


ML-Driven Real Time Optimization of a Chemical Reactor

Completed for the Certificate in Scientific Computation
Fall 2025

Pranav Abbaraju
Candidate for B.S. in Chemical Engineering
McKetta Department of Chemical Engineering
Cockrell School of Engineering - University of Texas at Austin


Dr. Thomas Badgwell
Professor of Practice
McKetta Department of Chemical Engineering

ABSTRACT

The purpose of this project was to evaluate the potential for applying machine learning (ML) to the real-time optimization (RTO) of a chemical reactor process. Using Python, I developed a program that simulates proportional-integral (PI) control and RTO of a continuous stirred tank reactor (CSTR), in which a simple chemical reaction involving two species (A and B) occurs. The main simulation function runs a PI control function at a time interval specified for control frequency, and an RTO function for profit maximization at a different time interval specified for optimization frequency. The PI control function adjusts feed flowrate to control the concentration of species B, which is the desired product sold for revenue. The RTO function returns the optimal setpoint for the concentration of B, which the PI control algorithm uses for dynamic profit maximization.

The main element of variation in this simulation was the chosen algorithm for RTO calculation. I first built a traditional RTO algorithm based on the optimization theory of nonlinear programming (NLP). Subsequently, I built a framework that uses data generated from the traditional algorithm to train an ML model, which is then run as a proxy algorithm for calculating optimal solutions. Ultimately, the objective was to benchmark and compare the performance of a tuned ML-driven RTO algorithm with traditional RTO. Based on the results, I concluded ML RTO to be a potentially superior technology that can provide advantages over traditional RTO in terms of computational speed and solution optimality. The results also served as a basis for making recommendations on potential extensions of this work to further evaluate the implementation of ML for industrial optimization.

INTRODUCTION

In this project, the central problem was simulating the real-time optimization of a CSTR to dynamically find optimal concentration setpoint that maximizes the profit generated by the process. Although the term for this is real-time optimization, it is normally performed over discrete time intervals in industrial application at a less frequent rate than the controller action. These RTO time intervals, despite being wider than the time intervals of controller action, are often chosen to be narrow enough to approximate a continuous setpoint optimization, hence the term “real-time”. Accordingly, my Python simulation involved controller action at a very frequent (nearly continuous) rate of 0.1 second, while RTO was run every 5 seconds. The comparative evaluation of interest was to benchmark performance of RTO for the CSTR using the traditional algorithm vs. ML models.

One of the well-known advantages of ML is that it is much faster than traditional numerical solvers. Given this theoretical trend observed across various applications of ML, my prediction was that this project would yield similar results, where ML RTO executes with a much faster runtime than the traditional RTO algorithm for the identical set of process and simulation conditions. However, prior to having started programming the simulation, I did not know the specific extent to which the runtime would differ between the two algorithms. To test my general prediction of ML having a faster runtime and to more specifically obtain the relative runtime

advantage of ML for this specific process optimization application, I used benchmarking methods in Python that tracked computational runtimes for the two RTO algorithms and returned them for ease of comparison.

Contrary to the predicted speed advantage, I predicted the actual profit generated from the ML RTO to be slightly less than that generated from traditional RTO. Theoretically, traditional RTO should always give the exact optima through mathematical calculations of the optimization problem, whereas the solutions given by ML regression models are predictions generated quickly through mappings of the solution dataset. These predictions will always have a residual error, hence making the generated optima slightly less optimal than the exact mathematical optima. Based on this theoretical expectation, I predicted that the generated profit from ML RTO would be slightly less than that generated from traditional RTO based on exact optima. Despite this predicted drawback, I believed the speed advantage and reduced costs of computation would qualify ML as a promising option to incorporate nevertheless into industrial RTO.

The results of the simulation generally confirmed the prediction of speed increase from using ML over a traditional solver for RTO, with the occasional exception of some random forest models. While all the tested model types other than random forests demonstrated a consistent increase in speed, the exact factor of speed increase varied based on the type of model and the chosen tuning parameters. The decision tree ML models ran particularly quickly, with the tuned model exhibiting a speed increase factor of almost 21 compared to the traditional RTO algorithm. Meanwhile, the tuned KNN model ran almost 12 times faster than the traditional RTO. The slowness of the random forest model can be attributed to the fact that it uses multiple decision trees. Gradient boosting models, on the other hand, iteratively improve a fewer number of decision trees to achieve better accuracy and faster performance.

The simulation results also showed that, contrary to my predictions, the generated profit from the reactor generally increased when using well-tuned ML RTO models compared to the traditional RTO algorithm, especially when using the well-tuned gradient boosting models. While the exact reason for this was not concretely proven due to it being outside the scope of this project, a likely reason is that the well-tuned ML models learn patterns in the training dataset effectively enough to find more stable and exact global optima compared to the traditional RTO solver, which may have involved some inaccuracies and noise in its calculated optima of the non-convex objective function.

MATERIALS & METHODS

Reactor Design Equations

The simulation was based on the following non-elementary reaction of a fractional order (chosen to be 1.9) and the corresponding rate law.

$$\begin{aligned} A &\rightarrow B \\ \text{Rate law: } r &= kC_A^{1.9} \end{aligned} \quad (1)$$

Additionally, the Arrhenius relationship shown below was used to model the rate constant k .

$$k = k_0 \cdot e^{\frac{-E_a}{RT}} \quad (2)$$

From fundamentals of reactor design, the following material balance equations were derived for a CSTR that is not at steady state. Since the program intended to simulate dynamic control of the reactor, steady state was not assumed when solving balances at each simulation step. However, the steady state assumption was used when converting between the setpoint concentration of species B and optimal flowrate required for that concentration (see **RTO and Control Scheme** subsection).

$$\frac{dC_A}{dt} = \frac{(C_{Af} - C_A) \cdot Q - V \cdot r}{V} \quad (3)$$

$$\frac{dC_B}{dt} = \frac{(C_{Bf} - C_B) \cdot Q - V \cdot r}{V} \quad (4)$$

Since the simulation has purely species A in the feed, the value of 0 was substituted for C_{Bf} when including **Equation 4** in the simulation program.

The following energy balance equation was used to simulate the non-isothermal CSTR.

$$\frac{dT}{dt} = \frac{Q \cdot \rho \cdot C_p \cdot (T_f - T) - \Delta H_{rxn} \cdot V \cdot r + UA \cdot (T_c - T)}{V \cdot \rho \cdot C_p} \quad (5)$$

Appendix A tabulates information about the parameters used in **Equations 1 to 5**.

Process Control Algorithm

From process control theory^[11], the velocity form of the proportional-integral-derivative (PID) control algorithm was used. For simplicity, the derivative term was omitted to simulate proportional-integral (PI) control in this simulation, as PI control is often enough for satisfactory control responses in industrial processes.

$$\Delta p_k = p_k - p_{k-1} = K_c \cdot [(e_k - e_{k-1}) + \frac{\Delta t}{\tau_I} e_k] \quad (6)$$

Realistic Plant Operation

To build a somewhat realistic process simulation, variation was introduced in the process parameters through methods that were consistent with chemical engineering fundamentals. One element of variation introduced was noise and transient relationship in the pre-exponential rate constant k_0 . This k_0 term was defined as a Gaussian (normal) distribution, where the mean was a time-dependent function, and the noise was determined by a specified standard deviation (see **Appendix A, Table 1**). Another element of variation was the Arrhenius relationship for temperature dependence of the actual rate constant k . This relationship was given in **Equation 2**. The simulation used a transient piece-wise function for concentration of species A in the reactor feed (see **Appendix A, Table 1**). The justification for this function is that it is a more realistic model of plant operation than simply using one constant value throughout the simulation timespan.

The simulation also involved limits on controller action to more realistically model the physical limitations of valves and actuators in real plant operation. The feed flowrate was bound to be between 0.1 and 3 L/s. This represented real-world limitations of the equipment (such as valves, piping, and vessels), which often have minimum and maximum allowable flowrates to prevent damage or excessive economic losses. Additionally, the change in flowrate (dQ) at each control iteration was limited such that its magnitude is less than 10% of the time interval (seconds) for control action. Although the units of dQ are L/s and the time interval was in seconds, the defined relation was a convenient method to relate the magnitudes of the two quantities for an imposed limitation on the change in flowrate allowed per control iteration. Both the implemented limitations had considerable stabilizing effects on the simulation, preventing the controller from inducing unrealistic spikes in the flowrate to maximize profit, which would not be physically possible in real plant operation.

Optimization - Nonlinear Programming (NLP)

Although the *scipy* Python library contains functions that solve optimization problems automatically with the necessary algorithm, some background knowledge of optimization theory of nonlinear programming (NLP) problems was used in formulating an appropriate objective function for this simulation. NLP is a classification of optimization problems that involves

optimizing a nonlinear function that may be subject to a set of constraints and/or bounds [6]. In this simulation, a realistic objective function formulation was attempted by including revenue and cost terms that depend nonlinearly on process parameters. Additionally, physical bounds were defined for the feed flowrate ($Q \in [0.1, 3]$ L/s), qualifying the formulated profit objective as a bound-constrained optimization problem. For a balance of realism and simplicity, additional constraints were not included: a more complex process simulation would involve additional relevant constraints. In the *scipy.optimize.minimize* function, the method was set to L-BFGS-B (Limited-memory Broyden, Fletcher, Goldfarb and Shanno method with bounds) [10], which is an appropriate solver method for bound-constrained minimization problems. Although the objective was to maximize profit, the formulation was built to return the negative value of profit. This allowed for a minimization of negative profit (which is mathematically equivalent to profit maximization) to be implemented. The standard BFGS method uses the gradient of the objective function to converge to a solution more quickly than the standard simplex method: L-BFGS-B is a modification of the BFGS method which uses a limited memory BFGS matrix to approximate the Hessian matrices for bound-constrained problems [2].

RTO and Control Scheme

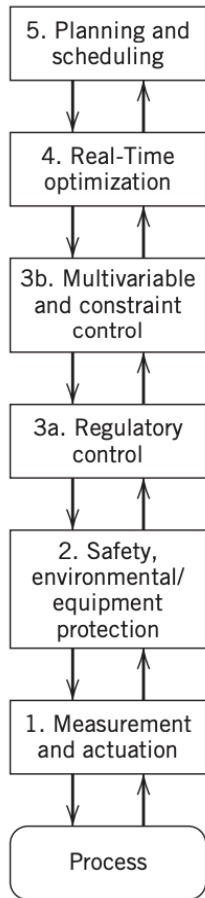


Figure 1. Diagram depicting the optimization and control hierarchy of a plant, sourced from "Process Control and Dynamics" [11]

The real-time optimization (RTO) objective in this simulation was to dynamically determine the optimal feed flowrate that maximizes instantaneous profit generation from the CSTR. After determining this optimal flowrate, a theoretical steady-state was assumed in order to mathematically find the optimal concentration of species B (C_B) that corresponds with this optimal flowrate. In the simulation, the piecewise variation of species A concentration (C_{Af}) in the feed involves large periods of time where the C_{Af} retains a set constant value. The time required by the controller to achieve its setpoint is less than the duration of each set C_{Af} value, which is primarily what influences the setpoint generated by RTO. Therefore, the assumption of an eventual steady-state relationship between optimal flowrate and setpoint concentration of species B was considered to be a valid approximation for this specific simulation. Future extensions of this work would benefit from a more rigorous approach involving a dynamic RTO methodology.

After solving for the optimal concentration of B, its value is then passed as the setpoint to the controller function to use in the subsequent control iteration. The control problem in this program is to adjust the feed flowrate (the manipulated variable) in order to control the concentration of species B (the controlled variable), as B is the product sold for revenue. At each control iteration, the error between the value and the setpoint of C_B at both the current time point and the previous point time is used by the PI controller function to determine the required change in flowrate to achieve the setpoint. The controller changes the flowrate dynamically to reach the C_B setpoint that was passed down from the optimizer. This pipeline aligns with the general hierarchy of industrial plant operation (see **Figure 1**).

Figure 1 shows that the optimizers pass down setpoints to the multivariable and constraint controllers of the plant (levels 4 and 3b, respectively). Accordingly, this simulation involved taking in process parameters as input to perform RTO calculations that return the C_B setpoint as an output, which is then passed down the hierarchy as an input for control action. Due to the limited scope of project complexity, the controller function was formulated to control a single variable: the concentration of species B in the reactor. To model real chemical plants, future extensions of the program could implement a multivariable control scheme (i.e. controlling both C_B and the reactor temperature T). Additionally, incorporating additional steps of the hierarchy into the simulation would support a more realistic representation of real plant operation.

Python Programming Fundamentals

In building the simulation, basic computational methods from programming fundamentals were used, including:

- Existing functions in Python libraries
- Custom functions built in this program
- For loops
- Conditional logic (if/elif/else)

Additionally, several Python libraries were leveraged for different necessary applications:

- *Time*: benchmarking runtime of code snippets
- *Numpy*, *Math*: mathematical operations
- *Pandas*: building and using dataframes
- *Matplotlib*: plotting results
- *Scipy*: traditional solver of optimization objective function, ordinary differential equation (ODE) solver, numerical integrator, numerical algebraic equation solver
- *Scikit-Learn (or sklearn)*: training, testing and tuning ML models of four types, and finding performance metrics

Machine Learning in Python

In the implementation of machine learning in python, first the solutions returned by the traditional RTO algorithm are used to generate a training dataset. The computational steps needed for this dataset generation are ensured to remain outside of the main RTO calculations, so that they were not factored into the runtime metrics. The generated dataset is returned by the main simulation function. Using the data, ML models are trained to predict the optimal feed flowrate that maximizes the instantaneous profit. The main simulation function includes logic to pass in the trained ML model, which can be used for RTO calculations. The RTO is then performed with the steps previously outlined.

Common practices were used to build ML models in Python using the *sklearn* library. This involved the following steps using functions from the *sklearn* library:

- Train-test split, and model training

- Generating test data with the *predict* function, and finding R^2 performance metric, and returning the final profit generated from the process
- Iteratively tuning hyperparameters of models to achieve the best balance of performance

In the process of implementing the final step listed above (hyperparameter tuning), the code was tailored to this specific application of using ML for RTO: it benchmarked the final yearly profit that was generated using the ML model, rather than solely relying on standard performance metrics. This was a more direct quantification of ML performance specifically when applied as an RTO solver dynamically in this CSTR simulation.

The following four types of ML regression models were tested in the simulation: K-nearest neighbors regressor, decision tree regressor, random forest regressor and gradient boosting regressor. The K-nearest neighbor (KNN) regressor is a simple nonparametric algorithm. It finds the specified number of nearest neighbors based on the Euclidean distance between points in the training data. Random forest and gradient boosting regressors use an ensemble technique. The random forest regressor combines many decision trees into one model, which is known as bootstrap aggregation (bagging) ^[9]. The gradient boosting regressor, on the other hand, builds decision tree instances sequentially. The algorithm places greater weight on the instances with greater prediction error, effectively focusing on “boosting” the prediction accuracy of the weaker trees during model training. The key difference between the two ensemble techniques is that the former builds many independent decision trees and aggregates them, whereas the latter sequentially boosts fewer trees to improve prediction accuracy ^[1]. Gradient boosting has been regarded as one of the top techniques used to solve prediction problems” in an academic study, due to its ability to sequentially learn from its own prediction errors ^[8].

The prediction steps for each of the four ML model types are performed in the program using the *predict* function of the *sklearn* library. This function takes in a dataset of the feature values as the input parameter. The function then performs a computation to generate a dataset of predicted target values. The computation steps depend on the type of ML model. The KNN regressor model first calculates distance between the input point and all other data points, selects the number of nearest neighbors to the data point, then averages the target values of these nearest neighbors to return the final predicted value ^[12]. The decision tree regressor model starts at the root node of the tree and moves downward to internal tree nodes. At each node, the model compares the feature value against the decision rule and accordingly chooses the branch to follow: this process is repeated until a leaf node is reached. The returned prediction is the average of the target values of all training samples corresponding to the final leaf node ^[9]. The random forest regressor model follows the same computational steps for each individual decision tree, which returns its own prediction for the target variable. Afterwards, the model takes the average of all predictions from the tree to generate the final predicted value ^[9]. After following the appropriate computational steps to generate the prediction based on the chosen ML model, the *predict* function returns the final predicted value of the target variable, which is the optimal feed flowrate that maximizes profit.

RESULTS

The plots shown below in **Figure 2** are of dynamic control and optimization using the traditional RTO algorithm, which optimizes the NLP objective with the L-BFGS-B solver method via the `scipy.optimize.minimize` function.

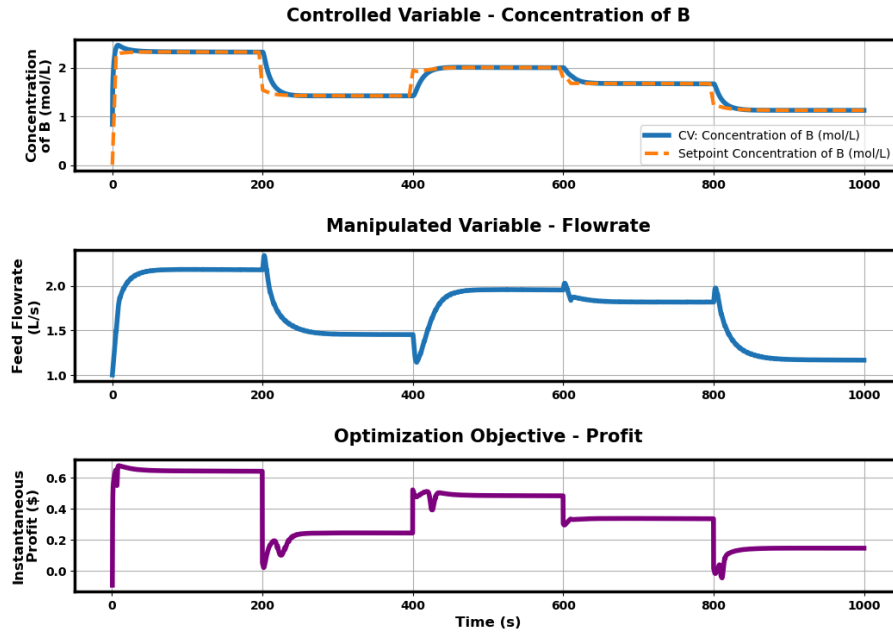


Figure 2. Plots of dynamic responses with the optimization algorithm set to traditional RTO for the CSTR simulation

The plots in **Figure 2** were useful in verifying that the simulated PI controller was working properly, which is proven by the fact that the controlled variable is brought to its setpoint continuously. To verify the RTO algorithm, further evidence was required. **Figure 3** is a plot of the objective function, with instantaneous profit generation graphed against the feed flowrate. This plot verifies that traditional RTO is choosing the optimal feed flowrate to maximize profit, as intended. In the program, I generated such plots at several time points and ensured that the optimizer was detecting global optima as intended (see **Appendix E, Figures 12 to 19**).

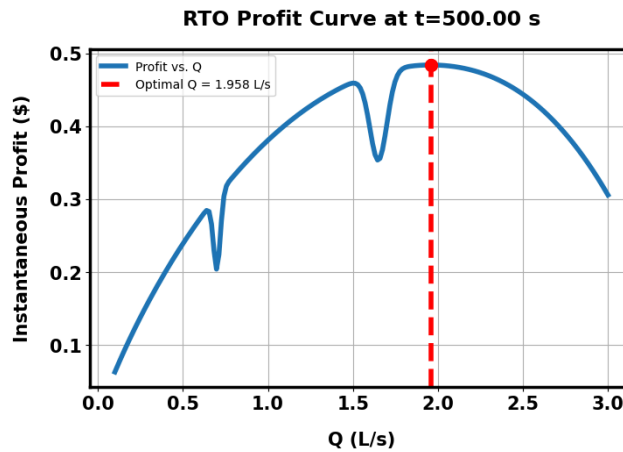


Figure 3. Plot of profit objective vs. feed flowrate at the simulation time point of 500 seconds

The figure below shows the results for tuning the number of initial guesses of the traditional optimizer to generate optimal profit. Based on the tuning results, 2 initial guesses are enough in this particular simulation; only one guess, however, is not enough to generate maximum profit.

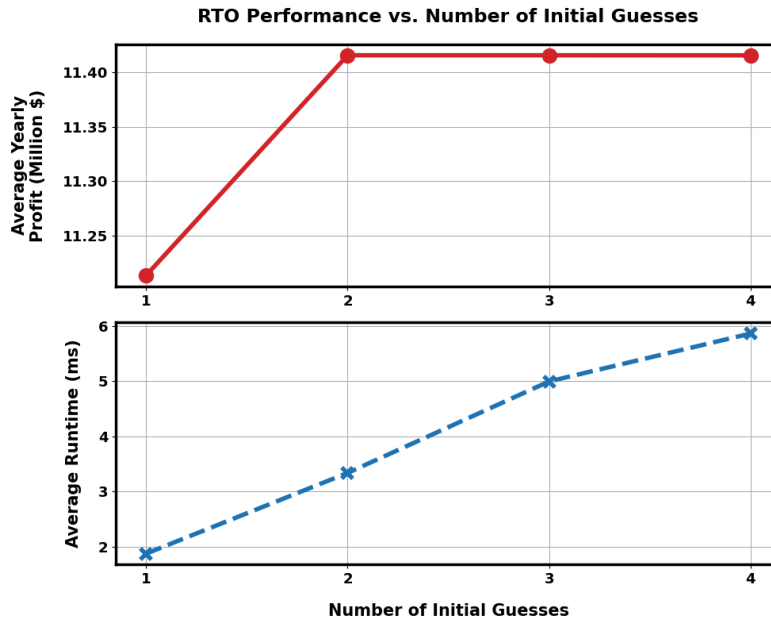


Figure 4. Tuning plot to determine the optimal number of initial guesses for traditional RTO

To ensure a fair comparison of the traditional RTO with ML RTO, I consistently used 2 initial guesses across all runs of traditional RTO, since the results visualized in **Figure 4** demonstrated that the best balance between profit and runtime was achieved with 2 guesses. Of the ML model types, I first trained the KNN regressor models, where the number of nearest neighbors was varied (see **Appendix G, Figure 21**). Despite achieving moderately high R^2 of at least 0.89, the KNN regressor was overall not a beneficial model type for profit generation. It may have overfitted to the training data without effectively learning the true patterns, as the profit relative to traditional RTO generally decreased with an increasing number of nearest neighbors. The tuning results for the decision tree regressor models (see **Appendix G, Figure 22**) yielded a better performance than KNN. However, I noticed that this type of ML model was highly inconsistent across different runs of the program. Although, there was usually a profit gain relative to traditional RTO for several maximum depth values, the optimal maximum depth parameter value for profit generation varied between different runs of the code, suggesting model inconsistency. The model also frequently returned slight losses in profit, further supporting this conclusion. The key strength of this decision tree model is its speed, as it boasts the fastest runtimes out of all the types of ML models tested in the simulation program. With the random forest models, the tuning results (see **Appendix G, Figure 23**) show the poorest results out of all models. There was consistently a profit loss across all tested tuning parameters, with greater losses for greater number of estimators, suggesting that the model may have been overfitting (similar to KNN). The factor of runtime decrease was also the lowest of all the model types, and the runtime would frequently be even slower than traditional RTO. Thus, random forest models did not prove to be beneficial for ML RTO in this simulation.

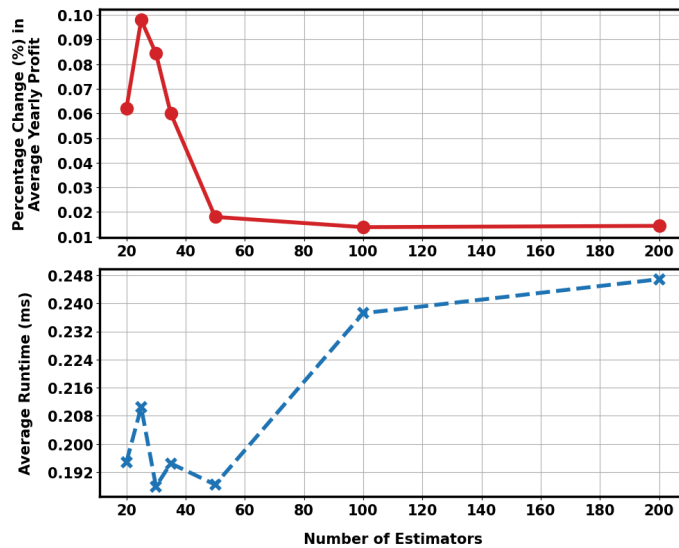


Figure 5. Tuning results for gradient boosting ML RTO model

The simulation and tuning results of the gradient boosting regressor model, which performed better than the other model types, are shown in **Figure 5**. Out of the tested model types, this was overall the most consistent and advantageous in achieving a balance between high profit gain and a significant runtime decrease relative to traditional RTO. The R^2 metric was also consistently above 0.98 for gradient boosting models (see **Appendix G, Table 6**) Several runs of the code yielded a very similar optimal number of estimators in the approximate range of 22 to 30, suggesting a high degree of consistency.

To make a closer determination of the best tuned model, I performed a finer tuning of the Gradient Boosting Regressor in later parts of the program.

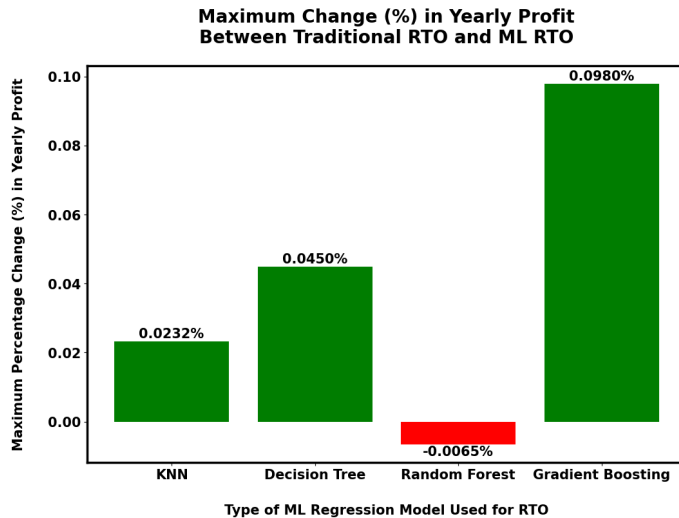


Figure 6. Maximum profit gain from each type of ML model

Figure 6 compares the maximum change in profit from the best-tuned model of each type. Based on the results, it is clear that the gradient boosting algorithm offers the greater advantage in fulfillment of the optimization objective. The profit gain from the decision tree model is neither as high nor as consistent. The KNN model generates a much lower profit gain, and the random forest model returns a loss in profit, even with the optimal number of estimators chosen from the tested values.

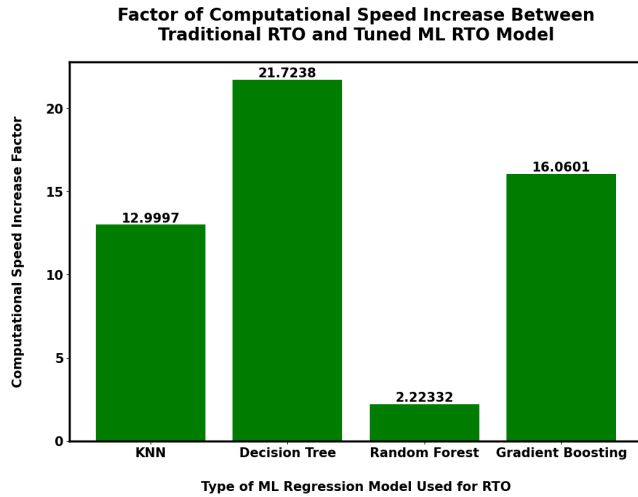


Figure 7. Factor of speed increase with each type of ML model

Figure 7 shows the speed increase (runtime decrease) relative to traditional RTO for the best-tuned model of each type. The results show that the decision tree model has the greatest speed advantage (21x faster than traditional RTO). However, when considering the balance of profit gain and high speed computation, I would recommend the gradient boosting regressor as the superior choice. The well-tuned gradient boosting model offered the greatest profit increase, while maintaining a considerable increase in speed relative to traditional optimization (16x faster).

Figure 8 given below displays the results of fine-tuning the number of estimators in the gradient boosting model. Earlier, 25 estimators was the optima out of the tested values. After testing the tuning parameters at a greater resolution, 24 estimators resulted in a slightly higher profit gain. However, the results do involve some inconsistency, as some runs still returned 25 estimators as the optimal value. Despite the slight inconsistency, I observed that the optimal number of estimators remains close to 24, generating a profit gain of at least 0.085% on most runs, while also offering a significant speed increase compared to traditional optimization.

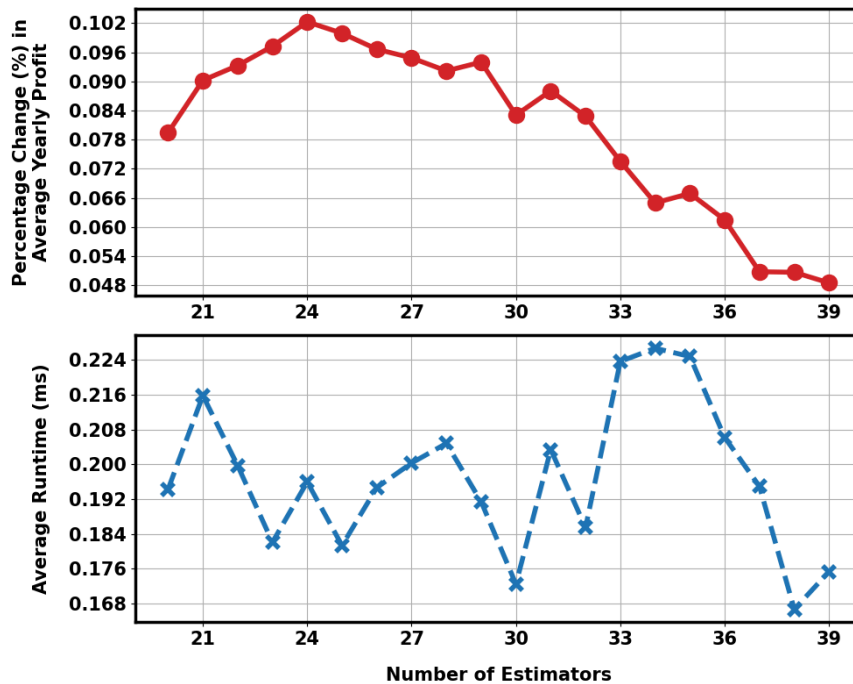


Figure 8. Results of fine-tuning number of estimators for gradient boosting ML RTO model

The output of statistics for the final optimally tuned ML RTO model is shown below in **Figure 9**.

Best tuned ML RTO model: Gradient Boosting Regressor with 24 estimators
Maximum yearly profit change compared to traditional RTO: gain of 0.102% (= \$11674.53)
Factor of speed increase compared to traditional RTO: 17.086

Figure 9. Output of metrics for best-tuned ML RTO model

As a final verification of performance for the best-tuned ML RTO model, I re-ran the reactor simulation and generated the dynamic plots observed earlier when running the same simulation with the traditional RTO algorithm instead. The results show a similar plot (see **Appendix H, Figure 28**) as the one observed earlier, which confirms that the tuned gradient boosting RTO model supports a stable dynamic response. It offers the benefits of speed and profit while not compromising on the stability of the dynamic response.

DISCUSSION

After completing the simulation code, I observed that ML RTO generally performed faster, on average, than traditional RTO across all the tested process and simulation conditions. Although the reported results show that all the tested models ran faster on average, it is important to note that on some runs of the code, the random forest models occasionally ran more slowly than traditional RTO that used the L-BFGS-B solver method. All the types of ML models other than random forests consistently exhibited significantly faster runtimes than traditional RTO. The specific extent to which the runtime decreased depended greatly on the type of ML model that was trained and the optimal tuning parameter value selected. For example, the best tuned decision tree model ran at least 21 times faster than traditional RTO, whereas the best tuned KNN model ran at least 12 times faster. For both types of models, these runtimes also fluctuated when different tuning parameter values were selected. While the factor of runtime decrease (i.e. speed increase) was different across the ML model types and hyperparameter settings, the consistent existence of a runtime decrease was true for all the tested models other than random forest models, proving that ML RTO shows promise as a faster computational proxy compared to traditional RTO.

Solidifying the conclusion that ML RTO shows promise for industrial application required additional benchmarking of the R^2 value and profit generation for each ML model. R^2 is a performance metric of prediction accuracy of a regression model. Most of the models had a high R^2 of at least 0.89, with well-tuned models exceeding an R^2 of 0.98. The generated profits from RTO also improved slightly for many well-tuned models compared to the traditional algorithm. These results indicated that ML shows promise in decreasing computational runtime while not compromising on prediction accuracy of optimal solutions, thereby not sacrificing the profit objective of RTO either. It is important to note, however, that profit may not solely be determined by model R^2 , as control dynamics can also play a role. Hence, it was important to also consider profit as the final performance measure, rather than solely the R^2 metric that

represents model accuracy but may not always lead to a higher final return on the optimization objective. Evaluating both measures and observing that neither were compromised, I concluded that ML does, indeed, show promise in decreasing the runtime without sacrificing the maximal fulfillment of the optimization objective. It is also important to note that this was only true for the tuned ML models, as poorly tuned or untuned models sometimes exhibited low R^2 and decreases in profit.

The speed advantage of ML models is a well-known phenomenon. However, I found the advantage of slightly greater profit generation in my simulation to be less straightforward. This result differed from my initial prediction, which was that ML RTO would generate slightly less profit than traditional RTO, as the latter calculates exact optimal while the former only uses approximations involving slight residual error.

One possible reason for the profit gain from using ML for RTO: the traditional RTO algorithm could have involved slight numerical inaccuracies and noise which cumulatively added into lower total profit being generated over time. To make the traditional RTO more accurate, I had optimized the number of initial guesses and had verified (using visualized curves of the objective function at selected time points) that the algorithm was returning the global maxima. However, it was not possible to rigorously verify this for every single iteration, as there were hundreds of them that occurred through the simulation time window. It is possible that some iterations were still not able to converge on the global maximum, leading to lower profit. The tuned ML models, though they were trained on this slightly inaccurate solutions set generated by traditional RTO, were able to smoothen the inaccuracies and noise of the training data to learn patterns needed to more effectively find the global maxima consistently. As a result, the tuned ML models could have calculated optima that were slightly more stable.

Making a concrete determination as to why the tuned ML RTO models generated slightly greater profit than traditional RTO was outside the scope of the completed project, which focused on a comparative evaluation of algorithms rather than an in-depth proof of the cause behind observed differences in performance. Further work is needed to examine and prove the exact reason why the tuned ML models have higher profit. It remains highly likely that the presented possibility of greater solution stability was the reason for the difference in profit. A recommended extension of this work would be to perform a rigorous analysis on the optima returned by each of the two RTO algorithms in each iteration, and concretely determine the reason behind the greater robustness of the ML model based on the results.

Though the exact reason for greater profit generation from the tuned ML models is yet to be proven concretely, the results of this project tentatively demonstrate great promise for ML RTO compared to a traditional RTO algorithm. The best model out of the ones tuned and tested was the Gradient Boosting Regressor ML RTO model that used 24 estimators. Compared to traditional RTO, it generated a 0.102% gain in yearly profit generated from the CSTR process and was at least 17 times faster. Although the percentage of profit gain was marginal, it translates to over \$11,600 of yearly profit gain in this specific simulation of a single CSTR unit. In real plant operation, small profit changes of this magnitude scaled across multiple units and sites has the potential to produce economically meaningful gains in profit. Even in the contrary scenario (if the profit gain remains insignificant), the results of this work show that there is at least a

concrete benefit to ML RTO in the area of computational speed. The speed benefit of ML can potentially support advantages in plant operation that were not directly simulated in this study, such as: reduced computational costs, greater robustness and safety of computerized plant operation, and greater profit generation from highly dynamic unsteady-state process units. Such indirect benefits were not demonstrated directly in this work and thus remain hypothetical; however, the direct speed advantage itself was concretely proven in the small-scale simulation of a CSTR. Furthermore, non-ideal behavior in real plants would further complicate the optimization objective, potentially increasing the factor of runtime decrease with the implementation of ML RTO. This conclusion, too, must be verified through a thorough digital twin study and remains hypothetical based on an extrapolation of the current results. As previously stated, the scope of this project was to demonstrate the concept of a measurable runtime decrease and to test a possible increase in profit generation achieved by using ML RTO on a small-scale, simple chemical process. Ultimately, the obtained results showing a significant runtime decrease and a slight profit gain from the best-tuned ML RTO model demonstrate that, compared to traditional RTO, machine learning RTO is a potentially advantageous technology for the optimization of chemical processes.

ACKNOWLEDGEMENTS

This project was completed under the supervision of Dr. Thomas Badgwell, who has expertise in the field of process control and optimization. Dr. Badgwell and I had regular check-in meetings to review progress on the program throughout the project duration of a few months. He was a supportive supervisor who gave the necessary guidance every step of the way in the process of completing this project. He gave constructive feedback on results obtained from program drafts, and always gave useful advice on how to proceed best with the project from one phase to the next. His background, knowledge and guidance was invaluable to the successful completion of this project.

REFERENCES

1. Belyadi, H., & Haghighat, A. (2021). *Machine learning guide for oil and gas using Python - a step-by-step breakdown with data, algorithms, codes and applications*.
<https://www.sciencedirect.com/science/chapter/monograph/pii/B9780128219294000044>
2. Byrd, R. H., Lu, P., Nocedal, J., & Zhu, C. (1994, May). *A limited memory algorithm for bound constrained optimization*. <https://users.iems.northwestern.edu/~nocedal/PDFfiles/limited.pdf>
3. DecisionTreeRegressor. (n.d.). In *Sklearn API reference*. Retrieved November 13, 2025, from <https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.html>
4. Gradient boosting regression. (n.d.). In *Sklearn examples*. Retrieved November 13, 2025, from https://scikit-learn.org/stable/auto_examples/ensemble/plot_gradient_boosting_regression.html
5. KNeighborsRegressor. (n.d.). In *Sklearn API reference*. Retrieved November 13, 2025, from <https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsRegressor.html>
6. Nonlinear programming. (n.d.). In *ScienceDirect*. Retrieved November 13, 2025, from <https://www.sciencedirect.com/topics/computer-science/nonlinear-programming>
7. RandomForestRegressor. (n.d.). In *Sklearn API reference*. Retrieved November 13, 2025, from <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html>
8. Rizkallah, L. W. (2025). Enhancing the performance of gradient boosting trees on regression problems. *Journal of Big Data*. <https://journalofbigdata.springeropen.com/articles/10.1186/s40537-025-01071-3>
9. Salman, H. A., Kalatech, A., & Steiti, A. (June 2024). Random forest algorithm overview. *Babylonian Journal of Machine Learning*.
https://www.researchgate.net/publication/382419308_Random_Forest_Algorithm_Overview
10. SciPy community. (2022, May 20). *SciPy reference guide* (Release 1.8.0 ed.). Retrieved November 13, 2025, from <https://docs.scipy.org/doc/scipy-1.8.0/scipy-ref-1.8.0.pdf>
11. Seborg, D. E., Edgar, T. F., Mellichamp, D. A., & Doyle, F. J. (2017). *Process dynamics and control* (4th ed.). Wiley.
12. Srisuradetchai, P., & Suksrikran, K. (2024, June 30). *Random kernel k-nearest neighbors regression*.
<https://www.frontiersin.org/journals/big-data/articles/10.3389/fdata.2024.1402384/full>

REFLECTION

This project, which simulates the dynamics of a CSTR, was a significant undertaking during the limited time of a single semester. As a graduating senior, it was a final culmination of the courses I have taken both in my degree and certificate programs. Through my undergraduate studies, I had gained the necessary background in the areas of chemical engineering, programming, data science, and machine learning to complete such an undertaking. However, applying the skillsets from different areas of study to produce one comprehensive project proved itself a challenging task; it taught me how to effectively apply computational methods to solve chemical engineering problems. I would like to extend my gratitude to my supervisor for his indispensable support and guidance throughout this challenging learning experience.

This research project presented an opportunity to select and conduct an in-depth study in a subject area that deeply interests me. As a result of my interest in the chosen topic, I spent a significant amount of effort pursuing it within the limited timeframe, which allowed me to learn about the subject matter beyond what my standard coursework had required me to know. Being the most complicated program I have written to date, the simulation code gave me extensive practice to improve my Python programming fundamentals. Additionally, it taught me how to apply computational methodologies to more effectively solve traditional chemical engineering problems. Already, the traditional RTO algorithm explored in the paper was an application of computation to improve chemical processes at the time of its advent. This study focused on applying the more recent computational technology of machine learning to the long-standing problem of optimizing a chemical process. Studying such an application taught me about the potential for computation in advancing fields of engineering, and further deepened my interest in applied computation as a relevant subject for the present-day world.

Appendices

APPENDIX A: PHYSICAL PARAMETERS

Table 1. Information about physical parameters used in reactor modelling equations

Parameter	Physical Meaning	Specification	Unit
C_A	Concentration of species A in the reactor	N/A*	$\frac{mol}{L}$
C_{Af}	Concentration of species A in the feed	<u>Piecewise function**</u> $0s \leq t < 200s \Rightarrow C_{Af} = 4$ $200s \leq t < 400s \Rightarrow C_{Af} = 2.5$ $400s \leq t < 600s \Rightarrow C_{Af} = 3.5$ $600s \leq t < 800s \Rightarrow C_{Af} = 3$ $800s \leq t < 1000s \Rightarrow C_{Af} = 2$	$\frac{mol}{L}$
C_B	Concentration of species B in the reactor	N/A*	$\frac{mol}{L}$
C_{Bf}	Concentration of species B in the feed	0	$\frac{mol}{L}$
r	Reaction rate	N/A*	$\frac{mol}{s \cdot L}$
E_a / R	Activation energy of reaction divided by the gas constant	5000	K (Kelvin)
T	Reactor temperature	N/A*	K (Kelvin)
k_0	Pre-exponential rate constant	<u>Gaussian distribution</u> $k_{0,mean} = 1.5E6 - 100t$ (where t is time in seconds) $k_{0,stdev} = 2E2$	$\frac{1}{s} \left(\frac{L}{mol} \right)^{0.9}$ See footnote for dimensional analysis***
k	Reaction rate constant	N/A*	$\frac{1}{s} \left(\frac{L}{mol} \right)^{0.9}$

ΔH_{rxn}	Enthalpy change of the reaction	-20,000	$\frac{J}{mol}$
Q	Feed flowrate	Manipulated by the PI controller	$\frac{L}{s}$
V	Reactor volume	20	L
ρ	Fluid density	1000	$\frac{g}{L}$
C_p	Specific heat capacity of the fluid at constant pressure	2.5	$\frac{J}{g \cdot K}$
T_f	Temperature of the reactor feed	350	K
T_c	Coolant temperature	300	K
UA	Product of the overall heat transfer coefficient and the heat transfer area	200,000	$\frac{J}{s \cdot K}$

Table Footnotes

*:

If a parameter specification is given as “N/A”, it indicates that the parameter changes dynamically throughout the simulation based on the state of the process

**:

For the estimation of yearly profit, the assumption was that this piecewise function repeats every 1000 seconds. However, since the chosen simulation timespan was 1000 seconds in the program, a mechanism for this pattern for larger timespan was not built into the code. ***Therefore, in the current simulation, the returned yearly profits are only valid if the simulation timespan is less than 1000 seconds.*** To support proper yearly profit estimations from the results returned by simulations of larger timespans, the program would need further modification.

Additionally, the assumption itself may be unrealistic in modelling real plant operation. Future work would benefit from a re-evaluation of the modelling assumption used for concentration of

species A in the feed to represent real operation more accurately. However, the yearly profit presented in the report are still valid metrics to estimate the average, given the specified assumption.

***:

Dimensional analysis was required to derive the units of the rate constant based on the rate equation. The calculation is shown below.

$$r = kC_A^{1.9}$$

$$\frac{mol}{s \cdot L} = [u_k] \cdot \left(\frac{mol}{L}\right)^{1.9}$$

where $[u_k]$ represents the unit of the rate constant k

$$[u_k] = \frac{mol}{s \cdot L} \left(\frac{L}{mol}\right)^{1.9}$$

$$[u_k] = \frac{1}{s} \left(\frac{L}{mol}\right)^{0.9}$$

APPENDIX B - IMPORT STATEMENTS

```
import time
import math
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.ticker as mticker
from matplotlib.ticker import MaxNLocator
from textwrap import wrap
import scipy as sp
from scipy.integrate import solve_ivp
from scipy.optimize import minimize
from scipy.optimize import fsolve
from scipy import stats
from scipy import integrate

import sklearn as sk
from sklearn.model_selection import train_test_split
from sklearn import neighbors
from sklearn.neighbors import KNeighborsRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from sklearn.svm import SVR
from sklearn.neural_network import MLPRegressor
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_error
```

APPENDIX C - SIMULATION FUNCTIONS

```
# Initialize all simulation parameters by passing in their values as arguments
def init_vars(args):
    Tc_setting = args[0] # K
    Tf = args[1] # K
    T0 = args[2] # K
    V = args[3] # L
    Qi = args[4] # L/s

    total_time = args[5] # s
    dt_control = args[6] # s
    dt_RTO = args[7] # s

    E_by_R = args[8] # K
    rho = args[9] # g/L
    Cp = args[10] # J/g/K
    dHr = args[11] # J/mol
    UA = args[12] # J/s/K

    Kc = args[13]
    tauI = args[14]

    dip1_magnitude = args[15]
    dip2_magnitude = args[16]
    num_guesses = args[17]

    rxn_order = args[18]

    algorithm = args[19]
    fixed_setpoint = args[20]

    return Tc_setting, Tf, T0, V, Qi, total_time, dt_control, dt_RTO, \
        E_by_R, rho, Cp, dHr, UA, Kc, tauI, \
        dip1_magnitude, dip2_magnitude, \
        num_guesses, rxn_order, algorithm, fixed_setpoint
```

```
# Get the concentration of A in the feed at any time point
```

```
def get_CAf(t):
```

```
    if t < 200:
```

```
        return 4.0
```

```
    elif t < 400:
```

```
        return 2.5
```

```
    elif t < 600:
```

```
        return 3.5
```

```
    elif t < 800:
```

```
        return 3.0
```

```
    elif t < 1000:
```

```
        return 2.0
```

```
    return 2.0
```

```
# Get the pre-exponential rate constant k0 at any given time, based on a transient  
mean and a normal distribution
```

```
def get_k0(t):
```

```
    k0_mean = 1.5e6 - t*1e2
```

```
    k0_stdev = 2e2
```

```
    k0 = np.random.normal(k0_mean,k0_stdev)
```

```
    return k0
```

```

# Define differential equations that model the reactor
def reactorODEs(t, sol, Q, V, k0, E_by_R, Tf, rho, Cp, dHr, UA, Tc, rxn_order):
    CAf = get_CAf(t)

    CA, CB, T = sol

    k = k0 * math.exp(-E_by_R / T)
    r = k * CA**rxn_order

    # MBs
    dCA_dt = (Q * (CAf - CA) - V * r) / V
    dCB_dt = (Q * (0 - CB) + V * r) / V
    # EB
    dT_dt = ( (Q * rho * Cp * (Tf - T)) + ((-dHr) * V * r) + (UA * (Tc - T)) ) \
        / (V * rho * Cp)

    return [dCA_dt, dCB_dt, dT_dt]

# Forces the ODE solver to avoid negative values, which are physically impossible
def negative_event(t, sol, Q, V, k0, E_by_R, Ti, rho, Cp, dHr, UA, Tc, rxn_order):
    return min(sol)

# Numerically solves for the steady state concentration of A
def calculate_CA_steady_state(CAf, V, k, Q, rxn_order):

    def ca_balance_root(CA_guess):
        return CAf - CA_guess - (V * k / Q) * CA_guess**rxn_order

    CA_guess_initial = CAf / 2.0
    CA_solution = fsolve(ca_balance_root, CA_guess_initial)[0]
    return max(0.0, CA_solution)

```

```

# Simulates the PI controller action (based on the velocity form of PI algorithm)
using the current setpoint error, previous setpoint error, controller gain and
integral time constant
def PI_controller(previous_error, error, Kc, tauI, dt):
    dP = Kc * (error - previous_error)
    dI = (Kc / tauI) * error * dt

    dQ = dP+dI # velocity form

    return dQ


# The optimization objective function to choose the optimal flowrate that optimizes
profit (optimal flowrate is then converted to optimal CB setpoint, which is passed
down to the controller)
def profit_objective(Qo, t, CAf, V, k, T, Tc_setting, Tmax, \
                    dip1_magnitude, dip2_magnitude, rxn_order):

    # solving for steady state CA
    CA_ss = calculate_CA_steady_state(CAf, V, k, Qo, rxn_order)
    r_ss = k * CA_ss*rxn_order
    Qo = max(Qo, 1e-6)
    CBo = (V * r_ss) / Qo
    # finding production rate
    production_rate = CBo * Qo

    # inerts that need to be pretreated
    CIf = 0.1 * CAf

    # base prices
    product_price_base = 300
    feed_cost_base = 25
    waste = 50

    # revenue term
    T_revenue_penalty = 1.0 - 0.05 * np.exp(0.05 * (T - 380)) \
        if T > 380 else 1.0
    revenue = product_price_base * production_rate * T_revenue_penalty

```

```

# utility cost
utility_cost = Qo * (50 + 100 * (T - Tc_setting) + 10 * Qo**2)

# pretreatment cost
feed_total_cost = feed_cost_base * CAf * Qo
pretreatment_cost = 50 * (Cif * Qo)

# temperature penalty
temp_penalty = 50 * np.exp(0.01 * k) * max(0, T - 385)

# nonideal mixing penalty
Q_ideal = 1.3
Q_max_penalty = 3.0
flowrate_penalty = 40 * ((Qo - Q_ideal) / Q_max_penalty)**3

conversion = 1 - CA_ss / CAf
# Penalty for low conversion
penalty_low_conversion = 100 * np.exp(-5 * conversion)
# Penalty for high conversion
conversion_target = 0.90 # 90%
penalty_high_conversion = \
    500 * max(0, conversion - conversion_target)**2

# high T maintenance penalty
temperature_margin = T - Tmax
maintenance_penalty = 500 * np.exp(2.5 * temperature_margin) \
    if temperature_margin > 0 else 0

# local minima
Q_dip_center1 = 0.7
dip_width1 = 0.02
localized_penalty1 = dip1_magnitude * \
    np.exp(-((Qo - Q_dip_center1)**2) / (2 * dip_width1**2))
Q_dip_center2 = 1.65
dip_width2 = 0.05
localized_penalty2 = dip2_magnitude * \
    np.exp(-((Qo - Q_dip_center2)**2) / (2 * dip_width2**2))

# fouling
fouling_penalty = 50*max(0,Qo-2)**3

```

```
# profit
P = (revenue
      - feed_total_cost
      - utility_cost
      - pretreatment_cost
      - penalty_low_conversion
      - penalty_high_conversion
      - flowrate_penalty
      - temp_penalty
      - maintenance_penalty
      - localized_penalty1
      - localized_penalty2
      - fouling_penalty) / 1000

return -P
```

```

# Main function that simulates the CSTR (dynamic response of controller and RTO)
def reactor_simulator(Tc_setting, Tf, T0, V, E_by_R, rho, Cp, dHr, UA, Qi, \
                      Kc, tauI, total_time, dt_control, dt_RTO, \
                      algorithm, fixed_setpoint, \
                      dip1_magnitude, dip2_magnitude, \
                      num_guesses, rxn_order, MLmodel):

    time_points_control = np.round(np.arange(0,
                                              total_time + dt_control,
                                              dt_control),
                                   decimals=4)

    num_steps_control = len(time_points_control)
    time_points_RTO = np.round(np.arange(0,
                                          total_time + dt_RTO,
                                          dt_RTO),
                               decimals=2)

    CAf = get_CAf(0) # get CAf at t=0

    IC = np.array([CAf, 0, T0])
    Q = Qi
    T = T0
    I = 0
    Qmin = 0.1
    Qmax = 3
    Tmax = 400
    CB_setpoint = 0.01
    CB_setpoints_array = [CB_setpoint]

    errors_array = [0]
    profit_array = []
    concs_and_temp = [[CAf, 0, T0]]
    flowrates = [Q]
    k_values = []
    placeholder = np.zeros(len(time_points_RTO))

    rto_runtime_array = []
    total_rto_runtime = 0.0
    rto_times_set = set(np.round(time_points_RTO[1:], 8))

```

```

generated_data = []

for i in range(1, num_steps_control):

    t = time_points_control[i]
    t_start_ODE = time_points_control[i-1]
    t_end_ODE = t

    CAf = get_CAf(t)
    k0 = get_k0(t)

    ODE_sol = solve_ivp(reactorODEs, (t_start_ODE, t_end_ODE), IC,
                        args=(Q, V, k0, E_by_R, Tf, rho, Cp, dHr,
                              UA, Tc_setting, rxn_order),
                        events=negative_event,
                        dense_output=True, max_step=0.01)
    CA, CB, T = ODE_sol.y[:, -1]
    k = k0 * math.exp(-E_by_R / T)
    concs_and_temp += [[CA, CB, T]]
    flowrates += [Q]

    previous_error = errors_array[-1]
    error = CB - CB_setpoint
    errors_array.append(error)

    # control algo
    dQ = PI_controller(previous_error, error, Kc, tauI, dt_control)
    # actuator limits for dQ
    if dQ > dt_control/10:
        dQ = dt_control/10
    elif dQ < -dt_control/10:
        dQ = -dt_control/10
    else:
        None

    # update flowrate
    Q += dQ
    Q = np.clip(Q, Qmin, Qmax) # valve limits
    IC = np.array([max(0, CA), max(0, CB), max(0, T)])
    # update IC for next control iteration

    if np.round(t, 8) in rto_times_set:

```

```

flowrate_bounds = [(Qmin, Qmax)]
RTO_time_index = np.where(np.isclose(time_points_RTO, t,
                                     atol=1e-6))[0][0]

if algorithm in ["Traditional RTO", "Traditional RTO - Generate Data"] \
    and not fixed_setpoint:

    initial_guesses = np.linspace(Qmin, Qmax, num_guesses)

    best_profit = -np.inf
    Q_optimal = 0.1

    '''
    RTO Start
    '''
    rto_start = time.perf_counter()

    for guess in initial_guesses:

        optimal_solution = minimize(profit_objective, guess,
                                   args = (t, CAf, V, k,
                                           T, Tc_setting, Tmax,
                                           dip1_magnitude,
                                           dip2_magnitude,
                                           rxn_order),
                                   bounds = flowrate_bounds,
                                   method='L-BFGS-B')

        if optimal_solution.success:
            current_profit = -optimal_solution.fun

            if current_profit > best_profit:
                best_profit = current_profit
                Q_optimal = float(optimal_solution.x[0])

    if Q_optimal > 1e-6:
        CA_ss = calculate_CA_steady_state(CAf, V, k, Q_optimal, rxn_order)
        r_ss = k * CA_ss**rxn_order
        CB_setpoint = (V * r_ss) / Q_optimal

```

```

else:
    CB_setpoint = CAf

    rto_end = time.perf_counter()
    '''
RTO end
'''

    CB_setpoints_array += [CB_setpoint]

if algorithm == "Traditional RTO - Generate Data":

    generated_data.append({
        'time': t,
        'rate_constant': k,
        'T_reactor': T,
        'CAf_feed': CAf,
        'Tc_set': Tc_setting,
        'Q_optimal': Q_optimal
    })

elif algorithm == "ML-Based RTO" and not fixed_setpoint:

    rto_start = time.perf_counter()

    features = np.array([[t, k, T, CAf, Tc_setting]])
    Q_optimal = MLmodel.predict(features)[0]
    Q_optimal = np.clip(Q_optimal, Qmin, Qmax)

    CA_ss = calculate_CA_steady_state(CAf, V, k,
                                      Q_optimal, rxn_order)

    r_ss = k * CA_ss**rxn_order
    CB_setpoint = (V * r_ss) / Q_optimal

    rto_end = time.perf_counter()

    CB_setpoints_array += [CB_setpoint]

elif fixed_setpoint:

```

```

    rto_start = time.perf_counter()
    CB_setpoint = fixed_setpoint
    rto_end = time.perf_counter()

    CB_setpoints_array += [CB_setpoint]

    rto_step_time = rto_end - rto_start
    rto_runtime_array.append(rto_step_time)
    total_rto_runtime += rto_step_time

    P_actual = -profit_objective(Q, t, CAf, V, k,
                                T, Tc_setting, Tmax,
                                dip1_magnitude,
                                dip2_magnitude,
                                rxn_order)
    profit_array.append(P_actual)

    concs_and_temp = np.array(concs_and_temp)
    avg_rto_runtime = sum(rto_runtime_array) / len(rto_runtime_array) \
        if rto_runtime_array else 0.0

    return time_points_control, time_points_RTO, CB_setpoints_array, \
        flowrates, concs_and_temp, generated_data, profit_array, \
        rto_runtime_array, avg_rto_runtime, total_rto_runtime

```

```

# Integrates the curve of instantaneous profit over time to return the total profit
earned over the simulation timespan
def integrate_cumulative_profit(time_points, profit_array):
    time_points = np.array(time_points)
    profit_array = np.array(profit_array)

    cumulative_profit = \
        integrate.cumulative_trapezoid(profit_array,
                                        time_points,
                                        initial=0)

    total_profit = cumulative_profit[-1]
    return cumulative_profit, total_profit


# Plots the dynamic response of the controller and optimizer, which are returned from
the main simulation function
def simulation_plot(time_points_control, time_points_RTO, CB_setpoints_array, \
                   flowrates, concentrations, profit_array):

    fig = plt.figure(figsize=(12,8))
    plt.tight_layout()
    plt.style.use('default')

    plt.subplot(3,1,1)
    plt.title("Controlled Variable - Concentration of B",
              fontweight='bold', fontsize=15, pad=15)
    plt.plot(time_points_control[1:], concentrations[1:,1],
              label = "CV: Concentration of B (mol/L)", linewidth=4)
    plt.plot(time_points_RTO, CB_setpoints_array, '--',
              label = "Setpoint Concentration of B (mol/L)", linewidth=3)
    plt.ylabel("\n".join(wrap("Concentration of B (mol/L)",15)),
              fontweight='bold', fontsize=12)
    plt.legend()
    plt.grid(True)

    ax1 = plt.gca()
    ax1.tick_params(axis='both', which='major', labels=10)
    plt.setp(ax1.get_xticklabels(), fontweight='bold')
    plt.setp(ax1.get_yticklabels(), fontweight='bold')

```


Plots the RTO objective function of profit vs feed flowrate at one specified time point of the simulation

```
def plot_RTO_profit_vs_Q(t, V, k, T, Tc_setting, Tmax, \
                        dip1_magnitude, dip2_magnitude, \
                        rxn_order, Qmin=0.1, Qmax=3, \
                        n_points=200, n_guesses=50):

    Q_grid = np.linspace(Qmin, Qmax, n_points)
    CAf = get_CAf(t)

    profits = [-profit_objective(Q, t, CAf, V, k,
                                T, Tc_setting, Tmax,
                                dip1_magnitude, dip2_magnitude,
                                rxn_order) for Q in Q_grid]

    flowrate_bounds = [(Qmin, Qmax)]

    guesses = list(np.linspace(Qmin, Qmax, n_guesses))
    initial_guesses = np.unique(guesses)
    best_profit = -np.inf

    Q_opt = 1
    for guess in initial_guesses:
        opt_result = minimize(profit_objective, x0=guess,
                              args=(t, CAf, V, k,
                                    T, Tc_setting, Tmax,
                                    dip1_magnitude, dip2_magnitude,
                                    rxn_order),
                              bounds=flowrate_bounds,
                              options={'ftol': 1e-8,
                                       'maxiter': 1000})

        if opt_result.success:
            current_profit = -opt_result.fun
            if current_profit > best_profit:
                best_profit = current_profit
                Q_opt = float(opt_result.x[0])
            else:
                None

    profit_opt = best_profit
    fig = plt.figure(figsize=(8,5))
    plt.plot(Q_grid, profits, label='Profit vs. Q',
             linewidth=4)
```

```

plt.axvline(Q_opt, color='red', linestyle='--', linewidth=4,
            label=f'Optimal Q = {Q_opt:.3f} L/s')
plt.scatter([Q_opt], [profit_opt], color='red', zorder=5, s=100)
plt.xlabel("Q (L/s)", fontweight='bold', fontsize=15, labelpad=15)
plt.ylabel("Instantaneous Profit ($)",
            fontweight='bold', fontsize=15, labelpad=15)
plt.title(f'RTO Profit Curve at t={t:.2f} s',
          fontweight='bold', fontsize=17, pad=20)
plt.legend()
plt.grid(True)

ax = plt.gca()
ax.tick_params(axis='both', which='major', labelsize=15)
plt.setp(ax.get_xticklabels(), fontweight='bold')
plt.setp(ax.get_yticklabels(), fontweight='bold')
for side in ['top', 'bottom', 'left', 'right']:
    ax.spines[side].set_linewidth(2.5)

return fig

```

```
# Iteratively runs the reactor simulation with traditional RTO for various numbers of
initial guesses, and visualizes the results
```

```
def tune_RTO_guesses(args, guess_range, n_trials=10):
```

```
    avg_rto_runtime_avgacross_trials = []
```

```
    yearly_traditional_profit_avgacross_trials = []
```

```
    Tc_setting, Tf, T0, V, Qi, total_time, dt_control, dt_RTO, E_by_R, \
    rho, Cp, dHr, UA, Kc, tauI, dip1_magnitude, dip2_magnitude, \
    num_guesses, rxn_order, algorithm, fixed_setpoint = init_vars(args)
```

```
    # tuning number of guesses for RTO
```

```
    for num_guesses in guess_range:
```

```
        avg_rto_runtimes = []
```

```
        yearly_traditional_profits = []
```

```
        for i in range(n_trials):
```

```
            time_points_control, _, _, _, _, profit_array, \
```

```
            rto_runtime_array, avg_rto_runtime, _ = \
```

```
                reactor_simulator(Tc_setting, Tf, T0, V, E_by_R, rho,
                                   Cp, dHr, UA, Qi, Kc, tauI,
                                   total_time, dt_control, dt_RTO,
                                   algorithm, fixed_setpoint,
                                   dip1_magnitude, dip2_magnitude,
                                   num_guesses, rxn_order, None)
```

```
            # avg runtime
```

```
            avg_rto_runtimes.append(avg_rto_runtime*1000) # s to ms
```

```
            # profit
```

```
            cumulative_profit, total_profit = \
```

```
                integrate_cumulative_profit(time_points_control[1:], profit_array)
```

```
            yearly_traditional_profit = \
```

```
                total_profit * (365*24*3600 / total_time) / 1000000
```

```
            yearly_traditional_profits.append(yearly_traditional_profit)
```

```
    # average the runtime and profit across trials
```

```
    avg_runtime = np.average(avg_rto_runtimes)
```

```
    avg_profit = np.average(yearly_traditional_profits)
```

```
    avg_rto_runtime_avgacross_trials.append(avg_runtime)
```

```
    yearly_traditional_profit_avgacross_trials.append(avg_profit)
```

```

print(f"Guesses={num_guesses}: Avg Profit=${\
    avg_profit:.4f}M, Avg Runtime={avg_runtime:.2f}ms")

plt.figure(figsize=(10, 8))

# plot profit
ax1 = plt.subplot(2, 1, 1)
plt.plot(guess_range, yearly_traditional_profit_avgacross_trials,
         color='tab:red', marker='o',
         markersize = 10, markeredgewidth = 4,
         label='Profit', linewidth=4)
plt.title("\n".join(wrap("RTO Performance vs. Number of Initial Guesses", 50)),
         fontweight='bold', fontsize=17, pad=20)
plt.ylabel("\n".join(wrap('Average Yearly Profit (Million $)', 20)),
         fontweight='bold', fontsize=15, labelpad=15)
ax1.xaxis.set_major_locator(mticker.MaxNLocator(integer=True))
plt.grid(True)

ax1.tick_params(axis='both', which='major', labelsize=13)
plt.setp(ax1.get_xticklabels(), fontweight='bold')
plt.setp(ax1.get_yticklabels(), fontweight='bold')
for side in ['top', 'bottom', 'left', 'right']:
    ax1.spines[side].set_linewidth(2.5)

# plot runtime
ax2 = plt.subplot(2, 1, 2)
plt.plot(guess_range, avg_rto_runtime_avgacross_trials,
         color='tab:blue', marker='x',
         markersize = 10, markeredgewidth = 4,
         linestyle='--',
         label='Runtime', linewidth=4)
plt.ylabel('Average Runtime (ms)',
         fontweight='bold', fontsize=15, labelpad=15)
plt.xlabel('Number of Initial Guesses',
         fontweight='bold', fontsize=15, labelpad=15)
ax2.xaxis.set_major_locator(mticker.MaxNLocator(integer=True))
plt.grid(True)

ax2.tick_params(axis='both', which='major', labelsize=13)
plt.setp(ax2.get_xticklabels(), fontweight='bold')

```

```
plt.setp(ax2.get_yticklabels(), fontweight='bold')
for side in ['top', 'bottom', 'left', 'right']:
    ax2.spines[side].set_linewidth(2.5)

plt.tight_layout()

# results dataframe
results_df = pd.DataFrame({
    'num_guesses': guess_range,
    'avg_profit_million': yearly_traditional_profit_avgacross_trials,
    'avg_runtime_s': avg_rto_runtime_avgacross_trials
})

return results_df
```

```

# Main function to train, test and tune ML RTO models
def analyze_ml_performance(args, ml_tuning_params, \
                           n_iterations_traditional_RTO, \
                           tuning_iterations):

    # init vars
    Tc_setting, Tf, T0, V, Qi, total_time, dt_control, dt_RTO, \
    E_by_R, rho, Cp, dHr, UA, Kc, tauI, dip1_magnitude, dip2_magnitude, \
    num_guesses, rxn_order, algorithm, fixed_setpoint = init_vars(args)

    # profit and runtime for traditional RTO
    traditional_RTO_profits = []
    traditional_RTO_runtimes = []

    for iteration in range(n_iterations_traditional_RTO):
        # running traditional RTO for profit and runtime data
        _, _, _, _, _, generated_data, profit_array, \
        rto_runtime_array, avg_rto_runtime, _ = \
        reactor_simulator(Tc_setting, Tf, T0, V, E_by_R, rho, Cp, dHr, UA, Qi,
                           Kc, tauI, total_time, dt_control, dt_RTO,
                           algorithm, fixed_setpoint,
                           dip1_magnitude, dip2_magnitude, num_guesses,
                           rxn_order, None)

        _, total_profit_trad = \
        integrate_cumulative_profit(np.arange(dt_control,
                                                total_time + dt_control,
                                                dt_control),
                                    profit_array)

        yearly_profit_trad = \
        total_profit_trad * (365*24*3600 / total_time)

        traditional_RTO_profits.append(yearly_profit_trad)
        traditional_RTO_runtimes.append(avg_rto_runtime)

    avg_trad_profit = np.mean(traditional_RTO_profits)
    avg_trad_runtime_ms = np.mean(traditional_RTO_runtimes) * 1000

    # generate training data for ML
    algorithm = "Traditional RTO - Generate Data"
    _, _, _, _, _, generated_data, profit_array, \
    rto_runtime_array, avg_rto_runtime, _ = \

```

```

    reactor_simulator(Tc_setting, Tf, T0, V, E_by_R, rho, Cp,
                      dHr, UA, Qi, Kc, tauI,
                      total_time, dt_control, dt_RTO,
                      algorithm, fixed_setpoint,
                      dip1_magnitude, dip2_magnitude,
                      num_guesses, rxn_order, None)

generated_data_df = pd.DataFrame(generated_data)
X_base = np.array(generated_data_df[['time', 'rate_constant', \
                                     'T_reactor', 'CAF_feed', 'Tc_set']])
y_base = generated_data_df['Q_optimal']

# hyperparameter tuning
model_results = {}

for model_type, tuning_list in ml_tuning_params.items():

    model_specific_results = []

    for tuning_param in tuning_list:
        ML_RTO_profits = []
        ML_RTO_runtimes = []
        ML_RTO_R2 = []

        for run in range(tuning_iterations):

            X_train, X_test, y_train, y_test = \
                train_test_split(X_base, y_base,
                                test_size=0.2,
                                random_state=42+run,
                                shuffle=True)

            if model_type == "KNN":
                MLmodel = \
                    KNeighborsRegressor(n_neighbors=tuning_param).fit(X_train,
                                                                    y_train)

            elif model_type == "Decision Tree":
                MLmodel = \
                    DecisionTreeRegressor(max_depth=tuning_param).fit(X_train,
                                                                    y_train)

            elif model_type == "Random Forest":
                MLmodel = \

```

```

        RandomForestRegressor(n_estimators=tuning_param,
                               random_state=42).fit(X_train,
                                                       y_train)

elif model_type == "Gradient Boosting":
    MLmodel = \
        GradientBoostingRegressor(n_estimators=tuning_param,
                                   learning_rate=0.1,
                                   max_depth=5,
                                   random_state=42).fit(X_train,
                                                         y_train)

# metrics
y_pred = MLmodel.predict(X_test)
R2 = r2_score(y_test, y_pred)
ML_RTO_R2.append(R2)

# ML RTO run
algorithm = "ML-Based RTO"
_, _, _, _, _, profit_array_ML, \
    rto_runtime_array_ML, avg_rto_runtime_ML, _ = \
    reactor_simulator(Tc_setting, Tf, T0, V, E_by_R,
                      rho, Cp, dHr, UA, Qi, Kc, tauI,
                      total_time, dt_control, dt_RTO,
                      algorithm, fixed_setpoint,
                      dip1_magnitude, dip2_magnitude,
                      num_guesses, rxn_order, MLmodel)

_, total_profit_ML = \
    integrate_cumulative_profit(np.arange(dt_control,
                                           total_time + dt_control,
                                           dt_control),
                                profit_array_ML)

yearly_profit_ML = \
    total_profit_ML * (365*24*3600 / total_time)

ML_RTO_profits.append(yearly_profit_ML)
ML_RTO_runtimes.append(avg_rto_runtime_ML)

# average profit, runtime, R2 with a particular tuning setting
avg_ml_profit = np.mean(ML_RTO_profits)
avg_ml_runtime_ms = np.mean(ML_RTO_runtimes) * 1000
avg_ml_r2 = np.mean(ML_RTO_R2)

```

```

# profit change (ML profit - traditional profit)
profit_change_diff = \
    avg_ml_profit - avg_trad_profit
# profit change percentage
profit_change_percent = \
    100 * (avg_ml_profit - avg_trad_profit) / avg_trad_profit

# results
model_specific_results.append({
    'Hyperparameter_Value': tuning_param,
    'R2_Mean': avg_ml_r2,
    'Profit_Mean_ML': avg_ml_profit,
    'Profit_Mean_Trad': avg_trad_profit,
    'Profit_Change_($)': profit_change_diff,
    'Profit_Change_%': profit_change_percent,
    'Runtime_Mean_ML_ms': avg_ml_runtime_ms,
    'Runtime_Mean_Trad_ms': avg_trad_runtime_ms
})

model_results[model_type] = \
    pd.DataFrame(model_specific_results)
# results dataframe for one model type

return model_results, avg_trad_profit, avg_trad_runtime_ms

```

```

# Plots the results of ML RTO model tuning
def plot_ML_tuning_results(model_results, tuning_parameter_name):
    tuning_parameter_values = model_results['Hyperparameter_Value']
    model_profit_change = model_results['Profit_Change_%']
    model_runtime = model_results['Runtime_Mean_ML_ms']

    plt.figure(figsize=(10, 8))

    # plot profit
    plt.subplot(2, 1, 1)
    plt.plot(tuning_parameter_values, model_profit_change,
             color='tab:red', marker='o', label='Profit',
             markersize=10, markeredgewidth=4,
             linewidth=4)

    plt.ylabel("\n".join(wrap('Percentage Change (%) in Average Yearly Profit',25)),
              fontsize=15, fontweight='bold', labelpad=15)
    plt.grid(True)

    ax1 = plt.gca()
    ax1.tick_params(axis='both', which='major', labelsize=15)
    ax1.yaxis.set_major_locator(MaxNLocator(integer=True))
    ax1.xaxis.set_major_locator(MaxNLocator(integer=True))
    plt.setp(ax1.get_xticklabels(), fontweight='bold')
    plt.setp(ax1.get_yticklabels(), fontweight='bold')
    for side in ['top', 'bottom', 'left', 'right']:
        ax1.spines[side].set_linewidth(2.5)

    # plot runtime
    plt.subplot(2, 1, 2)
    plt.plot(tuning_parameter_values, model_runtime,
             color='tab:blue', marker='x',
             markersize=10, markeredgewidth=4,
             linestyle='--', linewidth=4,
             label='Runtime')

    plt.ylabel('Average Runtime (ms)',
              fontsize=15, fontweight='bold', labelpad=15)
    plt.xlabel(tuning_parameter_name,
              fontsize=15, fontweight='bold', labelpad=15)
    plt.grid(True)

```

```
ax2 = plt.gca()
ax2.tick_params(axis='both', which='major', labelsize=15)
ax2.yaxis.set_major_locator(MaxNLocator(integer=True))
ax2.xaxis.set_major_locator(MaxNLocator(integer=True))
plt.setp(ax2.get_xticklabels(), fontweight='bold')
plt.setp(ax2.get_yticklabels(), fontweight='bold')
for side in ['top', 'bottom', 'left', 'right']:
    ax2.spines[side].set_linewidth(2.5)

plt.tight_layout()
```

APPENDIX D - TRADITIONAL RTO

```
args = [300,350,350,20,1,
        1000,0.1,5,
        5000,1000,2.5,-20000,200000,
        5,10,
        100,120,2,
        1.9,
        "Traditional RTO - Generate Data",False]

Tc_setting, Tf, T0, V, Qi, total_time, dt_control, dt_RTO, \
E_by_R, rho, Cp, dHr, UA, Kc, tauI, \
dip1_magnitude, dip2_magnitude, \
num_guesses, rxn_order, algorithm, fixed_setpoint = init_vars(args)

time_points_control, time_points_RTO, CB_setpoints_array, \
flowrates, concentrations, generated_data, profit_array, \
rto_runtime_array, avg_rto_runtime, total_rto_runtime = \
    reactor_simulator(Tc_setting, Tf, T0, V, E_by_R, rho, Cp, dHr,
                      UA, Qi, Kc, tauI, total_time, dt_control, dt_RTO,
                      algorithm, fixed_setpoint, dip1_magnitude, dip2_magnitude,
                      num_guesses, rxn_order, None)
```

```

print(f"Average Runtime = {avg_rto_runtime*1000:.2f}ms")

cumulative_profit, total_profit = \
    integrate_cumulative_profit(time_points_control[1:], profit_array)
yearly_traditional_profit = \
    total_profit * (365*24*3600 / total_time) / 1000000
print(f"Yearly Profit = ${yearly_traditional_profit:.4f}M")

```

<p>Average Runtime = 3.32ms</p> <p>Yearly Profit = \$11.4155M</p>

Figure 10. Average runtime and yearly profit generated from CSTR simulation when using the traditional RTO algorithm

```

p = simulation_plot(time_points_control, time_points_RTO, CB_setpoints_array,
    flowrates, concentrations, profit_array)

```

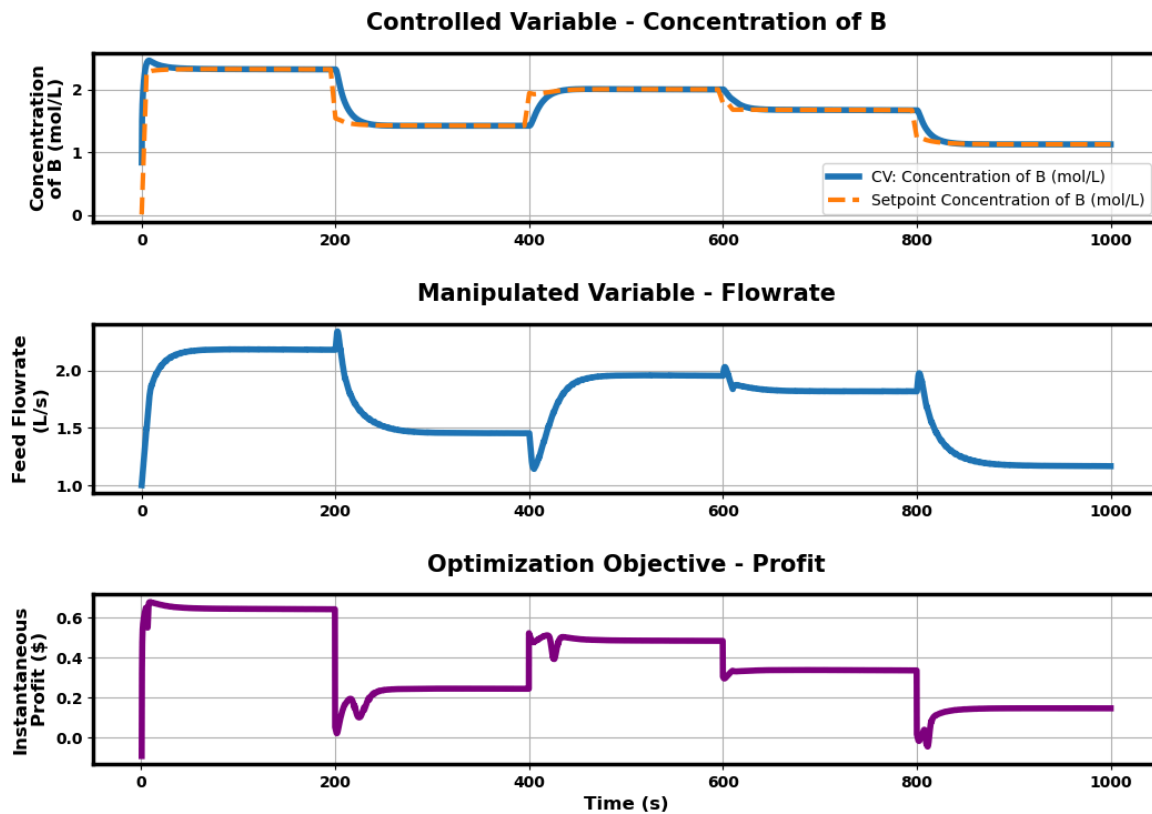


Figure 11. Plots of dynamic responses with the optimization algorithm set to traditional RTO for the CSTR simulation (repeat of Figure 2)

APPENDIX E - VERIFICATION OF RTO

```
args = [300,350,350,20,1,
        1000,0.1,5,
        5000,1000,2.5,-20000,200000,
        5,10,
        100,120,2,
        1.9,
        "Traditional RTO - Generate Data",False]

Tc_setting, Tf, T0, V, Qi, total_time, dt_control, dt_RTO, \
E_by_R, rho, Cp, dHr, UA, Kc, tauI, \
dip1_magnitude, dip2_magnitude, \
num_guesses, rxn_order, algorithm, fixed_setpoint = init_vars(args)

time_points_control, time_points_RTO, CB_setpoints_array, \
flowrates, concentrations, generated_data, profit_array, \
rto_runtime_array, avg_rto_runtime, total_rto_runtime = \
    reactor_simulator(Tc_setting, Tf, T0, V, E_by_R, rho, Cp, dHr,
                      UA, Qi, Kc, tauI, total_time, dt_control, dt_RTO,
                      algorithm, fixed_setpoint, dip1_magnitude, dip2_magnitude,
                      num_guesses, rxn_order, None)

Tmax=400

if isinstance(generated_data, list):
    generated_data = pd.DataFrame(generated_data)

t_array = [1,10,150,200,250,500,750,900]
for t in t_array:
    RTO_index = np.argmin(np.abs(time_points_RTO - t))
    k = generated_data.loc[RTO_index, 'rate_constant']
    control_idx = np.argmin(np.abs(time_points_control - t))
    T = concentrations[control_idx, 2]

    objective_plot = \
        plot_RTO_profit_vs_Q(t, V, k, T, Tc_setting, Tmax,
                              dip1_magnitude, dip2_magnitude,
                              rxn_order)
```

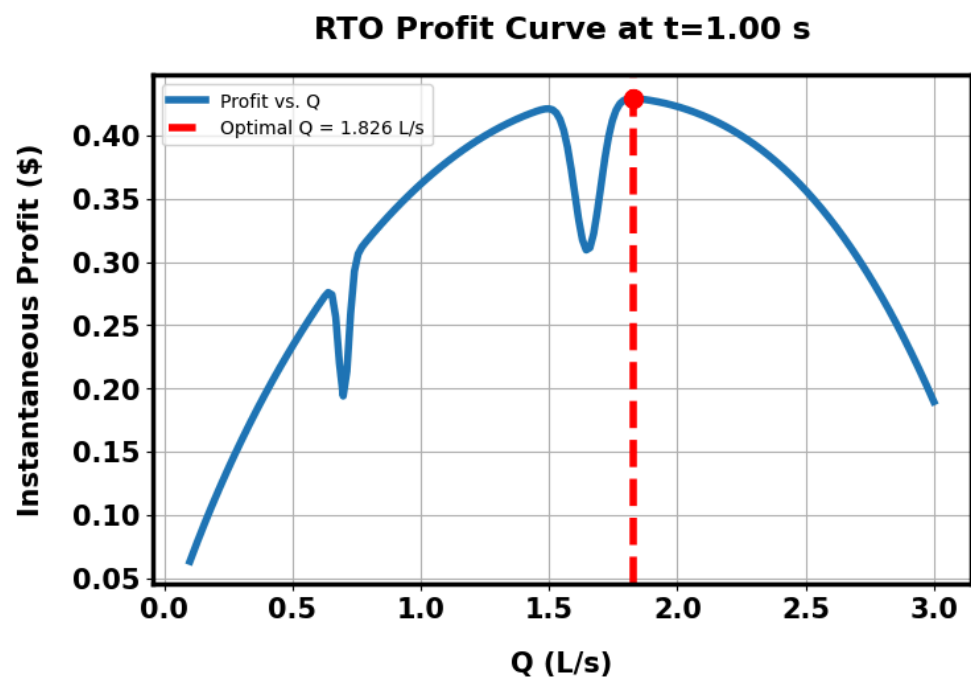


Figure 12. Plot of profit objective vs. feed flowrate at the simulation time point of 1 second

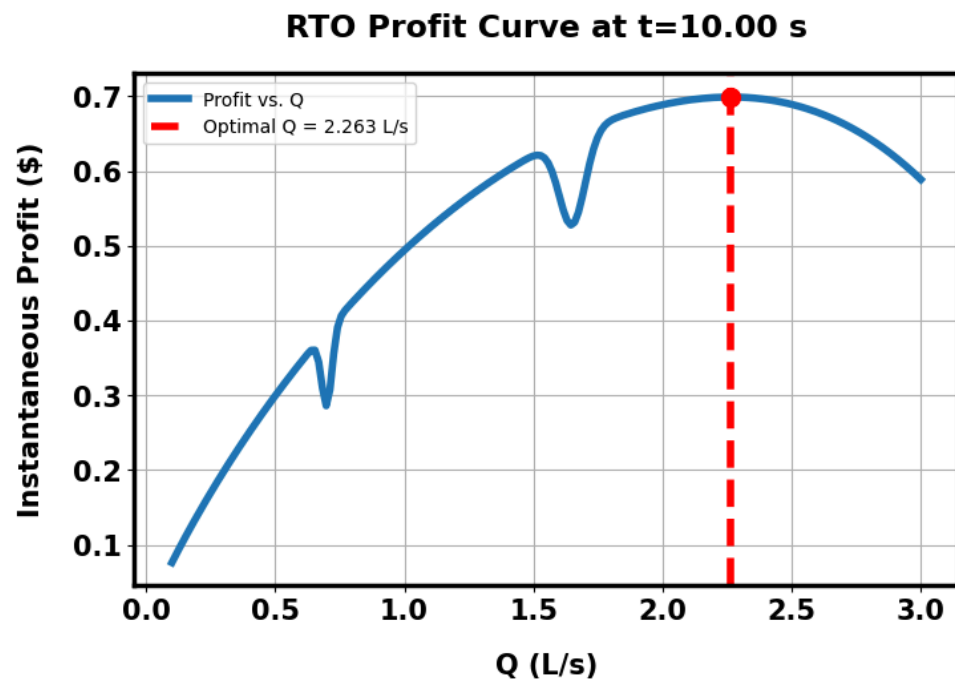


Figure 13. Plot of profit objective vs. feed flowrate at the simulation time point of 10 seconds

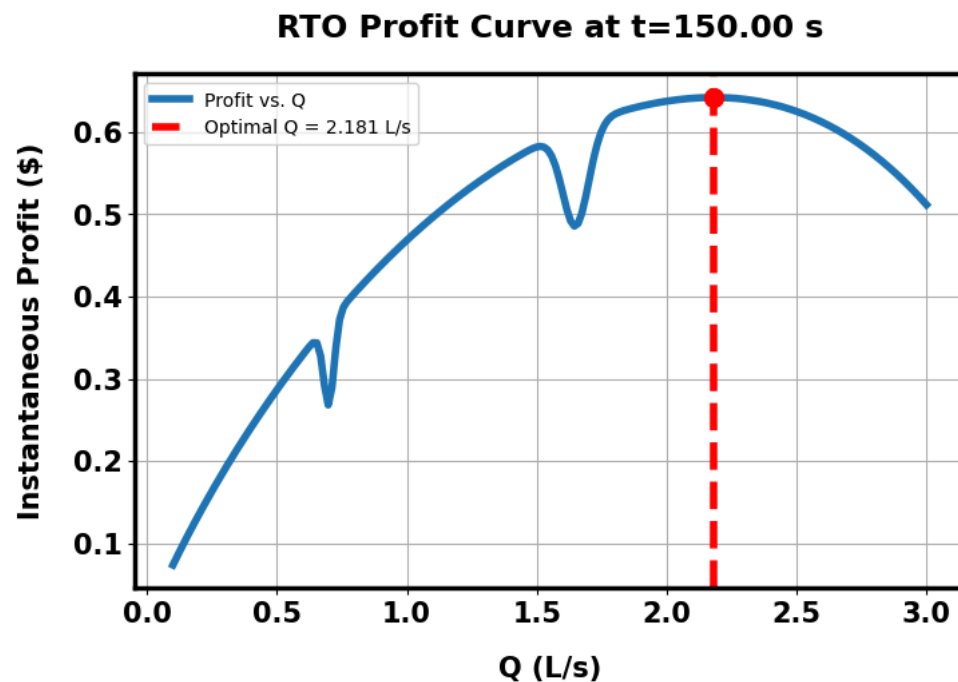


Figure 14. Plot of profit objective vs. feed flowrate at the simulation time point of 150 seconds

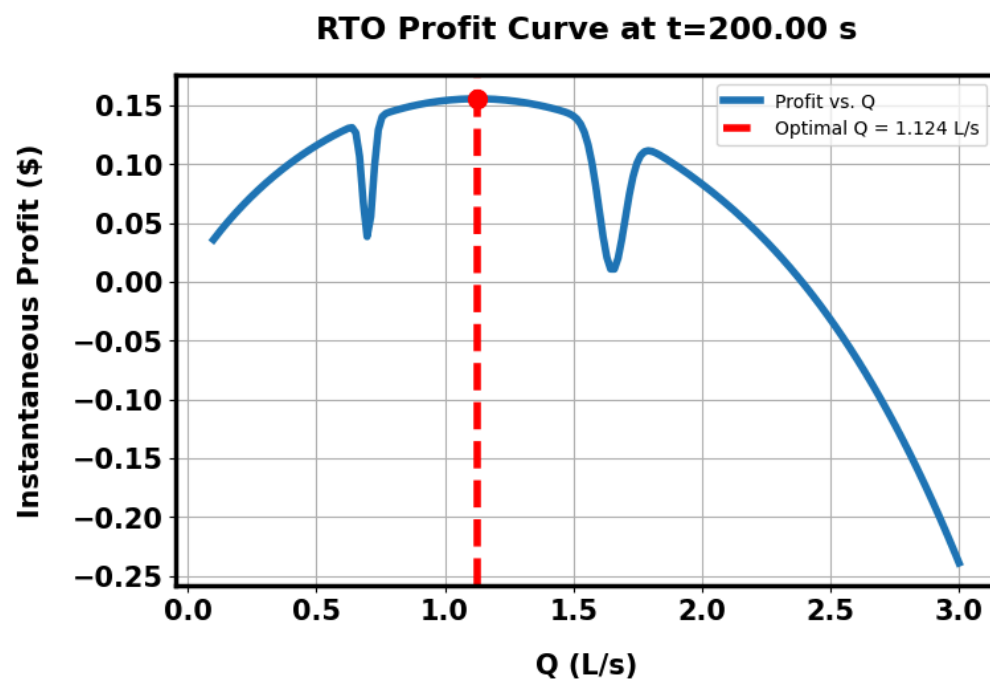


Figure 15. Plot of profit objective vs. feed flowrate at the simulation time point of 200 seconds

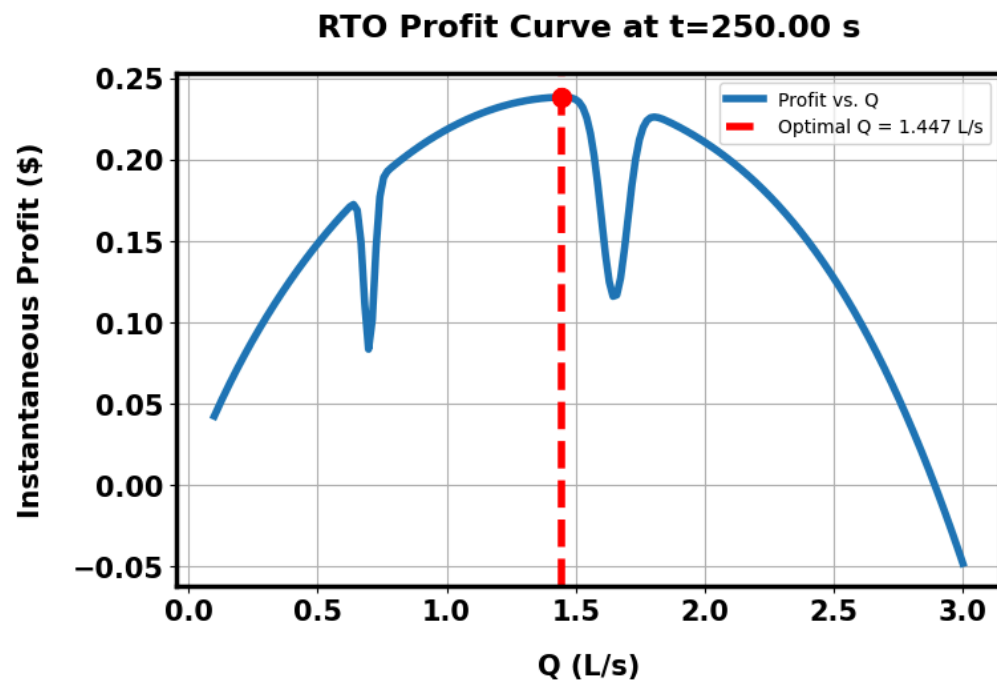


Figure 16. Plot of profit objective vs. feed flowrate at the simulation time point of 250 seconds

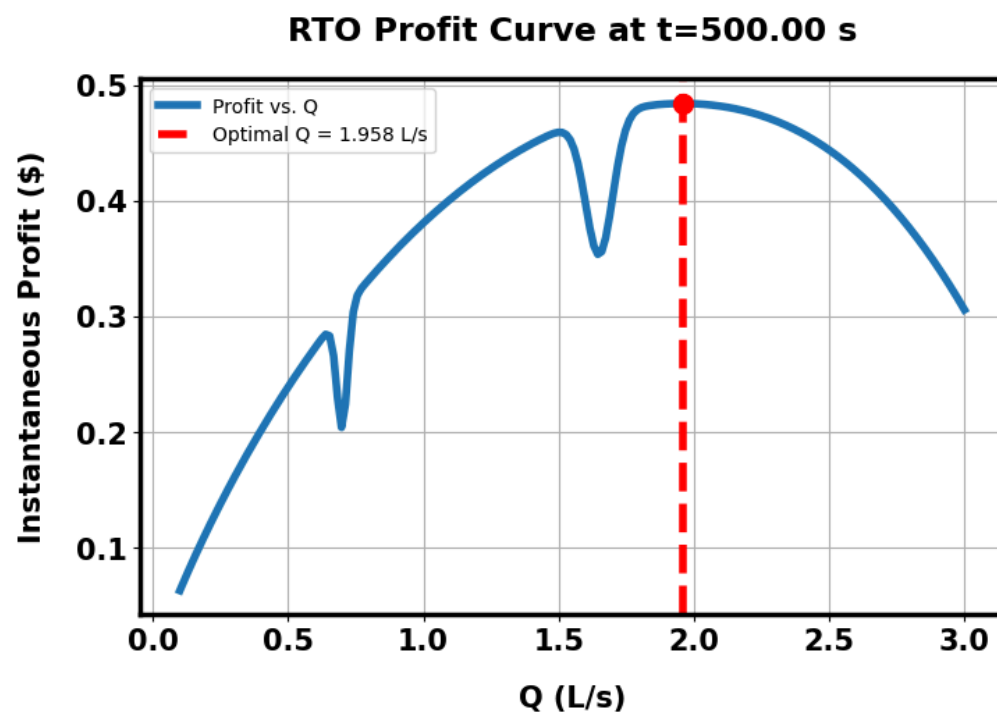


Figure 17. Plot of profit objective vs. feed flowrate at the simulation time point of 500 seconds (repeat of Figure 3)

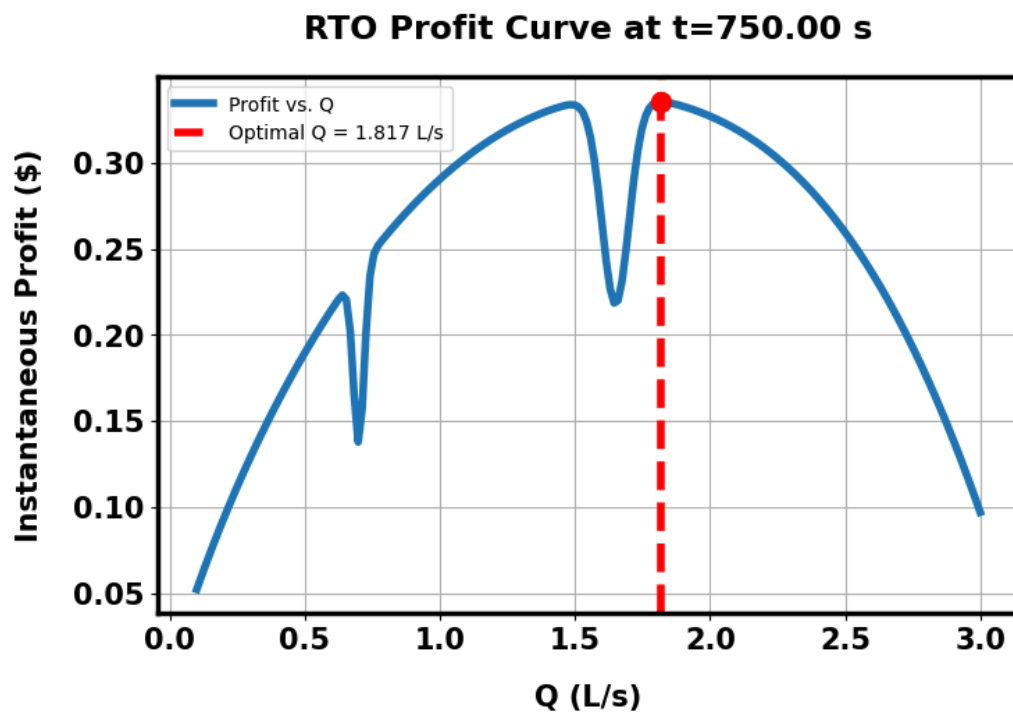


Figure 18. Plot of profit objective vs. feed flowrate at the simulation time point of 750 seconds

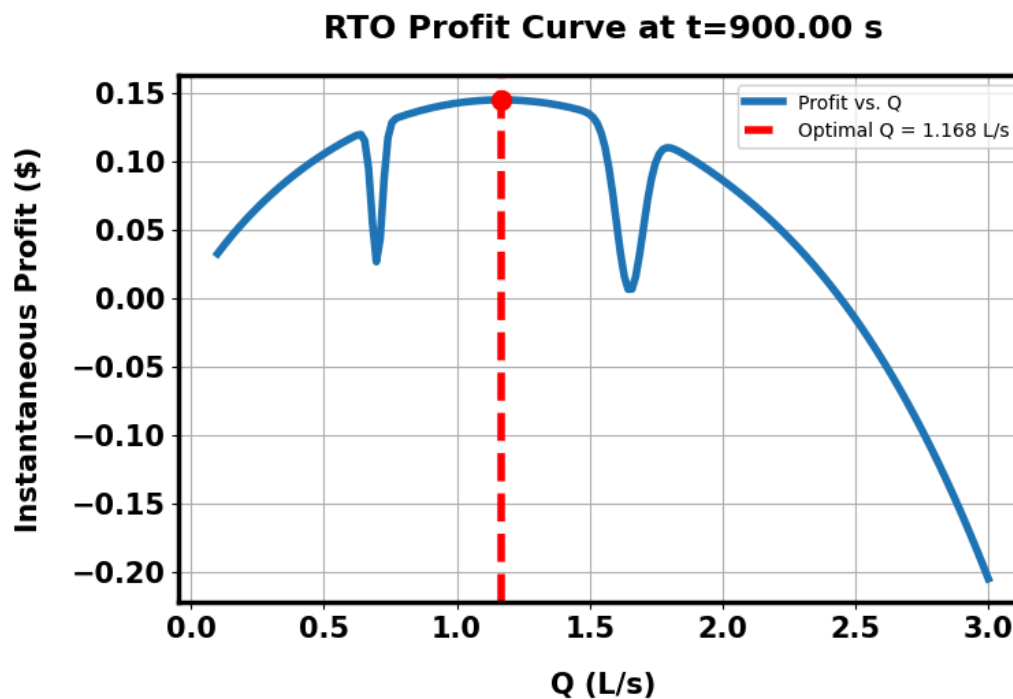


Figure 19. Plot of profit objective vs. feed flowrate at the simulation time point of 900 seconds

APPENDIX F - TUNING THE TRADITIONAL RTO ALGORITHM

```
args = [300,350,350,20,1,
        1000,0.1,5,
        5000,1000,2.5,-20000,200000,
        5,10,
        100,120,2,
        1.9,
        "Traditional RTO - Generate Data",False]

Tc_setting, Tf, T0, V, Qi, total_time, dt_control, dt_RTO,\
E_by_R, rho, Cp, dHr, UA, Kc, tauI, \
dip1_magnitude, dip2_magnitude, \
num_guesses, rxn_order, algorithm, fixed_setpoint = init_vars(args)

guess_range = np.arange(1,5)
n_trials = 10

tune_RTO_guesses(args, guess_range, n_trials)
```

Table 2. Tuning results showing the average profit (million USD) and average runtime (ms) based on different number of initial guesses passed into traditional RTO algorithm

Number of Initial Guesses	Average Profit (Million\$)	Average Runtime (ms)
1	11.2133	1.874
2	11.4155	3.338
3	11.4155	4.998
4	11.4144	5.866

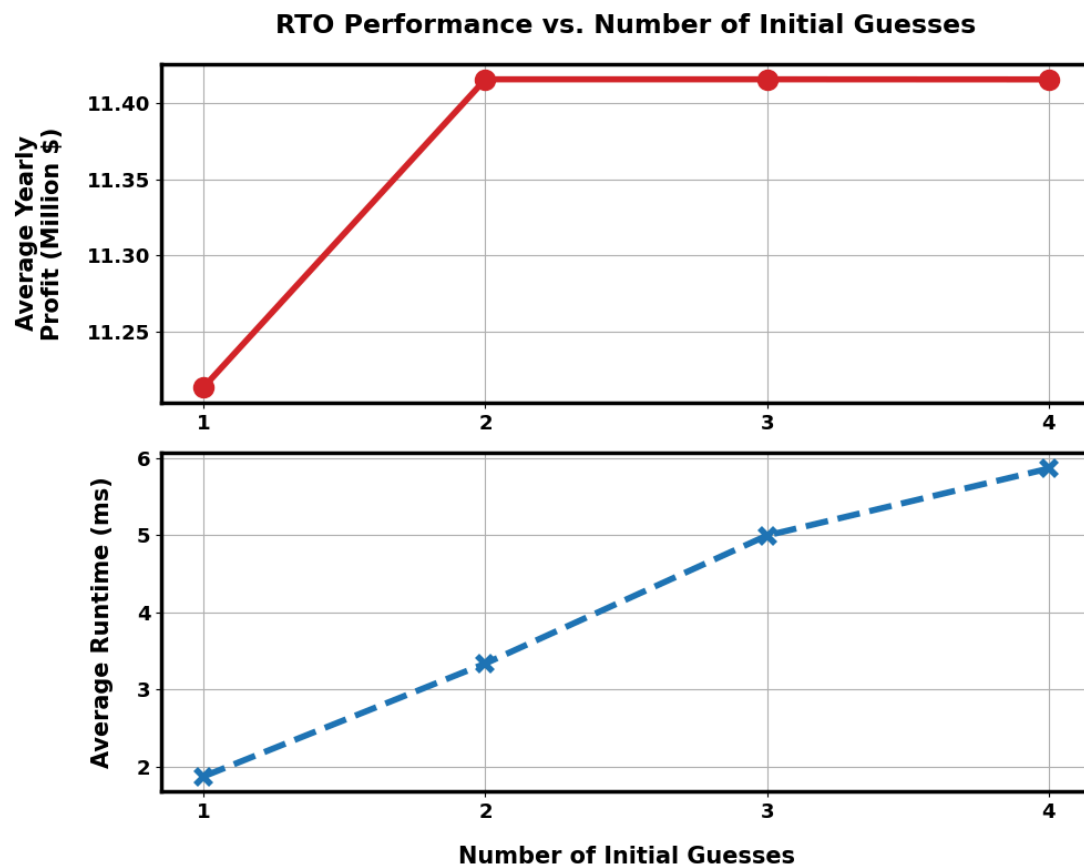


Figure 20. Tuning plot to determine the optimal number of initial guesses for traditional RTO (repeat of **Figure 4**)

APPENDIX G - TRAINING AND TUNING ML RTO MODELS

```
args = [300,350,350,20,1,
        1000,0.1,5,
        5000,1000,2.5,-20000,200000,
        5,10,
        100,120,2,
        1.9,
        "Traditional RTO - Generate Data",False]

Tc_setting, Tf, T0, V, Qi, total_time, dt_control, dt_RTO,\
E_by_R, rho, Cp, dHr, UA, Kc, tauI, \
dip1_magnitude, dip2_magnitude, \
num_guesses, rxn_order, algorithm, fixed_setpoint = init_vars(args)

ml_tuning_params = {
    "KNN": [1, 2, 3, 5, 10],
    "Decision Tree": [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15],
    "Random Forest": [50, 100, 200, 300],
    "Gradient Boosting": [20, 25, 30, 35, 50, 100, 200]
}

n_iterations = 3
tuning_iterations = 5

model_results, avg_trad_profit, avg_trad_runtime_ms = \
    analyze_ml_performance(args, ml_tuning_params,
                           n_iterations, tuning_iterations)

print(f"Traditional RTO Avg Profit: ${\
avg_trad_profit/(10**6):.4f}M | Avg Runtime: {avg_trad_runtime_ms:.4f}ms\n")
```

Traditional RTO Avg Profit: \$11.4155M Avg Runtime: 3.3796ms
--

Figure 20. Average profit and average runtime of the traditional RTO algorithm returned before training of ML models

```
model_results['KNN'].head()
```

Table 3. Tuning results for KNN regression ML model when used for RTO in the CSTR simulation

	Hyperparameter_Value	R2_Mean	Profit_Mean_ML	Profit_Mean_Trad	Profit_Change_(\$)	Profit_Change_%	Runtime_Mean_ML_ms	Runtime_Mean_Trad_ms
0	1	0.948826	1.141810e+07	1.141545e+07	2650.049104	0.023215	0.259975	3.379587
1	2	0.956713	1.141091e+07	1.141545e+07	-4546.220911	-0.039825	0.332498	3.379587
2	3	0.937224	1.141675e+07	1.141545e+07	1300.162327	0.011389	0.365480	3.379587
3	5	0.926614	1.140821e+07	1.141545e+07	-7245.221142	-0.063469	0.308061	3.379587
4	10	0.890622	1.136569e+07	1.141545e+07	-49764.478796	-0.435940	0.341665	3.379587

```
plot_ML_tuning_results(model_results['KNN'],
                       "Nearest Neighbors")
```

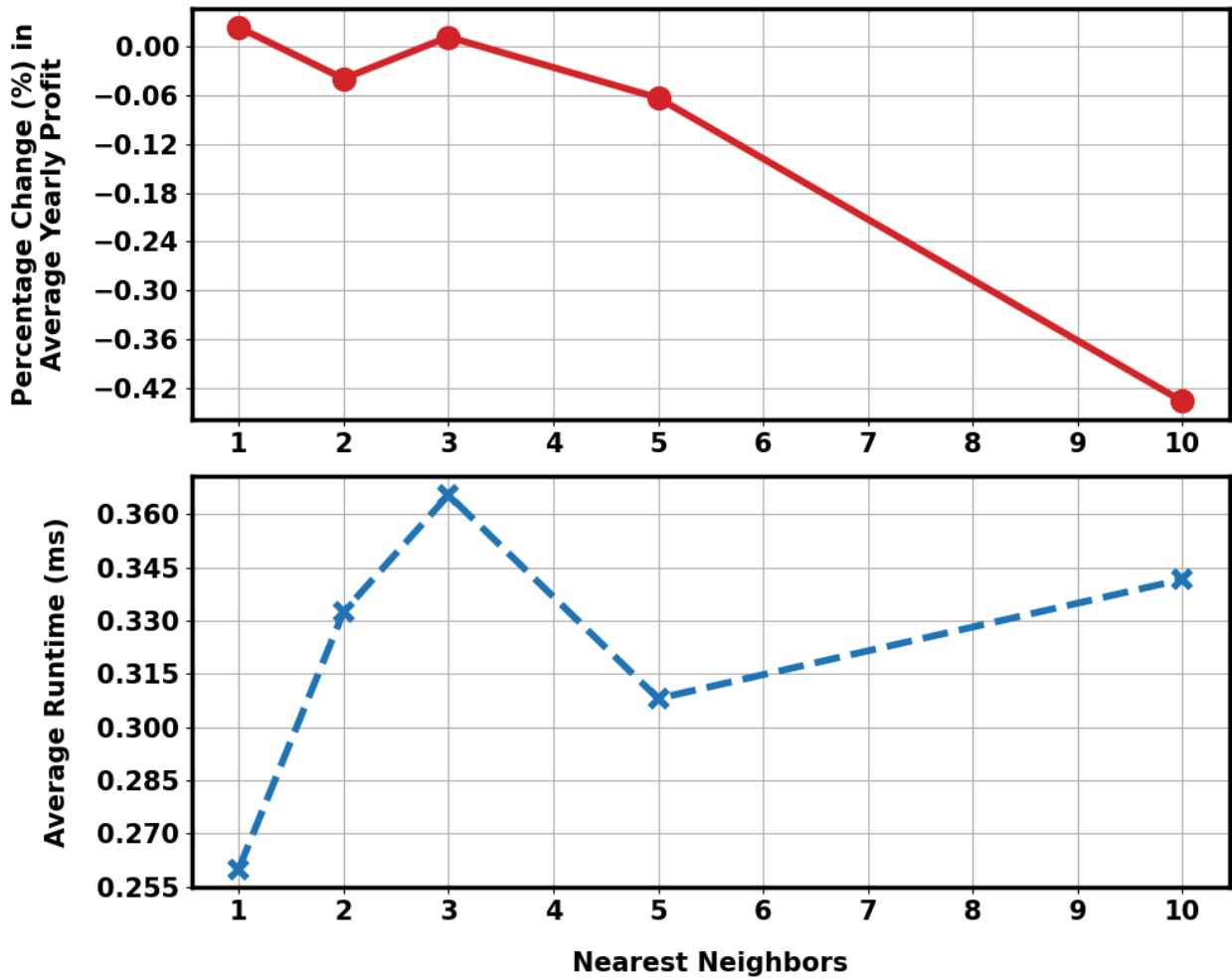


Figure 21. Tuning results showing the average percentage change (%) in yearly profit and average runtime (ms) based on different specified number of the nearest neighbors tuning parameter for KNN regressor models

```
model_results['Decision Tree'].head()
```

Table 4. Tuning results for decision tree regression ML model when used for RTO in the CSTR simulation

	Hyperparameter_Value	R2_Mean	Profit_Mean_ML	Profit_Mean_Trad	Profit_Change_(\$)	Profit_Change_%	Runtime_Mean_ML_ms	Runtime_Mean_Trad_ms
0	5	0.994706	1.141705e+07	1.141545e+07	1594.656230	0.013969	0.129944	3.379587
1	6	0.998383	1.141345e+07	1.141545e+07	-2002.240214	-0.017540	0.131594	3.379587
2	7	0.994791	1.142051e+07	1.141545e+07	5057.266308	0.044302	0.137378	3.379587
3	8	0.998474	1.141793e+07	1.141545e+07	2477.748411	0.021705	0.146673	3.379587
4	9	0.998416	1.141946e+07	1.141545e+07	4009.196052	0.035121	0.165034	3.379587

```
plot_ML_tuning_results(model_results['Decision Tree'],
                      "Maximum Depth")
```

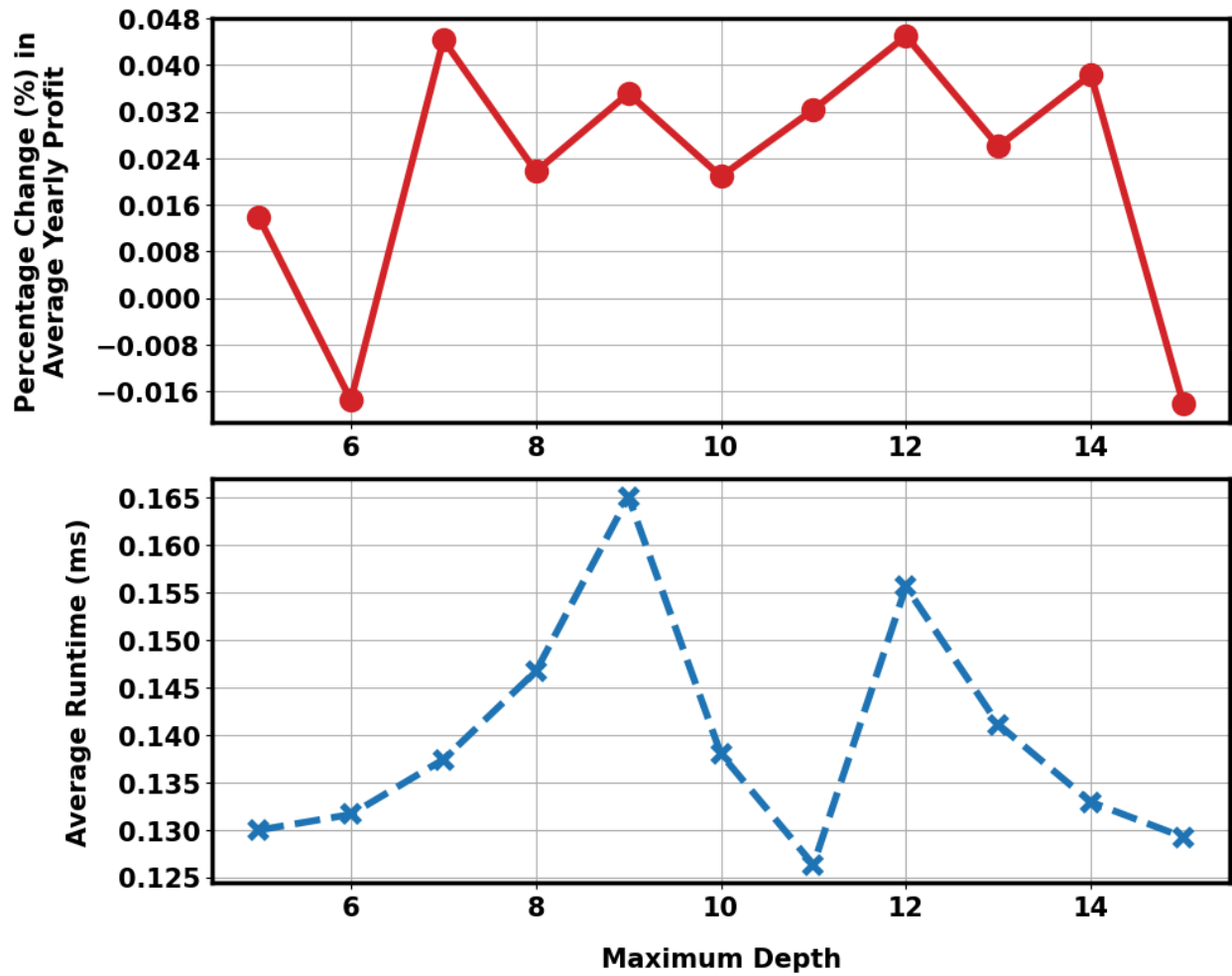


Figure 22. Tuning results showing the average percentage change (%) in yearly profit and average runtime (ms) based on different specified number of the nearest neighbors tuning parameter for decision tree regressor models

```
model_results['Random Forest'].head()
```

Table 5. Tuning results for random forest regression ML model when used for RTO in the CSTR simulation

	Hyperparameter_Value	R2_Mean	Profit_Mean_ML	Profit_Mean_Trad	Profit_Change_(\$)	Profit_Change_%	Runtime_Mean_ML_ms	Runtime_Mean_Trad_ms
0	50	0.997581	1.141471e+07	1.141545e+07	-743.621421	-0.006514	1.520067	3.379587
1	100	0.998007	1.141454e+07	1.141545e+07	-907.101273	-0.007946	2.750608	3.379587
2	200	0.998080	1.141439e+07	1.141545e+07	-1062.327218	-0.009306	5.242587	3.379587
3	300	0.998141	1.141363e+07	1.141545e+07	-1819.299415	-0.015937	7.652041	3.379587

```
plot_ML_tuning_results(model_results['Random Forest'],
                       "Number of Estimators")
```

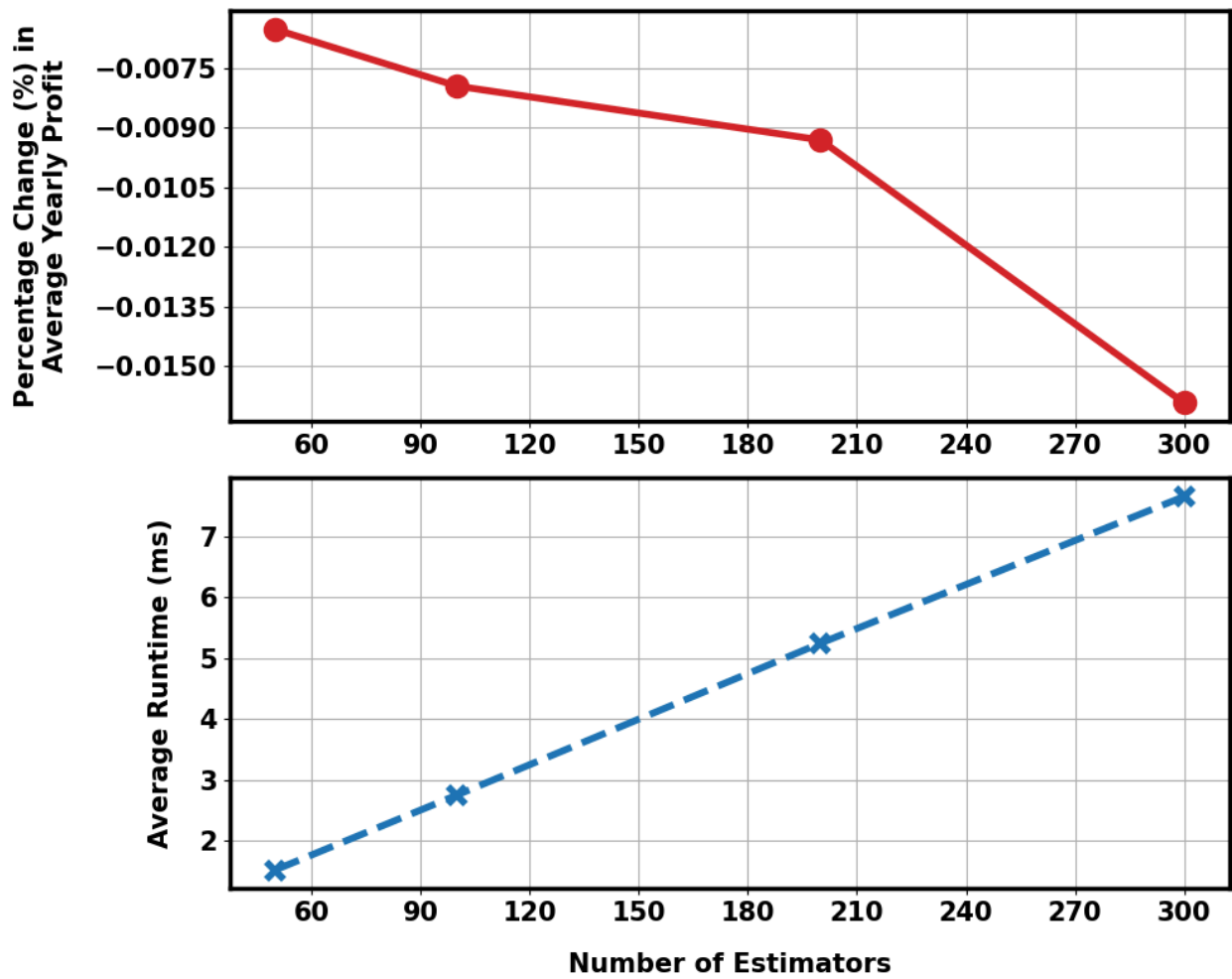


Figure 23. Tuning results showing the average percentage change (%) in yearly profit and average runtime (ms) based on different specified number of the nearest neighbors tuning parameter for random forest regressor models

```
model_results['Gradient Boosting'].head()
```

Table 6. Tuning results for gradient boosting regression ML model when used for RTO in the CSTR simulation

	Hyperparameter_Value	R2_Mean	Profit_Mean_ML	Profit_Mean_Trad	Profit_Change_(\$)	Profit_Change_%	Runtime_Mean_ML_ms	Runtime_Mean_Trad_ms
0	20	0.982671	1.142252e+07	1.141545e+07	7067.735920	0.061914	0.194758	3.379587
1	25	0.992650	1.142663e+07	1.141545e+07	11182.611986	0.097960	0.210433	3.379587
2	30	0.996128	1.142508e+07	1.141545e+07	9629.643731	0.084356	0.187880	3.379587
3	35	0.997335	1.142230e+07	1.141545e+07	6850.710610	0.060013	0.194444	3.379587
4	50	0.997965	1.141751e+07	1.141545e+07	2054.677288	0.017999	0.188380	3.379587

```
plot_ML_tuning_results(model_results['Gradient Boosting'],
                        "Number of Estimators")
```

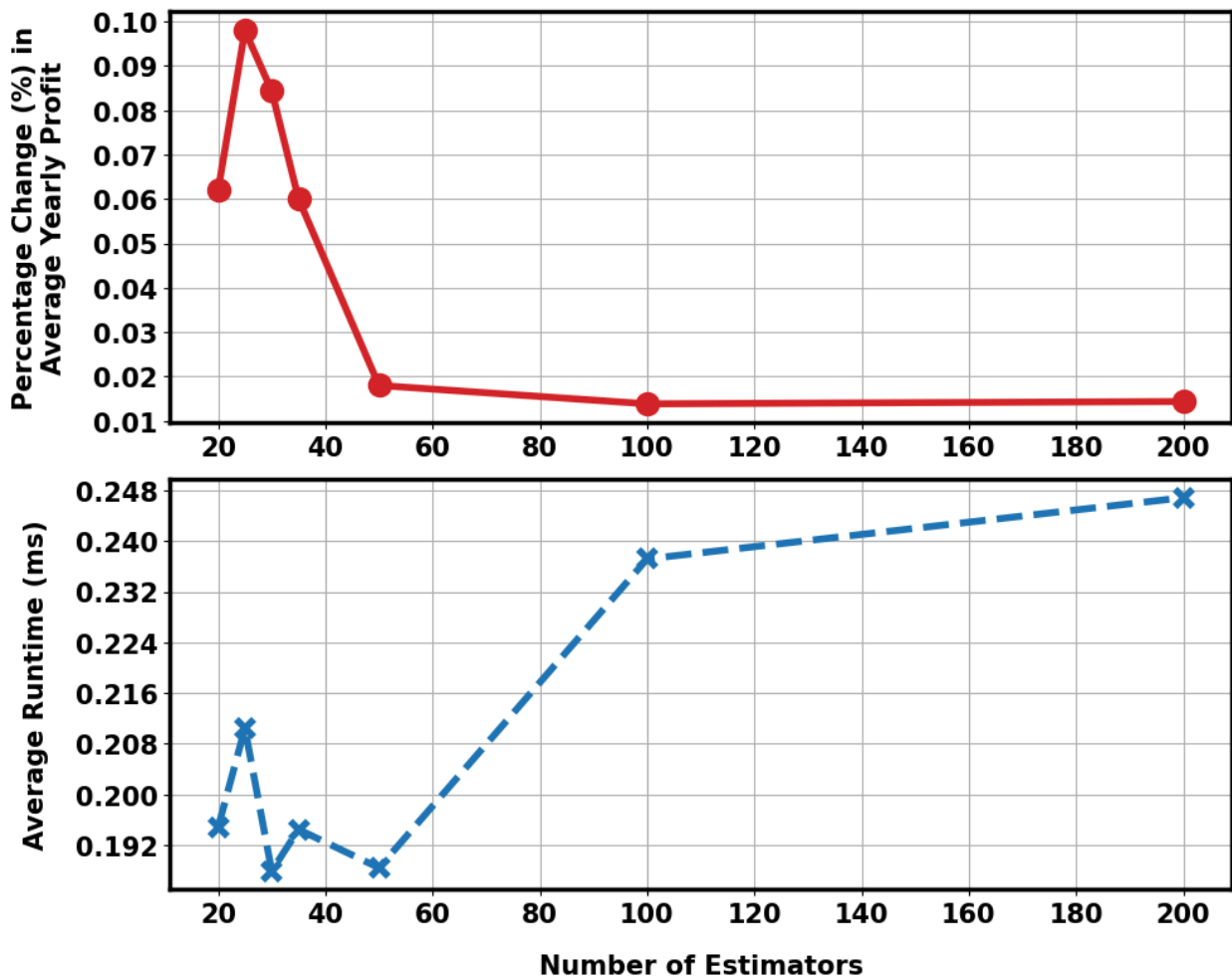


Figure 24. Tuning results showing the average percentage change (%) in yearly profit and average runtime (ms) based on different specified number of the nearest neighbors tuning parameter for gradient boosting regressor models

```

categories = ['KNN',
              'Decision Tree',
              'Random Forest',
              'Gradient Boosting']
max_profit_change = \
    [np.max(model_results['KNN']['Profit_Change_%']),
     np.max(model_results['Decision Tree']['Profit_Change_%']),
     np.max(model_results['Random Forest']['Profit_Change_%']),
     np.max(model_results['Gradient Boosting']['Profit_Change_%'])]

colors = []
for value in max_profit_change:
    color = 'green' if value>0 else 'red'
    colors.append(color)

plt.figure(figsize=(12, 8))
bar_plot = plt.bar(categories, max_profit_change, color=colors)
plt.bar_label(bar_plot, fontsize=15, fontweight='bold',
              padding=1, fmt='{:.4f}'+'%')

ax = plt.gca()
ax.tick_params(axis='both', which='major', labelsize=15)
plt.setp(ax.get_xticklabels(), fontweight='bold')
plt.setp(ax.get_yticklabels(), fontweight='bold')
for side in ['top', 'bottom', 'left', 'right']:
    ax.spines[side].set_linewidth(2.5)

title=\
    plt.title("\n".join(wrap("Maximum Change (%) in Yearly Profit Between Traditional
RTO and ML RTO",40)),
              fontweight='bold', fontsize=20, pad=25)

xlabel=\
    plt.xlabel("Type of ML Regression Model Used for RTO", fontweight='bold',
              fontsize=15, labelpad=25)

ylabel=\
    plt.ylabel("Maximum Percentage Change (%) in Yearly Profit", fontweight='bold',
              fontsize=15, labelpad=25)

```

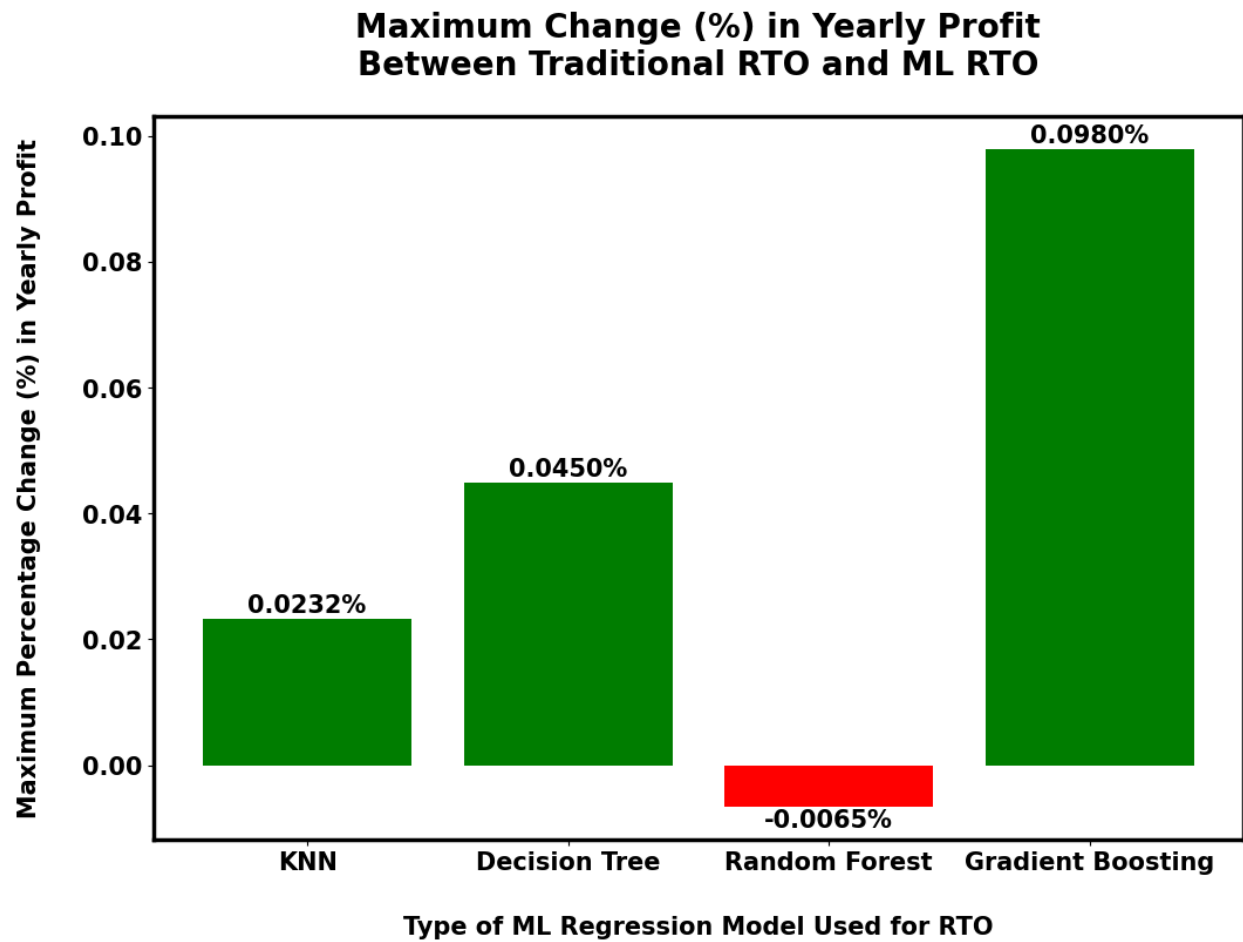


Figure 25. Maximum profit gain from each type of ML model (repeat of Figure 6)

```

runtime_factors_of_best_models = []

for category in categories:
    df = model_results[category]
    max_profit = np.max(df['Profit_Change_($)'])
    index_of_max_profit = df.index[df['Profit_Change_($)']==max_profit][0]

    avg_ML_runtime = df.loc[index_of_max_profit, 'Runtime_Mean_ML_ms']
    runtime_change_factor = avg_trad_runtime_ms / avg_ML_runtime
    runtime_factors_of_best_models.append(runtime_change_factor)

colors = []
for value in runtime_factors_of_best_models:
    color = 'green' if value>=1 else 'red'
    colors.append(color)

plt.figure(figsize=(12, 8))
bar_plot = plt.bar(categories, runtime_factors_of_best_models, color=colors)
plt.bar_label(bar_plot, fontsize=15, fontweight='bold')

ax = plt.gca()
ax.tick_params(axis='both', which='major', labelsize=15)
plt.setp(ax.get_xticklabels(), fontweight='bold')
plt.setp(ax.get_yticklabels(), fontweight='bold')
for side in ['top', 'bottom', 'left', 'right']:
    ax.spines[side].set_linewidth(2.5)

title=\
    plt.title("\n".join(wrap("Factor of Computational Speed Increase Between
Traditional RTO and Tuned ML RTO Model",50)),
              fontweight='bold', fontsize=20, pad=25)
xlabel=\
    plt.xlabel("Type of ML Regression Model Used for RTO", fontweight='bold',
              fontsize=15, labelpad=25)
ylabel=\
    plt.ylabel("Computational Speed Increase Factor", fontweight='bold', fontsize=15,
              labelpad=25)

```

Factor of Computational Speed Increase Between Traditional RTO and Tuned ML RTO Model

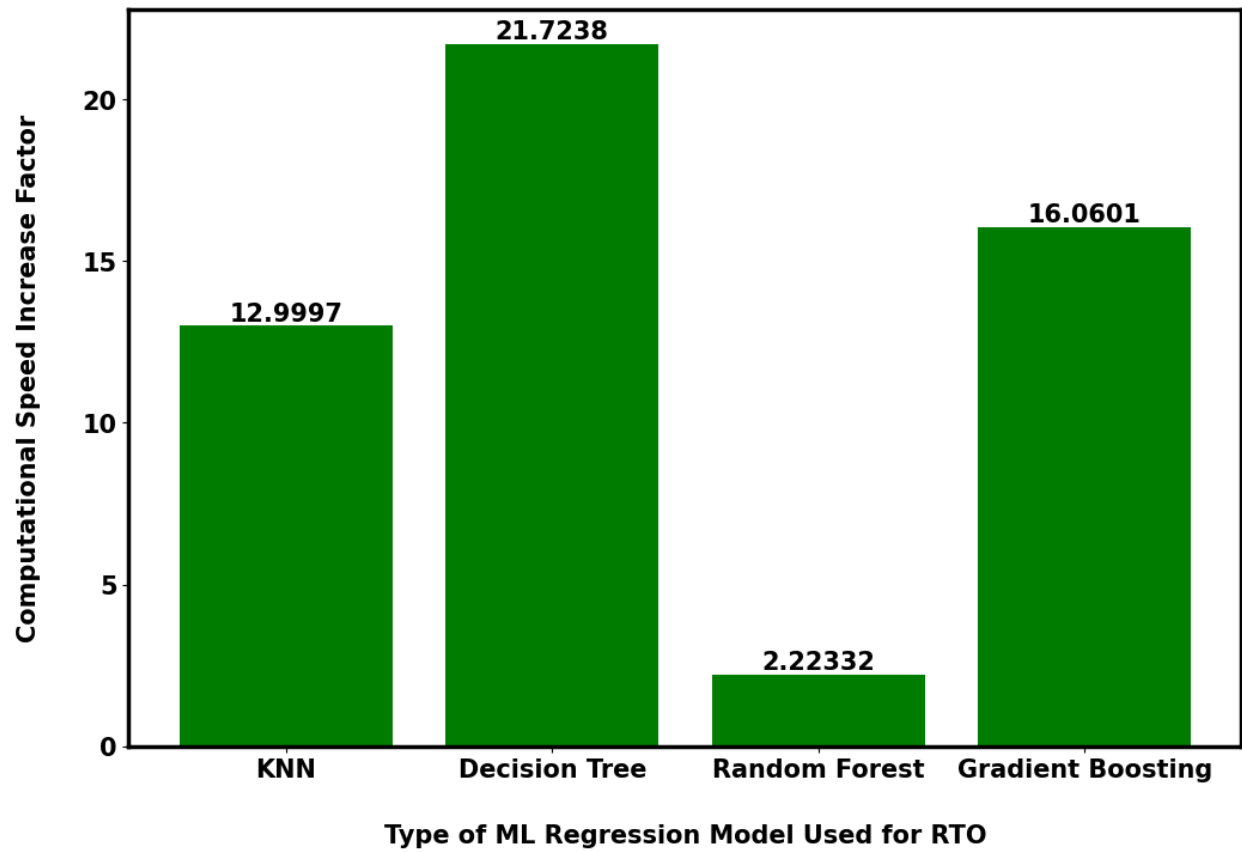


Figure 26. Factor of speed increase with each type of ML model (repeat of Figure 7)

```

args = [300,350,350,20,1,
        1000,0.1,5,
        5000,1000,2.5,-20000,200000,
        5,10,
        100,120,2,
        1.9,
        "Traditional RTO - Generate Data",False]

Tc_setting, Tf, T0, V, Qi, total_time, dt_control, dt_RTO,\
E_by_R, rho, Cp, dHr, UA, Kc, tauI, \
dip1_magnitude, dip2_magnitude, \
num_guesses, rxn_order, algorithm, fixed_setpoint = init_vars(args)

ml_tuning_params = {
    "Gradient Boosting": list(np.arange(20,40,1))
}
n_iterations = 3
tuning_iterations = 10

model_results, avg_trad_profit, avg_trad_runtime_ms = \
    analyze_ml_performance(args, ml_tuning_params,
                           n_iterations, tuning_iterations)
plot_ML_tuning_results(model_results['Gradient Boosting'],
                       "Number of Estimators")

```

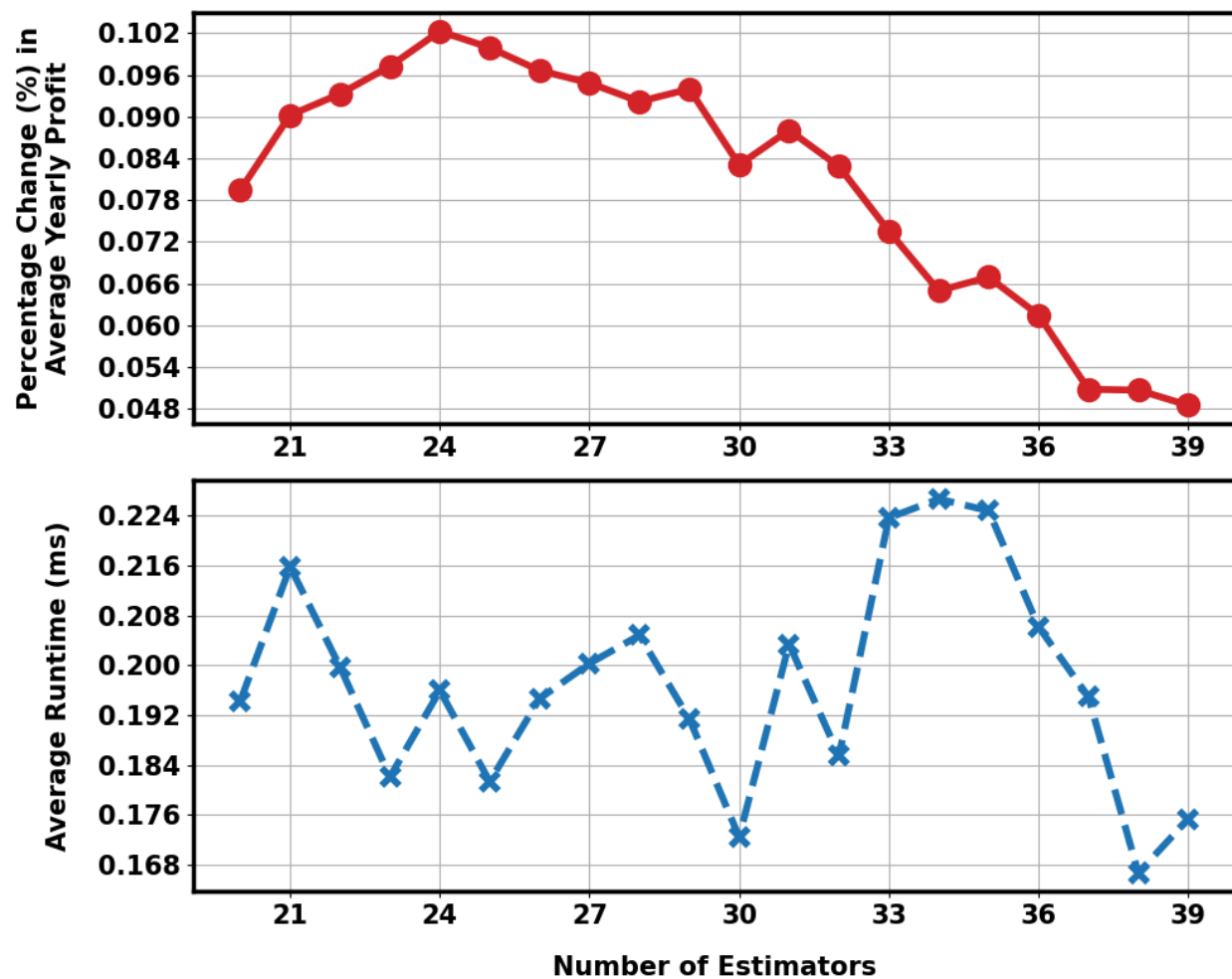


Figure 27. Results of fine-tuning number of estimators for gradient boosting ML RTO model (repeat of Figure 8)

```

runtime_factors_of_best_model = []

df = model_results['Gradient Boosting']
max_profit_percentchange = np.max(df['Profit_Change_%'])
max_profit_dollarchange = np.max(df['Profit_Change_($)'])
index_of_max_profit = df.index[df['Profit_Change_%']==max_profit_percentchange][0]

tuning_param_value = df.loc[index_of_max_profit, 'Hyperparameter_Value']

avg_ML_runtime = df.loc[index_of_max_profit, 'Runtime_Mean_ML_ms']
runtime_change_factor = avg_trad_runtime_ms / avg_ML_runtime

print(f"Best tuned ML RTO model: Gradient Boosting Regressor with {tuning_param_value}
estimators")
print(f"Maximum yearly profit change compared to traditional RTO: gain of {\
max_profit_percentchange:.3f}% (= ${max_profit_dollarchange:.2f})")
print(f"Factor of speed increase compared to traditional RTO:
{runtime_change_factor:.3f}")

```

```

Best tuned ML RTO model: Gradient Boosting Regressor with 24 estimators
Maximum yearly profit change compared to traditional RTO: gain of 0.102% (= $11674.53)
Factor of speed increase compared to traditional RTO: 17.086

```

Figure 28. Output of metrics for best-tuned ML model (repeat of Figure 9)


```

algorithm = "ML-Based RTO"

time_points_control, time_points_RTO, CB_setpoints_array, \
flowrates, concentrations, generated_data, profit_array, \
rto_runtime_array, avg_rto_runtime, total_rto_runtime = \
    reactor_simulator(Tc_setting, Tf, T0, V, E_by_R, rho, Cp, dHr,
                      UA, Qi, Kc, tauI, total_time, dt_control, dt_RTO,
                      algorithm, fixed_setpoint, dip1_magnitude, dip2_magnitude,
                      num_guesses, rxn_order, MLmodel)

p = simulation_plot(time_points_control, time_points_RTO, CB_setpoints_array,
                   flowrates, concentrations, profit_array)
print(f"Average Runtime = {avg_rto_runtime*1000:.2f}ms")

cumulative_profit, total_profit = \
    integrate_cumulative_profit(time_points_control[1:], profit_array)
yearly_traditional_profit = \
    total_profit * (365*24*3600 / total_time) / 1000000
print(f"Yearly Profit = ${yearly_traditional_profit:.4f}M")

```

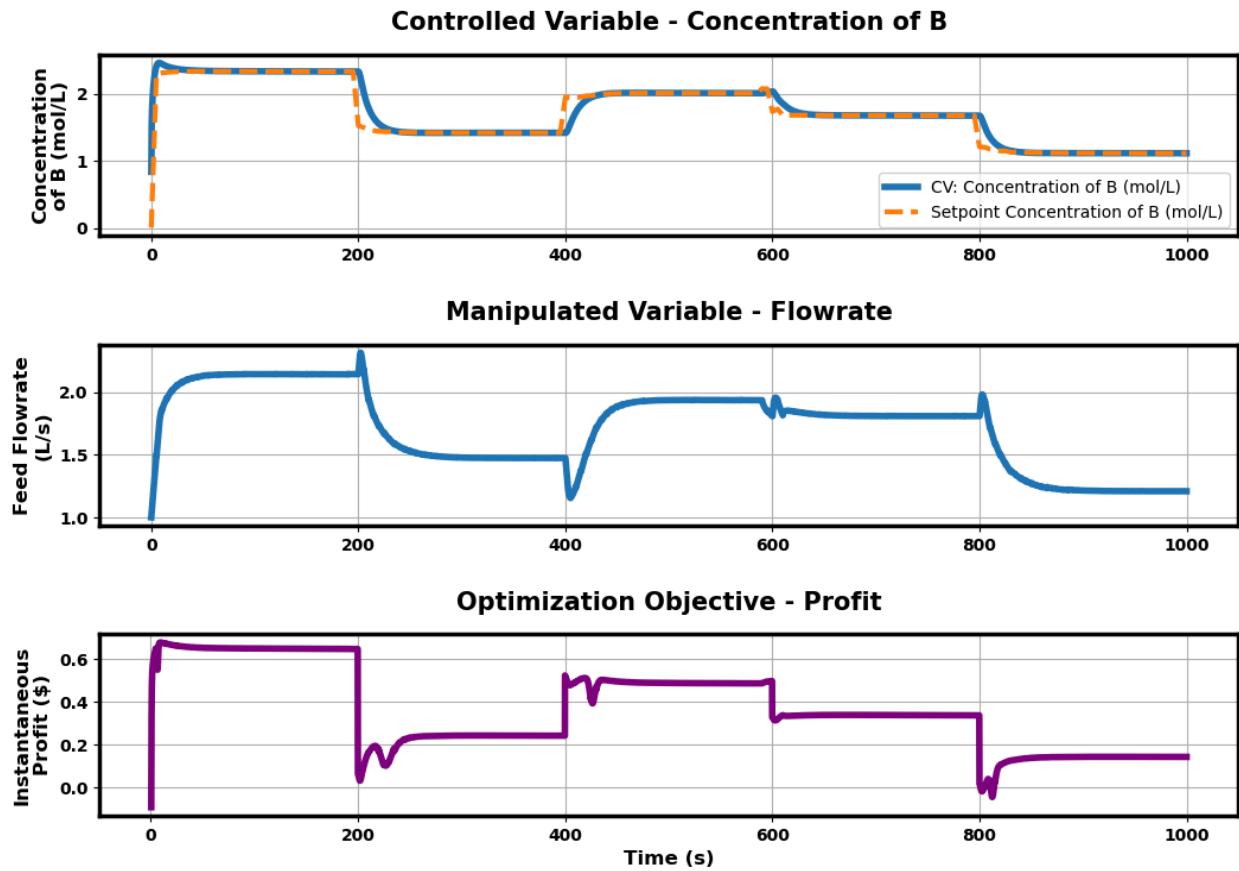


Figure 29. Plots of dynamic responses with the RTO in the CSTR simulation using the best-tuned gradient boosting ML model

Average Runtime = 0.17ms
Yearly Profit = \$11.4258M

Figure 30. Output showing the average runtime and yearly profit for a single simulation with the best-tuned gradient boosting ML RTO model