Maximising the Competitive Edge in Formula 1 Racing through Particle Swarm Optimisation

[redacted]

MATH 3001

Supervisor: [redacted]

University of Leeds
BSc Mathematics (Industrial)
March 24, 2023

Contents

1 Introduction

1.1 Summary

The Formula 1 Race Strategy Competition is a simulated Formula 1 race season in which a number of teams compete to maximise a season points total by investing a fixed budget across four variables. Each of the four variables, marketing, reliability, chassis and engine, are components that affect the lap time model for each given race [1]. The question of developing an optimal budget strategy for the competition is made difficult by the stochastic processes involved in the competition rules, the incomplete information of unknown competing team strategies and the huge number of possible strategy combinations, which we show to be $1.701 \cdot 10^{43}$ [2]. It is not considered computationally feasible to test all the possible budget split strategies a team may implement. Our objective is to develop a computationally feasible method that will maximise our chance of success in the competition by obtaining an optimal budget split for a single team.

Beaume (2021) demonstrates that computational methods can be designed to develop competitive budget strategies. In particular, success is obtained using a combination of an iterative method, a genetic algorithm and artificial intelligence to predict participating team budget strategies [3]. In this paper, we develop an alternative computational method to obtain a competitive budget split strategy, Appendix A, and consider the problem of addressing the unknown values of competing team strategies. We develop a computational program to simulate a large number of race seasons, which enables us to define our satisfaction with the expected performance of a given budget split strategy through a defined cost metric. Heuristic methods can be applied in conjunction with our computational simulation as a method to obtain an optimal budget split strategy [4]. We will consider the computational cost of implementing a heuristic tool and the computational trade-offs involved in their application [5].

Particle swarm optimisation (PSO) is an optimisation technique that draws analogies from the movement of a flock of birds collectively searching for food [6]. PSO has been applied to a variety of optimisation problem applications due to its ease of implementation and relative speed of convergence [6, 7]. An important component of PSO is its inertia weight parameter, w, and acceleration parameters c_1 and c_2 [8, 9]. We will examine the impact of these parameters on the convergence and success of the particle swarm optimisation algorithm when it is implemented to optimise the budget split strategy of a team [10, 11]. We will develop a penalty function to increase the likelihood of particles taking admissible positions during PSO implementation [12]. By considering the absolute distance moved by each particle between iterations we will determine suitable and unsuitable parameter values for our application of PSO, in particular, we will show that $w = 0.9$ performs significantly worse than $w = 0.42$ for particle convergence due to the impact on the scale of particle velocity. We will obtain a competitive budget split strategy for a team and evaluate the effectiveness of this strategy compared to the strategy of competing teams.

This paper demonstrates the effectiveness of PSO at generating near-optimal budget split strategies for a single team in the competition. While these results were obtained for a sample of competing team budget splits, our method could be extended to consider a range of competing strategies or altered to compete against dynamic strategies that can evolve between iterations. We believe our methodology and results could also be adapted for a wide range of constrained and unconstrained optimisation problems.

1.2 Ethical Considerations

Our research aimed to develop an optimal strategy for a Formula 1 race team. Although our research focused on a simulated Formula 1 competition [1], our aim was that the particle swarm optimisation techniques we used could be implemented by a professional race strategy engineer during a real Formula 1 race season. Additionally, our methodology could be adapted to develop optimal strategies for a wide range of constrained optimisation problems across several diverse industries including hardware design and cyber security. With the high level and wide-ranging potential applications of our research, it is important to consider both the benefits to society and also the potential risks and negative applications. In demonstrating the adaptability of particle swarm optimisation to efficiently discover "near optimal" solutions to complex, constrained optimisation problems, there is a risk that these techniques may be used maliciously to cause harm. There is also a risk that the techniques we propose are used improperly, in a way which may contribute towards causing unintentional harm. For example, our current optimisation problem does not model driver safety and only focuses on maximising the finishing position according to our defined metric. As such, our method may suggest budget strategies that, while successful, may negatively impact drivers' safety.

2 Problem Setup

2.1 Competition Rules

We consider a number of teams, n, competing in a Formula 1 Race Strategy Competition [1]. Each team has a budget of $8\imath$ to be allocated across four variables, marketing, chassis, engine and reliability, with a precision of up to one decimal place. We wish to optimise the budget split of a given team, which we define to be the combination of the four variables that the team possesses, in order to maximise the team's expected performance as judged by a metric which we will later define.

Let the marketing budget of team i be m_i, then a given driver will sign with team i with the following probability [1]:

$$p_i = \frac{m_i^4}{\sum_{j=1}^{n} m_j^4}, \tag{1}$$

where there are $2n$ drivers in total, and each team is assigned exactly two drivers. Each driver possesses a driving ability, denoted *Driving*, measured from $0\star$ to $5\star$. The drivers are assigned to teams in order of driving ability, where the driver with the highest *Driving* is assigned first. Each time a team has been assigned two drivers it is considered full, and p_i is recalculated with n reduced by 1 to represent the one less team available for the remaining drivers to sign to. The investment of this removed team is discounted from the calculation of Equation (1). In the case where each remaining team has a marketing investment, m_i, equal to zero, then a given driver has an equal probability of signing with each available team.

Let the chassis investment of a team i be c_i, then the associated chassis performance, cp_i (given in $\star$) is [1]:

$$cp_i\star \leftarrow c_i. \tag{2}$$

Let the engine investment of a team i be e_i, then the associated engine performance, ep_i (given in $\star$) is [1]:

$$ep_i\star \leftarrow e_i. \tag{3}$$

So we have that chassis and engine investment, in $\imath$, are equal to chassis and engine performance, given in $\star$, respectively. Let the reliability investment of team i be r_i, then the retirement probability of a driver assigned to team i, for a given race, is [1]:

$$p_{DNF} = \frac{[1 - erf(0.9r_i - 1.5)]^2}{4}, \tag{4}$$

where *DNF* stands for "Did Not Finish", and *erf* represents the error function. The lap time, t, for a given driver, $Driver$, signed to a given team i, racing on a given circuit, is [1]:

$$t = t_b + t_{perf} + p_t + p_f + R_{lap}, \tag{5}$$

where p_t and p_f are circuit-specific time penalty functions modelling tire wear and fuel load respectively. R_{lap} is the random function modelling pace irregularities specific to each driver, defined as [1]:

$$R_{lap} = 4(X_{RNG} - 0.4)^3 + 0.3X_{RNG}, \tag{6}$$

where X_{RNG} is a random number drawn from the uniform distribution between 0 and 1. t_b is the base time associated with the given circuit and t_{perf} is defined as [1]:

$$t_{perf} = -0.15(Performance + Setup), \tag{7}$$

where *Performance* is [1]:

$$Performance = \frac{3(Driving \cdot c_d + c_i \cdot c_c + e_i \cdot c_e)}{c_d + c_c + c_e}, \tag{8}$$

where c_d, c_c and c_e are circuit-specific weightings denoting that a circuit favours driver, favours chassis and favours engine respectively. Each circuit-specific weighting is equal to 1 unless the given circuit favours the component it denotes, where it is doubled so equals 2. For example, a balanced circuit would have the following values, $c_d = 1$, $c_d = 1$ and $c_e = 1$. A circuit that favours engine investment would have the following values, $c_d = 1$, $c_c = 1$ and $c_e = 2$ and a circuit favouring driving and favouring chassis would have values as follows, $c_d = 2$, $c_c = 2$ and $c_e = 1$. *Setup* is defined as the sum of two random numbers drawn from two uniform distributions [1]. The first number is associated with the given team, i, and is drawn from a uniform distribution between -0.5 and 0.5. The second number is associated with the given driver, $Driver$, and is drawn from a uniform distribution between -1 and 1. Traffic, the drag reduction system (DRS) and overtaking rules are also defined as aspects of the Formula 1 Race Strategy Competition [1]. However, due to the increase in computational complexity associated with modelling these aspects, which we will later examine, in comparison with the increased relatively small impact we assume they have, we choose to ignore these components in our research.

2.2 Initial Problem Analysis

Let us consider the budget split of some participating team, "Team 1". We first consider the investments of "Team 1", in chassis, c_i, and engine, e_i. From Equation (2) and Equation

(3) we have that the chassis and engine performances, cp_i and ep_i, are equal to the amount "Team 1" invests in each variable respectively. If a circuit favours chassis and engine equally, that is if $c_c = c_e$, then from our lap time equation, Equation (5), and the *Performance* component, Equation 8, the chassis and engine investment of "Team 1" will have equal effects on the lap time of a given driver assigned to "Team 1". Now let us consider the investment into marketing for "Team 1". Unlike for chassis and engine, we can see that from Equation (1), increasing marketing investment does not guarantee an increase in the driving ability, *Driving*, of the team's assigned drivers. Instead, we can see that increasing marketing investment for "Team 1" increases the probability that drivers that possess a high driving ability, *Driving*, will be assigned to it. From Equation 1 we can see that the relationship between driver assignment probability, p_i, and marketing investment is dependent on the budget splits of the $n-1$ teams participating against "Team 1". To illustrate this, consider two teams, "Team 2" and "Team 3", competing against each other to sign the following two drivers who have the driving ability, *Driving*, of 4.7$\star$ and the driving ability, *Driving*, of 2.2$\star$ respectively. If both "Team 2" and "Team 3" invested 0$\imath$ into marketing, such that $m_2 = m_3 = 0\imath$, then each team would have an equal, 50% chance of signing the 4.7$\star$ driver. However, if "Team 2" increased the investment into marketing, while "Team 3" remained the same, such that $m_2 = 0.1\imath$ and $m_3 = 0\imath$, then by applying Equation (1) we can see that "Team 2" would have a 100% of signing 4.7$\star$ driver. However if "Team 3" also increased the investment in marketing, such that $m_2 = 0.1\imath$ and $m_3 = 0.1\imath$, we have that both teams would again have a 50% chance of signing the 4.7$\star$ driver. Hence, we demonstrate that this probability is dependent on the marketing investments of the competing team or teams. Beaume (2022) gives another example of this dependency and goes further by demonstrating that an excessively large marketing investment of a team, "Team 2", can be exploited through the withdrawal of marketing investment by the competing team, "Team 3" [3].

Let us consider the reliability investment of some participating team, "Team 1". Unlike marketing, chassis and engine investments, the reliability investment of a team is not made with the purpose of improving the lap times of a team's drivers on a given race. It is instead a way to increase the probability that a team's drivers will successfully complete a given race. If a driver retires during a given race then they are not eligible to receive points for that race. As such, the reliability investment of "Team 1" can be framed as a trade-off between attaining a low retirement probability and decreasing its driver's lap times. For example, investing a large value of $\imath$ in reliability at the cost of decreased investment across chassis, engine and marketing (the *Performance* components from Equation (8)) will produce a car that is expected to finish a large number of races. However, in those races it does finish, the car will not be as fast due to the sacrificed investment in the three *Performance* components from Equation (8). As a result, the team will be expected to attain a lower number of points per race finished, which we will denote as ppf, due to the slower expected lap times. In contrast, if a team invests a low amount of $\imath$ into reliability, and instead increases the investment across variables that increase the components of *Performance* in Equation (8), that is chassis, engine and marketing, they are expected to finish a lower amount of races but produce quicker lap times. As a result, they should achieve comparatively higher finishing positions in races that they do finish, resulting in a higher ppf. Hence they are expected to earn more points per race finished, ppf, but finish fewer races per season. An optimal reliability investment could be defined as one that maximises the average product of the expected value of ppf and the expected number of races completed for a team's two drivers. In Figure 1 we can see that the relationship between reliability investment and the proba-

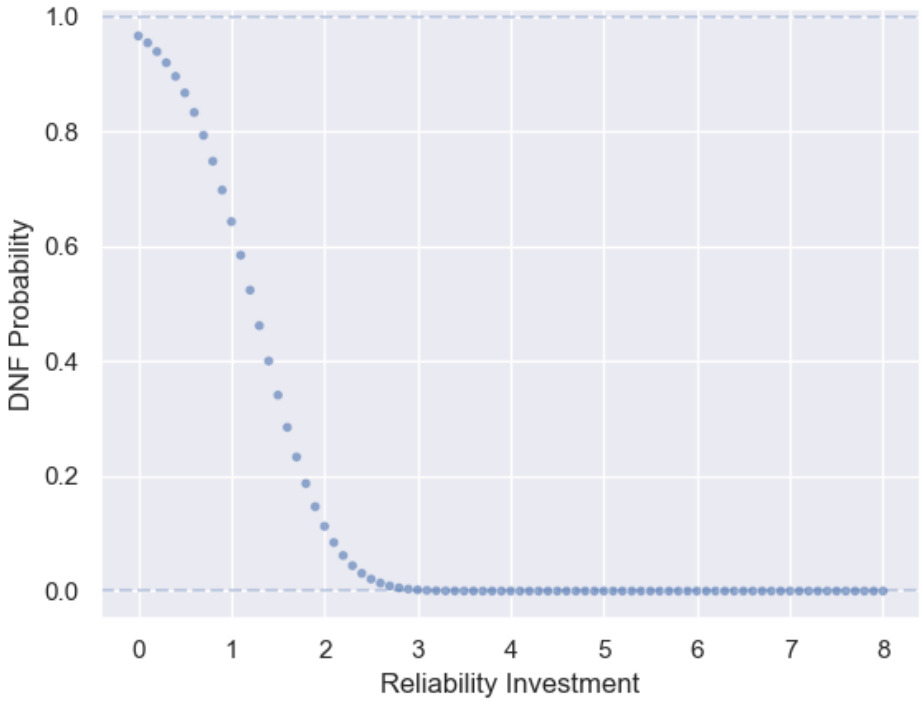

Figure 1
DNF Probability, P_{DNF}, for a single race plotted as a function of a team's reliability investment, r_i, as described by Equation (4).

bility that a driver will retire in a given race, denoted "DNF Probability", is a non-linear decreasing function, Equation (4). Figure 1 illustrates that for investments in reliability above $2.4\imath$, there is an extremely low return on investment in terms of "DNF Probability" reduction and that beyond $2.9\imath$ the "DNF Probability" appears to converge to just above zero, taking its lowest value, $1.4312 \cdot 10^{-31}$, at $8\imath$. From Figure 1 we see that for reliability investments below $2.4\imath$, there is a significantly larger decrease in "DNF Probability" as reliability investment increases compared to beyond $2.4\imath$, where "DNF Probability" begins to converge. This indicates that we may expect that an effective reliability investment is likely to be within the range $(0, 2.4\imath)$, as this range gives us a higher return on investment. That is, we achieve significantly higher increases in "DNF Probability" per each additional $0.1\imath$ invested in reliability, r_i.

3 Model

3.1 Competition Objective

We aim to locate the optimal budget split for a given team, "Team 1", assuming we know the budget splits of each other team participating in the Formula 1 Strategy Competition [1], where there are n teams participating in total. We define the optimal budget split for "Team 1" to be the combination of chassis investment, c_1, engine investment, e_1, marketing investment, m_1, and reliability investment, r_1, that maximises the satisfaction we have with the expected finishing position of "Team 1" for a single race season. To calculate the expected finishing position for each of the n participating teams we develop a season simulation subroutine. For every single race, the subroutine uses Equation (4) to remove the drivers that it calculates do not finish, DNF. It then ranks the remaining drivers in each race by t_{perf} and ranks teams based on the sum of each team's associated driver's points [1] after all ten races. The subroutine ignores random effects but does not ignore the probabilistic driver allocation and driver DNF processes. Therefore, we must consider the variance in the output of the season simulation subroutine and determine a large enough number of calls to the subroutine to generate an accurate expected finishing position for each team. A higher number of calls to the subroutine will give a more accurate expected finishing position, however, this comes at the cost of increased computational complexity. We will further examine this trade-off when we later implement our particle swarm optimisation program.

We define a metric that quantifies our satisfaction level with a given season finishing position for a team, "Team 1". We chose to define our metric as follows, as it best describes our personal satisfaction with each final season finishing position for a given team:

$$metric = \begin{cases} 32, & 1^{\text{st}} \text{ Place.} \\ 16, & 2^{\text{nd}} \text{ Place.} \\ 4, & 3^{\text{rd}} \text{ Place.} \\ 2, & 4^{\text{th}} \text{ Place.} \\ 1, & 5^{\text{th}} \text{ Place.} \\ 0, & \text{otherwise.} \end{cases} \tag{9}$$

Although we could aim to maximise this metric, we instead seek to minimise the negative of this metric, $-metric$, as the difference in these two aims is trivial for our purpose. Then we can define the budget split for "Team 1" that minimises the expected value of $-metric$ for "Team 1", where we assume we know the fixed budget splits for each n participating team, to be the optimal budget split strategy for "Team 1".

3.2 Season Simulation Subroutine

We develop our subroutine using Python, due to the extensive collection of available, relevant libraries and visualisation packages. To use our subroutine, we load into Python three excel data frames, which we denote as "budgetsplit", "driverlist" and "circuitlist". The "budgetsplit" data frame contains the set of budget splits for each of the n participating teams. The "driverlist" data frame contains the names of the participating drivers and their associated driving ability, *Driving*. The drivers are given in descending order

by *Driving*. The "circuitlist" data frame contains the circuit name, favourability weighting, follow delta, overtake delta, number of laps, and base lap time for each of the ten circuits. We note that the follow and overtake deltas are aspects of the competition related to traffic, DRS and overtaking which we choose to ignore. We define the function "seasonsim(budgetsplitnew, driverlist, circuitlist)" to take as input two of the three above-defined data frames, "driverlist" and "circuitlist". We exclude "budgetsplit", which we have replaced with "budgetsplitnew". "budgetsplitnew" is a copy of the "budgetsplit" data frame, but which we update with a different budget split for a single participating team. This is necessary for our subsequent implementation of the particle swarm optimisation algorithm which we shall later describe. The output of our "seasonsim(budgetsplitnew, driverlist, circuitlist)" function is the ordered season finishing position of each n participating team and the associated cost metric value, $-metric$, we choose to assign to each respective finishing position. The function works through two primary components, the driver allocation function and the lap time calculation function. The Python code to produce the function can be seen in Appendix A.

3.2.1 Driver Allocation Function

The first component of the "seasonsim(budgetsplitnew, driverlist, circuitlist)" function is the driver allocation process. We use Equation (1) to assign drivers to a team sequentially, in descending order based upon driving ability, *driving*. Our program works by calculating the value of Equation (1) and assigning it to be the dictionary value corresponding to the associated team, assigned as a key. We repeat a similar technique for a new dictionary in which we assign to each driver one random probability value taken from a uniform distribution between 0 and 1, which we denote as the driver's probability value. We then take the teams in a given order, which is trivial, and calculate the sum of all Equation (1) values up to the current team, for each team. We denote this value as p_{range_i} of each team. We then begin with the driver with the largest *Driving* and proceed in descending order, comparing the driver's probability rating to the value of p_{range_i} of each team. We assign the driver to the team with the smallest value of p_{range_i} that is greater than the driver's probability value. For example, if we have 3 teams, named A, B, and C, with marketing budgets of $2.2\imath$, $0.8\imath$ and $1.6\imath$ respectively we can calculate that they have the following values of Equation (1), 0.77, 0.01 and 0.22. Let us consider 6 participating drivers, with Driving of $5\star$, $4.8\star$, $4.4\star$, $3.0\star$, $2.2\star$ and $1.0\star$ respectively. In this case, we would begin with the $5\star$ driver. Consider we have drawn a probability value of 0.47 from our uniform distribution for this driver. Then from our Equation (1) values, we would assign this driver to team A. We repeat this process for each driver until any team has been assigned two drivers. At this point, we recalculate the Equation (1) values for each team, first removing the team which has two drivers assigned from the calculation. We do this each and every time a team is assigned two drivers until all drivers have been assigned to a team.

3.2.2 Lap Time Calculation Function

The second component of the "seasonsim(budgetsplitnew, driverlist, circuitlist)" function is the lap time calculation. We calculate a lap time for a given driver, *driver*, driving for a given team, "Team i", using an adaptation of Equation (5). However, we ignore p_t and p_f as we consider these components to be independent of the budget split strategy of a team. We also adapt this equation to ignore random effects by not considering the R_{lap} and the

setup component of t_{perf}. The resulting equation is as follows:

$$t = t_b - 0.15(Performance), \tag{10}$$

which gives us an expected lap time for each driver. For a given circuit this equation is entirely determined by the driving ability of the driver, *Driving*, chassis investment (c_1), engine investment (e_1) and reliability investment (r_1), and is independent of lap number, remaining constant for all laps in any given race. As such, we rank the driver finishing position in any given race in ascending order by the result of Equation (10). This method reduces the computational cost of our season simulation function, as we compute Equation (10) for n drivers, looping through 10 circuits for a single season. If our method was instead dependent on the original lap time equation, Equation (5), and as such did consider random effects described in *setup*, R_{lap}, p_t and p_f, we would have to loop through between 44 and 88 laps per circuit, for n drivers across 10 circuits. Hence we would have at minimum a 44 times increase in complexity by implementing this alternative method. As such we do not consider the random effects and the saving in computational complexity allows us to run our simulation a larger number of times at the same computational cost. The final step of the lap time calculation function is to calculate the final team rankings for a given season. This is calculated according to the individual race points allocation [1] for each driver, and then the sum of the season total points of each team's associated drivers is calculated. The teams are ranked by total points, and given a cost metric score, $-metric$, as described in Equation (9).

4 Model Analysis

4.1 Budget Split Generation

In order to use our season simulation function, we must first provide three inputs, "budgetsplitnew", "driverlist" and "circuitlist"."driverlist" and "circuitlist" are given for any race season, and are taken to be known. However for some participating team, "Team 1", the budget splits of each other $n-1$ participating teams in the Formula 1 Race Strategy Competition [1] are unknown. In our competition objective, we aim to locate the optimal budget split for a given team, "Team 1", assuming that the $n-1$ budget splits are known. We start by optimising for a single, known set of budget splits because it is a less complicated case to consider, and because by developing a method to optimise the budget split of "Team 1" for any known set of $n-1$ budget splits it gives us a platform to address the case of optimising for some unknown set of $n-1$ budget splits. As such, we will use some method to predict the budget splits of each team, which we will set to be the "budgetsplitnew" input in our season simulation function. If available, we could use previous competition budget split data to develop a predictive model. However, given that the competition has not run previously with the exact same rules as our current season [1], we do not have this desired data available. Instead, we shall randomly generate a budget split for each n team, where we make some key assumptions about the behaviour of each participating team. First, as shown in 1, we will restrict reliability investment to below $2.4\imath$, as we have seen that beyond $2.4\imath$ the return on investment increase categorised by the "DNF Probability" decrease is close to 0. We will further restrict the reliability investment range for each team, r_i, to below $2.0\imath$, as beyond this investment level we see a significant increasing diminishing of the return on investment shown by the Figure 1 curve becoming rapidly increasingly shallow

with each additional $0.1\imath$ invested into r_i. In addition, we make the assumption that participating teams will desire a "DNF Probability" of below 40%. We make this assumption as we believe participating teams will want their associated drivers to finish most races. If there is a viable high "DNF Probabililty" but fast expected lap time strategy, we can see that this strategy would be high variance with teams performing well or badly predominantly down to the DNF function, that is how many races the drivers finish. In real-world racing, this would not be considered viable as it takes a significant risk with driver safety. Although the risks are less significant in our simulated competition [1], we still work with the same assumption that teams will get more satisfaction from seeing drivers finishing races. This causes us to select the floor of the range from which we will select r_i to be $1.5\imath$, as the "DNF Probability", p_{DNF}, is equal to 0.3411 at $1.5\imath$ and p_{DNF}, is equal to 0.4000 at $1.4\imath$. So from Figure 1 we have $r_i < 1.5 \implies p_{DNF} > 40\%$. Due to this assumption, we select r_i as a random number, XR_{RNG}, drawn from a uniform distribution between 1.45 and 1.95 and rounded to one decimal place.

On the marketing investment assumptions for a team, i, we have that the maximum driving ability of any driver is $5\star$. As such, any team investing above $5\imath$ could certainly gain a better return on investment for a given driver by investing in either chassis, c_i or engine, e_i, given that the investment in these variables is equal to the performances cp_i and ep_i respectively, so the return on investment is 1 for each variable. For example, consider a team, "Team 1", investing above $5\imath$ in marketing, m_1. The best possible outcome would be to sign the two drivers with the highest driving ability, $Driving$, that is $5.0\star$ and $4.8\star$. Only considering the higher rated driver, $5.0\star$, we have the return on investment given by $\frac{5}{m_1} < 1$ since $m_1 > 5$. In contrast, the return on investment for chassis and engine, c_i and e_i, are equal to 1. So we have that the return on investment for "Team 1" would always be greater invested in chassis or engine, c_i and e_i, for marketing investment greater than 5, $m_i > 5\imath$. We make the assumption that participating teams will aim to maximise the return on budget investment and so we select m_i as a random number, XM_{RNG}, drawn from a uniform distribution between -0.05 and 5.05 and rounded to one decimal place.

Once we have selected the values for reliability and marketing investment, $XR_{RNG} = r_i$ and $XM_{RNG} = m_i$, we will denote our remaining budget after subtracting marketing and reliability investment, RB_i, as $8\imath - r_i - m_i = RB_i$. From the given ranges for m_i and r_i we have that $1.0\imath < RB_i < 6.6\imath$. We can therefore select the chassis investment, c_i, for team i, to be a random number, XC_{RNG} chosen from a uniform distribution between -0.05 and $RB_i + 0.05$ rounded to one decimal place. We can then define engine investment, e_i to be the remaining budget after chassis, marketing and reliability investment have been made. That is, $e_i = RB_i - c_i$. An example of this budget selection for "Team 1" is that first we randomly select our marketing investment as $0.3\imath$, that is, $XM_{RNG} = m_1 = 0.3\imath$. We then randomly select $XR_{RNG} = r_1 = 1.7\imath$, giving our reliability investment as $1.7\imath$. We calculate $RB_1 = 8\imath - 1.7\imath - 0.3\imath = 6\imath$, and select c_i from a uniform distribution between -0.05 and 6.05. The chassis investment c_i is randomly selected to equal $5.1\imath$, and so we can calculate $e_i = 6\imath - 5.1\imath = 0.9\imath$, giving our engine investment as $0.9\imath$. Hence the resultant budget split for "Team 1" is given as follows, $[m_i, r_i, c_i, e_i] = [0.3\imath, 1.7\imath, 5.1\imath, 0.9\imath]$. We repeat this process for each n team, which gives us a sample set of budget splits, BS_1, shown in Table 1.

Table 1: Randomly generated budget split set, BS_1, for $n = 11$ participating teams.

Team	Marketing	Reliability	Chassis	Engine	SUM
1	0.3	1.7	5.1	0.9	8
2	1.1	1.9	4.6	0.4	8
3	2.4	1.7	0.5	3.4	8
4	3	1.8	2.4	0.8	8
5	2.8	1.6	0.4	3.2	8
6	2.2	1.5	2.1	2.2	8
7	2	1.8	4	0.2	8
8	1.3	1.6	2	3.1	8
9	2.2	1.5	3.4	0.9	8
10	0.4	1.7	2.4	3.5	8
11	3.5	1.8	1.9	0.8	8

4.2 Simulation Analysis

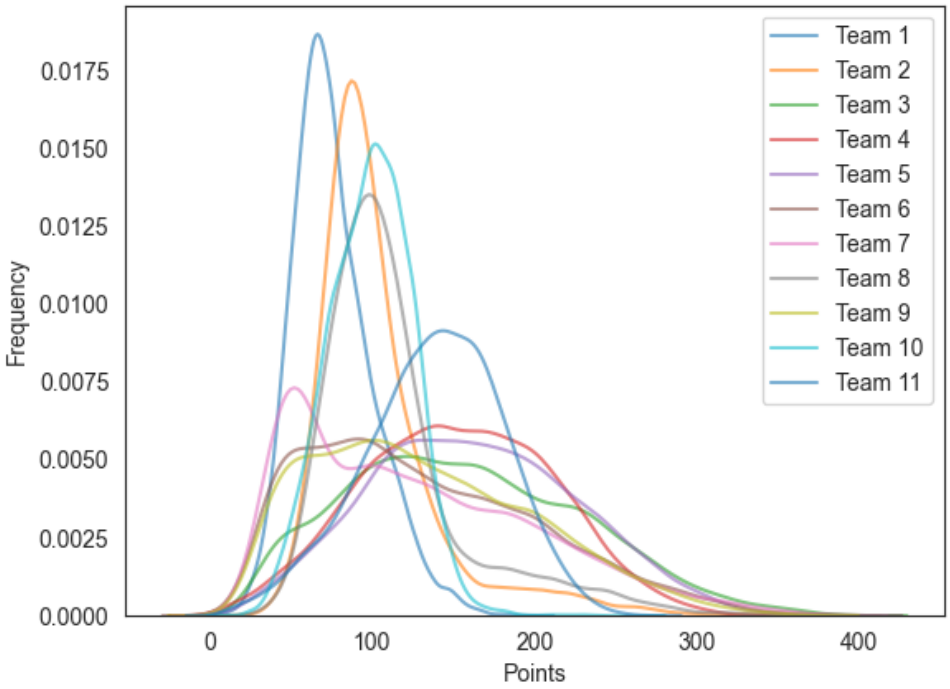

Figure 2
Smoothed density plots of season points for $n = 11$ teams for budget split set BS_1. Data generated through 10000 calls to our season simulation function.

Now that we have generated a set of $n = 11$ budget splits, BS_1, we can analyse the data that our season simulation function produces by setting it to be the function input, "budgetsplitnew" $= BS_1$. As we discussed in Section 3.2.1, our season simulation function is probabilistic due to the stochastic driver allocation and driver DNF processes. This

implies that both the total season points and our cost metric, $-metric$, will follow some probability distribution, which will differ for each $n = 11$ teams for a given set of budget splits, BS_1. We examine this by using our running our season simulation function 10000 times to generate a large sample of total season points for each $nteam$. In Figure 2 we have produced smoothed density plots for all $n = 11$ teams, which show the distributions of the total season points for each team. We can see from Figure 2 that "Team 4" and "Team 5" have sample distributions that appear to reasonably approximate bell curves. In contrast, "Team 3", "Team 6", "Team 7" and "Team 9" show positive skews, where the majority of data points for each of these teams are clustered towards the left side of the distribution. For "Team 1" and "Team 10" we see much narrower distributions, with a slight positive and a slight negative skew respectively. "Team 2" and "Team 8" both show large tails towards the right side of the distribution, also possessing positive skews but in a more pronounced manner than the previous positively skewed distributions we have discussed. Finally, "Team 11" shows a negatively skewed distribution, where the majority of data points are clustered towards the right-hand side.

5 Particle Swarm Optimisation (PSO)

5.1 Method

Using our season simulation subroutine we aim to determine the budget split for a given team, "Team 1", which minimises our defined cost metric, $-metric$, when given the budget splits of $n - 1$ other participating teams. Our season simulation subroutine allows us to generate performance data and our cost metric score for a given team, "Team 1", for any combination of budget splits for some number, n, of participating teams. An initial idea would be to test each possible budget split for "Team 1" by simulating the season some large number, s, of times in order to return an accurate expected cost metric for each different possible budget split. Let us consider the number of feasible budget splits we would be required to test. Feller [2] set out a stars and bars method for calculating the number of ways to divide some number of identical items, j between some number of bins, q, using the following formula:

$$\binom{j+q-1}{q-1} = \frac{(j+q-1)!}{(q-1)!(j)!}, \tag{11}$$

which we can apply to calculate the number of feasible budget splits. Given that "Team 1" has a budget of $8\imath$, which can be spent with a precision of up to one decimal place, then this implies there are 81 identical items to divide between the four variables that encompass the budget split. That is, we have $j = 81$ and $q = 4$. Applying this to Equation (11) gives us the following:

$$\binom{81+4-1}{4-1} = 95284, \tag{12}$$

which is the number of possible budget splits "Team 1" could take, on the condition that it uses the entire budget of $8\imath$. Let us define $comp$ to be the computational complexity of one call to our season simulation function. Then we have the complexity of testing every possible budget split to be $comp \cdot s \cdot 95284$. This is not feasible due to the limited computing resources at our disposal, so we shall aim to use an alternative method to locate the budget split for "Team 1" that minimises our cost metric. Another idea is that we may choose to only test a random sample of possible budget splits, and settle on the best budget split

out of those tested. However, as we have demonstrated with Equation (12), this method gives us a high probability of not considering the best possible budget splits if we choose a small sample. Alternatively, if we choose a large sample, we run into the previous problem of facing a computation that is too complex to be feasible with our existing facilities. As such, we seek to implement an alternative method. The criterion for the method we wish to implement is to attain a significant reduction in computational complexity compared to testing all possible budget splits. We also aim for our method to converge to or close to a global minimum in the search space, and we aim to be able to escape any local minimum at which our method may be at risk of getting trapped. By this, we mean that our particles are unable to navigate away from some local minimum, as they falsely believe it to be a global minimum of the search space.

5.2 Heuristic Tools

Lee and El-Sharkawi [5, Introduction] set out two advantages of using heuristic tools to facilitate solving optimisation problems. The first advantage is that compared to alternative approaches the development time is considerably shorter. This benefit is valuable to our use case, given the limited time frame we have to develop, implement and analyse our method to solve the defined optimisation problem. The second advantage is that systems are resilient to cases of missing or noisy data, and are considered robust. For our optimisation problem, we do not anticipate missing or noisy data to be a problem, as we are able to generate our cost data using our defined season simulation subroutine. However, if we wished to adapt our method in future to a real-world use case, having a robust method that is resilient to changes in the quality of data available could be considered an advantage. The drawbacks of heuristic methods include that the solution generated is not guaranteed to be the optimal solution and that the time required to generate a "near optimal" solution can be large in an unlucky case [4]. We define a "near optimal" solution to be one within a close tolerance of the optimal solution. Given that our optimisation problem is one of incomplete information, where we must make some prediction or assumption about the budget split choices of each participating team, we consider a heuristic approach as viable despite the lack of a guarantee that it will converge to the optimal solution for a given set of budget splits. This is because due to the computational complexity, the exact optimal solution is unfeasible to obtain. In addition, due to the budget splits of the competing teams being unknown, an exact optimal solution is unknowable. To illustrate the reason for this choice, consider we are able to generate the optimal budget split, denoted O_1, of "Team 1" for a given set of $n-1$ budget splits, denoted BS_1. Now let us consider that O_1 may not be the optimal budget split for any other possible set of $n-1$ budget splits, denoted BS_2. Now consider that from (12), if any team can take any one of 95284 budget splits then, by (11), the possible sets of budget splits, denoted as BS_2, can be calculated as:

$$\binom{n-1+95284-1}{95284-1}-1, \tag{13}$$

If we have $n=11$ participating teams we have the following calculation:

$$\binom{11-1+95284-1}{95284-1}-1 \approx 1.701 \cdot 10^{43}, \tag{14}$$

which describes the number of different budget split combinations for the 10 teams that a given team, "Team 1", would have to compete against. It is computationally unfeasible for

us to consider each possible set of budget splits. As such, we will fix the budget splits of the competing teams, to be equal to the sample budget split we generated, BS_1, and aim to generate an optimal budget split for "Team 1".

5.3 Particle Swarm Optimisation Definition

Particle Swarm Optimisation (PSO) is an optimisation method that draws analogies from the movement of a flock of birds collectively searching for a source of food [6]. To implement the PSO algorithm, we place some number of particles within the search space of the function we aim to optimise. Each particle then evaluates the objective function at its current location. A given particle computes the evaluation of the objective function by passing its position as the function input, therefore the dimension of our search space must be equivalent to the arity, a_f, of the objective function, g. We denote a particle's evaluation of the objective function as its fitness., f Each particle combines aspects of the history of its current and best fitness, along with random perturbations, in order to determine how it moves through the search space. Once all particles have moved location the next iteration begins. Eventually, after some number of iterations, the entire particle swarm is expected to move close to the optimal fitness value for the objective function [6].

Each individual particle in the swarm is composed of three vectors, each equivalent in dimension to the search space, thus each having dimension a_f. For a particle j, the vectors are current position, $\mathbf{c\vec{p}_j}$, best position, $\mathbf{b\vec{p}_j}$, and velocity, $\mathbf{\vec{v}_j}$. At each iteration of the PSO algorithm, we evaluate the current position of each particle, $\mathbf{c\vec{p}_j}$, by passing its position as input to our objective function and calculating its fitness. If the particle position gives a better fitness than any position that has been previously evaluated then the position coordinates are stored in the best position vector, $\mathbf{b\vec{p}_j}$. We additionally store the value of the position attaining the best fitness value so far in a variable that we denote $pbest_j$, referring to "previous best", which allows us to compare the best position of a particle across iterations. We choose new points by adding $\mathbf{c\vec{p}_j}$ to $\mathbf{\vec{v}_j}$, where the algorithm adjusts $\mathbf{\vec{v}_j}$ for each particle after each iteration. We update the velocity of a particle, j, as follows [6, 13]:

$$\mathbf{\vec{v}_j} \leftarrow w * \mathbf{\vec{v}_j} + c_1 r_1 \cdot (\mathbf{b\vec{p}_j} - \mathbf{c\vec{p}_j}) + c_2 r_2 \cdot (pbest_j - \mathbf{c\vec{p}_j}), \tag{15}$$

where r_1 and r_2 are random parameters taken from a uniform distribution between 0 and 1. w is the inertia weight coefficient and c_1 and c_2 are acceleration coefficients. A higher w value allows a particle to be more influenced by its previous velocity, which helps in the exploration of the global search space, called "global search". Conversely, a lower value of w leads to a particle being less reliant on its previous velocity, encouraging local search. $c1$ determines how much weight the velocity equation places on a particle's own best position. $c2$ determines the amount of weight the velocity equation places on the best position of other particles in the swarm [8]. We calculate the updated position of a particle j by [13]:

$$\mathbf{c\vec{p}_j} \leftarrow \mathbf{c\vec{p}_j} + \mathbf{\vec{v}_j}, \tag{16}$$

where the current position of a particle is its previous current position added to its velocity. We can denote BP as the matrix containing the particle's best positions, $\mathbf{b\vec{p}_j}$, for all particles in the swarm, and CP to be the matrix containing the particle's current positions, $\mathbf{c\vec{p}_j}$. Then, Algorithm 1 describes a process for implementing PSO [13, 14]:

Algorithm 1 Particle Swarm Optimization Algorithm

$S \leftarrow$ InitSwarm();
fitness $\leftarrow$ Eval(S)
$pbest \leftarrow S$;
$BP \leftarrow$ best(fitness);
while Stop criterion not reached **do**
  $V \leftarrow w * V + c_1 r_1 \cdot (pbest - S) + c_2 r_2 \cdot (BP - S)$;
  $S \leftarrow S + V$;
  fitness $\leftarrow$ Eval(S);
  if best(fitness) is better than fitness($pbest$) **then**;
    $pbest \leftarrow$ best(fitness);
  end if
  if fitness(S) is better than fitness(BP) **then**;
    $BP \leftarrow S$;
  end if
end while

where InitSwarm() initialises the particles in random initial positions throughout the search space and with random initial velocities.

5.4 Penatly Function

We consider the case where the budget split of each team competing in the Formula 1 Race Strategy Competition is known. We have set these investment values randomly as described in Section 4.1, and denote the set of budget splits generated as BS_1. The random set of budgets we generated as BS_1 is described in Table 1. To implement PSO, we fix the known budget splits of each team, "Team i", $\{i \in \mathbb{N} \mid 2 \leq i \leq 11\}$, as they were initially randomly defined in BS_1. We treat the budget split of "Team 1" as a variable, defined by the position of each particle in our swarm. That is, when we evaluate the fitness of a particle we substitute its current position for the budget split of "Team 1" in BS_1, where all other team's budget splits remain unchanged. We will denote this updated budget split list as BS_j. Each particle, p, in our swarm takes position $[m_{1j}, c_{1j}, e_{1j}]$, where we then define $r_{1j} = 8\imath - (m_{1j} + c_{1j} + e_{1j})$. That is, we denote the position of a particle, p, as values of marketing, chassis and engine investment. We then define the reliability investment as $8\imath$, such that the sum of all variables always adds up to our budget of $8\imath$. We recognise that $m_{1p} + c_{1p} + e_{1j} > 8 \implies r_{1p} < 0\imath$. So since reliability investment can not be negative, we consider a particle position, $[m_{1p}, c_{1p}, e_{1j}]$, that sums to greater than $8\imath$ to be non-admissible, as it violates the reliability constraint. Given we have a total budget of $8\imath$, we implement the following bounds for each particle, $0 \leq m_{1j}, c_{1j}, e_{1j} \leq 8$. We want to increase the likelihood of particles taking an admissible position, that is where reliability investment is non-negative, $0 \leq r_{1p} \leq 8$. So we develop a penalty function, pf, to increase the value of the objective function, which we seek to minimise, for non-admissible positions. The penalty function acts only on particles that take a non-admissible position, that is where $8 \leq (m_{1j} + c_{1j} + e_{1j})$. Currently, there are no methods to develop appropriate penalty functions for constrained optimisation problems other than trial-and-error [12], so this is the method we use to develop our penalty function, pf. In developing the design of our penalty function, we aim to guide any non-admissible particles towards an admissible position. We do this by reducing in size the penalty applied to a non-admissible particle in

proportion to its distance outside the region of admissibility, where the region of admissibility is defined by $0 \leq m_{1j}+c_{1j}+e_{1j} \leq 8$. We define our penalty function on the current position of a particle, $\mathbf{c\vec{p}_j} = [m_{1j}, c_{1j}, e_{1j}]$ as follows:

$$pf(\mathbf{c\vec{p}_j}) = pf([m_{1j}, c_{1j}, e_{1j}]) = (m_{1j} + c_{1j} + e_{1j} - 8)^2, \tag{17}$$

where the penalty function converts our constrained optimisation problem into an unconstrained one, by modifying the objective function in a way we will later describe. Utilising a penalty function is the most efficient tool that we can implement to solve a constrained optimisation problem. Alternative techniques include rejecting and repairing generated non-admissible positions, however, this is a time-consuming process and is often unsuccessful [10].

5.5 Fitness Function

In defining our objective function we make use of our previously defined season simulation function, "seasonsim(budgetsplitnew, driverlist, circuitlist)". We pass as input to our season simulation the previously defined driverlist and circuitlist data frames, along with setting budgetsplitnew = BS_j. That is, we pass the BS_1 budget split data frame, altered with the budget split of "Team 1" set to be the position of particle j. For example if a particle, j, has position $[1.8, 1.2, 3.5]$, we calculate $r_1 = 8\imath - m_1 - c_1 - e_1 = 8\imath - 1.8\imath - 1.2\imath - 3.5\imath = 1.5\imath$. As such we would obtain BS_j by updating BS_1 with the position of particle j, and its implied reliability investment, assigned to "Team 1" as shown in Table 2.

Table 2: Shows BS_j, in which BS_1 is updated with the position of particle j.

Team	Marketing	Reliability	Chassis	Engine	SUM
1	1.8	1.5	1.2	3.5	8
2	1.1	1.9	4.6	0.4	8
3	2.4	1.7	0.5	3.4	8
4	3	1.8	2.4	0.8	8
5	2.8	1.6	0.4	3.2	8
6	2.2	1.5	2.1	2.2	8
7	2	1.8	4	0.2	8
8	1.3	1.6	2	3.1	8
9	2.2	1.5	3.4	0.9	8
10	0.4	1.7	2.4	3.5	8
11	3.5	1.8	1.9	0.8	8

We then calculate the expected value of our cost metric, $-metric$, by running "seasonsim(budgetsplitnew, driverlist, circuitlist)" some number, s, of times. We calculate the sum of $-metric$ for each "Team 1", and divide this by s to give us the expected value of $-metric$ for "Team 1". That is, for a particle , j, at current position, $\mathbf{c\vec{p}_j}$, the objective function is defined as follows:

$$g(\mathbf{c\vec{p}_j}) = \frac{1}{s}\sum_{j=1}^{s} q(BS_j), \tag{18}$$

where $q(BS_j)$ is the value of $-metric$ for "Team 1" returned from a single call to the season simulation function, "seasonsim(budgetsplitnew, driverlist, circuitlist)", where budgetsplitnew = BS_j. The number of calls to the season simulation function, s, can be considered to

be a function parameter. Increasing s will improve the accuracy of the objective function at the cost of increasing the computational complexity. Our implementation of the fitness function takes place as follows:

$$fitness(\mathbf{c}\vec{\mathbf{p}}_\mathbf{j}) = \begin{cases} pf(\mathbf{c}\vec{\mathbf{p}}_\mathbf{j}), & \text{if } 0 \leq m_{1j}, c_{1j}, e_{1j} \leq 8. \\ g(\mathbf{c}\vec{\mathbf{p}}_\mathbf{j}), & \text{otherwise.} \end{cases} \tag{19}$$

That is, non-admissible particles take the value of the penalty function, pf, where $pf > 0$ by definition, Equation (17). The further the distance from the region of admissibility, the larger the value of pf. Hence non admissible particles have a high likelihood of moving towards the admissible region over a sufficiently large number of iterations. Admissible particles take the value of the objective function, g, where $g \leq 0$ by definition, Equation (18). As such, admissible positions will always have a lower fitness value, which we seek to minimise, than non-admissible positions. This means that given enough iterations, particles will move towards positions that are admissible and away from positions that are non-admissible.

5.6 Initial PSO Implementation

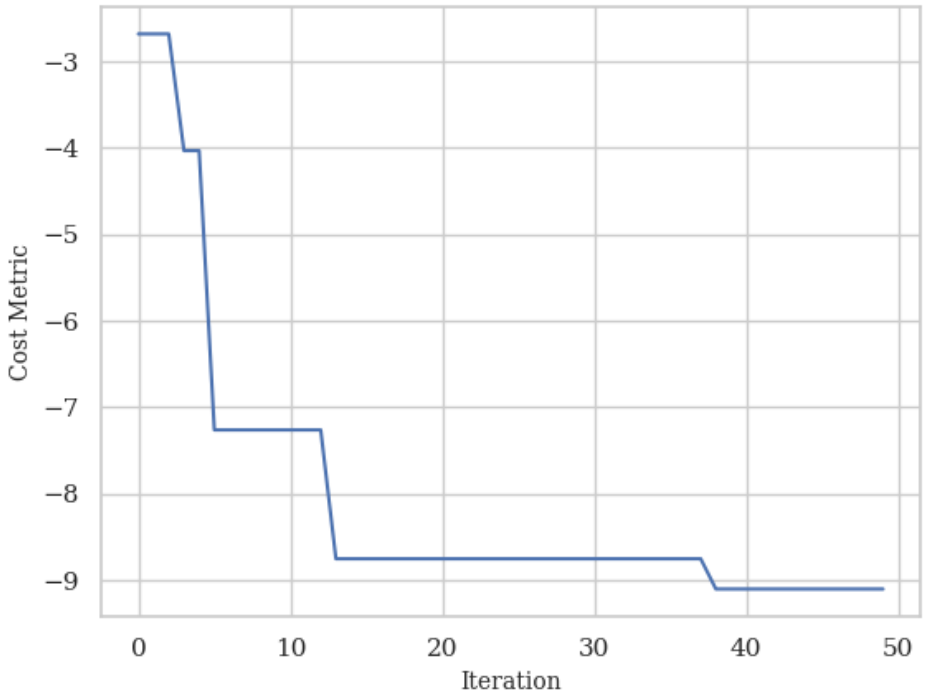

Figure 3
Cost history for PSO implementation 1, seeking to minimise the cost metric. Algorithm parameters: $c1 : 1$, $c2 : 1$, $w : 0.9$, swarm size: 15 particles, 50 iterations, objective metric season simulation function calls: 100. Best particle position: [2.73513616, 0.54609236, 3.15448412], lowest fitness: -9.1.

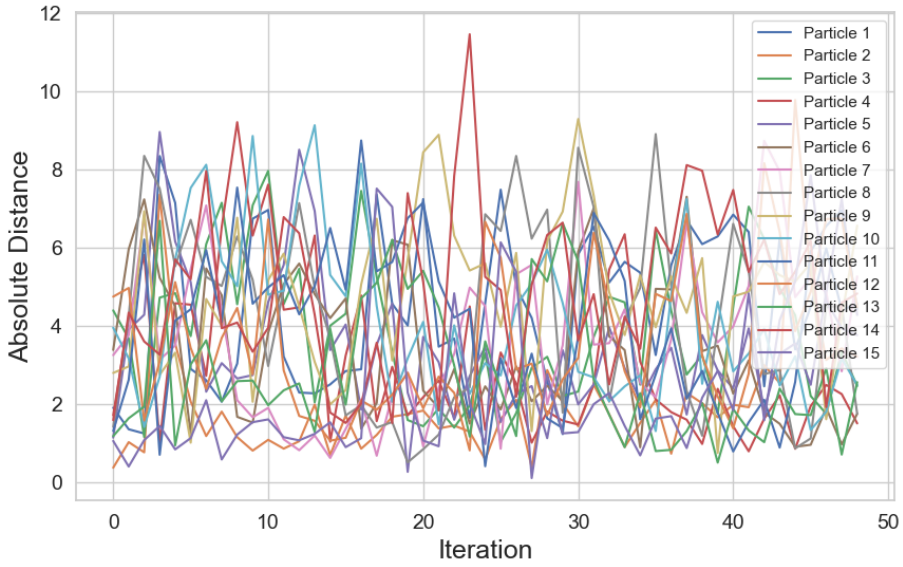

Figure 4
Particle movement per iteration for PSO implementation 1. Algorithm parameters: $c_1 : 1$, $c_2 : 1$, $w : 0.9$, swarm size: 15 particles, 50 iterations, objective metric season simulation function calls: 100. Best particle position: [2.73513616, 0.54609236, 3.1544841], lowest fitness: -9.1.

By implementing the PSO algorithm on our generated budget split set, BS_1, we aim to minimise the fitness function, f, which we have defined to be the negative of our satisfaction metric. To do this, we must select parameters for our PSO algorithm, $c1$, $c2$ and w, which we previously defined. We seek parameters that allow the particles to sufficiently explore the search space, but allow the algorithm to be computed in a reasonable time frame with the computational resources available. When we have selected parameters we deem suitable, we will run the code seen in Appendix A to produce an initial run of our PSO algorithm, alongside associated performance data and figures. The cost metric history of this initial run of our PSO algorithm is seen in Figure 3, in which we have made a reasoned initial estimate of the parameters based on our understanding of the domain of the problem which we will outline as follows. (Shi and Eberhart, 1998) suggest that suitable inertia weight is dependent on the maximum velocity, $V - max$, of the PSO implementation. Through analyse of experimental results for a benchmark problem, Shaffer's F6 function [11], it is suggested that for good performance, a value of w between 0.8 and 1 is recommended. This selection of inertia weight is aimed at striking a balance between local and global exploration of the search space. As such, we hope that it will require a smaller number of iterations, on average, in order to converge [9]. For c_1 and $_2$ we set $c_1 = c_2 = 1$. We selected these parameters because for an unconstrained simplified PSO implementation which includes the inertia parameter w, particle trajectory converges if the following condition holds [8]:

$$1 > w > \frac{1}{2}(c_1 + c_2) - 1 \geq 0, \tag{20}$$

hence our parameter selection satisfies this equation. We choose $c_1 = c_2$ as particles are often most effective when c_1 and c_2 coexist with close balance [8]. We note that for functions including our fitness function, which contains stochastic components, a selection of parameters w, c_1 and c_2, which violate Equation (20) may still lead to swarm convergence, as we will later discuss. This is because Equation (20) was derived for simplified PSO systems that do not contain a stochastic component. We select $s = 100$ in Equation (18) as this gives a sample size large enough to compute our $fitness$ metric to reasonable accuracy, without occurring too large of a computational cost. For our iteration number, we considered implementing convergence criteria that would halt our iterations when we judged convergence to have occurred, by some absolute distance threshold. However, we opted to instead run our PSO algorithm for a fixed number, 50, of iterations. We do this in order to better compare the success of different parameter PSO implementations by fixing the computational cost as equal across implementations.

From the cost history of our initial implementation depicted in Figure (3) we see that the global best position, [2.73513616, 0.54609236, 3.15448412], is found at iteration 38. To convert this to a viable budget split for team 1, we first round each coordinate to one decimal place. $r_{1j} = 8\imath - (m_{1j} + c_{1j} + e_{1j})$, indicating the best budget split strategy for "Team 1" was found to be $[m_{1j}, r_{1j}, c_{1j}, e_{1j}] = [2.7\imath, 1.6\imath, 0.5\imath, 3.2\imath]$. From Figure (3) we see that the global best cost metric, which we define as the lowest fitness function value for any particle across all previous iterations, does decrease as expected. In Figure (4) we have plotted the absolute distance each particle moves between iterations. We expect that as a swarm of particles converges, the absolute distance each particle travels should tend to 0. We can see from Figure (4) that the absolute particle distance travelled between iterations does not converge. Instead, we can see that the absolute distance moved maintains its high volatility throughout all 50 iterations, and both the average distance moved and variance in distance moved both appear to remain constant. This indicates that our parameter values are two large which makes the velocity, $\vec{\mathbf{V}}_\mathbf{j}$, of a particle j too large. The large velocity of the particles means that instead of moving closer towards the optimal position, each particle takes huge steps, often moving a long way past the optimal point they intended to move towards. This means that from iteration to iteration particles do not converge on the optimal point, instead they jump wildly from point to point constantly trying to improve on the global best point. Although this strategy has found a viable position, we will aim to refine our parameters, w, c_1 and c_2. The goal of this is to allow our algorithm to converge to a global "near optimal" position, which we expect will locate a lower fitness value, and thus a better budget split strategy for "Team 1".

5.7 Analysis

In order to refine our parameter choices, we decided to run a series of performance tests in which we implemented our particle swarm optimisation function for different combinations of parameter choices. Due to the computational cost of running our PSO implementation, we were unable to generate large samples of PSO implementations for any one set of parameter values. Instead, we ran initial tests for a range of parameter combinations and validated successful parameter choices by generating a small sample of implementations. Although this is a limitation as our method does not give us accurate insight into the level of variance that may occur between PSO implementations of the parameters, it does give us a higher confidence in our observations compared to only considering a single instance.

The second instance of parameter values we will analyse are $w = 0.42$, $c_1 = 0.3$ and

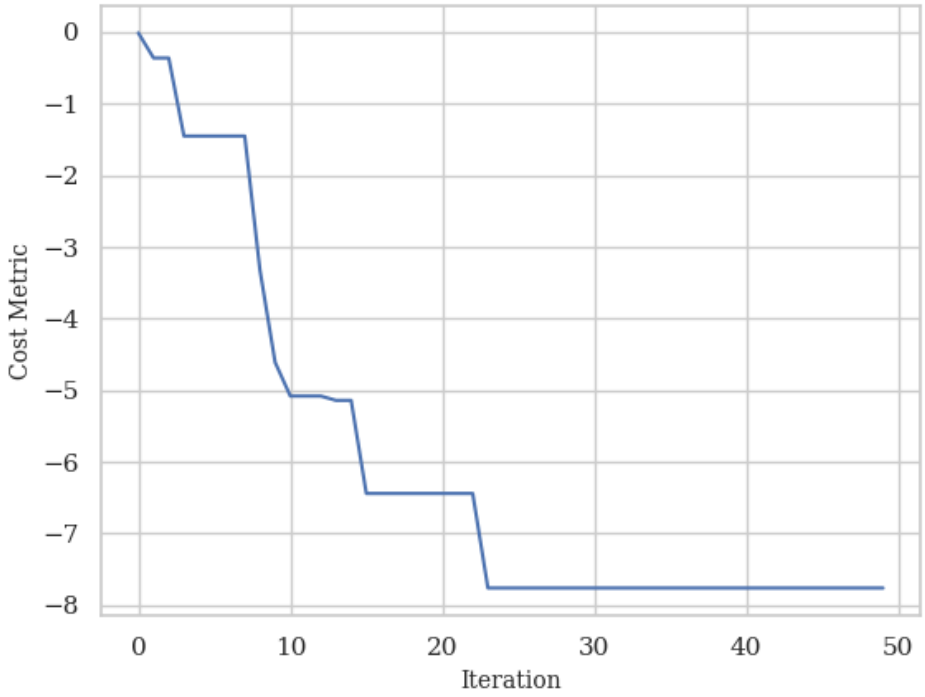

Figure 5
Cost history for PSO implementation 2, seeking to minimise the cost metric. Algorithm parameters: $c1 : 0.18$, $c2 : 0.3$, $w : 0.42$, swarm size: 15 particles, 50 iterations, objective metric season simulation function calls: 100. Best particle position: [1.79255206, 2.81055583, 1.86159528], lowest cost: -7.76.

$c_2 = 0.18$, in PSO implementation 2. Across a small sample of 5 performance tests, this set of parameter values consistently demonstrated a global best cost metric that appeared to converge and critically the iteration-wise distance between particles also converged to close to 0 across all tests. Figure 5 shows the cost metric history for a single PSO implementation taken from our sample. From Figure 5 we can see the cost history decreases across a larger number of steps than our previous implementation, before converging to a minimum at iteration 23. The minimum cost found by this implementation was 7.76. Figure 6 shows the iteration-wise absolute distance a particle moves. The figure shows that the absolute distances converge close to 0 at around iteration 23 which is when the lowest cost metric value was found, as we would expect. This implies that the particles became stuck in either a local or global minimum at iteration 23. Given that our first implementation found a lower cost metric, -9.1, at position [2.73513616, 0.54609236, 3.15448412] we know that our implementation shown in Figure 5 can not have found or be close to a global minimum. It must follow that the convergence in Figure 6 is to a local minimum, from which the particles have been unable to escape. This is not unexpected, as in this implementation $c_1 = 0.18 < 0.3 = c_2$. From Equation 15 we can see that when $c_1 \ll c_2$, particles are

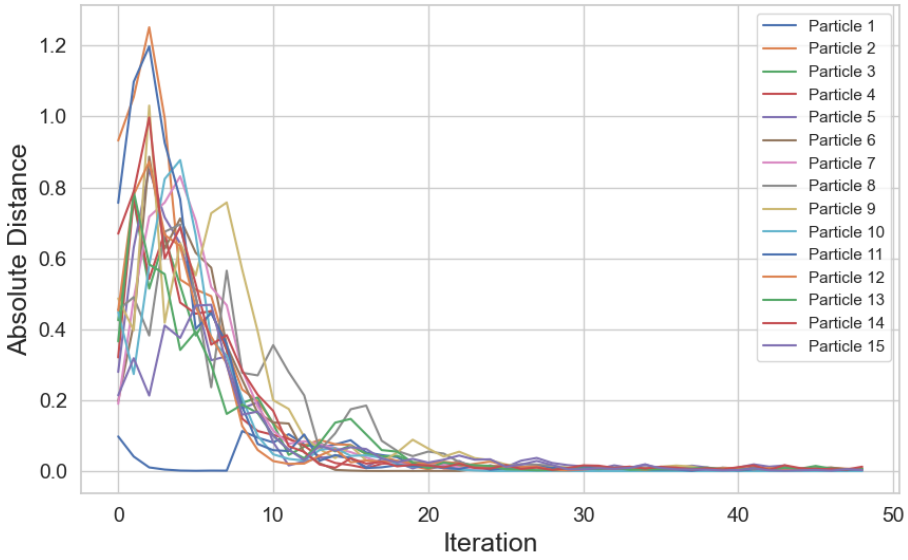

Figure 6
Particle movement per iteration for PSO implementation 2. Algorithm parameters: $c1 : 0.18$, $c2 : 0.3$, $w : 0.42$, swarm size: 15 particles, 50 iterations, objective metric season simulation function calls: 100. Best particle position: [1.79255206, 2.81055583, 1.86159528], lowest cost: -7.76.

more attracted to the global best position than their only personal best position. This is more likely to lead to premature convergence to a local minimum, as opposed to particles being encouraged to explore the search space [7, 8]. We address this premature convergence in PSO implementation 3 by adjusting our parameters such that $c_1 \gg c_2$. We shall fix $w = 0.42$, as we have seen that this inertia weight was successful in attaining convergence of our swarm.

We take $w = 0.42$, $c_1 = 0.18$ and $c_2 = 0.3$ for our PSO implementation 3. Across a small sample of 5 performance tests, this set of parameter values consistently demonstrated a global best cost metric that appeared to converge and the iteration-wise distance between particles also converged to close to 0 across all tests. Additionally, this implementation found on average lower cost metric values than PSO implementations 1 and 2. A single implementation taken from the sample is shown in Figure 7, in which we see that the cost metric value converges very quickly, with the lowest metric, -11.49, being obtained at position [2.96819077, 0.05462524, 3.31977874], at iteration 10. The much quicker convergence of PSO implementation 3, compared to previous implementations, suggests that this may also be a local minimum and that the particles have not sufficiently explored the search space. However, by obtaining a cost metric of -11.49, much lower than in implementations 1 and 2, it is plausible that implementation 3 was able to find a global minimum much quicker, and thus also converge quicker. The absolute distance particles moved per iteration, shown in Figure 8, does provide evidence for the latter explanation. Figure 8 shows that not all particles converge to travel an absolute distance of close to 0 between iterations until around

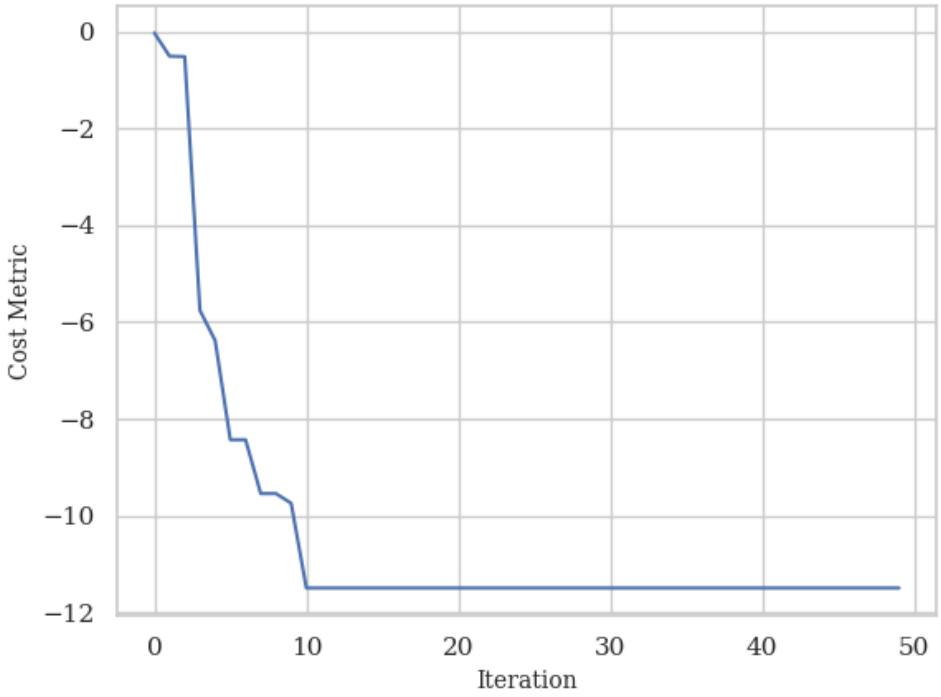

Figure 7
Cost history for PSO implementation 3, seeking to minimise the cost metric. Algorithm parameters: $c1 : 0.3$, $c2 : 0.18$, $w : 0.42$, swarm size: 15 particles, 50 iterations, objective metric season simulation function calls: 100. Best particle position: [2.96819077, 0.05462524, 3.31977874], lowest cost: -11.49.

iteration 13. We see that 3 particles were still exploring the search space when the lowest cost metric was found at iteration 10. However, it is true that 3 particles are unlikely to be a large enough swarm to locate a global minimum, and it is likely that these particles were still moving at iteration 10 because they were the furthest away from the minimum that they were travelling to converge to. Another possibility is that the variance in our objective function exaggerated the fitness score of the optimal position. To analyse this, we first find the budget split strategy associated with the particle's global best position at $[2.9\iota, 1.6\iota, 0.1\iota, 3.4\iota] = [m_1, c_1, c_1, e_1]$. We then calculate the expected cost metric, $-metric$, for $s = 1000$ in Equation (18), for the budget split set we shall denote BS_3. Increasing the sample size, s, gives us a more accurate expected value of $-metric$, as seen in Table 3.

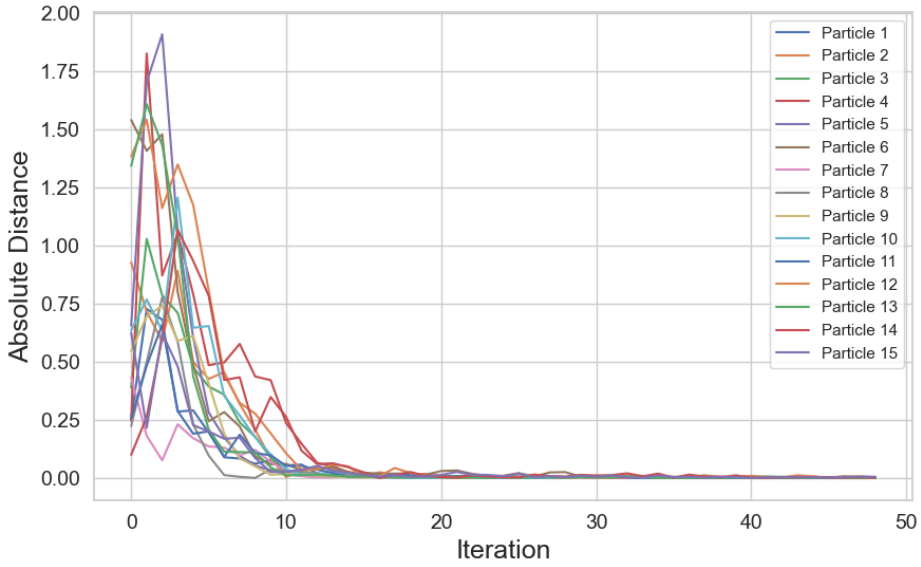

Figure 8
Particle movement per iteration for PSO implementation 3. Algorithm parameters: $c1 : 0.3$, $c2 : 0.18$, $w : 0.42$, swarm size: 10 particles, objective metric season simulation function calls: 100. Best particle position: [2.96819077, 0.05462524, 3.31977874], lowest cost: -11.49.

Table 3: Shows BS_3 and the associated expected cost metric, $-metric$, where $s = 1000$.

Team	Marketing	Reliability	Chassis	Engine	$-metric$
1	1.8	1.5	1.2	3.5	-8.485
2	1.1	1.9	4.6	0.4	-0.902
3	2.4	1.7	0.5	3.4	-7.766
4	3	1.8	2.4	0.8	-7.771
5	2.8	1.6	0.4	3.2	-8.315
6	2.2	1.5	2.1	2.2	-5.013
7	2	1.8	4	0.2	-5.386
8	1.3	1.6	2	3.1	-1.597
9	2.2	1.5	3.4	0.9	-5.135
10	0.4	1.7	2.4	3.5	-0.072
11	3.5	1.8	1.9	0.8	-4.558

We see from Table 3 that the expected $-metric$ value for "Team 1" is -8.485 from our sample of $s = 1000$ calls to the season simulation function. This is much less than the global minimum value obtained in PSO implementation 3. We can determine from this that it is very likely that the global best strategy in PSO implementation 3 was exaggerated due to variance. This suggests that our method could be improved by increasing the number of calls to the season simulation function within our PSO implementation. However, the global

best strategy obtained in PSO implementation 3, $[2.9\imath,\ 1.6\imath,\ 0.1\imath,\ 3.4\imath] = [m_1, c_1, c_1, e_1]$, is shown in Table 3 to have the minimum value of $-metric$ across all $n = 11$ participating teams. So although this strategy is unlikely to be the global optimal, it is still expected to be a strategy that would leave us more satisfied than any other team in the long run as judged by our metric.

6 Discussion

In this paper, we have demonstrated the applications of a season subroutine for understanding the Formula 1 Race Strategy Competition [1] problem dynamics. We have shown that the expected points distribution of a random sample of teams can vary significantly in width and distribution dependent on team budget strategy. Interestingly, we observed that the most competitive and effective strategies we were able to produce only possessed relatively small statistical advantages over competing teams. The implication of this is that while superior strategies can be obtained, success in any one season of the Formula 1 Competition is predominantly down to luck. We noted that the DNF probability draw and the driver allocation components were the competition features that caused variance in the season simulation function, given that we ignored other random effects. Overperformance relative to investment in both the driver allocation process and the number of driver DNFs had very significant effects on team performance in terms of expected points and the expected value of $-metric$.

Our decision to aim to optimise a team's budget split assuming we knew the budget splits of competing teams had significant benefits. We were able to reduce an element of problem complexity while still delivering a method that is able to obtain competitive budget split strategies. In addition, if we are able to obtain past strategy data that we deem relevant to the current Formula 1 Season Competition, we could improve the accuracy of our optimisation model by attempting to predict the budget split of competing teams. Beaume (2022) demonstrates this successfully using artificial intelligence [1]. The drawback of our decision is that if the randomly generated set of budget splits, BS_1, differs significantly from the actual budget split set then it is likely to have a large impact on our optimal strategy. For further research, we would benefit from undertaking an analysis of the variance in the cost metric, $-metric$ of our optimal strategy when it is tested against differing budget split sets. The difficulty in undertaking this analysis is that the search space of possible budget split sets is so large, it is not feasible to consider all or even the most possible sets. Instead, we would have to rely on the results for some small sample being representative of the whole space.

Our design and implementation of a penalty function, pf, allowed us to remove the variable sum constraint from our problem. Additionally, we showed that it successfully moved non-admissible particles towards and into admissible positions. Reducing the dimension of our optimisation problem by defining $r_{1j} = 8\imath - (m_{1j} + c_{1j} + e_{1j})$ allowed us to ensure that all admissible particles used the entire $8\imath$ budget. This made our optimisation more efficient, as the computational cost was not wasted on particles that did not use the entire budget and so could not be optimal. The success of our particle swarm implementation demonstrates that it is an effective technique for applying to constrained optimisation problems when coupled with a suitable penalty function [12]. We have shown that $w = 0.42$ resulted in a convergence of particles within 50 iterations, whereas $w = 0.9$ did not. In addition, we saw that values of $c_1 = 0.3$ and $c_2 = 0.18$ resulted in the lowest obtained values of our cost

metric. However, we identified a significant drawback in our PSO method. The variance in our fitness function when we set $s = 100$ resulted in inaccurate global best positions. To address this would require increasing our computational resources, or to reduce the number of iterations or particles in our swarm. Alternatively, we could attempt to better optimise our season simulation function by reducing its complexity. Despite our issue with fitness function variance, we obtained a competitive strategy for "Team 1", $[2.9\imath,\ 1.6\imath,\ 0.1\imath,\ 3.4\imath] = [m_1, c_1, c_1, e_1]$, which we found to have the lowest expected metric value, $-metric$, compared to every team competing in BS_1.

7 Acknowledgments

Thank you to [redacted] for his advice, feedback and time.

8 References

[1] Beaume, Cédric. Formula 1 Strategy Competition. `http://www.cbeaume.com/download/F1StratCompetition.pdf`, October 2022. [Accessed March 8 2023].

[2] Feller, William. *An introduction to probability theory and its applications*. Wiley series in probability and mathematical statistics. Wiley, New York ;, third edition, revised pr. edition, 1968.

[3] Beaume, Cédric. On BURP Investments. `http://www.cbeaume.com/download/Teaching_howBURPinvests.pdf`, December 2021. [Accessed March 8 2023].

[4] Feng-Sheng Wang and Li Hsunan Che .Eds. *Heuristic Optimization*. Springer New York, New York, 2013. In: Cho, K.-H. (Kwang-Hyun), Encyclopedia of systems biology.

[5] Lee, Kwang Y. and El-Sharkawi, Mohamed A. *Modern heuristic optimization techniques : theory and applications to power systems*. IEEE press series on power engineering ; 39. Wiley, Hoboken, New Jersey, 2015 - 2008.

[6] Poli, Riccardo and Kennedy, James and Blackwell, Tim. Particle swarm optimization: An overview. *Swarm intelligence*, 1:33–57, 2007.

[7] George Rossides, Benjamin Metcalfe, and Alan Hunter. Particle swarm optimization—an adaptation for the control of robotic swarms. *Robotics*, 10(2), 2021.

[8] Basic pso parameters. `https://web2.qatar.cmu.edu/~gdicaro/15382-Spring18/hw/hw3-files/pso-book-extract.pdf`. Accessed: 2023-03-16.

[9] Yuhui Shi and Russell C. Eberhart. Parameter selection in particle swarm optimization. In V. W. Porto, N. Saravanan, D. Waagen, and A. E. Eiben, editors, *Evolutionary Programming VII*, pages 591–600, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[10] H. Hamedi and E. Khamehchi. A nonlinear approach to gas lift allocation optimization with operational constraints using particle swarm optimization and a penalty function. *Petroleum science and technology*, 30(8):775–785, 2012.

[11] Qiang Guo, Guoqing Ruan, and Jian Wan. A sparse signal reconstruction method based on improved double chains quantum genetic algorithm. *Symmetry*, 9(9), 2017.

[12] Parsopoulos, Konstantinos E. and Vrahatis, Michael N. . Particle Swarm Optimization Method for Constrained Optimization Problems. https://www.researchgate.net/profile/Michael-Vrahatis/publication/2527227_Particle_Swarm_Optimization_Method_for_Constrained_Optimization_Problem/links/0fcfd50e4af5e434cb000000/Particle-Swarm-Optimization-Method-for-Constrained-Optimization-Problem.pdf?_sg%5B0%5D=started_experiment_milestone&origin=journalDetail. Accessed: 2023-03-16.

[13] Helder Pereira Borges, Omar Andres Carmona Cortes and Dario Vieira. *An Adaptive Metaheuristic for Unconstrained Multimodal Numerical Optimization.* Theoretical Computer Science and General Issues ; 10835. Springer,International Publishing, Cham, 1st ed. 2018. edition, 2018. In: Korošec, Peter, Malab, Nouredine and Talbi, El-Ghazali .Eds, Bioinspired Optimization Methods and Their Applications: 8th International Conference, BIOMA 2018, Paris, France, May 16-18, 2018, Proceedings.

[14] Lester James V. Miranda. pyswarms.single package. https://pyswarms.readthedocs.io/en/latest/api/pyswarms.single.html. Accessed: 2023-03-16.

A Python Program: PSO Implementation

A Python program defining the season simulation function and particle swarm optimisation implementation.

```
from decimal import Decimal
from time import perf_counter
import random
import math
import pandas as pd
import numpy as np
budgetsplit1 = pd.read_excel(r'C:/Users/New/OneDrive/Documents/
    S4BudgetAllocationcopy.xlsx')
# Driver list to be imported in order from highest to lowest skill
driverlist = pd.read_excel(r'C:/Users/New/OneDrive/Documents/S4Driverscopy.
    xlsx')
circuitlist = pd.read_excel(r'C:/Users/New/OneDrive/Documents/S4Circuits.xlsx
    ')


# Define season simulation function
def seasonsim(driverlist, budgetsplit, circuitlist):
    # function to return chasis investment of a team
  def chasisfinder(team):
    for i in range(len(budgetsplit.Team)):
      if team == "team{0}".format(budgetsplit.Team[i]):
        return budgetsplit.Chasis[i]

  # function to return engine investment of a team
  def enginefinder(team):
    for i in range(len(budgetsplit.Team)):
      if team == "team{0}".format(budgetsplit.Team[i]):
        return budgetsplit.Engine[i]

  # function to return reliability investment of a team
  def reliabilityfinder(team):
    for i in range(len(budgetsplit.Team)):
      if team == "team{0}".format(budgetsplit.Team[i]):
        return budgetsplit.Reliability[i]
  ## Driver Allocation

  # Define dictionary to contain marketing budget ^4 for each team
  team4marketing = {}
  # Calculate marketing budget ^4 for each team
  for x in range(len(budgetsplit.Marketing)):
      team4marketing["team{0}".format(x+1)] = Decimal(str(budgetsplit.
    Marketing[x]))**4
  # list of team marketing budgets ^ 4
  team4marketing_list = []
  for x in range(len(budgetsplit.Marketing)):
    team4marketing_list.append(team4marketing["team{0}".format(x+1)])

  # dictionary to contain driver probability
  driverprobability = {}
  # Assign each driver a probability in [0,1) from a unifrom distribution
  for x in range(len(driverlist.Skill)):
      driverprobability["{0}".format(driverlist.Driver[x])] =  random.uniform
    (0,1)

  # Define dictionary to contain team driver assignment results
```

```
teamdrivers = {}

# Assign each driver to a team based on driverprobability
for x in range(len(driverlist.Skill)):
  # Define relevant lists and dictionaries
  team4marketing_inplay = []
  teamprobability = {}
  teamprobability_list = []
  teamname_list = []
  # Check if team has 2 drivers allocated,
  # Append its marketing budget ^ 4 to list if not
  for i in range(len(budgetsplit.Team)):
    if "team{0}".format(i+1) in teamdrivers:
      if len(teamdrivers["team{0}".format(i+1)]) > 1:
        continue
    team4marketing_inplay.append(team4marketing["team{0}".format(i+1)])
  for i in range(len(budgetsplit.Team)):
    if "team{0}".format(i+1) in teamdrivers:
      if len(teamdrivers["team{0}".format(i+1)]) > 1:
        continue
    # Assign probabilites to dictionary and list, and available team names to list
    if sum(team4marketing_inplay) != 0:
      teamprobability["team{0}".format(i+1)] = (team4marketing_list[i])/sum(team4marketing_inplay)
    else:
      teamprobability["team{0}".format(i+1)] = 1/Decimal(len(team4marketing_inplay))
    teamprobability_list.append(teamprobability["team{0}".format(i+1)])
    teamname_list.append("team{0}".format(i+1))
  # Use new variable j to assign drivers based on probability rule defined
  for j in range(len(teamprobability_list)):
    if driverprobability[driverlist.Driver[x]] <= math.fsum(teamprobability_list[0:j+1]):
      if teamname_list[j] in teamdrivers:
        if len(teamdrivers[str(teamname_list[j])]) > 1:
          continue
        else:
          teamdrivers[teamname_list[j]].append(driverlist.Driver[x])
          break
      else:
        teamdrivers[teamname_list[j]] = [driverlist.Driver[x]]
        break

##Season Simulation

# function to return assigned team of a driver
def teamfinder(driver):
  for key, value in teamdrivers.items():
    for i in range(2):
      if driver == value[i]:
        return key

# Season Simulator
# define dictionary to contain rank of each driver for each circuit
circuitrank = {}
for j in range(len(circuitlist.Circuit)):
  # Define dict. to contain laptimes, circuittime and if a driver DNF
  laptimes = {}
  circuittime = {}
```

```
    # Loop through each driver to calculate lap time
  for x in range(len(driverlist.Driver)):
    # Assume setup is 0
    # Define variables used in tperf formula
    Driving = Decimal(driverlist.Skill[x])
    Chasis = Decimal(chasisfinder(str(teamfinder(driverlist.Driver[x]))))
    Engine = Decimal(enginefinder(str(teamfinder(driverlist.Driver[x]))))
    circuitdriver = Decimal(float(circuitlist.Driver[j]))
    circuitchasis = Decimal(float(circuitlist.Chasis[j]))
    circuitengine = Decimal(float(circuitlist.Engine[j]))
    # lap time model, based on tperf and tbase.
    tperf = Decimal(-0.15)*(3*(circuitdriver*Driving+circuitchasis*Chasis+
circuitengine*Engine))/(circuitdriver+circuitchasis+circuitengine)
    # Add laptime to dict.
    if "{0}".format(driverlist.Driver[x]) in laptimes:
      laptimes["{0}".format(driverlist.Driver[x])].append(tperf +
circuitlist.Base[j])
      continue
    else:
      laptimes["{0}".format(driverlist.Driver[x])] = [tperf + circuitlist.
Base[j]]
      continue
  # Calculate if a driver DNF, append this to DNFtracker
  for y in range(len(driverlist.Driver)):
    randomDNFprob = Decimal(random.uniform(0,1))
    reliabilityprob = Decimal(((1-math.erf(Decimal(reliabilityfinder(str(
teamfinder(driverlist.Driver[y])))))*Decimal(0.9)-Decimal(1.5)))**2)/4)
    if randomDNFprob < reliabilityprob:
      if "{0}".format(driverlist.Driver[y]) in circuitrank:
        circuitrank[driverlist.Driver[y]].append("DNF")
        continue
      else:
        circuitrank[driverlist.Driver[y]] = ["DNF"]
        continue
    # If not DNF, calculate circuit time
    else:
      circuittime["{0}".format(driverlist.Driver[y])] = math.fsum(laptimes[
"{0}".format(driverlist.Driver[y])])
  # rank finishing drivers by circuit time
  circuitranklist = sorted(circuittime.items(), key=lambda item: item[1])
  for m in range(len(circuitranklist)):
    if "{0}".format(circuitranklist[m][0]) in circuitrank:
      circuitrank["{0}".format(circuitranklist[m][0])].append(int(m+1))
      continue
    else:
      circuitrank["{0}".format(circuitranklist[m][0])] = [m+1]
      continue

# Driver point allocator, based on 14 teams - alter when number of teams
  known:
# 10pts, 8pts, 6pts, 5pts, 4pts, 3pts, 2pts, 1pt.

driverpoints = {}
for x in range(len(circuitrank)):
  driverpointslist = 0
  for y in range(len(circuitlist.Circuit)):
    if circuitrank[driverlist.Driver[x]][y] == 1:
      driverpointslist += 25
      continue
```

```
      elif circuitrank[driverlist.Driver[x]][y] == 2:
        driverpointslist += 20
        continue
      elif circuitrank[driverlist.Driver[x]][y] == 3:
        driverpointslist += 16
        continue
      elif circuitrank[driverlist.Driver[x]][y] == 4:
        driverpointslist += 13
        continue
      elif circuitrank[driverlist.Driver[x]][y] == 5:
        driverpointslist += 11
        continue
      elif circuitrank[driverlist.Driver[x]][y] == 6:
        driverpointslist += 10
        continue
      elif circuitrank[driverlist.Driver[x]][y] == 7:
        driverpointslist += 9
        continue
      elif circuitrank[driverlist.Driver[x]][y] == 8:
        driverpointslist += 8
        continue
      elif circuitrank[driverlist.Driver[x]][y] == 9:
        driverpointslist += 7
        continue
      elif circuitrank[driverlist.Driver[x]][y] == 10:
        driverpointslist += 6
        continue
      elif circuitrank[driverlist.Driver[x]][y] == 11:
        driverpointslist += 5
        continue
      elif circuitrank[driverlist.Driver[x]][y] == 12:
        driverpointslist += 4
        continue
      elif circuitrank[driverlist.Driver[x]][y] == 13:
        driverpointslist += 3
        continue
      elif circuitrank[driverlist.Driver[x]][y] == 14:
        driverpointslist += 2
        continue
      elif circuitrank[driverlist.Driver[x]][y] == 15:
        driverpointslist += 1
        continue

    driverpoints[driverlist.Driver[x]] = [driverpointslist]

  sorteddriverpoints = sorted(driverpoints.items(), key=lambda item: item[1],
     reverse = True)

  # Team point allocator

  teampoints = {}
  for i in range(len(sorteddriverpoints)):
    if teamfinder(sorteddriverpoints[i][0]) in teampoints:
      teampoints[teamfinder(sorteddriverpoints[i][0])].append(
    sorteddriverpoints[i][1])
    else:
      teampoints[teamfinder(sorteddriverpoints[i][0])] = [sorteddriverpoints[
    i][1]]

  teampointssum = {}
```

```
  for i in teampoints:
    teampointssum[i] = [teampoints[str(i)][0][0] + teampoints[str(i)][1][0]]

  sortedteamsum = sorted(teampointssum.items(), key =lambda item: item[1],
    reverse = True)
  sortedteammetric = {}
  count = 1
  for i in sortedteamsum:
    if count == 1:
      sortedteammetric[i[0]] = [32]
      count += 1
      continue
    elif count == 2:
      sortedteammetric[i[0]] = [16]
      count += 1
      continue
    elif count == 3:
      sortedteammetric[i[0]] = [4]
      count += 1
      continue
    elif count == 4:
      sortedteammetric[i[0]] = [2]
      count += 1
      continue
    elif count == 5:
      sortedteammetric[i[0]] = [1]
      count += 1
      continue
    else:
      sortedteammetric[i[0]] = [0]
      continue

  return(sorted(sortedteammetric.items(), key =lambda item: item[1], reverse
    = True))

def expectedpointfunction(numberofsims, budgetsplit_current):
  totalpoints = {}
  expectedpoints = {}
  simscount = numberofsims
  for i in range(simscount):
    currentseason = seasonsim(driverlist, budgetsplit_current, circuitlist)
    for i in range(len(currentseason)):
      if currentseason[i][0] in totalpoints:
        totalpoints[currentseason[i][0]].append(currentseason[i][1])
      else:
        totalpoints[currentseason[i][0]] = [currentseason[i][1]]
  for i in range(len(totalpoints)):
    expectedpoints["team{0}".format(i+1)] = sum(sum(totalpoints["team{0}".
    format(i+1)],[]))/simscount
  epsorted = sorted(expectedpoints.items(), key =lambda item: item[1],
    reverse = True)
  return(epsorted)

def expectedpointfunction1(numberofsims, budgetsplit_current):
  totalpoints = {}
  expectedpoints = {}
  simscount = numberofsims
  for i in range(simscount):
    currentseason = seasonsim(driverlist, budgetsplit_current, circuitlist)
    for i in range(len(currentseason)):
```

```
      if currentseason[i][0] in totalpoints:
        totalpoints[currentseason[i][0]].append(currentseason[i][1])
      else:
        totalpoints[currentseason[i][0]] = [currentseason[i][1]]
  for i in range(len(totalpoints)):
    expectedpoints["team{0}".format(i+1)] = sum(sum(totalpoints["team{0}".
    format(i+1)],[]))/simscount
  return(expectedpoints["team1"])

budgetsplitcopy = budgetsplit1
# particle in form [marketing, reliability, chasis, engine]
# team should be number in range [1, n]
# Changing values of team 1 to be updated values in budgetsplitcopy
def changebudgetsplit(particle):
  # Assign particle values to the team
  reliabilityparticle = max(0,8-sum(particle))
  particle1 = [particle[0], reliabilityparticle, particle[1], particle[2]]
  budgetsplitcopy.loc[budgetsplitcopy["Team"] == 1, ["Marketing", "
    Reliability", "Chasis", "Engine"]] = particle1
  return budgetsplitcopy

## PSO

import pyswarms as ps

# PSO Parameter Adjustment
options = {
    "c1": 0.3,
    "c2": 0.18,
    "w": 0.42,
}
# val 1 = marketing, val 2 = reliability, val 3 = chasis, val 4 = engine
# particle fitness
def fitness(values):
  if sum(values) > 8:
    points = -((sum(values)-8)**2)
  else:
    points = expectedpointfunction1(150, changebudgetsplit(values))
  return -points # negative as we seek to minimise

# swarm fitness
def f(x):
    n_particles = x.shape[0]
    j = [fitness(x[i]) for i in range(n_particles)]
    return np.array(j)

# Define the bounds of the variables
bounds = (0*np.ones(3),8*np.ones(3))

ps.single.GlobalBestPSO.maximize = True
# Create an instance of the pyswarms.single.GlobalBestPSO class
optimizer = ps.single.GlobalBestPSO(n_particles=15, dimensions=3, options=
    options, bounds=bounds)

# Run the optimization and get the best solution found
t1 = perf_counter()
best_score, best_values = optimizer.optimize(f, iters=50)

t2 = perf_counter()
# Print the best solution found
```

```
print("Best values:", best_values)

print("Best score:", best_score)
print("Elapsed time during the whole program in seconds:",
                                        t2-t1)
import matplotlib.pyplot as plt
plt.rcParams.update({
    'font.family': "serif",
    "font.size": 10
})
from pyswarms.utils.plotters import (plot_cost_history, plot_contour,
    plot_surface)
fig = plt.figure()
plt.plot(optimizer.cost_history)
plt.xlabel('Number of Iterations')
plt.locator_params(axis="x", integer=True)
plt.ylabel('Objective Metric')
plt.show()

print(seasonsim(driverlist, budgetsplit1, circuitlist))
# Assume optimizer is an instance of PSO class

## 3d plot of all iterations

# run the PSO algorithm and store the position history
pos_history = optimizer.pos_history

# convert the position history list into a NumPy array
pos_history = np.array(pos_history)

# plot the position history in 3D
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
for i in range(pos_history.shape[1]):
    x = pos_history[:, i, 0]
    y = pos_history[:, i, 1]
    z = pos_history[:, i, 2]
    ax.plot(x, y, z, '-o', label=f"Particle {i}")
ax.legend()
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
ax.set_title('Position history of the swarm')
plt.show()

## 3d plot at given iteration
# run the PSO algorithm and store the position history
pos_history = optimizer.pos_history

# convert the position history list into a NumPy array
pos_history = np.array(pos_history)

# specify the iteration number
iterlist = [0]

# plot the particle positions at the specified iteration
for j in iterlist:
  fig = plt.figure()
  ax = fig.add_subplot(111, projection='3d')
  for i in range(pos_history.shape[1]):
```

```
        x = pos_history[j, i, 0]
        y = pos_history[j, i, 1]
        z = pos_history[j, i, 2]
        ax.scatter(x, y, z, label=f"Particle {i+1}")
    ax.legend()
    ax.set_xlabel('Marketing')
    ax.set_ylabel('Chassis')
    ax.set_zlabel('Engine')
    ax.set_title(f'Particle positions at iteration {j}')
    ax.set_box_aspect([1,1,1])
    ax.legend(bbox_to_anchor=(1.05, 1), loc='upper left')

plt.show()

# same fig
fig, axes = plt.subplots(1, 3, figsize=(15, 5), subplot_kw={'projection': '3d'})
lines = []

for i, j in enumerate(iterlist):
    for k in range(pos_history.shape[1]):
        x = pos_history[j, k, 0]
        y = pos_history[j, k, 1]
        z = pos_history[j, k, 2]
        line = axes[i].scatter(x, y, z, label=f"Particle {k+1}")
        if i == 0:  # add each particle to the lines list only once
            lines.append(line)

    axes[i].set_xlabel('Marketing')
    axes[i].set_ylabel('Chassis')
    axes[i].set_zlabel('Engine')
    axes[i].set_title(f'Particle positions at iteration {j}')
    axes[i].set_box_aspect([1, 1, 1])

fig.legend(handles=lines, labels=[l.get_label() for l in lines],
    bbox_to_anchor=(1.05, 0.5), loc='center right')

plt.tight_layout()
plt.show()

# animation

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
import matplotlib.animation as animation
from matplotlib.animation import FuncAnimation
import seaborn as sns

pos_history = optimizer.pos_history
pos_history = np.array(pos_history)

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
ax.set_box_aspect([1, 1, 1])

def animate(i):
```

```
    ax.clear()
    for j in range(pos_history.shape[1]):
        x = pos_history[:i, j, 0]
        y = pos_history[:i, j, 1]
        z = pos_history[:i, j, 2]
        ax.plot(x, y, z, '-o', label=f"Particle {j}")
    ax.set_title(f'Particle positions at iteration {i}')
    ax.legend(loc='center left', bbox_to_anchor=(1, 0.5))

ani = FuncAnimation(fig, animate, frames=pos_history.shape[0], interval=500)
plt.show()

num_particles = len(pos_history[0])
num_iterations = len(pos_history)

# initialize the distances array
distances = np.zeros((num_particles, num_iterations-1))

# calculate the absolute distance for each particle between each iteration
for i in range(num_iterations-1):
    for j in range(num_particles):
        distances[j, i] = np.linalg.norm(pos_history[i+1][j] - pos_history[i
    ][j])

# create a Seaborn line plot showing how the absolute distance changes for
    each particle
sns.set(style='whitegrid')
plt.figure(figsize=(10,6))
for i in range(num_particles):
    plt.plot(distances[i], label=f'Particle {i+1}')
plt.xlabel('Iteration', size = 12)
plt.ylabel('Absolute Distance', size = 12)
plt.legend()
plt.show()
```

Academic integrity statement

You must sign this (typing in your details is acceptable) and include it with each piece of work you submit.

I am aware that the University defines plagiarism as presenting someone else's work, in whole or in part, as your own. Work means any intellectual output, and typically includes text, data, images, sound or performance.

I promise that in the attached submission I have not presented anyone else's work, in whole or in part, as my own and I have not colluded with others in the preparation of this work. Where I have taken advantage of the work of others, I have given full acknowledgement. I have not resubmitted my own work or part thereof without specific written permission to do so from the University staff concerned when any of this work has been or is being submitted for marks or credits even if in a different module or for a different qualification or completed prior to entry to the University. I have read and understood the University's published rules on plagiarism and also any more detailed rules specified at School or module level. I know that if I commit plagiarism I can be expelled from the University and that it is my responsibility to be aware of the University's regulations on plagiarism and their importance.

I re-confirm my consent to the University copying and distributing any or all of my work in any form and using third parties (who may be based outside the EU/EEA) to monitor breaches of regulations, to verify whether my work contains plagiarised material, and for quality assurance purposes.

I confirm that I have declared all mitigating circumstances that may be relevant to the assessment of this piece of work and that I wish to have taken into account. I am aware of the University's policy on mitigation and the School's procedures for the submission of statements and evidence of mitigation. I am aware of the penalties imposed for the late submission of coursework.

Name [redacted]

Student ID [redacted]