ChatGPT

Calibration and Optimization Strategy for 3-Bot MASBotS

To accurately calibrate the 3-bot Magnetically-Augmented Spinning Robot Swarm (MASBotS) model, we will follow an 8-part implementation roadmap. This covers data preparation, objective definition, optimization methods (with a human-in-loop option), visualization, validation, tool selection, MPC integration, and synthetic testing. Each phase is detailed below with action steps and best practices.

1. Extracting and Preprocessing Training Data

First, gather and process experimental data from the 30 fps videos to serve as ground truth for calibration [1]. We will use the per-frame 2D positions of each of the 3 bots (already obtained via tracking) and prepare them for comparison with simulation output:

- **Collect and Organize Raw Data:** Load the tracked positions for each frame and each bot (e.g. from CSV or MATLAB files). Ensure data is structured as time-series of coordinates for each bot (e.g. `positions[frame, bot_id] = (x, y)`). If tracking was done in pixel units, convert to real-world units (using a known scale) so simulation and data share units.
- **Synchronize Time and Inputs:** Note the duration and frame rate of the experiment. Prepare a time vector (e.g. 0 to T seconds at 1/30s intervals) corresponding to frames. Record or reconstruct the control inputs (spin rates) given to each bot over time – for example, if one bot was the "fast spinner" and others "slow," create a time-indexed array of spin speeds for each bot, matching what was done in the experiment. These inputs will be fed into the simulator for a fair comparison.
- **Filter and Clean Trajectories:** Smooth the position data to reduce high-frequency noise from tracking. For instance, apply a moving average or low-pass filter to each bot's X and Y sequence to eliminate jitter. This prevents the optimizer from chasing noise instead of true motion patterns. Keep the filtered data for calibration, but retain an unfiltered copy for reference in validation.
- **Compute Derived Metrics:** Optionally, calculate high-level cluster metrics from the data for additional calibration targets or sanity checks. For example, compute the center-of-mass trajectory of the three bots (the average of their coordinates at each frame) and the **cluster radius** over time (e.g. the root-mean-square distance of bots from the center-of-mass, or the convex hull radius). These metrics condense the swarm behavior (e.g. how tightly the bots cluster and whether the cluster drifts). We will later compare these metrics between real and simulated data.
- **Prepare Initial Conditions:** Determine the starting configuration in the experiment (e.g. the 2D coordinates of each bot in the first frame). This will be used to initialize the simulation state. Ensure the simulation starts from the same arrangement of bots as the experiment for a one-to-one trajectory comparison. If needed, adjust the reference frame (e.g. translate the coordinates so the initial center-of-mass is at the origin for both real and simulated data).
- **Segmentation (if applicable):** If the experiment has distinct phases or if multiple trial runs are available, segment the data accordingly. You might use one representative trial (or the first phase of a trial) for calibration, and reserve others for later validation. Consolidate the training dataset that

the optimizer will fit to – for instance, combine position time-series from a few similar runs if you want a calibration that generalizes across them.

By the end of this stage, you should have a *training dataset* consisting of time-indexed bot positions (and possibly cluster metrics) along with corresponding input signals (spin rates). This is the foundation for defining the calibration objective.

2. Defining the Objective Function(s)

Next, formulate a quantitative objective that measures how well the simulation outcomes match the real data. This objective (or "loss" function) will be minimized by adjusting the model parameters (alpha, beta, f0). We consider the following components for the error metric:

- **Trajectory Reproduction Error:** The primary objective is to minimize the discrepancy between simulated and real trajectories of each bot. A suitable metric is the root-mean-square error (RMSE) over time for all bot positions. For example, if real data has bot positions `p_real[i,t] = (x_real, y_real)` and simulated positions `p_sim[i,t]`, define:

$$E_{\text{traj}} = \sqrt{\frac{1}{N_{\text{bots}}\,N_t}\sum_{i=1}^{N_{\text{bots}}}\sum_{t=1}^{N_t} |(x_{sim}^{i,t} - x_{real}^{i,t},\; y_{sim}^{i,t} - y_{real}^{i,t})|^2}.$$

This yields the average position error across all time frames and bots. We can use the sum-of-squared errors as the function to minimize (since RMSE has a square root, sometimes sum of squares is easier for optimization). - **Cluster Metric Error:** To capture collective behavior, include terms for derived metrics like cluster radius and center-of-mass. For instance, compute the cluster radius from sim and real at each time (using the same definition as in data preprocessing) and calculate an RMSE for that as well. Similarly, compare the center-of-mass trajectories. These metrics ensure the optimizer accounts for both relative spacing (via radius) and overall motion (via COM drift) of the swarm. If the cluster radius is a critical behavior, its error can be weighted more heavily in the objective. - **Multi-Objective Combination:** Combine the above errors into a single scalar objective. A common approach is a weighted sum: **Objective =** `w1*E_traj + w2*E_radius + w3*E_COM` (for example). By adjusting weights `w1, w2, ...`, you emphasize certain aspects of the behavior. If trajectory RMSE alone is sufficient (since it inherently captures cluster behavior), you might simplify to just that. However, including cluster-level metrics can help guide the optimization if direct trajectory matching is too fine-grained or if, say, small rigid rotations of the entire formation would cause large trajectory error but no change in radius – in such cases a radius term provides rotation-invariant matching. - **Normalization and Scaling:** It's wise to normalize error terms so that none of them dominates due to units or magnitude differences. For example, if position coordinates are on the order of 100 (in some unit) while cluster radius is ~10, you might normalize each error by the variance of that metric in the data or use dimensionless percentages. This prevents the optimizer from focusing only on minimizing one part of the error. - **Regularization (if needed):** If certain parameter values are known to be physically plausible (e.g. alpha and beta should be positive, f0 in a certain range) or if you want to discourage extreme parameter values, you can add a mild regularization term. For example, a penalty for negative values or an L2 penalty pushing parameters toward an expected nominal value. With only three parameters and physical intuition, you may instead enforce bounds in the optimizer (discussed below) rather than adding to the objective function. - **Multiple Datasets:** If using multiple experimental runs for calibration, formulate the objective to aggregate errors across all those runs. For example, sum the RMSE errors for each run to get a total error that the optimizer will minimize. This way, the chosen parameters are

a best compromise that works for all provided scenarios (improving generality). Alternatively, if focusing on one scenario, you'll calibrate on that and later test on others.

In code, you will implement the objective as a function `f(alpha, beta, f0)` that runs the simulation for the given parameters (with the fixed initial conditions and inputs from the experiment), computes the differences to the real data (trajectory and any other metrics), and returns a single error value. This function is the core of the calibration – it encodes "how good" a given set of parameters is. It will be treated as a black-box by the optimizer (no analytical gradients needed), so it can be arbitrarily complex. Keep it deterministic (for a given input it should produce the same error) and reasonably fast, since it may be called many times. We now have a concrete objective to minimize: typically the trajectory RMSE or a similar measure of fit between sim and reality.

3. Optimization Loop and Modular Tuning Interface

With the data and objective in place, the next step is to automatically search for the parameter values (alpha, beta, f0) that minimize the objective. We will use an optimization loop that is **modular** (easily swapped algorithms or manual intervention) and that supports a **human-in-the-loop** approach for visualization and tweaking. Key design decisions and steps:

- **Choose an Optimization Algorithm:** For a robust calibration, a **derivative-free global optimizer** is recommended, since the simulation is non-linear and potentially non-convex in these parameters. Two good options are **CMA-ES** (Covariance Matrix Adaptation Evolution Strategy) and **PSO** (Particle Swarm Optimization), as mentioned. Both are stochastic global search methods that don't require gradient computations and can handle complex error landscapes. For instance, CMA-ES is known to perform well on difficult, non-convex black-box problems [2] and is invariant to parameter scaling [3], making it suitable for tuning multiple parameters simultaneously. PSO is another heuristic method that is simpler to implement and parallelize, searching by evolving a swarm of candidate solutions. There are readily available Python libraries for these (e.g. the `cma` library for CMA-ES, or `PySwarms` for PSO [4]). If you prefer a built-in solution, SciPy's `scipy.optimize` has global optimizers like `differential_evolution` (genetic algorithm) that could be used similarly.
- **Incorporate Parameter Bounds/Constraints:** Based on physical reasoning, set sensible bounds for each parameter (e.g. alpha, beta, f0 >= 0, or an upper bound if too high values would lead to unstable simulations). Many optimizers allow bound constraints (SciPy's L-BFGS-B, differential_evolution, and PSO all accept bounds). This keeps the search in a realistic range and speeds up convergence by avoiding extreme values. If using CMA-ES (which by itself doesn't handle bounds directly), you can enforce bounds by penalizing the objective outside bounds or by restarting if out-of-bounds proposals occur.
- **Initial Guess and Algorithm Settings:** Provide an initial parameter guess (perhaps from prior experience or a quick manual tune). For CMA-ES, you supply an initial mean and a initial standard deviation for the search distribution. For PSO, you might start with a random population within the bounds. Because our parameter space is only 3-dimensional, even a coarse initial guess is okay – the global methods will explore. Set the algorithm population size and iteration budget such that it can adequately search the space (for example, CMA-ES defaults might sample ~$\lambda=4+\lfloor3\ln(n)\rfloor$ candidates per generation, which is about 10 for n=3, and you might run for 50–100 generations as a start). Ensure these parameters are configurable.
- **Optimization Loop Structure:** Implement the optimization as a loop that can be **paused or interacted with**. For example, if coding from scratch or using a library that allows callbacks:

- Run a generation or a fixed number of iterations of the optimizer.
- After each generation (or every few), record the current best parameter set and its error. Invoke a callback function that can, for instance, plot intermediate results (more on visualization below) or check a pause flag.
- Allow the loop to terminate early if a satisfactory error threshold is reached, or if improvements stall (e.g. no significant error reduction for N iterations).
- Design the loop such that it can be cleanly stopped and restarted. For instance, you might run 20 iterations of CMA-ES, then inspect results, then decide to run 20 more. This could be done by storing the optimizer state (CMA-ES library lets you reinitialize from a previous mean and covariance) or simply re-running with a new initial guess near the current best.
- **Manual Tweaking Interface:** To facilitate manual intervention, make the system modular:
- **Parameter Configuration:** Keep the current best parameters in an accessible variable or file. This way, a user can manually edit these values (based on intuition or visual diagnostics) and re-run the simulation to see the effect.
- **Switchable Optimizer Modules:** Write the optimization routine in a way that you can swap the algorithm easily (e.g. have a flag for `method = "CMAES"` vs `"PSO"` vs `"LBFGSB"`). Encapsulate each method's invocation in a function. This allows testing different optimizers or strategies if one isn't performing well.
- **Partial Loop Runs:** Provide a mechanism (even as simple as a keyboard prompt or a Jupyter notebook cell stop) to pause after some iterations. During the pause, the user can inspect plots and perhaps adjust hyperparameters (like search range or weights in the objective) before continuing.
- **Gradient-Based Refinement:** Once the parameters are roughly in the right zone from the global search, an optional step is to fine-tune with a local optimizer. For example, take the best solution from CMA-ES and feed it into `scipy.optimize.minimize` with the L-BFGS-B method (which can handle bound constraints) for a few iterations to squeeze out extra accuracy. Since L-BFGS-B is gradient-based, SciPy will internally approximate gradients by finite differences if not provided, which is fine for 3 parameters. This hybrid approach (global then local) can accelerate convergence at the end. Keep this as a separate function or step that can be run on demand.
- **Logging and Saving Progress:** Throughout the optimization loop, log the progress. For instance, print or save to a file the current generation, the best error, and the best parameters found so far. This provides a trace of the optimization and is useful for debugging or resuming if something crashes. You might also save checkpoints (the optimizer state or the best params) every so often. This modular design lets you stop the process and later restart from the last best solution, or even try different manual tweaks in between.

By structuring the optimization this way, you get the benefits of automation **and** the flexibility of human insight. The automatic optimizer will handle the heavy lifting of searching the parameter space, while the modular loop and callbacks enable you to monitor and intervene. This is especially useful if, for example, you notice the optimizer focusing on a wrong trend – you can adjust the objective weights or bounds and continue. Overall, this approach balances efficiency and insight, leveraging tools like CMA-ES (well-suited for difficult black-box tuning [3]) while keeping you in the loop for critical judgments.

4. Visualization for Diagnostics During/After Optimization

Visualization is crucial both during the calibration process and after obtaining a solution. It helps verify that the simulation is matching reality in the ways we expect and can reveal *why* the optimizer is choosing certain parameter values. We will generate several types of plots and visual diagnostics:

- **Real vs Simulated Trajectory Plots:** This is the most direct visualization. Plot the 2D trajectories of the bots from the real data and from the simulated data on the same axes. For clarity, use different styles or colors (e.g. solid line for real trajectory and dashed line for simulated, or different colors for each). Since there are 3 bots, you can have one figure with all 3 trajectories plotted (label the bots or color-code them). Mark the start and end positions (for example, a circle at the start and a square at the end of each trajectory) to see where they begin and finish [5] [6]. A good match means the sim trajectories should closely overlay the real ones. If they diverge, you'll visually see when and how (e.g. one bot might wander off more in sim than real, indicating a parameter issue).
- **Trajectory vs Time Plots:** In addition to the 2D paths, plot the X and Y coordinates of each bot over time (real vs sim). These time-series plots (perhaps in separate subplots per bot or per coordinate) allow precise inspection of dynamics. For instance, you might see that the simulated oscillations have a higher amplitude than real in the X-direction – a clue that attraction/repulsion (alpha/beta) might be off. Or perhaps a phase lag in a bot's oscillatory motion – possibly indicating the tangential coupling f0 needs tuning. Overlaying or side-by-side plotting helps pinpoint such discrepancies.
- **Cluster Radius/Center-of-Mass Over Time:** Plot the cluster radius vs time for real and simulated data on the same graph. Similarly, plot the center-of-mass trajectory vs time (or its speed). If the calibration is good, these high-level behaviors should align (e.g. the cluster radius in simulation expands or contracts in sync with the real cluster). A mismatch here, even if individual trajectories look okay, could indicate the overall spacing or cohesion isn't captured. These plots condense the swarm behavior into single curves that are easy to compare.
- **Error Evolution Plot:** During optimization, plot the objective value (error) vs iteration/generation. This could be a simple line plot showing how the best-so-far error decreases over iterations. It helps diagnose the optimization progress (e.g. a plateau might mean a local minimum or too low mutation in CMA-ES). If using an evolutionary algorithm, you could also plot the population's error distribution per generation (e.g. min, median, max error in each generation) to see convergence. This kind of plot can be updated live in a Jupyter notebook or saved after the run. It's useful for deciding when to stop (if it has flatlined) or whether to adjust settings.
- **Parameter Trajectory Plot:** If applicable, visualize how the candidate parameters are changing. For instance, in CMA-ES you can record the mean or best solution each generation; plotting alpha, beta, f0 values vs iteration might show, say, alpha trending upward to some value. This can indicate which parameters the optimizer is most sensitive to. If one parameter hardly changes, it might mean it's not identifiable from the data (multiple combinations yield similar error) or the initial guess was already near-optimal for that parameter.
- **Intermediate Simulation Visualization:** Since we have the ability to pause the optimization, we can at that point take the current best parameters and run a full simulation (with those params) to generate trajectories, then visualize them against the real data as described. Doing this in the middle of the run (e.g. after some generations) helps to verify we're on the right track. If the visual mismatch is still large in some aspect, a human might deduce a needed tweak (for example, "the robots are clumping too tightly in simulation – maybe increase beta (repulsion)").
- **Animated Comparisons (optional):** For a final presentation or deeper analysis, you could create an animation showing the real vs simulated motion over time. For example, draw the bots in real

experiment moving (from video data) and the simulated bots moving side-by-side, or on the same frame if properly aligned. This is more complex to implement (synchronizing an animation), but it can dramatically reveal any dynamic mismatch. In practice, a simpler approach is to animate the simulation and ensure you've seen the original video; if the calibrated simulation "looks" like the video qualitatively, that's a strong sign of success.

For implementing these visualizations, use **Matplotlib** for static plots. You can set up a function `plot_diagnostics(params)` that runs a sim with given parameters and produces all the comparison plots in one figure or a set of figures. This can be called during the optimization (via callback) and after final calibration. By citing these plots in reports or lab meetings, you also have evidence of how well the model matches real data.

During optimization, you might not want to plot every iteration (to avoid slowdown), but perhaps every Nth iteration or when a new best is found. After optimization, definitely generate the full suite of plots with the final calibrated parameters. These diagnostics will confirm if the error minimization indeed corresponds to visually improved agreement, and they may highlight any remaining deviations (which could drive further model refinement or re-checking data).

5. Validation: Residual Analysis and Generalization Tests

After calibration, it's important to validate the calibrated model to ensure it truly captures the system dynamics and isn't overfitting to the specific data used. We will perform several validation steps:

- **Residual Analysis on Training Data:** Compute the *residuals* – the differences between simulated and real values – over time for the data that was used in calibration. Plot the residuals for each bot's position vs time. Ideally, these residuals should be roughly zero-mean noise (within the noise level of the tracking), with no obvious systematic trend. Check if residuals grow over time (which might indicate a slight model drift), or if there are cyclic patterns (indicating some oscillatory behavior not captured by the model). For example, if you see that the simulation always lags behind the real trajectory by a bit, the model might be slightly slower in response – perhaps f0 or alpha could not perfectly capture some drag effect. Quantify the final RMSE and perhaps the maximum error over time to get a sense of accuracy.
- **Validation on Unseen Data:** Use your calibrated parameters on **new experimental runs** that were not part of the calibration (if available). Feed the initial positions and inputs from those runs into the simulation (with the fixed calibrated params) and then compare the simulation output to the real trajectories for those runs. This tests generalization. Look at the same metrics: trajectory error, cluster radius, etc. If the errors remain low and the plots show good alignment, it indicates the model has captured the underlying physics well. If the errors are much larger for a new run, analyze what's different – perhaps the new run has a different spin rate pattern or initial spacing, revealing a limitation of the model (e.g. maybe alpha, beta, f0 aren't constant for all conditions, or there's unmodeled friction when speeds differ).
- **Statistical Measures:** Calculate summary statistics on the validation errors: e.g. mean RMSE across several runs, standard deviation of error, etc. If possible, compute confidence intervals or perform a hypothesis test that the errors are within an acceptable range. This is more formal, but for a small number of runs, just ensuring qualitatively and quantitatively that errors are small is sufficient.
- **Corner Case Tests:** Validate the model on any edge conditions if relevant – for instance, if in some experiment the bots start unusually far apart or close together, check if the model still behaves

realistically in those regimes with the same parameters. If your calibration data didn't cover that regime, this is an extrapolation test. For example, if bots start very far, does the simulation with calibrated alpha still attract them appropriately? Or if they start almost touching, does the strong repulsion (beta) prevent overlap as in real life? Successful reproduction of such scenarios boosts confidence that the model isn't just fit to one specific scenario.

- **Compare Derived Quantities:** Even if your primary fit was on trajectories, look at other derived quantities in validation runs. For instance, measure the **cluster rotation speed** (how fast the group spins, if they orbit each other) in real vs sim, or the frequency of any oscillatory behavior. If the model is truly calibrated, these secondary aspects should also match. Another example: how long does it take for the three bots to cluster together or reach a steady formation? Compare that settling time in real vs simulation.
- **Revisit Model Assumptions if Needed:** If significant discrepancies persist in validation, consider that the model form might be missing something (e.g. maybe a damping term is needed or spin coupling depends on both bots' spin, not just one). Calibration can reveal model inadequacies: for example, if no single set of (alpha, beta, f0) works for two different experiments, it might mean the model needs an additional parameter or a context-dependent term. At that point, one might refine the model and then recalibrate. Document any such residual mismatches as targets for future improvement.
- **Repeatability:** It's also valuable to see how stable the calibrated parameters are. If you calibrate on two different data sets (or even the same data but starting the optimizer from different random seeds), do you get similar alpha, beta, f0? If yes, that's good – the data contain enough information to identify a clear optimum. If not (the solutions differ significantly yet produce similar error), the parameters might be partially unidentifiable (e.g. maybe increasing alpha and beta together produces similar behavior, meaning there's a trade-off in solutions). In such a case, you might constrain one parameter or collect more varied data to break the degeneracy.

In summary, validation ensures that the calibrated model truly predicts *new* behavior and that the residual errors are random rather than systematic. A successful validation would show low errors on data that the optimizer never saw [7], confirming that the model has captured the real system's dynamics and can be trusted for control. Any validation failures provide insight into how the model or calibration process might be improved (additional data, more parameters, etc.).

6. Python Libraries and Toolchain

We will utilize a Python-based toolchain to implement the above strategy. The following libraries and tools are recommended for efficiency and clarity:

- **NumPy**: For all array and numerical computations. The simulation state (positions of bots) can be stored in NumPy arrays, and vectorized operations can compute distances or forces efficiently. NumPy will be used heavily in the simulation code (as in the provided model) and in data handling (loading positions, computing metrics, etc.).
- **SciPy**: Particularly the `scipy.optimize` module for optimization routines. SciPy provides:
- `scipy.optimize.minimize` for local optimization (with methods like **L-BFGS-B** for bounded problems [8]).
- `scipy.optimize.differential_evolution` for a built-in global optimization (genetic algorithm).

- `scipy.optimize.least_squares` which is useful if you formulate the problem as least-squares (it can leverage the structure if you supply residuals), though in our case a custom objective is fine. SciPy's optimizers are well-tested and easy to use; they also allow callback functions (to implement our pause/visualize logic).
- **CMA-ES Library**: The `cma` library (pycma) is a convenient implementation of CMA-ES in Python [9]. It's pip-installable (`pip install cma`) and provides a function `cma.fmin` where you give it the objective function, initial guess, and initial sigma. This library will handle the rest. It also allows retrieving the best solution at each iteration and can be instructed to stop after a certain number of iterations, aligning with our iterative approach.
- **Particle Swarm Optimization Library**: If exploring PSO, **PySwarms** is an extensible toolkit for PSO in Python [4]. It provides a high-level interface to run PSO given an objective and bounds. Alternatively, a simpler library `pyswarm` (note: different from PySwarms) offers a basic PSO implementation. Depending on familiarity, you can choose either or implement PSO manually for learning purposes (PSO is relatively straightforward to code for 3 parameters).
- **Matplotlib**: For all plotting needs (diagnostic plots, etc.). We will create plots during optimization (perhaps using interactive mode or updating plots in Jupyter) and after optimization for final reports. Matplotlib can also create animations (using `matplotlib.animation.FuncAnimation`), which could be used to animate the swarm movement as part of validation or presentation.
- **Pandas** (optional): If the experimental data is in CSV or needs cleaning, Pandas can simplify data loading and filtering. For instance, reading a CSV of positions into a DataFrame, applying a rolling average filter, etc. After processing, we would convert to NumPy arrays for the simulation and objective function.
- **OpenCV or Image Processing** (if needed): Since the user already has tracked positions, we might not need to do image processing. But if further video analysis is required (say to verify the tracking or measure spin rates visually), OpenCV (cv2) could be used. For example, to detect a bot's orientation or spin, one might analyze video frames – however, this seems outside our current scope as spin rates are likely known from the experiment control.
- **Parallel Computing Utilities** (optional): If using population-based optimization (CMA-ES, PSO), note that evaluating the objective for each candidate can be done in parallel to speed up each generation. With 3 bots and a lightweight sim, it might not be necessary, but if simulation becomes a bottleneck, consider Python's `multiprocessing` or joblib to distribute simulations across CPU cores. The `cma` library allows a user-defined evaluation function that could internally spawn parallel evaluations. PySwarms also supports parallel updates. This can cut down runtime significantly, especially if each simulation run (for a given param set) takes a noticeable fraction of a second.
- **Code Organization**: Structure the code into modules/scripts for clarity:
- A **simulation module** (or function) that given parameters and initial conditions, runs the ODE integration. For instance, a function `simulate_swarm(alpha, beta, f0, initial_positions, inputs, dt, T)` that returns the trajectory (positions over time). Internally it will implement the force calculations and integration (using RK4 as in the provided model, or even using SciPy's ODE integrators for robustness).
- A **data module** that loads experimental data and has utility functions like `compute_cluster_radius(positions_over_time)`.
- A **calibration script or notebook** that glues everything: loads data, defines the objective (maybe as a closure capturing the data and calling simulate), and sets up the optimizer loop.
- A **visualization module** with functions for plotting (e.g. `plot_trajectories(real_positions, sim_positions)` and `plot_metrics(real, sim)` etc.).

- By separating these, you can more easily reuse parts (for example, the simulation module will also be reused in the MPC integration).
- **Version Control & Experiment Logging**: As you tweak the calibration process, it helps to keep track of parameter versions. Consider using Git to version your calibration code (so any changes to the objective or process are tracked). Also, for each calibration run, log the settings (which optimizer, what data used, initial guess, etc.) and outcome. This could be as simple as writing to a text log or as structured as storing in a JSON or using an experiment tracking tool. This practice will make it easier to reproduce results or try variations (e.g. "did adding the radius term actually improve validation error?" can be answered by comparing logged outcomes).

Overall, the Python ecosystem provides all necessary tools: SciPy and specialized libraries for optimization, Matplotlib for visualization, and robust data handling with NumPy/Pandas. This ensures our implementation is efficient and our code is organized for testing and future integration.

7. Integration with the MPC Pipeline

Once the model is calibrated, the tuned parameters (alpha, beta, f0) will be used in a Model Predictive Control (MPC) framework for real-time control of the robot swarm. It's important to integrate the calibrated model in a way that MPC can utilize it for predictions within each control loop:

- **Embedding the Model in MPC:** The MPC needs a dynamical model of the system to predict future states. We will incorporate the calibrated simulation model as this predictive model. This can be done in continuous or discrete form:
- *Continuous-time approach:* Use the ODE (equations of motion with the calibrated parameters) directly in the MPC solver, typically by discretizing it over the prediction horizon. For example, you can use a fixed integration step (like the same RK4 with dt as in simulation) for each step in the horizon. Tools like **CasADi** can help create an integrator from ODEs that is usable in an optimization problem [10]. With CasADi, you define the state evolution $\dot{X} = f(X, u;\;\alpha,\beta,f0)$ (where u might be the spin inputs or other control variables), and CasADi can internally perform the integration and provide the result to the nonlinear programming solver. This allows the MPC to treat the simulation model equations as constraints in the optimization.
- *Discrete-time approach:* Alternatively, derive a discrete update function from the model. For instance, a function $X_{k+1} = F(X_k, U_k)$ that advances the state by one control timestep (e.g. one video frame or any convenient step) using the calibrated forces. You can hard-code one step of RK4 or Euler integration as this update. This discrete state transition function can then be used in a classical MPC formulation where you iterate it over the horizon.
- **Real-Time Considerations:** Ensure the model is simplified enough for real-time use. A 3-bot model with our simple equations is not heavy to compute, but if MPC runs at e.g. 10 Hz and has a horizon of 2-3 seconds, it will be integrating the model many times per optimization. We might simplify the computations by vectorizing wherever possible. Since $N=3$, it's already trivial, but keep an eye on unneeded overhead (for instance, if using Python in the loop vs. letting a solver handle it in C/C++ under the hood via CasADi or similar). Test how long it takes to compute the model forward propagation over the horizon; it must be a small fraction of the control period to allow the solver to converge.
- **MPC Formulation:** With the model in place, design the MPC cost function and constraints according to your control goals. For example, if the goal is to make the bots move in a certain formation or reach a target location, the MPC cost would penalize deviation from desired positions, while the

model constraints ensure the predictions follow the calibrated dynamics. Incorporate any actuation limits (e.g. maximum spin rate changes) as constraints. The calibrated model improves the fidelity of these predictions, meaning the MPC's planned trajectory should more closely match what the real robots will do [11].

- **Use of Calibrated Parameters:** The values of alpha, beta, f0 become fixed constants in the MPC model after calibration. It's good to code them as configuration parameters that can be loaded (so if you recalibrate in the future, you just update a config file). The MPC controller does not change these online (assuming the environment doesn't change); they are part of the plant model. If there is concern that the system might change over time (e.g. battery levels affecting magnet strength), you could periodically re-calibrate or adapt these parameters, but initially assume they are fixed after our offline calibration.
- **Testing MPC in Simulation:** Before deploying on the real robots, test the MPC in a simulated environment using the same model. Essentially, do a *software-in-the-loop* test: the MPC will control a copy of the simulation. Because the simulator is now calibrated, this test will be more meaningful – success here should translate to success on the actual hardware. During these tests, you can verify that the MPC meets real-time requirements (how fast it solves) and that the behavior is stable and achieves goals (e.g. the bots respond correctly to control inputs). This simulation test can also reveal if the model, even calibrated, has any residual issues that affect control (like slight delays or overshoots).
- **Integration into Codebase:** Implement the MPC using frameworks as needed:
- If using CasADi, formulate the optimal control problem and solve with an NLP solver (like IPOPT or ACADOS). CasADi will use the model equations (with calibrated params) for predictions and can compute necessary gradients for the solver via automatic differentiation [10].
- If using a more hand-coded MPC (like a sequential quadratic programming loop or Linear MPC), you might linearize the calibrated model around operating points. For example, derive a Jacobian of the system dynamics at a nominal state to use in a linear MPC if needed. However, given the nonlinearity of magnetic interaction, a nonlinear MPC is more direct.
- Integrate sensor feedback: the MPC at runtime will get the current positions of bots (from tracking or state estimation) and use the model to predict forward. Ensure that the coordinate frames and scaling in MPC match those used in calibration (for consistency).
- **Interface and Real-Time Execution:** The final MPC controller will likely run in a loop where each cycle it:
- Reads the current state (bot positions, etc.).
- Solves the MPC optimization using the model (with calibrated parameters) to get optimal control actions (e.g. adjustments to spin rates or magnet currents).
- Sends those commands to the robots for the next small time step.
- Repeats.

Because of this, packaging the calibrated model in an easily callable form is helpful. For example, you might create a function or class `MASBotsModel` with a method `predict_state(x_current, u_current, dt)` that advances the state by one step. This can be used by the MPC solver or even for simple roll-out predictions. The calibration work ensures that `MASBotsModel` is accurate.

- **Future Adaptation:** Keep in mind that if the MPC is extended to more bots or different conditions, the model might need re-calibration. The pipeline we designed can then be reused. In a real deployment, one could envision periodically using new data to refine the model, and feeding that back into the MPC – an adaptive control loop. In literature, calibrated simulators are seen as a way to continually improve control policies [11].

In essence, integrating with MPC means using the calibrated model as the “digital twin” of the real swarm inside the controller. By doing so, the MPC’s performance is greatly improved since its internal predictions align with reality [11]. This closes the sim-to-real loop: we calibrate the sim from real data, and then use the sim to control the real system.

8. Synthetic Data Test of the Calibration Loop (Optional but Recommended)

Before applying the calibration procedure to real-world data, it’s wise to verify the process on synthetic data. This means using the simulator itself to generate data (with known parameters) and seeing if the optimization loop can recover those parameters. Such a test serves as a proof-of-concept and can uncover issues in the pipeline (objective formulation, optimizer settings, etc.) under controlled conditions [7]. Here’s how to conduct a synthetic calibration test:

- **Generate Synthetic “Experimental” Data:** Choose a set of “true” parameter values for alpha, beta, f0 – ideally in the ballpark of expected real values. For example, take alpha=0.1, beta=0.02, f0=0.2 (or whatever seems plausible). Use the simulation code to simulate a 3-bot scenario with those parameters, starting from some initial positions and perhaps with a certain spin rate pattern. This simulation should produce trajectories analogous to what a real experiment might show. You can even add some artificial noise to the generated positions (e.g. add small random jitter each time step to mimic measurement noise or unmodeled disturbances). Save this synthetic trajectory data.
- **Apply the Calibration Procedure:** Now pretend this synthetic data is the real experiment data and run your entire calibration pipeline on it (data extraction is trivial here since we have direct states). Give the optimizer an initial guess that is different from the true values (to test if it converges to the truth). Run the optimization loop (CMA-ES/PSO or whichever method) to see if it finds parameters close to the ones you used to generate the data. Because the model perfectly matches (in structure) the data generator in this synthetic case, a successful calibration should recover the exact parameters (within some tolerance, depending on noise).
- **Evaluate Recovery Accuracy:** Check the results of the synthetic calibration. Did the optimizer converge to the known true values? If not, examine why:
- If the solution is way off, there might be an issue (maybe the objective function isn’t capturing differences properly, or the optimizer got stuck). This is a safe environment to debug those issues. For example, perhaps you discover that alpha and f0 can trade off to produce similar trajectories (an identifiability problem) – then you might revisit the objective or consider collecting different types of maneuvers to break that symmetry.
- If it converged but slowly or with difficulty, you might tweak optimizer settings (population size, mutation rate, etc.) to improve performance before dealing with real data.
- If there was noise added, see how close the recovered parameters are and how the residual error looks. This gives a sense of how robust the calibration is to measurement noise. If the noise caused noticeable estimation error, consider if your real data noise needs filtering or if you should increase the data length for a clearer signal.
- **Test Various Scenarios:** You can run multiple synthetic tests: try different initial conditions or different true parameters to ensure the method works generally. For example, test one where bots start far apart vs one where they start close. The pipeline should succeed in both. This helps ensure the calibration won’t fail simply because of a peculiarity in one experiment’s setup.

- **Use Findings to Refine Process:** Any insight from synthetic tests can refine the actual calibration. You might adjust the weighting of objective terms if, say, you noticed it was focusing too much on a minor metric. Or you might realize the need to collect a certain type of data (e.g. if alpha and beta were hard to distinguish in a static cluster, maybe an experiment where bots intentionally move in a gradient is needed).
- **Dry-Run the MPC Integration:** As an extended synthetic experiment, you could also simulate an *MPC scenario* using the synthetic model and then pretend to calibrate. However, this is usually not needed – it's more important to ensure calibration is working. After calibration, though, you could simulate the MPC controlling the synthetic model to ensure the whole loop (calibration + control) is consistent and stable before trying on real robots.

Performing this synthetic calibration is like a unit test for your system identification procedure [7]. It gives confidence that, barring modeling mismatch, your optimization can indeed find the right answers. Only then do we move on to using the precious real-world data for calibration. It can save time and prevent frustration to iron out kinks in silico.

References: The approach combines best practices from system identification and optimization literature. For example, CMA-ES has been highlighted as a powerful method for calibrating complex model parameters [9], thanks to its robustness on non-convex problems and invariance to parameter scaling [3]. We have drawn on insights that calibrated simulations can significantly bridge the sim-to-real gap, allowing learned controllers to operate reliably on hardware [11]. The overall strategy emphasizes a rigorous, testable pipeline: from data processing to objective design, automated and manual hybrid optimization, thorough validation, and finally deployment in an MPC context for real-time control. By following this roadmap, one can systematically tune the MASBotS model to mirror reality, leading to more effective and confident control of the 3-bot swarm.

---

[1] [11] proceedings.mlr.press
https://proceedings.mlr.press/v155/mehta21a/mehta21a.pdf

[2] [3] nnw.cz
https://nnw.cz/doi/2019/NNW.2019.29.020.pdf

[4] Welcome to PySwarms's documentation! — PySwarms 1.3.0 ...
https://pyswarms.readthedocs.io/

[5] [6] bot_swarm_simulation.pdf
file://file-2ELuCbEVYMDJ5YyuHY9wK9

[7] A multi-step calibration strategy for reliable parameter determination ...
https://www.sciencedirect.com/science/article/pii/S1365160924002879

[8] minimize — SciPy v1.16.2 Manual
https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html

[9] Estimating process-based model parameters from species ...
https://besjournals.onlinelibrary.wiley.com/doi/full/10.1111/2041-210X.14119

[10] CasADi
https://web.casadi.org/