Hybrid Particle/Evolutionary Optimization Framework

A Practical Guide for Model Fitting and Optimal Experimental Design

Bob van Sluijs

October 16, 2025

Abstract

This guide explains how to use the provided hybrid metaheuristic optimizer to (i) fit parameters of one or multiple dynamical system models to data and (ii) design informative input profiles for optimal experiments (OED). We start with the user-facing interface — how to define or import a model, how to declare measurements and conditions, and how to launch the optimizers. We then dive into the underlying algorithm and solver pipeline so that new users can interpret logs, adjust settings, and extend the framework safely. The document targets novices; it includes runnable code snippets, clear terminology, and step-by-step workflows.

Contents

1 Prerequisites and Installation

1.1 Python Packages

- Core scientific stack: `numpy`, `scipy`, `pandas`, `matplotlib`.
- Modeling stacks (choose what you have): `amici`, `libsbml`, and optionally `tellurium`.
- Optional: `statsmodels`, `seaborn`, `pyDOE`.

If AMICI is unavailable on your platform, you can still run models via Tellurium or SciPy's LSODA; the code selects the best available backend at runtime.

1.2 Folder Outputs

By default, models and figures may be written to a Desktop subfolder (e.g., `Desktop/__Models__/`) for convenience. You can change those paths in the source if needed.

2 Core Concepts

2.1 States, Parameters, and Controls

A *state* is a dynamic variable in your ODEs. A *parameter* is any model constant (rate, affinity, etc.). A *control* is a parameter you *apply over time* (e.g., an input concentration or stimulus); controls are mutated in OED.

2.2 Index-Based Parameter Grids

Each parameter is discretized over a value grid (by default logarithmic). Optimizers mutate integer indices; the framework maps indices to numeric values. This makes both local ($\pm 1 \ldots 3$ steps) and global (jump anywhere) moves simple and robust.

3 Defining a Model (Interface)

3.1 Build a `ModelObject`

You can define your ODEs as human-readable strings, list the state names in order, and provide parameter bounds. A minimal pattern:

```
from Model import ModelObject, runModel

model_equations = (
    "A' = + k1*A - k2*A*B\n"
    "B' = - k3*B + k4*A*B\n"
)

states = ["A", "B"]
boundaries = {
  "k1": (1e-3, 1e1),
  "k2": (1e-5, 1e-1),
  "k3": (1e-3, 1e1),
  "k4": (1e-5, 1e-1),
}
fixed = {}            # Optional: give fixed parameter values
observables = ["A","B"]
control_parameters = []  # Parameters that are considered "controls" (
   OED)
initial_control = []     # Optional mapping from parameter->state
   initial
name = "ToyModel"

M = ModelObject(
    model=model_equations, states=states,
    boundaries=boundaries, fixed=fixed,
    control_parameters=control_parameters,
    initial_control=initial_control,
    observables=observables, name=name
)
```

What happens under the hood. The model preprocessor tokenizes equations, builds a stoichiometric matrix and flux vector, prepares symbol maps for states/parameters, and sets up derivative structures for Jacobians/Hessians.

3.2 Compile and Prepare the Model

```
# Optionally compute derivative structures, generate SBML/Antimony, and
    compile via AMICI.
M = runModel(M, derivatives=True, compilation=True, show_reactions=
   False)
```

If compilation fails or AMICI is not installed, simulations automatically fall back to Tellurium or SciPy/LSODA.

3.3 Set Conditions and Initials with `ModelVariables`

Use `ModelVariables` to connect model definitions to a particular experimental setup:

```
from Model import ModelVariables

conditions = {"A": 1.0, "B": 0.5}  # states at t0
overrides  = {"k1": 0.2}           # (optional) parameter overrides

vars = ModelVariables(M, conditions=conditions, modification=overrides)
```

For OED, you can also define time-dependent controls (pulse sequences) as grids (covered in §7).

4 Simulating a Model

4.1 The Solver `ModelSolver`

```
from SolveSystem import ModelSolver

solver = ModelSolver(
    M, variables=vars,
    forward_sensitivity=True,  # optional
    measurement_time=[],       # or provide an explicit time vector
    simtime=(0, 100), dt=1.0   # coarse time stepping
)
data = solver.__getData__()    # returns an object with .simdata, .time
   , ...
```

Backends. The solver tries AMICI first (C++ speed, sensitivities). If AMICI is not present or compilation fails, it tries Tellurium (Antimony/SBML). As a last resort, it integrates the symbolic Python ODEs with LSODA.

Time-Dependent Inputs. You can provide a mapping from time windows to control values; the solver will run sequential segments and stitch trajectories. Forward sensitivities (if enabled) are propagated across segments.

5 Preparing Measurements (Interface)

A *measurement* bundles (i) a model reference, (ii) a time grid, (iii) observed state series as a dict of arrays, and (iv) conditions and optional time-dependent inputs. Minimal schema:

```
class Measurement:
    def __init__(self, model, time, profile, conditions,
        time_dependent_parameters=None):
         self.model = model
         self.time = time                  # 1D array
         self.profile = profile            # dict: {state -> 1D array}
         self.conditions = conditions      # dict: initial/control values
         self.observables = list(profile.keys())
         self.time_dependent_parameters = time_dependent_parameters or
            {}

# Example
import numpy as np
t = np.linspace(0, 100, 101)
meas = Measurement(
  model=M, time=t,
  profile={"A": np.zeros_like(t), "B": np.zeros_like(t)},
  conditions={"A":1.0, "B":0.5}
)
measurements = [meas]
```

The optimizer modules operate on lists/dicts of such measurement objects.

6 Parameter Estimation (Interface)

6.1 Quick Start

```
from ParameterOptimizationModelSets import Optimization

opt = Optimization(
    measurements=measurements,
    include=[],            # which parameters may mutate; empty => all
       (except controls)
    generations=250,
    agents=10,
    time=(0, 100), dt=1,
    startsamples=100,      # Monte Carlo seeds for multistart
    sobol=False,
    forward=True,          # enable sensitivity-guided mutation
    probability_random_mutation=0.25,
    sf="leastsquaresfit"   # scoring function (see below)
)
# After construction, the optimizer runs.
best = opt.fittest           # agent object with best fit
params = best.fitset         # dict of parameter -> value
sim = best.simdata           # simulated trajectories for the best
   result
```

6.2 What `Optimization` Does

1. Builds a combined parameter grid across all participating models.
2. Creates a set of agents, each with an initial parameter vector from a multistart routine (Monte Carlo or Sobol).
3. Iteratively proposes mutations (local/global) to each agent; accepts a move only if it improves the score.
4. Periodically recombines underperformers toward fitter agent coordinates.
5. Tracks convergence and stops when the score plateaus or a generation/time limit is reached.

6.3 Scoring Functions

The `sf` argument selects how fitness is computed. Common options include `"leastsquaresfit"` and `"standardeviation"` for fitting; specialized criteria (e.g., Fisher information or ESS) are used in OED. You can extend or swap scoring rules in the dedicated scoring module.

7 Optimal Experimental Design (Interface)

7.1 Control Grids and Pulse Coordinates

Controls are discretized in time windows (e.g., 20-step windows), each window taking an index value on the control grid. The OED optimizer mutates these indices to find informative stimuli.

7.2 Quick Start

```
from OptimizeExperiment import OptimizeInformation

oed = OptimizeInformation(
    model=M,
    time_dependent_parameters=["u1", "u2"],  # control parameter names
    conditions={"A": 1.0, "B": 0.5},
    observables=["A", "B"],
    generations=30, agents=5,
    time=(0, 300), dt=1,
    pulsestart=10, plength=20, pindex=10,   # time grid and control
        index grid
    multistart=10,
    sf="D_Fisher"       # OED metric; alternatives include "ESS"
)
best = oed.fittest
best_pattern = best.coordinates   # time->index maps for each control
best_sim = best.simdata
```

7.3 What Gets Optimized

The OED agent stores a dictionary `coordinates`:

```
{control_name: {(t0,t1): index, (t1,t2): index, ...}}
```

This is mutated by contiguous block moves (left/right along the time grid) and occasional random reindexing. The solver maps indices to numeric control values via the control parameter grid ("pulsespace") and simulates the experiment end-to-end.

8 Algorithm Overview (Under the Hood)

8.1 Two Layers: Landscape and Agents

Each agent holds: current parameter (or control) coordinates, the corresponding numeric vector, current score, simulation results, and a move history. The landscape layer schedules move proposals, accepts improvements, and periodically recombines populations.

8.2 Moves

Local mutations shift indices by a few steps. **Global mutations** reassign indices randomly. For OED, block mutations change an entire contiguous time window. Some proposals use *forward sensitivities* (if available) to bias the sign/direction of moves.

8.3 Acceptance and Recombination

A proposed move is accepted iff it strictly improves the chosen score. After a fixed number of iterations, underperformers are pulled toward the fittest agent (centroidal or parent-based recombination).

8.4 Convergence & Termination

The optimizer tracks the best score per generation. It stops when no improvement is observed for a while (plateau), a maximum generation count is reached, or diversity has collapsed (e.g., all agents coalesce).

9 Advanced Usage

9.1 Sensitivity-Aware Mutations

When `forward=True`, the simulation stack returns forward sensitivity matrices and the optimizer increases mutation probabilities for parameters with large projected improvements.

9.2 Manual Time-Dependent Inputs

You may bypass the OED layer and directly provide `manual_TDI` to the solver, e.g. for fixed experimental plans:

```
manual_TDI = {
  (0, 20): {"u1": 0.1},
  (20, 40): {"u1": 0.5},
}
solver = ModelSolver(M, variables=vars, manual_TDI=manual_TDI, simtime
   =(0,40))
```

9.3 Multiple Models and Shared Parameters

You can fit several models simultaneously to different measurements while sharing parameters. The framework builds a combined grid; a shared parameter mutates once and each model resimulates with the shared value.

10 Troubleshooting

- **Namespace errors.** Ensure state and parameter names in data/conditions match those declared in the model.
- **Stiff or failing integrations.** Try enabling AMICI (if installed), or relax tolerances, or reduce `dt`. Ensure initial conditions are physically valid (non-negative, etc.).
- **No convergence.** Increase agents/generations, raise random mutation probability, or widen parameter bounds. For OED, increase the control index grid (`pindex`) or mutate larger blocks.
- **Unrealistic best fits.** Constrain parameters by narrowing `boundaries` or marking some parameters as `fixed`.

11 Minimal End-to-End Example

```
# 1) Define model
M = ModelObject(...)
M = runModel(M, derivatives=True, compilation=True)

# 2) Prepare measurement
meas = Measurement(model=M, time=t, profile=..., conditions=...)
measurements = [meas]

# 3) Fit parameters
opt = Optimization(measurements, include=[], generations=150, agents=8,
    forward=True)
print("Best parameters:", opt.fittest.fitset)

# 4) (Optional) Optimize an experimental input pattern
oed = OptimizeInformation(model=M, time_dependent_parameters=["u1"],
   time=(0,200), dt=1,
                          pulsestart=20, plength=20, pindex=10,
                             generations=30, agents=5)
print("Best OED pattern:", oed.fittest.coordinates)
```

Acknowledgments

This framework builds on AMICI, libSBML, Tellurium, SciPy, and the scientific Python ecosystem.