# ProBioSim v0.1

Matthew R. Lakin*[1,2,3]

[1]Department of Computer Science, University of New Mexico, USA
[2]Department of Chemical & Biological Engineering, University of New Mexico, USA
[3]Center for Biomedical Engineering, University of New Mexico, USA

July 27, 2021

## Contents

*Email: `mlakin@cs.unm.edu`

# 1 Introduction

## 1.1 About

ProBioSim is a simulator for mass-action chemical reaction networks, written in the Python programming language (specifically, Python3). It includes both deterministic (ODE-based) and stochastic simulation engines.

Most notably, ProBioSim provides powerful features for modifying the state of the simulation at specified timepoints, by specifying these modifications as arbitrary Python functions. This enables state- and time-dependent modifications to the simulation, which can be used to model external interventions in the evolution of the system by a responsive internal environment.

Models can be specified in ProBioSim using either a text-based interface (to load model definitions from an external file or a multi-line Python string) or programmatically by calling methods to add new elements to a model object. This enables models to be rapidly prototyped using the text-based interface before potentially moving to a more structured approach in which model elements can be added programmatically, e.g., to automate the generation of classes of models.

## 1.2 Installation instructions

To install, simply extract the source files into a folder of your choice, then add that folder to your `$PYTHONPATH` environment variable. This will enable Python to find the files when you `import` them into your own Python code.

## 1.3 Dependencies

In addition to the Python standard library, the following additional libraries are either required or optional for use with ProBioSim:

- `numpy`: **required** [2].
- `scipy`: **required** [5].
- `matplotlib`: optional (only needed to run the plotting parts of the supplied examples) [3].
- `openpyxl`: optional (only needed to save simulation results to an Excel file).
- `pandas`: optional (only needed to save simulation results to a `DataFrame`) [4].

If you do not have the optional libraries installed, the code will still run, but the corresponding functionality will either be skipped over (in the case of plotting the results of the examples) or an error will be raise (in the cases of the output functionalities to Excel or to Pandas `DataFrames`).

# 2 Simulation algorithms

## 2.1 Deterministic simulations

Deterministic simulations are implemented by forming an ordinary differential equation (ODE) representation of the mass-action system kinetics. Briefly, for each species $X_i$, the corresponding ODE is given by:

$$\frac{d[X_i]}{dt} = \sum_j (stoich(X_i, r_j) \times rate(r_j))$$

where the $r_j$ are the reactions included in the model, the $stoich(X_i, r_j)$ function returns the stoichiometry (net change) in $X_i$ from one occurrence of reaction $r_j$, and the $rate(r_j)$ function returns the rate constant value associated with reaction $r_j$. The system of such mass-action ODEs for all species $X_i$ in the model is formed and then integrated over time using the `solve_ivp` function from the `scipy.integrate` module [5]. The user can choose between using a stiff solver, which uses the ``LSODA'' method, and a non-stiff solver, which uses the ``RK45'' Runge-Kutta method.

## 2.2 Stochastic simulations

Stochastic simulations are implemented using Gillespie's direct method [1], which provides an exact implementation of stochastic mass-action kinetics. Briefly, for each reaction $r_j$, the counts of all reactant species are multiplied together and by the reaction's rate constant $rate(r_j)$ to calculate the propensity $\rho_j$ of that reaction. The next reaction is chosen at random, with the probability of choosing $r_j$ given by $\rho_j / (\sum_j \rho_j)$, that is, with probability proportional to the corresponding reaction propensity. The *time* until the next reaction fires is drawn from an exponential distribution with mean $1/(\sum_j \rho_j)$. The reaction is applied to update the state, the propensities are recalculated, and the simulation loop restarts.

## 3 Specification and semantics of perturbations

There are two types of perturbation in ProBioSim: perturbation *actions* and perturbation *functions*. In each case, the perturbation is scheduled to occur at a specific timepoint in the simulation, at which point the simulation (either stochastic or deterministic) is paused, the specified changes to the simulation state are applied, and the simulation is restarted. Multiple perturbation actions and functions can be specified to execute at the same timepoint. In this section, we outline the similarities and differences between perturbation actions and perturbation functions, and how they are applied when both are set to occur simultaneously.

### 3.1 Perturbation actions

Perturbation actions are explicitly specified modifications to the state of the simulation. As outlined below, they can be specified within the ASCII CRN specification language included within ProBioSim, and can be exported to and imported from text files that specify the model. The set of perturbation actions that can be specified in this way is limited to the following:

- *increment* the concentration or count of a species by a specified amount,
- *decrement* the concentration or count of a species by a specified amount, or
- *set* the concentration or count of a species to a specified amount.

As an example, a perturbation that adds 10 units of species `X` at simulation time 5.0 would be specified by the following line in the model definition input:

```
perturbation X += 10 @ 5.0
```

See Section 4 for the full syntax for specifying each of these kinds of perturbation action.

### 3.2 Perturbation functions

Perturbation functions are modifications to the simulation state that are specified in terms of functions written in Python, the host language of ProBioSim. As such, perturbation functions are significantly more powerful than perturbation actions as they allow arbitrary computations to be carried out to determine how the state of the system should be updated. ProBioSim itself does not check the content of user-supplied perturbation functions. However, to abstract any from internal implementation details of the simulator without impacting the expressiveness of perturbation functions, those perturbation functions must be written in a particular way to enable them to be called by the simulator as required. Specifically, the form of a perturbation function must be as follows:

```
def pfun(t, x0, get, set, adjust):
    # ... perturbation code here ...
```

where the meanings of the function arguments `t`, `x0`, `get`, `set`, `adjust` are as follows:

- `t` is the simulation time at which the function is called;
- `x0` is the simulation state passed into the function;
- `g` is a function that, when called as `get(x0, sp)`, returns the current value associated with species `sp` in the current state `x0`;
- `set` is a function that, when called as `set(x0, sp, n)`, updates the state `x0` by assigning the value `n` to the species `sp`; and
- `adjust` is a function that, when called as `adjust (x0, sp, n)`, updates the state `x0` by adding `n` (which could be negative) to the value currently assigned to the species `sp`.

Note that the exact names used for these arguments in the function are unimportant but the number of arguments does need to be correct. In principle, the `adjust` function can be implemented in terms of the `get` and `set` functions, but is included for the sake of convenience. These functions can be used as outlined above to get state values, perform arbitrary computations on them, and update the state *as a side-effect* in response. The time is passed to the perturbation function as an argument, meaning that perturbation functions can be made to be time-dependent. In addition, free variables in the perturbation function can be used to pass in additional information to the perturbation function; see Section 5 for an extended example. For example, the perturbation action example from above can be implemented (with some terminal output) as the following function:

```
def pfun(t, x0, get, set, adjust):
    print('>>> Value of X __before__ perturbation at time '+str(t)+' = '+str(get(x0, 'X')))
    adjust(x0, 'X', 10.0)
    print('>>> Value of X __after__ perturbation at time '+str(t)+' = '+str(get(x0, 'X')))
```

For simple examples consisting of a single function call, the Python `lambda` keyword can be used to implement simple perturbation functions. Once defined, these functions must be attached to the model object using the `addPerturbationAsCode` or `addPerturbationAsCodeMultipleFunctions` function, as outlined in Section 6.4.

### 3.3 The application of perturbations

Perturbations can be defined in any order in the input, and are organized into time order by the simulator. If a multiple perturbation actions or functions are specified at the same timepoint (or at timepoints that are indistinguishable up to the tolerance of floating-point arithmetic) then all of the perturbation *actions* will be applied to the simulation first, followed by all of the perturbation *functions*. Within these two categories, the actions and functions will be applied in the order in which they were added to the model. (Note that changing the order of addition could therefore change the overall result of the perturbations applied at a given time point.)

After the perturbation actions and functions are applied, the resulting state must be checked to ensure the integrity of the simulation. First, the resulting state of the system is checked to ensure that there are no negative values; if there are, these are set to zero (negative concentrations or species counts being unrealistic). Then, if the simulation is stochastic, any non-integer values are converted to integers, by *rounding down* if necessary. This prevents fractional species counts being recorded. The simulator is then restarted, beginning from the new state that resulted from the application of the perturbations..

Note that the simulator records two values at the simulation time, both before and after the perturbation is applied. Note also that the simulator does *not* explicitly model dilution due to the addition of species during a perturbation, based on the assumption that the volume of any samples added during the perturbation are negligible compared to the overall reaction volume. However, one could implement dilution effects within a custom perturbation function, if desired.

# 4 CRN syntax specification

In this section we specify the syntax of CRN definitions that can be parsed, and exported, by ProBioSim. With the exception of model perturbations specified as Python functions, all model aspects can be specified or exported in this manner. *The model syntax is organized on a line-by-line basis.* CRN models can be loaded from a text file or multi-line Python string using this syntax. Alternatively, strings representing single lines can be processed individually, providing a rapid way to programmatically construct models from snippets of the model representation language. When given multi-line input, lines are processed in order, and later lines will overwrite values for the same parameter set by earlier lines. **Note that the various tokens outlined above must be separated by whitespace to be correctly parsed.** The rest of this section is organized as follows: we begin by specifying the comment syntax then specify the different kinds of line that can be parsed.

### 4.1 Comments

The hash character ("`#`") delimits the start of a comment. The first appearance of this character on any line is interpreted as the start of a comment and the rest of the line is therefore discarded prior to parsing. If this character is the first (or first non-whitespace) character on a line, the line will be treated as empty and ignored altogether.

### 4.2 Valid names for species and rate constant names

Valid species names and rate constant names consist of a letter followed by zero or more letters, digits, and/or underscores. Examples of valid names include "`x`", "`Y3`", "`k_1`", "`K_2b`", and "`k3_Bwd_New`".

### 4.3 Reaction lines

Lines that specify additional reactions to be added to the model begin with the keyword "`reaction`" and must take one of the forms:

- `rection` is not used; `reaction` *`xs`* `->{`*`kf`*`} ys` — which specifies a single irreversible reaction.
- `reaction` *`xs`* `{`*`kb`*`}<->{`*`kf`*`} ys` — which specifies a reversible pair of reactions.

The variable components specified above are defined as follows:

- *`xs`* and *`ys`* are finite lists of valid species names: either empty, a single species name, or a list of two or more species names separated by "`+`", for example "`A + B`" or "`X + X + Y1`". The only restriction is that the two lists cannot be identical (considered modulo ordering).
- *`kf`* is either a valid rate constant name or valid floating point value that specifies the rate of an irreversible reaction or the "forward" reaction of a reversible reaction pair.
- *`kb`* is similarly either a valid rate constant name or valid floating point value that specifies the rate of the "backward" reaction of a reversible reaction pair.

## 4.4 Rate constant definition lines

Lines that define rate constant values begin with the keyword "`rate`" and must take the form:

- `rate` *r* `=` *n* — where *r* is a valid rate constant name, as defined above, and *n* is a valid floating-point number representing the specified value for the rate constant named *r*.

## 4.5 Species initialization lines

Lines that define initial species values begin with the keyword "`init`" and must take the form:

- `init` *x* `=` *n* — where *x* is a valid species name, as defined above, and *n* is a valid integer or floating-point number representing the initial value of the concentration of the species named *x* (or its count, in the case of a stochastic simulation).

Note that attempting to run a stochastic simulation with non-integer species initial values will result in an error being thrown.

## 4.6 Perturbation lines

Lines that define perturbation actions begin with the keyword "`perturbation`" and must take one of the following forms:

- `perturbation` *x* `=` *n* `@` *t* — where *x* is a valid species name, *n* is a valid floating-point number representing the absolute value to set species *x* to in this perturbation, and *t* is a valid floating-point number representing the time at which to apply this perturbation.
- `perturbation` *x* `+=` *n* `@` *t* — where *x* is a valid species name, *n* is a valid floating-point number representing the absolute value to add to the concentration or count of species *x* in this perturbation, and *t* is a valid floating-point number representing the time at which to apply this perturbation.
- `perturbation` *x* `-=` *n* `@` *t* — where *x* is a valid species name, *n* is a valid floating-point number representing the absolute value to subtract from the concentration or count of species *x* in this perturbation, and *t* is a valid floating-point number representing the time at which to apply this perturbation.

Lines of the first form specify an *absolute* value to assign to the species *x* in the perturbation, whereas lines of the second and third forms specify *relative* changes to value; that is, an amount to add or subtract. See Section 3 for more detail on the semantics of how perturbation are applied.

## 4.7 Simulation settings lines

Lines that specify simulation settings begin with the keyword "`simulation`" and must take one of the following forms:

- `simulation length` *n* — where *n* is a valid positive integer or floating-point number representing the desired simulation length (all simulations start from time 0).
- `simulation points` *n* — where *n* is a valid positive integer representing the desired number of points to sample in the simulation (these will be spaced evenly between time 0 and the desired simulation length).
- `simulation stiff` *b* — where *b* is either `true`, `True`, `false`, or `False` and specifies whether or not to use a stiff solver in a deterministic simulation of the model.
- `simulation seed` *n* — where *n* is either `None` or a valid integer, representing the desired pseudorandom number generator seed to use in a stochastic simulation of the model.
- `simulation rtol` *n* — where *n* is a valid floating-point number representing the desired relative tolerance parameter to use in a deterministic simulation of the model.
- `simulation atol` *n* — where *n* is a valid floating-point number representing the desired absolute tolerance parameter to use in a deterministic simulation of the model.
- `simulation recordAllIfStochastic` *b* — where *b* is either `true`, `True`, `false`, or `False` and specifies whether or not to sample the state after every reaction in a stochastic simulation of the model.

```
1  import random
2  import matplotlib.pyplot as plt
3  from psim_model import getEmptyModel
4  from psim_simulate import runStochastic
5  for simseed in [42, 99]:
6      for perturbationseed in [909693, 100000, 2468]:
7          model = getEmptyModel()
8          model.setSimulationLength(50)
9          model.setSimulationPoints(1001)
10         model.updateFromString(’reaction A ->{ka}’)
11         model.updateFromString(’reaction B ->{kb}’)
12         model.updateFromString(’reaction C ->{kc}’)
13         model.setRateConstant(’ka’, 1.0)
14         model.setRateConstant(’kb’, 0.5)
15         model.setRateConstant(’kc’, 0.25)
16         model.setSpeciesInit(’A’, 100.0)
17         model.setSpeciesInit(’B’, 50.0)
18         model.setSpeciesInit(’C’, 25.0)
19         model.setSimulationSeed(simseed)
20         perturbationRNG = random.Random(perturbationseed)
21         def rpFun(t, x0, get, set, adjust):
22             print(’>>> Applying a perturbation at time ’+str(t))
23             (sp,amt) = perturbationRNG.choice([(’A’,100.0),(’B’,50.0),(’C’,25.0)])
24             adjust(x0, sp, amt)
25         for t in [5, 10, 35, 40]:
26             model.addPerturbationAsCode(t, rpFun)
27         res = runStochastic(model)
28         outfilebase = ’results_’+str(simseed)+’_’+str(perturbationseed)
29         res.toFile(outfilebase+’.txt’)
30         plt.figure()
31         plt.plot(res.times(), res.get(’A’), ’r’, label=’A’)
32         plt.plot(res.times(), res.get(’B’), ’b’, label=’B’)
33         plt.plot(res.times(), res.get(’C’), ’g’, label=’C’)
34         plt.xlabel(’Time’)
35         plt.ylabel(’Count’)
36         plt.legend()
37         plt.title(’simseed=’+str(simseed)+’ perturbationseed=’+str(perturbationseed))
38         plt.savefig(outfilebase+’.pdf’, bbox_inches=’tight’)
39         plt.close()
```

Figure 1: Example code illustrating the creation and simulation of simple ProBioSim models involving randomized perturbations expressed as Python code, with associated plotting.

## 5 Example

Figure 1 presents example code for the creation and simulation of ProBioSim models involving randomized perturbations, and plotting of the resulting simulation results. The required imports are made on lines 1–4. For example of the seed values specified for the stochastic simulator itself (line 5) and for a separate pseudorandom number generator to be used in the perturbations (line 6), a new model is created (line 7) and basic simulation settings for the length and sampling interval are provided (lines 8–9). Degradation reactions for the species `A`, `B`, and `C` are added (lines 10–12), along with associated rate constant values (lines 13–15), and initial values for the species (lines 16–18). The seed for the stochastic simulator is set on line 19 and a separate pseudorandom number generator for use in perturbations is created with a separate seed on line 20. Lines 21–24 define a perturbation function (`rpFun`) that randomly chooses a species to perturb, weighted by the provided values, and

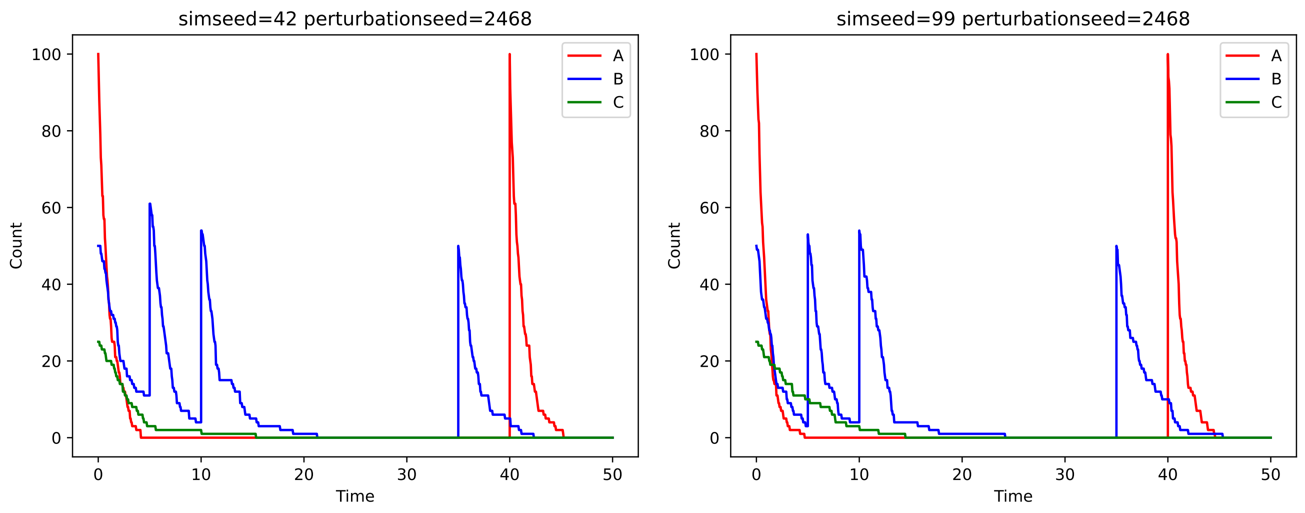

Figure 2: Example stochastic simulation timecourses for two different seeds for the simulation pseudorandom number generator (42 and 99), but with the same seed for the separate random number generator used for randomized perturbations (2468).

adds a specified amount to that species count while also printing a message for the user. The `rpFun` function is attached to the model as a perturbation function to be applied at four distinct timepoints (lines 25–26). The stochastic simulation is run (line 27) and the results are saved to an ASCII text file (lines 28–29). Finally, the results for each simulation are plotted and saved to a PDF file (lines 30–39). Example plots from two of the simulations executed by the code in Figure 1 are presented in Figure 2.

This example demonstrates the use of perturbation functions to implement randomized perturbations, with a separate pseudorandom number generator used for the decision-making in the perturbation function, independently from the pseudorandom number generator used by the stochastic simulator itself. As shown in Figure 2, changing the seed for the simulator produces a different underlying sequence of reactions and reaction timings, whereas keeping the same seed for the perturbations means that the same set of perturbation choices is made in both simulations.

## 6 Codebase documentation

The codebase is organized into five main source files that implement the core functionality of the simulator (`psim_lib.py`, `psim_parse.py`, `psim_model.py`, `psim_result.py`, and `psim_simulate.py`) plus two additional source files that provide extra functionality and some illustrative examples (`psim_seeds.py` and `psim_examples.py`). We now discuss the contents of these files in turn, focusing on the most relevant components for the end-user of the ProBioSim simulation system.

### 6.1 `psim_lib.py`

This file contains general-purpose utility functions used by the rest of the codebase. It is unlikely that an end-user would need to use these, with the possible exception of `psim_lib.__version__`, which is a string containing the version number of the current codebase.

### 6.2 `psim_parse.py`

This file contains utility functions used to parse models and model elements from string-based input. The code in this file sets out the inputs that will be parsed as valid; see below for a description of what those inputs should actually be. Using this file directly is not recommended; rather, the user should call the relevant functions in `psim_model.py` to parse input and use it to create a model.

### 6.3 `psim_seeds.py`

This file provides some additional utility functions to simplify the generation and importing of seed values for running batteries of stochastic simulations in a reproducible manner.

- **Function** `saveSeeds(fname, n, comment=None, overwrite=False, failOnDuplicate=True)`**:** Generate a random list of seed values and save them to a file. The argument `fname` should be a string representing the target filename; if this file exist and `overwrite` is `False`, an error will be raised. The argument `n` should be a positive integer representing the number of seeds to generate. If `comment` is not `None`, it should be a string representing a title that will be prepended to the file as a comment line prefixed by `#`. If `failOnDuplicate` is `True`, all seeds will be checked for uniqueness and an error will be raised if a duplicate is found (though this is unlikely to occur).
- **Function** `loadSeeds(fname)`**:** Load list of seed values from a specified file. The argument `fname` should be a string representing the input filename. Lines that start with `#` are ignored as comments. Thus, any file generated by the `saveSeeds` function can be directly loaded back in using the `loadSeeds` function.
- **Function** `appendSeeds(fname, n, comment=None, failOnDuplicate=True)`**:** Similar to `saveSeeds`, except the specified filename must already exist and the generated seeds will be appended to that file.
- **Function** `seedsAreUnique(seeds)`**:** Given a list-like sequence of seeds, returns true if they are all unique and false otherwise.

## 6.4 `psim_model.py`

This file defines the `Model` class, which stores the mass-action models to be simulated. This takes the form of a sequence of timepoints with associated sequences of values recorded for each species, either floating-point concentrations (if a deterministic simulation) or integer-valued counts (if a stochastic simulation). Key components of the file are outlined below:

- **Function** `positiveDualRailSpecies(x)`: Defines a naming convention for the positive species of a dual rail signal pair, which is to add "`p`" to the name of the dual rail signal. So, for a dual rail signal named `X`, the positive species would be named `Xp`. Adhering to this naming convention when creating models involving dual rail signals makes it possible to use built-in capabilities for manipulating the results of simulations involving dual rail signals, defined in `psim_result.py` below.
- **Function** `negativeDualRailSpecies(x)`: Defines a naming convention for the negative species of a dual rail signal pair, which is to add "`m`" to the name of the dual rail signal. So, for a dual rail signal named `X`, the negative species would be named `Xm`. Adhering to this naming convention when creating models involving dual rail signals makes it possible to use built-in capabilities for manipulating the results of simulations involving dual rail signals, defined in `psim_result.py` below.
- **Class** `Model(object)`**:** this class represents models in ProBioSim. Important aspects of this class are discussed below. Note that while there are methods to *add* reactions and perturbations to a model, there are currently no methods to *remove* these. The intent is that new models can be straightforwardly reconstructed in a programmatic manner.
  - **Special method** `__init__(self)`: The initializer for the `Model` class produces a model object that is empty by default and has default values for the simulation settings. The simulation settings and their default values are as follows:
    - `length`: the length of the simulation in arbitrary time units (default value: `1.0`).
    - `points`: the number of sampling points during the simulation (default value: `101`).
    - `stiff`: whether or not to run a deterministic simulation using a stiff solver (default value: `True`).
    - `seed`: the seed to use for running a stochastic simulation (default value: `None`).
    - `rtol`: the relative tolerance parameter to use for running a deterministic simulation (default value: `1e-8`).
    - `atol`: the absolute tolerance parameter to use for running a deterministic simulation (default value: `1e-8`).
    - `recordAllIfStochastic`: whether or not to record the state after every reaction in a stochastic simulation (default value: `False`).
  - **Method** `checkWellFormedness(self)`**:** Carries out various checks on the model after any alteration, to ensure that it remains in a well-formed state. This is automatically called every time the model is modified, to catch any potential mistakes as early as possible. An error is raised if any of the checks fail. The checks that are made can be summarized as follows:
    - Checks that the simulation length is a positive number.
    - Checks that the number of simulation sample points is a positive `int`.
    - Checks that the stochastic simulation seed parameter is either `None` or an `int`.
    - Checks that the deterministic simulation stiffness parameter is a `bool`.
    - Checks that the deterministic stimulation relative tolerance parameter is a positive `float`.
    - Checks that the deterministic stimulation absolute tolerance parameter is a positive `float`.

    * Checks that the stochastic simulation "record after every reaction" parameter is a `bool`.
    * Checks that all assigned rate constant values are non-negative.
    * Checks that all assigned initial species values are non-negative.
    * Checks that all species names are syntactically valid and are all assigned an initial value (species are assigned initial value of zero by default).
    * Checks that all reactions have at least one reactant or product.
    * Checks that no reaction has identical reactants and products.
    * Checks that all reactions have rate constants that are either a non-negative numeric value or a syntactically valid rate constant name.
    * Checks that all named rate constants are assigned a value (rate constants are assigned a value of zero by default).
    * Checks that all reactions that contain an annotation entry are either annotated with a string or with `None`.
    * Checks that no reaction is contains the same reactants and products as another reaction.
    * Checks that the perturbation data structures are well-formed; see below for more details.
        · Briefly, each perturbation should be a `dict` whose keys should include `''time''`, `''actions''`, and `''functions''`.
        · The value for `''time''` should be the perturbation time, which should be positive (that is, no perturbation can be scheduled at time zero or at a negative time point).
        · The value for `''actions''` should be a list-like sequence of individual action specifications, each of which should be a `dict` containing keys `''speciesName''`, `''amount''`, and `''relative''`. The value for `''speciesName''` should be a valid species name. The value for `''amount''` should either be an `int` or a `float`. The value for `''relative''` should be a `bool`.
        · The value for `''functions''` should be a list-like sequence of `callable` objects or functions. See above for details of how to write functions to implement model perturbations. Note, however, that the system does not check these for correctness beyond confirming that they are, in fact, `callable`.

- **Method** `getAllSpeciesNames(self)`**:** Returns the list of all species names referenced in the model.
- **Method** `getRateConstants(self)`**:** Returns the current mapping from rate constant names (as strings) to values, as a `dict`.
- **Method** `getRateConstant(self, rateName)`**:** Returns the rate constant value currently assigned to `rateName`. Throws an error if the corresponding rate constant is not defined.
- **Method** `setRateConstant(self, rateName, value, strict=False)`**:** Sets the value of the rate constant `rateName` (which should be a string) to be `value` (which should be a non-negative int or float). If `strict=True`, the code will produce an error if `rateName` already has a value defined for it.
- **Method** `getSpeciesInits(self)`**:** Returns the current mapping from species names (as strings) to their initial values, as a `dict`.
- **Method** `getSpeciesInit(self, species)`**:** Returns the initial value currently assigned to the species named `species`. Throws an error if the corresponding species does not have an initial value defined.
- **Method** `setSpeciesInit(self, species, value, strict=False)`**:** Sets the value of the initial value for species `species` (which should be a string) to be `value` (which should be a non-negative int or float). If `strict=True`, the code will produce an error if `species` already has an initial value defined for it.
- **Method** `getSpeciesInitDual(self, signal)`**:** Returns the initial value currently assigned to the dual-rail signal named `signal`. Note, that can return a negative value as it is calculating the represented initial dual-rail value from the initial values of the two underlying species. Throws an error if either of the two corresponding underlying species does not have an initial value defined.
- **Method** `setSpeciesInitDual(self, signal, value, strict=False)`**:** Sets the value of the initial value for the dual-rail signal `signal` (which should be a string) to be `value` (which should be an int or float). Note that `value` may be negative in this case; the code correctly assigns the corresponding non-negative value to the positive or negative rail species as required, using the naming convention defined by the `positiveDualRailSpecies` and `negativeDualRailSpecies` functions. If `strict=True`, the code will produce an error if either of these species already has an initial value defined for it.
- **Method** `getReactions(self)`**:** Returns the data structure representing the current set of reactions in the model. This will be a list of `dicts` representing individual reactions, each of the form:

```
{''reactants'':rs, ''rate'':r, ''products'':ps}
```

where `rs` is a list of strings representing the reactant species, `r` represents the rate constant, and `ps` is a list of strings representing the product species. Optionally, there may also be an ``‘‘annotation’’`` key, which is used by the model export functions outlined below. These should satisfy the correctness criteria outlined in the above description of the `checkWellFormedness` method.

- **Method** `addReaction(self, reaction)`: Add a single reaction to the model; the argument `reaction` should be a `dict` of the form outlined in the above description of the `getReactions` method.
- **Method** `addReactions(self, reactions)`: Add multiple reactions to the model; the argument `reaction` should be a list-like sequence of `dicts` of the form outlined in the above description of the `getReactions` method.
- **Method** `addReactionSeparately(self, reactants, rate, products, annotation=None)`: Add a single reaction to the system, with the values for the `‘‘reactants’’`, `‘‘rate’’`, and `‘‘products’’` fields of the resulting `dict` supplied as separate arguments. Note that the value for `‘‘annotation’’` is an optional argument to this method.
- **Method** `getSimulationLength(self)`: Returns the current value of the simulation length parameter.
- **Method** `setSimulationLength(self, length)`: Assigns a new value for the simulation length parameter. Argument `length` should be a valid value for this parameter as per the well-formedness checks outlined above.
- **Method** `getSimulationPoints(self)`: Returns the current value of the parameter determining the number of simulation sample points.
- **Method** `setSimulationPoints(self, points)`: Assigns a new value for the parameter determining the number of simulation sample points. Argument `points` should be a valid value for this paramter as per the well-formedness checks outlined above.
- **Method** `getSimulationStiff(self)`: Returns the current value of the parameter determining whether to use a stiff solver for deterministic simulations.
- **Method** `setSimulationStiff(self, stiff)`: Assigns a new value for the parameter determining whether to use a stiff solver for deterministic simulations. Argument `stiff` should be a valid value for this paramter as per the well-formedness checks outlined above.
- **Method** `getSimulationSeed(self)`: Returns the current value of the pseudorandom number generator seed to be used for stochastic simulations.
- **Method** `setSimulationSeed(self, seed)`: Assigns a new value for the pseudorandom number generator seed to be used for stochastic simulations. Argument `seed` should be a valid value for this paramter as per the well-formedness checks outlined above.
- **Method** `getSimulationRtol(self)`: Returns the current value of the relative tolerance parameter for deterministic simulations.
- **Method** `setSimulationRtol(self, rtol)`: Assigns a new value for the relative tolerance parameter for deterministic simulations. Argument `rtol` should be a valid value for this paramter as per the well-formedness checks outlined above.
- **Method** `getSimulationAtol(self)`: Returns the current value of the absolute tolerance parameter for deterministic simulations.
- **Method** `setSimulationAtol(self, atol)`: Assigns a new value for the absolute tolerance parameter for deterministic simulations. Argument `atol` should be a valid value for this paramter as per the well-formedness checks outlined above.
- **Method** `getSimulationRecordAllIfStochastic(self)`: Returns the current value of the parameter determining whether to sample after every reaction in a stochastic simulation.
- **Method** `setSimulationRecordAllIfStochastic(self, b)`: Assigns a new value for the parameter determining whether to sample after every reaction in a stochastic simulation. Argument `b` should be a valid value for this parameter as per the well-formedness checks outlined above.
- **Method** `getPerturbations(self)`: Returns the data structure representing the current set of perturbations in the model. This will be list of `dicts` representing the perturbations to be applied at different time points, each of the form:

  ```
  {‘‘time’’:t, ‘‘actions’’:cs, ‘‘functions’’:fs}
  ```

  where `t` is a positive time for the perturbations to occur, `fs` is a list of functions or callable objects, and `cs` is a list of individual actions, each of the form:

  ```
  {‘‘speciesName’’:s, ‘‘amount’’:a, ‘‘relative’’:r}
  ```

where `s` is a valid species name, `a` is the amount of the perturbation, and "r" is a `bool` indicating whether the new value should be relative to the old value. These should satisfy the correctness criteria outlined in the above description of the `checkWellFormedness` method. See above for a description of the semantics of perturbation actions. Note that the system automatically collapses multiple perturbations applied at the same time point into a single entry; the actions and functions are queued up to be applied in the order that they were added to the model.

– **Method** `addPerturbation(self, time, action)`: Add a new perturbation consisting of the single action at the specified time. The values of the arguments `time` and `action` should be of the form outlined in the above description of the `getPerturbations` method. If existing perturbations are specified at this time, the new action will be queued *after* the pre-existing actions. See the above description of the semantics of model perturbations for more detail on this topic.

– **Method** `addPerturbationSeparately(self, time, speciesName, amount, relative)`: Add a new perturbation at the specified time, consisting of a single action whose constituent parts (`speciesName`, `amount`, and `relative`) are specified as separate arguments to this function call. The values of the arguments `time` and `action` should be of the form outlined in the above description of the `getPerturbations` method.

– **Method** `addPerturbationMultipleActions(self, time, actions)`: Add a new perturbation consisting of multiple actions at the specified time. The values of the arguments `time` and the individual elements of `actions` should be of the form outlined in the above description of the `getPerturbations` method. Note that these multiple actions will be queued up in the order they are presented in `actions`, and *after* any pre-existing actions scheduled at that time. See the above description of the semantics of model perturbations for more detail on this topic.

– **Method** `addPerturbationDual(self, time, dual_action)`: Similar to the `addPerturbation` method, but where the target is a dual-rail signal (that is, a pair of underlying species that follow the naming convention outlined above in the `positiveDualRailSpecies` and `negativeDualRailSpecies` functions). The argument `dual_action` should be a `dict` similar to the argument of the `addPerturbation` method, except that the "`speciesName`" key should be replaced by a "`signalName`" key.

– **Method** `addPerturbationSeparatelyDual(self, time, signalName, amount, relative)`: Dual-rail version of the `addPerturbationSeparately` method outlined above.

– **Method** `addPerturbationMultipleActionsDual(self, time, dual_actions)`: Dual-rail version of the method `addPerturbationMultipleActions` outlined above, where the `dual_actions` argument should be a list-like sequence of multiple dual-rail actions like those expected by `addPerturbationDual`.

– **Method** `addPerturbationAsCode(self, time, f)`: Add a perturbation at the specified timepoint `time` in the simulation, where the modifications to the simulation state are specified by the callable `f`. See above for details of how to define the function `f` and how perturbations defined as functions interact with those defined as actions. Note also that no checks are made on the structure or correctness of `f` besides the fact that it is callable.

– **Method** `addPerturbationAsCodeMultipleFunctions(self, time, fs)`: Add a list-like sequence `fs` of perturbations expressed as functions or callable objects, at timepoint `time`. Note that these functions will be executed in the order specified in `fs`.

– **Method** `updateFromString(self, x)`: Parse the argument string `x`, whose should contain a single line of input following the ASCII CRN syntax outlined above, and update the model appropriately. An error will be thrown if `x` is not syntactically valid or if the resulting model is not well-formed.

– **Method** `updateFromStrings(self, xs)`: Parse each string from the list-like sequence `xs`, each of which should contain a single line of input following the ASCII CRN syntax outlined above, and update the model appropriately. An error will be thrown if any of the strings from `xs` are not syntactically valid or the resulting model is not well-formed.

– **Method** `inferDualRailSignals(self)`: Method that attempts to infer the names of all dual-rail signals utilized in the model, by searching for pairs of species whose names adhere to the naming convention defined by the `positiveDualRailSpecies` and `negativeDualRailSpecies` functions. This is primarily intended to aid in user analysis of large models involving dual-rail signals. Note that "false positive" results could be returned by this function if the model contains unrelated species that happen to follow the naming convention for dual-rail signals.

– **Method** `getASCII(self, title=None, includeZeroInits=False,`
`reaction_annotation_sections=None)`: Returns a string containing an ASCII representation of the model that can be re-parsed. Optional argument `title`, if not `None`, specifies a string title for the model that is included as a comment at the beginning. Optional argument `includeZeroInits` should be a `bool` indicating whether to explicitly list initial values for those species with initial value of zero (this is the default assumed when re-parsing). Optional argument `reaction_annotation_sections`, if not `None`, provides a list-like sequence of string titles to break out annotated reactions into. If reactions with no annotations, or annotations not supplied in that list-like

sequence, are present in the model, they appear after the annotated reactions, identified as "other reactions". If `reaction_annotation_sections` is `None`, any reaction annotations are included as comments on the same line as the reaction itself. An error will be thrown if any name is duplicated in the `reaction_annotation_sections` argument. Note that any perturbations in the model defined as Python functions will **not** be saved in the ASCII format; if present, a warning will be raised (and added to the output as a comment to illustrate that a perturbation defined as code was omitted). This function is also used to define the `__str__` special method for producing a string representation of the `Model` object.

  - **Method** `writeASCIIToFile(self, outfile, title=None, includeZeroInits=False, reaction_annotation_sections=None)`: Produces an ASCII representation of the model as per the `getASCII` method and writes it to the output file named `outfile`.
  - **Method** `getLaTeX(self, title=None, includeZeroInits=False, reaction_annotation_sections=None)`: Produces a string containing a LaTeX document representing the model. The meanings of the optional arguments are the same as in the `getASCII` method outlined above.
  - **Method** `writeLaTeXToFile(self, outfile, title=None, includeZeroInits=False, reaction_annotation_sections=None)`: Produces a LaTeX representation of the model as per the `getLaTeX` method and writes it to the output file named `outfile`.
  - **Method** `writeLaTeXToFileAndTypeset(self, outfile, latexProgram='‘pdflatex’', title=None, includeZeroInits=False, reaction_annotation_sections=None)`: Produces a LaTeX representation of the model as per the `getLaTeX` method, writes it to the output file named `outfile`, and attempts to typeset it using the LaTeX executable specified as the optional argument `latexProgram`, which must be in the system path. The LaTeX source produces requires the following packages to compile: `amssymb`, `amsmath`, `mathpazo`, and `geometry`.

- **Function** `getEmptyModel()`: Returns an empty `Model` object with default simulation settings, as a starting point for building up a full model.
- **Function** `parseStrings(xs, strict=False)`: Parse each string from the list-like sequence `xs`, each of which should contain a single line of input following the ASCII CRN syntax outlined above, and use it to produce a model starting with the empty model. An error will be thrown if any of the strings are syntactically invalid or if the resulting model is not well-formed. The value of the `strict` optional argument is passed to the relevant model update methods.
- **Function** `parseString(x, strict=False)`: Uses the string `x` to create a `Model` object by splitting on newlines and passing the resulting list to `parseStrings`. Thus, `x` should be a string containing a (potentially) multi-line specification of the model. An error will be thrown if any of the strings are syntactically invalid or if the resulting model is not well-formed. The value of the `strict` optional argument is passed to the relevant model update methods.
- **Function** `parseFile(infile, strict=False, quiet=False)`: Parse the text file `inFile` and use the `parseStrings` function to produce a corresponding `Model` object based on its content. An error will be thrown if any of the strings are syntactically invalid or if the resulting model is not well-formed. The value of the `strict` optional argument is passed to the relevant model update methods. The `quiet` optional argument determines whether to print certain status messages on the file loading process.

## 6.5 `psim_result.py`

This file defines the `SimulationResult` class, which stores the results of ProBioSim simulations. Key components of the file are outlined below:

- **Class** `SimulationResult(object)`: this class represents the results of ProBioSim simulations. Important aspects of this class are discussed below. Note that we do not discuss the initializer as the end-user of ProBioSim should not need to construct these objects manually.
  - **Method** `_getTimeIndexes_(self, ts)`: Given an increasing, list-like sequence `ts` of times, return a list of the indexes into the numpy array that represent the corresponding times in the simulation. If a time falls between two recorded timepoints, the index of the preceding timepoint is returned. If a time falls precisely on the time of a perturbation, for which there will be two recorded values in the simulation output and thus two distinct indexes, the **later** index (corresponding to the value *after* the perturbation is applied) is returned. This function carries out a linear scan through the recorded simulation times; therefore, **the most efficient way to obtain multiple array indexes for particular timepoints is to request them all in one go via this method.** The method raises an error if the input sequence is empty, if multiple times in the input sequence are identical, if the sequence of times is not sorted into ascending order, or if any requested times fall outside the simulated range.
  - **Method** `_getTimeIndex_(self, t)`: Get the index associated with the specified time `t`. This method is defined in terms of the `__getTimeIndexes__` above, therefore, the restrictions outlined above apply here also.

- **Method** `times(self)`: Returns the full sequence of times for which data is stored in this object, as a 1D numpy array. Note that, as per the semantics of perturbations outlined above, there could be more entries in this array than requested points, due to time duplications caused by perturbations.
- **Method** `timeInitial(self)`: Returns the time of the first timepoint for which data is stored in this object.
- **Method** `timeFinal(self)`: Returns the time of the final timepoint for which data is stored in this object.
- **Method** `get(self, x)`: Returns the sequence of values recorded for species `x`, as a 1D numpy array. The length of this array will be the same as the length of the array returned by the `times` method.
- **Method** `getInitial(self, x)`: Returns the initial value of species `x`.
- **Method** `getFinal(self, x)`: Returns the final value recorded for species `x`.
- **Method** `_getIndex_(self, x, idx)`: Returns the value recorded for species `x` at index `idx` in the output array. Note that `idx` here is an index into the output array, not a simulation time index.
- **Method** `_getIndexes_(self, x, idxs)`: Returns a 1D numpy array containing the values recorded for species `x` at the sequence of indexes `idxs` in the output array.
- **Method** `getAt(self, x, t)`: Returns the value recorded for species `x` at simulation time `t`. This method uses the `_getTimeIndex_` method to convert from simulation times to indexes into the result array. **Note that other methods below will be more efficient than making multiple calls to this method at different time points.**
- **Method** `getMultiple(self, xs, t)`: Return a list containing the values of the species from the list-like sequence `xs` at simulation time `t`.
- **Method** `getAtMultiple(self, x, ts)`: Return a 1D numpy array containing the values recorded for species `x` at the list-like sequence of increasing simulation times `ts`. This method uses the `_getTimeIndexes_` method to convert from simulation times to indexes into the result array, so the limitations outlined on `ts` above for that method apply here also. **This is the most efficient way to get the values recorded for a single species at multiple time points.**
- **Method** `getMultipleAtMultiple(self, xs, ts)`: Return a list of 1D numpy arrays containing the values recorded for multiple species `xs` at the list-like sequence of increasing simulation times `ts`. This method uses the method `_getTimeIndexes_` to convert from simulation times to indexes into the result array, so the limitations outlined on `ts` above for that method apply here also. **This is the most efficient way to get the values recorded for multiple species at multiple time points.**
- **Method** `getDual(self, x)`: Dual-rail version of the `get` method.
- **Method** `getInitialDual(self, x)`: Dual-rail version of the `getInitial` method.
- **Method** `getFinalDual(self, x)`: Dual-rail version of the `getFinal` method.
- **Method** `_getIndexDual_(self, x, idx)`: Dual-rail version of the `_getIndex_` method.
- **Method** `_getIndexesDual_(self, x, idxs)`: Dual-rail version of the `_getIndexes_` method.
- **Method** `getAtDual(self, x, t)`: Dual-rail version of the `getAt` method.
- **Method** `getMultipleDual(self, xs, t)`: Dual-rail version of the `getMultiple` method.
- **Method** `getAtMultipleDual(self, x, ts)`: Dual-rail version of the `getAtMultiple` method.
- **Method** `getMultipleAtMultipleDual(self, xs, ts)`: Dual-rail version of the `getMultipleAtMultiple` method.
- **Method** `allSpecies(self)`: Returns the sorted list of all species names referenced in the `SimulationResult` object.
- **Method** `getInitialAll(self)`: Returns a `dict` mapping all species names to their initial values.
- **Method** `getFinalAll(self)`: Returns a `dict` mapping all species names to their final recorded values.
- **Method** `_getIndexAll_(self, idx)`: Returns a `dict` mapping all species names to the values as recorded at index `idx` in the output array.
- **Method** `getAtAll(self, t)`: Returns a `dict` mapping all species names to the values as recorded at simulation timepoint `t`.
- **Method** `toFile(self, fname, savetimes=None, savespecies=None, separator=‘‘\t’’, decimalPlaces=None)`: Writes the data from the `SimulationResult` object to an output text file named `fname`. By default, all recorded values for all species are saved to the file. The optional argument `savetimes`, if not `None`, should provide a list-like sequence of times at which to save; in which case the `getMultipleAtMultiple` method is used to extract the values to save. The optional argument `savespecies`, it not `None`, should provide a list-like sequence of species to save. The optional argument `separator` provides the separator to use to delimit columns in the output file (tab-separated by default). The optional argument `decimalPlaces`, if not `None`, should be an integer specifying how many decimal places to round all numbers to in the output. The generated file can be re-parsed using the `fromFile` function below to recreate the object.

  - **Method** `toPandas(self, savetimes=None, savespecies=None)`: Returns a Pandas `DataFrame` [4] containing the data from the `SimulationResult` object. An error will be raised if the Pandas library is not available. The `savetimes` and `savespecies` optional arguments are interpreted as in the `toFile` method above.
  - **Method** `toExcel(self, fname, savetimes=None, savespecies=None)`: Saves the data from a `SimulationResult` object as an XLSX file named `fname`. An error will be raised if the Openpyxl library is not available. The `savetimes` and `savespecies` optional arguments are interpreted as in the `toFile` method above.

- **Function** `fromFile(fname, separator=‘‘\t’’)`: Parse the text file `fname`, which should be of the format produced by the `toFile` method above, to produce a `SimulationResult` object. An error will be thrown if the file is not well-formed, for example, if it is empty, if the header line is incorrect (the first column should be entitled "Times" and the rest should be valid species names), if there are duplicate column headers, if the lines are of unequal lengths, or if there are non-numeric entries. The `separator` optional argument provids the delimiter to use when parsing the file into columns.

## 6.6 `psim_simulate.py`

This file implements the simulation engine of ProBioSim. The `gillespie` and `simulateWithPerturbations` functions are for internal use; the recommended interface to start simulations is outlined below.

- **Function** `run(model, simType=‘‘deterministic’’, verbose=False)`: Runs a mass-action simulation of `Model` object passed as the positional argument `model`, and returns a `SimulationResult` object containing the resulting data. See above for the semantics of perturbations and how they affect the details of how results are saved. Optional argument `simType` must either be `‘‘deterministic’’`, which specifies a simulation based on forming and integrating mass-action ODEs over time, or `‘‘stochastic’’`, which specifies a simulation using Gillespie's direct method [1]. Otherwise, an error will be thrown. Optional argument `verbose` controls whether various status messages are printed during the simulation process. During the model preprocessing, any perturbations with times that occur at or after the specified simulation end time are omitted, and a warning is printed to the terminal alerting the user to this fact. An error will be thrown if a stochastic simulation is selected but one of the initial species values is not an integer, or if any of the perturbation actions calls for a non-integer value.
- **Function** `runDeterministic(model, verbose=False)`: A wrapper that simply calls the `run` function above with the `simType` optional argument set to `‘‘deterministic’’`.
- **Function** `runStochastic(model, verbose=False)`: A wrapper that simply calls the `run` function above with the `simType` optional argument set to `‘‘stochastic’’`.

## 6.7 `psim_examples.py`

This file contains some illustrative examples of simple ProBioSim models, including those that use custom code to implement model perturbations.

- **Function** `example001_perturbations_explicit()`: A simple example illustrating programmatic model creation, including the addition of explicitly defined perturbation actions.
- **Function** `example002_perturbations_arbitraryCode()`: An example illustrating the use of perturbations defined as Python functions.
- **Function** `example003_perturbations_arbitraryCode_randomized()`: An example illustrating the incorporation of randomized elements into perturbations defined as Python functions, including the use of a separate seed for a separate random number generator used to create reproducible pseudorandom behavior in the perturbations. The latter aspect is implemented by creating a random number generator object that is referenced as a free variable of the function defining the perturbation.
- **Function** `example004_perturbations_arbitraryCode_randomized_timeDependent()`: An example illustrating both randomized and time-dependent perturbation actions defined as Python functions.
- **Function** `example005_perturbations_arbitraryCode_stateInspection()`: An example showing how perturbations defined as Python functions can inspect the current state of the system. In this example, the capability is just used to print the values but could easily be used to implement perturbations that are conditional on the current state.

## Acknowledgments

This material is based upon work supported by the National Science Foundation under Grants 1518861, 1525553, and 1935087.

## References

[1] Daniel T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *Journal of Physical Chemistry*, 81(25):2340–2361, 1977. doi:10.1021/j100540a008.

[2] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585:357–362, 2020. doi:10.1038/s41586-020-2649-2.

[3] John D. Hunter. Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007. doi:10.1109/MCSE.2007.55.

[4] Wes McKinney. Data structures for statistical computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56–61, 2010. doi:10.25080/Majora-92bf1922-00a.

[5] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C. J. Carey, İlhan Polat, Yu Feng, , Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature Methods*, 17:261–272, 2020. doi:10.1038/s41592-019-0686-2.