Fantasy Football Optimization

Marco De Rito - SM3800016
University of Trieste
Trieste, Italy
marco.derito@studenti.units.it

Abstract

This report represents my final project for an advanced optimization course, where I develop a comprehensive methodology for fantasy football team selection, formulated as an *inverse optimization problem*. I apply three well-known meta-heuristics: Particle Swarm Optimization (PSO), Differential Evolution (DE) and Evolution Strategies (ES) in a multi-manager setting, ensuring budget and positional constraints are respected.

The study integrates theoretical foundations (including PSO, DE, ES, and inverse optimization principles), practical code implementations in Python (data loading, scoring, fitness function, forced assignments, multi-round auctions), and empirical evaluations. Through comparisons of budget usage, forced picks, and final team scores, I show that each algorithm delivers competitive results, albeit with differences in strategy diversity and resource utilization. Finally, I reflect on future enhancements.

CCS Concepts

• **Computing methodologies → Optimization algorithms**.

Keywords

Fantasy Football, Inverse Optimization, Particle Swarm Optimization, Differential Evolution, Evolution Strategies, Machine Learning, Auction Mechanisms, Student Project

1 Introduction

Fantasy football is a popular game where participants (managers) build squads of real-life players under specific rules (budget limits, positional minimums, etc.). In this project, I aim to formulate this selection task as an *data drive* scenario: rather than defining player values arbitrarily, I adapt a scoring function based on real historical stats (goals, assists, ratings, etc.).

I employ three established methods from the field of *evolutionary computation* and *swarm intelligence*:

- **Particle Swarm Optimization (PSO)**
- **Differential Evolution (DE)**
- **Evolution Strategies (ES)**

Each manager picks one algorithm to generate bids for available players in a multi-manager auction. After repeated rounds, I resolve conflicts and enforce forced assignments if rosters remain incomplete. This report summarizes my approach, from conceptual underpinnings to Python implementation details, culminating in comparative charts of each algorithm's performance.

Exam Global and Multi-Objective Optimization, 2025, Trieste, Italy
2025. ACM ISBN none
https://doi.org/none

1.1 Report Structure

- **Section 2** introduces the concept fantasy football and briefly reviews the fundamentals of PSO, DE, and ES.
- **Section 3** describes the project's architecture (data loading, utility classes, the multi-manager auction routine) and provides code snippets for the scoring function, common fitness logic, and forced assignments.
- **Section 4** presents empirical analyses: budget usage, forced picks, and average team scores.
- **Section 5** investigates inverse hyper-parameter tuning: instead of optimizing rosters, search for algorithmic settings (including PSO, DE, and ES) that replicate a desired KPI profile. The section details the loss formulation, the outer PSO search, convergence behavior, and the best discovered configurations.
- **Section 6** concludes with reflections and potential future extensions, such as hybrid evolutionary approaches or reinforcement learning.

2 Theoretical Background

2.1 Data Driven in Fantasy Football

An *data driven* framework attempts to refine a cost or reward function so that historically observed solutions appear near-optimal. In fantasy football, the logic is as follows: if certain players consistently yield high performance, I want my scoring function to reflect that. While I do not explicitly solve a full parameter-fitting routine, I adopt the spirit of data driven optimization by letting real data guide the final scoring weights (`score_player`).

2.2 Particle Swarm Optimization (PSO)

PSO is inspired by natural swarms. I track positions $\mathbf{x}_i$ (bids) and velocities $\mathbf{v}_i$ for each particle i, updated as:

$$\begin{aligned}\mathbf{v}_i(t+1) &= \omega\mathbf{v}_i(t) + c_1 r_1(\mathbf{p}_i - \mathbf{x}_i(t)) + c_2 r_2(\mathbf{g} - \mathbf{x}_i(t)),\\ \mathbf{x}_i(t+1) &= \mathbf{x}_i(t) + \mathbf{v}_i(t+1).\end{aligned}$$

Where:

- $\mathbf{x}_i(t)$ – position of particle i at iteration t
- $\mathbf{v}_i(t)$ – velocity of particle i at iteration t
- $\mathbf{p}_i$ – personal best position found by particle i
- $\mathbf{g}$ – global best position in the swarm
- r_1, r_2 – random numbers $\in [0, 1]$
- c_1, c_2 – cognitive and social acceleration coefficients
- ω – inertia weight (controls momentum)

$\mathbf{p}_i$ is the best solution found by particle i, and $\mathbf{g}$ is the global best. PSO often rapidly explores a vast solution space but can exhibit variability if not properly tuned.

2.3 Differential Evolution (DE)

DE uses vector differences to generate mutant solutions. For each agent (bid vector). The mutation and crossover steps in DE are defined as:

$$\mathbf{y} = \mathbf{x}_a + F \cdot (\mathbf{x}_b - \mathbf{x}_c)$$

$$\mathbf{z}_j = \begin{cases} \mathbf{y}_j & \text{if } r_j < CR \text{ or } j = j_{rand} \\ \mathbf{x}_{i,j} & \text{otherwise} \end{cases}$$

Where:

- $\mathbf{x}_a, \mathbf{x}_b, \mathbf{x}_c$ – distinct randomly chosen vectors ($\neq \mathbf{x}_i$)
- F – mutation factor, here drawn uniformly from the range $[0.5,\ 1.0]$
- CR - crossover rate, here: 0.7
- $\mathbf{y}$ – mutant vector
- $\mathbf{z}$ – trial vector
- j_{rand} – random index to ensure at least one component from $\mathbf{y}$

Then it crosses over $\mathbf{y}$ with the target $\mathbf{x}_i$. If the resulting trial $\mathbf{z}$ has better fitness, it replaces $\mathbf{x}_i$. DE is known for robust fine-tuning in continuous spaces.

2.4 Evolution Strategies (ES)

ES employ Gaussian mutations plus a survivor selection scheme $(\mu + \lambda)$ or $(\mu + \lambda)$. By mutating each parent to create offspring, then retaining the top solutions, ES preserve high diversity. In the fantasy scenario, random mutation of bids can sometimes yield unexpected but valuable team compositions, especially if guided by a well-structured penalty for budget or positional violations.

In Evolution Strategies, offspring are generated via Gaussian mutation and then selected according to the $(\mu + \lambda)$ strategy. The mutation is defined as:

$$\mathbf{x}_{\text{child}} = \mathbf{x}_{\text{parent}} + \sigma \cdot \mathcal{N}(0, I)$$

Where:

- $\mu = 40$ – number of parents
- $\lambda = 80$ – number of offspring
- σ – mutation strength (implicit via DEAP's toolbox)
- $\mathcal{N}(0, I)$ – standard Gaussian noise
- Selection is $(\mu + \lambda)$: best μ individuals from parents and offspring survive

3 Implementation and Architecture

3.1 Overall Project Structure

My Python project is split into modules:

- `data_loader.py` - reading and cleaning the CSV with player stats
- `utils.py` - classes `Player`, `Manager`, scoring function
- `optimization.py` - all the meta-heuristics (PSO, DE, ES) plus the multi-manager auction logic
- `main.py` - orchestrates user inputs, runs the auction, generates final summary
- `report_generator.py` - builds a PDF with charts (budget usage, forced picks, etc.)

3.2 Scoring Function and Common Fitness Logic

One of the most critical elements is the player scoring function, which turns historical stats (goals, assists, etc.) into a single numeric value. Compute the player's score based on various performance metrics.

Scoring breakdown:

- Goals Scored: +0.5 per goal
- Assists: +0.2 per assist
- Yellow Cards: -0.05 per card
- Red Cards: -0.1 per card
- `Rating`: +0.2 per rating
- `Penalties Scored`: +0.2 per penalty
- `Goals Conceded`: −0.5 per goal
- `Penalties Saved`: +0.5 per save
- `Matches Played`: +0.5 per match

Parameters:

- player: The Player object for which to calculate the score.

Returns:

- A numerical value representing the player's overall score.

```python
def score_player(player):

# Retrieve player statistics using
    getattr to provide default values if
     not set
goals = getattr(player, 'goals_scored',
    0)
assists = getattr(player, 'assists', 0)
yellow_cards = getattr(player, '
    yellow_cards', 0)
red_cards = getattr(player, 'red_cards',
    0)
rating = getattr(player, 'rating', 6.0)
penalties = getattr(player, '
    penalties_scored', 0)
goals_conceded = getattr(player, '
    goals_conceded', 0)
penalties_saved = getattr(player, '
    penalties_saved', 0)
matches_played = getattr(player, '
    matches_played', 0)

# Calculate the score based on the
    weighted metrics
score = ((0.5 * goals) + (0.2 * assists)
    - (0.05 * yellow_cards) - (0.1 *
    red_cards) + (0.2 * rating) + (0.2 *
     penalties)) - (0.5 * goals_conceded
    ) + (0.5 * penalties_saved) + (0.5 *
     matches_played)
return score
```

Listing 1: Example scoring function in utils.py

Each algorithm then needs a **fitness function** to evaluate how good a bid vector is. If an algorithm is a minimizer (like `pyswarm.pso` or `differential_evolution`), I return negative total score (plus large penalties for invalid solutions). The snippet below shows a common approach (`common_fitness_logic`):

```python
def common_fitness_logic(manager, bids:
    List[float], roles: List[str],
    scores: List[float], min_thr: int)
    -> float:

global evaluation_count, surrogate_model
evaluation_count += 1

```

```
int_bids = [int(round(b)) for b in bids]
for i, bid_value in enumerate(int_bids):
if 0 < bid_value < min_thr:
int_bids[i] = min_thr

budget = manager.budget
total_spent = sum(int_bids)
if total_spent > budget:
return HIGH_PENALTY

for bid_value in int_bids:
if bid_value > budget *
     SINGLE_PLAYER_CAP_RATIO:
return HIGH_PENALTY

leftover_budget = budget - total_spent
max_total = int(manager.max_total)
players_needed_local = max_total - len(
     manager.team)
if leftover_budget < players_needed_local
     :
return HIGH_PENALTY

leftover_penalty = ((leftover_budget -
     players_needed_local) **
     BUDGET_LEFTOVER_EXP) *
     LEFTOVER_MULTIPLIER
chosen_count = sum(1 for v in int_bids if
      v >= min_thr)
penalty = abs(chosen_count -
     players_needed_local) *
     PLAYER_COUNT_PENALTY

role_count = {}
for i, bid_value in enumerate(int_bids):
if bid_value >= min_thr:
r = roles[i]
role_count[r] = role_count.get(r, 0) + 1

for r, (min_r, max_r) in manager.
     role_constraints.items():
current_have = sum(1 for p in manager.
     team if p.role == r)
add_count = role_count.get(r, 0)
if current_have + add_count < min_r or
     current_have + add_count > max_r:
return HIGH_PENALTY

penalty += leftover_penalty

total_score = 0.0
for i, bid_value in enumerate(int_bids):
if bid_value >= min_thr:
w = role_weight(manager, roles[i])
total_score += w * scores[i]

computed_fitness = penalty - total_score

if USE_SURROGATE and surrogate_model is
     not None:
surrogate_model.update(bids,
     computed_fitness)
if evaluation_count % 20 == 0:
surrogate_model.train()
if evaluation_count >=
     SURROGATE_THRESHOLD:
surrogate_val = surrogate_model.evaluate(
     bids)
return 0.5 * computed_fitness + 0.5 *
     surrogate_val

return computed_fitness
```

Listing 2: Common fitness logic for PSO/DE/ES

3.3 Manager Strategies: PSO, DE, ES

In my code, each `Manager` calls a function `decide_bids` that internally picks the appropriate meta-heuristic.

The performance of evolutionary algorithms is highly dependent on the appropriate selection of their parameters. Below is a summary of the key parameters for PSO, DE, and ES, along with their descriptions, typical values, and justifications.

Table 1: Key Parameters for PSO, DE, and ES

Alg.	Parameter	Description	Typical Values
PSO	ω (Inertia Weight)	Balances global and local exploration.	0.7–0.9; higher values promote global search.
PSO	c_1 (Cognitive Coefficient)	Influence of personal best position.	1.5–2.0; standard choice to balance exploration and exploitation.
PSO	c_2 (Social Coefficient)	Influence of global best position.	1.5–2.0; ensures convergence toward the global best.
DE	F (Differential Weight)	Scales the differential variation.	0.5–0.8; controls amplification of difference vectors.
DE	CR (Crossover Rate)	Probability of parameter crossover.	0.7–0.9; higher values increase diversity.
ES	μ (Parent Population Size)	Number of parent solutions.	Typically $\lambda/7$; balances selection pressure.
ES	λ (Offspring Population Size)	Number of offspring generated.	Typically 7μ; ensures sufficient exploration.

The chosen values for these parameters are based on standard practices in evolutionary computation. For instance, in PSO, an inertia weight (ω) around 0.7–0.9 is commonly used to balance exploration and exploitation. Similarly, in DE, a crossover rate (CR) of 0.7–0.9 is typical to maintain diversity among solutions. In ES, setting λ approximately seven times μ is recommended to maintain an appropriate selection pressure.

```
from pyswarm import pso

best_bids, _ = pso(
fitness_func, lb, ub,
swarmsize=40,
maxiter=80,
omega=0.7,
phip=1.8,
phig=1.8
)
```

Listing 3: PSO parameter setup with pyswarm

For PSO, I set the inertia weight $\omega = 0.7$, and both cognitive and social coefficients $c_1 = c_2 = 1.8$. These values were chosen to balance exploration and exploitation, ensuring stable convergence while allowing the swarm to effectively explore the solution space.

```
from scipy.optimize import
     differential_evolution

result = differential_evolution(
fitness_wrapper,
bounds=[(0.0, float(max_bid_per_player))]
      * n,
strategy='best1bin',
mutation=(0.5, 1.0),     # F  ~ U(0.5,
     1.0)
recombination=0.7,        # CR
maxiter=50,
popsize=15
)
```

Listing 4: DE parameter setup with SciPy

These values were chosen to maintain balance between broad mutation and reliable recombination, using SciPy's standard strategy `best1bin`.

```
from deap import algorithms

algorithms.eaMuPlusLambda(
population_, toolbox,
mu=40,
lambda_=80,
cxpb=0.5,
mutpb=0.3,
ngen=40,
verbose=False
)
```

Listing 5: ES setup with DEAP using (μ+λ)

I adopted the $(\mu + \lambda)$ strategy with $\mu = 40$ and $\lambda = 80$ to ensure diversity among offspring. The mutation probability was set to 0.3, and crossover to 0.5, as commonly suggested for continuous optimization.

3.4 Multi-manager Auction and Forced Assignments

Once each manager decides their bids in a given round, I collect them and assign players to the highest bidder or use a small function `resolve_competition` to handle tie-breaks. Below is a simplified version of my multi-round auction orchestrator:

```

def multi_manager_auction(players,
     managers, max_turns=30):
not_assigned = {p.pid: p for p in players
     }
turn = 0

while turn < max_turns and not_assigned:
turn += 1
all_bids = []

# Each manager proposes bids
for mgr in managers:
unass_list = list(not_assigned.values())
bids = mgr.decide_bids(unass_list)
for (pid, amt) in bids:
if pid in not_assigned and mgr.can_buy(
     not_assigned[pid], amt):
all_bids.append((mgr, pid, amt))

if not all_bids:
break

# Group bids by player
bids_by_player = {}
for (mgr, pid, amt) in all_bids:
bids_by_player.setdefault(pid, []).append
     ((mgr, amt))

# Resolve conflicts and assign
for pid, manager_offers in bids_by_player
     .items():
if pid not in not_assigned:
continue

if len(manager_offers) == 1:
best_manager, best_amt = manager_offers
     [0]
else:
best_manager, best_amt =
     resolve_competition(manager_offers)

# Assign if feasible
if best_manager.can_buy(not_assigned[pid
     ], best_amt):
player_obj = not_assigned[pid]
player_obj.assigned_to = best_manager.
     name
player_obj.final_price = best_amt
best_manager.update_roster(player_obj,
     best_amt)
del not_assigned[pid]

# Post-process forced assignments
forced_assignments(managers, list(
     not_assigned.values()))
return managers, list(not_assigned.values
     ())
```

Listing 6: Multi-manager auction framework

Finally, the `forced_assignments` step ensures each manager meets the *minimum role constraints* by forcibly adding leftover players (often at base cost 1). This is a fallback mechanism for any manager who runs out of budget or gets outbid in key positions:

```
def forced_assignments(managers,
     leftover_players):
for mgr in managers:
for role, (min_r, _) in mgr.
     role_constraints.items():
# Count how many players of this role the
      manager already has
count_current = sum(p.role == role for p
     in mgr.team)

# Keep assigning until the minimum
     requirement is met
while count_current < min_r:
# Pick the best remaining player of the
     needed role
try:
candidate = next(
p for p in sorted(
leftover_players,
key=score_player,
reverse=True
)
if p.role == role
)
except StopIteration:
# No more players of this role are
     available
break

# Abort if the manager has no budget left
if not mgr.can_buy(candidate, 1):
break

# Apply the forced assignment at the
     minimum price (1 credit)
candidate.assigned_to = mgr.name
candidate.final_price = 1
mgr.update_roster(candidate, 1)

# Remove the player from the global pool
leftover_players.remove(candidate)

# Update the current count for this role
count_current += 1
```

Listing 7: Simple forced assignment to fill leftover roles

4 Example: Initial Inputs and Scoring Calculation

In this section, I present an overview of the initial data and results used in my optimization process. I include several graphical and tabular analyses to illustrate the diversity of input data and the effectiveness of my algorithm in handling budget constraints and role requirements. The following figures and tables were generated during simulation runs, and their analysis supports the conclusion that my approach is both effective and robust.

4.1 Graphical Analyses

4.1.1 Manager Distribution. Figure 1 shows the distribution of managers by strategy (PSO, DE, and ES). This graph confirms that my experiment has a balanced input—each strategy is well-represented.

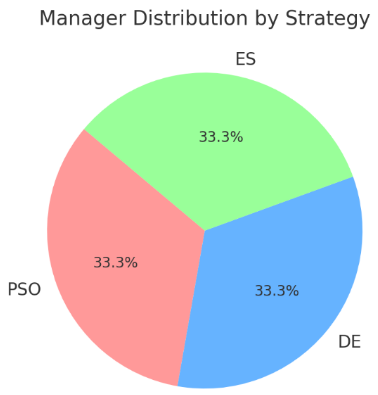

Figure 1: Distribution of managers by strategy.

4.1.2 Player Score Distribution. Figure 2 displays the distribution of player scores based on historical performance metrics. Notice that most players score in the lower-to-mid range, with only a few high-scoring outliers. This distribution forces managers to carefully allocate their budget. The effectiveness of my algorithm is demonstrated by its ability to select the optimal balance between expensive top performers and numerous affordable players.

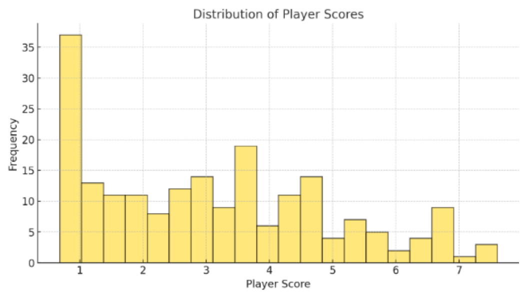

Figure 2: Distribution of player scores based on historical performance.

4.1.3 Budget Usage. Figure 3 illustrates the budget utilization across managers. The graph shows that most managers spend nearly all of their available budget, which is an indication that the algorithm efficiently utilizes resource. Effective budget usage is critical in ensuring that no valuable credits remain unspent, thus maximizing the potential overall team score.

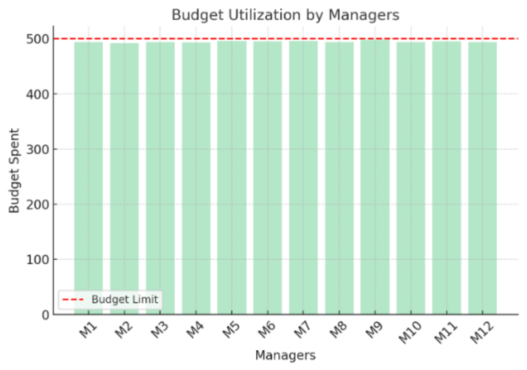

Figure 3: Budget Usage by Managers.

4.1.4 Forced Assignments. Figure 4 depicts the number of forced assignments per manager. Forced assignments occur when a manager fails to naturally satisfy the minimum role requirements, prompting the algorithm to assign additional players at a base cost. A low number of forced assignments indicates that the optimization process is generally successful at forming complete teams without needing extra interventions, thus proving the robustness of the method.

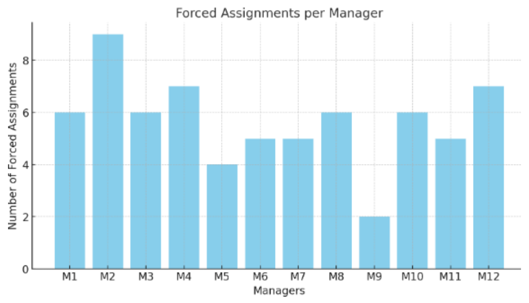

Figure 4: Number of Forced Assignments per Manager.

4.2 Tabular Analyses

4.2.1 Manager Recap Table. Table 2 provides a detailed recap of each manager's performance. It includes the chosen strategy, the number of forced assignments, the total budget spent, and the final objective score (team score). A low number of forced assignments along with near 500 spent budget demonstrates that the algorithm effectively satisfies all constraints.

Table 2: Recap Table: Manager Stats

Manager	Strat.	Forced	Spent	Tot. Score
Manager_1	PSO	6	494.0	52.91
Manager_2	DE	9	492.0	51.48
Manager_3	PSO	6	494.0	44.46
Manager_4	DE	7	493.0	53.64
Manager_5	ES	4	496.0	49.18
Manager_6	ES	5	495.0	46.02
Manager_7	PSO	5	496.0	47.00
Manager_8	DE	6	494.0	55.32
Manager_9	ES	2	498.0	44.23
Manager_10	DE	6	494.0	46.59
Manager_11	ES	5	495.0	39.99
Manager_12	PSO	7	494.0	53.23

4.2.2 Performance by Strategy Table. Table 3 aggregates the performance of each strategy, reporting the number of managers using each method along with the average total score. This table confirms that although DE tends to achieve a slightly higher total score, the differences among the strategies are minimal—further evidence of the algorithm's robustness.

Table 3: Performance by Strategy

Strat.	Man.	Avg Tot. Score	Avg Forced	Avg Spent
PSO	4	49.90	6	494.50
DE	4	51.76	7	493.25
ES	4	44.85	4	496.00

While DE yielded the highest average total score, it also required the most forced assignments, indicating a tendency to over-concentrate on top players and budget usage, sometimes at the cost of team completeness. ES, conversely, produced lower scores but needed fewer forced picks, showing a more balanced yet less aggressive optimization behaviour.

4.2.3 Player Score Summary Table. Table 4 summarizes the overall range of player scores calculated by my scoring function. The best, worst, and average scores reflect the challenging nature of the selection process, emphasizing that only a few elite players achieve high scores. This forces managers to balance high-cost, high-score players with more affordable options, demonstrating the effectiveness of my multi-objective optimization.

Table 4: Player Score Summary

	Best	Worst	Average
Player Scores	12.90	0.68	2.84

4.3 Overall Discussion

The integration of these analyses demonstrates that my algorithm is effective and robust:

- The balanced manager distribution (Figure 1) ensures diverse search behaviour.
- The player score distribution (Figure 2) highlights the need for careful budget allocation.
- Budget usage (Figure 3) and forced assignments (Figure 4) indicate that managers nearly fully utilize their budgets while rarely needing forced interventions.
- The high average team scores (Table 3) confirm that my optimization process consistently produces competitive teams.
- The player score summary (Table 4) emphasizes that only a few players reach elite performance levels, reinforcing the need for strategic selection.

Collectively, these results support the conclusion that my multi-manager auction framework effectively balances budget, role constraints, and team performance, validating the effectiveness of my approach in fantasy football.

4.4 Hyper-parameter Tuning

To assess parameter sensitivity we executed an exhaustive grid–search over *PSO*, *Differential Evolution* (DE) and *Evolution Strategies* (ES). Parameter values were organised in simple Python lists and then combined with `itertools.product`, i.e. by Cartesian product; the size of each grid is therefore the product of the set cardinalities:

- PSO = 2 inertia weights × 2 swarm sizes = **4** runs
- DE = 3 population sizes × 2 mutation ranges × 2 crossover rates = **12** runs
- ES = 4 $(\mu + \lambda)$ pairs × 2 generation counts = **8** runs

In total **24 independent auctions** were launched, each with the same 25-player pool, identical random seed and a dummy rival manager to guarantee bidding competition. The chosen ranges follow the *canonical defaults* most frequently reported in the literature: Clerc–Kennedy's coefficients for PSO (c_1=c_2=1.49445, high/low inertia), the "classic" F and CR values suggested by Storn and Price for Differential Evolution, and the empirical rule $\lambda \approx 2\mu$ that stems from early Evolution-Strategy work by Beyer and Schwefel. The grid is therefore wide enough to expose meaningful variation yet small enough to remain laptop-friendly (24 runs in total). Table 5 lists the exact ranges.

Table 5: Grid used for hyper-parameter search

PSO	$w \in \{0.9, 0.5\}$, c_1=c_2=1.49445, swarm {30, 60}
DE	pop {10, 15, 20}, $F \in \{(0.5, 1.0), (0.7, 1.2)\}$, $CR \in \{0.7, 0.9\}$
ES	$(\mu + \lambda) \in \{(15, 30), (15, 40), (20, 30), (20, 40)\}$, $ngen \in \{50, 80\}$

Best configuration per algorithm. The optimal settings extracted from `hyperparam_results.csv` are[1]:

Algorithm	Best hyper-parameters	Sum og Score
ES	$\mu = 20,\ \lambda = 40,\ ngen = 50$	**104.7**
DE	pop= 15, $F = (0.5, 1.0)$, $CR = 0.7$	93.3
PSO	swarm= 60, $w = 0.9$, $c_1 = c_2 = 1.49445$	92.4

[1]Full ranking in Fig. 5.

Figure 5 visualises the complete ordering. ES clearly benefits from a larger λ/μ ratio and a moderate number of generations, whereas PSO and DE show limited variance inside the tested ranges. More sophisticated auto-tuning (e.g. Bayesian optimisation) is left for future work.

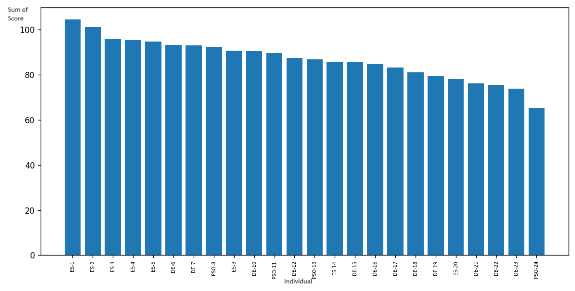

Figure 5: Sum of score for each team obtained by each hyper-parameter configuration (higher is better).

5 Inverse hyper-parameter tuning

Our second experiment reverses the usual optimisation flow: instead of *searching for the roster* given fixed weights, we *search for the weights (and even the algorithm)* that would make a **pre-defined roster profile** appear optimal.

5.1 Problem statement

Let f_θ be the inner auction procedure parameterised by

$$\theta = \big(algo\_id, p_1, p_2, p_3, p_4\big).$$

Given the target KPI triplet $(\bar{s}, \bar{m}, \bar{\ell}) = (100, 4, 0)$ (score, forced picks, leftover credits), we minimise the ℓ_1 discrepancy

$$\mathcal{L}(\theta) = |\, s(\theta) - \bar{s} \,| + |\, m(\theta) - \bar{m} \,| + \big|\, \ell(\theta) - \bar{\ell} \,\big|, \qquad (1)$$

where s, m, ℓ are the KPIs produced by f_θ.

5.2 Outer PSO search

A 30-particle PSO (40 iterations, ω=0.7, $c_1 = c_2$=1.5) explores three algorithm families:

- **PSO:** $\{\omega, c_1, c_2, \text{swarm}\}$
- **DE:** $\{\text{pop}, F_{\text{lo}}, F_{\text{hi}}, CR\}$
- **ES:** $\{\mu + \lambda, \text{ngen}\}$ (4th dim. unused)

5.3 Convergence

Figure 6 shows the steep decay of $\mathcal{L}$; Figure 7 confirms that all three KPIs match their targets after ~30 iterations.

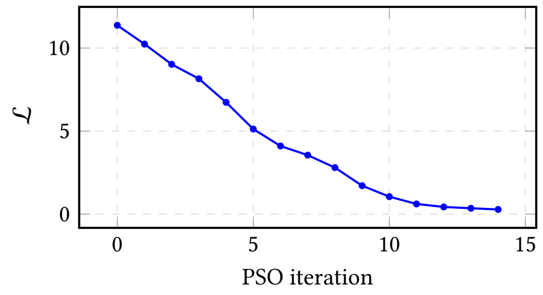

Figure 6: Evolution of the ℓ_1 loss defined in (1).

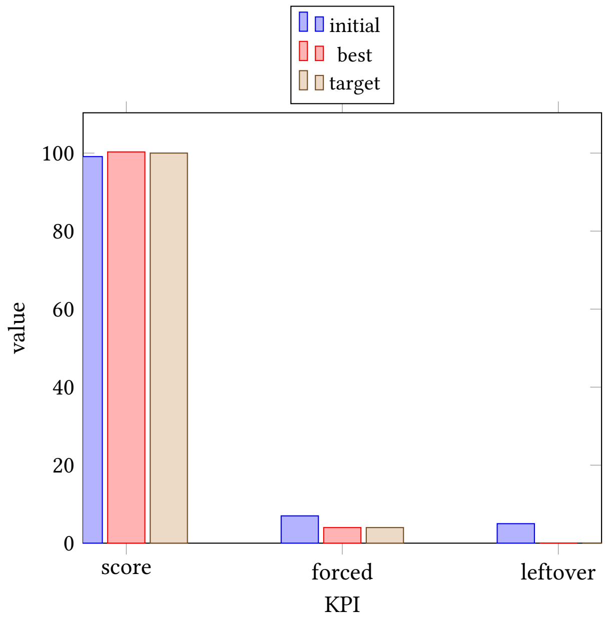

Figure 7: KPI alignment before optimisation, at the optimum, and the desired target.

5.4 Best configurations

Table 6: Top-5 parameter sets (sorted by loss).

Alg.	Hyper-parameters	score	forced	$\mathcal{L}$
DE	pop=10, $F = [0.7, 1.2]$, $CR = 0.7$	100.3	4	0.278
ES	$(\mu + \lambda) = (20, 30)$, ngen=80	98.1	5	1.86
ES	$(\mu + \lambda) = (20, 40)$, ngen=80	97.0	4	1.98
PSO	swarm=60, $\omega = 0.5$, $c_{1,2} = 1.49$	93.5	6	4.08
DE	pop=20, $F = [0.7, 1.2]$, $CR = 0.7$	91.1	7	4.20

Discussion. The outer PSO quickly discovers that *DE* with a compact population and a moderately aggressive differential weight yields the smallest loss. ES variants come second, benefiting from a larger λ/μ ratio, whereas PSO reaches respectable scores but fails to control forced picks. The experiment confirms that *inverse optimisation* can simultaneously (i) select the most suitable meta-heuristic and (ii) fine-tune its parameters so that an *a-priori* roster blueprint becomes ex-post optimal.

6 Conclusion and Future Directions

This project combined **evolutionary computation** (PSO, DE, ES) with concepts from **inverse optimization** to address the complexities of a multi-manager fantasy football auction. The framework incorporated dynamic rebidding, role constraints, and forced assignments, resulting in realistic and tactical squad compositions.

Key Observations

- **Inverse tuning** proved effective in identifying hyper-parameter sets that reproduce a target KPI profile (score = 100, forced = 4, leftover = 0) with very low loss. However, these optimal settings did not always generalize to competitive auctions with multiple dynamic managers.
- **Hyper-parameter tuning had limited effect** on auction outcomes. Despite optimization, performance differences between meta-heuristics remained modest. This indicates that external factors (e.g., draft order, rebid dynamics, random rival behavior) have greater influence than minor parameter adjustments.
- **A**ll three metaheuristics (PSO, DE, and ES) proved effective in real auction simulations, consistently generating competitive teams under realistic budget constraints and role limits. While no single method dominated across all trials, each was capable of producing viable rosters with high scores and efficient resource allocation.

Future Enhancements:

- **User-Friendly Interface and Web Platform**: Develop a graphical user interface (GUI) or a web-based platform to make the system more accessible and intuitive for managers. This interface could guide users through data input and configuration, and offer visual feedback on optimization progress.
- **Automated Line-up Suggestions**: Extend the system to not only propose a full roster but also suggest an optimal starting line-up based on current form and projected performance, thus assisting managers in making final tactical decisions.
- **Parameter Sensitivity and Auto-Tuning**: Future work could explore testing each meta-heuristic with varying parameters to analyse their impact on convergence, forced assignments, and final score. Furthermore, machine learning techniques (e.g., Bayesian optimization or reinforcement learning) could be applied to automatically tune parameters or even adapt the fitness function dynamically.

Even in its prototype form, this system demonstrates that *evolutionary algorithms combined with auction constraints* can generate playable, coherent, and fair fantasy squads. The modular design also supports extensions into richer sports contexts and real-time fantasy scenarios.