Coursework Part 2: PID Tuning using Data-Driven Optimisation

Problem Definition:
In this coursework, you will design a data-driven optimisation algorithm to tune a PID controller that is embedded within a simple reactor problem. You have complete freedom in what type of DDO algorithm you want to implement - direct, model-based, evolutionary search, etc. 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 rather than a time budget constraint. However, we will not consider algorithms that take longer than 1 minute for a budget of 50 evaluations. 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 1-minute mark (see for example).

Your team's submission will be a **single** DDO algorithm function that is your best attempt at tuning the PID parameters to subsequently minimise the error in the CSTR system.

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 *PID_Tuning_Task.py* and how you might want to use *ML4CE_evaluate_algorithms_part2.py* to test your algorithm.

Materials Provided:
PID_Tuning_Task.py
- This script has the CSTR model implemented along with the untuned PID controller. Most importantly, it also has the objective function your algorithms will be minimising by optimising the values of the controller gains.
- This script has the function *opt_PID()* that you can use to visualise your algorithm's training of the PID parameters, simulate the CSTR controlled by your tuned controller, and observe the error.

example.py
- This script is an example algorithm that is implemented initially in the *PID_Tuning_Task.py*.
- Your final submission must also be in the form of a .py file, contained in a similar function format that requires the same arguments and returns the same outputs as this script. Essentially, you may use it as a template for your algorithm.

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 rational 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.
- Your implementation will be graded using a similar script to *ML4CE_evaluate_algorithms_part2.py* provided (80%).
  - You will upload a single python file ".py" such that the file is *"team_name_part2.py"* and the algorithm to be called is *"algorithm_team_name"*. The algorithm itself can be a function or a python object (class).

IMPORTANT:

- 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_evaluate_algorithms_part2.py* and ensuring it follows the format of the *example.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. 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** numpy, scipy, sobol-seq, random, time, and scikit-learn packages are available.
- Ensure you add your team's names and CIDs to your function, as shown in the example provided.

Algorithm Grading:

To help you evaluate your algorithm's performance and guide its development, you have been provided with a simplified version of the script (*ML4CE_evaluate_PID_algorithms_part2.py*) that will be used to assess your final algorithm submission.

```python
# -*- coding: utf-8 -*-

###########
# Imports #
###########
# importing task
from PID_Tuning_Task import*
# importing algorithm(s)
from example import*

######################
# Testing Algorithms #
######################

# Initialising Test
student_algorithms = [opt_Powell]
results = []
n_iter = 5

for algorithm in student_algorithms:
    sum_error = 0
    for n in range(n_iter):
        K_Opt, data_res, best_Y, team_names = opt_PID(algorithm)
        raw_error = sum(best_Y[10:50])
        sum_error += raw_error
    avg_error = sum_error/n_iter
    df = {'team': team_names, 'score': avg_error}
    results.append(df)
    print(df)
```

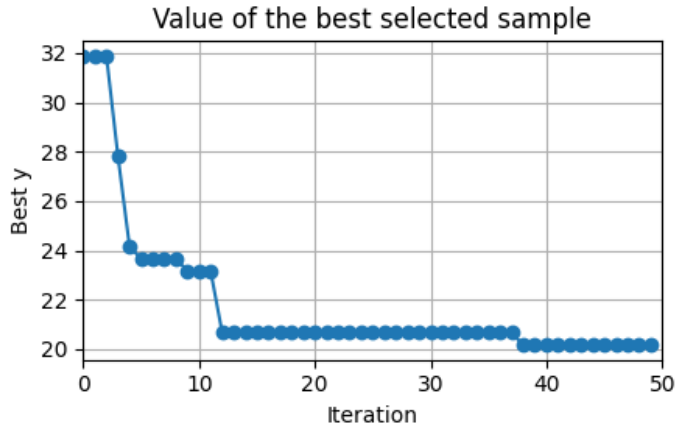


Figure 1 Error vs Iterations from Example Algorithm

This code is relatively simple and starts by importing *PID_Tuning_Task.py* and *example.py* (the example algorithm provided).

In the *student_algorithms* array the function to be tested (from *example.py*) has been added.

n_iter = 5 does not refer to the 50 iterations your algorithm gets for the optimization – instead, it is asking your algorithm to be tested 5 times such that an average error can be calculated.

For each of the 5 iterations, the raw error (defined as the total error from the optimization's 10^{th} to 50^{th} iteration) is stored. The average is then calculated of the 5 *raw_error*'s.

This is an example convergence plot. Your algorithm's score is based on the sum of the error from the 10^{th} - 50^{th} data points as described above.

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

The lower your algorithm's score, the lower the error in the system – which of course is the desired outcome.

```
PS C:\Users\Emma\Documents\ML4ChemEng\Coursework1_DDO\CW_DDO_Part2> & C:/Users/Emma/AppData/Local/Programs/Python/P
ython310/python.exe c:/Users/Emma/Documents/ML4ChemEng/Coursework1_DDO/CW_DDO_Part2/ML4CE_evaluate_PID_algorithms_p
art2.py
this optimization took 0.421614408493042  (s)
this optimization took 0.4265305995941162  (s)
this optimization took 0.41460275650024414  (s)
this optimization took 0.420623779296875  (s)
this optimization took 0.43059539794921875  (s)
this optimization took 0.5379946231842041  (s)
{'team': ['Emma Pajak', 'Antonio Del Rio Chanona'], 'CIDs': ['01234567', '01234567'], 'score': 1202.577601731897}
```

Above is an example output from *ML4CE_evaluate_PID_algorithms_part2.py*. Please remember to add your team's names and CIDs to the algorithm!

Your grade will be determined by the score your algorithm achieves on a similar script. The actual grading script will change the reaction parameters and the sepoints of the CSTR such that no algorithm can achieve an unfair advantage from over-tuning to a specific instance.

Problem Walkthrough:

1. CSTR Model

CSTR with reaction A → B taking place. The reactor has a cooling jacket of temperature T_c that acts as the input of the system, with the reactor temperature, T, and concentration of component A, C_a, as the two states.

$$\frac{C_a}{dt} = \frac{\left(C_{af} - C_a\right) q}{V} - r_A$$

$$\frac{dT}{dt} = \frac{q\left(T_f - T\right)}{V} + \frac{\Delta H}{\left(\rho C_p\right)} r_A + \frac{U_A}{\left(\rho V C_p\right)} (Tc - T)$$

$$r_A = \mathrm{k}_0 \exp\left(-\frac{E}{RT}\right) C_a$$

```
################
#  CSTR model  #
################

# Taken from http://apmonitor.com/do/index.php/Main/NonlinearControl

def cstr(x,t,u):

    # ==  Inputs == #
    Tc  = u   # Temperature of cooling jacket (K)

    # == States == #
    Ca = x[0] # Concentration of A in CSTR (mol/m^3)
    T  = x[1] # Temperature in CSTR (K)

    # == Process parameters == #
    Tf     = 350    # Feed temperature (K)
    q      = 100    # Volumetric Flowrate (m^3/sec)
    Caf    = 1      # Feed Concentration (mol/m^3)
    V      = 100    # Volume of CSTR (m^3)
    rho    = 1000   # Density of A-B Mixture (kg/m^3)
    Cp     = 0.239  # Heat capacity of A-B Mixture (J/kg-K)
    mdelH  = 5e4    # Heat of reaction for A->B (J/mol)
    EoverR = 8750   # E -Activation energy (J/mol), R -Constant = 8.31451 J/mol-K
    k0     = 7.2e10 # Pre-exponential factor (1/sec)
    UA     = 5e4    # U -Heat Transfer Coefficient (W/m^2-K) A -Area - (m^2)

    # == Equations == #
    rA     = k0*np.exp(-EoverR/T)*Ca # reaction rate
    dCadt  = q/V*(Caf - Ca) - rA     # Calculate concentration derivative
    dTdt   = q/V*(Tf - T) \
             + mdelH/(rho*Cp)*rA \
             + UA/V/rho/Cp*(Tc-T)    # Calculate temperature derivative

    # == Return xdot == #
    xdot    = np.zeros(2)
    xdot[0] = dCadt
    xdot[1] = dTdt
    return xdot
```

2. CSTR Simulation

Before introducing the PID controller, the function below defines some initial conditions and simulates the CSTR model under aleatoric (random) conditions – demonstrating poor control of state variables.

```
def simulate_CSTR(u_traj, data_simulation, repetitions):
    '''
    u_traj: Trajectory of input values
    data_simulation: Dictionary of simulation data
    repetitions: Number of simulations to perform
    '''
    # loading process operations
    Ca    = copy.deepcopy(data_simulation['Ca_dat'])
    T     = copy.deepcopy(data_simulation['T_dat'])
    x0    = copy.deepcopy(data_simulation['x0'])
    t     = copy.deepcopy(data_simulation['t'])
    noise = data_simulation['noise']
    n     = copy.deepcopy(data_simulation['n'])

    # control preparation
    u_traj = np.array(u_traj)
    u_traj = u_traj.reshape(1,n-1, order='C')
    Tc     = u_traj[0,:]

    # creating lists
    Ca_dat    = np.zeros((len(t),repetitions))
    T_dat     = np.zeros((len(t),repetitions))
    Tc_dat    = np.zeros((len(t)-1,repetitions))
    u_mag_dat = np.zeros((len(t)-1,repetitions))
    u_cha_dat = np.zeros((len(t)-2,repetitions))

    # multiple repetitions
    for rep_i in range(repetitions):
        x   = x0

        # main process simulation loop
        for i in range(len(t)-1):
            ts      = [t[i],t[i+1]]
            # integrate system
            y       = odeint(cstr,x,ts,args=(Tc[i],))
            # adding stochastic behaviour
            s       = np.random.uniform(low=-1, high=1, size=2)
            Ca[i+1] = y[-1][0] + noise*s[0]*0.1
            T[i+1]  = y[-1][1] + noise*s[1]*5
            # state update
            x[0] = Ca[i+1]
            x[1] = T[i+1]

        # data collection
        Ca_dat[:,rep_i]    = copy.deepcopy(Ca)
        T_dat[:,rep_i]     = copy.deepcopy(T)
        Tc_dat[:,rep_i]    = copy.deepcopy(Tc)

    return Ca_dat, T_dat, Tc_dat
```

3. PID Controller

Next, the PID controller is introduced as a black-box optimisation problem:

$$f = \sum_{k=0}^{k=T_f-1} w_e|e(k)| + w_u|u(k)| + w_{diff}|u(k+1) - u(k)|$$

where w_e, w_{uc}, w_{diff} are weights that assign importance (or "adimensionalize") to the different elements of the objective function.

The objective function comprises three contributions:

1. Overall system error, *e(k)* – the most important contribution.
2. Magnitude of control action, *u(k)* – i.e., how much “fuel” you are using. Important to consider as in reality this is linked to monetary costs.
3. Magnitude of the control action change (i.e., from one step to another), |u(k+1)−u(k)| – it is also important not to drastically change the control action from one time point to another e.g., from a maintenance perspective of valves and actuators, a smooth trajectory is preferable.

N.B. The latter two ‘penalties’ also make the problem more numerically stable.

```
[ ]  ###################
     # PID controller #
     ###################

     def PID(Ks, x, x_setpoint, e_history):

         Ks    = np.array(Ks)
         Ks    = Ks.reshape(7, order='C')

         # K gains
         KpCa = Ks[0]; KiCa = Ks[1]; KdCa = Ks[2]
         KpT  = Ks[3]; KiT  = Ks[4]; KdT  = Ks[5];
         Kb   = Ks[6]
         # setpoint error
         e = x_setpoint - x
         # control action
         u  = KpCa*e[0] + KiCa*sum(e_history[:,0]) + KdCa*(e[0]-e_history[-1,0])
         u += KpT *e[1] + KiT *sum(e_history[:,1]) + KdT *(e[1]-e_history[-1,1])
         u += Kb
         u  = min(max(u,data_res['Tc_lb']),data_res['Tc_ub'])

         return u
```

4. Simulating CSTR with PID Controller

```
def J_ControlCSTR(Ks, data_res=data_res, collect_training_data=True, traj=False):

    # load data
    Ca    = copy.deepcopy(data_res['Ca_dat'])
    T     = copy.deepcopy(data_res['T_dat'])
    Tc    = copy.deepcopy(data_res['Tc_dat'])
    t     = copy.deepcopy(data_res['t'])
    x0    = copy.deepcopy(data_res['x0'])
    noise = data_res['noise']

    # setpoints
    Ca_des = data_res['Ca_des']; T_des = data_res['T_des']

    # upper and lower bounds
    Tc_ub  = data_res['Tc_ub'];  Tc_lb  = data_res['Tc_lb']

    # initiate
    x         = x0
    e_history = []

    # Simulate CSTR with PID controller
    for i in range(len(t)-1):
        # delta t
        ts      = [t[i],t[i+1]]
        # desired setpoint
        x_sp    = np.array([Ca_des[i],T_des[i]])
        # compute control
        if i == 0:
            Tc[i] = PID(Ks, x, x_sp, np.array([[0,0]]))
        else:
            Tc[i] = PID(Ks, x, x_sp, np.array(e_history))
        # simulate system
        y         = odeint(cstr,x,ts,args=(Tc[i],))
        # add process disturbance
        s       = np.random.uniform(low=-1, high=1, size=2)
        Ca[i+1] = y[-1][0] + noise*s[0]*0.1
        T[i+1]  = y[-1][1] + noise*s[1]*5
        # state update
        x[0] = Ca[i+1]
        x[1] = T[i+1]
        # compute tracking error
        e_history.append((x_sp-x))

    # == objective == #
    # tracking error
    error = np.abs(np.array(e_history)[:,0])/0.2+np.abs(np.array(e_history)[:,1])/15
    # penalize magnitud of control action
    u_mag = np.abs(Tc[:]-Tc_lb)/10
    u_mag = u_mag/10
    # penalize change in control action
    u_cha = np.abs(Tc[1:]-Tc[0:-1])/10
    u_cha = u_cha/10

    # collect data for plots
    if collect_training_data:
        data_res['Ca_train'].append(Ca)
        data_res['T_train'].append(T)
        data_res['Tc_train'].append(Tc)
        data_res['err_train'].append(error)
        data_res['u_mag_train'].append(u_mag)
        data_res['u_cha_train'].append(u_cha)
        data_res['Ks'].append(Ks)

    # sums
    error = np.sum(error)
    u_mag = np.sum(u_mag)
    u_cha = np.sum(u_cha)

    if traj:
      return Ca, T, Tc
    else:
      return error + u_mag + u_cha
```

The *J_ControlCSTR()* function now brings together the CSTR model with the PID controller.

Inputs include the optimised controller K-values (*Ks*) and the simulation information stored in a dict (*data_re*s).

For this coursework you will NOT need to edit this function, however, it is of use to have a working understanding of its purpose. The function takes the optimised K-values for the PID and simulates the CSTR being controlled by the tuned PID.

The system error is calculated, alongside the trajectories of the input and state variables, T, C_a, and T_c.

5. Importing in the Algorithm and Optimizing the PID Parameters to Minimize Error

```
##################################################
# Practising Optimising the PID Tuning Parameters #
##################################################

# Importing the example optimisation algorithm from its file
from example import opt_Powell
# maximum available iterations
iter_tot =  50

# optimises the PID parameters using your algorithm and returns the optimal values
# you do not need to change this function!
def opt_PID(your_opt_alg):
    # bounds
    boundsK = np.array([[0.,10./0.2]]*3 + [[0.,10./15]]*3 + [[Tc_lb-20,Tc_lb+20]])
    # plot training data
    data_res['Ca_train']    = []; data_res['T_train']     = []
    data_res['Tc_train']    = []; data_res['err_train']   = []
    data_res['u_mag_train'] = []; data_res['u_cha_train'] = []
    data_res['Ks']          = []

    start_time = time.time()
    K_Opt, f_opt, other_outputs, team_names, cids = your_opt_alg(J_ControlCSTR, 7, boundsK, iter_tot)
    end_time   = time.time()

    print('this optimization took', end_time - start_time,' (s)')

    evals = np.array(data_res['Ks']).shape[0]
    best_Y = plot_convergence(np.array(data_res['Ks']).reshape(evals,7), None, J_ControlCSTR)

    return K_Opt, data_res, best_Y, team_names, cids


# Replace 'opt_Powell' with your algorithm
K_Opt, data_res, best_Y, team_names, cids = opt_PID(opt_Powell)

##################################################
# Plotting Training & Simulation with Optimised Ks #
##################################################
# Plot training runs
plot_training(data_res, iter_tot)

reps = 10
Ca_eval = np.zeros((data_res['Ca_dat'].shape[0], reps))
T_eval = np.zeros((data_res['T_dat'].shape[0], reps))
Tc_eval = np.zeros((data_res['Tc_dat'].shape[0], reps))

for r_i in range(reps):
    Ca_eval[:,r_i], T_eval[:,r_i], Tc_eval[:,r_i] = J_ControlCSTR(K_Opt,
                                                    collect_training_data=False,
                                                    traj=True)
# Plot the results with tuned PID
plot_simulation(Ca_eval, T_eval, Tc_eval, data_res)
```

This is the most important part of the script for your coursework – however, there is very little you need to change here. The script pulls in the example algorithm and *opt_PID()* calls your function and has it optimize the objective function discussed above. The most important output is *K_opt*, which stores the PID's optimized K-values as optimized by your algorithm. The function also pulls through your team's names and CIDs.

The plotting training and simulation section does not require you to edit it at all, however, it provides a way for you to visualise your algorithm's convergence, and the resulting simulation of the CSTR with the PID tuned by your algorithm. Ultimately, you want to edit *"from example import opt_Powell"* to reflect your own algorithm and replace instances of *opt_Powell* with your own function.

Example Algorithm

```python
import scipy
import numpy as np

def opt_Powell(f, x_dim, bounds, iter_tot):
    """
    Tuning the PID parameters by optimising the objective function using Powell's method.
    https://docs.scipy.org/doc/scipy/reference/optimize.minimize-powell.html

    Parameters:
        f (callable): The objective function to be minimized.
        x_dim (int): Number of state variables (tuning parameters).
        bounds (list of tuple): The bounds for tuning parameter, e.g., [(min1, max1), ...]
        iter_tot (int): The maximum number of iterations available.

    Returns:
        myopt.x: numpy array of tuned parameter values.
        myopt.fun: optimal value of objective funtion.
    """

    # Iterations to find good starting point
    n_rs           = 5

    # Evaluate first point using random search
    f_best, x_best = Random_search(f, x_dim, bounds, n_rs)
    iter_          = iter_tot - n_rs

    # Restructure bounds for Powell method
    bnds  = tuple(map(tuple, bounds))

    myopt = scipy.optimize.minimize(f, x_best, bounds=bnds,
                                    method='Powell', options={'maxfev':iter_},
                                    )

    team_names = ['Emma Pajak', 'Antonio Del Rio Chanona']
    cids = ['01234567', '01234567']
    return myopt.x, myopt.fun, myopt, team_names, cids
```

This is the *example.py* script you have been provided with.

It is good practice to start a function with a docstring, which at least details the parameters and returns of your function.

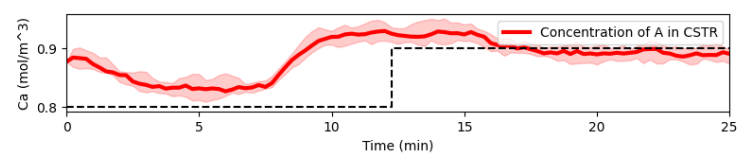
Although not necessary for this course, if you are interested in developing your coding practice, check out PEP8 which is a widely used Python style guide.

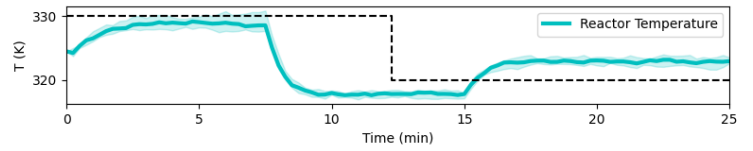
You can see this optimization algorithm simply implements scipy's optimize using the method "Powell".

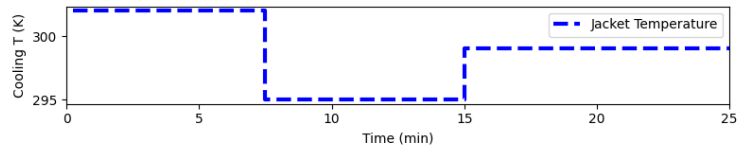
Importantly, note the team's names and CIDs are returns.

(This algorithm also calls a *Random_search()* function to find a 'good' starting point which is embedded within the *example.py* script)

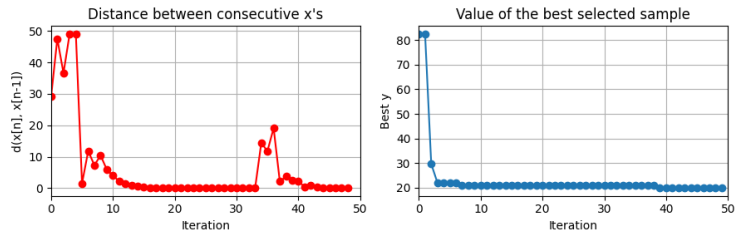
So, finally, what should happen if you run *example.py* in the PID_Tuning_Task.py?



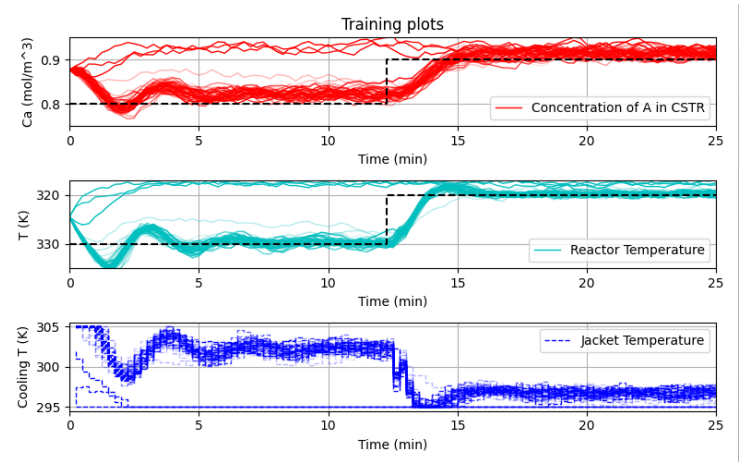




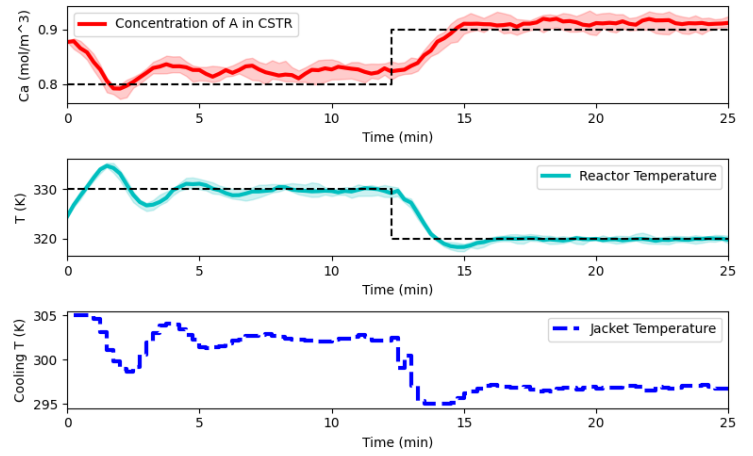
These plots visualise the results of *simulate_cstr()*, i.e., the system without any PID implementation to minimize error.



These convergence plots show the evolution of your algorithm across the 50 iterations. Firstly, the distance between consecutive best *K_opt*, and secondly, what the system error is at each iteration. Remember, you will be graded on the cumulative total from 10^{th} – 50^{th}.



These training plots help you visualise input and two state variables' trajectories for each of the 50 iterations.



Finally, these plots visualise the simulation of the PID-controlled CSTR, tuned with your optimal *K_opt* found by your algorithm.