplus_output_dir()

# get output directories of specified simulations
bc$eplus_output_dir(c(1, 4))

## End(Not run)
```

```
## ------------------------------------------------
## Method `BayesCalibJob$eplus_locate_output`
## ------------------------------------------------

## Not run:
# get the file path of the error file
bc$eplus_locate_output(c(1, 4), ".err", strict = FALSE)

# can use to detect if certain output file exists
bc$eplus_locate_output(c(1, 4), ".expidf", strict = TRUE)

## End(Not run)


## ------------------------------------------------
## Method `BayesCalibJob$eplus_errors`
## ------------------------------------------------

## Not run:
bc$errors()

# show all information
bc$errors(info = TRUE)

## End(Not run)


## ------------------------------------------------
## Method `BayesCalibJob$eplus_report_data_dict`
## ------------------------------------------------

## Not run:
bc$eplus_report_data_dict(c(1, 4))

## End(Not run)


## ------------------------------------------------
## Method `BayesCalibJob$eplus_report_data`
## ------------------------------------------------

## Not run:
# read report data
bc$report_data(c(1, 4))

# specify output variables using report data dictionary
dict <- bc$report_data_dict(1)
bc$report_data(c(1, 4), dict[units == "C"])

# specify output variables using 'key_value' and 'name'
bc$report_data(c(1, 4), "environment", "site outdoor air drybulb temperature")
```

```
# explicitly specify year value and time zone
bc$report_data(c(1, 4), dict[1], year = 2020, tz = "Etc/GMT+8")

# get all possible columns
bc$report_data(c(1, 4), dict[1], all = TRUE)

# return in a format that is similar as EnergyPlus CSV output
bc$report_data(c(1, 4), dict[1], wide = TRUE)

# return in a format that is similar as EnergyPlus CSV output with
# extra columns
bc$report_data(c(1, 4), dict[1], wide = TRUE, all = TRUE)

# only get data at the working hour on the first Monday
bc$report_data(c(1, 4), dict[1], hour = 8:18, day_type = "monday", simulation_days = 1:7)

## End(Not run)


## ------------------------------------------------
## Method `BayesCalibJob$eplus_tabular_data`
## ------------------------------------------------

## Not run:
# read all tabular data
bc$eplus_tabular_data(c(1, 4))

# explicitly specify data you want
str(bc$eplus_tabular_data(c(1, 4),
    report_name = "AnnualBuildingUtilityPerformanceSummary",
    table_name = "Site and Source Energy",
    column_name = "Total Energy",
    row_name = "Total Site Energy"
))

## End(Not run)


## ------------------------------------------------
## Method `BayesCalibJob$eplus_save`
## ------------------------------------------------

## Not run:
# save all parametric models with each model in a separate folder
bc$save(tempdir())

# save all parametric models with all models in the same folder
bc$save(tempdir(), separate = FALSE)

## End(Not run)


## ------------------------------------------------
```

```
## Method `BayesCalibJob$stan_run`
## ------------------------------------------------

## Not run:
bc$stan_run()

## End(Not run)


## ------------------------------------------------
## Method `BayesCalibJob$stan_file`
## ------------------------------------------------

## Not run:
bc$stan_file()

## End(Not run)


## ------------------------------------------------
## Method `BayesCalibJob$post_dist`
## ------------------------------------------------

## Not run:
bc$post_dist()

## End(Not run)


## ------------------------------------------------
## Method `BayesCalibJob$prediction`
## ------------------------------------------------

## Not run:
bc$prediction()

## End(Not run)


## ------------------------------------------------
## Method `BayesCalibJob$evaluate`
## ------------------------------------------------

## Not run:
bc$evaluate()

## End(Not run)
```

---

bayes_job *Create a Bayesian Calibration Job*

Description

`bayes_job()` takes an IDF and EPW as input, and returns an `BayesCalibJob` object for conducting Bayesian calibration on an EnergyPlus model. For more details, please see BayesCalibJob.

Usage

```
bayes_job(idf, epw)
```

Arguments

`idf`	A path to an local EnergyPlus IDF file or an `Idf` object.
`epw`	A path to an local EnergyPlus EPW file or an `Epw` object.

Value

An `BayesCalibJob` object.

Author(s)

Hongyuan Jia

See Also

`sensi_job()` for creating a sensitivity analysis job.

Examples

```
## Not run:
if (eplusr::is_avail_eplus(8.8)) {
    idf_name <- "1ZoneUncontrolled.idf"
    epw_name <-  "USA_CA_San.Francisco.Intl.AP.724940_TMY3.epw"

    idf_path <- file.path(eplusr::eplus_config(8.8)$dir, "ExampleFiles", idf_name)
    epw_path <- file.path(eplusr::eplus_config(8.8)$dir, "WeatherData", epw_name)

    # create from local files
    bayes_job(idf_path, epw_path)

    # create from an Idf and an Epw object
    bayes_job(read_idf(idf_path), read_epw(epw_path))
}

## End(Not run)
```

choice_space *Specify optimizatino parameter of character type*

Description

Specify optimizatino parameter of character type

Usage

```
choice_space(choices, init = choices[1])
```

Arguments

choices	A character vector of choices
init	Initial value. Currently not used.

Value

A ChoiceSpace object

Examples

```
choice_space(c("Roughness", "Smooth"))
```

float_space *Specify optimizatino parameter of float type*

Description

Specify optimizatino parameter of float type

Usage

```
float_space(min, max, init = mean(c(min, max)))
```

Arguments

min	Minimum value
max	Maximum value
init	Initial value. Currently not used.

Value

A FloatSpace object

Examples

```
float_space(1.0, 5.0)
```

GAOptimJob *Conduct Multi-Objective Optimization on An EnergyPlus Model*

Description

GAOptimJob class provides a prototype of conducting single- or multi- objective(s) optimizations on an EnergyPlus model using Genetic Algorithm

Details

The basic workflow is basically:

Super classes

eplusr::EplusGroupJob -> eplusr::ParametricJob -> GAOptimJob

Methods

Public methods:

- GAOptimJob$new()
- GAOptimJob$param()
- GAOptimJob$apply_measure()
- GAOptimJob$objective()
- GAOptimJob$recombinator()
- GAOptimJob$mutator()
- GAOptimJob$selector()
- GAOptimJob$terminator()
- GAOptimJob$validate()
- GAOptimJob$run()
- GAOptimJob$best_set()
- GAOptimJob$pareto_set()
- GAOptimJob$population()
- GAOptimJob$print()

Method `new()`**:** Create a GAOptimJob object

Usage:

```
GAOptimJob$new(idf, epw)
```

Arguments:

`idf` A path to an local EnergyPlus IDF file or an eplusr::Idf object.

`epw` A path to an local EnergyPlus EPW file or an eplusr::Epw object.

Returns: A GAOptimJob object.

Examples:

```
\dontrun{
if (eplusr::is_avail_eplus(8.8)) {
    idf_name <- "1ZoneUncontrolled.idf"
    epw_name <-  "USA_CA_San.Francisco.Intl.AP.724940_TMY3.epw"

   idf_path <- file.path(eplusr::eplus_config(8.8)$dir, "ExampleFiles", idf_name)
   epw_path <- file.path(eplusr::eplus_config(8.8)$dir, "WeatherData", epw_name)

    # create from local files
    GAOptimJob$new(idf_path, epw_path)

    # create from an Idf and an Epw object
   opt <- GAOptimJob$new(eplusr::read_idf(idf_path), eplusr::read_epw(epw_path))
}
}
```

Method `param()`**:**

Usage:

```
GAOptimJob$param(..., .names = NULL)
```

Method `apply_measure()`**:**

Usage:

```
GAOptimJob$apply_measure(measure, ..., .names = NULL)
```

Method `objective()`**:**

Usage:

```
GAOptimJob$objective(..., .n = NULL, .dir = "min")
```

Method `recombinator()`**:**

Usage:

```
GAOptimJob$recombinator(
  ...,
  .float = setwith(ecr::recSBX, eta = 15, p = 0.7),
  .integer = setwith(recPCrossover, p = 0.7),
  .choice = setwith(recPCrossover, p = 0.7)
)
```

Method `mutator()`**:**

Usage:

```
GAOptimJob$mutator(
  ...,
  .float = setwith(ecr::mutPolynomial, eta = 25, p = 0.1),
  .integer = mutRandomChoice,
  .choice = mutRandomChoice
)
```

Method `selector()`**:**

Usage:

```
GAOptimJob$selector(
  parent = ecr::selSimple,
  survival = ecr::selNondom,
  strategy = "plus"
)
```

Method `terminator()`**:**

Usage:

```
GAOptimJob$terminator(
  fun = NULL,
  name,
  message,
  max_gen = NULL,
  max_eval = NULL,
  max_time = NULL
)
```

Method `validate()`**:**

Usage:

```
GAOptimJob$validate(param = NULL, ddy_only = TRUE, verbose = TRUE)
```

Method `run()`**:**

Usage:

```
GAOptimJob$run(
  mu = 20L,
  p_recomb = 0.7,
  p_mut = 0.1,
  dir = NULL,
  wait = TRUE,
  parallel = TRUE
)
```

Method `best_set()`**:**

Usage:

```
GAOptimJob$best_set(unique = TRUE)
```

Method `pareto_set()`**:**

Usage:

```
GAOptimJob$pareto_set(unique = TRUE)
```

Method `population()`**:**

Usage:

```
GAOptimJob$population()
```

Method `print()`**:**

Usage:

```
GAOptimJob$print()
```

Author(s)

Hongyuan Jia

Examples

```
## ------------------------------------------------
## Method `GAOptimJob$new`
## ------------------------------------------------

## Not run:
if (eplusr::is_avail_eplus(8.8)) {
    idf_name <- "1ZoneUncontrolled.idf"
    epw_name <-  "USA_CA_San.Francisco.Intl.AP.724940_TMY3.epw"

    idf_path <- file.path(eplusr::eplus_config(8.8)$dir, "ExampleFiles", idf_name)
    epw_path <- file.path(eplusr::eplus_config(8.8)$dir, "WeatherData", epw_name)

    # create from local files
    GAOptimJob$new(idf_path, epw_path)

    # create from an Idf and an Epw object
    opt <- GAOptimJob$new(eplusr::read_idf(idf_path), eplusr::read_epw(epw_path))
}

## End(Not run)
```

---

gaoptim_job *Create an Optimization Job*

Description

`gaoptim_job()` takes an IDF and EPW as input, and returns an `GAOptimJob` object for conducting optimization on an EnergyPlus model. For more details, please see GAOptimJob.

Usage

```
gaoptim_job(idf, epw)
```

Arguments

`idf`	A path to an local EnergyPlus IDF file or an `Idf` object.
`epw`	A path to an local EnergyPlus EPW file or an `Epw` object.

Value

A `GAOptimJob` object.

Author(s)

Hongyuan Jia

See Also

`sensi_job()` for creating a sensitivity analysis job.

Examples

```
## Not run:
if (eplusr::is_avail_eplus(8.8)) {
    idf_name <- "1ZoneUncontrolled.idf"
    epw_name <-  "USA_CA_San.Francisco.Intl.AP.724940_TMY3.epw"

    idf_path <- file.path(eplusr::eplus_config(8.8)$dir, "ExampleFiles", idf_name)
    epw_path <- file.path(eplusr::eplus_config(8.8)$dir, "WeatherData", epw_name)

    # create from local files
    gaoptim_job(idf_path, epw_path)

    # create from an Idf and an Epw object
    gaoptim_job(read_idf(idf_path), read_epw(epw_path))
}

## End(Not run)
```

---

integer_space *Specify optimizatino parameter of integer type*

Description

Specify optimizatino parameter of integer type

Usage

```
integer_space(integers, init = integers[1])
```

Arguments

`integers`	An integer vector.
`init`	Initial value. Currently not used.

Value

A `IntegerSpace` object.

Examples

```
integer_space(1:5)
```

mutRandomChoice *Random Choice Mutator*

Description

"Random Choice" mutation operator for discrete parameters: with probability p chooses one of the available categories at random (this *may* be the original value!)

Usage

```
mutRandomChoice(ind, values, p = 0.1)
```

Arguments

ind	[character] individual to mutate.
values	[list of character] set of possible values for ind entries to take. May be a list of length 1, in which case it is recycled.
p	[numeric(1)] per-entry probability to perform mutation.

Value

[character]

print.ChoiceSpace *Print choice parameter*

Description

Print choice parameter

Usage

```
## S3 method for class 'ChoiceSpace'
print(x, ...)
```

Arguments

x	A ChoiceRange object
...	Further arguments passed to or from other methods.

print.FloatRange *Print float parameter*

Description

Print float parameter

Usage

```
## S3 method for class 'FloatRange'
print(x, ...)
```

Arguments

x	A FloatRange object
...	Further arguments passed to or from other methods.

print.IntegerRange *Print integer parameter*

Description

Print integer parameter

Usage

```
## S3 method for class 'IntegerRange'
print(x, ...)
```

Arguments

x	An IntegerRange object
...	Further arguments passed to or from other methods.

recPCrossover *General Uniform Crossover*

Description

Crossover recombination operator that crosses over each position iid with prob. p and can also be used for non-binary operators.

Usage

```
recPCrossover(inds, p = 0.1, ...)
```

Arguments

inds	[list of any] list of two individuals to perform uniform crossover on
p	[numeric(1)] per-entry probability to perform crossover.
...	further arguments passed on to the method.

Value

[list of any] The mutated individuals.

SensitivityJob *Conduct Sensitivity Analysis for An EnergyPlus Model*

Description

sensi_job() takes an IDF and EPW as input and returns a SensitivityJob, which provides a prototype of conducting sensitivity analysis of EnergyPlus simulations using Morris method.

Details

SensitivityJob inherits from eplusr::ParametricJob class, which means that all methods provided by eplusr::ParametricJob class are also available for SensitivityJob class.

The basic workflow is basically:

1. Adding parameters for sensitivity analysis using $param() or $apply_measure().
2. Check parameter sampled values and generated parametric models using $samples() and $models(), respectively.
3. Run EnergyPlus simulations in parallel using $run(),
4. Gather EnergyPlus simulated data using $report_data() or $tabular_data().
5. Evaluate parameter sensitivity using $evaluate().

Super classes

eplusr::EplusGroupJob -> eplusr::ParametricJob -> SensitivityJob

Methods

Public methods:

- SensitivityJob$param()
- SensitivityJob$apply_measure()
- SensitivityJob$samples()
- SensitivityJob$evaluate()
- SensitivityJob$print()

Method `param()`**:** Set parameters for sensitivity analysis

Usage:

```
SensitivityJob$param(
  ...,
  .names = NULL,
  .r = 12L,
  .grid_jump = 4L,
  .scale = TRUE
)
```

Arguments:

`...` Lists of paramter definitions. Please see above on the syntax.

`.names` A character vector of the parameter names. If `NULL`, the parameter will be named in format `theta` + `number`, where `number` is the index of parameter. Default: `NULL`.

`.r` An positive integer specifying the number of elementary effect computed per factor. For details, see sensitivity::morris. Default: `12`.

`.grid_jump` An integer or a vector of integers specifying the number of levels that are increased/decreased for computing the elementary effects. Default: `1L`. For details, see sensitivity::morris.

`.scale` If `TRUE`, the input design of experiments is scaled after building the design and before computing the elementary effects so that all factors vary within the range [0,1]. Default: `TRUE`. For details, see sensitivity::morris.

Details: `$param()` takes parameter definitions in list format, which is similar to `$set()` in eplusr::Idf class except that each field is not assigned with a single value, but a numeric vector of length 3, indicating the minimum value, maximum value and number of levels of each parameter.
Similar like the way of modifying object field values in eplusr::Idf$set(), there are 3 different ways of defining a parameter in epluspar:

- `object = list(field = c(min, max, levels))`: Where `object` is a valid object ID or name. Note object ID should be denoted with two periods `..`, e.g. `..10` indicates the object with ID `10`, It will set that specific field in that object as one parameter.
- `.(object, object) := list(field = c(min, max, levels))`: Simimar like above, but note the use of `.()` in the left hand side. You can put multiple object ID or names in `.()`. It will set the field of all specified objects as one parameter.

- `class := list(field = c(min, max, levels))`: Note the use of `:=` instead of `=`. The main difference is that, unlike `=`, the left hand side of `:=` should be a valid class name in current eplusr::Idf. It will set that field of all objects in specified class as one parameter.

For example, the code block below defines 3 parameters:

- Field `Fan Total Efficiency` in object named `Supply Fan 1` in class `Fan:VariableVolume` class, with minimum, maximum and number of levels being 0.1, 1.0 and 5, respectively.
- Field `Thickness` in all objects in class `Material`, with minimum, maximum and number of levels being 0.01, 1.0 and 5, respectively.
- Field `Conductivity` in all objects in class `Material`, with minimum, maximum and number of levels being 0.1, 0.6 and 10, respectively.

```
sensi$param(
  `Supply Fan 1` = list(Fan_Total_Efficiency = c(min = 0.1, max = 1.0, levels = 5)),
  Material := list(Thickness = c(0.01, 1, 5), Conductivity = c(0.1, 0.6, 10))
)
```

Returns: The modified `SensitivityJob` object itself.

Examples:

```
\dontrun{
sensi$param(
  `Supply Fan 1` = list(Fan_Total_Efficiency = c(min = 0.1, max = 1.0, levels = 5)),
  Material := list(Thickness = c(0.01, 1, 5), Conductivity = c(0.1, 0.6, 10))
)
}
```

Method `apply_measure()`**:** Set parameters for sensitivity analysis using function

Usage:

```
SensitivityJob$apply_measure(
  measure,
  ...,
  .r = 12L,
  .grid_jump = 4L,
  .scale = TRUE
)
```

Arguments:

`measure` A function that takes an eplusr::Idf and other arguments as input and returns an eplusr::Idf object as output.

`...` Arguments **except first** `Idf` **argument** that are passed to that `measure`.

`.r` An positive integer specifying the number of elementary effect computed per factor. For details, see sensitivity::morris.

`.grid_jump` An integer or a vector of integers specifying the number of levels that are increased/decreased for computing the elementary effects. For details, see sensitivity::morris.

`.scale` If `TRUE`, the input design of experiments is scaled after building the design and before computing the elementary effects so that all factors vary within the range [0,1]. Default: `TRUE`. For details, see sensitivity::morris.

Details: `$apply_measure()` works in a similar way as the `$apply_measure` in eplusr::ParametricJob class, with only exception that each argument supplied in `...` should be a numeric vector of length 3, indicating the minimum, maximum and number of levels of each parameter.
Basically `$apply_measure()` allows to apply a measure to an eplusr::Idf. A measure here is just a function that takes an eplusr::Idf object and other arguments as input, and returns a modified eplusr::Idf object as output.
The names of function parameter will be used as the names of sensitivity parameter. For example, the equivalent version of specifying parameters described in `$param()` using `$apply_measure()` can be:

```
# set sensitivity parameters using $apply_measure()
# (a) first define a "measure"
measure <- function (idf, efficiency, thickness, conducitivy) {
    idf$set(
        `Supply Fan 1` = list(Fan_Total_Efficiency = efficiency),
        Material := list(Thickness = thickness, Conductivity = conducivity)
    )
    idf
}
# (b) then apply that measure with parameter space definitions as
# function arguments
sensi$apply_measure(measure,
    efficiency = c(min = 0.1, max = 1.0, levels = 5),
    thickness = c(0.01, 1, 5), conductivity = c(0.1, 0.6, 10)
)
```

Returns: The modified `SensitivityJob` object itself.

Examples:

```
\dontrun{
# set sensitivity parameters using $apply_measure()
# (a) first define a "measure"
measure <- function (idf, efficiency, thickness, conducitivy) {
    idf$set(
        `Supply Fan 1` = list(Fan_Total_Efficiency = efficiency),
        Material := list(Thickness = thickness, Conductivity = conducivity)
    )
    idf
}
# (b) then apply that measure with parameter space definitions as
# function arguments
sensi$apply_measure(measure,
    efficiency = c(min = 0.1, max = 1.0, levels = 5),
    thickness = c(0.01, 1, 5), conductivity = c(0.1, 0.6, 10)
)
}
```

Method `samples()`**:** Get sampled parameter values

Usage:

```
SensitivityJob$samples()
```

Details: `$samples()` returns a `data.table::data.table()` which contains the sampled value for each parameter using Morris method. The returned data.table has `1 + n` columns, where `n` is the parameter number, while `1` indicates an extra column named `case` giving the index of each sample.

Returns: A `data.table::data.table()`.

Examples:

```
\dontrun{
sensi$samples()
}
```

Method `evaluate()`**:** Evaluate sensitivity

Usage:

```
SensitivityJob$evaluate(results)
```

Arguments:

`results` A numeric vector. Usually the output of parametric simulations extracted using $report_data() or $tabular_data().

Details: `$evaluate()` takes a numeric vector with the same length as total sample number and returns the a `sensitivity::morris()` object. The statistics of interest (mu, mu* and sigma) are stored as an attribute named `data` and can be retrieved using `atrr(sensi$evaluate(), "data")`.

Returns: a `sensitivity::morris()` object with an extra `data` attribute.

Examples:

```
\dontrun{
# run parametric simulations
sensi$run(wait = TRUE)

# status now includes a data.table with detailed information on each simulation
sensi$status()

# print simulation errors
sensi$errors()

# extract a target simulation output value for each case to evaluate the
# sensitivity results
eng <- sen$tabular_data(table_name = "site and source energy",
    column_name = "energy per total building area",
    row_name = "total site energy")[, as.numeric(value)]
(result <- sensi$evaluate(eng))

# extract sensivitity data
attr(result, "data")

# plot
```

```
plot(result)
}
```

Method `print()`**:** Print `SensitivityJob` object

Usage:

```
SensitivityJob$print()
```

Details: `$print()` shows the core information of this `SensitivityJob`, including the path of IDFs and EPWs and also the simulation job status.
`$print()` is quite useful to get the simulation status, especially when `wait` is `FALSE` in `$run()`. The job status will be updated and printed whenever `$print()` is called.

Returns: The `SensitivityJob` object itself, invisibly.

Examples:

```
\dontrun{
sen$print()
}
```

Author(s)

Hongyuan Jia

Examples

```
## ------------------------------------------------
## Method `SensitivityJob$param`
## ------------------------------------------------

## Not run:
sensi$param(
    `Supply Fan 1` = list(Fan_Total_Efficiency = c(min = 0.1, max = 1.0, levels = 5)),
    Material := list(Thickness = c(0.01, 1, 5), Conductivity = c(0.1, 0.6, 10))
)

## End(Not run)


## ------------------------------------------------
## Method `SensitivityJob$apply_measure`
## ------------------------------------------------

## Not run:
# set sensitivity parameters using $apply_measure()
# (a) first define a "measure"
measure <- function (idf, efficiency, thickness, conducitivy) {
    idf$set(
        `Supply Fan 1` = list(Fan_Total_Efficiency = efficiency),
        Material := list(Thickness = thickness, Conductivity = conducivity)
    )
    idf
```

```
}
# (b) then apply that measure with parameter space definitions as
# function arguments
sensi$apply_measure(measure,
    efficiency = c(min = 0.1, max = 1.0, levels = 5),
    thickness = c(0.01, 1, 5), conductivity = c(0.1, 0.6, 10)
)

## End(Not run)


## ------------------------------------------------
## Method `SensitivityJob$samples`
## ------------------------------------------------

## Not run:
sensi$samples()

## End(Not run)


## ------------------------------------------------
## Method `SensitivityJob$evaluate`
## ------------------------------------------------

## Not run:
# run parametric simulations
sensi$run(wait = TRUE)

# status now includes a data.table with detailed information on each simulation
sensi$status()

# print simulation errors
sensi$errors()

# extract a target simulation output value for each case to evaluate the
# sensitivity results
eng <- sen$tabular_data(table_name = "site and source energy",
    column_name = "energy per total building area",
    row_name = "total site energy")[, as.numeric(value)]
(result <- sensi$evaluate(eng))

# extract sensivitity data
attr(result, "data")

# plot
plot(result)

## End(Not run)


## ------------------------------------------------
## Method `SensitivityJob$print`
```

```
## ------------------------------------------------

## Not run:
sen$print()

## End(Not run)
```

`sensi_job` *Create a Sensitivity Analysis Job*

Description

`sensi_job()` takes an IDF and EPW as input, and returns an `SensitivityJob` object for conducting sensitivity analysis on an EnergyPlus model. For more details, please see SensitivityJob.

Usage

```
sensi_job(idf, epw)
```

Arguments

`idf`	A path to an local EnergyPlus IDF file or an `Idf` object.
`epw`	A path to an local EnergyPlus EPW file or an `Epw` object.

Value

An `SensitivityJob` object.

Author(s)

Hongyuan Jia

See Also

`bayes_job()` for creating a Bayesian calibration job.

Examples

```
## Not run:
if (eplusr::is_avail_eplus(8.8)) {
    idf_name <- "1ZoneUncontrolled.idf"
    epw_name <-  "USA_CA_San.Francisco.Intl.AP.724940_TMY3.epw"

    idf_path <- file.path(eplusr::eplus_config(8.8)$dir, "ExampleFiles", idf_name)
    epw_path <- file.path(eplusr::eplus_config(8.8)$dir, "WeatherData", epw_name)

    # create from local files
    sensi_job(idf_path, epw_path)
```

```
    # create from an Idf and an Epw object
    sensi_job(read_idf(idf_path), read_epw(epw_path))
}

## End(Not run)
```

`setwith` *Partial apply a operator, filling in some arguments.*

Description

`setwith()` allows you to modify an operator by pre-filling some of the arguments.

Usage

```
setwith(fun, ...)
```

Arguments

`fun`	An `ecr_operator` object
`...`	Named arguments to `fun` that should be partially applied.

Value

An `ecr_operator_setwith` object

`stopOnMaxTime` *Stopping on Maximum Time of Evaluations*

Description

Stop the EA after a given cutoff time.

Usage

```
stopOnMaxTime(max.time = NULL)
```

Arguments

`max.time`	Time limit in seconds. Default: `NULL`.

Value

An `ecr_terminator` object

Index