README for ODE Optimization and Experiment Design software

Bob van Sluijs

February 26, 2025

Contents

1 Overview

This repository provides a flexible and modular Python-based framework for developing, simulating, and optimizing ODE models, particularly those compiled via `AMICI`. It supports forward sensitivities, multi-model parameter estimation, and advanced experiment design (e.g., time-dependent or pulse-based inputs).

Users can define custom reaction networks, automatically generate SBML and C++ code, and then conduct parameter estimation or experiment optimization across multiple datasets and models that share parameters.

2 Key Features

- **Automated Model Building:** Create ODE models programmatically (string-based or partial SBML) with easy extension to advanced kinetics.
- **SBML & AMICI Compilation:** Convert textual or Python-based ODE definitions to SBML, compile them into efficient C++ code via AMICI, and run simulations with forward sensitivities.
- **Multi-Model & Multi-Dataset:** Train multiple interconnected ODE models that share parameters on different datasets simultaneously.
- **Experiment Design:** Tools for time-dependent input patterns (pulses), random input generation, and advanced scoring or identifiability metrics to find informative experiments.
- **Parameter Optimization:** Comprehensive optimization classes
- **Identifiability Analyses:** Inspect parameter identifiability (FIM, correlation, SVD) to understand cross-correlations among parameters.

3 File Structure and Purpose

A possible layout:

```
your_project/
 ModelBuilderOperations.py
 OptimizationOperations.py
 OptimizeExperiment.py
 ExperimentOptimizationOperations.py
 Measurements.py
 Model.py
 SolveSystem.py
 ParameterOptimizationModelSets.py
```

```
IdentifiabilityAnalysisModelSets.py
TransformModeltoSBML.py
...
```

1. `[insert specific operation]Operations.py`: Utilities that are used repeatedly within the core modules. The Operations files are all named separately if they relate to a specific module.
2. `Model.py`: Primary `ModelObject`, storing ODE definitions, parameter boundaries, stoichiometry, etc.
3. `SolveSystem.py`: Core solver logic, hooking to AMICI or Python integrators. Produces uniform data objects.
4. `Measurements.py`: Classes such as `MeasurementObject` and `Observable` to unify data and model output.
5. `ParameterOptimizationModelSets.py`: Multi-agent or evolutionary parameter optimization classes (`Agent`, `Convergence`, `Optimization`).
6. `OptimizeExperiment.py`: Specialized agent-based optimization focusing on experiment design (collinearity, FIM, etc.).
7. `IdentifiabilityAnalysisModelSets.py`: Sensitivity-based identifiability checks, correlation analysis, SVD, etc.
8. `TransformModeltoSBML.py`: Tools to parse textual reactions and construct SBML or an intermediate representation for compilation.

4 Dependencies

- Python 3.x
- `AMICI`
- `NumPy`, `SciPy`
- `pandas`
- `matplotlib`, `seaborn`
- `pyDOE` (Latin Hypercube, Sobol sequences)
- `libsbml` (if using SBML import/export)
- `tellurium` (optional fallback ODE solver)
- `statsmodels` (optionally used in advanced analytics)

5 Workflow Overview

1. **Model Definition:** Provide a text-based or SBML-like representation to create a `ModelObject`.
2. **Compilation:** Optionally use `model.SBMLconversion()` and `model.PytoCompile()` to generate AMICI C++ code.
3. **Data Representation:** Convert real or synthetic data to `MeasurementObjects`.
4. **Simulation:** Use `ModelSolver` (with or without AMICI) to generate time-courses.
5. **Parameter Optimization:** Run `Optimization` in `ParameterOptimizationModelSets.py`.
6. **Experiment Design:** Optionally use `OptimizeExperiment.py` or `ExperimentOptimizationOperations.p` to design input protocols.
7. **Identifiability Analysis:** Evaluate sensitivity-based or SVD-based identifiability.

6 Detailed Module Descriptions

7 Model.py

7.1 Overview

The code defines a framework for parsing, analyzing, and compiling dynamical system models. It supports converting text-based models into various formats (such as SBML and Antimony), performing symbolic differentiation to compute Jacobians, Hessians, and sensitivity equations, and finally compiling the model for simulation using AMICI. The structure is modular, relying on functions and several classes to handle distinct tasks.

7.2 Dependencies and Imports

The file begins by importing several external libraries and user-defined modules:

- **External Libraries:**
  - `libsbml` for SBML file manipulation.
  - `amici` for converting SBML to C++ and simulation.
  - `numpy` for numerical computations.
  - `sympy` for symbolic mathematics.
  - Other utilities such as `pandas`, `datetime`, `copy`, `os`, `sys`, and `time`.
- **User-Defined Modules:**
  - `Parse`, `DataTransform`, and `Operations` provide helper functions for model parsing and data manipulation.

7.3 Function: `runModel`

Purpose: Acts as the main entry point to activate key subfunctions of a model. It handles either a single model or a dictionary of models.

Parameters:

- `model`: A single model object.
- `models`: A dictionary of model objects.
- `derivatives` (bool): If `True`, activates derivative calculations.
- `matrix` (bool): A flag (unused in the function body).
- `compilation` (bool): If `True`, triggers SBML conversion and compilation.
- `forward_observables`: List of observables for sensitivity analysis.
- `sensitivity_parameters`: List of parameters for sensitivity computation.
- `show_reactions` (bool): If `True`, prints generated reactions.

Behavior:

- For a single model, it converts the model to SBML and compiles it via `model.PytoCompile(...)`.
- For multiple models, it iterates through the dictionary and performs the same operations on each model.

7.4 Class: `RateLaw`

Purpose: Encapsulates a reaction rate expression along with the state and parameter dependencies.

Key Attributes:

- `rate`: The string representing the rate expression.
- `states`: A list of state variables present in the rate expression.
- `parameters`: A list of parameter names that appear in the rate.
- `state_indices`: Indices of the included states based on their order.

Initialization: Iterates over provided states and parameters, checking if each appears in the rate string, and constructs the corresponding lists and indices.

7.5 Class: `ModelVariables`

Purpose: Manages the model's parameter settings and initial conditions, including support for time-dependent variables.

Initialization:

- Takes a model object, conditions dictionary, and modifications.
- Differentiates between state initial conditions and control parameters (from model boundaries).

- Validates that condition keys exist in either the states or boundaries.
- Constructs a parameter vector `p` (using `numpy`) and a mapping `pID`.
- Sets up initial conditions (`ic`) for states, including adjustments from `initial_control`.

Methods:

`set_time_dependent_variables(time_dependent_variables, space)`: Sets up time-dependent input vectors for both Python and AMICI simulations.

`update(varlist)`: Dynamically updates parameter values and initial conditions, allowing modifications during optimizations

7.6 Class: `ModelObject`

Purpose: Serves as the core model class encapsulating the model definition, simulation settings, and compilation workflow.

Initialization:

- **Input:** A model string, list of states, boundaries, fixed parameter values, and other options like experimental and initial conditions.
- **Preprocessing:** The model string is preprocessed (e.g., replacing custom delimiters) and split into segments.
- **Mathematical Framework:**
  - Calls `build_equations()` to generate equations from the model string.
  - Creates a unique index mapping for parameters to ensure consistency during SBML conversion.
- **Stoichiometry and Rate Laws:**
  - Constructs a stoichiometric matrix S where each entry S_{ij} is set to $+1$ or -1 based on the reaction direction.
  - Instantiates `RateLaw` objects for each flux.
- **Directory Setup:** Automatically creates folders on the Desktop for storing models, SBML files, AMICI models, figures, and classes.

Key Methods:

`Derivatives(forward_observables, sensitivity_parameters)`: Computes symbolic derivatives (Jacobian, Hessian) using `sympy`, and builds sensitivity equations.

- Jacobian:

$$J_{ij} = \frac{\partial f_i}{\partial y_j}$$

- Hessian:

$$H_{ijk} = \frac{\partial^2 f_i}{\partial y_j \partial y_k}$$

- Sensitivity equations for state derivatives with respect to parameters:

$$\frac{d}{dt}\left(\frac{\partial y_i}{\partial p_j}\right) = \sum_k \frac{\partial f_i}{\partial y_k}\frac{\partial y_k}{\partial p_j} + \frac{\partial f_i}{\partial p_j}$$

`AntimonyConversion(show_reactions)`: Converts the model into an Antimony format, useful for simulations with Tellurium, and sets up mappings between human-readable and SBML IDs.

`SBMLconversion(show_reactions)`: Generates an SBML file if it does not already exist. Uses Tellurium to convert an Antimony string into SBML and updates mapping dictionaries.

`PytoCompile(show_reactions)`: Compiles the SBML model into a C++ Python module using AMICI.

- Reads the SBML file with `libsbml`.
- Optionally prints reaction details.
- Uses `amici.SbmlImporter` to compile the model.
- Imports the compiled module and sets up internal mappings.

7.7 Directories and File Management

The code creates several directories on the user's Desktop:

- A main folder `__Models__` containing subfolders for `Figures` and `Classes`.
- A folder for SBML models named `__SBMLmodels__`.
- A folder for AMICI models named `__AMICImodels__`.

This setup ensures that all generated files are neatly organized.

7.8 Compilation and Execution Flow

The workflow involves:

- Parsing and processing the text-based model.
- Converting the model into SBML (if not already available).
- Compiling the SBML model into a C++ module via AMICI.
- Using `os.execv` to restart the Python process so that the compiled module is properly loaded.

7.9 Conclusion

In summary, the code provides a comprehensive framework to:

1. Parse and define dynamical system models from text.
2. Compute symbolic derivatives (Jacobian, Hessian) and sensitivity equations (note that AMICI does this automatically).
3. Convert models between text, Antimony, and SBML formats.
4. Compile the model for high-performance simulation with AMICI.

The modular design supports iterative development, parameter estimation, and sensitivity analysis.

8 SolveSystem.py

Runs simulations either via AMICI (for compiled models) or Python-based integrators as a fallback. Manages time segmentation if pulses or piecewise conditions exist. Produces data objects with states and forward sensitivities, AMICI has its own FWD function. This document explains in detail the code in `SolveSystem.py`, which is responsible for solving the model's differential equations and handling simulation data. The code supports multiple simulation backends (compiled via AMICI, Tellurium-based, and a pure Python integrator) and manages both state and sensitivity information. Mathematical notations are used to describe the underlying equations.

8.1 Overview and Dependencies

The solver module imports several libraries and modules for:

- **Numerical Computation:** `numpy` and `scipy.integrate` for handling arrays and ODE integration.
- **Plotting:** `matplotlib.pylab` for visualization.
- **Experimental Design:** `pyDOE` and modules like `Distributions` and `DataEvaluation` for design of experiments and data analysis.
- **Model Handling:** `libsbml`, `amici`, and `importlib` for SBML reading, simulation via compiled C++ modules, and dynamic module loading.

The module defines several data object classes to store simulation results and a main solver class (`ModelSolver`) that orchestrates the simulation process.

8.2 Data Object Classes

The module defines three classes for encapsulating simulation data, Note that this is an intermedeate object with the same structure for different solvers, but since each solver has its own interface seperate objects needed to be defined:

8.2.1 AmiciDataObject

This class handles raw simulation output obtained from AMICI (compiled model simulation). Key attributes include:

- **Raw Data:** Stores the full output from the AMICI simulation.
- **Time Series and State Data:** `time` (timepoints) and `rdata` (state trajectories). Additionally, `ydata` is available.
- **Simulation Data by Observable:** Maps AMICI state IDs to the corresponding model variable names.
- **Jacobian and Sensitivities:** Stores the Jacobian matrix J and the forward sensitivity arrays $\frac{\partial y}{\partial p}$.
- **Initial State:** The state at the end of a simulation segment.

Methods such as `TDI_update` allow appending new simulation segments (useful for time-dependent inputs), while `sensitivity_update` calculates sensitivity factors by comparing simulated data against measurements.

8.2.2 PythonDataObject

This class is a simple container for simulation data when the system is solved using a Python-based integrator. Its attributes include:

- **Name and Time:** Model name and time vector.
- **State Trajectories:** `rdata` holds the raw integrated state values.
- **Simulation Data:** A mapping from state names to their corresponding time series.
- **Initial State:** The final state from the integration.

8.2.3 TelluriumDataObject

This class is used when the simulation is performed using Tellurium. It gathers simulation data similarly to the PythonDataObject but uses the mapping defined in the model (`StringIDToSBML`) to retrieve data by observable. The initial state is stored as a dictionary mapping variable names to their final values.

8.3 The `ModelSolver` Class

Purpose:
The `ModelSolver` class orchestrates the simulation of the model. It accepts simulation settings, including time span, tolerances, integration method, and sensitivity options. It supports three scenarios:

1. **Compiled Simulation:** Using an AMICI-generated C++ module.
2. **Tellurium Simulation:** Using the Tellurium library for SBML-based simulation.
3. **Pure Python Integration:** Using SciPy's `LSODA` when no compiled version is available.

Key Parameters:

- `model`: The model object containing equations and parameters.
- `variables`: A container (e.g., an instance of `ModelVariables`) with initial conditions, parameter vectors, and potential time-dependent inputs.
- `simtime` and `dt`: Define the simulation time span and time step.
- `atol` and `rtol`: Absolute and relative tolerances for the solver.
- `forward_sensitivity` and `adjoint_sensitivity`: Flags to enable sensitivity calculations.
- `manual_TDI`: Allows manual specification of time-dependent parameter changes.
- `pythonintegrator`: Specifies the integration method (default `"lsoda"`) if using Python's ODE solver.

Solver Execution Flow:

- **Time and Pulse Configuration:** The solver sets up the time vector, either from a fixed simulation interval or provided measurement times. If time-dependent inputs are used, the variable `amici_TDI` (or `python_TDI`) is updated accordingly.

- **Compiled vs. Python Integration:**
  - *Compiled Simulation:* If the model has a compiled C model (`Cmodel`), the solver configures the internal solver (setting tolerances, maximum steps N_{steps}, and sensitivity methods if enabled) and runs the simulation via AMICI:
  $$\frac{dy}{dt} = f(t, y, p)$$
  Forward sensitivities, if requested, are computed via AMICI's built-in routines.
  - *Tellurium/Python Simulation:* If the compiled version is not available, the code attempts to use Tellurium (with the `cvode` integrator) to simulate the system. If Tellurium fails or is not applicable, the module falls back on a Python integration method.
- **Python Integration:** For a system of ODEs
  $$\frac{dy}{dt} = f(t, y, p)$$
  the solver defines a function `ODEfunc(t, y, p)` that evaluates the right-hand side (with or without forward sensitivity terms) and uses `scipy.integrate.ode` (with LSODA) to integrate the equations over time.
- **Time-Dependent Inputs:** When pulse patterns or time-dependent parameters are provided, the solver iterates over sub-intervals, updating parameter values at each segment and appending results using the appropriate data object's `TDI_update` method. The simulation time is discretized as
  $$t \in \{t_0, t_0 + \Delta t, t_0 + 2\Delta t, \ldots, t_{\text{end}}\}$$
  with the LSODA integrator or AMICI's compiled solver used to advance the solution. The input for this is defined as dictionary with time bounds and the state of the parameters within that bound. The time delta equates to the bound (i.e. it is not a fixed interval). For example:
  $$input|(0, 5) = p_1 : 1, p_2 : 2$$
  $$input|(5, 18) = p_1 : 0, p_2 : 0$$
  etc. etc. where the tuple is the time window and the dictionary the vector with parameters.

Data Retrieval:
The method `__getData__` returns the simulation data object (whether it is an instance of `AmiciDataObject`, `PythonDataObject`, or `TelluriumDataObject`), which can then be used for further analysis or visualization.

8.4 Conclusion

In summary, `SolveSystem.py` provides a versatile framework for solving differential equation models:

- It encapsulates simulation results through specialized data object classes.
- It supports multiple simulation backends, automatically choosing between compiled AMICI modules, Tellurium-based simulation, or Python LSODA integration.
- It handles forward sensitivity analysis and time-dependent parameter updates.

Note that AMICI updated their library since and spline inputs are possible. Therefore a start-stop approach is not strictly required any longer.

9 Measurements.py

Holds classes such as `Observable` and `MeasurementObject`, linking empirical data or synthetic results with the model states for scoring and display.

9.1 Overview and Dependencies

The module imports several libraries and modules:

- **Numerical and Scientific Libraries:** `numpy`, `scipy.integrate`, `math`, and `random`.
- **Experimental Design and Plotting:** `pyDOE` (for design of experiments), `matplotlib.pyplot`, and `seaborn` for data visualization.
- **System Utilities:** `os` for file and directory management.
- **Custom Modules:** `DataTransform` and `Scoring` for data manipulation and performance scoring.

The module defines classes to store measurement data, functions to simulate measurements using model simulations, and a class to generate and optimize experimental conditions.

9.2 Class `Observable`

This class encapsulates a single observable measurement.

Attributes:

- `name`: The species name.
- `time`: A vector of timepoints (converted to integers) at which measurements were taken.
- `data`: The corresponding concentration values.
- `unit`: Time unit (default is "min").

This class is used to represent a time series measurement:

$$\text{Observable}: \quad \{(t_i, y_i)\}_{i=1}^{N}$$

where t_i is the time and y_i the measured concentration at that time.

9.3 Class `TimeDependentInputs`

This class is similar to `Observable` but is tailored for time-dependent inputs (e.g., control signals).

Attributes:

- `name`: The species or input name.
- `time`: Extracted from the provided dictionary (typically under the key "Time").
- `stock`: The identifier for the stock solution.
- `stock_concentration`: A dictionary to hold stock concentration data.
- `data`: The concentration or input values over time.

9.4 Function `simulate_measurement`

This function simulates experimental measurements from a given model. It performs the following steps:

1. **Parameter and Initial Condition Setup:** Creates a `ModelVariables` instance (from the `Model` module) using provided conditions and parameter modifications.
2. **Time-Dependent Parameters:** If coordinates for time-dependent inputs are provided, the function updates the corresponding inputs via `set_time_dependent_variables`.
3. **Model Simulation:** A `ModelSolver` (from `SolveSystem`) is instantiated to solve the model equations,
$$\frac{dy}{dt} = f(t, y, p)$$
with the given time span and step size.
4. **Data Extraction:** Simulation results (state trajectories and time series) are stored and mapped to observables. If `model.observables` is empty, the states are used.
5. **Measurement Object Construction:** A `MeasurementObject` is created using the simulated data, conditions, lumping information, and other parameters.

9.5 Class `MeasurementObject`

This class stores experimental measurement data and related metadata.

Attributes:

- `rawdata`: The original measurement data (e.g., from an Excel sheet or simulation).
- `measurement_time`: The time vector over which data was collected.
- `time_dependent_parameters`: Inputs that vary over time (if any).
- `conditions` and `initial_conditions`: Dictionaries that hold experimental conditions.
- `profile`: A dictionary mapping each observable to its time series data. This may be computed from rawdata using interpolation:
$$\text{Profile}(s) = \{(t_i, y_i)\}_{i=1}^{N}, \quad s \in \text{Observables}$$
- `observables`: The list of measured species (keys from `profile`).
- `lumped`: Groups of observables that are treated as a single measurement.
- `directory`: The path where simulated experiment figures are stored.

Methods:

- `prune(prune)`: Removes parts of the data with calibration errors over a specified time range.
- `show(...)`: Plots the measurement data and experimental conditions. In the plots:
  - Left subplot: Time series of concentrations versus time.
  - Right subplot: Bar charts displaying control parameters (conditions).

9.6 Class `GenerateExperiments`

This class is responsible for generating simulated experimental designs and optimizing the set of experiments.

Key Features:

- **Global Search for Conditions:** Uses functions such as `build_global_search` (from `DataTransform`) to sample experimental conditions from the model's parameter space.
- **Simulation of Multiple Experiments:** For each set of conditions, `simulate_measurement` is called to create a corresponding `MeasurementObject`.
- **Least-Squares (LSQ) Analysis:** The module computes a LSQ error defined as

  $$\text{LSQ} = \sum_i (y_i - \bar{y}_i)$$

  where y_i is the simulated observable and $\bar{y}_i$ is the mean over experiments.
- **Selection and Optimization:** Based on LSQ distances, the experiments are ranked, and subsets (or supersets) of measurements are selected for as synthetic data. Methods for specific types of data include:
  - `return_single_measurement`: Returns a randomly selected measurement (no time dependent input).
  - `return_random_measurents`: Returns a set of random measurements, i.e. multiple measurement objects (no time dependent input).
  - `return_multiplexed_measurements`: Returns a multiplexed set of experiments.
  - `return_oscillating_measurement` and `return_random_pulse_measurement`: Generate measurements with oscillatory or pulsed time-dependent inputs.
  - `optimized_pulse_measurement`: Uses an optimization routine (via `OptimizeExperiment`) to select the most informative pulse sequence.

9.7 Conclusion

In summary, `Measurements.py` provides a framework for:

- Representing raw experimental data and time-dependent inputs to be used by `ModelObject`
- Generating, selecting, and optimizing experimental designs with the `GenerateExperiments` class.

These tools are essential for model calibration, least-squares fitting.

10 ParameterOptimizationModelSets.py

Implements multi-agent or evolutionary parameter fitting. The main class `Optimization` drives the iterative process, while `Agent` and `Convergence` track individual and population-level progress.

10.1 Overview

This module implements an evolutionary optimization algorithm that fits parameters for N models to M measurements simultaneously. When multiple models share parameters, the algorithm forces these parameters to be equal across models. It uses evolutionary operators—mutation, micro-mutation, and recombination—to explore a combined parameter space and selects the candidate with the lowest deviation between simulation and measurement. The fitness is typically computed via a score such as the standard deviation or least-squares error:

$$\mathrm{SD} = \sqrt{\frac{1}{T}\sum_{t}\left(y(t;p) - y_{\mathrm{meas}}(t)\right)^2}.$$

10.2 Dependencies and Modules

The code relies on several custom and external modules:

- **Custom Modules:** `OptimizationOperations`, `Scoring`, `Operations`, `Distributions`, `DataTransform`, `PlotData`. These modules supply helper functions for computing deviation scores, mutation ranges, global search space generation, and plotting.
- **External Libraries:** `pandas`, `seaborn`, `matplotlib`, `statsmodels`, `scipy`, `numpy`, `math`, `random`, `copy`, `os`, `time`.

10.3 Mathematical Formulation

Let $y(t;p)$ denote the solution of the model's ODEs with parameter vector p, and let $y_{\mathrm{meas}}(t)$ be the measurement. The optimization objective is:

$$p^* = \arg\min_{p\in\mathcal{P}} \mathrm{SD}(p),$$

where the score (or deviation) is calculated as:

$$\mathrm{SD}(p) = \sqrt{\frac{1}{T}\sum_{t}\left(y(t;p) - y_{\mathrm{meas}}(t)\right)^2}.$$

The combined parameter space $\mathcal{P}$ is built from the union of the parameter spaces of all models, with shared parameters constrained to have identical values.

10.4 Pseudocode

The core evolutionary algorithm follows this pseudocode:

```
Input: Measurements, combined parameter space p_space, models, and list of parameters to optim
Output: Best parameter set p* and convergence history

// Combine parameter space from all models
p_space_combined = Union(model.p_space for each model in models)

// INITIALIZATION
if no startset provided:
    ipos = GlobalSearch(p_space_combined, include, samples=startsamples)
    ipos = SelectOptimalPositions(ipos)  // via multimodalstart
else:
    ipos = Process(startset)

Create agents:
for each agent from 1 to num_agents:
    agent.position <- Recombine(ipos)  // initial candidate parameter set
    Initialize Agent(measurements, p_space_combined, include, position)

for each agent:
    (score, simdata, fwd) = simulate_measurements(measurements, agent.position)
    agent.initial_height(score, simdata)

Initialize convergence object with a very high initial score

// EVOLUTIONARY LOOP
for it = 1 to generations:
    for each agent:
        With probability 0.25:
            agent.update_position(Recombine(current_positions), it)
        With probability p_random_mutation:
            agent.random_mutation(it)
        else:
            agent.micromutation(it)

        (score, simdata, forward_vector) = simulate_measurements(measurements, agent.position)
        agent.update_height(score, simdata, it)
        if agent.forward:
            agent.update_mutation_probability(forward_vector, it)
        agent.convergence_track()

    (terminate, agents, converge_history) = convergence.update_convergence(agents, it)
    if terminate and it > minimum_iterations:
        break
    current_positions = SelectPositions(agents based on ranking)

Best_agent = Agent with lowest score among all agents
Output: Best_agent.position, convergence history, and simulation data
```

10.5 Class Descriptions

10.5.1 Convergence

Purpose: Tracks the best (lowest) score encountered during optimization.

Attributes:

- `lowest_SD`: Current lowest score.

- `simdata, iteration, position, coordinates, scores, converge`: Lists for storing history.
- `update_count`: Counter for updates.

Method: `update_convergence(agents, it, threshold)` Updates the convergence record by:

- Extracting scores from agents.
- Updating history if a new minimum is found.
- Implementing termination criteria (e.g., no improvement over 20 generations, or average deviation below a threshold).
- Reinitializing agents that are significantly worse than the best.

10.5.2 Agent

Purpose: Represents a candidate parameter set and encapsulates its evolutionary behavior.

Key Attributes:

- `measurements, models`: Input measurement data and associated models.
- `boundaries, observables, control, fixed`: Model-derived metadata.
- `p_space`: Dictionary defining allowed parameter values.
- `position` and `coordinates`: Current parameter values (both as numerical values and discrete indices).
- `height`: Current fitness (score).
- `track` and `coordinate_track`: History of parameter positions.
- `accepted_movements`: Records of accepted improvements.
- `forward_sensitivity`: Sensitivity data used to adjust mutation probabilities.

Key Methods:

- `initial_height(scores, simdata)`: Initializes the agent's fitness.
- `update_position(coordinates, it)`: Updates the agent's parameter set and records the history.
- `random_mutation(it)` and `micromutation(it)`: Apply mutations to explore the parameter space.
- `update_mutation_probability(sensitivities, it)`: Adjusts mutation probability based on sensitivity analysis.
- `update_height(scores, simdata, it, sf)`: Updates fitness and accepts new positions if they yield a lower score.
- `convergence_track(threshold)`: Checks if the agent's recent movements indicate stagnation.

10.5.3 Optimization

Purpose: Controls the evolutionary loop that updates agents over several generations.
Workflow:

1. Build a combined parameter space from all models.
2. Generate initial candidate positions (using global search and multimodal start).
3. Create a set of agents initialized with these positions.
4. For each generation:
    - Update each agent using recombination and mutation.
    - Simulate the models with the agent's parameters and update fitness.
    - Record convergence statistics using the `Convergence` object.
5. Terminate early if convergence criteria are met.
6. Output the fittest agent's parameter set and overall convergence history.

Additional Features:

- Generation of a GIF showing the evolution of the parameter fit.
- Visualization of simulation fits compared to measurements.

10.5.4 BoxplotOptimization

Purpose: Performs multiple independent optimizations and aggregates the results.
Features:

- Runs several optimizations and collects the best parameter sets.
- Generates visual summaries (e.g., boxplots, density plots) to assess the distribution of optimized parameters.
- Saves results to files for later analysis.

10.6 Usage and Output

To use the optimization algorithm:

1. Prepare the measurement data and corresponding models.
2. Define the parameters to be optimized (the `include` list) and the combined parameter space (`p_space`).
3. Instantiate the `Optimization` class with desired settings (number of generations, agents, simulation time, etc.).
4. Run the optimization loop; the algorithm simulates each agent's performance and updates the parameter vectors.
5. The fittest agent (with the lowest score) is returned as the optimal parameter set.
6. Optionally, use `BoxplotOptimization` to run multiple optimizations and produce visual analyses.

10.7 Conclusion

This module implements an evolutionary algorithm for simultaneously optimizing parameters of multiple models against multiple measurements. By combining recombination, random mutations, and sensitivity-driven micro-mutations, the algorithm efficiently searches the high-dimensional parameter space. The convergence criteria and visualization tools (including GIF generation and boxplots) provide insights into the quality and robustness of the parameter estimates, ensuring that shared parameters are optimized consistently across all models. Note using the forward sensitivities to turn the algorithm into a pseudo-greedy method is very costly from a computational perspective in this workflow, if the system can converge, do not use!

11 OptimizeExperiment.py

Focuses on designing experiments by maximizing some measure of information (Fisher, collinearity, *etc.*). Features specialized `Convergence` and `ExperimentAgent` classes. Note that the algorithm is identical to the `ParameterOptimizationModelSets.py` algorithm to train a model.

12 TransformModeltoSBML.py

Contains functions to parse textual reaction notations, build stoichiometric matrices, and produce a valid SBML representation for AMICI compilation. Direct translations are cumbersome therefore we utilize antimony as an intermediate format (used by tellurium) which then converts to SBML i.e. our text based ODEs are converted to antimony, then to SBML, then the AMICI.

12.1 Overview

This script converts a text-based model into various formats such as Antimony and SBML. It parses a user-defined model (including its states, reactions, and parameter definitions) and processes it to create a structured representation. The output can be stored as an SBML file, and the script also provides utilities for computing rate laws, creating stoichiometric matrices, and handling experimental conditions. In mathematical terms, given a model defined by reaction expressions, the goal is to generate a structured format where each reaction is expressed as

$$\texttt{Reactants} \xrightarrow{k_i} \texttt{Products},$$

with states replaced by indexed variables y_i and parameters by k_i.

Dependencies and Imports

The script imports several standard and custom modules:

- **Numerical and File Handling:** `numpy`, `os`, `pickle`.
- **Plotting:** `matplotlib.pylab`.
- **Custom Functions:** Functions from the module `Operations` (which include helper routines for string manipulation and model processing).
- (Optionally) `tellurium` is used (commented out in parts) for converting Antimony strings to SBML.

12.2 Mathematical Formulation of the Conversion

The conversion process involves:

1. **Parsing the Reaction String:** The model is provided as a text string that contains reaction rate expressions. The script locates symbols such as “+” and “-” to split the string into individual reaction rate laws.

2. **Mapping States and Parameters:** States are mapped to indexed variables y_i and parameters to k_i based on a predefined mapping. This is crucial to maintain consistency during conversion.

3. **Building a Stoichiometric Matrix:** The function `model_matrix_format` constructs a stoichiometric matrix S where each element S_{ij} indicates the contribution of flux j to state i:

$$S_{ij} = \begin{cases} +1, & \text{if the reaction contributes positively to state } i, \\ -1, & \text{if negatively,} \\ 0, & \text{otherwise.} \end{cases}$$

4. **Generating the Antimony String:** Using the parsed reaction fluxes and the stoichiometric information, the function `antimony_parser` constructs a model string in Antimony format. Delimiters and operators are standardized (for example, replacing “`**`” with “$'', and custom delimiters like$“`|+|`$'' with$“`+`$''$).

12.3 Pseudocode of the Conversion Process

```
Input:
    - Text-based model string with reaction equations.
    - Lists of states and parameters.
    - Fixed parameter values and mapping (model.map).

Process:
1. For each reaction string in the model:
    a. Identify positions of '+' and '-' (using find_elements).
    b. Split the string into individual rate laws (define_ratelaw).

2. For each rate law:
    a. Create a RateLaw object:
       - Determine which states and parameters appear.
       - Record indices based on the states list.

3. Construct the model matrix:
    a. Call model_matrix_format with the split model, states, and fixed parameters.
    b. Obtain:
          - Fluxes: Unique reaction identifiers.
          - Stoichiometric matrix S.
          - Mapping of rate laws to reactions.

4. Generate the Antimony model:
    a. Use antimony_parser with:
          - States, parameters, fluxes, S, and rate law data.
    b. Replace state and parameter names by their indexed representations (y{i}, k{i}).
    c. Standardize the expression format (adjust operators, delimiters).

5. (Optional) Convert the Antimony string to SBML:
    a. Use tellurium's antimonyToSBML function.
    b. Store the resulting SBML file (storeSBML).

Output:
    - A properly formatted Antimony model string.
    - An SBML file stored on disk.
```

12.4 Detailed Function Descriptions

12.4.1 RateLaw Class

Purpose: Encapsulates a reaction rate expression. It scans the provided rate string for state and parameter names.

Key Steps:

- Store the original rate string.
- Loop through the provided states and parameters:

$$\text{if state/parameter} \in \text{rate then add to list.}$$

- Record the index positions of the states.

experimental_conditions Class

Purpose: Processes a dictionary of conditions to separate those acting as initial conditions for states and those acting as control parameters (boundaries).

Key Steps:

- Filter out any condition keys that are not in the model's states or boundaries.
- Construct two dictionaries: one for initial state conditions and one for control parameters.
- Handle special cases when control parameters are provided as part of the initial conditions.

12.4.2 find_elements and define_ratelaw Functions

Purpose: These functions help in parsing the text-based model:

- `find_elements`: Finds positions in a string where a given character ("+" or "-") occurs followed by a space.
- `define_ratelaw`: Uses the indices from `find_elements` to split the model string into individual reaction rate segments.

12.4.3 model_matrix_format Function

Purpose: Generates the stoichiometric matrix S and identifies fluxes from the parsed model equations.

Key Steps:

- For each reaction (corresponding to a state), extract the list of rate laws.
- Determine the sign (positive or negative) and map the reaction to its corresponding state.
- Build the stoichiometric matrix S where:

$$S_{ij} = \begin{cases} -1, & \text{if the reaction decreases state } i, \\ +1, & \text{if it increases state } i, \\ 0, & \text{otherwise.} \end{cases}$$

- Return the unique fluxes, the matrix S, a dictionary of RateLaw objects, and a mapping of state rates.

12.4.4 antimony_parser Function

Purpose: Converts the processed model information into an Antimony formatted string.

Key Steps:

- Determine the nonzero entries of S and construct connections between states and fluxes.
- For each connection, generate a reaction string in the form:

```
J#: Reactant -> Product ; flux
```

- Concatenate all reaction strings and add initial conditions for states and parameter assignments.
- Replace original state and parameter names with indexed identifiers y_i and k_i using the mapping.
- Replace non-standard delimiters (e.g., replace "`|+|`" with "`+`").

12.4.5 Store and Conversion Functions

storeSBML Function: Saves the SBML file to a designated folder (e.g., on the Desktop).

directSBMLConversion Function: Uses Tellurium's conversion tool to transform an Antimony string into SBML and then calls `storeSBML`.

SBMLConversion Function: Iterates over multiple models, converts each to the Antimony format using `model_matrix_format` and `antimony_parser`, applies condition settings, converts to SBML, and stores the file.

convertModeltoSBML Function: Converts a single model into SBML following similar steps as above.

12.5 Conclusion

This conversion script takes a text-based model and processes it into multiple standardized formats. It parses reaction strings to extract rate laws, builds a stoichiometric matrix to represent the network, and then generates an Antimony model string with indexed state and parameter names. Finally, by utilizing Tellurium's conversion functions, the script produces an SBML file that can be stored and further used for simulations or analysis.

13 Typical Usage Flow

A short pseudo-code example, note that flux terms are marked by two brackets for + and - i.e. the ModelObject parses the ODEs into a matrix format where dx/dt = S*V:

```
from Model import ModelObject, ModelVariables
from SolveSystem import ModelSolver
from Measurements import MeasurementObject
from ParameterOptimizationModelSets import Optimization

# 1. Create a model
my_model = ModelObject(
  model=" + s0*k1 - s0*k2; ",
  states=["S0","S1","S2"],
  boundaries={"k1":(1e-4,1e-1), "k2":(1e-5,1e-2)},
  ...
)
my_model.SBMLconversion()
my_model.PytoCompile()

# 2. Run a simulation
vars_init = ModelVariables(my_model, conditions={"S0":1.0}, modification={"k1":5e-3,"k2":1e-3}
solver = ModelSolver(my_model, variables=vars_init, simtime=(0,50), dt=1)
data_obj = solver.__getData__()

# 3. Create a MeasurementObject
measurement = MeasurementObject(
  rawdata=...,
  time=data_obj.time,
  name="Experiment1"
)
measurement_1.model = my_model_1
measurement_2.model = my_model_2
measurement_3.model = my_model_1  #repitition possible
.
measurement_N.model = my_model_N

# 4. Optimize parameters
opt = Optimization(
  measurements=[measurement_1,...],
  include=["k1","k2"],
  generations=100,
  agents=10
)
opt.run()
best_solution = opt.convergence.position[-1]
```

14 ModelBuilder used to automatically construct ODEs

This project provides an automated framework for constructing ordinary differential equation (ODE) models for biochemical reaction networks. The framework allows users to define reaction kinetics, enzyme interactions, and reactor conditions, and automatically generates corresponding mathematical equations. The generated models can be compiled into SBML format and further processed using AMICI for numerical simulation.

14.1 Features

- Automated generation of biochemical reaction models.
- Supports multiple kinetic mechanisms:
  - Michaelis-Menten (MM)
  - Generalized Hill (GH)
  - Rapid Equilibrium Random (RER)
  - Steady-State Ordered (SSO)
  - Equilibrium Ordered (EO)
- Allows explicit modeling of enzyme and substrate inflow in reactors.
- Provides different reactor types:
  - Predefined control (staged reactor)
  - Predefined control (single reactor)
  - Individual control
  - Uniform control
- Generates and organizes parameters for efficient numerical optimization.

Overview

This document summarizes the core mathematical formulations used within the reactor and reaction classes. The **ReactorKinetics** class constructs mass balance equations for a biochemical reactor by defining inflow and outflow terms under various control schemes. The **ReactionKinetics** class formulates reaction rate equations for different kinetic models (e.g., mass action, Michaelis–Menten, Generalized Hill, MWC, etc.), including the incorporation of allosteric activation and inhibition.

14.2 Define a Reaction Network

The reaction network is defined as a list of reactions, where each reaction follows the structure:

```
network = [
["HK", True, ["Glucose", "ATP"], ["ADP", "G6P"], [], []],
["G6PDH", True, ["G6P", "NADP"], ["6PGL", "NADPH"], [], []],
]
```

14.3 Construct Reaction Equations

Use the categorize reactions function to convert the reaction network into a structured format:

```
reaction_network = categorize_reactions(network)
```

14.4 Select Reactor Type

Define how species flow into the reactor using different control strategies:

```
reactor_kinetics = ReactorKinetics(states)
reactor_kinetics.reactor_individual_control()
```

14.5 Generate Model Combinations

```
constructor_modelcombinations(reactor_kinetics, reaction_kinetics)
```

14.6 Convert Model to SBML

The generated model can be converted to SBML and compiled using AMICI:

```
model.AntimonyConversion()
model.SBMLconversion()
model.PytoCompile()
```

14.7 BuildModelModule.py

This file contains two core classes:

- **ReactorKinetics**: Defines inflow and outflow terms for species in different reactor configurations.
- **ReactionKinetics**: Defines reaction rates and kinetic terms for each biochemical reaction.

14.8 BaseModel Class

The baseModel class integrates reaction equations and reactor conditions to generate a fully defined ODE model. It also:

- Organizes kinetic and control parameters.
- Defines states and flux terms.
- Converts the model into SBML and C++ formats.

15 Example Workflow

```
enzymes_TOY_glycolysis = [
                  ['Ex_glc' , False, 'MM', ['Glucose'],['Glucose_D'],[],[]],
                  ['HEX1'   , False, 'GH', ['Glucose_D','ATP'],['ADP','G6P'],[],[]],
                  ['pts'    , False, 'GH', ['PEP','Glucose_D'],['Pyruvate','G6P'],[],[]],
                  ]


"""the modelbuilder scripts"""
from ModelBuilderOperations import *

"""Enzyme All Enzymes model"""
network      = glycolysis
modelname    = 'ModelBuilderTest'

"""Syringe load"""
syringe_load = {}

"""Get the reactions and categorize them"""
reactions = categorize_reactions(network)

"""construct the flux terms i.e. if you define the system as:
                 dx/dt = S*V where V contains all the fluxes"""
reactor_kinetics,reaction_kinetics = construct_model_fluxterms(reactions)

"""Construct the model: This is the kinetics of:
                        the reactor + the kinetics of the reactions!"""
basemodel = construct_kinetic_combinations
(reactor_kinetics,reaction_kinetics,name = modelname,reactor_type = 'reactor_predefined_contro

"""Get the base model to work"""
model = basemodel.return_model()
```

16 Installation

Ensure Python is installed on your system. Additionally, install the required dependencies. AMICI is the toughest to compile in Microsoft Windows. It requires a specific compiler and you need to define the path that links to that compiles in your environment variables. For more information please see AMICIs install instructions https://amici.readthedocs.io/en/latest/about.html To use the code described, download the ZIP and open the main demo file " OED interface.py" in the editor of choice (pycharm obviously). The code does not have any particular system requirements. The average runtime on a single CPU for the optimal design depends on the size of the model. Approximately a few hours for a system with a thousand equations (i.e. default system of ODEs and forward sensitivity equations).

17 DEMO

This is a demonstration on how the code can be used in a type of working file. How the modules can be called. The model can be compiled, how data can be generated (synthetic data). Finally how multiple models can be fit to the data and new experiments can be designed. Below you will find the pseudo code with a comprehensive description of each step. If you want to use real data yourself, please make sure you parse it into the MeasurementObject, which can subsequently be given to the relevant algorithms as an input. This demo script can be found in the OED folder on the github and is called "OED interface.py". Note make sure the required packages are installed.

```
"""the libraries that are needed"""
import math
import copy
import numpy
import time as timing
import matplotlib.pylab as plt

# STEP 1) import the models in the model module
"""import the model"""
import __NucleotideSalvagePathway__ as model
models, control = model.main()

# STEP 2) compile the models from the model module
"""please ensure you have installed AMICI, tellurium, and libSBML"""
for i in range(len(models)):
    model = models[i]
    model.SBMLconversion()
    model.PytoCompile()

# STEP 3) Define the individual models
"""Load multiple models simultaneously, if needed"""
NSP_model = models[0]

# STEP 4) Import experiment generator
from Measurements import GenerateExperiments
m = GenerateExperiments(NSP_model)

# STEP 5) Design an experiment
measurement = m.return_random_pulse_measurement(
    store=False, name='RATE_ID_TEST', time=(0, 12*60), plength=20)
measurement.show(show=True)

# STEP 6) Import and configure optimization module
from ParameterOptimizationModelSets import BoxplotOptimization
measurement.model = NSP_model
```

```
# STEP 7) Train models and optimize parameters
directory = "[PATH]"  # Folder where estimates are stored
BoxplotOptimization(
    {0: measurement}, optimization_number=50, generations=150, agents=5,
    storedata=directory, startsamples=100)

For more information see
https://www.nature.com/articles/s41467-022-31306-3
https://www.nature.com/articles/s41467-024-45886-9

#--------------------------------------#
The model __NucleotideSalvagePathway__
#--------------------------------------#
This model can be substituted for any other model also but it is always imported
from a seperate file. In this way, parsing, compilation etc. is done in the background
so just define your own model file with your ODE and the required parameters and
metaparameters in the a seperate [MODEL NAME].py file and follow the format below

from Model import ModelObject
def main():
    models = {}
    control = {}

    # Define the model description
    description = "Phosphorylation model"

    """Define parameters in the model"""
    kinetic_parameters = [
        'k_cat13_AK_ADP', 'K_M13_ADP_AK', 'k_cat14_AK_AMP_ATP', 'K_M14_AMP_AK',
        'K_M14_ATP_AK', 'k_cat15_PK_ADP_PEP', 'K_M15_ADP_PK', 'K_M15_PEP_PK',
        'k_cat16_PK_PEP_UDP', 'K_M16_PEP_PK', 'K_M16_UDP_PK', 'k_cat17_PK_GDP_PEP',
        'K_M17_GDP_PK', 'K_M17_PEP_PK', 'k_cat18_GMPK_ATP_GMP', 'K_M18_ATP_GMPK',
        'K_M18_GMP_GMPK', 'k_cat19_UMPK_ATP_UMP', 'K_M19_ATP_UMPK', 'K_M19_UMP_UMPK',
        'k_cat21_UPRT_PRPP_Ura', 'K_M21_PRPP_UPRT', 'K_M21_Ura_UPRT', 'k_cat22_APRT_Ade_PRPP',
        'K_M22_Ade_APRT', 'K_M22_PRPP_APRT', 'k_cat26_PK_ATP_Pyr', 'K_M26_ATP_PK',
        'K_M26_Pyr_PK', 'k_cat27_PK_Pyr_UTP', 'K_M27_Pyr_PK', 'K_M27_UTP_PK',
        'k_cat28_PK_GTP_Pyr', 'K_M28_GTP_PK', 'K_M28_Pyr_PK', 'k_cat29_GMPK_ADP_GDP',
        'K_M29_ADP_GMPK', 'K_M29_GDP_GMPK', 'k_cat30_UMPK_ADP_UDP', 'K_M30_ADP_UMPK',
        'K_M30_UDP_UMPK'
    ]

    enzymes = ['PK', 'UMPK', 'UPRT', 'GMPK', 'APRT', 'AK']
    kinetic_parameters.extend(enzymes)

    """Define control parameters"""
    control_parameters = ['Ade_in', 'Gua_in', 'PEP_in', 'PRPP_in', 'Ura_in', 'ATP_in']

    """Initialize fixed parameters with default values"""
```

```
fixed = {k: 1 for k in kinetic_parameters}
fixed.update({c: 10 for c in control_parameters})
fixed.update({c: 0 for c in [
    'ADP_in', 'AMP_in', 'ATP_in', 'Ade_in', 'GDP_in', 'GMP_in', 'GTP_in',
    'Gua_in', 'PEP_in', 'PPi_in', 'PRPP_in', 'Pyr_in', 'UDP_in', 'UMP_in',
    'UTP_in', 'Ura_in'
]})
fixed["kf"] = 0.125

"""Define parameter boundaries"""
boundaries = {
    'Ade_in': (0.01, 10), 'Gua_in': (0.01, 1), 'PEP_in': (0.01, 10),
    'PRPP_in': (0.1, 10), 'Ura_in': (0.01, 1), 'ATP_in': (0.01, 1)
}
boundaries.update({k: (0.001, 50) for k in fixed if k not in control_parameters})
boundaries["kf"] = (0.01, 0.25)

"""Define the system equations"""
stringmodel = '''
    -2*ADP**2*AK*k_cat13_AK_ADP/(K_M13_ADP_AK*(ADP/K_M13_ADP_AK |+| 1))
    ... etc etc etc.
'''

"""Format model equations"""
symbols = ['-', '+', ')', '(', '*', '**', '|+|', "|-|", "/", '\n']
replacements = {('| + |', '|+|'), ('| - |', '|-|'), ("*  *", "**")}

for s in symbols:
    stringmodel = stringmodel.replace(s, " " + s + " ")
for old, new in replacements:
    stringmodel = stringmodel.replace(old, new)

print(stringmodel)

"""Define model states and observables"""
states = ['ADP', 'AMP', 'ATP', 'Ade', 'GDP', 'GMP',
          'GTP', 'PEP', 'PPi', 'PRPP', 'Pyr', 'UDP',
          'UMP', 'UTP', 'Ura']

observables = ['ADP', 'ATP', 'Ade', 'GDP', 'GMP', 'GTP', 'UDP', 'UTP']

"""Define optimization conditions"""
conditions = {e: 0.1 for e in enzymes}
conditions.update({c: 10 for c in control_parameters})
conditions["kf"] = 0.125

"""Set up model storage and return"""
modelname = 'NSP_model'
```

```
    models[len(models)] = ModelObject(
        stringmodel, states, boundaries, fixed, observables=observables,
        name=modelname, control_parameters=control_parameters
    )
    control[len(control)] = optimization_information(conditions=conditions)
    return models, control
```

18 License

This project is open-source under the MIT license and available for use and modification. Proper attribution to the original authors is appreciated.