blossom Documentation

Release 2.0.0

Bryan Brzycki

Apr 22, 2024

CONTENTS

blossom is a Python package for producing simulations of evolving organisms.

CHAPTER

ONE

TABLE OF CONTENTS

1.1 Installation

You can use pip to install the latest version of the package automatically:

```
pip install blossom
```

Alternately, execute:

```
git clone git@github.com:blossom-evolution/blossom.git
python setup.py install
```

1.2 Basic Usage

To start a simulation project, create a new directory to house all custom scripts, including a configuration file `config.yml`. In this config file, you specify species parameters, including starting population, max age, and action methods. Some action methods are built-in (for movement, eating, drinking, and reproduction), but you may use custom methods defined in external Python scripts, which are imported at runtime by Blossom via `linked_modules`.

World parameters are also specified in the config file, including dimensions and the distribution of water and food. You may also set limits on the number of timesteps and organisms, in order to control the simulation in case of runaway populations.

With your project directory set up, you may run the simulation using the included command-line interface (CLI):

```
blossom run
```

Note that for reproducibility, you can set the random seed either in the config file or at the CLI. For additional options, run `blossom run -h`.

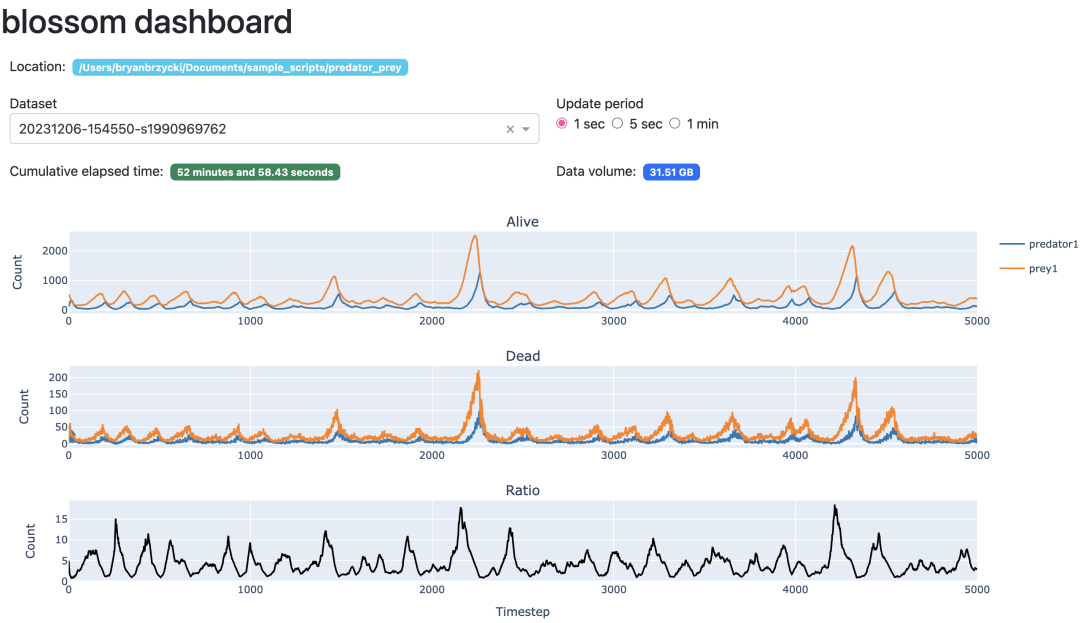
1.2.1 Dashboard

Blossom provides a dashboard that runs in your local browser, tracking the progress of your population runs.

Initiate the dashboard, run

```
blossom dashboard TRACK_DIR [-p PORT]
```

where `TRACK_DIR` is the simulation project directory. You can then view the dashboard at `localhost:PORT`.



1.3 Cookbook

On this page, we present a complete simulation structure to get started with Blossom. In this case, we aim to model a predator-prey dynamic.

First, we can create a basic directory structure for our project:

```
predator-prey/
    config.yml
    custom.py
```

Your config file might look like this:

```
species:
  - name: predator
    population: 100
    max_age: inf
    action: predator_action
    movement: simple_random
    reproduction:
```

(continues on next page)

(continued from previous page)

```
      type: pure_replication
    eating:
      type: eat_prey
      capacity: 250
      initial: 200
      metabolism: 10
      intake: 80
      days_without: 5
    linked_modules:
      - custom.py
  - name: prey
    population: 500
    max_age: inf
    action: prey_action
    movement: simple_random
    reproduction:
      type: pure_replication
    linked_modules:
      - custom.py
world:
  dimensionality: 2
  size: [1, 100]
  water:
    peak: inf
  food:
    peak: inf
  obstacles:
    peak: 0
timesteps: 5000
organism_limit: 20000
```

This produces a 1 x 100 world (technically one-dimensional) with unlimited resources, to focus on the organism interactions.

The custom methods are defined in external modules, such as this in `custom.py`:

```
import numpy as np

def predator_action(organism, universe):
    if universe.rng.random() < 1/2:
        if organism.food_current > organism.food_capacity // 2:
            return 'reproduce'
        else:
            return 'eat'
    else:
        if universe.rng.random() < 2/10: # 1/10:
            return 'eat'
        else:
            return 'move'

def eat_prey(organism, universe):
    location = tuple(organism.location)
    colocated_prey = []
```

(continues on next page)

(continued from previous page)

```
    colocated_predators = []
    for org in universe.organisms_by_location[location]:
        if org.alive:
            if org.species_name == 'prey1':
                colocated_prey.append(org)
            elif org.species_name == 'predator1':
                colocated_predators.append(org)
    if len(colocated_prey) == 0 or len(colocated_prey) <= len(colocated_
↪predators):
        return [organism]
    elif len(colocated_prey) == 1:
        prey = colocated_prey[0]
    else:
        prey = universe.rng.choice(colocated_prey)

    # food_from_prey = 0.8 * (prey.food_capacity)
    food_from_prey = organism.food_intake
    diff = organism.food_capacity - organism.food_current
    intake = min(food_from_prey, diff)
    organism = organism.update_parameter('food_current',
                                         intake,
                                         method='add')

    prey = prey.die('eaten')

    return [organism, prey]

def prey_action(organism, universe):
    if universe.rng.random() < 1/30:
        return 'reproduce'
    else:
        return 'move'
```

Notice that in the config file, the custom methods are listed by name and the external modules are linked via the `linked_modules` keyword.

To execute simulations, we can run this command within the project directory:

```
blossom run -s SEED
```

While it isn't necessary by any means, setting a seed at runtime promotes reproducibility. If the run is interrupted, you can re-run this same command and it will attempt to continue from the last point. Otherwise, if you wish to restart from the beginning, run `blossom run` with the `-r` flag.

1.4 blossom package

1.4.1 Subpackages

blossom.simulation package

Subpackages

blossom.simulation.organism_behavior package

Submodules

blossom.simulation.organism_behavior.action module

blossom.simulation.organism_behavior.action.**move_and_drink**(*organism*, *universe*)

Move and drink. Each occurs with probability 1/2.

blossom.simulation.organism_behavior.action.**move_and_reproduce**(*organism*, *universe*)

Move and reproduce. Reproduction occurs with probability 1/8.

blossom.simulation.organism_behavior.action.**move_only**(*organism*, *universe*)

Only move.

blossom.simulation.organism_behavior.action.**move_reproduce_drink**(*organism*, *universe*)

Move, drink, and reproduce. Reproduction occurs with probability 1/8. Drinks with probability 3/8, and moves with probability 1/2.

blossom.simulation.organism_behavior.drinking module

blossom.simulation.organism_behavior.drinking.**constant_drink**(*organism*, *universe*)

Intake constant amount of water from world if water is present.

blossom.simulation.organism_behavior.eating module

blossom.simulation.organism_behavior.eating.**constant_eat**(*organism*, *universe*)

Intake constant amount of food from world if food is present.

blossom.simulation.organism_behavior.movement module

blossom.simulation.organism_behavior.movement.**simple_random**(*organism*, *universe*)

Move in random direction with equal probability. For 2D, organisms walk diagonally.

blossom.simulation.organism_behavior.movement.**stationary**(*organism*, *universe*)

Organism stays still.

blossom.simulation.organism_behavior.reproduction module

blossom.simulation.organism_behavior.reproduction.**pure_replication**(*organism*, *universe*)

Replace organism with two organism with similar parameters. Essentially, only differences in parameters are organism id, ancestry, age, and water / food levels.

Module contents

Built-in methods for organism behaviors.

Submodules

blossom.simulation.dataset_io module

Load information from a certain dataset, e.g. to resume a simulation, and write world and organism data back to file.

class blossom.simulation.dataset_io.**NPEncoder**(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)

Bases: JSONEncoder

Class to help serialize numpy types to json.

default(*obj*)

Implement this method in a subclass such that it returns a serializable object for o, or calls the base implementation (to raise a TypeError).

For example, to support arbitrary iterators, you could implement default like this:

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    # Let the base class default method raise the TypeError
    return JSONEncoder.default(self, o)
```

blossom.simulation.dataset_io.**load_universe**(*fn*, *seed=None*)

Load dataset file from JSON.

Parameters

- **fn** (*str*) – Input filename of saved universe dataset
- **seed** (*int, Generator, optional*) – Random seed for the simulation

Returns

- **population_dict** (*dict*) – A dict of Organism objects reconstructed from the saved dataset

- **world** (*World*) – World object reconstructed from the saved dataset
- **seed** (*int, Generator*) – Numpy random number generator from last timestep

blossom.simulation.dataset_io.**save_universe**(*universe*)

Save population_dict and world to file in JSON format.

Parameters
universe (Universe) – Universe containing organism

blossom.simulation.default_fields module

Built-in dictionaries with world, species, and organism parameters (or fields).

Both Organism and World objects are initialized based on these field dictionaries, and values are either populated from parameter files or take on the default values specified in this module.

blossom.simulation.organism module

class blossom.simulation.organism.**Organism**(*init_dict={}*, *seed=None*)

Bases: object

A basic organism structure for all species.

act(*universe*)

Method that decides and calls an action for the current timestep. Searches through custom methods and built-in movement methods. The action method specifically selects an action to take, from "move", "reproduce", "drink", and "eat". Then the appropriate instance method from this class is executed to yield the final list of affect organisms.

Parameters
universe (Universe) – Universe containing organism
Returns
affected_organisms – Organism or list of organisms affected by this organism's action.
Return type
Organisms, or list of Organisms

classmethod clone(*organism*)

Makes a new Organism object identical to the current one.

Parameters
organism (Organism) – Organism to copy.
Returns
new_organism – Copied organism.
Return type
Organism

clone_self()

Clone this organism.

die(*cause*, *in_place=False*, *original=None*)

Method that "kills" organism.

Parameters
- **cause** (*str*) – Cause of this organism's death.
- **in_place** (*bool*) – If True, modifies self, otherwise, copy organism and return new Organism object.

Returns
dead_organism – New "dead" state of this organism.
Return type
Organism

`drink`(*universe*)

Method for handling drinking. Searches through custom methods and built-in drinking methods.
Parameters
universe (Universe) – Universe containing organism
Returns
affected_organisms – Organism or list of organisms affected by this organism's drinking.
Return type
Organisms, or list of Organisms

`eat`(*universe*)

Method for handling eating. Searches through custom methods and built-in eating methods.
Parameters
universe (Universe) – Universe containing organism
Returns
affected_organisms – Organism or list of organisms affected by this organism's eating.
Return type
Organisms, or list of Organisms

`get_child`(*other_parent=None*, *seed=None*)

Creates an Organism object with similar properties to self, and can add another parent if it exists. Note that this doesn't assume anything about how much food / water the child is left with, so these should be set with custom / default reproduction methods.
Parameters
other_parent (Organism) – Parent that reproduces with self to produce the child.
Returns
child – Generated child.
Return type
Organism

`get_new_id`(*seed=None*)

Generates pseudo-random ID for the organism, seeded by the universe.

`is_at_death`(*cause*)

Check various conditions for death.
Parameters
cause (*str*) – Potential cause of this organism's death.
Returns
is_dead – Returns True if organism is dead from the specified cause, False otherwise.
Return type
bool

`move`(*universe*)

Method for handling movement. Searches through custom methods and built-in movement methods.
Parameters
universe (Universe) – Universe containing organism
Returns
affected_organisms – Organism or list of organisms affected by this organism's movement.

Return type
Organisms, or list of Organisms

`reproduce`(*universe*)

Method for handling reproduction. Searches through custom methods and built-in reproduction methods.

Parameters
universe (Universe) – Universe containing organism
Returns
affected_organisms – Organism or list of organisms affected by this organism's reproduction. For example, this would include both parent and child organisms.
Return type
Organisms, or list of Organisms

`step`(*universe*, *do_action=True*)

Steps through one time step for this organism. Reflects changes based on actions / behaviors and updates to health parameters.

Returns a list of organisms that the action produced (either new or altered organisms).

Parameters
- **universe** (Universe) – Universe containing organism
- **do_action** (*bool*) – If True, this organism will act, otherwise, it will not.

Returns
affected_organisms – List of organisms affected by this organism's actions or health. This could be an updated version of this organism, especially if the organism dies during the time step, but could also be multiple other organisms affected by actions (i.e. children from reproduction).
Return type
list of Organisms

`step_without_acting`()

Steps through one time step without acting for this organism.

Returns
Note that this returns an Organism object, not a list.
Return type
organism

`to_dict`()

Convert Organism to dict.

`update_parameter`(*parameter*, *value*, *method='set'*, *in_place=False*, *original=None*)

Update a specific parameter of the organism.

Parameters
- **parameter** (*string*) – Parameter to update.
- **value** – Value with which to update.
- **method** (*string*) – Method types are: 'set', 'add', 'subtract', 'append'.
- **in_place** (*bool*) – If True, modifies self, otherwise, copy organism and return new Organism object.
- **original** (Organism *or None*) – Original organism we are changing. If it is the original, clone organism so that we aren't editing the original.

Returns
updated_organism – Organism object with updated parameter.
Return type
Organism

blossom.simulation.parameter_io module

Load information from parameter files and construct world and organism objects at the initial timestep.

blossom.simulation.parameter_io.**create_organisms**(*species_init_dict*, *init_world=<blossom.simulation.world.World object>*, *location_callback=None*, *seed=None*)

Make organism list from an species_init_dict either provided directly or scraped from parameter file. All organisms are from a single species.

blossom.simulation.parameter_io.**load_from_config**(*fn*, *seed=None*)

Create initial population and world from .yml configuration file.

blossom.simulation.parameter_io.**load_species**(*fns=None, init_dicts=[{}], init_world=<blossom.simulation.world.World object>, custom_module_fns=[]*)

Load organisms from available species parameter files or dictionaries.

Parameters

- **fns** (*list of str*) – Input filenames of species parameter files. Different species get different species parameter files, from which the individual organisms are initialized.
- **init_dicts** (*list of dict*) – Parameter dicts for each species.
- **init_world** (*World*) – Initial World instance for this Universe.
- **custom_module_fns** (*list of str*) – List of external Python scripts containing custom organism behaviors. *blossom* will search for methods within each filename included here.

Returns

population_dict – A dict of Organism objects constructed from the parameter file.

Return type

dict of Organisms

blossom.simulation.parameter_io.**load_species_from_dict**(*init_dicts*, *init_world*, *custom_module_fns=None*, *seed=None*)

Create a list of organisms loaded from Python dicts.

Parameters

- **init_dicts** (*list of dict*) – Input species dictionaries from which the individual organisms are initialized. Each dictionary is for a different species.
- **init_world** (*World*) – Initial World instance for this Universe.
- **custom_module_fns** (*list of str*) – List of external Python scripts containing custom organism behaviors. *blossom* will search for methods within each filename included here.
- **seed** (*int, Generator, optional*) – Random seed for the simulation

Returns

population_dict – A dict of Organism objects constructed from the parameter file.

Return type

dict of Organisms

blossom.simulation.parameter_io.**load_species_from_param_files**(*fns*, *init_world*, *custom_module_fns=None*, *seed=None*)

Load all available species parameter files.

Parameters

- **fns** (*list of str*) – Input filenames of species parameter files. Different species get different species parameter files, from which the individual organisms are initialized.
- **init_world** (*World*) – Initial World instance for this Universe.
- **custom_module_fns** (*list of str*) – List of external Python scripts containing custom organism behaviors. *blossom* will search for methods within each filename included here.
- **seed** (*int, Generator, optional*) – Random seed for the simulation

Returns
population_dict – A dict of Organism objects constructed from the parameter file.

Return type
dict of Organisms

blossom.simulation.parameter_io.**load_world**(*fn=None*, *init_dict={}*)

Load world from either parameter file or dictionary and construct initial World object.

Parameters

- **fn** (*str*) – Input filename of parameter file.
- **init_dict** (*dict*) – Dictionary containing world parameters.

Returns
world – World object constructed from the parameter file.

Return type
World

blossom.simulation.parameter_io.**load_world_from_dict**(*init_dict*)

blossom.simulation.parameter_io.**load_world_from_param_file**(*fn*)

Load world parameter file and construct initial World object.

Parameters
fn (*str*) – Input filename of parameter file.

Returns
world – World object constructed from the parameter file.

Return type
World

blossom.simulation.parameter_io.**parse_config_number**(*x*)

If config number is the string 'inf', use `np.inf`.

blossom.simulation.parse_intent module

blossom.simulation.parse_intent.**parse**(*intent_list*, *organism_list*, *seed=None*)

Determine whether the intent list is valid and fix it in the event of conflicts.

Parameters

- **intent_list** (*list of lists of Organisms*) – List of lists of organisms with proposed organism states, after each organism has 'acted'. Length equals number of organisms in the current time step.
- **organism_list** (*list of Organisms*) – List of current organisms
- **seed** (*int, Generator, optional*) – Random seed

Returns

- **updated_organism_list** (*list of Organisms*) – List of updated organisms, where organism states that conflict between `intent_list` and `organism_list` are resolved.
- *Conflicts may be cases in which an organism has different states in the*
- *intent list, perhaps arrising from the actions of other organisms that*
- *somehow effect its state. This method resolves those conflicts, so that*
- *there is only one organism with a given organism id present in the final*
- *output list at all times.*

blossom.simulation.population_funcs module

blossom.simulation.population_funcs.**get_organism_list**(*population_dict*)

Constructs organism list from population dict data structure.

blossom.simulation.population_funcs.**get_population_dict**(*organism_list*, *species_names*)

Constructs population dict from organism list data structure.

blossom.simulation.population_funcs.**hash_by_id**(*organism_list*)

Simple hashing by organism id over a list of organisms.

blossom.simulation.population_funcs.**hash_by_location**(*organism_list*)

Simple hashing by organism location over a list of organisms.

blossom.simulation.population_funcs.**organism_filter**(*organism_list*, **conditions*)

Selects organisms from organism list according to a set of conditions. Each condition should be a function that receives an Organism object as input and returns a boolean as output.

Example:

```
organism_filter(
    population_dict['prey1']['organisms'],
    lambda organism: organism.alive
)
```

blossom.simulation.population_funcs.**organism_list_copy**(*organism_list*)

blossom.simulation.universe module

class blossom.simulation.universe.**Universe**(*dataset_fn=None*, *config_fn=None*, *world_param_fn=None*, *species_param_fns=None*, *world_param_dict={}*, *species_param_dicts=[{}]*, *custom_module_fns=None*, *current_time=0*, *end_time=1000*, *project_dir='datasets/'*, *pad_zeros=4*, *seed=None*, ***kwargs*)

Bases: object

Create the universe of the simulation.

current_info(*verbosity=1*, *expanded=True*)

initialize(*seed=None*, *project_dir=None*)

Initialize world and organisms in the universe, from either saved datasets or from parameter files (and subsequently writing the initial time step to file).

run(*verbosity=1*, *expanded=True*)

step()

Steps through one time step, iterating over all organisms and computing new organism states. Saves all organisms and the world to file at the end of each step.

blossom.simulation.utils module

Common utilities used throughout *blossom*

blossom.simulation.utils.**cast_to_list**(*x*)

Make a list out of the input if the input isn't a list.

blossom.simulation.utils.**time_to_string**(*seconds*)

Convert time in seconds to the most reasonable representation.

blossom.simulation.world module

class blossom.simulation.world.**World**(*init_dict={}*)

Bases: object

World class for the environment of the simulation.

step()

to_dict()

Convert World to dict.

blossom.simulation.world_generator module

blossom.simulation.world_generator.**constant_2d_list**(*val*, *size*)

Generate a constant-valued two dimensional array.

blossom.simulation.world_generator.**constant_list**(*val*, *length*)

Generate a constant-valued list.

blossom.simulation.world_generator.**write_environment**(*water*, *food*, *obstacles*, *environment_fn='environment.json'*)

Write water, food, and obstacles lists to an environment file.

Module contents

blossom.visualization package

Submodules

blossom.visualization.dashboard module

blossom.visualization.parsing module

class blossom.visualization.parsing.**Snapshot**(*dataset_fn*)

Bases: object

Single time snapshot of universe.

plot_2d(*label*, *attr_func*)

attr_func is a function that accepts a Snapshot object and a location, and returns a quantity

class blossom.visualization.parsing.**TimeSeries**(*dataset_dir*)

Bases: object

Series of dataset objects for iterating over.

plot_ts(*attr_funcs*)

attr_funcs is a list of tuples (label, function), where the function calculates desired attributes given a Snapshot at each timestep in the simulation.

blossom.visualization.parsing.**read_log**(*fn*)

blossom.visualization.render module

Module contents

1.4.2 Submodules

1.4.3 blossom.blossom_exe module

1.4.4 Module contents

blossom is a package for simulating evolution

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

b

INDEX