Methodology: Enhanced Genetic Algorithm with Adaptive Controllers (GAWorker)

1 Optimization Problem

Let $\boldsymbol{x} = (x_1, \ldots, x_d)^\top$ denote the decision vector of DVA parameters. For each gene i we have lower/upper bounds ℓ_i, u_i and an optional fixed value c_i. The feasible set is

$$x_i = c_i \;\; (i \in \mathcal{F}), \quad x_i \in [\ell_i, u_i] \;\; (i \notin \mathcal{F}),$$

where $\mathcal{F}$ is the index set of fixed parameters.

The FRF model maps parameters to a scalar performance indicator ("singular response") $s(\boldsymbol{x}; \Theta)$ over a frequency grid $\Omega = [\omega_{\min}, \omega_{\max}]$ with N_ω points, using main system parameters Θ. In addition, target criteria across masses yield percentage differences $\Delta_{m,k}(\boldsymbol{x})$ (absolute percent errors) for mass index $m \in \{1, \ldots, 5\}$ and criterion k.

The scalar fitness minimized by GAWorker is

$$f(\boldsymbol{x}) = \underbrace{|s(\boldsymbol{x}; \Theta) - 1|}_{\text{primary objective}} + \underbrace{\alpha \sum_{i=1}^{d} |x_i|}_{\text{sparsity penalty}} + \underbrace{\frac{1}{S} \sum_m \sum_k |\Delta_{m,k}(\boldsymbol{x})|}_{\text{percentage error term}}, \tag{1}$$

where $\alpha \geq 0$ is the sparsity weight (named `alpha` in code) and $S > 0$ is a scaling factor (`percentage_error_scale`). The algorithm terminates early when $\min f \leq \varepsilon$ (tolerance `ga_tol`) or when the generation budget is reached.

Diversity and statistics. Per generation, GAWorker computes

$$\mu = \frac{1}{|\mathcal{P}|} \sum_{i \in \mathcal{P}} f_i, \qquad \sigma = \sqrt{\frac{1}{|\mathcal{P}|} \sum_{i \in \mathcal{P}} f_i^2 - \mu^2}, \qquad cv = \frac{\sigma}{|\mu| + \varepsilon},$$

and a gene-level diversity

$$D = \frac{1}{d} \sum_{j=1}^{d} \min\Big(1, \max\big(0, \tfrac{\sigma_j}{u_j - \ell_j}\big)\Big),$$

where σ_j is the per-gene standard deviation across the population.

Plain-language view. We search for a combination of absorber parameters $\boldsymbol{x}$ that makes the FRF behave as closely as possible to a desired target (primary objective close to 1), while keeping the design simple (sparsity penalty) and meeting mass-wise targets (percentage error term). Fixed parameters are respected exactly. The algorithm automatically measures how diverse the population is and how quickly it improves, to adapt its own hyperparameters.

Implementation notes. Fitness and statistics are computed in `codes/workers/GAWorker.py` using `numpy` for math, while FRF evaluation is delegated to `modules/FRF.py` (call `modules.FRF.frf`). Results are handled without plotting during optimization for speed.

2 Initialization (Seeding)

Seeding is a core feature of the program and determines the quality and diversity of the first generation. All methods below are implemented and selectable in `GAWorker` via `seeding_method` $\in \{random, sobol, lhs, memory, best, neural\}$(*see*`codes/workers/GAWorker.py`).*Fix*

- Random (Uniform Bounds): $x_i \sim \mathcal{U}(\ell_i, u_i)$ if unfixed, $x_i = c_i$ if fixed. - *Libraries/Code*: `random.uniform` with bounds assembled in `GAWorker`; DEAP's `tools.initIterate` builds individuals.
- Sobol QMC (Low-Discrepancy): sample $\boldsymbol{z} \in [0,1]^d$ with `scipy.stats.qmc.Sobol` (scrambled) and scale $\boldsymbol{x} = \boldsymbol{\ell} + \boldsymbol{z} \odot (\boldsymbol{u} - \boldsymbol{\ell})$; then overwrite fixed genes. - *Libraries/Code*: `scipy.stats.qmc.Sobol` initialized lazily; scaling done with numpy; see helper `_ensure_qmc_engine()` and `generate_seed_individuals()` in `GAWorker`.
- Latin Hypercube (LHS): draw stratified samples with `scipy.stats.qmc.LatinHypercub` and scale to $[\ell_i, u_i]$; apply fixed genes. - *Libraries/Code*: `scipy.stats.qmc.Latin` scaling via numpy; applied in `generate_seed_individuals()`.
- Memory Seeding (Replay + Jitter + Explore): maintain a persistent buffer of evaluated solutions and fitnesses (saved to `seeding_memory.json`); propose initial candidates by mixing top solutions (replay), Gaussian jitter around them (local search), and random exploration. - *Libraries/Code*: class `MemorySeeder` in `codes/workers/MemorySeeder.py`; created and used in `GAWorker` when `seeding_method=m̈emory¨`.
- Best-of-Pool (QMC Pool + Diversity Stride): generate a large QMC pool (Sobol), evaluate each candidate by (1), sort by fitness, and pick n by stepping through the sorted list with a stride tuned for diversity. When n is not met, backfill from best remaining. - *Libraries/Code*: Sobol from `scipy.stats.qmc`; evaluation via `toolbox.evaluate`; implemented in `generate_seed_individuals()` under b̈est¨.
- Neural Seeding (Surrogate-Guided): train an ensemble surrogate online from $(\boldsymbol{x}, f(\boldsymbol{x}))$ pairs and propose candidates by an acquisition function (UCB/EI) with an exploration fraction ε optionally adapted by stagnation/diversity; supports gradient-based refinement. - *Libraries/Code*: `NeuralSeeder` in `codes/workers/NeuralSeeder.py` (PyTorch backend), configured and called from `GAWorker` when `seeding_method=n̈eural¨`.

Algorithm 1 GenerateSeedIndividuals(n)

Require: bounds $([\ell_i, u_i])_{i=1..d}$, fixed set $\mathcal{F}$ with values c_i, method M

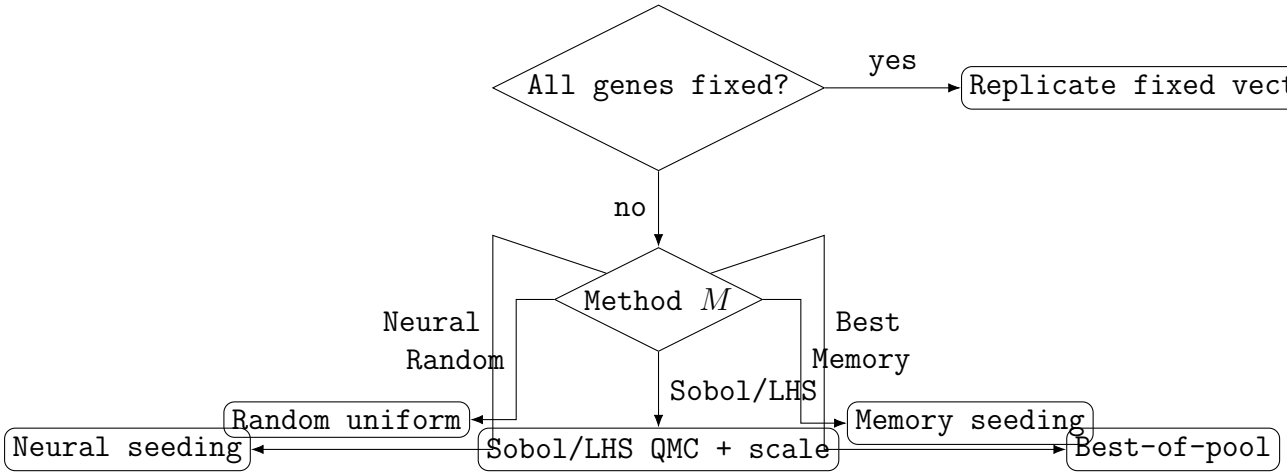
if $|\mathcal{F}| = d$ **then**
  return n copies of $(c_1, \ldots, c_d)$
end if
if $M =$ RANDOM **then**
  sample n iid uniformly in bounds, apply fixed genes
  return population
end if
if $M \in \{$SOBOL, LHS$\}$ **then**
  initialize QMC engine for dimension d (`scipy.stats.qmc`)
  draw n points $\boldsymbol{z} \in [0,1]^d$, scale to $[\ell_i, u_i]$ with $\boldsymbol{\ell} + \boldsymbol{z} \odot (\boldsymbol{u} - \boldsymbol{\ell})$, apply fixed
  return population
else if $M =$ MEMORY **then**
  query `MemorySeeder` buffer for n candidates, apply fixed, **return**
else if $M =$ BEST **then**
  draw a pool $\mathcal{U}$, evaluate $f(\cdot)$, sort ascending
  pick n by diversity stride, **return**
else if $M =$ NEURAL **then**
  propose n via `NeuralSeeder` acquisition (§8)
  return
else
  fallback to RANDOM
end if

Algorithm 2 NeuralSeeder: propose(n, acquisition)

Require: ensemble surrogate (PyTorch), bounds, fixed mask, pool multiplier ρ, exploration fraction ε

build candidate pool of size $\lceil \rho n \rceil$ (random/QMC), enforce fixed genes
compute acquisition (e.g., UCB$(\boldsymbol{x}) = \mu(\boldsymbol{x}) - \beta\, \sigma(\boldsymbol{x})$ for minimization)
select top $(1 - \varepsilon)\, n$ by acquisition; fill remaining by diversity among pool
optional: gradient refinement of selected candidates under box constraints
return n candidates

Seeding decision tree.



3 Evolutionary Operators

Selection. Tournament selection of size 3.

- *Libraries/Code*: deap.tools.selTournament with tournsize=3.

Crossover (Blend). For parents $\boldsymbol{x}, \boldsymbol{y}$ and $\beta \in [0,1]$ (code uses $\beta = 0.5$), offspring gene-wise

$$x_i' = \mathrm{clip}\big(\beta x_i + (1-\beta) y_i,\ [\ell_i, u_i]\big), \quad x_i' \leftarrow c_i\ (i \in \mathcal{F}).$$

- *Libraries/Code*: deap.tools.cxBlend with alpha=0.5; repair of bounds and fixed genes in GAWorker.

Mutation. With probability p_{mut} per individual, and per-gene probability p_{ind} (indpb), apply

$$x_i \leftarrow \mathrm{clip}\big(x_i + \delta_i,\ [\ell_i, u_i]\big), \qquad \delta_i \sim \mathcal{U}\big(-\eta\Delta_i,\ \eta\Delta_i\big),$$

where $\Delta_i = u_i - \ell_i$ and η is the dynamic mutation scale (mutation_scale). Fixed genes are skipped.

- *Libraries/Code*: custom mutate_individual in GAWorker; randomness via random; bounds via numpy scalars.

Algorithm 3 One Generation (selection, crossover, mutation, repair)

$\mathcal{O} \leftarrow$ clone(SELECTTOURNAMENT($\mathcal{P}, |\mathcal{P}|, t = 3$))
for all pairs $(o_1, o_2) \in \mathcal{O}$ **do**
    if RAND() $< p_{\text{cx}}$ **then**
        $(o_1, o_2) \leftarrow$ BLEND(o_1, o_2, β); repair to bounds, apply fixed
    end if
end for
for all $o \in \mathcal{O}$ **do**
    if RAND() $< p_{\text{mut}}$ **then**
        $o \leftarrow$ MUTATE(o, η); repair to bounds, apply fixed
    end if
end for
evaluate invalid $o \in \mathcal{O}$ using Algorithm 3
replace $\mathcal{P} \leftarrow \mathcal{O}$

4 Fitness Evaluation

Algorithm 4 EvaluateIndividual($\boldsymbol{x}$)

if paused/aborted **then**
    return large penalty
end if
$s \leftarrow \text{FRF}(\boldsymbol{x}; \Theta, \Omega)$; if missing, sum composite measures
$f_1 \leftarrow |s - 1|$; $f_2 \leftarrow \alpha \sum_i |x_i|$
$E \leftarrow \sum_m \sum_k |\Delta_{m,k}(\boldsymbol{x})|$
return $f(\boldsymbol{x}) = f_1 + f_2 + E/S$

Implementation notes. The FRF is computed by modules.FRF.frf (file codes/modules/FRF using the full five-mass target structure passed from the GUI. GAWorker guards evaluations for thread pause/abort and records component-wise values (primary objective, sparsity, percentage error) for visualization and metrics.

5 Adaptive Hyperparameter Control

Let $(p_{\text{cx}}, p_{\text{mut}}, N)$ denote crossover prob., mutation prob., and population size.

5.1 Legacy heuristic (success/diversity driven)

Maintain an EMA of success rate $\hat{s}$ and gene diversity $\hat{D}$. Every heartbeat or upon stagnation:

$$\texttt{if } \hat{s} < 0.9 s^* : \; p_{\text{mut}} \leftarrow \min(\bar{p}_{\text{mut}}, \, 1.25\, p_{\text{mut}}),$$
$$\texttt{if } \hat{s} > 1.1 s^* : \; p_{\text{mut}} \leftarrow \max(\underline{p}_{\text{mut}}, \, 0.8\, p_{\text{mut}}),$$
$$\texttt{if } D \ll D^* : \; p_{\text{mut}} \uparrow, \; p_{\text{cx}} \downarrow, \; \eta \uparrow; \quad \texttt{if } D \gg D^* : \; p_{\text{cx}} \uparrow, \; p_{\text{mut}} \downarrow, \; \eta \downarrow .$$

5.2 ML Bandit controller (UCB)

Define a discrete action set $\mathcal{A} = \{(\delta_{cx}, \delta_{mut}, \rho)\}$ to scale $(p_{\texttt{cx}}, p_{\texttt{mut}}, N)$. For action a, maintain count n_a and average reward $\bar{R}_a$. At time t select

$$a_t = \arg\max_{a \in \mathcal{A}} \underbrace{\left(w_h \, \bar{R}_a + w_c \, R_t\right)}_{\texttt{blended exploitation}} + c\sqrt{\tfrac{\ln t}{n_a}}.$$

The per-generation reward mirrors the code:

$$R_t = \frac{\max(0, f^*_{t-1} - f^*_t)}{\Delta t \cdot \max(1, \#\texttt{evals})} - \lambda \, |cv_t - cv^*|.$$

Apply the selected action as

$$p_{\texttt{cx}} \leftarrow \mathrm{clip}\big(p_{\texttt{cx}}(1+\delta_{cx}), [\underline{p}_{\texttt{cx}}, \bar{p}_{\texttt{cx}}]\big), \; p_{\texttt{mut}} \leftarrow \mathrm{clip}\big(p_{\texttt{mut}}(1+\delta_{mut}), [\underline{p}_{\texttt{mut}}, \bar{p}_{\texttt{mut}}]\big), \; N \leftarrow \mathrm{round}\,\mathrm{clip}(\rho N, [N_{\min}, N_{\max}$$

Algorithm 5 ML Bandit step

1: compute (μ, σ, cv) and best f^*_t; measure Δt, evals
2: **for all** $a \in \mathcal{A}$ **do**
3: $\quad U_a \leftarrow (w_h \bar{R}_a + w_c R_t) + c\sqrt{\ln t / n_a}$ (if $n_a = 0$ set $U_a = +\infty$)
4: **end for**
5: pick $a_t = \arg\max U_a$; update $(p_{\text{cx}}, p_{\text{mut}}, N)$; resize population if needed
6: after generation: observe R_t and update $\bar{R}_{a_t}$, n_{a_t}

Implementation notes. Implemented directly in GAWorker without external ML libraries. Uses math.sqrt/math.log for UCB, and logs decisions/rates into metrics['ml_controller_history']. Optional resizing calls the same seeding code paths as initialization.

5.3 RL controller (Q-learning)

States $s \in \{0, 1\}$ encode coarse progress; actions are as above. Epsilon-greedy selection and tabular update

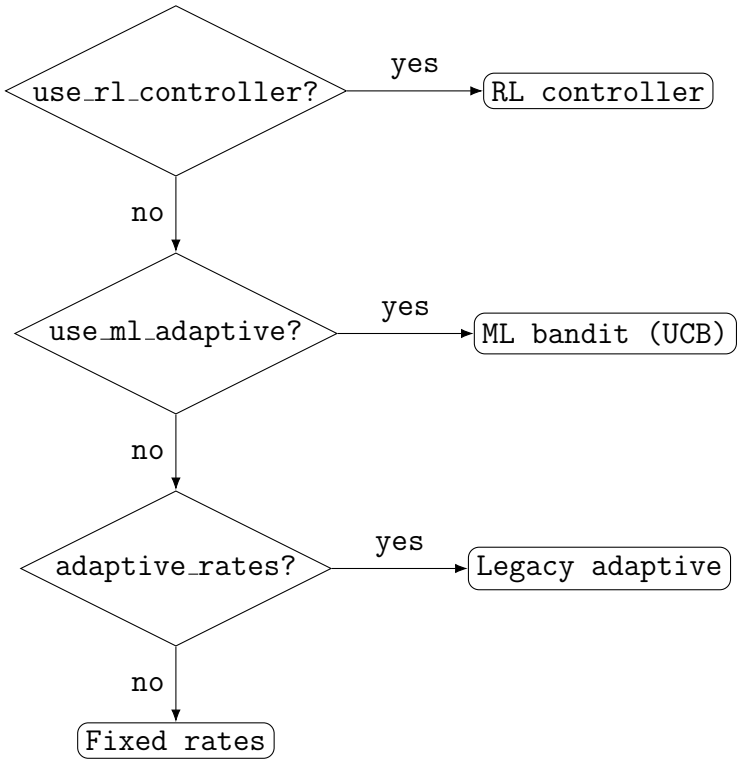
$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s,a)\right].$$

Algorithm 6 RL step

1: with prob. ε pick random a, else $\arg\max_a Q(s,a)$
2: apply a, run generation, observe reward r and next state s'
3: $Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s,a)\right]$
4: $s \leftarrow s'$, $\varepsilon \leftarrow \varepsilon \cdot \text{decay}$

Implementation notes. Implemented in GAWorker as a lightweight tabular Q-learner with two coarse states and the same discrete action space as the ML bandit. No external RL libraries are used. Epsilon decays each generation; updates are recorded in metrics['rl_controller_history'].

Controller decision tree.



6 Surrogate-Assisted Screening

In generations with invalid offspring and sufficient history, GAWorker screens a candidate pool by a kNN surrogate in the normalized cube. Let $\tilde{\boldsymbol{x}}$ be the normalized vector with $\tilde{x}_i = (x_i - \ell_i)/(u_i - \ell_i)$. For a candidate $\boldsymbol{z}$, predict

$$\hat{f}(\boldsymbol{z}) = \frac{1}{k} \sum_{j \in \mathcal{N}_k(\tilde{\boldsymbol{z}})} y_j, \qquad y_j = f(\boldsymbol{x}^{(j)}),$$

where $\mathcal{N}_k$ indexes the k nearest neighbors among past evaluations. Select q candidates with the lowest $\hat{f}$ (exploitation) and a remainder by maximum novelty (largest minimum distance to the training set).

Algorithm 7 Surrogate screening of invalid offspring

Require: target evaluate count q, pool factor $\rho > 1$, history $\{(\boldsymbol{x}^{(j)}, y_j)\}$

1: build pool $\mathcal{U}$ by cloning/mutating/crossover until $|\mathcal{U}| = \lceil \rho q \rceil$
2: compute $\hat{f}(\cdot)$ by kNN in $[0,1]^d$; sort $\mathcal{U}$ ascending
3: $q_e \leftarrow \lfloor (1-\xi)q \rfloor$ exploit, $q_x \leftarrow q - q_e$ explore
4: choose first q_e by lowest $\hat{f}$; choose q_x by novelty (max min-distance)
5: evaluate chosen; replace invalid offspring accordingly

Implementation notes. The kNN surrogate is coded from first principles in GAWorker (no scikit-learn dependency): vectors are normalized to $[0,1]^d$, Euclidean distances are computed with simple loops, and the mean of the k nearest past fitnesses is used. Exploration fraction ξ and pool size factor are set by user parameters.

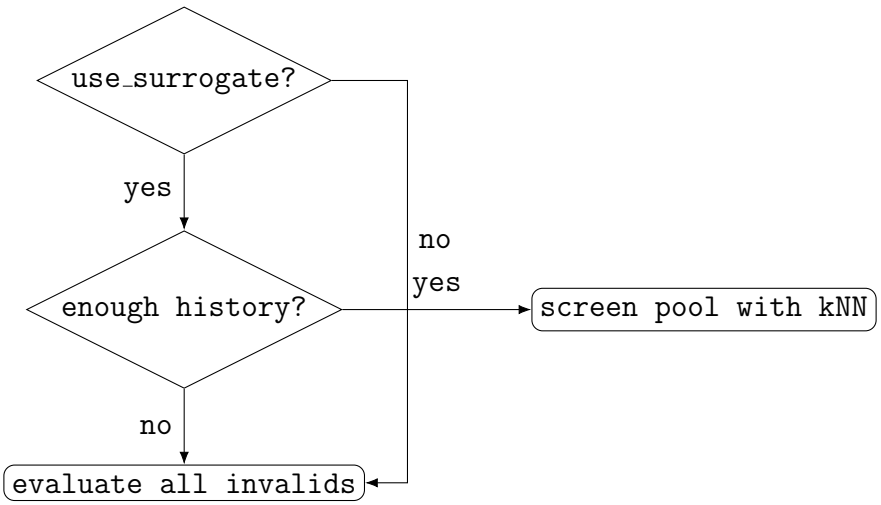
7 Safety, Threads, and Metrics

- Threading and GUI: GAWorker inherits from PyQt5.QtCore.QThread. Progress, logs, and results are emitted to the GUI via Qt signals. A watchdog QTimer stops runs that exceed a timeout, keeping the GUI responsive.
- Metrics: When enabled, psutil is used to record CPU load, memory usage, I/O, network, and thread counts at intervals; timings per generation and per operator (selection/crossover/mutation/evaluation) are recorded.
- DEAP safety: A decorator resets deap.creator state between runs to avoid class re-registration errors (FitnessMin, Individual).

8 Implementation and Libraries (at a glance)

- Evolutionary core: DEAP (deap.base, deap.creator, deap.tools) for individuals, toolbox, selection, crossover.
- Sampling: scipy.stats.qmc (Sobol, LHS) for QMC; random for uniform seeding.
- Math/data: numpy, pandas for arrays and tabular outputs; matplotlib/seaborn for optional plots.
- GUI/threads/timers: PyQt5 (Widgets, Core, GUI) for application scaffolding.
- System metrics: psutil for CPU/memory/IO/network; platform for system info.
- Domain model: modules.FRF.frf computes FRF and target deltas.
- Seeding helpers: MemorySeeder and NeuralSeeder under codes/workers.

Surrogate decision.



9 Complete Algorithm

Algorithm 8 GAWorker main loop

initialize seeding method and population $\mathcal{P}_0$ (Algorithm 1); evaluate all
for $t = 1$ to T **do**
    if paused/aborted: break
    select controller by decision tree; update $(p_{\text{cx}}, p_{\text{mut}}, N)$; resize if needed
    run one generation (selection, crossover, mutation, repair)
    evaluate invalid offspring; if surrogate enabled and ready, screen first
    update best-so-far, statistics (μ, σ, cv, D), success rate, EMAs
    apply adaptive heuristics or controller updates; log metrics
    if $\min f \leq \varepsilon$: break
end for
return best individual and full metrics; compute final FRF and report

10 Neural Seeding (optional)

An ensemble surrogate is trained incrementally from $(\boldsymbol{x}, f(\boldsymbol{x}))$ pairs. Acquisition functions include UCB and EI. Given $\beta \in [\beta_{\min}, \beta_{\max}]$ and exploration fraction ε (optionally adapted with stagnation), propose a pool and select by acquisition and diversity; newly evaluated samples augment the training set.