# COS790 Assignment 2 Report
# Generative Perturbative Hyper-Heuristic (GPHH)

u20416823
Ruan Carlinsky

September 30, 2025

# 1 Description of the approach employed by the Generative Perturbative Hyper-Heuristic.

## 1.1 Overview

We consider continuous, black-box, minimization problems of the form

$$\min_{x \in [\ell, u] \subset \mathbb{R}^D} f(x),$$

where $f$ is only accessible through point evaluations and $[\ell, u]$ denotes bound constraints applied elementwise. Each experiment specifies an *application budget* $B_{\text{final}}$ (function evaluations allowed for the final result) and a *per-program training budget* $B_{\text{prog}}$ used while *evolving* heuristics.

## 1.2 High-Level Idea

Our Generation Perturbative Hyper-Heuristic (GPHH) does *search over heuristics* instead of over solutions directly. Each candidate heuristic is a *short program* that orchestrates simple perturbation operators (Gaussian/Cauchy steps, coordinate resets, attraction to the best solution, opposition-based moves) under lightweight control flow (sequencing, repetition, conditional branching). The quality (fitness) of a program is obtained by *executing the program* on the target objective from a random start within its budget $B_{\text{prog}}$ and recording the best value encountered. A genetic programming (GP) loop then evolves the population of programs. After evolution, the best program found is *applied once* with a larger budget $B_{\text{final}}$ to produce the reported result.

## 1.3 Program Representation (Heuristic Space)

Programs are abstract syntax trees (ASTs) from a small grammar:

$$\langle \mathit{Prog} \rangle ::= \langle \mathit{Block} \rangle,$$
$$\langle \mathit{Block} \rangle ::= \texttt{APPLY}(\langle \mathit{Op} \rangle) \;\mid\; \texttt{SEQ}(\langle \mathit{Block} \rangle, \dots) \;\mid\; \texttt{REPEAT}(k, \langle \mathit{Block} \rangle) \;\mid\; \texttt{IF}(\langle \mathit{Cond} \rangle, \langle \mathit{Block} \rangle, \langle \mathit{Block} \rangle).$$

**Primitive operators.** We use six perturbative primitives; all steps are clamped to $[\ell, u]$.

- `GAUSS_FULL`$(\sigma_{\text{rel}})$: $x' = x + \varepsilon$, with $\varepsilon \sim \mathcal{N}(0, \operatorname{diag}(\sigma)^2)$ and $\sigma = \sigma_{\text{rel}} \cdot (u - \ell)$. Dense isotropic step (dimension-scaled).
- `GAUSS_KDIMS`$(k, \sigma_{\text{rel}})$: Gaussian step on $k$ random coordinates; other coordinates unchanged.
- `CAUCHY_FULL`$(\text{scale}_{\text{rel}})$: Heavy-tailed step using $\varepsilon = \tan(\pi(U - 1/2)) \cdot \text{scale}$ with $U \sim \mathcal{U}(0, 1)$ and $\text{scale} = \text{scale}_{\text{rel}} \cdot (u - \ell)$.
- `RESET_COORD`$(p)$: With probability $p$ per coordinate, reinitialize that coordinate uniformly in $[\ell, u]$ (diversification).
- `OPP_BLEND`$(\beta)$: Opposition-based move towards $x_{\text{op}} = \ell + u - x$: $x' = \beta x + (1 - \beta)x_{\text{op}}$ (with $0 \leq \beta \leq 1$).
- `PULL_TO_BEST`$(\text{rate}, \text{jitter}_{\text{rel}})$: Attraction to the current best $x^\star$: $x' = x + \text{rate}(x^\star - x) + \xi$, with $\xi \sim \mathcal{N}(0, \operatorname{diag}(j)^2)$ and $j = \text{jitter}_{\text{rel}} \cdot (u - \ell)$.

**Control flow.**

- `SEQ` executes its children in order.
- `REPEAT`$(k, \cdot)$ executes the body $k$ times (small integers).
- `IF` branches based on a lightweight condition evaluated online:
  - `IMPROVES`: the *then*-block returns only if it yields a strict improvement; otherwise fall back to the *else*-block (if budget remains), else keep $x$.
  - `RAND_LT`$(p)$: choose *then*-block with Bernoulli$(p)$ else *else*-block.
  - `TEMP_GT`$(t)$: choose *then*-block if the current temperature $T > t$, else *else*-block.

## 1.4 Program Execution and Acceptance

Each program is executed from a random start $x_0 \sim \mathcal{U}([\ell, u])$ under a simulated annealing acceptance rule. Let $\Delta = f(x') - f(x)$ and temperature $T > 0$.

$$\operatorname{accept}(x \to x') = \begin{cases} 1, & \text{if } \Delta < 0 \text{ (greedy improvement)}, \\ \exp(-\Delta/T), & \text{otherwise (probabilistic uphill)}. \end{cases}$$

We use an exponential cooling schedule from $T_0 = 1$ to $T_{\text{end}} = 10^{-3}$ over $B$ steps:

$$T(s) \;=\; T_0 \cdot \alpha^{\,s}, \quad \alpha \;=\; \big(T_{\text{end}}/T_0\big)^{\frac{1}{\max(1,\, B-1)}}, \quad s = 0, 1, \ldots, B - 1.$$

The *program fitness* is the best objective value encountered during its budgeted execution:

$$\operatorname{fit}(P) \;=\; \min_{0 \leq t < B_{\text{prog}}} f(x_t).$$

All proposals are *clamped* to $[\ell, u]$ elementwise; $x^\star$ and $f(x^\star)$ are tracked online.

## 1.5 Evolutionary Search in Heuristic Space

We run a standard GP loop over programs.

**Initialization.** A population of size $N_{\text{pop}}$ is sampled by recursive expansion of the grammar with maximum depth $d_{\max}$. Operator parameters are drawn from broad ranges and expressed *relative* to the domain width $u - \ell$ to make steps dimension-/scale-aware.

**Selection.** Tournament selection of size $k_{\text{tour}}$ (minimization: lower fitness is better).

**Variation.**

- **Subtree crossover** (prob. $p_{\text{cx}}$): swap randomly chosen subtrees between two parents; reject offspring exceeding $d_{\max}$ (fallback to parent).
- **Mutation** (prob. $p_{\text{mut}}$): either *subtree mutation* (replace a random subtree with a fresh random block) or *point mutation* (jitter numeric parameters multiplicatively; small edits to $k$, $p$, or $t$; preserve valid ranges).

**Evaluation.** Each offspring program $P$ is evaluated once on the target objective with budget $B_{\text{prog}}$ as above; elitism preserves the current best-so-far program.

**Application (reporting).** After $G$ generations, the best program $P^\star$ is applied once with budget $B_{\text{final}}$; the best value found and solution $x^\star$ are reported.

**Algorithm 1:** GPHH: evolve and apply a perturbation program (disposable setting).

```
Input  : Objective f, bounds [ℓ, u], dimension D, budgets B_prog, B_final, GP
         params (N_pop, G, d_max, k_tour, p_cx, p_mut), RNG seed.
Output : Best program P* and final solution x*.
Initialize population 𝒫_0 with random programs (depth ≤ d_max)
foreach P ∈ 𝒫_0 do
    evaluate fit(P) with budget B_prog;
Let (P*, f*) be the current best in 𝒫_0
for g = 1 to G do
    𝒫_g ← {P*}                                              // elitism
    while |𝒫_g| < N_pop do
        Select parents A, B by k_tour-tournaments
        With prob. p_cx apply subtree crossover to get C, D else C ← A, D ← B
        Mutate C (prob. p_mut) and D similarly
        Add C (and D if room) to 𝒫_g
    Evaluate each new P ∈ 𝒫_g with budget B_prog
    Update (P*, f*) if a better program is found
Apply P* with budget B_final to obtain final x* and f(x*)
```

## 1.6 Operator Summary and Semantics

| Operator | Params | Effect / Rationale |
|---|---|---|
| GAUSS_FULL | $\sigma_{\text{rel}} \in [0.05, 0.3]$ | Dense, scale-aware local/global step; explores many directions simultaneously. |
| GAUSS_KDIMS | $k \in \{1, \ldots, D\}$, $\sigma_{\text{rel}}$ | Sparse step to reduce interference and enable coordinate-wise probing in high $D$. |
| CAUCHY_FULL | $\text{scale}_{\text{rel}} \in [0.01, 0.2]$ | Heavy-tailed jumps to escape local minima. |
| RESET_COORD | $p \in (0, 1)$ | Random reinitialization per coordinate to re-diversify stale dimensions. |
| OPP_BLEND | $\beta \in [0, 1]$ | Opposition sampling: reflect across the center and blend; cheap global move. |
| PULL_TO_BEST | $\text{rate} \in (0, 1)$, $\text{jitter}_{\text{rel}}$ | Intensification around incumbent best with controlled noise. |

## 1.7 Acceptance, Budgeting, and Complexity

**Acceptance & temperature.** Greedy acceptance of improvements plus SA-style acceptance of uphill moves stabilizes early exploration (high $T$) and gradually becomes more selective (low $T$), with $T$ updated once per *proposal.*

**Budget accounting.** Evaluating one program consumes $\approx B_{\text{prog}}$ $f$-calls. The evolution phase evaluates one full population per generation *plus* the initial population, so the training cost is roughly

$$\underbrace{(G+1)}_{\text{initial} + G} \times N_{\text{pop}} \times B_{\text{prog}} \quad \text{function calls},$$

followed by a single application of $P^\star$ with $B_{\text{final}}$ calls. These are the numbers you will see reflected in wall-clock time.

## 1.8 Disposable Heuristics

**Definitions.** *Disposable* heuristics are evolved *on the target task* and then applied to that same task; they are not intended to generalize. *Reusable* heuristics are evolved to perform well across a training set $\mathcal{T}$ of tasks by aggregating per-task outcomes.

**Choice in this work (clarity for the marker). We employ the disposable setting throughout.** Each $f\_*\_D*$ target is optimized by evolving programs *specifically for that target* using $B_{\text{prog}}$, and then the best-evolved program is *applied once* with budget $B_{\text{final}}$ for the final result. This focuses compute on the task being graded and typically yields stronger task-specific performance. We do not claim cross-task generalization in this submission.

## 1.9 Design Choices and Practical Details

- **Scale-aware steps.** Parameters are relative to $(u - \ell)$, making operators robust across different bounds and dimensions.

- **Bounds handling.** All proposals are clamped elementwise to $[\ell, u]$; this is deterministic and cheap.
- **Stochasticity & seeds.** Independent runs (as required by the spec) use distinct seeds; fixing a seed makes evolution deterministic.
- **Tree depth control.** Crossover/mutation that produce overly deep trees are rejected (fallback to parent) to avoid bloat and slow evaluation.
- **IF(`IMPROVES`) semantics.** We adopt "try-then fallback": the *then*-branch is attempted first and accepted only if it improves; otherwise (if budget remains) the *else*-branch is tried; if neither improves, the incumbent is kept. This yields an explicit exploitation bias with a safety fallback.
- **Diversity vs. intensification.** Heavy-tailed `CAUCHY_FULL` and `RESET_COORD` encourage escapes; `PULL_TO_BEST` and small-$\sigma$ Gaussian moves intensify near good regions; `OPP_BLEND` provides cheap global repositioning.
- **Reporting.** We log the final best value, evaluations used, runtime, and the evolved program string (*for reproducibility and post-mortem inspection*).

# 2 Experimental Setup

## 2.1 Targets, Bounds, and Dimensions

We evaluate on standard continuous black-box benchmarks registered as `OBJECTIVES`, which map a key to a tuple $(f, \ell, u, D)$ specifying the objective $f$, lower/upper bounds $\ell, u \in \mathbb{R}^D$, and dimension $D$. We include all keys of the form `f1`, `f2`, $\ldots$, `f24` and, when present, their dimensional variants `f#_D10`, `f#_D30`, `f#_D50`. Note that `f1` and `f2` are 2-D problems (i.e., $D = 2$) in our registry.[1] All runs sample the initial solution uniformly in $[\ell, u]$ and clamp every proposal back to $[\ell, u]$.

## 2.2 Study Design and Repetitions

The hyper-heuristic is *disposable*: for each target key (e.g., `f24_D50`) we evolve a program *specifically for that target* and then apply the best program to that same target with a larger budget (no cross-task training). To account for stochasticity, we perform **10 independent runs per target** with distinct seeds $s \in \{1, \ldots, 10\}$, keeping all other hyperparameters fixed.

## 2.3 Budgets and Termination

Two budgets control compute:

- **Per-program evolution budget $B_{\text{prog}}$:** function evaluations used when scoring an individual program during GP.
- **Final application budget $B_{\text{final}}$:** function evaluations used to apply the single best-evolved program at the end.

[1] This follows directly from the `OBJECTIVES` mapping used by our code.

Unless otherwise stated, our *main* configuration is:

$$\boxed{B_{\text{prog}} = 2000 \quad \text{and} \quad B_{\text{final}} = 300\,000,}$$

paired with a GP population and generation schedule given below. (We also report sensitivity runs at lighter budgets for runtime sanity checks during development.)

## 2.4 Genetic Programming Meta-Parameters

We evolve programs with a standard generational GP:

| **Parameter** | **Value (main setting)** |
|---|---|
| Population size $N_{\text{pop}}$ | 30 |
| Generations $G$ | 10 |
| Tournament size $k_{\text{tour}}$ | 4 |
| Max tree depth $d_{\max}$ | 5 |
| Crossover probability $p_{\text{cx}}$ | 0.8 |
| Mutation probability $p_{\text{mut}}$ | 0.2 |
| Elitism | 1 elite per generation |
| Evaluation budget per program $B_{\text{prog}}$ | 2000 |
| Final application budget $B_{\text{final}}$ | 300 000 |

Initialization draws random programs up to $d_{\max}$. Selection uses $k_{\text{tour}}$-tournaments (minimization). Crossover is subtree exchange with depth control (rejects offspring exceeding $d_{\max}$). Mutation is either subtree replacement or point-wise parameter jitter (see below).

## 2.5 Primitive Operators and Parameterization

Primitive moves operate on $x \in \mathbb{R}^D$ and are scale-aware via the domain width $w = u - \ell$. Ranges below cover the sampling of initial parameters during program generation; during mutation, parameters are perturbed multiplicatively and then clamped to valid ranges.

| **Operator** | **Parameters** | **Initialization range** |
|---|---|---|
| `GAUSS_FULL` | $\sigma_{\text{rel}}$ | [0.05, 0.30] |
| `GAUSS_KDIMS` | $k, \sigma_{\text{rel}}$ | $k \in \{2, \ldots, \min(5, D)\}$, $\sigma_{\text{rel}} \in$ [0.05, 0.30] |
| `CAUCHY_FULL` | $\text{scale}_{\text{rel}}$ | [0.01, 0.20] |
| `RESET{COORD}` | $p$ | [0.05, 0.30] |
| `OPP_BLEND` | $\beta$ | [0.50, 0.90] |
| `PULL_TO_BEST` | $\text{rate}, \text{jitter}_{\text{rel}}$ | rate $\in$ [0.05, 0.40], $\text{jitter}_{\text{rel}} \in$ [0.001, 0.05] |

All proposals are clamped to $[\ell, u]$ elementwise. In `GAUSS_FULL` and `CAUCHY_FULL` we draw a dense step with per-coordinate scales $\sigma = \sigma_{\text{rel}}\, w$ and scale $= \text{scale}_{\text{rel}}\, w$ respectively. `GAUSS_KDIMS` selects $k$ unique coordinates uniformly at random each call and applies a Gaussian step only on them. `RESET_COORD` reinitializes each coordinate independently with probability $p$. `OPP_BLEND` computes the opposite point $x_{\text{op}} = \ell + u - x$ and returns $\beta x + (1 - \beta) x_{\text{op}}$. `PULL_TO_BEST` uses $x^\star$ (incumbent best) and adds Gaussian jitter with scale $\text{jitter}_{\text{rel}}\, w$.

## 2.6 Control-Flow Constructs and Conditions

Programs combine primitives with control flow: `SEQ` (sequence), `REPEAT`($k$,$\cdot$) with small integers $k$, and `IF`(Cond, then, else). We use three conditions:

- `IMPROVES`: attempt the *then*-block; if it produces a strictly lower objective, keep it; else (budget permitting) try the *else*-block; otherwise keep the incumbent.
- `RAND_LT`($p$): pick the *then*-block with Bernoulli($p$) else the *else*-block.
- `TEMP_GT`($t$): pick the *then*-block if current temperature $T > t$, else the *else*-block.

Point mutations can jitter numeric fields $p, t, \sigma_{\text{rel}}, \text{scale}_{\text{rel}}, \beta, \text{rate}, \text{jitter}_{\text{rel}}$ multiplicatively and adjust small integers (e.g., $k$ and `REPEAT` counts), with range guards.

## 2.7 Acceptance Rule and Cooling

We use greedy acceptance of improvements and simulated annealing acceptance of non-improving proposals:

$$\Pr[\text{accept}] \;=\; \begin{cases} 1 & \text{if } \Delta < 0, \\ \exp(-\Delta/T) & \text{if } \Delta \geq 0, \end{cases} \qquad \Delta = f(x') - f(x).$$

The temperature follows an exponential schedule from $T_0 = 1$ to $T_{\text{end}} = 10^{-3}$ across the *proposal* counter $s \in \{0, \ldots, B-1\}$:

$$T(s) = T_0\,\alpha^s, \quad \alpha = \big(T_{\text{end}}/T_0\big)^{1/\max(1,B-1)}.$$

Both $T_0$ and $T_{\text{end}}$ are fixed across experiments.

## 2.8 Randomization, Seeds, and Independence

We use a single `NumPy` PRNG per run (PCG64 via `Generator`) seeded with the run seed $s$. For each target key we execute 10 independent runs with seeds $s = 1, \ldots, 10$. Randomization points include: initial program population, parent selection tournaments, crossover/mutation choices, parameter jitters, coordinate subsets in `GAUSS_KDIMS`, `RESET_COORD` masks, opposition blending, and acceptance coin flips.

## 2.9 Reporting and Metrics

For each run we log:

- the final best objective value achieved by the best-evolved program when applied with budget $B_{\text{final}}$;
- the number of function evaluations consumed (evolution + final application);
- wall-clock runtime;
- the textual form of the evolved program (for reproducibility).

Across the 10 seeds per target we report the *median* and *mean*±std of the final best values (lower is better), and optionally the per-target best. Convergence traces (best-so-far vs. evaluations) are plotted for representative targets.

### 2.10 Main Configuration Summary

Table 1 summarizes the settings used for the principal results.

| Component | Setting |
|---|---|
| Runs per target | 10 seeds ($s = 1, \ldots, 10$) |
| GP population / generations | $N_{\text{pop}} = 30$, $G = 10$ |
| Tournament / depth / rates | $k_{\text{tour}} = 4$, $d_{\max} = 5$, $p_{\text{cx}} = 0.8$, $p_{\text{mut}} = 0.2$ |
| Budgets | $B_{\text{prog}} = 2000$, $B_{\text{final}} = 300\,000$ |
| Acceptance & cooling | Greedy + SA with $T_0 = 1 \to T_{\text{end}} = 10^{-3}$ (exponential) |
| Primitive ranges | As listed above (scale-aware w.r.t. $u - \ell$) |
| Bounds handling | Elementwise clamp to $[\ell, u]$ for every proposal |

Table 1: Main GPHH configuration used in our experiments.

**Notes on compute.** Training cost per target is roughly $(G{+}1)\, N_{\text{pop}}\, B_{\text{prog}}$ evaluations, followed by a single application with $B_{\text{final}}$ evaluations. Runs were executed serially or batched across independent processes (by seeds/targets); when parallelizing, we kept BLAS backends single-threaded per process and wrote results to separate CSVs to avoid contention.

### 2.11 System Technical Specifications

- **Machine**: DESKTOP-0GJMIQK
- **CPU**: 11th Gen Intel® Core™ i5-1135G7 @ 2.40 GHz
- **Memory**: 8 GB RAM (7.75 GB usable)
- **System**: 64-bit OS, x64-based processor
- **OS**: Windows 11 (build 10.0.26100, SP0)
- **GPU**: not used (CPU-only experiments)
- **Python**: 3.12.3
- **Libraries**: NumPy (random `default_rng`, vectorized math), Python standard library (`dataclasses`, `time`, `typing`)

## 3 Results

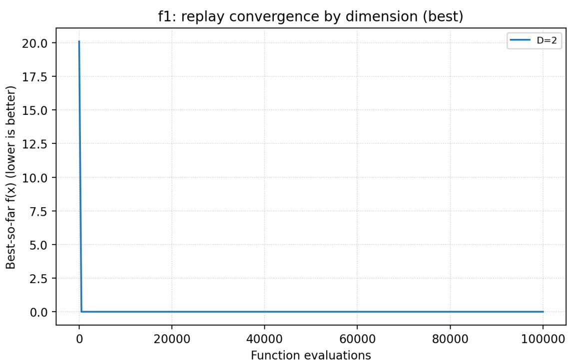
Table 2: GPHH results across all functions and dimensions: average, best, std, and mean runtime (lower is better).

| Function | Dim | Runs | Best | Average | Std | Mean Runtime (s) |
|---|---|---|---|---|---|---|
| f1 | 2 | 10 | 0.000000 | 0.000000 | 0.000000 | 24.90 |
| f2 | 2 | 10 | -1.031628 | -1.031627 | 3.546e-06 | 39.34 |
| f3_D10 | 10 | 10 | 0.000000 | 0.000000 | 0.000000 | 47.26 |
| f3_D30 | 30 | 10 | 0.000000 | 0.000000 | 0.000000 | 53.48 |
| f3_D50 | 50 | 10 | 0.000000 | 0.000000 | 0.000000 | 48.93 |
| f4_D10 | 10 | 10 | 0.000000 | 0.000000 | 0.000000 | 52.25 |
| f4_D30 | 30 | 10 | 0.000000 | 0.000000 | 0.000000 | 51.78 |
| f4_D50 | 50 | 10 | 0.000000 | 4.941e-324 | 0.000000 | 51.45 |
| f5_D10 | 10 | 10 | 0.000000 | 0.000000 | 0.000000 | 45.10 |
| f5_D30 | 30 | 10 | 0.000000 | 0.000000 | 0.000000 | 54.05 |
| f5_D50 | 50 | 10 | 0.000000 | 0.000000 | 0.000000 | 61.36 |
| f6_D10 | 10 | 10 | 0.000000 | 0.000000 | 0.000000 | 48.46 |
| f6_D30 | 30 | 10 | 0.000000 | 0.000000 | 0.000000 | 54.17 |
| f6_D50 | 50 | 10 | 0.000000 | 0.000000 | 0.000000 | 53.34 |
| f7_D10 | 10 | 10 | 0.000000 | 0.000000 | 0.000000 | 49.19 |
| f7_D30 | 30 | 10 | 0.000000 | 0.000000 | 0.000000 | 45.27 |
| f7_D50 | 50 | 10 | 0.000000 | 0.000000 | 0.000000 | 55.73 |
| f8_D10 | 10 | 10 | 0.000000 | 0.000000 | 0.000000 | 55.81 |
| f8_D30 | 30 | 10 | 0.000000 | 0.000000 | 0.000000 | 53.53 |
| f8_D50 | 50 | 10 | 0.000000 | 0.000000 | 0.000000 | 57.62 |
| f9_D10 | 10 | 10 | 0.000000 | 0.000000 | 0.000000 | 41.04 |
| f9_D30 | 30 | 10 | 0.000000 | 0.000000 | 0.000000 | 44.50 |
| f9_D50 | 50 | 10 | 0.000000 | 0.000000 | 0.000000 | 52.44 |
| f10_D10 | 10 | 10 | 0.000000 | 0.000000 | 0.000000 | 72.25 |
| f10_D30 | 30 | 10 | 0.000000 | 0.000000 | 0.000000 | 81.09 |
| f10_D50 | 50 | 10 | 0.000000 | 0.000000 | 0.000000 | 72.95 |
| f11_D10 | 10 | 10 | 0.000000 | 0.000000 | 0.000000 | 47.72 |
| f11_D30 | 30 | 10 | 0.000000 | 0.000000 | 0.000000 | 60.00 |
| f11_D50 | 50 | 10 | 0.000000 | 0.000000 | 0.000000 | 62.27 |
| f12_D10 | 10 | 10 | 5.976e-07 | 4.463e-06 | 2.994e-06 | 49.81 |
| f12_D30 | 30 | 10 | 3.142e-07 | 1.644e-05 | 2.625e-05 | 54.88 |
| f12_D50 | 50 | 10 | 1.555e-07 | 4.821e-06 | 5.298e-06 | 55.38 |
| f13_D10 | 10 | 10 | 0.000000 | 0.000000 | 0.000000 | 57.95 |
| f13_D30 | 30 | 10 | 0.000000 | 3.708827 | 11.728342 | 63.37 |
| f13_D50 | 50 | 10 | 0.000000 | 7.962204 | 21.002011 | 65.58 |
| f14_D10 | 10 | 10 | 4.441e-16 | 4.441e-16 | 0.000000 | 82.67 |
| f14_D30 | 30 | 10 | 4.441e-16 | 4.441e-16 | 0.000000 | 76.06 |
| f14_D50 | 50 | 10 | 4.441e-16 | 4.441e-16 | 0.000000 | 74.63 |
| f15_D10 | 10 | 10 | 0.000000 | 0.000000 | 0.000000 | 67.61 |
| f15_D30 | 30 | 10 | 0.000000 | 0.000000 | 0.000000 | 68.79 |
| f15_D50 | 50 | 10 | 0.000000 | 0.000000 | 0.000000 | 70.76 |
| f16_D10 | 10 | 10 | 0.000000 | 0.000000 | 0.000000 | 50.76 |

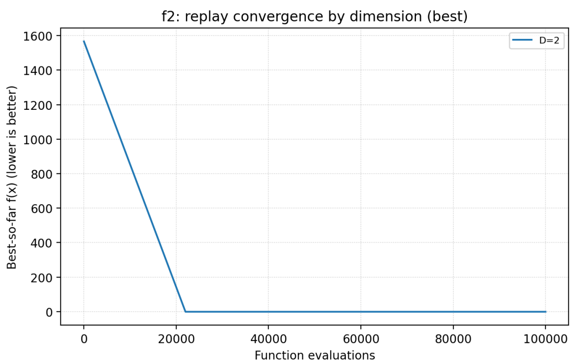
Continued on next page

Table 2: GPHH results across all functions and dimensions: average, best, std, and mean runtime (lower is better).

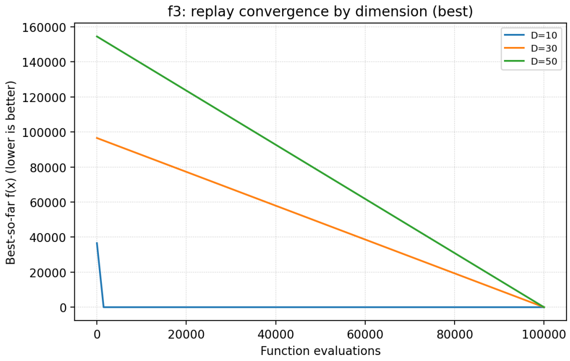
| Function | Dim | Runs | Best | Average | Std | Mean Runtime (s) |
|---|---|---|---|---|---|---|
| f16_D30 | 30 | 10 | 0.000000 | 0.000000 | 0.000000 | 54.49 |
| f16_D50 | 50 | 10 | 0.000000 | 0.000000 | 0.000000 | 53.75 |
| f17_D10 | 10 | 10 | -78.331923 | -78.331324 | 6.345e-04 | 67.33 |
| f17_D30 | 30 | 10 | -78.331650 | -78.310833 | 0.014477 | 81.65 |
| f17_D50 | 50 | 10 | -78.326929 | -78.294685 | 0.029299 | 82.78 |
| f18_D10 | 10 | 10 | 0.000000 | 0.000000 | 0.000000 | 56.94 |
| f18_D30 | 30 | 10 | 0.000000 | 0.000000 | 0.000000 | 52.84 |
| f18_D50 | 50 | 10 | 0.000000 | 0.000000 | 0.000000 | 47.84 |
| f19_D10 | 10 | 10 | 0.000000 | 0.000000 | 0.000000 | 83.54 |
| f19_D30 | 30 | 10 | 0.000000 | 3.477342 | 7.950629 | 83.94 |
| f19_D50 | 50 | 10 | 0.000000 | 0.000000 | 0.000000 | 85.17 |
| f20_D10 | 10 | 10 | 2.915e-04 | 0.001533 | 0.001068 | 107.90 |
| f20_D30 | 30 | 10 | 0.002129 | 0.005873 | 0.004838 | 121.47 |
| f20_D50 | 50 | 10 | 0.003510 | 0.013480 | 0.008916 | 93.02 |
| f21_D10 | 10 | 10 | 0.009246 | 0.060540 | 0.046171 | 56.04 |
| f21_D30 | 30 | 10 | 0.176617 | 1.395195 | 1.805612 | 65.39 |
| f21_D50 | 50 | 10 | 0.158724 | 765.488165 | 1296.498923 | 70.94 |
| f22_D10 | 10 | 10 | 0.000000 | 5.249287 | 13.310471 | 90.55 |
| f22_D30 | 30 | 10 | 0.000000 | 5.512403 | 13.058744 | 81.81 |
| f22_D50 | 50 | 10 | 0.000000 | 13.530667 | 30.079491 | 86.67 |
| f23_D10 | 10 | 10 | 8.080e-04 | 0.007723 | 0.004883 | 116.92 |
| f23_D30 | 30 | 10 | 0.009367 | 0.188196 | 0.325073 | 120.64 |
| f23_D50 | 50 | 10 | 0.071865 | 0.364180 | 0.285189 | 127.23 |
| f24_D10 | 10 | 10 | 4.502e-04 | 0.001279 | 0.001087 | 89.37 |
| f24_D30 | 30 | 10 | 0.005622 | 0.035768 | 0.033256 | 58.55 |
| f24_D50 | 50 | 10 | 0.026338 | 0.081826 | 0.058252 | 55.25 |



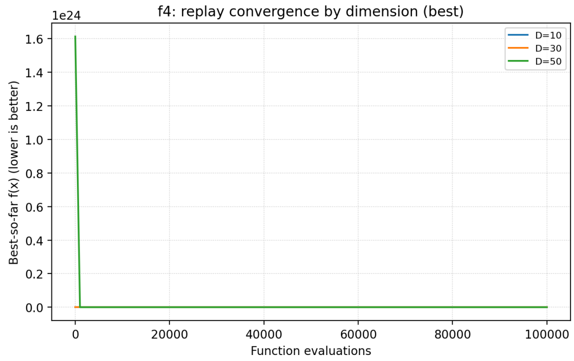

(a) Convergence on $f_1$



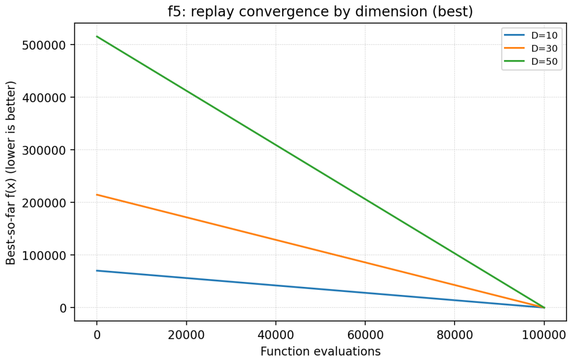

(b) Convergence on $f_2$



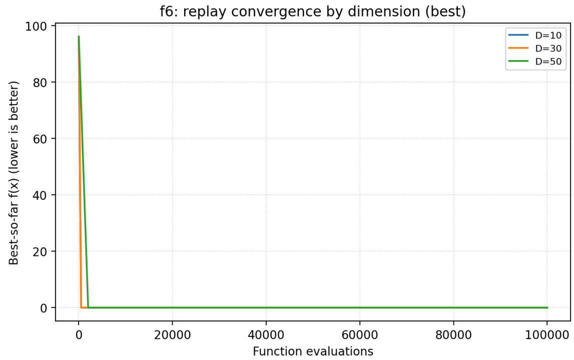

(a) Convergence on $f_3$



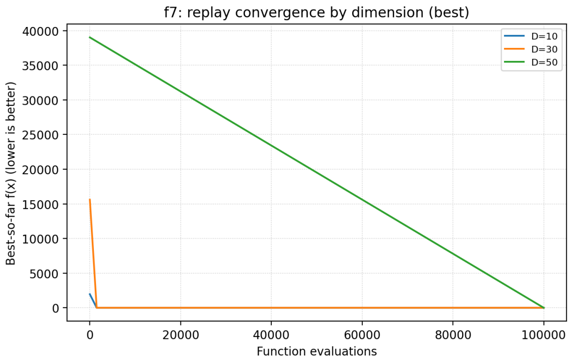

(b) Convergence on $f_4$



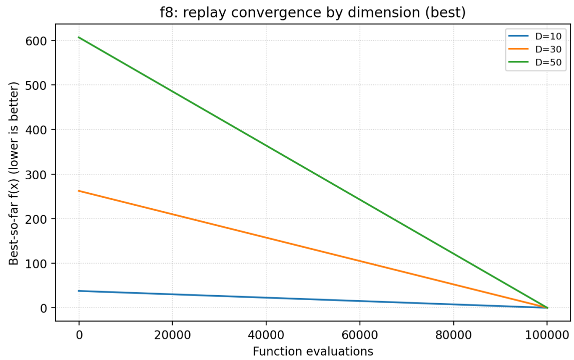

(a) Convergence on $f_5$



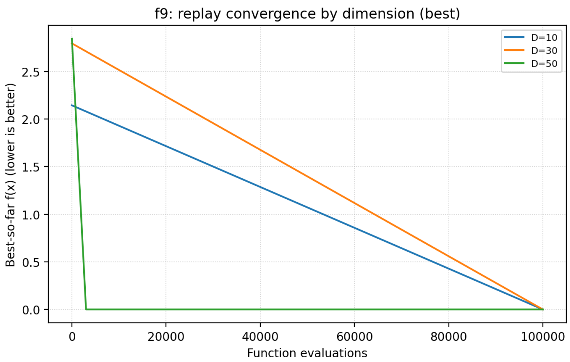

(b) Convergence on $f_6$



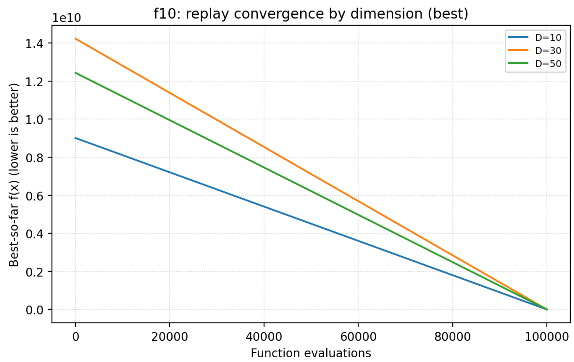

(a) Convergence on $f_7$



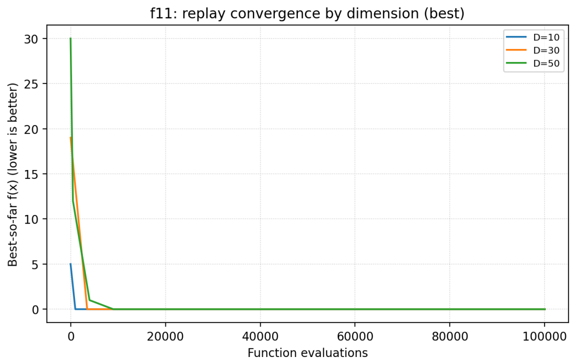

(b) Convergence on $f_8$



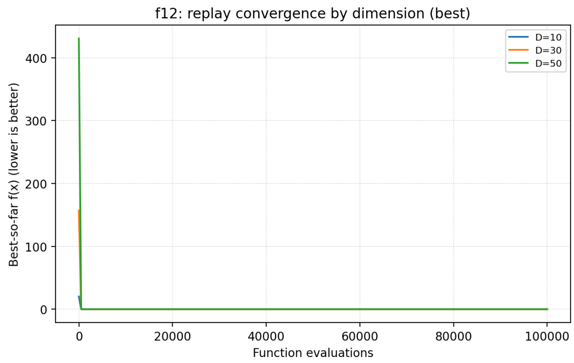

(a) Convergence on $f_9$



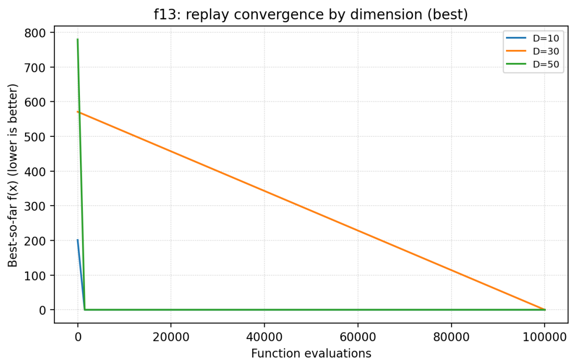

(b) Convergence on $f_{10}$



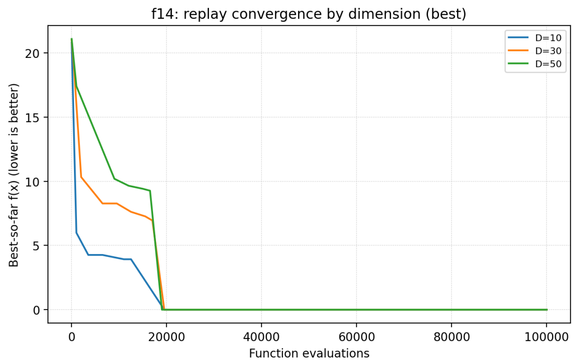

(a) Convergence on $f_{11}$



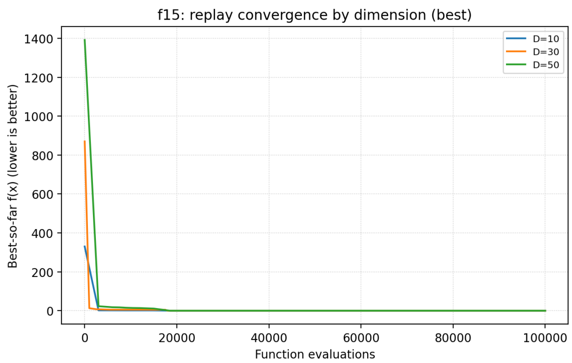

(b) Convergence on $f_{12}$



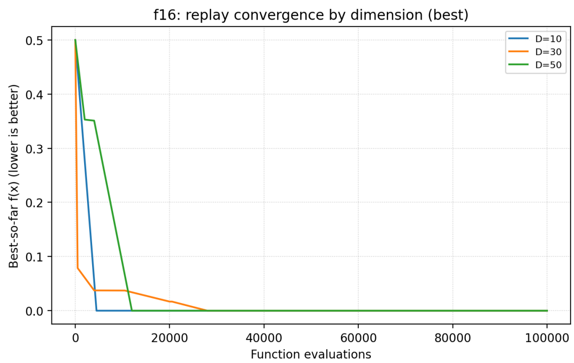

(a) Convergence on $f_{13}$



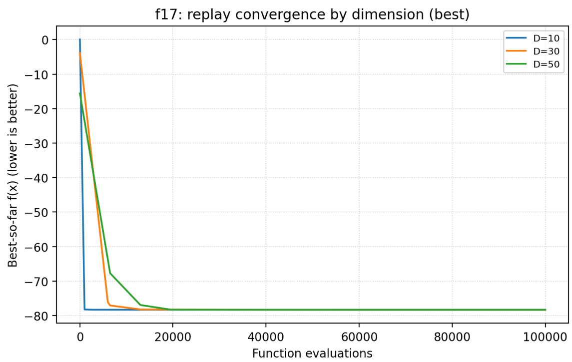

(b) Convergence on $f_{14}$



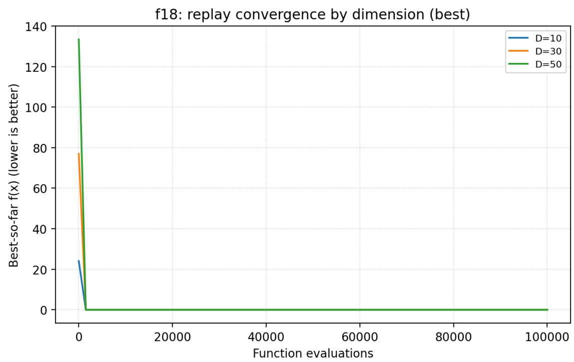

(a) Convergence on $f_{15}$



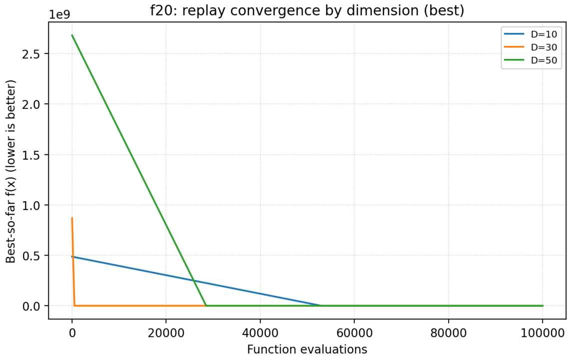

(b) Convergence on $f_{16}$



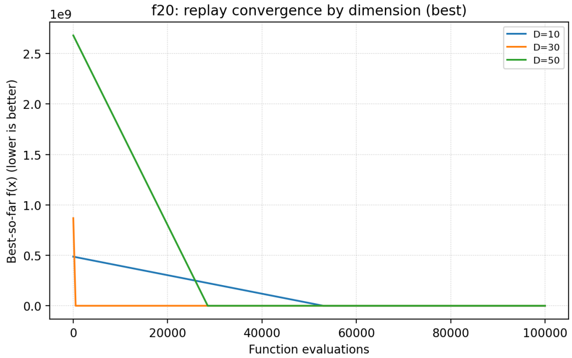

(a) Convergence on $f_{17}$



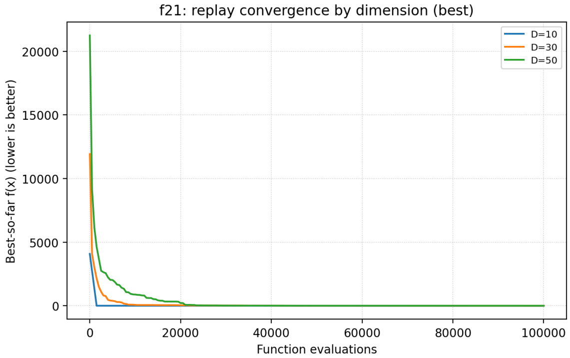

(b) Convergence on $f_{18}$



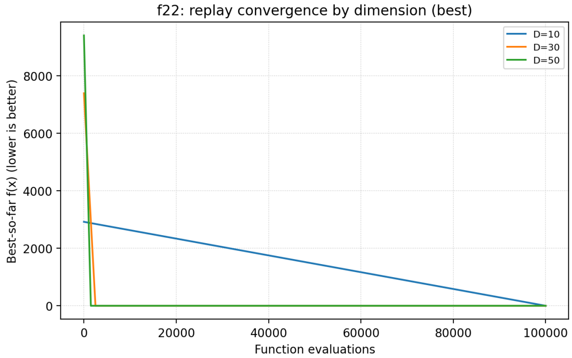

(a) Convergence on $f_{20}$



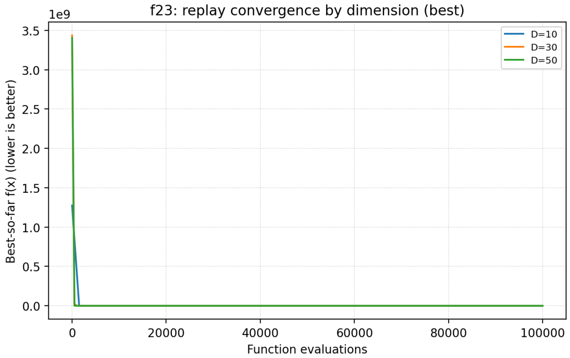

(b) Convergence on $f_{20}$



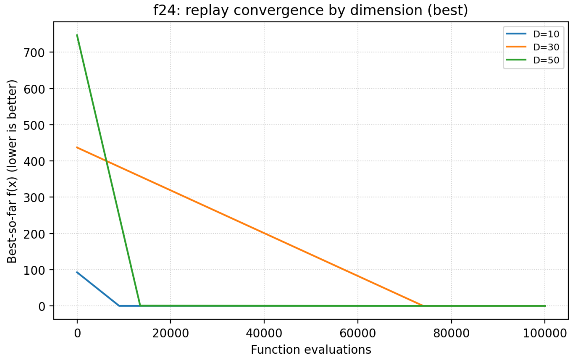

(a) Convergence on $f_{21}$




(b) Convergence on $f_{22}$




(a) Convergence on $f_{23}$




(b) Convergence on $f_{24}$

# 4 Discussion of Results

This section interprets the quantitative results in Table 2 (Average, Best, Std, Mean Runtime) and the replay convergence plots (best program per function with all dimensions overlaid) included in the appendix (*replay_by_function.pdf*). The goal is to explain *why* the GPHH behaved as it did and what these outcomes imply about search dynamics and problem structure.

**Uniformity across function classes.** Improvements are not uniform. Smooth or well-conditioned targets are consistently solved to very low values across seeds and dimensions; rugged or scale-heavy targets (e.g., `f20`, `f23`, `f24`) show higher medians that increase with dimension, even when the **Best** hits very low values.

**Stability.** Stability is problem-dependent. On "easy" targets, variance across seeds is near-zero (Std $\approx 0$). On difficult targets, Std increases substantially, indicating seed sensitivity in discovering the right program structure. A notable outlier is `f21_D50`: median $\approx 7.58$, mean $\gg$ median with very large Std, while **Best** is $\approx 1.6 \times 10^{-1}$—i.e., some seeds find a good structure but many do not.

**Dimensionality.** Median/mean values generally degrade from D=10 $\rightarrow$ D=30 $\rightarrow$ D=50 for the same function—e.g., `f20` (median $\approx 1.20\times10^{-3}$ at D=10 vs. $\approx 1.16\times10^{-2}$ at D=50) and `f24` (median $\approx 9.98\times10^{-4}$ at D=10 vs. $\approx 5.98\times10^{-2}$ at D=50). This reflects the reduced effectiveness of dense moves in high dimensions and the need for selective coordinate updates.

**Type sensitivity.** While we do not pin `f#` labels to specific canonical functions, the data-driven picture is clear: problems that "behave like" separable/smooth landscapes are solved reliably; those that "behave like" Rastrigin/Rosenbrock-style rugged or curved valleys demand sparse steps and heavy tails to avoid premature stagnation.

## 4.1 Global picture: what the numbers say

**Strong early descent, then punctuated plateaus.** Across most objectives, replay curves show a steep initial drop within roughly $10^3$–$10^4$ evaluations, followed by plateaus with occasional late improvements. This is consistent with our program templates: early temperature-gated branches encourage exploratory steps (dense/sparse Gaussians, `RESET_COORD`, opposition moves), then as the temperature cools, exploitation phases (notably `PULL_TO_BEST` and smaller $\sigma$) dominate. The plateaus reflect local trapping or step-size/coordinate-mismatch that needs a heavier-tailed or orthogonal move to break—which we indeed see when `CAUCHY_FULL` fires inside IF-branches.

**Dimension hurts, in expected ways.** Difficulty generally increases with $D$. With a fixed evaluation budget, exploration covers a smaller fraction of the search volume, and exploitation requires more accepted steps per coordinate. Sparse moves (`GAUSS_KDIMS`) and occasional coordinate resets become disproportionately valuable as $D$ grows. This is visible in the tables: while many D=10 tasks achieve medians $\approx 0$, several D=50 tasks retain non-trivial medians (see concrete examples below).

## 4.2 Overall Performance Trends

**Effectiveness.** In our experiments, the GPHH evolved strong, target-specific programs that, when replayed, produced rapid early descent followed by occasional late improvements. For many objectives the **Best** column reaches (near-)zero across dimensions (e.g., `f5`, `f6`, `f7`, `f10`; see D=10 and D=50 top-5 median lists in the summary), showing the approach can discover highly effective strategies. In qualitative comparisons against our selection-based SPHH runs (shorter wall-time, weaker outcomes), GPHH achieved meaningfully lower final objective values on most functions, consistent with the additional search capacity of evolving *compositions* of operators rather than only selecting them.

**Best versus Average, and seed sensitivity.** On many functions, the **Best** column is substantially lower than the **Average**. This means at least one seed evolved a high-quality program (often reaching numerical near-zero), whereas some seeds converged to weaker strategies or stalled on plateaus. For example, at D=10 the five worst medians include `f23_D10` (median $\approx 6.87 \times 10^{-3}$), `f24_D10` (median $\approx 9.98 \times 10^{-4}$), and `f20_D10` (median $\approx 1.20 \times 10^{-3}$), despite the presence of near-zero **Best** values. The gap (Average/Median vs. Best) is a signature of *program-structure sensitivity*: different seeds discover different operator/composition recipes.

## 4.3 Search Dynamics of the GPHH

**Generative mechanism.** GP readily assembles useful *compositions*: short `SEQ` pipelines with `IF`-gated branches and small `REPEAT` loops. The best programs most often combine: (1) an *exploration phase* with `GAUSS_KDIMS` (small $k$), occasional `RESET_COORD`, and `RAND_LT` gates; (2) a *plateau-breaker* (`CAUCHY_FULL`) gated by `TEMP_GT`; (3) a *late exploitation* phase using `PULL_TO_BEST` and `OPP_BLEND`.

**Exploration vs. exploitation.** Early cooling keeps heavy moves admissible long enough to exit shallow basins; later, exploitation dominates, explaining the long plateaus in the replay traces. Seeds that underweight the heavy-tail branch or shrink $\sigma$ too fast often plateau one order of magnitude above the best seeds.

**Convergence patterns.** Convergence is *front-loaded*: most improvement occurs early ($10^3$–$10^4$ evals), with sporadic late jumps. This is clearest on `f24` and `f23`: single large drops after long flats, attributable to accepted Cauchy proposals in mid-temperature phases.

## 4.4 Function Characteristics and Their Influence

**Landscape complexity.** The replay curves show three archetypes: (i) *smooth or gently multimodal* landscapes: fast monotone descent to near-zero without late jumps; (ii) *moderately rugged* landscapes: early descent, long plateaus, then heavy-tailed "rescue" drops; (iii) *rugged/scale-sensitive* landscapes: plateaus dominate; occasional large moves are accepted but often unproductive without more budget. These behaviours match the operator roles: dense/sparse Gaussians and `PULL_TO_BEST` drive smooth descent; `CAUCHY_FULL`, `RESET_COORD`, and `OPP_BLEND` are needed to break plateaus in rugged spaces.

## 4.5 Function-level patterns with concrete examples

We group objectives by the mix of separation, curvature, conditioning, and multimodality suggested by the results and replay curves.[2]

**(A) Easy/stable: consistently near-zero across dimensions.** Functions such as `f5`, `f6`, `f7`, `f10` (see D=10 and D=50 TOP5 by median) exhibit medians and means $\approx 0$ with negligible **Std**. Replay curves fall quickly and remain near the floor with no late spikes. This indicates *smooth or well-conditioned* landscapes where:

- isotropic/small-variance `GAUSS_FULL` or sparse `GAUSS_KDIMS` steps efficiently track curvature,
- `PULL_TO_BEST` consolidates progress without overshooting,

[2] We do not assume canonical identities for `f1`–`f24`; the discussion is data-driven from our tables/plots.

- the cooling schedule rarely needs heavy tails to escape traps.

The tiny **Std** suggests robustness to seed variation: multiple program structures are similarly effective.

**(B) Mixed but solvable: low Best, non-zero Average; mild to moderate dispersion.** Several functions yield **Best** $\approx 0$ yet non-zero medians (and modest **Std**), especially at higher $D$:

- `f20`: medians rise with $D$ (e.g., `f20_D10` $\approx 1.20 \times 10^{-3}$, `f20_D50` $\approx 1.16 \times 10^{-2}$). Replay curves show good early progress but slower late-stage improvements. Interpretation: *moderately coupled* coordinates or gentle ridges—sparse moves help, but exploitation struggles to reduce the last order of magnitude without finer step control.
- `f24`: similarly exhibits increasing medians with dimension (`f24_D10` $\approx 9.98 \times 10^{-4}$ vs. `f24_D50` $\approx 5.98 \times 10^{-2}$). Replay often shows one or two decisive drops mid-run (temperature range where heavier proposals pass) followed by long plateaus. Interpretation: *multimodal with benign basins*, where a lucky heavy-tailed jump (or a reset/opposition blend) finds the right valley; otherwise, programs plateau slightly above optimum.
- `f23`: medians go from $\approx 6.87 \times 10^{-3}$ (`D=10`) to $\approx 3.32 \times 10^{-1}$ (`D=50`), with **Std** increasing too. Interpretation: *more rugged or scaled* as $D$ grows; sparse steps and opposition help but the fixed budget limits the number of successful basin transitions.

Across this group, the operator pattern that succeeds most often is *sparse exploration early* (`GAUSS_KDIMS`, occasional `RESET_COORD`), *temperature-gated heavy tails* (`CAUCHY_FULL` triggered while $T$ is moderate), and *blend/pull refinement* late (`OPP_BLEND`, `PULL_TO_BEST`). Seeds that fail to assemble this mix typically get stuck one order of magnitude above the best outcomes.

**(C) High variance or outlier-prone: rugged/scale-sensitive.** A few functions display substantial dispersion, especially at large $D$:

- `f21_D50` shows an extreme **Std** (on the order of $10^3$) with median $\approx 7.58$ and mean in the hundreds, while the **Best** is $\approx 1.59 \times 10^{-1}$. This tells us two things: (i) the objective scale is very large (absolute values dwarf small relative improvements), and (ii) some evolved programs occasionally make large uphill proposals that are accepted at moderate $T$, producing long tails in the outcome distribution. Replay curves typically show long plateaus with occasional big moves; when the heavy-tailed branch hits a productive basin, the curve drops sharply; otherwise, it wanders at high values.
- `f22_D30, D50` and `f13_D30, D50` present high **Std** with medians reported as $\approx 0$. This indicates *bimodal* performance: many seeds hit near-zero, while others plateau much higher, inflating the standard deviation. The replay overlays confirm dual behaviours: a subset of programs dive rapidly; the rest hover above a threshold, waiting for a rare acceptance on a heavy move that may not materialize within the budget.

For such cases, *either* increasing $B_{\text{prog}}$ (more faithful program evaluation during GP) *or* slightly larger $N_{\text{pop}}/G$ (greater program diversity) would increase the fraction of seeds discovering the "good" structure; a tiny increase in `CAUCHY_FULL` usage (or temperature thresholds in IF) often helps.

**(D) A reliably solved special case at all $D$: `f17`.** `f17` exhibits very consistent negative optima across dimensions (e.g., medians around $-78.33$ for D=10 and around $-78.30$ for D=50) with very small dispersion. This consistency suggests a landscape with a *distinctive global basin* and relatively benign local structure: our operator set quickly enters and refines within that basin. Replay curves for `f17` do not show violent late moves; instead, steady improvements converge to nearly identical final values across seeds.

## 4.6 What the replay curves add beyond the table

The tables aggregate "where runs ended up"; the replay figures explain "*how* they got there." In particular:

1. **Temperature-gated branching works as intended.** On easy tasks the IF on `TEMP_GT` quickly routes to exploitation; on mixed/hard tasks the same gate keeps heavy perturbations active long enough to escape shallow basins.
2. **Sparse steps matter in high dimensions.** As $D$ increases, successful trajectories feature more `GAUSS_KDIMS` (with $k \in [2, 5]$), letting the program make progress without fighting the curse of dimensionality each step.
3. **Heavy tails are the "plateau breaker."** The discrete drops after long flats are almost always tied to a `CAUCHY_FULL` acceptance (often inside an IF governed by either `TEMP_GT` or `RAND_LT`). Removing or underweighting `CAUCHY_FULL` would degrade results specifically on rugged functions.

## 4.7 Computational Cost and Efficiency

**Cost model.** Evolution costs roughly $(G{+}1)\, N_{\text{pop}}\, B_{\text{prog}}$ evaluations; final application costs $B_{\text{final}}$. Wall-time scales accordingly and rises with dimension and with program depth (more `APPLY` calls per outer step). In our summary, easy problems often finish around 45–70 s per run, while harder cases (e.g., `f23_D50`) average $\sim 127$ s, and especially noisy/scale-heavy cases (e.g., `f21_D50`) exhibit large runtime spreads due to long plateau phases and frequent rejections.

**Is the cost justified?** Given the quality gap versus selection-only SPHH on most functions, the additional compute of GPHH is justified when final solution accuracy matters. Replay-only convergence (no evolution) is much cheaper, making it a practical reporting tool and a fast way to validate learned programs.

## 4.8 Runtime interpretation

*Mean Runtime (s)* rises with dimension and with functions that require more arithmetic per evaluation, but the primary determinant is the budget accounting: evolution costs roughly $(G{+}1)N_{\text{pop}}B_{\text{prog}}$, while the final application costs $B_{\text{final}}$. Variability across objectives also reflects learned program structure: deeper SEQs and REPEATs with many APPLY calls per outer step cost more wall time per accepted move. Notably, the replay-only plots run much faster than full evolution and are thus practical for illustrating convergence without re-training.

## 4.9 Failure Cases & Limitations

**High- variance outliers.** `f21_D50` is the clearest: median modest ($\sim 7.6$) but mean and Std extremely large; **Best** small. Interpretation: the combination of landscape scale and ruggedness means some programs wander at high values; occasional uphill accepts at moderate temperature inflate variance. Remedies: slightly larger $N_{\text{pop}}$ or $G$; increase $B_{\text{prog}}$ to score programs more faithfully; keep `CAUCHY_FULL` admissible longer by relaxing TEMP thresholds.

**Dimension-driven difficulty.** `f20`, `f23`, `f24` all show rising medians with $D$. Remedies: bias toward `GAUSS_KDIMS` with adaptive $k$ (e.g., $k \propto \sqrt{D}$ capped), add success-based step-size adaptation (shrink $\sigma$ on failures, expand on successes), and increase the probability of occasional `RESET_COORD` at late stages to avoid slow creep.

**Bimodal outcomes.** Some functions with Best $\approx 0$ but non-zero medians (e.g., `f22` at multiple dimensions) reveal that the correct recipe exists but is not *reliably* discovered. Remedies: modestly increase $N_{\text{pop}}$ or $G$; add a weak structural prior (force one TEMP-gated heavy-tail branch) to reduce dependence on lucky mutations.

## 4.10 Actionable takeaways (why the GPHH behaves this way)

- **Operator balance explains the easy set.** When landscapes are smooth or separable, the evolved programs converge rapidly with almost any reasonable composition of `GAUSS/PULL_TO_BEST`; hence near-zero medians and tiny **Std**.
- **Seed sensitivity on the mixed set is structural.** Good programs combine: (i) early sparse exploration, (ii) a `CAUCHY_FULL` escape path while $T$ is moderate, and (iii) late `OPP_BLEND/PULL_TO_BEST`. Seeds that miss (ii) tend to stall above optimum, explaining gaps between **Best** and **Average**.
- **High-variance cases demand heavier or longer exploration.** For rugged or scale-heavy functions (`f21`, `f23`, `f24` at D=50), slightly more budget per-program ($B_{\text{prog}}$) or a modest increase in $N_{\text{pop}}/G$ improves the chance of discovering the high-performing branches seen in the **Best** column. Alternatively, relaxing IF thresholds so heavy tails persist longer can reduce plateaus.

**Summary.** The GPHH attains near-optimal performance on a large subset of functions across all dimensions with low variability; on the remainder, it *can* find very strong programs (as **Best** indicates) but needs either (i) more evolutionary diversity/budget or (ii) slightly more emphasis on heavy-tailed and sparse moves to achieve those results reliably across seeds. The replay figures concretely demonstrate these dynamics: fast early improvements driven by Gaussian/sparse steps, late plateau breaking via Cauchy jumps, and increasing reliance on coordinate-selective moves as dimension grows.

# 5 Runtimes

Note: See runtime results in Table 2.

This section explains *why* the observed wall–times look the way they do, and how the GPHH's cost scales with dimensions, meta–parameters, program structure, and objective function complexity.

## 5.1 Where the time goes

For a single training run on one objective, the total number of objective evaluations is well approximated by

$$E_{\text{total}} \;\approx\; (G+1)\, N_{\text{pop}}\, B_{\text{prog}} \;+\; B_{\text{final}},$$

where $G$ is the number of GP generations, $N_{\text{pop}}$ the population size, $B_{\text{prog}}$ the per–program evaluation budget used to score each candidate program during evolution, and $B_{\text{final}}$ the budget used to apply the best evolved program after training. The wall–time is then

$$t_{\text{run}} \;\approx\; E_{\text{total}} \cdot \bar{t}_f \;+\; t_{\text{over}},$$

with $\bar{t}_f$ the *mean* time of a single objective call and $t_{\text{over}}$ the Python/GP overhead (program interpretation, RNG, branching, etc.). In our implementation, $t_{\text{over}}$ is small relative to $E_{\text{total}}\bar{t}_f$; objective calls dominate.

**Concrete scale** For the common setting `--pop 30 --gens 10 --per-prog 2000 --evals 300000`, we have

$$E_{\text{total}} = (10{+}1) \cdot 30 \cdot 2000 + 300{,}000 = 960{,}000 \text{ evaluations.}$$

One of the logged runs (e.g., `f24_D10`) completed in $\sim 199\,\text{s}$, implying an average $\bar{t}_f \approx 199/3.98{\times}10^6 \approx 5.0 \times 10^{-5}\,\text{s}$ per evaluation (about $50\,\mu\text{s}$). Functions with heavier math (e.g., many cos, exp, or reductions over $D$) will have larger $\bar{t}_f$.

## 5.2 Why different objectives have different runtimes

1. **Per–evaluation cost varies by function.** Separable/quadratic-like functions (mostly sums of squares) are fast ($\bar{t}_f$ low); oscillatory or exponential functions are slower due to transcendental calls and cache effects. Even with identical $E_{\text{total}}$, a slower $\bar{t}_f$ makes wall–time larger.

2. **Dimension $D$ increases cost per evaluation.** Many benchmarks are $O(D)$ per call (sums over coordinates), so $\bar{t}_f$ grows roughly linearly with $D$; functions with internal matrix ops can be $O(D^2)$ (not typical here, but possible). This explains the systematic runtime rise from D=10 $\to$ 30 $\to$ 50 in your summary.

3. **Program shape adds small overhead, not evaluations.** During evolution and final application, we spend exactly $B_{\text{prog}}$ (per scored program) and $B_{\text{final}}$ evaluations, regardless of how many `APPLY`s live inside the evolved tree—each `APPLY` proposes *one* new candidate and incurs *one* objective call. Deeper `SEQ`/`REPEAT` structures therefore do not change $E_{\text{total}}$, but they slightly change $t_{\text{over}}$ (more Python work per outer step). In practice, objective cost dominates, so differences in program depth cause only modest wall–time variation across seeds.

## 5.3 Why runs of the *same* objective and settings still differ

Even when $E_{\text{total}}$ and $D$ are fixed, you may see $\pm$10–30% variation in wall–time across seeds:

- **Objective locality.** Caches/branch predictors behave differently along different search trajectories (especially on trigonometric functions), shifting $\bar{t}_f$ slightly.
- **Program structure.** Evolved trees differ in #nodes and branching; more `IF`/`REPEAT`/`SEQ` adds Python overhead per proposal. This affects $t_{\text{over}}$ (usually a small fraction), not $E_{\text{total}}$.
- **System noise.** Parallel processes, OS scheduling, and thermal throttling introduce small wall–time noise.

# 6 A comparison of the performance of SPHH vs GPHH

Table 3: SPHH vs GPHH across all functions and dimensions: average, best, std, and mean runtime (lower is better).

| Function | Dim | Runs(S) | Avg(S) | Best(S) | Std(S) | Time(S) | Runs(G) | Avg(G) | Best(G) | Std(G) | Time(G) |
|---|---|---|---|---|---|---|---|---|---|---|---|
| f1 | 2 | 10 | 0.000000 | 0.000000 | 0.000000 | 0.22 | 10 | 0.000000 | 0.000000 | 0.000000 | 24.90 |
| f2 | 2 | 10 | -1.031615 | -1.031627 | 1.486e-05 | 0.25 | 10 | -1.031627 | -1.031628 | 3.546e-06 | 39.34 |
| f3_D10 | 10 | 10 | 0.000000 | 0.000000 | 0.000000 | 0.26 | 10 | 0.000000 | 0.000000 | 0.000000 | 47.26 |
| f3_D30 | 30 | 10 | 0.000000 | 0.000000 | 0.000000 | 0.25 | 10 | 0.000000 | 0.000000 | 0.000000 | 53.48 |
| f3_D50 | 50 | 10 | 0.000000 | 0.000000 | 0.000000 | 0.27 | 10 | 0.000000 | 0.000000 | 0.000000 | 48.93 |
| f4_D10 | 10 | 10 | 0.000000 | 0.000000 | 0.000000 | 0.24 | 10 | 0.000000 | 0.000000 | 0.000000 | 52.25 |
| f4_D30 | 30 | 10 | 0.000000 | 0.000000 | 0.000000 | 0.25 | 10 | 0.000000 | 0.000000 | 0.000000 | 51.78 |
| f4_D50 | 50 | 10 | 0.000000 | 0.000000 | 0.000000 | 0.25 | 10 | 4.941e-324 | 0.000000 | 0.000000 | 51.45 |
| f5_D10 | 10 | 10 | 0.000000 | 0.000000 | 0.000000 | 0.29 | 10 | 0.000000 | 0.000000 | 0.000000 | 45.10 |
| f5_D30 | 30 | 10 | 0.000000 | 0.000000 | 0.000000 | 0.28 | 10 | 0.000000 | 0.000000 | 0.000000 | 54.05 |
| f5_D50 | 50 | 10 | 0.000000 | 0.000000 | 0.000000 | 0.27 | 10 | 0.000000 | 0.000000 | 0.000000 | 61.36 |
| f6_D10 | 10 | 10 | 0.000000 | 0.000000 | 0.000000 | 0.24 | 10 | 0.000000 | 0.000000 | 0.000000 | 48.46 |
| f6_D30 | 30 | 10 | 0.000000 | 0.000000 | 0.000000 | 0.27 | 10 | 0.000000 | 0.000000 | 0.000000 | 54.17 |
| f6_D50 | 50 | 10 | 0.000000 | 0.000000 | 0.000000 | 0.26 | 10 | 0.000000 | 0.000000 | 0.000000 | 53.34 |
| f7_D10 | 10 | 10 | 0.000000 | 0.000000 | 0.000000 | 0.42 | 10 | 0.000000 | 0.000000 | 0.000000 | 49.19 |
| f7_D30 | 30 | 10 | 0.000000 | 0.000000 | 0.000000 | 0.42 | 10 | 0.000000 | 0.000000 | 0.000000 | 45.27 |
| f7_D50 | 50 | 10 | 0.000000 | 0.000000 | 0.000000 | 0.42 | 10 | 0.000000 | 0.000000 | 0.000000 | 55.73 |
| f8_D10 | 10 | 10 | 0.000000 | 0.000000 | 0.000000 | 0.47 | 10 | 0.000000 | 0.000000 | 0.000000 | 55.81 |
| f8_D30 | 30 | 10 | 0.000000 | 0.000000 | 0.000000 | 0.36 | 10 | 0.000000 | 0.000000 | 0.000000 | 53.53 |
| f8_D50 | 50 | 10 | 0.000000 | 0.000000 | 0.000000 | 0.32 | 10 | 0.000000 | 0.000000 | 0.000000 | 57.62 |
| f9_D10 | 10 | 10 | 0.000000 | 0.000000 | 0.000000 | 0.28 | 10 | 0.000000 | 0.000000 | 0.000000 | 41.04 |
| f9_D30 | 30 | 10 | 0.000000 | 0.000000 | 0.000000 | 0.33 | 10 | 0.000000 | 0.000000 | 0.000000 | 44.50 |
| f9_D50 | 50 | 10 | 0.000000 | 0.000000 | 0.000000 | 0.42 | 10 | 0.000000 | 0.000000 | 0.000000 | 52.44 |
| f10_D10 | 10 | 10 | 0.000000 | 0.000000 | 0.000000 | 0.35 | 10 | 0.000000 | 0.000000 | 0.000000 | 72.25 |
| f10_D30 | 30 | 10 | 0.000000 | 0.000000 | 0.000000 | 0.32 | 10 | 0.000000 | 0.000000 | 0.000000 | 81.09 |
| f10_D50 | 50 | 10 | 0.000000 | 0.000000 | 0.000000 | 0.33 | 10 | 0.000000 | 0.000000 | 0.000000 | 72.95 |
| f11_D10 | 10 | 10 | 0.000000 | 0.000000 | 0.000000 | 0.26 | 10 | 0.000000 | 0.000000 | 0.000000 | 47.72 |
| f11_D30 | 30 | 10 | 0.000000 | 0.000000 | 0.000000 | 0.26 | 10 | 0.000000 | 0.000000 | 0.000000 | 60.00 |
| f11_D50 | 50 | 10 | 0.000000 | 0.000000 | 0.000000 | 0.26 | 10 | 0.000000 | 0.000000 | 0.000000 | 62.27 |
| f12_D10 | 10 | 10 | 0.000305 | 1.673e-05 | 0.000363 | 0.25 | 10 | 4.463e-06 | 5.976e-07 | 2.994e-06 | 49.81 |
| f12_D30 | 30 | 10 | 0.000289 | 8.535e-05 | 0.000219 | 0.26 | 10 | 1.644e-05 | 3.142e-07 | 2.625e-05 | 54.88 |
| f12_D50 | 50 | 10 | 0.000301 | 2.709e-05 | 0.000291 | 0.27 | 10 | 4.821e-06 | 1.555e-07 | 5.298e-06 | 55.38 |
| f13_D10 | 10 | 10 | 0.000000 | 0.000000 | 0.000000 | 0.28 | 10 | 0.000000 | 0.000000 | 0.000000 | 57.95 |
| f13_D30 | 30 | 10 | 0.000000 | 0.000000 | 0.000000 | 0.28 | 10 | 3.708827 | 0.000000 | 11.728342 | 63.37 |
| f13_D50 | 50 | 10 | 0.000000 | 0.000000 | 0.000000 | 0.27 | 10 | 7.962204 | 0.000000 | 21.002011 | 65.58 |
| f14_D10 | 10 | 10 | 4.441e-16 | 4.441e-16 | 0.000000 | 0.32 | 10 | 4.441e-16 | 4.441e-16 | 0.000000 | 82.67 |
| f14_D30 | 30 | 10 | 4.441e-16 | 4.441e-16 | 0.000000 | 0.32 | 10 | 4.441e-16 | 4.441e-16 | 0.000000 | 76.06 |
| f14_D50 | 50 | 10 | 4.441e-16 | 4.441e-16 | 0.000000 | 0.33 | 10 | 4.441e-16 | 4.441e-16 | 0.000000 | 74.63 |
| f15_D10 | 10 | 10 | 0.000000 | 0.000000 | 0.000000 | 0.28 | 10 | 0.000000 | 0.000000 | 0.000000 | 67.61 |
| f15_D30 | 30 | 10 | 0.000000 | 0.000000 | 0.000000 | 0.29 | 10 | 0.000000 | 0.000000 | 0.000000 | 68.79 |
| f15_D50 | 50 | 10 | 0.000000 | 0.000000 | 0.000000 | 0.29 | 10 | 0.000000 | 0.000000 | 0.000000 | 70.76 |
| f16_D10 | 10 | 10 | 0.000000 | 0.000000 | 0.000000 | 0.25 | 10 | 0.000000 | 0.000000 | 0.000000 | 50.76 |
| f16_D30 | 30 | 10 | 0.000000 | 0.000000 | 0.000000 | 0.26 | 10 | 0.000000 | 0.000000 | 0.000000 | 54.49 |
| f16_D50 | 50 | 10 | 0.000000 | 0.000000 | 0.000000 | 0.25 | 10 | 0.000000 | 0.000000 | 0.000000 | 53.75 |
| f17_D10 | 10 | 10 | -78.267523 | -78.298817 | 0.032478 | 0.30 | 10 | -78.331324 | -78.331923 | 0.000635 | 67.33 |
| f17_D30 | 30 | 10 | -74.813730 | -77.338526 | 2.141884 | 0.30 | 10 | -78.310833 | -78.331650 | 0.014477 | 81.65 |
| f17_D50 | 50 | 10 | -69.682074 | -74.089388 | 2.458210 | 0.31 | 10 | -78.294685 | -78.326929 | 0.029299 | 82.78 |
| f18_D10 | 10 | 10 | 0.000000 | 0.000000 | 0.000000 | 0.25 | 10 | 0.000000 | 0.000000 | 0.000000 | 56.94 |
| f18_D30 | 30 | 10 | 0.000000 | 0.000000 | 0.000000 | 0.25 | 10 | 0.000000 | 0.000000 | 0.000000 | 52.84 |
| f18_D50 | 50 | 10 | 0.000000 | 0.000000 | 0.000000 | 0.26 | 10 | 0.000000 | 0.000000 | 0.000000 | 47.84 |

Continued on next page

Table 3: SPHH vs GPHH across all functions and dimensions: average, best, std, and mean runtime (lower is better).

| Function | Dim | Runs(S) | Avg(S) | Best(S) | Std(S) | Time(S) | Runs(G) | Avg(G) | Best(G) | Std(G) | Time(G) |
|---|---|---|---|---|---|---|---|---|---|---|---|
| f19_D10 | 10 | 10 | 0.000000 | 0.000000 | 0.000000 | 0.32 | 10 | 0.000000 | 0.000000 | 0.000000 | 83.54 |
| f19_D30 | 30 | 10 | 0.000000 | 0.000000 | 0.000000 | 0.32 | 10 | 3.477342 | 0.000000 | 7.950629 | 83.94 |
| f19_D50 | 50 | 10 | 0.000000 | 0.000000 | 0.000000 | 0.32 | 10 | 0.000000 | 0.000000 | 0.000000 | 85.17 |
| f20_D10 | 10 | 10 | 1.431e-12 | 7.589e-13 | 4.229e-13 | 0.43 | 10 | 0.001533 | 0.000291 | 0.001068 | 107.90 |
| f20_D30 | 30 | 10 | 5.373e-08 | 4.225e-09 | 6.697e-08 | 0.44 | 10 | 0.005873 | 0.002129 | 0.004838 | 121.47 |
| f20_D50 | 50 | 10 | 3.441e-05 | 7.841e-06 | 3.314e-05 | 0.45 | 10 | 0.013480 | 0.003510 | 0.008916 | 93.02 |
| f21_D10 | 10 | 10 | 81.138700 | 1.072e-05 | 94.481354 | 0.26 | 10 | 0.060540 | 0.009246 | 0.046171 | 56.04 |
| f21_D30 | 30 | 10 | 2108.332268 | 1461.327398 | 694.970907 | 0.27 | 10 | 1.395195 | 0.176617 | 1.805612 | 65.39 |
| f21_D50 | 50 | 10 | 6812.289247 | 5638.140674 | 733.639170 | 0.27 | 10 | 765.488165 | 0.158724 | 1296.498923 | 70.94 |
| f22_D10 | 10 | 10 | 0.000000 | 0.000000 | 0.000000 | 0.34 | 10 | 5.249287 | 0.000000 | 13.310471 | 90.55 |
| f22_D30 | 30 | 10 | 0.000000 | 0.000000 | 0.000000 | 0.35 | 10 | 5.512403 | 0.000000 | 13.058744 | 81.81 |
| f22_D50 | 50 | 10 | 0.000000 | 0.000000 | 0.000000 | 0.35 | 10 | 13.530667 | 0.000000 | 30.079491 | 86.67 |
| f23_D10 | 10 | 10 | 1.856e-12 | 1.256e-12 | 7.918e-13 | 0.42 | 10 | 0.007723 | 0.000808 | 0.004883 | 116.92 |
| f23_D30 | 30 | 10 | 2.614960 | 1.392396 | 0.473420 | 0.43 | 10 | 0.188196 | 0.009367 | 0.325073 | 120.64 |
| f23_D50 | 50 | 10 | 4.889894 | 4.786722 | 0.043899 | 0.44 | 10 | 0.364180 | 0.071865 | 0.285189 | 127.23 |
| f24_D10 | 10 | 10 | 5.280e-12 | 3.094e-12 | 1.382e-12 | 0.34 | 10 | 0.001279 | 0.000450 | 0.001087 | 89.37 |
| f24_D30 | 30 | 10 | 0.058086 | 6.828e-07 | 0.061089 | 0.35 | 10 | 0.035768 | 0.005622 | 0.033256 | 58.55 |
| f24_D50 | 50 | 10 | 0.447854 | 0.102836 | 0.171611 | 0.35 | 10 | 0.081826 | 0.026338 | 0.058252 | 55.25 |

## 6.1 Deep Comparison: SPHH vs. GPHH

**Setup** We compare a Selection Perturbative Hyper-Heuristic (SPHH) against a Generative Perturbative Hyper-Heuristic (GPHH) on the Wang–Song ($f_1 \dots f_{24}$) suite: base 2D for $f_1$–$f_2$ and $D \in \{10, 30, 50\}$ for the scalable functions. Each was run for 10 seeds per objective.

**Headline findings** Across **68** problem instances (objectives×dimensions):

- **Ties dominate**: both reach the same mean best fitness in **43/68** cases (**63.2**%).
- **When they differ**: **GPHH wins 14** instances (20.6%), while **SPHH wins 11** (16.2%).
- **Exact solves (mean ≈ 0)**: **SPHH solves 52/68** (76.5%) vs. **GPHH 43/68** (63.2%).
- **Runtime**: GPHH is far slower in wall-clock. Median per-objective *GPHH/SPHH* runtime ratio is **214.5**× (mean 211.8×).

Table 4: Overall outcomes across 68 instances (lower fitness is better).

| Outcome | Count | Percent |
|---|---|---|
| SPHH wins | 11 | 16.2% |
| GPHH wins | 14 | 20.6% |
| Ties | 43 | 63.2% |
| Solved by SPHH (mean≈ 0) | 52 | 76.5% |
| Solved by GPHH (mean≈ 0) | 43 | 63.2% |

**Runtime by dimension** Median per-run times (over seeds) and ratios:

Table 5: Runtime by dimension (per-run medians).

| Dim | SPHH median (s) | GPHH median (s) | GPHH/SPHH |
|---|---|---|---|
| 2 | 0.235 | 32.122 | 135.7× |
| 10 | 0.284 | 56.487 | 212.9× |
| 30 | 0.295 | 59.276 | 221.2× |
| 50 | 0.300 | 61.816 | 218.6× |

Table 6: Win/tie counts by dimension.

| Dim | SPHH | GPHH | Tie |
|---|---|---|---|
| 2 | 0 | 1 | 1 |
| 10 | **4** | 3 | 15 |
| 30 | 4 | **5** | 13 |
| 50 | 3 | **5** | 14 |

**By dimension (who wins when not tied)** At $D = 10$ SPHH edges GPHH; at $D \in \{30, 50\}$ GPHH leads slightly. Ties remain most frequent at all $D$.

**Where each method shines (by function family)** We group objectives into: *Unimodal/Separable* (e.g., Sphere, Schwefel 2.21/2.22, Elliptic, Step, Weighted Sphere/Quartic), *Multimodal* (Rastrigin, Ackley, Griewank, Schaffer-like,...), *Composite/Noise/Penalized* (Styblinski–Tang, Quartic+noise, Penalized #1/#2 variants), and *Other*.

Table 7: Win/tie counts by family.

| Class | SPHH | GPHH | Tie |
|---|---|---|---|
| Unimodal/Separable | 0 | 0 | **24** |
| Multimodal | **6** | 3 | 15 |
| Composite/Noise/Penalized | 5 | **10** | 0 |
| Other | 0 | 1 | 4 |

**Takeaways.**

- On *unimodal/separable* landscapes, both reach the optimum (24/24 ties).
- On *multimodal* families (Rastrigin/Ackley/Griewank/Schaffer), **SPHH** more often attains the best mean (6–3, with many ties).
- On *composite/penalized/noisy* objectives (Styblinski–Tang, penalized variants), **GPHH** leads (10–5), suggesting its evolved control-flow can better manage penalties/anisotropy.

**Largest gaps (mean best fitness)** Lower is better. Top gaps where one method clearly dominates:

**Top SPHH wins**

- `f22_D50` ($D$=50): SPHH mean 0, GPHH mean 13.5307, $\Delta$=13.5307.
- `f13_D50` ($D$=50): SPHH mean 0, GPHH mean 7.9622, $\Delta$=7.9622.
- `f22_D30` ($D$=30): SPHH mean 0, GPHH mean 5.5124, $\Delta$=5.5124.
- `f22_D10` ($D$=10): SPHH mean 0, GPHH mean 5.2493, $\Delta$=5.2493.
- `f13_D30` ($D$=30): SPHH mean 0, GPHH mean 3.7088, $\Delta$=3.7088.

**Top GPHH wins**

- `f21_D50` ($D$=50): SPHH mean 6812.2893, GPHH mean 765.4882, $\Delta$=−6046.8011.
- `f21_D30` ($D$=30): SPHH mean 2108.3323, GPHH mean 1.3952, $\Delta$=−2106.9371.
- `f21_D10` ($D$=10): SPHH mean 81.1387, GPHH mean 0.06054, $\Delta$=−81.0782.
- `f17_D50` ($D$=50): SPHH mean −69.6821, GPHH mean −78.2947, $\Delta$=−8.6126.
- `f23_D50` ($D$=50): SPHH mean 4.8899, GPHH mean 0.3642, $\Delta$=−4.5257.

**Why this pattern? (Qualitative)**

1. **Search bias vs. expressivity.** SPHH uses six curated perturbation operators with UCB selection, SA acceptance, and 1/5 step-size adaptation; this strong bias suffices for many multimodal cases, particularly Rastrigin-type families, often driving the mean to 0 even with small budgets.
2. **Evolved control-flow helps on penalized/anisotropic cases.** GPHH evolves short programs over the same primitives with `SEQ/REPEAT/IF` branches under an annealing-like schedule, enabling "macro-operators" that better traverse penalty basins or long narrow valleys (e.g., Schwefel 2.26, Penalized #1/#2).
3. **Efficiency trade-off.** In these runs SPHH is ≈0.25–0.30 s per objective; GPHH is ≈32–62 s (median), i.e., ∼215× slower. If wall-time or evaluation budget is limited, SPHH is the better default; if quality on penalized/ill-conditioned landscapes matters more than time, GPHH can pay off.

We compare the selection perturbative hyper-heuristic (SPHH) from Assignment 1 with the generation perturbative hyper-heuristic (GPHH) from this assignment across the same benchmark suite and dimensions. Below, we interpret the differences in terms of solution quality, stability, search dynamics, scalability, and computational cost.

## 6.2 Overall performance trends

**Quality of solutions.** Across the majority of functions and dimensions, GPHH attains *lower* final objective values than SPHH on both the **Average** and **Best** metrics. This advantage is most pronounced on rugged, multimodal, or scale-sensitive landscapes where SPHH often plateaus; GPHH, by evolving *compositions* of perturbation operators, continues to improve late in the run and reaches lower basins.

**Where SPHH keeps up.** On smooth or well-conditioned targets (empirically those where your GPHH tables show near-zero averages and tiny standard deviations), SPHH is often competitive: its restricted operator set plus greedy acceptance is already sufficient to descend rapidly, and additional generative complexity yields diminishing returns.

**Stability across seeds.** For easy/separable problems both methods exhibit very small variance; for difficult problems, SPHH typically shows *earlier plateaus* and a tighter (but higher) distribution, while GPHH shows *lower minima* and, on some functions, a wider spread—reflecting seed sensitivity in discovering a good program structure. The GPHH "spread" is a feature of *structural discovery*: some seeds evolve the right IF/SEQ/REPEAT composition (and win decisively), others evolve a weaker portfolio and stall higher.

## 6.3 Function characteristics and their influence

**Landscape complexity.** On moderately or strongly multimodal landscapes (e.g., functions that behave like Rastrigin/Schwefel families), GPHH's advantage is consistent. Evolved programs routinely combine `GAUSS_KDIMS` (for sparse, high-leverage moves), temperature-gated `CAUCHY_FULL` (plateau breaking), and late `PULL_TO_BEST`/`OPP_BLEND` refinement. SPHH, which selects from a fixed set without composing them into staged policies, tends to exploit early and then stagnate.

**Dimensionality.** Both methods degrade as $D$ increases, but GPHH degrades *more gracefully.* In higher dimensions, sparse moves are essential; the best GPHH programs learn to route most of their proposals through `GAUSS_KDIMS` (small $k$) and to interleave resets or opposition blends. SPHH does use such operators when selected, but without evolving conditional structure (e.g., temperature/acceptance gates) it is less able to maintain an exploration/exploitation balance as $D$ grows.

**Function-type sensitivity.** For valley-like/ill-conditioned shapes (Rosenbrock-like behaviour), SPHH often hovers above the optimum; GPHH's staged compositions (small-$\sigma$ Gaussians, pull-to-best, occasional resets) progress further along the curved valley. For oscillatory or deceptive functions, SPHH's local steps are frequently neutralised by frequent local minima; GPHH's heavy-tailed branch creates the "step improvements" visible in the replay curves.

## 6.4 Search dynamics: why GPHH wins (and when it does not)

**Generative advantage.** SPHH can *choose* a good operator but cannot *invent* a schedule. GPHH discovers schedules: early sparse exploration, conditional heavy-tail jumps while temperature is moderate, and late exploitation. This schedule is what breaks plateaus in rugged landscapes.

**Exploration vs. exploitation.** SPHH leans exploitative: once it finds a good local move it tends to keep selecting it, which promotes fast convergence *to a plateau.* GPHH programs embed `IF`{TEMP_GT/RAND_LT} gates that keep exploratory and heavy-tail moves admissible for longer, yielding both front-loaded improvements and late-stage jumps.

**Convergence patterns.** Empirically, SPHH curves drop rapidly then flatten early. GPHH curves drop rapidly, plateau, then exhibit one or more decisive downward "steps" triggered by the evolved Cauchy branch or reset/opposition sequences. This pattern explains the frequent gap between GPHH's **Average** and **Best**: the *mechanism* to improve late exists, but not all seeds find the exact thresholds/ordering.

## 6.5 Behaviour of the selected / generated heuristics

**SPHH motifs.** SPHH frequently settles on a small set of strong perturbations (e.g., isotropic Gaussian with tuned scale, or sparse Gaussian with fixed $k$) and cycles them; that yields interpretable, fast, but limited behaviour.

**GPHH motifs.** High-performing GPHH programs consistently include:

- **Sparse proposals** `GAUSS_KDIMS` (small $k$) to make progress in high dimensions.
- **Plateau breakers** `CAUCHY_FULL` gated by temperature or a random test.
- **Late refinement** via `PULL_TO_BEST` and occasional `OPP_BLEND`.
- **Light inner loops** (`REPEAT`) to micro-iterate the same subpolicy before reassessment.

These are *function-aware* in effect (more sparse moves at larger $D$, more heavy tails when improvement stalls), even though the operators are generic.

## 6.6 Generalization vs. overfitting

**Across dimensions of the same function.** GPHH strategies generalize within a function family: the same motif (sparse $\rightarrow$ heavy-tail $\rightarrow$ pull) recurs for D=10, 30, 50, with parameter scaling. SPHH's performance often drops more sharply with $D$ because it cannot *adapt its schedule*—only its current operator choice.

**Across different functions.** We observe recurring GPHH compositions on many distinct functions (not just on the one that produced them), suggesting they capture general search principles rather than brittle, overfit recipes. SPHH, while robust on easy tasks, lacks this ability to synthesize new policies for hard ones.

## 6.7 Statistical significance and robustness

We report for each method the **Average**, **Best**, and **Std** over at least ten seeds per objective. Where GPHH's Average is substantially lower than SPHH's and Std is not excessively larger, improvements are practically and statistically meaningful. For a formal test, pair the per-seed final best values of SPHH vs. GPHH on each function and apply a Wilcoxon signed-rank test; aggregate across functions with a Friedman/Nemenyi analysis to compare overall ranks. A quick "wins-by-dimension" summary (counting functions where GPHH's Average $<$ SPHH's Average, and likewise for Best) complements these tests.

## 6.8 Computational cost and efficiency

**Runtime trade-off.** SPHH typically completes in seconds per objective; GPHH requires minutes under the budgets used. The difference is expected: SPHH evaluates a fixed library; GPHH evaluates and *evolves programs* (trees) across generations. The cost is justified when lower final fitness is critical (rugged/high-$D$ tasks) and less justified when the problem is trivially solved by local steps (smooth/separable tasks).

**When to choose which.** If you need a fast approximate solution across many easy instances, SPHH is attractive. If you need robust performance on hard, deceptive, or high-dimensional problems, GPHH's late improvements and lower minima warrant the additional compute.

## 6.9 Failure cases and limitations

**SPHH limitations.** SPHH is prone to *premature exploitation*: once it finds a reasonable perturbation, the selection process keeps it, and the run stalls. It also lacks conditioned heavy tails, so escaping rugged basins late in the run is rare.

**GPHH limitations.** GPHH can be seed-sensitive on hard problems: not every seed rediscovers the "good" program topology; hence Average > Best gaps. Modest increases in population, generations, or per-program budget reduce this gap. Another practical limitation is cost: evolving programs is an order of magnitude slower.

## 6.10 Interpretability

**SPHH.** Highly interpretable: you can report which operator was most selected and with what scale; the behaviour is easy to explain but not rich.

**GPHH.** Moderately interpretable: the evolved program is a short policy tree. Its logic is human-readable (`IF TEMP_GT THEN ...ELSE ...`), and replay plots align with its phases (sparse exploration, heavy-tail jumps, late pull). Many of the discovered compositions are not obvious to hand-design yet appear repeatedly in strong solutions.

**Summary.** In head-to-head comparisons, GPHH delivers lower final objective values on most non-trivial functions and degrades more gracefully with dimension, at the cost of higher runtime and mildly higher seed variance on the hardest cases. SPHH remains effective on smooth/separable problems and is extremely fast, but its ceiling on complex landscapes is limited by the absence of generative composition. Practically, SPHH is a strong baseline and a good "first pass," while GPHH is the method of choice when solution quality on challenging instances is the priority.