**Coursework Part 3: Constrained Optimization of Williams-Otto Problem using Data-Driven Optimisation**

**Problem Definition:**
In this coursework, you will design a data-driven optimisation (DDO) algorithm to optimize the constrained Williams-Otto (CWO) benchmarking problem. You have complete freedom in what type of DDO algorithm you want to implement - direct, model-based, evolutionary search, etc. Keep in mind, that constraints need to be handled in this problem, which will be presented in more detail later. Since the takeaway is on optimising chemical engineering systems, we operate under the following assumption:

- Evaluations are expensive, meaning that the runtime of the algorithms is limited by a fixed evaluation **budget of 20 iterations**. However, we will not consider algorithms that take longer than **5 minutes** overall. Therefore, it is a good idea to have your algorithm exit and return the best value found so far if the timing is nearing the 5-minute mark (see for example).

Your team's submission will be a **single** DDO algorithm function that is your best attempt at optimizing the CWO Problem.

**Next Steps:**
It is advised to read this handout thoroughly to understand the submission format and criteria. Then, to develop your understanding, it is recommended to follow the walkthrough included to gain an understanding of the script *ML4CE_WO_Task.py* and how you might want to use *ML4CE_eval_algs_WO.py* to test your algorithm.

**Material Provided:**
*ML4CE_MyAlg.py*

- This file contains a wrapper function for your own algorithm. You may insert it indicated in the file.
- **This file will also be your final submission.**

*ML4CE_WO_eval_algs.ipynb*

- This is where all comes together
- Here, the benchmarking is done, you will include the name of your algorithm here and then run the script for benchmarking and plotting

*ML4CE_WO.py*
- This script has the CWO benchmarking problem implemented. Most importantly, it also has the objective function your algorithms will be minimising by optimising the values of the reactor temperature and the flowrate of reactant B.
- **Nothing for you to do here**

*ML4CE_WO_Wrapper.py*

- This script wraps the CWO benchmarking problem
- **Nothing for you to do here**

*ML4CE_WO_algorithms.py*
- This script contains the exemplary algorithms that you will compete against while constructing your own algorithm. For the grading of the coursework however, your algorithm will only compete against other students' algorithms.
- **Nothing for you to do here**

*ML4CE_WO_utils.py*

- Here, the benchmarking functions as well as plotting functions are contained.
- **Nothing for you to do here**

**Grading (per group):**
- 2-page report on your algorithm (20%).
  - The report should have the following sections:
    - Big picture explanation and intuition behind the algorithm
    - Methodology
    - Pseudocode
    - You are allowed a figure for your algorithm which is not considered in the report length.
    - References are not considered for the length of the report.
  - The report should include a pseudocode following the format specified here.

- The report should also explain the rationale behind the algorithm and in paragraph form the main steps in the algorithm.
- Please do not use a letter smaller than size 11 (with a decent font: e.g., Arial, Times New Roman, Calibri, Latex font), and margins no smaller than 2cm top, bottom, left, right.
- You will be graded based on clarity of communication, creativity of your algorithm and scientific explanation of your methods.
- Note, that for your assessment one or all of the following will be altered:
  - Profit function parameters
  - Kinetic parameters

- Your implementation will be graded using a similar script to *ML4CE_WO_utils.py* provided (80%).
  - You will upload a single python file ".py" such that the file is *"ML4CE_WO_team_name.py"* and the algorithm to be called is *"ML4CE_WO_team_name_algorithm"*. The algorithm itself can be a function or a python object (class).

**IMPORTANT**:

Please ensure your algorithm adheres to the following rules, failure to do so will result in a **5 mark deduction** per infringement:

- Good coding practice: as aforementioned, make sure your function takes the same inputs and outputs as the example functions provided. This is **very important**, as otherwise, we have no way to test your algorithms and provide a mark. This is easily mitigated by **making sure your submission runs in *ML4CE_WO_eval_algs.ipynb*** and ensuring **it follows the format of the *ML4CE_WO_algorithms.py* script** (parameters and return values).
- Stochasticity: Do not set random seeds to your algorithm.
- Packages: please restrict to use numpy, scipy, sobol-seq, random, time, and scikit-learn, as well as the libraries mentioned in ML4CE_WO_requirements.txt. You are allowed to use scipy.optimize to train any surrogates if you wish, but you are not allowed to optimize f() directly. In other words, you are not allowed to use the optimization routines of python packages directly on the functions.
- **Make absolutely sure** that your code runs in an environment where **only** the packages in ***ML4CE_WO_requirements.txt*** (See below for installation) are available.
- Ensure you add your team's names and CIDs to your function, as shown in the example provided.

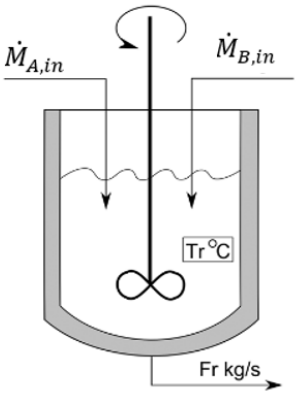
**Setup the environment:**

1. Install Anaconda
2. Create an environment with the latest version of python
3. In the console:
   i. Activate environment with "conda activate name_of_your_env"
   ii. Install the requirements from ***ML4CE_WO_requirements.txt*** with "pip install -r ML4CE_WO_requirements.txt"
4. In case of issues with the jupyter notebook (some kernel needs to be installed) type in the console:
   1. conda install jupyter
   2. conda install -c anaconda ipykernel

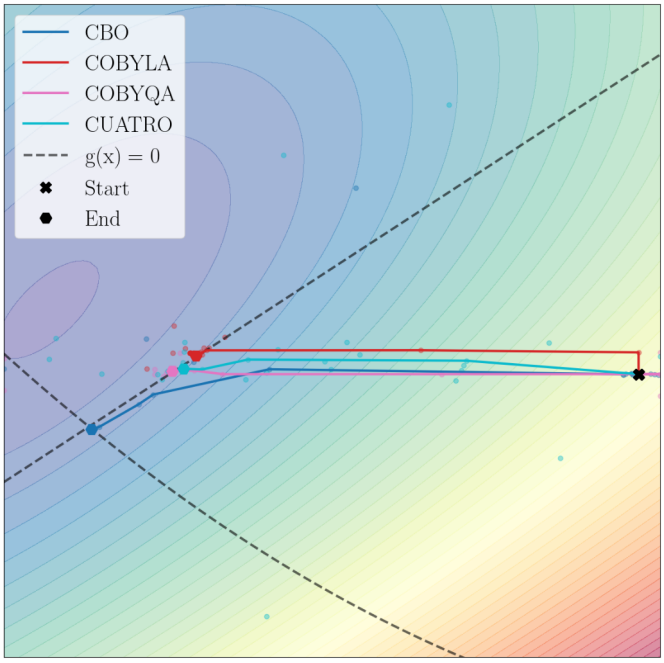
**Problem Walkthrough:**

The Williams and Otto continuous stirred tank reactor (CSTR) is a widely studied example, frequently used to benchmark algorithms. This process, depicted below, involves feeding the reactor with two pure component streams, Fa and Fb (consisting of components A and B, respectively). Components A and B react to form an intermediate product, C, which further reacts with another B molecule to yield the desired products, P and E. A side reaction occurs between components C and P, resulting in the formation of a byproduct, G, which has no commercial value and is considered waste. The reaction mechanisms and kinetics, Ea being the activation energy, and etta_p is the pre-exponential factor as are detailed below:

$$A + B \rightarrow C \qquad k1 = \eta p1 \ \exp(-Ea1/(T_R + 273.15))$$
$$B + C \rightarrow P + E \qquad k2 = \eta p2 \ \exp(-Ea2/(T_R + 273.15))$$
$$P + C \rightarrow G \qquad k3 = \eta p3 \ \exp(-Ea3/(T_R + 273.15))$$




*Above: Reactions and Reactor*




*Above: Williams-Otto contour with dashed constraint-line and a single trajectory for exemplary algorithms from an exemplary shared starting point. The lines show feasible best-so-far evaluation positions (based on which your algorithm will be ranked) and the scattered points show remaining evaluation positions. X-axis is mass flowrate of reactant B and y-axis is reactor temperature TR.*

The process is modelled at **steady state** using **mass balance equations**, with the **reactor temperature (TR)** and **the flow rate of component B (Fb)** as the controlled variables – Hence, this will end up being a 2D optimization problem. The flow rate of reactant A (Fa) and the mass holdup (W) are maintained at constant values.

The **objective** is to **maximize the profit-flow** given by

$$\dot{P} = price_R \dot{M}_R + price_E \dot{M}_E - cost_A \dot{M}_{A,in} - cost_B \dot{M}_{B,in}$$

Additionally, we want to keep the use of reactant A, as well as the production of waste G limited:

$$X_A \leq 0.12$$

$$X_G \leq 0.08$$

**Performance Metric**

Every algorithm is assessed on the WO problem and its constraints, and the performance is compared relative to the other optimization algorithms on the same function. The procedure allocates each algorithm a budget of 20 function evaluations, and the optimization is conducted 10 times per algorithm each time from a different starting point to account for algorithm and function evaluation stochastic factors. For a given algorithm, only the feasible best-so-far values within a trajectory are stored. It is important to note, that this performance assessment is not based on the final objective value that the algorithms arrive at. Instead, the assessment is based on the algorithms' respective trajectories; trajectories offer a more robust, and less arbitrary, measure for comparison.

| | | |
|---|---|---|
| **1** | $r_{k,a} = \frac{y_k - y_{k,a}^{mean}}{y_k - y_k^*}$<br>$0 \leq r_{k,a} \leq 1$ | Relative measure of performance for algorithm a at trajectory position k. y_k is the worst feasible function value of any algorithm at this position. y_k,a^mean is the mean feasible function value achieved by your algorithm a at position k. y_k^* is the best feasible function value achieved by any algorithm at position k. |
| **2** | $\begin{bmatrix} r_{1,a} \\ r_{2,a} \\ \vdots \\ r_{n,a} \end{bmatrix}$ | Store all relative performance measurements, n is the trajectory length (in our case 20) |
| **3** | $p_a = \frac{\sum_{k=1}^{n} r_{k,a}}{n}$<br>$0 \leq p_a \leq 1$ | Performance (in table "score") is the mean over the trajectory length. |

**The remaining part of the document is for the interested reader only.**

**Algorithm Grading:**

To help you evaluate your algorithm's performance and guide its development, you have been provided with the script (*ML4CE_WO_utils.py*) that contains methods which will be used to assess your final algorithm submission.

```
########################################
###########·BENCHMARKING·################
########################################
def·ML4CE_con_eval(···

def·ML4CE_con_table_coursework(···

def·ML4CE_con_table_plot_coursework(info,·test_res_raw,·test_res_norm,·vio_dict):···

def·ML4CE_con_graph_abs(test_res,·algs_test,·funcs_test,·N_x_l,·SafeFig=False):···

def·ML4CE_con_graph_abs_g(test_res,·algs_test,·funcs_test,·N_x_l,·SafeFig=False):···
```

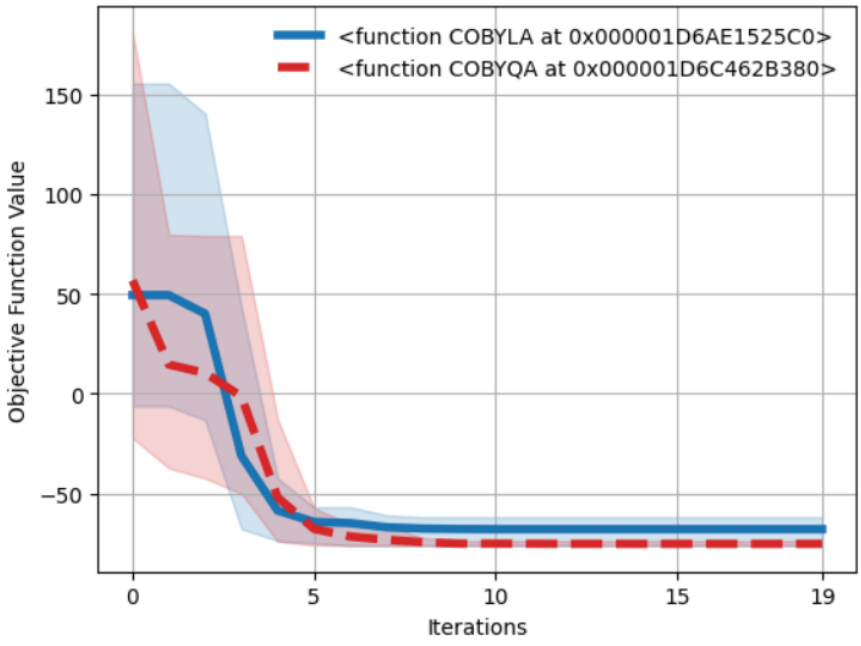
ML4CE_con_eval evaluates your algorithm and its sparring partners on the WO benchmarking problem. Details below.

ML4CE_con_table_coursework fetches the benchmarking results into the table shown below. Explanations below.

ML4CE_con_table_plot_coursework plots the table as sown below.

ML4CE_con_graph_abs plots the convergence graph as sown below

ML4CE_con_graph_abs_g plots the graph for the constraint violation as sown below




This is an example convergence plot. Your algorithm's score is based on the $0^{th}$ - $20^{th}$ data point as described below.

So, when developing your algorithm, you might want to use this convergence plot to gauge its performance.

The closer your algorithm's score is to 1, the better.

| Score | Rank | Feasible Samples [%] |
|---|---|---|
| 0.87 | 0.0 | 89.52 |
| 0.99 | 1.0 | 86.25 |

Score based on benchmarking procedure described below.
Rank is normalized score.
Feasible samples is the percentage of evaluations that do not violate the constraints.

**Code walkthrough (high-level explanation):**

In the following we will go through the benchmarking problem code and give some explanations on the underlying principles. First, lets have a look at the .py-file containing the problem:

*ML4CE_WO.py:*

```python
import numpy as np
from casadi import *

# solver options
options = {'disp': False, 'maxiter': 10000}

# Parameters
Fa = 1.8275 # Mass flowrate Reactand A [kg/s]
Mt = 2105.2 # Total mass hold-up [kg]

class WO_system:

    def __init__(self): ...

    def DAE_system(self): ...

    def integrator_system(self): ...

    def WO_obj_sys_ca(self, u): ...

    def WO_obj_sys_ca_noise_less(self, u): ...

    def WO_con1_sys_ca(self, u): ...

    def WO_con2_sys_ca(self, u): ...

    def WO_con1_sys_ca_noise_less(self, u): ...

    def WO_con2_sys_ca_noise_less(self, u): ...
```

This Python file contains the Williams Otto benchmarking problem, formulated as a differential algebraic system of equations.

On the left-hand side you'll find the global parameters for the solver-options, as well as the mass flowrate for reactant A, as well as the total mass hold up of the reactor (meaning the total mass that is in the system at any time)

The class WO_system consists of the methods listed, which will be explained in the following

The WO benchmarking problem in uses the methods listed above in the following order:

1. WO_obj_sys receives input u (Mass flowrate reactant B, and reactor temperature Tr)
2. WO_obj_sys solves DAE system for given initial conditions and inputs u
    a. Feed initial conditions and inputs to integrator_system
        i. integrator_system retrieves the components of the DAE system
        ii. integrator_system solves algebraic part of the system using a rootfinder (Newton's method) provided by CasADi, a tool for symbolic computation and automatic differentiation
        iii. integrator_system returns the solutions to the DAE system: mass fractions for species A, B, C, E, P, G
    b. Calculate objective function value given the solutions and the inputs
    c. Return objective function value

The remainder of the introduction will go through these steps in more detail

*WO_obj_sys_ca_noise_less(self, u) - Objective Function:*

```
141     def WO_obj_sys_ca_noise_less(self, u):
142         x = self.eval(
143             np.array(
144                 [0.114805,
145                  0.525604,
146                  0.0260265,
147                  0.207296,
148                  0.0923376,
149                  0.0339309])
150                 ,
151                 u)
152 
153         #read mass flowrate B
154         Fb = u[0]
155 
156         #calculate total mass flowrate (inlet and outlet)
157         Fr = Fa + Fb
158 
159         #calculate profit flow
160         obj = -(1043.38 * x[4] * Fr +
161                 20.92 * x[3] * Fr -
162                 79.23 * Fa -
163                 118.34 * Fb)
164 
165         self.f_list.append(float(obj))
166         self.x_list.append(u)
167 
168         return float(obj)
```

Line 142:
First: solve the DAE system for the initial conditions in lines 144-149 and the inputs (u) in line 151.

Lines 144-149:
initial conditions for the algebraic variables: Xa - mass fraction reactant A [kg/kg]
Xb - mass fraction reactant B [kg/kg]
Xc - mass fraction reactant C [kg/kg]
Xp - mass fraction product P [kg/kg]
Xe - mass fraction product E [kg/kg]
Xg - mass fraction product (waste) G [kg/kg]

Line 151:
Input u: Mass-flowrate inlet for reactant B and reactor temperature

The reactor temperature is only needed within the DAE system to solve the reaction kinetics. Flowrate B is then used again for the total flowrate (Line 157) and in the objective function in line 160.

After in 151 the DAE system is solved, and we have obtained the information we need for calculating the profit flow (objective function). These are x[3] and x[4] lines 160 and 161 which correspond to the products E and P, and the flowrates (total, A, and B)

Lines 165 and 166 append the values for the objective function and for the trajectory.

*integrator_system*

```
    def integrator_system(self):

        """
        This function constructs the integrator to be
        suitable with casadi environment, for the equations
        of the model and the objective function with variable
        time step.

        inputs: NaN
        outputs: F: Function([x, u, dt]--> [xf, obj])
        """

        xd, xa, u, ODEeq, Aeq, states, algebraics, inputs = self.DAE_system()
        VV = Function('vfcn', [xa, u], [vertcat(*Aeq)], ['w0', 'u'], ['w'])
        solver = rootfinder('solver', 'newton', VV)

        return solver
```

This function constructs a solver for the system of equations

Line 113:
retrieve components of the DAE system

Line 114:
create a function that takes algebraic variables (xa) and inputs (u) and computes the algebraic equations (Aeq).

Line 115:
Newton-based root-finding algorithm to solve the algebraic equations

*DAE_system*

```
    def DAE_system(self):
        '''
        Algebraic variables:
        Xa --mass fraction reactand A [kg/kg]
        Xb --mass fraction reactand B [kg/kg]
        Xc --mass fraction reactand C [kg/kg]
        Xp --mass fraction product P [kg/kg]
        Xe --mass fraction product E [kg/kg]
        Xg --mass fraction product (waste) G [kg/kg]

        Inputs:
        Fb --mass flow rate of reactand B [kg/s]
        Tr --reactor tempterature [K]
        '''

        #Define states
        states = ['x']
        nd = len(states)
        #define vector containing symbolic
        #variables named xd for state derivatives
        xd = SX.sym('xd', nd)
        for i in range(nd):
            globals()[states[i]] = xd[i]
```

Define algebraic variables, states and global state derivatives (globally to be accessible by the solver later)

```
        #Define algebraics
        algebraics = ['Xa', 'Xb', 'Xc', 'Xe', 'Xp', 'Xg']
        na = len(algebraics)
        xa = SX.sym('xa', na)
        for i in range(na):
            globals()[algebraics[i]] = xa[i]

        #Define inputs
        inputs = ['Fb', 'Tr']
        nu = len(inputs)
        u = SX.sym("u", nu)
        for i in range(nu):
            globals()[inputs[i]] = u[i]
```

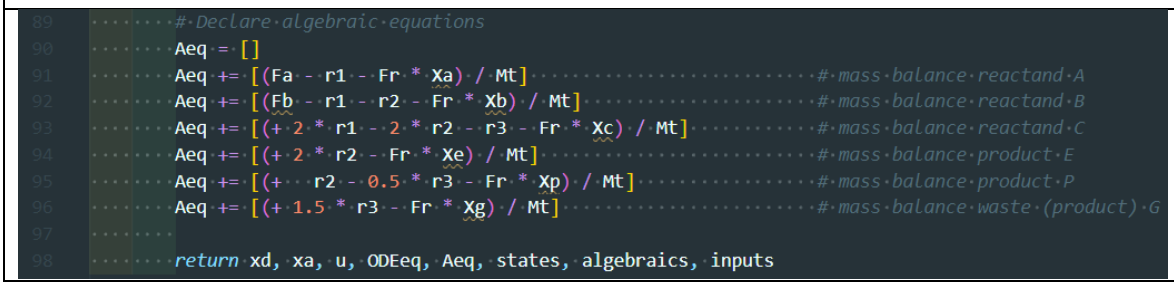
Same for algebraic variables and inputs

```
        #Reparametrization with plant parameters from [1]
        k1 = 1.6599e6 * np.exp(-6666.7 / (Tr + 273.15))
        k2 = 7.2117e8 * np.exp(-8333.3 / (Tr + 273.15))
        k3 = 2.6745e12 * np.exp(-11111. / (Tr + 273.15))

        #total mass flowrate
        Fr = Fa + Fb

        #reaction rate
        r1 = k1 * Xa * Xb * Mt
        r2 = k2 * Xb * Xc * Mt
        r3 = k3 * Xc * Xp * Mt

        #residual for x each algrbraic equation
        x_res = np.zeros((6, 1))
        x_res[0, 0] = (Fa - r1 - Fr * Xa) / Mt                  #residual for mass balance reactand A
        x_res[1, 0] = (Fb - r1 - r2 - Fr * Xb) / Mt             #residual for mass balance reactand B
        x_res[2, 0] = (+ 2 * r1 - 2 * r2 - r3 - Fr * Xc) / Mt   #residual for mass balance reactand C
        x_res[3, 0] = (+ 2 * r2 - Fr * Xe) / Mt                 #residual for mass balance product E
        x_res[4, 0] = (+ r2 - 0.5 * r3 - Fr * Xp) / Mt          #residual for mass balance product P
        x_res[5, 0] = (+ 1.5 * r3 - Fr * Xg) / Mt               #residual for mass balance waste (product) G

        #Define vectors with banes of input variables
        ODEeq = [0 * x]
```

Lines 67-69: Set Arrhenius kinetics for the reactions. Parameters obtained from literature.

Lines 71-77: Calculate total mass flowrate and reaction rates

Lines 80-89: Calculate residuals for

| | |
|---|---|
| | algebraic equations (for the solver) |
|  | Declare algebraic equations and return DAE system |