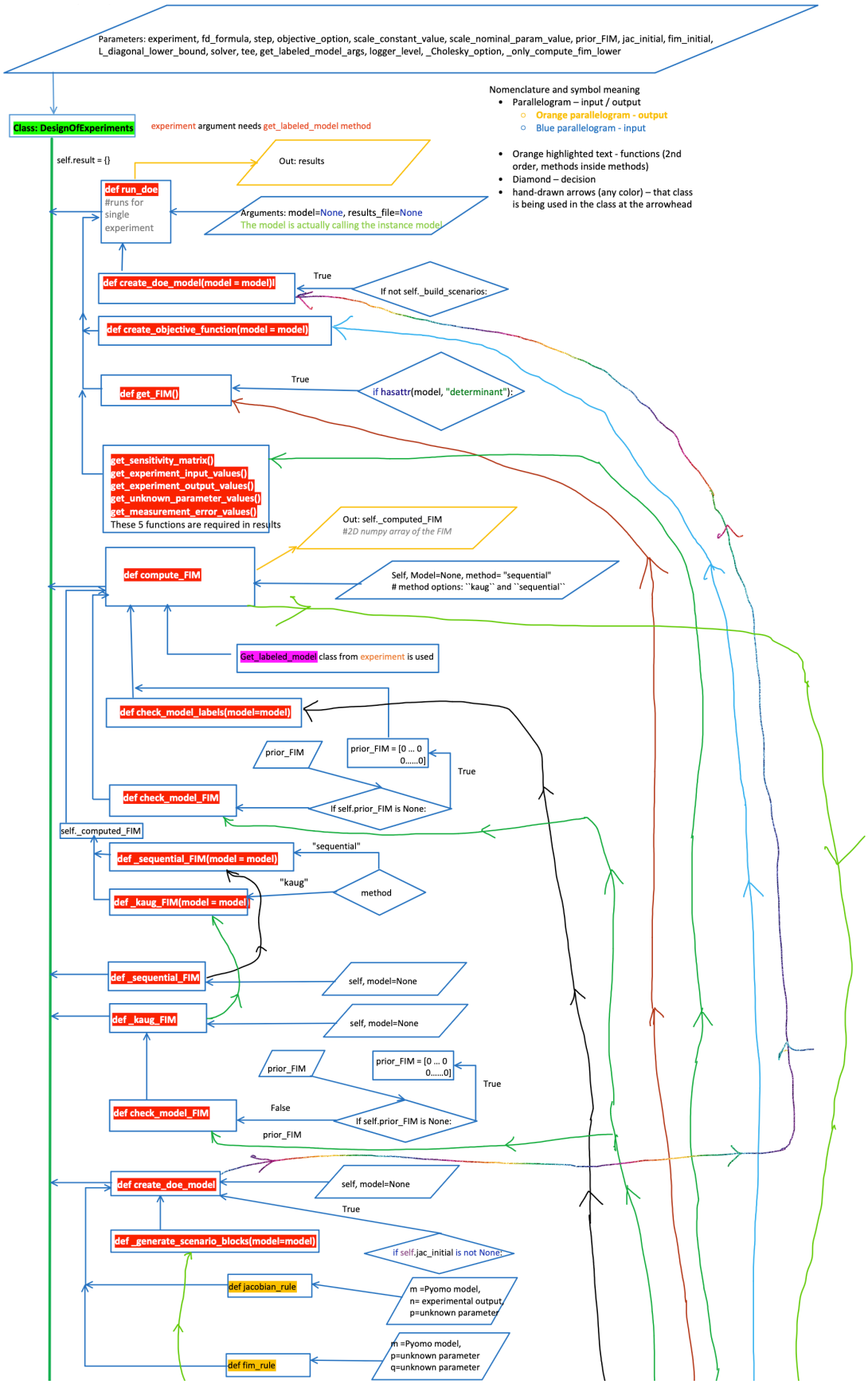
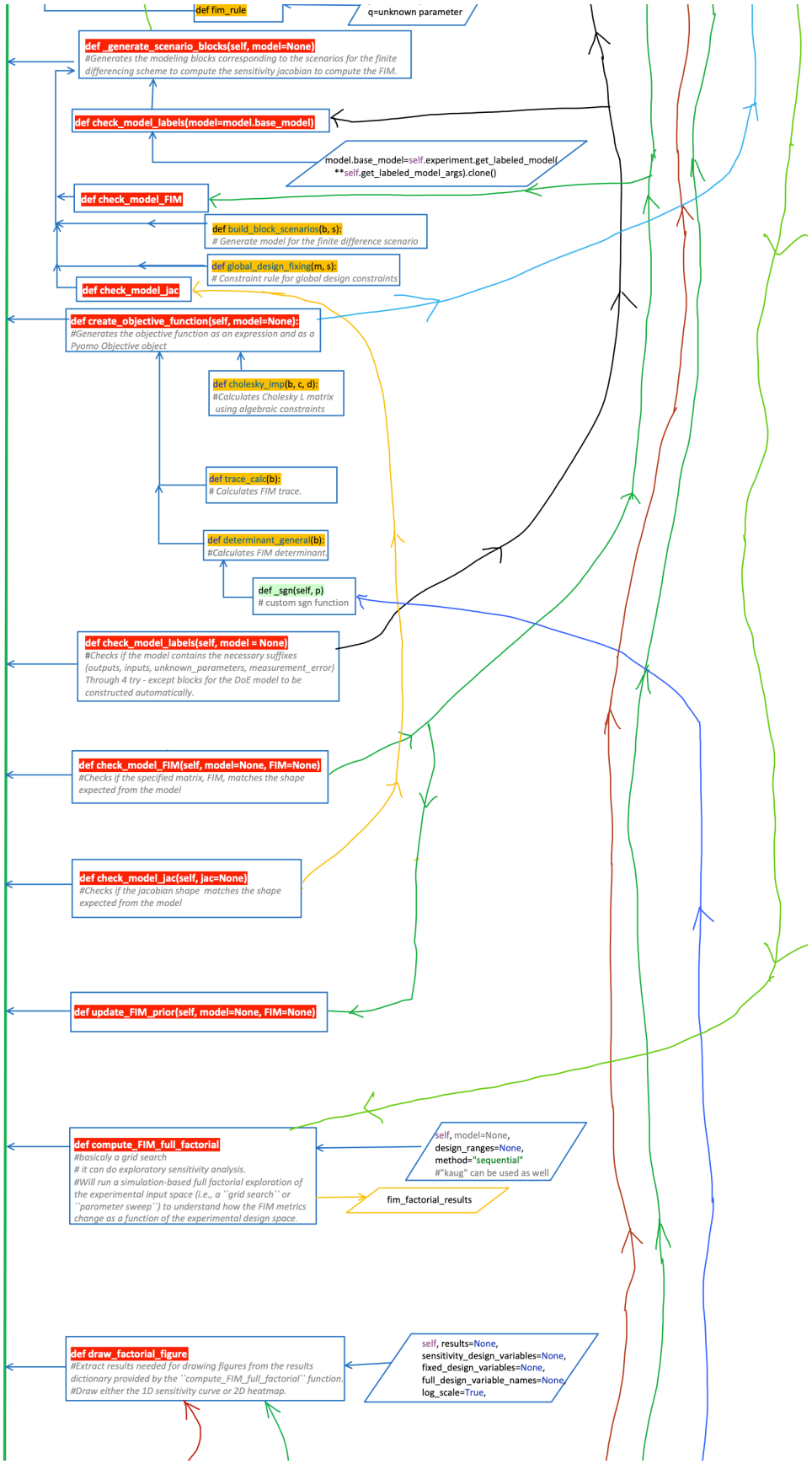
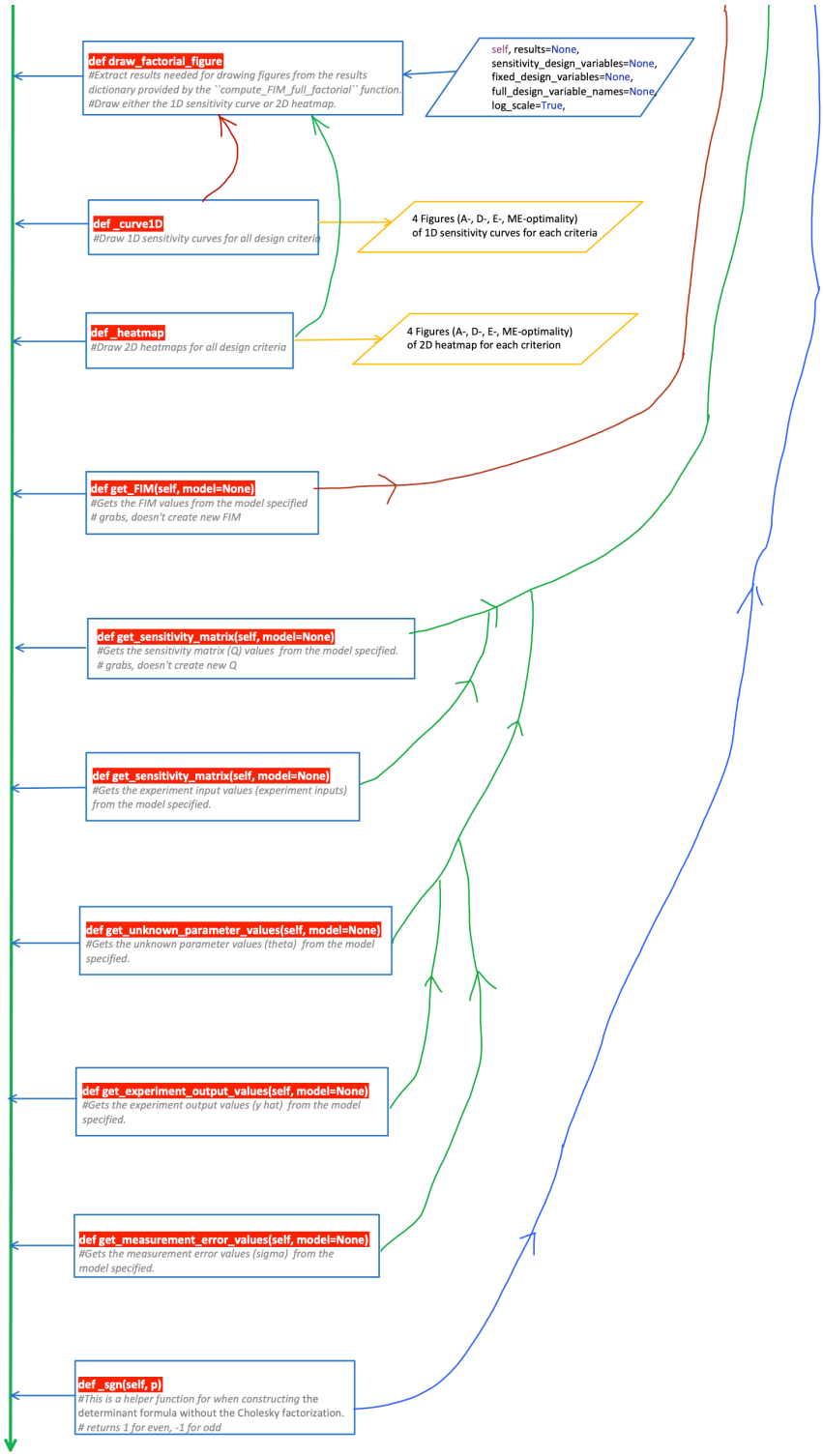

Parameters: experiment, fd_formula, step, objective_option, scale_constant_value, scale_nominal_param_value, prior_FIM, jac_initial, fim_initial, L_diagonal_lower_bound, solver, tee, get_labeled_model_args, logger_level, _Cholesky_option, _only_compute_fim_lower
Nomenclature and symbol meaning
Parallelogram – input / output
Orange parallelogram - output
Blue parallelogram - input
Orange highlighted text - functions (2nd order, methods inside methods)
Diamond – decision
hand-drawn arrows (any color) – that class is being used in the class at the arrowhead
Class: DesignOfExperiments
experiment argument needs get_labeled_model method
self.result = {}
Out: results
def run_doe
#runs for single experiment
Arguments: model=None, results_file=None
The model is actually calling the instance model
def create_doe_model(model = model)
True
If not self._build_scenarios:
def create_objective_function(model = model)
def get_FIM()
True
if hasattr(model, "determinant"):
get_sensitivity_matrix()
get_experiment_input_values()
get_experiment_output_values()
get_unknown_parameter_values()
get_measurement_error_values()
These 5 functions are required in results
Out: self._computed_FIM
#2D numpy array of the FIM
def compute_FIM
Self, Model=None, method= "sequential"
method options: ``kaug`` and ``sequential``
Get_labeled_model class from experiment is used
def check_model_labels(model=model)
prior_FIM
prior_FIM = [0 ... 0 0.....0]
True
def check_model_FIM
If self.prior_FIM is None:
self._computed_FIM
def _sequential_FIM(model = model)
"sequential"
"kaug"
method
def _kaug_FIM(model = model)
def _sequential_FIM
self, model=None
def _kaug_FIM
self, model=None
prior_FIM
prior_FIM = [0 ... 0 0.....0]
True
def check_model_FIM
False
If self.prior_FIM is None:
prior_FIM
def create_doe_model
self, model=None
True
def _generate_scenario_blocks(model=model)
if self.jac_initial is not None:
def jacobian_rule
m =Pyomo model,
n= experimental output,
p=unknown parameter
def fim_rule
m =Pyomo model,
p=unknown parameter
q=unknown parameter


def fim_rule
q=unknown parameter
def _generate_scenario_blocks(self, model=None)
#Generates the modeling blocks corresponding to the scenarios for the finite differencing scheme to compute the sensitivity jacobian to compute the FIM.
def check_model_labels(model=model.base_model)
model.base_model=self.experiment.get_labeled_model(**self.get_labeled_model_args).clone()
def check_model_FIM
def build_block_scenarios(b, s):
# Generate model for the finite difference scenario
def global_design_fixing(m, s):
# Constraint rule for global design constraints
def check_model_jac
def create_objective_function(self, model=None):
#Generates the objective function as an expression and as a Pyomo Objective object
def cholesky_imp(b, c, d):
#Calculates Cholesky L matrix using algebraic constraints
def trace_calc(b):
# Calculates FIM trace.
def determinant_general(b):
#Calculates FIM determinant.
def _sgn(self, p)
custom sgn function
def check_model_labels(self, model = None)
#Checks if the model contains the necessary suffixes (outputs, inputs, unknown_parameters, measurement_error) Through 4 try - except blocks for the DoE model to be constructed automatically.
def check_model_FIM(self, model=None, FIM=None)
#Checks if the specified matrix, FIM, matches the shape expected from the model
def check_model_jac(self, jac=None)
#Checks if the jacobian shape matches the shape expected from the model
def update_FIM_prior(self, model=None, FIM=None)
def compute_FIM_full_factorial
#basicaly a grid search
it can do exploratory sensitivity analysis.
#Will run a simulation-based full factorial exploration of the experimental input space (i.e., a ``grid search`` or ``parameter sweep``) to understand how the FIM metrics change as a function of the experimental design space.
self, model=None,
design_ranges=None,
method="sequential"
#"kaug" can be used as well
fim_factorial_results
def draw_factorial_figure
#Extract results needed for drawing figures from the results dictionary provided by the ``compute_FIM_full_factorial`` function.
#Draw either the 1D sensitivity curve or 2D heatmap.
self, results=None,
sensitivity_design_variables=None,
fixed_design_variables=None,
full_design_variable_names=None,
log_scale=True,


def draw_factorial_figure
#Extract results needed for drawing figures from the results dictionary provided by the ``compute_FIM_full_factorial`` function.
#Draw either the 1D sensitivity curve or 2D heatmap.
self, results=None,
sensitivity_design_variables=None,
fixed_design_variables=None,
full_design_variable_names=None,
log_scale=True,
def _curve1D
#Draw 1D sensitivity curves for all design criteria
4 Figures (A-, D-, E-, ME-optimality)
of 1D sensitivity curves for each criteria
def _heatmap
#Draw 2D heatmaps for all design criteria
4 Figures (A-, D-, E-, ME-optimality)
of 2D heatmap for each criterion
def get_FIM(self, model=None)
#Gets the FIM values from the model specified
grabs, doesn't create new FIM
def get_sensitivity_matrix(self, model=None)
#Gets the sensitivity matrix (Q) values from the model specified.
grabs, doesn't create new Q
def get_sensitivity_matrix(self, model=None)
#Gets the experiment input values (experiment inputs) from the model specified.
def get_unknown_parameter_values(self, model=None)
#Gets the unknown parameter values (theta) from the model specified.
def get_experiment_output_values(self, model=None)
#Gets the experiment output values (y hat) from the model specified.
def get_measurement_error_values(self, model=None)
#Gets the measurement error values (sigma) from the model specified.
def _sgn(self, p)
#This is a helper function for when constructing the determinant formula without the Cholesky factorization.
returns 1 for even, -1 for odd