Identifying Feasible Regions in Highly Constrained Simulation-Based Optimization Problems

Mentors: Joerg Gablonsky and Jeff Poskin

Optimization problems in industry may involve objectives and constraints determined by computationally expensive analysis codes. One of the major challenges in solving optimization problems in these settings is minimizing the total number of expensive function evaluations. This is particularly true in highly-constrained settings where many expensive evaluations may be immediately discarded by the optimizer as they fail to satisfy the constraints of the problem. This project will explore different ways to efficiently identify feasible regions in constrained optimization problems. Students will be encouraged to explore algorithms that leverage surrogate models such as Gaussian Processes, Neural Networks, and Radial Basis Functions. Model-free algorithms which work directly from an evaluated set of points may also be considered.

Students will participate in training session on best software development practices from experienced Boeing personnel. Project mentors will additionally perform one or more code review sessions to support code development.

Problem setup

We will introduce some classes and helper routines that allow us to provide examples of highly constrained problems and explore the difficulty in finding feasible points (or sites) for these examples. We are interspersing explanations of the code we provide.

In [ ]:
```
from typing import Tuple, Union, List, Type
from copy import deepcopy
import pandas as pd
import numpy as np
from numpy import number
from scipy.stats.qmc import LatinHypercube as lhs
from scipy.stats.qmc import scale
```

We define a helper class `CallableClass` that we use for typing.

In [ ]:
```
class CallableClass:
    def __call__(self, *args, **kwargs):
        pass
```

We now define two example problem from the well known suite of test problems defined by Hock and Schittkowski. For each of them we define a class with methods and properties that:

- Define the name of the example
- Returns the defined optimization problem as a Python dictionary
- Provide the known solution and a standard starting point for optimization as Pandas DataFrames
- Define a callable method that evaluates a set of points / sites defined in a Pandas DataFrame and return the modified DataFrame with the values of all responses in the test problem

HS 100 Description

The Hock-Schittkowski 100 problem is the following optimization problem.

$$\begin{aligned}
\min\quad & (x_1 - 10)^2 + 5(x_2 - 12)^2 + x_3^4 + 3(x_4 - 11)^2 \\
& + 10x_5^6 + 7x_6^2 + x_7^4 - 4x_6x_7 - 10x_6 - 8x_7 \\[1em]
\text{s.t.}\quad & 2x_1^2 + 3x_2^4 + x_3 + 4x_4^2 + 5x_5 \leq 127 \\
& 7x_1 + 3x_2 + 10x_3^2 + x_4 - x_5 \leq 282 \\
& 23x_1 + x_2^2 + 6x_6 - 8x_7 \leq 196 \\
& 4x_1^2 + x_2^2 - 3x_1x_2 + 2x_3^2 + 5x_6 - 11x_7 \geq 0
\end{aligned}$$

The following bounds are placed on the variables:

$$-10 \leq x_i \leq 10.075 \qquad i = 1, \ldots, 7$$

This has a known solution of

$$f(2.330499, 1.951372, -0.4775414, 4.365726, -0.6244870, 1.038131, 1.594227) = 680.630$$

An initial guess is also defined for this class as

$$f(1, 2, 0, 4, 0, 1, 1) = 714$$

In [ ]:

```
class HS100(CallableClass):
    r""".. math::
        \begin{align}
            \min\quad & (x_1 - 10)^2 + 5(x_2 - 12)^2 + x_3^4 + 3(x_4 - 11)^2\\
                        & + 10x_5^6 + 7x_6^2 + x_7^4 - 4x_6x_7 - 10x_6 - 8x_7\\[1em
            \text{s.t.}\quad & 2x_1^2 + 3x_2^4 + x_3 + 4x_4^2 + 5x_5 \leq 127\\
            & 7x_1 + 3x_2 + 10x_3^2 + x_4 - x_5 \leq 282\\
            & 23x_1 + x_2^2 + 6x_6 - 8x_7 \leq 196\\
            & 4x_1^2 + x_2^2 - 3x_1x_2 + 2x_3^2 + 5x_6 - 11x_7 \geq 0
        \end{align}

    The following bounds are placed on the variables:

    .. math::
        -10 \leq x_i \leq 10.075 \qquad i = 1,...,7

    This has a known solution of
    .. math::
        f(2.330499, 1.951372, -0.4775414, 4.365726, -0.6244870, 1.038131, 1.594227)

    An initial guess is also defined for this class as

    .. math::
        f(1, 2, 0, 4, 0, 1, 1) = 714
    """

    name = 'hs100'

    def problem(self) -> dict:
        """Initialize the test problem"""
        return {
            "variables": {
                f"x{i+1}": {
                    "type": "float",
                    "bounds": [-10, 10.075],
                    "shift": 0,
                    "scale": 0.1,
                }
                for i in range(7)
            },
            "responses": {
                "f": {"type": "float", "shift": 0.0, "scale": 1e-3},
                "c1": {
                    "type": "float",
                    "bounds": [0, np.inf],
                    "shift": 0,
                    "scale": 1e-3,
                },
                "c2": {
                    "type": "float",
                    "bounds": [0, np.inf],
                    "shift": 0,
                    "scale": 1e-2,
                },
                "c3": {
                    "type": "float",
                    "bounds": [0, np.inf],
                    "shift": 0,
                    "scale": 1e-2,
                },
                "c4": {
                    "type": "float",
                    "bounds": [0, np.inf],
                    "shift": 0,
                    "scale": 1e-2,
                },
            },
            "objectives": ["f"],
            "constraints": ["c1", "c2", "c3", "c4"],
        }

    def known_solution(self) -> pd.Series:
        return pd.Series(
            data={
                "x1": 2.330499,
                "x2": 1.951372,
                "x3": -0.4775414,
                "x4": 4.365726,
                "x5": -0.6244870,
                "x6": 1.038131,
                "x7": 1.594227,
            }
        )

    def initial_guess(self) -> pd.Series:
        """Provide an initial guess for an optimizer

        :return: List providing the initial guess to use with an optimizer
        :rtype: list
        """
        return [1, 2, 0, 4, 0, 1, 1]

    def __call__(self, sites: pd.DataFrame) -> None:
        """Call to the HS100 function

        :param df: The dataframe that contains the input values, and is updated wit
        :type df: DataFrame
        """
        sites["f"] = (
            (sites.x1 - 10.0) * (sites.x1 - 10.0)
            + 5.0 * (sites.x2 - 12.0) * (sites.x2 - 12.0)
            + sites.x3 * sites.x3 * sites.x3 * sites.x3
            + 3.0 * (sites.x4 - 11.0) * (sites.x4 - 11.0)
            + 10.0 * sites.x5 * sites.x5 * sites.x5 * sites.x5 * sites.x5 * sites.x
            + 7.0 * sites.x6 * sites.x6
            + sites.x7 * sites.x7 * sites.x7 * sites.x7
            - 4.0 * sites.x6 * sites.x7
            - 10.0 * sites.x6
            - 8.0 * sites.x7
        )
        sites["c1"] = (
            127.0
            - 2.0 * sites.x1 * sites.x1
            - 3.0 * sites.x2 * sites.x2 * sites.x2 * sites.x2
            - sites.x3
            - 4.0 * sites.x4 * sites.x4
            - 5.0 * sites.x5
        )
        sites["c2"] = (
            282.0
            - 7.0 * sites.x1
            - 3.0 * sites.x2
            - 10.0 * sites.x3 * sites.x3
            - sites.x4
            + sites.x5
        )
        sites["c3"] = (
            196.0
            - 23.0 * sites.x1
            - sites.x2 * sites.x2
            - 6.0 * sites.x6 * sites.x6
            + 8.0 * sites.x7
        )
        sites["c4"] = (
            -4.0 * sites.x1 * sites.x1
            - sites.x2 * sites.x2
            + 3.0 * sites.x1 * sites.x2
            - 2.0 * sites.x3 * sites.x3
            - 5.0 * sites.x6
            + 11.0 * sites.x7
        )
```

In [ ]:
```
class HS118(CallableClass):
    """Implement the Hock-Schittkowski number 118 problem

    x0 = ( 20.0, 55.0, 15.0, 20.0, 60.0, 20.0, 20.0, 60.0, 20.0, 20.0, 60.0, 20.0,
    20.0 )

    f(x0) = 942.7162499999998

    x* = ( 8.0, 49.0, 3.0, 1.0, 56.0, 0.0, 1.0, 63.0, 6.0, 3.0, 70.0, 12.0, 5.0, 77

    f(x*) = 664.82045000
    """
    name = 'hs118'

    def problem(self) -> dict:
        """Initialize the test problem"""
        return {
            "constraints": [
                "c1",
                "c2",
                "c3",
                "c4",
                "c5",
                "c6",
                "c7",
                "c8",
                "c9",
                "c10",
                "c11",
                "c12",
                "c13",
                "c14",
                "c15",
                "c16",
                "c17",
            ],
            "objectives": ["f"],
            "responses": {
                "f": {"scale": 0.001, "shift": 0.0, "type": "float"},
                "c1": {
                    "bounds": [0.0, 13.0],
                    "scale": 0.1,
                    "shift": 0.0,
                    "type": "float",
                },
                "c2": {
                    "bounds": [0.0, 13.0],
                    "scale": 0.1,
                    "shift": 0.0,
                    "type": "float",
                },
                "c3": {
                    "bounds": [0.0, 14.0],
                    "scale": 0.1,
                    "shift": 0.0,
                    "type": "float",
                },
                "c4": {
                    "bounds": [0.0, 13.0],
                    "scale": 0.1,
                    "shift": 0.0,
                    "type": "float",
                },
                "c5": {
                    "bounds": [0.0, 13.0],
                    "scale": 0.1,
                    "shift": 0.0,
                    "type": "float",
                },
                "c6": {
                    "bounds": [0.0, 14.0],
                    "scale": 0.1,
                    "shift": 0.0,
                    "type": "float",
                },
                "c7": {
                    "bounds": [0.0, 13.0],
                    "scale": 0.1,
                    "shift": 0.0,
                    "type": "float",
                },
                "c8": {
                    "bounds": [0.0, 13.0],
                    "scale": 0.1,
                    "shift": 0.0,
                    "type": "float",
                },
                "c9": {
                    "bounds": [0.0, 14.0],
                    "scale": 0.1,
                    "shift": 0.0,
                    "type": "float",
                },
                "c10": {
                    "bounds": [0.0, 13.0],
                    "scale": 0.1,
                    "shift": 0.0,
                    "type": "float",
                },
                "c11": {
                    "bounds": [0.0, 13.0],
                    "scale": 0.1,
                    "shift": 0.0,
                    "type": "float",
                },
                "c12": {
                    "bounds": [0.0, 14.0],
                    "scale": 0.1,
                    "shift": 0.0,
                    "type": "float",
                },
                "c13": {
                    "bounds": [0.0, np.inf],
                    "scale": 0.1,
                    "shift": 0.0,
                    "type": "float",
                },
                "c14": {
                    "bounds": [0.0, np.inf],
                    "scale": 0.1,
                    "shift": 0.0,
                    "type": "float",
                },
                "c15": {
                    "bounds": [0.0, np.inf],
                    "scale": 0.1,
                    "shift": 0.0,
                    "type": "float",
                },
                "c16": {
                    "bounds": [0.0, np.inf],
                    "scale": 0.1,
                    "shift": 0.0,
                    "type": "float",
                },
                "c17": {
                    "bounds": [0.0, np.inf],
                    "scale": 0.1,
                    "shift": 0.0,
                    "type": "float",
                },
            },
            "variables": {
                "x1": {
                    "bounds": [8.0, 21.0],
                    "scale": 0.01,
                    "shift": 0.0,
                    "type": "float",
                },
                "x2": {
                    "bounds": [43.0, 57.0],
                    "scale": 0.01,
                    "shift": 0.0,
                    "type": "float",
                },
                "x3": {
                    "bounds": [3.0, 16.0],
                    "scale": 0.01,
                    "shift": 0.0,
                    "type": "float",
                },
                "x4": {
                    "bounds": [0.0, 90.0],
                    "scale": 0.01,
                    "shift": 0.0,
                    "type": "float",
                },
                "x5": {
                    "bounds": [0.0, 120.0],
                    "scale": 0.01,
                    "shift": 0.0,
                    "type": "float",
                },
                "x6": {
                    "bounds": [0.0, 60.0],
                    "scale": 0.01,
                    "shift": 0.0,
                    "type": "float",
                },
                "x7": {
                    "bounds": [0.0, 90.0],
                    "scale": 0.01,
                    "shift": 0.0,
                    "type": "float",
                },
                "x8": {
                    "bounds": [0.0, 120.0],
                    "scale": 0.01,
                    "shift": 0.0,
                    "type": "float",
                },
                "x9": {
                    "bounds": [0.0, 60.0],
                    "scale": 0.01,
                    "shift": 0.0,
                    "type": "float",
                },
                "x10": {
                    "bounds": [0.0, 90.0],
                    "scale": 0.01,
                    "shift": 0.0,
                    "type": "float",
                },
                "x11": {
                    "bounds": [0.0, 120.0],
                    "scale": 0.01,
                    "shift": 0.0,
                    "type": "float",
                },
                "x12": {
                    "bounds": [0.0, 60.0],
                    "scale": 0.01,
                    "shift": 0.0,
                    "type": "float",
                },
                "x13": {
                    "bounds": [0.0, 90.0],
                    "scale": 0.01,
                    "shift": 0.0,
                    "type": "float",
                },
                "x14": {
                    "bounds": [0.0, 120.0],
                    "scale": 0.01,
                    "shift": 0.0,
                    "type": "float",
                },
                "x15": {
                    "bounds": [0.0, 60.0],
                    "scale": 0.01,
                    "shift": 0.0,
                    "type": "float",
                },
            },
        }

    def initial_guess(self) -> list:
        """Provide an initial guess for an optimizer
        :return: List providing the initial guess to use with an optimizer
        :rtype: list
        x0 = ( 20.0, 55.0, 15.0, 20.0, 60.0, 20.0, 20.0, 60.0, 20.0, 20.0, 60.0, 20
               60.0, 20.0 )
        f(x0) = 942.7162499999998
        """
        return [
            20.0,
            55.0,
            15.0,
            20.0,
            60.0,
            20.0,
            20.0,
            60.0,
            20.0,
            20.0,
            60.0,
            20.0,
            20.0,
            60.0,
            20.0,
        ]

    def known_solution(self) -> list:
        """
        Provide the known optimal solution.
        :return: List providing the variable values of the optimal solution of the
        :rtype: list
        x* = (8.0, 49.0, 3.0, 1.0, 56.0, 0.0, 1.0, 63.0, 6.0, 3.0, 70.0, 12.0, 5.0,
        f(x*) = 664.82045000
        """
        return [
            8.0,
            49.0,
            3.0,
            1.0,
            56.0,
            0.0,
            1.0,
            63.0,
            6.0,
            3.0,
            70.0,
            12.0,
            5.0,
            77.0,
            18.0,
        ]

    def __call__(self, sites: pd.DataFrame) -> None:
        """Call to the HS118 function
        :param df: The dataframe that contains the input values, and is updated wit
        :type df: DataFrame
        """
        sites["f"] = (
            2.3 * sites.x1
            + 1.0e-4 * sites.x1 * sites.x1
            + 1.7 * sites.x2
            + 1.0e-4 * sites.x2 * sites.x2
            + 2.2 * sites.x3
            + 1.5e-4 * sites.x3 * sites.x3
            + 2.3 * sites.x4
            + 1.0e-4 * sites.x4 * sites.x4
            + 1.7 * sites.x5
            + 1.0e-4 * sites.x5 * sites.x5
            + 2.2 * sites.x6
            + 1.5e-4 * sites.x6 * sites.x6
            + 2.3 * sites.x7
            + 1.0e-4 * sites.x7 * sites.x7
            + 1.7 * sites.x8
            + 1.0e-4 * sites.x8 * sites.x8
            + 2.2 * sites.x9
            + 1.5e-4 * sites.x9 * sites.x9
            + 2.3 * sites.x10
            + 1.0e-4 * sites.x10 * sites.x10
            + 1.7 * sites.x11
            + 1.0e-4 * sites.x11 * sites.x11
            + 2.2 * sites.x12
            + 1.5e-4 * sites.x12 * sites.x12
            + 2.3 * sites.x13
            + 1.0e-4 * sites.x13 * sites.x13
            + 1.7 * sites.x14
            + 1.0e-4 * sites.x14 * sites.x14
            + 2.2 * sites.x15
            + 1.5e-4 * sites.x15 * sites.x15
        )
        sites["c1"] = sites.x4 - sites.x1 + 7
        sites["c2"] = sites.x6 - sites.x3 + 7
        sites["c3"] = sites.x5 - sites.x2 + 7
        sites["c4"] = sites.x7 - sites.x4 + 7
        sites["c5"] = sites.x9 - sites.x6 + 7
        sites["c6"] = sites.x8 - sites.x5 + 7
        sites["c7"] = sites.x10 - sites.x7 + 7
        sites["c8"] = sites.x12 - sites.x9 + 7
        sites["c9"] = sites.x11 - sites.x8 + 7
        sites["c10"] = sites.x13 - sites.x10 + 7
        sites["c11"] = sites.x15 - sites.x12 + 7
        sites["c12"] = sites.x14 - sites.x11 + 7

        sites["c13"] = sites.x1 + sites.x2 + sites.x3 - 60.0
        sites["c14"] = sites.x4 + sites.x5 + sites.x6 - 50.0
        sites["c15"] = sites.x7 + sites.x8 + sites.x9 - 70.0
        sites["c16"] = sites.x10 + sites.x11 + sites.x12 - 85.0
        sites["c17"] = sites.x13 + sites.x14 + sites.x15 - 100.0
```

Next we define a class that allows us to move between the space in which the example is defined (called design space) and the shifted and scaled optimization space easily. This is used when we calculate constraint violations. Making sure a problem is well shifted and scaled when it is presented to an optimization algorithm is very important.

In [ ]:
```
class ShiftAndScale:
    """Used to get variable and response values into the same order of
    magnitude to yield better optimization results.
    """

    def __init__(self, bound_prob: dict) -> None:
        """Constructor method

        :param bound_prob: Optimization problem defined as dictionary. Used to
            gather information about variables and responses. The shift value
            will be added to the value and scale will be multiplied to it.
            Missing or None values are assumed to be zero or one for shift and
            scale, respectively.
        :type bound_prob: dict
        """
        # Get shift and scale values for variables and responses
        var_shift, var_scale = self._dict_to_series(bound_prob["variables"])
        res_shift, res_scale = self._dict_to_series(bound_prob["responses"])

        # Store values into series to make arithmetic easier
        self.shift = pd.concat((var_shift, res_shift))
        self.scale = pd.concat((var_scale, res_scale))
        self._all = list(self.shift.index)

    def design_to_optimizer_space(
        self, points: pd.DataFrame, cols: List[str] = None, suffix: str = None
    ) -> pd.DataFrame:
        """Perform the shift and scaling on specified columns of provided
        points. Rescaled columns can also have their name appended with an
        optional suffix.

        :param points: Data points containing the points to shift and scale.
        :type points: DataFrame
        :param cols: (Optional) List of columns to perform the rescale on.
        :type cols: list
        :param suffix: (Optional) Value to append at end of column names.
        :type suffix: str
        :return: A dataframe containing the columns the operation was performed
            on with the shifted and scaled values.
        :rtype: DataFrame
        """
        if cols is None:
            cols = self._all

        # Get requested columns and perform shift and scale
        ret = points.loc[:, points.columns.isin(cols)]
        ret += self.shift.loc[self.shift.index.isin(cols)]
        ret *= self.scale.loc[self.scale.index.isin(cols)]

        # Append suffix if given
        if suffix:
            ret.columns += suffix

        # Return result
        return ret

    def optimizer_to_design_space(
        self, points: pd.DataFrame, cols: List[str] = None
    ) -> pd.DataFrame:
        """Perform the transformation from Optimizer Space to Design Space

        :param points: Data points containing the points in Optimizer Space that wi
        :type points: DataFrame
        :param cols: (Optional) List of columns to perform the transformation on.
        :type cols: list
        :return: A dataframe containing the unshifted and unscaled columns, that is
        :rtype: DataFrame
        """
        if cols is None:
            cols = self._all
        # Get requested columns and perform shift and scale
        ret = points.loc[:, points.columns.isin(cols)]
        ret /= self.scale.loc[self.scale.index.isin(cols)]
        ret -= self.shift.loc[self.shift.index.isin(cols)]

        # Return result
        return ret

    def to_optimizer_problem(self, problem: dict) -> dict:
        """Return the optimization problem in Optimizer Space

        This method takes an optimization problem in the main space
        and shifts and scales it
        Parameters
        ----------
        problem : dict
            The problem in the Design Space

        Returns
        -------
        dict
            The optimization problem in optimizer space
        """
        # Create a copy of the problem that is passed in
        optimizer_problem = deepcopy(problem)
        for overall_type in ["variables", "responses"]:
            for element, values in problem[overall_type].items():
                if "bounds" in values:
                    lower = values["bounds"][0]
                    upper = values["bounds"][1]
                    lower = (lower + self.shift[element]) * self.scale[element]
                    upper = (upper + self.shift[element]) * self.scale[element]
                    optimizer_problem[overall_type][element]["bounds"] = [lower, up
                    if "default" in values:
                        optimizer_problem[overall_type][element]["default"] = (
                            optimizer_problem[overall_type][element]["default"]
                            + self.shift[element]
                        ) * self.scale[element]
                optimizer_problem[overall_type][element]["shift"] = 0.0
                optimizer_problem[overall_type][element]["scale"] = 1.0
        return optimizer_problem

    def to_design_space_problem(self, problem: dict) -> dict:
        """
        Return the design space problem in Design Space

        This method takes an optimization problem in the optimizer space
        and reverses the shift and scale operations

        Parameters
        ----------
        problem : dict
            The problem in Optimizer Space

        Returns
        -------
        dict
            The design space problem in design space
        """

        # Create a copy of the problem that is passed in
        design_space_problem = deepcopy(problem)
        for overall_type in ["variables", "responses"]:
            for element, values in problem[overall_type].items():
                if "bounds" in values:
                    lower = values["bounds"][0]
                    upper = values["bounds"][1]
                    lower = lower / self.scale[element] - self.shift[element]
                    upper = upper / self.scale[element] - self.shift[element]
                    design_space_problem[overall_type][element]["bounds"] = [lower,
                    if "default" in values:
                        design_space_problem[overall_type][element]["default"] = \
                            design_space_problem[overall_type][element]["default"]
                                / self.scale[element] - self.shift[element]
                design_space_problem[overall_type][element]["shift"] = self.shift[e
                design_space_problem[overall_type][element]["scale"] = self.scale[e
        return design_space_problem

    def _dict_to_series(self, var_def: dict) -> Tuple[pd.Series, pd.Series]:
        """Convert shift and scale values for variables/responses in
        optimization problem dictionary to a series.

        :param var_def: Variables/responses dictionary
        :type var_def: dict
        :return: Series containing shift and scale values for each variable/respons
        :rtype: Tuple[Series, Series]
        """
        # Create dictionary to hold shift and scale values
        shift = {}
        scale = {}

        # Loop through all variables/responses and get their shift/scale values
        for var, info in var_def.items():
            # Check and assign shift value. Can be any number
            if "shift" not in info or info["shift"] is None:
                shift_val = 0
            else:
                if not isinstance(info["shift"], (int, float, dict, number)):
                    raise TypeError(
                        f"ShiftAndScale: Shift value for {var} must "
                        "be a number or a dict! Received a "
                        f"{type(info['shift']).__name__} instead."
                    )

                if isinstance(info["shift"], dict):
                    if not {"value", "use"}.issubset(set(info["shift"])):
                        raise IndexError(
                            f"ShiftAndScale: Shift value for {var} must "
                            "contain fields for both value and use when"
                            f"a dictionary is used/ Received instead"
                            f"{info['shift']}"
                        )
                    if info["shift"]["use"]:
                        shift_val = info["shift"]["value"]
                    else:
                        shift_val = 0
                else:
                    shift_val = info["shift"]
            shift[var] = shift_val
            # Check and assign shift value. Must be a non-zero number
            if "scale" not in info or info["scale"] is None:
                scale[var] = 1
            else:
                if not isinstance(info["scale"], (int, float, dict, number)):
                    raise TypeError(
                        f"ShiftAndScale: Scale value for {var} must "
                        "be a number or a dict! Received a "
                        f"{type(info['scale']).__name__} instead."
                    )
                if isinstance(info["scale"], dict):
                    if not {"value", "use"}.issubset(set(info["scale"])):
                        raise IndexError(
                            f"ShiftAndScale: Scale value for {var} must "
                            "contain fields for both value and use when"
                            f"a dictionary is used/ Received instead"
                            f"{info['scale']}"
                        )
                    if info["scale"]["use"]:
                        scale_val = info["scale"]["value"]
                    else:
                        scale_val = 1
                else:
                    scale_val = info["scale"]
                if scale_val == 0:
                    raise ValueError(
                        f"ShiftAndScale: Scale value for {var} must "
                        "be a non-zero number!"
                    )

                scale[var] = scale_val

        # Return the values in a series
        return (pd.Series(shift), pd.Series(scale))
```

We also define a class that makes it easy to calculate the constraint violations for each site in a Pandas DataFrame given a specific problem.

In [ ]:
```
class ConstraintCalculator:
    """
    A class that calculates the constraint violation of an optimization
    problem that is passed in.

    >>> myCalc = ConstraintCalculator(optprob)

    >>> myCalc = ConstraintCalculator(optprob, individual=True)  # Return the
    overall constraint violation as well as each individual constraint
    violation.
    >>> viol = myCalc(data) # Here data is all points to calculate constraint
    violation of. We will only calculate the constraint violations of responses.
    """

    def __init__(
        self, problem: dict, individual: bool = False, method: str = "euclid"
    ) -> None:
        """Initialize the object.

        >>> myCalc = ConstraintCalculator(optprob)

        >>> myCalc = ConstraintCalculator(optprob, individual=True)  # Return
        the overall constraint violation as well as each individual constraint
        violation.

        Parameters
        ----------
        problem : dict
            Dictionary containing information about the variables and responses
            of the optimization problem.
        individual : bool, optional
            Whether or not calls to the constraint calculator returns the
            violation for each constraint., by default False
        method : str, optional
            Aggregation method to use. Can be one of the following values, by
            default 'euclid'

            - ``euclid``: Euclidean/:math:`L_2` norm

                .. math::
                    v = \\sqrt{x_1^2 + ... + x_n^2}

            - ``manhat``: Manhattan/taxicab/:math:`L_1` norm

                .. math::
                    v = x_1 + ... + x_n

            - ``max``: Maximum/:math:`L_\\infty` norm

                .. math::
                    v = \\max(x_1, ..., x_n)

            - ``avg``: Average violation value

                .. math::
                    v = \\frac{x_1 + ... + x_n}{n}

            - ``median``: Median violation value
            - ``prod``: Product of violations

                .. math::
                    v = x_1 * ... * x_n

            In each of these equations, :math:`x_i` represents the amount that
            a constraint deviates from its bounds in rescaled coordinates.
        """
        # Make sure a dictionary was passed in
        if not isinstance(problem, dict):
            raise TypeError(
                "ConstraintCalculator: opt_prob must be a "
                f"dictionary! Received a {type(problem).__name__} instead."
            )
        # Save optimization problem dictionary
        self._problem = problem

        # Create shift and scale object
        self._shift_scale = ShiftAndScale(problem)
```

```python
        # Save constraint names
        self._constraint_names = problem["constraints"]

        # Save bounds in easy dataframe assembly format
        bounds = [
            [
                problem["responses"][constraint]["bounds"][0]
                for constraint in self._constraint_names
            ],
            [
                problem["responses"][constraint]["bounds"][1]
                for constraint in self._constraint_names
            ],
        ]

        # Shift and scale bounds
        rescaled_bounds = self._shift_scale.design_to_optimizer_space(
            pd.DataFrame(data=bounds, columns=self._constraint_names),
            self._constraint_names,
        )

        # Save bounds
        self._lower_bounds = {
            constraint: rescaled_bounds.loc[0, constraint]
            for constraint in self._constraint_names
        }
        self._upper_bounds = {
            constraint: rescaled_bounds.loc[1, constraint]
            for constraint in self._constraint_names
        }

        # Set whether or not each individual constraint violation is returned
        self._individual = individual

        # Set which method to use
        valid_methods = {
            "euclid",  # Euclidean/L2 norm
            "manhat",  # Manhattan/taxicab/L1 norm
            "max",  # Maximium norm
            "avg",  # Average violation
            "median",  # Median violation
            "prod",  # Product of violations
        }

        # Check that method is valid
        if method.lower() not in valid_methods:
            raise ValueError(
                f"ConstraintCalculator: {method} is not a valid "
                f'method! Must be one of: {", ".join(valid_methods)}'
            )

        self._method = method

    def __call__(
        self, data: pd.DataFrame
    ) -> Union[pd.Series, Tuple[pd.Series, pd.DataFrame]]:
        """Calculate the constraint violation of the given points in rescaled coord

        Parameters
        ----------
        data : pd.DataFrame
            All data points to calculate the constraint violation of.

        Returns
        -------
        pd.Series or (pd.Series, pd.DataFrame)
            Constraint violation of each point. If `individual` was specified in
            the constructor then the constraint violation of each variable for
            all points is also returned.
        """
        # Create series to save violation of each point
        total_violation = pd.Series(index=data.index, dtype=np.float64)

        # Set violation of all points containing NaN values to NaN
        valid_vals = ~data.loc[:, self._constraint_names].isna().any(axis=1)
        total_violation.loc[~valid_vals] = np.nan

        # Shift and scale data
        rescaled_data = self._shift_scale.design_to_optimizer_space(
            data, self._constraint_names
        )

        # Calculate individual constraint violation
        violation = np.maximum(
            0,
            np.maximum(
                self._lower_bounds - rescaled_data, rescaled_data - self._upper_bou
            ),
        )

        # Calculate total violation for each point
        total_violation.loc[valid_vals] = self._compute_violation(violation[valid_v
        # Return results
        if self._individual:
            return (total_violation, violation)

        return total_violation

    def _compute_violation(self, violations: pd.DataFrame) -> pd.Series:
        """Compute the total constraint violation using the desired method.

        Parameters
        ----------
        violations : pd.DataFrame
            Individual constraint violations.

        Returns
        -------
        pd.Series
            Constraint violation for each site.

        Raises
        ------
        NotImplementedError
            Method has not been implemented.
        """
        # Euclidean distance
        if self._method == "euclid":
            ret = ((violations**2).sum(axis=1)) ** 0.5
        # Manhattan/taxicab distance
        elif self._method == "manhat":
            ret = violations.sum(axis=1)
        # Maximum norm
        elif self._method == "max":
            ret = violations.max(axis=1)
        # Average violation
        elif self._method == "avg":
            ret = violations.mean(axis=1)
        # Median violation
        elif self._method == "median":
            ret = violations.median(axis=1)
        # Product of violations
        elif self._method == "prod":
            ret = violations.prod(axis=1)
        else:
            raise NotImplementedError(
                f"ConstraintCalculator: {self._method} " "has not been implemented
            )

        return ret
```

Exploring the design space using Latin Hypercube experimental designs

We now have all the needed classes and methods to explore the feasible space of the examples we provided. To simplify this we wrote some more helper routines.

The first method takes a problem and the number of sites to use in an experiment, and creates a set of sites using the SciPy Latin Hypercube Sampling method. This method needs the number of variables to generate an unscaled set of points, and then uses the bounds defined for each variable to scale the problem such that it can be used. It then creates and returns a DataFrame of the sites in the design space.

In [ ]:
```
def simple_experiment(problem: dict, n_samples: int) -> pd.DataFrame:
    """ Simple class to create an experiment DataFrame

        Parameters
        ----------
        problem : dict
            The problem for this example

        n_sample: int
            The number of sites to generare
        Returns
        -------
        pd.DataFrame
            The DataFrame of experimental sites

    """
    variables = list(problem["variables"].keys())
    nind = len(variables)
    # Get all the bounds for the variables
    bounds = np.array([problem["variables"][var]["bounds"] for var in variables])
    # Generate the experiment
    lhs_instance = lhs(
        nind,
        scramble=True,
        strength=1,
        optimization=None,
        seed=1232,
    )
    # Get the experiment
    normalized_array = lhs_instance.random(n_samples)
    # Scale using the bounds
    exp_array = scale(normalized_array, bounds[:, 0], bounds[:, 1])
    # Create the DataFrame
    exp_df = pd.DataFrame(data=exp_array, columns=variables)
    return(exp_df)
```

The last helper method we defined takes an instance of an example and the number of sites to be used, creates the experiment dataset, calls the example function, calculates and prints statistics about the data set to the screen, and returns the final data set.

In [ ]:

```
def testing(local_eval: CallableClass, num_sites: int) -> pd.DataFrame:
    """ Simple wrapper to create an experiment, evaluate the passed in function, ca

        Parameters
        ----------
        local_eval : CallableClass
            The example evaluator to use

        n_sample: int
            The number of sites to generare

        Returns
        -------
        pd.DataFrame
            The DataFrame of evaluated experimental sites with constraint violation

    """

    eps = 1.e-6
    local_problem = local_eval.problem()
    exp_data = simple_experiment(local_problem, num_sites)
    local_eval(exp_data)
    my_constraint_calculator = ConstraintCalculator(local_problem)
    exp_data['__conviol__'] = my_constraint_calculator(exp_data)
    exp_data['__State__'] = pd.cut(exp_data['__conviol__'], [-np.inf, eps, 100*eps,
    feasible_sites = (exp_data['__State__'] == 'Feasible').sum()
    percentage = feasible_sites/num_sites * 100
    print(f"Test evaluator {local_eval.name} with {len(local_problem['variables'])}
    return(exp_data)
```

Numerical results

In this section we now instantiate the two examples, show the defined experiment, and then explore the effect of using different sized experiments on the ability to find feasible points.

HS100

The first problem is a 7 dimensional problem with an objective function `f` and four constraints (`c1` through `c4`).

In [ ]:
```
hs100 = HS100()
problem = hs100.problem()
problem
```

A good rule of thumbs for a good number of points to use in a Latin Hypercube Sampling (LHS) method is to use this formula:

$$n_s = \frac{(n+1)(n+2)}{2}$$

where n is the number of independent variables. This is the number of samples that in general position uniquely determine a quadratic polynomial model in the independent variables.

We show that this does not find a feasible point, and we need to go to a sample size of 20 times that number to find any site that is feasible. Depending on problem size and cthe computational cost of response evaluations, other scientists suggest a different formula:

$$n_s = 10n$$

where n is the number of independent variables. Again, we see that for this problem we only find a feasible point with 50 times that many sites.

In [ ]:
```
nind = len(problem['variables'])

num_sites = int((nind+1)*(nind+2)/2)
num_sites = int((nind+1)*(nind+2)/2)*10
num_sites = int((nind+1)*(nind+2)/2)*20
num_sites = nind*10
num_sites = nind*10*5

exp_data = testing(hs100, int((nind+1)*(nind+2)/2))
exp_data = testing(hs100, int((nind+1)*(nind+2)/2)*10)
exp_data = testing(hs100, int((nind+1)*(nind+2)/2)*20)
exp_data = testing(hs100, nind*10)
exp_data = testing(hs100, nind*10*5)
```

In [ ]:
```
exp_data
```

We can also visualize the design space, in this case using the Seaborn visualization library. There are many different Python visualization libraries available. Note that this pairplot only works well for this problem and not too many sites.

In [ ]:
```
import seaborn as sns
axgrid = sns.pairplot(exp_data, hue='__State__', height= 2.5)
```

The second problem is even harder, and even increasing the number of sites significantly does not lead to a feasible point found by the LHS experiments.

In [ ]:
```
hs118 = HS118()
problem = hs118.problem()
problem
```

In [ ]:
```
nind = len(problem['variables'])

num_sites = int((nind+1)*(nind+2)/2)
num_sites = int((nind+1)*(nind+2)/2)*10
num_sites = int((nind+1)*(nind+2)/2)*20
num_sites = nind*10
num_sites = nind*10*5

exp_data = testing(hs118, int((nind+1)*(nind+2)/2))
exp_data = testing(hs118, int((nind+1)*(nind+2)/2)*10)
exp_data = testing(hs118, int((nind+1)*(nind+2)/2)*20)
exp_data = testing(hs118, nind*10)
exp_data = testing(hs118, nind*10*5)
```