**Coursework 1 Part 2: PID Tuning using Data-driven Optimisation**

Team Abyss:Elton Lam, Zhongqi Zhuang, John Huang and Nicholas Gerard

*Department of Chemical Engineering, Imperial College London, U.K.*

**1. Preface**

This coursework takes advantage of Jordanla, the same algorithm used in Coursework 1 Part 1 [1]. The key difference lies in the manner in which the starting point was selected, where initial values for the controller gains were estimated using SIMC tuning heuristics.

**2. Intuition**

**2.1 Big picture**

The Jordanla algorithm utilises a representation of a polygon to optimize unknown functions in multidimensional space, initially placing vertices around a starting point. It iteratively improves by replacing the worst vertex through reflection and employs bisection to shrink the polygon and explore the solution space cautiously to converge towards optima. A detailed description about the algorithm can be found in Section 2.1 of the report for Coursework 1 Part 1 [1].

**2.2 Selection of initial point**

Selecting an appropriate starting point for algorithms aids rapid convergence to an optimal solution, as algorithms begin their search in the vicinity of the optima.

In order to find a well-informed initial point, specifically in the context of establishing the initial K values for the controller gains, SIMC tuning heuristics were used following the methodology outlined in [2], under the constraints that the transfer functions governing the system adhere to a first order plus time delay model. Reliable transfer functions for the model were obtained by performing several step changes with varying magnitudes on the model and computing the average.

**3. Methodology**

The main methodology employed by the algorithm is the same as in Coursework 1 Part 1 and is explained in Section 3 of that report [1].

However, two additional functions were included in this algorithm: simc\_tuning and bound\_values, that provides initial values for the controller gains and ensures that they are compatible with the system’s bounds, respectively. These additions can be seen in the pseudocode in Section 5.

**4. Expected performance**

A graph with green and red lines

Description automatically generated

**Figure 1**: Value of best minimum found by Jordanla for different starting points

**A graph with red and blue lines

Description automatically generated**

**Figure 2**: Distance between consecutive Ks

**Figure 1** highlights the impact of using SIMC tuning heuristics instead of a random search to choose a starting point for Jordanla. In some cases, random search provides a better starting point than that provided by SIMC, however the starting point is generally better and consistent when SIMC tuning is used.

**Figure 2** illustrates that the distance between consecutive Ks is lower than results from algorithms like Scipy’s Powell. This can be attributed to the way Jordanla selects its next point; due to the reflection methodology employed, it is only allowed to sample points in the vicinity of the current polygon, thereby reducing the magnitude that the Ks can change.

**Table 1** shows the performance scores of Jordanla compared to Scipy’s Powell and Cobyla for three randomly chosen set points, demonstrating Jordanla’s robustness for the problem at hand.

**Table 1:** Performance scores of Jordanla against other algorithms for different set points (SPs).

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **SP1** | **SP2** | **SP3** |
| Jordanla | 658 | 1946 | 1444 |
| Powell | 1013 | 2092 | 1765 |
| Cobyla | 818 | 2014 | 1537 |

**5. Pseudocode**

Jordanla(*func, x\_dim, bounds, iter\_tot*)

*start\_point ← array of zeros of length*

*x\_dim*

*n ← x\_dim + 1*

*p ←* Initialise\_p*(start\_point, n, x\_dim)*

*func\_vals ←* Evaluate\_func\_at\_p(*p, func*)

*iters ← n*

***for*** *i* ***from*** *0* ***to*** *iter\_tot - n*

*p, func\_vals ←* Update\_p(*p, func\_vals, bounds*)

*iters ← iters + 1*

***if***Time\_exceeded()***or*** *iters > iter\_tot*

*break*

*best\_idx ←* ***index of min value in*** *func\_vals*

*best\_x ← p[best\_idx]*

*best\_f ← func\_vals[best\_idx]*

***return*** *best\_x, best\_f*

Initialise\_p(*n, x\_dim, bounds*)

*step\_size ← 10 - 30% of bounds range*

*p ← array of zeros with size (n, x\_dim)*

*p [0] ←* SIMC\_tuning()

***for*** *i* ***from*** *1* ***to*** *n*

*p [i, i - 1] ← step\_size*

***for*** *j* ***from*** *0* ***to*** *x\_dim*

*p [j] ←* Bound\_values(*p [j]*)

***return*** *p*

SIMC\_tuning()

*Kp, tau, Td­, lambda ← transfer func values*

*Ks0 ← array of initial K values, calculated using SIMC tuning rules*

***return*** *Ks0*

Bound\_values(*Ks*)

*Ks: np.array of K values*

***for*** *j* ***from*** *0* ***to***Length(*Ks*)

*Ks[j] ← Clip each K to bounds*

***return*** *Ks*

Update\_P(*p, func\_vals, bounds*)

*worst\_x ← index of max value in func\_vals*

*centroid ← mean of p excluding p[worst\_x]*

*reflection ← centroid + (centroid - p[worst\_x])*

*reflection ←* Clip\_to\_bounds(*reflection, bounds*)

*reflection\_val ← func(reflection)*

***if*** *reflection\_val < func\_vals[worst\_x]*

*polygon[worst\_x] ← reflection*

*func\_vals[worst\_x] ← reflection\_val*

***else***

*p[worst\_x] ← (p [worst\_x] + centroid)/2*

***return*** *polygon, func\_vals*

**6. Figure of algorithm**

**A flowchart of a graph

Description automatically generated**

**7. References**

[1] ML Report for CW1a

[2] Thornhill’s Notes