ChromaX

Younis, Omar G. et al.

Mar 03, 2024

MODULES:

ChromaX is a Python library that enables the simulation of genetic recombination, genomic estimated breeding value calculations, and selection processes. The library is based on JAX to exploit parallelization. It can smoothly operate on CPU, GPU (NVIDIA, Apple, AMD, and Intel), or TPU.

CHAPTER

ONE

INSTALLATION

Note: To exploit parallelization capabilities of your hardware, it is recommended to install jax manually. You can find the instruction for your hardware in google/jax.

You can install the library via Python Package manager *pip*:

```
pip install chromax
```

This will install all the requirements, like JAX, NumPy and Pandas.

CHAPTER

TWO

CITING

Note: The sample data used in the examples is taken from *Wang, Shichen et al. "Characterization of polyploid wheat genomic diversity using a high-density 90 000 single nucleotide polymorphism array". Plant Biotechnology Journal 12. 6(2014): 787-796.*

To cite ChromaX in publications:

```
@article{10.1093/bioinformatics/btad691,
    author = {Younis, Omar G and Turchetta, Matteo and Ariza Suarez, Daniel and Yates,␣
↪Steven and Studer, Bruno and Athanasiadis, Ioannis N and Krause, Andreas and Buhmann,␣
↪Joachim M and Corinzia, Luca},
    title = "{ChromaX: a fast and scalable breeding program simulator}",
    journal = {Bioinformatics},
    volume = {39},
    number = {12},
    pages = {btad691},
    year = {2023},
    month = {11},
    abstract = "{ChromaX is a Python library that enables the simulation of genetic␣
↪recombination, genomic estimated breeding value calculations, and selection processes.␣
↪By utilizing GPU processing, it can perform these simulations up to two orders of␣
↪magnitude faster than existing tools with standard hardware. This offers breeders and␣
↪scientists new opportunities to simulate genetic gain and optimize breeding schemes.
↪The documentation is available at https://chromax.readthedocs.io. The code is␣
↪available at https://github.com/kora-labs/chromax.}",
    issn = {1367-4811},
    doi = {10.1093/bioinformatics/btad691},
    url = {https://doi.org/10.1093/bioinformatics/btad691},
    eprint = {https://academic.oup.com/bioinformatics/article-pdf/39/12/btad691/54143193/
↪btad691.pdf},
}
```

2.1 Simulator class

Module containing simulator class.

class chromax.simulator.**Simulator**(*genetic_map: Path | DataFrame*, *trait_names: List[str] | None = None*, *chr_column: str = 'CHR.PHYS'*, *position_column: str = 'cM'*, *recombination_column: str = 'RecombRate'*, *mutation_probability: float = 0.0*, *h2: ndarray | None = None*, *seed: int | None = None*, *device: Device = None*, *backend: str | Client = None*)

Breeding simulator class. It can perform the most common operation of a breeding program.

Parameters

- **genetic_map** (*Path or DataFrame*) – the path, or dataframe, containing the genetic map. It needs to have all the columns specified in trait_names, *CHR.PHYS* (with the name of the marker chromosome), and one between *cM* or *RecombRate*.
- **trait_names** (*List of strings*) – column names in the genetic_map. The values of the columns are the marker effects on the trait for each marker. The default value is *Yield*.
- **chr_column** (*str*) – name of the column containing the chromosome identifier. The default value is *CHR.PHYS*.
- **position_column** (*str*) – name of the column containing the position in cM of the marker. The default value is *cM*.
- **recombination_column** (*str*) – name of the column containing the probability that a recombination happens before the current marker and after the previous one. The default value is *RecombRate*.
- **mutation_probability** (*float*) – The probability of having a mutation in a marker.
- **h2** (*array of float*) – narrow-sense heritability value for each trait. The default value is 0.5 for each trait.
- **seed** (*int*) – the random seed for reproducibility.
- **device** (*XLA Device*) – the device for computing simulations. It will be automatically selected if not specified; by default to the first available GPU or TPU, or the CPU if neither is present.
- **backend** (*str or XLA client*) – the backend of the device. Common choices are *gpu*, *cpu* or *tpu*.

Example

```
>>> from chromax import Simulator, sample_data
>>> simulator = Simulator(genetic_map=sample_data.genetic_map)
>>> f1 = simulator.load_population(sample_data.genome)
>>> f2, _ = simulator.random_crosses(f1, n_crosses=10, n_offspring=20)
>>> f2.shape
(10, 20, 9839, 2)
```

set_seed(*seed: int*)

Set random seed for reproducibility.

Parameters

seed (*int*) – random seed.

load_population(*file_name: Path | str*) → Bool[Array, 'n m d']

Load a population from file.

Parameters
file_name (*path*) – path of the file with the population genome.

Returns
loaded population of shape (n, m, d), where n is the number of individual, m is the total number of marker, and d is the diploidy of the population.

Return type
ndarray

Example

```
>>> from chromax import Simulator, sample_data
>>> simulator = Simulator(genetic_map=sample_data.genetic_map)
>>> f1 = simulator.load_population(sample_data.genome)
>>> f1.shape
(371, 9839, 2)
```

save_population(*population: Bool[Array, 'n m d']*, *file_name: Path | str*)

Save a population to file.

Parameters
population (*ndarray*) – population to save.

File_name
file path to save the population.

Example

```
>>> from chromax import Simulator, sample_data
>>> simulator = Simulator(genetic_map=sample_data.genetic_map)
>>> f1 = simulator.load_population(sample_data.genome)
>>> f2, _ = simulator.random_crosses(f1, n_crosses=10, n_offspring=20)
>>> simulator.save_population(f2, "pop_file")
```

cross(*parents: Bool[Array, 'n 2 m d']*) → Bool[Array, 'n m d']

Main function that computes crosses from a list of parents.

Parameters
parents (*ndarray*) – parents to compute the cross. The shape of the parents is (n, 2, m, d), where n is the number of parents, m is the number of markers, and d is the ploidy.

Returns
offspring population of shape (n, m, d).

Return type
ndarray

Example

```
>>> from chromax import Simulator, sample_data
>>> import numpy as np
>>> simulator = Simulator(genetic_map=sample_data.genetic_map)
>>> f1 = simulator.load_population(sample_data.genome)
>>> parents_indices = np.array([
    [1, 5],
```

(continues on next page)

(continued from previous page)

```
    [4, 7],
    [5, 6]
])
>>> parents = f1[parents_indices]
>>> f2 = simulator.cross(parents)
>>> f2.shape
(3, 9839, 2)
```

property differentiable_cross_func: Callable

Experimental features that return a differentiable version of the cross function.

The differentiable crossing function takes as input:

- **population (array): starting population from which performing the crosses.**
  The shape of the population is (n, m, d).
- **cross_weights (array): Array of shape (l, n, d). It is used to compute**
  l crosses, starting from a weighted average of the n possible parents. When the n-axis has all zeros except of a single element equals to one, this function is equivalent to the cross function.
- random_key (JAX random key): random key used for recombination sampling.

And returns a population of shape (l, m, d).

Example

```
>>> from chromax import Simulator, sample_data
>>> import numpy as np
>>> import jax
>>> simulator = Simulator(genetic_map=sample_data.genetic_map)
>>> diff_cross = simulator.differentiable_cross_func
>>> def mean_gebv(pop, weights, random_key):
        new_pop = diff_cross(pop, weights, random_key)
        return simulator.GEBV(new_pop, raw_array=True).mean()
>>> grad_f = jax.grad(mean_gebv, argnums=1)
>>> f1 = simulator.load_population(sample_data.genome)
>>> weights = np.random.uniform(size=(10, len(f1), 2))
>>> weights /= weights.sum(axis=1, keepdims=True)
>>> random_key = jax.random.key(42)
>>> grad_value = grad_f(f1, weights, random_key)
>>> grad_value.shape
(10, 371, 2)
```

double_haploid(*population: Bool[Array, 'n m d']*, *n_offspring: int = 1*) → Bool[Array, 'n n_offspring m d']

Computes the double haploid of the input population.

Parameters

- **population** (*ndarray*) – input population of shape (n, m, 2).
- **n_offspring** (*int*) – number of offspring per plant. The default value is 1.

Returns

output population of shape (n, n_offspring, m, 2). This population will be homozygote.

Return type

ndarray

Example

```
>>> from chromax import Simulator, sample_data
>>> simulator = Simulator(genetic_map=sample_data.genetic_map)
>>> f1 = simulator.load_population(sample_data.genome)
>>> dh = simulator.double_haploid(f1, n_offspring=10)
>>> dh.shape
(371, 10, 9839, 2)
```

diallel(*population: Bool[Array, 'n m d']*, *n_offspring: int = 1*) → Bool[Array, 'n*(n-1)/2 n_offspring m d']

Diallel crossing function (crossing between every possible couple) except self-crossing.

Parameters

- **population** (*ndarray*) – input population of shape (n, m, d).
- **n_offspring** (*int*) – number of offspring per cross. The default value is 1.

Returns

output population of shape (l, n_offspring, m, d), where l is the number of possible pair, i.e *n * (n-1) / 2*.

Return type

ndarray

Example

```
>>> from chromax import Simulator, sample_data
>>> simulator = Simulator(genetic_map=sample_data.genetic_map)
>>> f1 = simulator.load_population(sample_data.genome)[:10]
>>> f2 = simulator.diallel(f1, n_offspring=10)
>>> f2.shape
(45, 10, 9839, 2)
```

random_crosses(*population: Bool[Array, 'n m d']*, *n_crosses: int*, *n_offspring: int = 1*) → Tuple[Bool[Array, 'n_crosses n_offspring m d'], Int[Array, 'n_crosses 2']]

Computes random crosses on a population.

Parameters

- **population** (*ndarray*) – input population of shape (n, m, d).
- **n_crosses** (*int*) – number of random crosses to perform.
- **n_offspring** (*int*) – number of offspring per cross. The default value is 1.

Returns

output population of shape (n_crosses, n_offspring, m, d) and parent indices of shape (n_crosses, 2) of performed crosses.

Return type

tuple of two ndarrays

Example

```
>>> from chromax import Simulator, sample_data
>>> simulator = Simulator(genetic_map=sample_data.genetic_map)
>>> f1 = simulator.load_population(sample_data.genome)
>>> f2, parent_ids = simulator.random_crosses(f1, 100, n_offspring=10)
>>> f2.shape
```

(continues on next page)

(continued from previous page)

```
(100, 10, 9839, 2)
>>> parent_ids.shape
(100, 2)
```

select(*population: Bool[Array, '_g n m d']*, *k: int*, *f_index: Callable[[Bool[Array, 'n m d']], Float[Array, 'n']] | None = None*) → Tuple[Bool[Array, '_g k m d'], Int[Array, '_g k']]

Function to select individuals based on their score (index).

Parameters

- **population** (*ndarray*) – input population of shape (n, m, d), or shape (g, n, m, d), to select k individual from each group population group g.
- **k** (*int*) – number of individual to select.
- **f_index** (*Callable*) – function that computes a score from each individual. The function accepts as input the population, i.e. and array of shape (n, m, d) and returns a n float numbers. The default f_index is the conventional index, i.e. the sum of the marker effects masked with the SNPs from the genetic_map.

Returns

output population of shape (k, m, d) or (g, k, m, d), depending on the input population, and respective indices of shape (k,) or (g, k)

Return type

tuple of two ndarrays

Example

```
>>> from chromax import Simulator, sample_data
>>> simulator = Simulator(genetic_map=sample_data.genetic_map, trait_
↪names=["Yield"])
>>> f1 = simulator.load_population(sample_data.genome)
>>> len(f1), simulator.GEBV(f1).mean().values
(371, [8.223844])
>>> f2, selected_indices = simulator.select(f1, k=20)
>>> len(f2), simulator.GEBV(f2).mean().values
(20, [14.595136])
>>> selected_indices.shape
(20,)
```

GEBV(*population: Bool[Array, 'n m d']*, ***, *raw_array: bool = False*) → DataFrame | ndarray

Computes the Genomic Estimated Breeding Values using the data from the genetic_map.

Parameters

- **population** (*ndarray*) – input population of shape (n, m, d).
- **raw_array** (*bool*) – whether to return a raw array or a DataFrame. Default value is False.

Returns

a DataFrame (or array) with n rows and a column for each trait. It contains the GEBV of each trait for each individual.

Return type

DataFrame or ndarray

Example

```
>>> from chromax import Simulator, sample_data
>>> simulator = Simulator(genetic_map=sample_data.genetic_map)
>>> f1 = simulator.load_population(sample_data.genome)
>>> simulator.GEBV(f1).mean()
Heading Date              0.196119
Protein Content          -0.228718
Plant Height             -5.888406
Thousand Kernel Weight   -1.029418
Yield                     8.223843
Fusarium Head Blight      5.318052
Spike Emergence Period   -0.933169
dtype: float32
```

create_environments(*num_environments: int*) → Float[Array, 'num_environments']

Create environments to phenotype the population.

In practice, it generates random numbers from a normal distribution.

Parameters
num_environments (*int*) – number of environments to create.

Returns
array of floating point numbers. This output can be used for the function *phenotype*.

Return type
ndarray

phenotype(*population: Bool[Array, 'n m d']*, ***, *num_environments: int | None = None*, *environments: ndarray | None = None*, *raw_array: bool = False*) → DataFrame | ndarray

Simulates the phenotype of a population.

This uses the Genotype-by-Environment model described in AlphaSimR.

Parameters

- **population** (*ndarray*) – input population of shape (n, m, d)
- **num_environments** (*int*) – number of environments to test the population. Default value is 1.
- **environments** (*ndarray*) – environments to test the population. Each environment must be represented by a floating number in the range (-1, 1). When drawing new environments use normal distribution to maintain heretability semantics.
- **raw_array** (*bool*) – whether to return a raw array or a DataFrame. Default value is False.

Returns
a DataFrame (or array) with n rows and a column for each trait. It contains the simulated phenotype for each individual.

Return type
DataFrame or ndarray

Example

```
>>> from chromax import Simulator, sample_data
>>> simulator = Simulator(genetic_map=sample_data.genetic_map,␣
↪seed=42)
>>> f1 = simulator.load_population(sample_data.genome)
```

(continues on next page)

(continued from previous page)

```
>>> envs = simulator.create_environments(4)
>>> simulator.phenotype(f1, environments=envs).mean()
Heading Date              0.105397
Protein Content          -0.172026
Plant Height             -5.813669
Thousand Kernel Weight   -1.372738
Yield                     8.306302
Fusarium Head Blight      4.286477
Spike Emergence Period   -0.575061
dtype: float32
```

corrcoef(*population: Bool[Array, 'n m d']*) → Float[Array, 'n']

Computes the correlation coefficient of the population against its centroid.

It can be used as an indicator of variance in the population.

Parameters

population (*ndarray*) – input population of shape (n, m, d)

Returns

vector of length n, containing the correlation coefficient of each individual against the average of the population.

Return type

ndarray

Example

```
>>> from chromax import Simulator, sample_data
>>> simulator = Simulator(genetic_map=sample_data.genetic_map,␣
↪seed=42)
>>> f1 = simulator.load_population(sample_data.genome)
>>> corrcoef = simulator.corrcoef(f1)
>>> corrcoef.shape
(371,)
```

2.2 Functional module

Functional module.

chromax.functional.**cross**(*parents: Bool[Array, 'n 2 m d']*, *recombination_vec: Float[Array, 'm']*, *random_key: Array*, *mutation_probability: float = 0.0*) → Bool[Array, 'n m d']

Main function that computes crosses from a list of parents.

Parameters

- **parents** (*ndarray*) – parents to compute the cross. The shape of the parents is (n, 2, m, d), where n is the number of parents, m is the number of markers, and d is the ploidy.
- **recombination_vec** – array of m probabilities. The i-th value represent the probability to recombine before the marker i.
- **random_key** (*jax.Array*) – JAX random key, for reproducibility purpose.
- **mutation_probability** (*float*) – The probability of having a mutation in a marker.

Returns
offspring population of shape (n, m, d).

Return type
ndarray

Example

```
>>> from chromax import functional
>>> import numpy as np
>>> import jax
>>> n_chr, chr_len, ploidy = 10, 100, 2
>>> n_crosses = 50
>>> parents_shape = (n_crosses, 2, n_chr * chr_len, ploidy)
>>> parents = np.random.choice([False, True], size=parents_shape)
>>> rec_vec = np.full((n_chr, chr_len), 1.5 / chr_len)
>>> rec_vec[:, 0] = 0.5  # equal probability on starting haploid
>>> rec_vec = rec_vec.flatten()
>>> random_key = jax.random.key(42)
>>> f2 = functional.cross(parents, rec_vec, random_key)
>>> f2.shape
(50, 1000, 2)
```

chromax.functional.**double_haploid**(*population: Bool[Array, 'n m d']*, *n_offspring: int*, *recombination_vec: Float[Array, 'm']*, *random_key: Array*, *mutation_probability: float = 0.0*) → Bool[Array, 'n n_offspring m d']

Computes the double haploid of the input population.

Parameters

- **population** (*ndarray*) – input population of shape (n, m, d).
- **n_offspring** (*int*) – number of offspring per plant.
- **recombination_vec** (*ndarray*) – array of m probabilities. The i-th value represent the probability to recombine before the marker i.
- **random_key** (*jax.Array*) – JAX random key, for reproducibility purpose.
- **mutation_probability** (*float*) – The probability of having a mutation in a marker.

Returns
output population of shape (n, n_offspring, m, d). This population will be homozygote.

Return type
ndarray

Example

```
>>> from chromax import functional
>>> import numpy as np
>>> import jax
>>> n_chr, chr_len, ploidy = 10, 100, 2
>>> pop_shape = (50, n_chr * chr_len, ploidy)
>>> f1 = np.random.choice([False, True], size=pop_shape)
>>> rec_vec = np.full((n_chr, chr_len), 1.5 / chr_len)
>>> rec_vec[:, 0] = 0.5  # equal probability on starting haploid
>>> rec_vec = rec_vec.flatten()
>>> random_key = jax.random.key(42)
```

(continues on next page)

(continued from previous page)

```
>>> dh = functional.double_haploid(f1, 10, rec_vec, random_key)
>>> dh.shape
(50, 10, 1000, 2)
```

chromax.functional.**select**(*population: Bool[Array, 'n m d']*, *k: int*, *f_index: Callable[[Bool[Array, 'n m d']], Float[Array, 'n']]*) → Tuple[Bool[Array, 'k m d'], Int[Array, 'k']]

Function to select individuals based on their score (index).

Parameters

- **population** (*ndarray*) – input grouped population of shape (n, m, d)
- **k** (*int*) – number of individual to select.
- **f_index** (*Callable*) – function that computes a score for each individual. The function accepts as input a population, i.e. and array of shape (n, m, 2) and returns an array of n float number.

Returns

output population of shape (k, m, d), output indices of shape (k,)

Return type

tuple of two ndarrays

Example

```
>>> from chromax import functional
>>> from chromax.trait_model import TraitModel
>>> from chromax.index_functions import conventional_index
>>> import numpy as np
>>> n_chr, chr_len, ploidy = 10, 100, 2
>>> pop_shape = (50, n_chr * chr_len, ploidy)
>>> f1 = np.random.choice([False, True], size=pop_shape)
>>> marker_effects = np.random.randn(n_chr * chr_len)
>>> gebv_model = TraitModel(marker_effects[:, None])
>>> f_index = conventional_index(gebv_model)
>>> f2, selected_indices = functional.select(f1, k=10, f_index=f_index)
>>> f2.shape
(10, 1000, 2)
```

2.3 Index functions

Utility module with common index functions.

chromax.index_functions.**phenotype_index**(*simulator*, *environments: ndarray*) → Callable

Function to select based on phenotyping with some environments.

Parameters

- **simulator** (*chromax.Simulator*) – chromax simulator instance to use for phenotyping simulation.
- **environments** – environments created using *create_environments* method from the simulator.

Returns
the phenotyping index function to use for selection.

Return type
Callable[[Population["n"]], Float[Array, "n"]]

chromax.index_functions.**conventional_index**(*GEBV_model: TraitModel*)

Function to select based on Genomic Estimated Breeding Value (GEBV).

Parameters
GEBV_model (*chromax.TraitModel*) – GEBV model to estimate the genomic breeding value. It must return a single value for an individual, i.e. estimate a single trait.

Returns
the conventional genomic selection index function.

Return type
Callable[[Population["n"]], Float[Array, "n"]]

chromax.index_functions.**visual_selection**(*simulator*, *noise_factor: int = 1*, *seed: int = None*) → Callable

Function to select based on visual selection.

Practically, this is similar to phenotyping but with more noise.

Parameters

- **simulator** (*chromax.Simulator*) – chromax simulator instance to use for phenotyping simulation.
- **noise_factor** (*int*) – variance ratio between the phenotype and artificial noise added to simulate visual selection inaccuracy.
- **seed** (*int*) – random seed for reproducibility.

Returns
the visual selection index function.

Return type
Callable[[Population["n"]], Float[Array, "n"]]

2.4 Data format

We explain here how we represent genomes in silico and how to format the genetic linkage map spreadsheet.

2.4.1 Genome data

We represent an individual using a boolean array with a shape of `(m, d)`, where *m* corresponds to the total number of markers, and *d* represents the ploidy. Thus, the genome data of a population is an array of shape `(n, m, d)`, where *n* is the size of the population. Each element in the boolean array represents a genetic variant. Note that we don't differentiate the various chromosomes on separate axes, but we do distinguish the haploids.

We load and save the genome data using the homonymous NumPy functions. As a result, the file format of the genome data follows the NPY format; for more information please consult the NumPy's documentation.

2.4.2 Genetic linkage map

The genetic linkage map is a spreadsheet that provides information about marker effects for traits, chromosome identifiers, and marker positions measured in *centiMorgans* (cM). The column labeled `CHR.PHYS` indicates the chromosome identifier, while the column labeled `cM` represents the marker position. Each trait has its own column in the spreadsheet. Each row in the spreadsheet corresponds to a single marker. Here's an example of how a spreadsheet for a genetic linkage map might appear:

CHR.PHYS	cM	Yield	Height	Protein
1A	0	0.0378	0.0189	0.0027
1A	32.79	0.0721	-0.0867	0.0010
...	...	...	...	...
7B	139.78	-0.0123	-0.0129	0.0031
7B	152.78	0.1082	0.1201	-0.0017

We deliver with the package a sample genome data and genetic linkage map adapted from *Wang, Shichen et al. "Characterization of polyploid wheat genomic diversity using a high-density 90 000 single nucleotide polymorphism array". Plant Biotechnology Journal 12. 6(2014): 787-796.*

2.5 Simulate a wheat breeding program

We describe here a sample breeding program to develop inbred cultivars. Specifically, we follow the conventional breeding scheme for wheat described in *Gaynor, R., et al. "A Two-Part Strategy for Using Genomic Selection to Develop Inbred Lines."* `

First of all, let's import the simulator and the module containing the index functions we will use to select the plants:

```
from chromax import Simulator
from chromax import index_functions
```

and then we initialize the simulator and load the population. For the example, we can use the sample data in the package:

```
from chromax.sample_data import genome, genetic_map
simulator = Simulator(
    genetic_map=genetic_map,
    trait_names=["Yield"],
    h2=[0.4]
)
f0 = simulator.load_population(genome)
```

We are interested in selecting based on *Yield* only and we set an heritability of `0.4`. We start the breeding program by performing some random crosses that produce a new population. Then, we obtain a line of plants from each individual by performing double haploid induction:

```
f1, _ = simulator.random_crosses(f0, 100)
dh_lines = simulator.double_haploid(f1, n_offspring=100)
```

In this way we obtain 100 lines, each containing 100 plants. We then start a typical bottleneck selection process, were we start with low accuracy methodologies to reduce the number of plants and we iterativeley increase the accuracy (and cost) of selection method. In particular, we start with a visual selection on the rows and then we test the plants on an increasing number of locations. The code will be like this:

```
headrows = simulator.select(
    dh_lines,
    k=5,
    f_index=index_functions.visual_selection(simulator, seed=7)
)
headrows = headrows.reshape(len(dh_lines) * 5, *dh_lines.shape[2:])

envs = simulator.create_environments(num_environments=16)
pyt = simulator.select(
    headrows,
    k=50,
    f_index=index_functions.phenotype_index(simulator, envs[0])
)
ayt = simulator.select(
    pyt,
    k=10,
    f_index=index_functions.phenotype_index(simulator, envs[:4])
)

released_variety = simulator.select(
    ayt,
    k=1,
    f_index=index_functions.phenotype_index(simulator, envs)
)
```

In this way we simulate the developing of a cultivar after a breeding cycle. If we want to continue with multiple cycle, we also need to compose the founder population of the next cycle. For example:

```
hdrw_next_cycle = simulator.select(
    dh_lines.reshape(dh_lines.shape[0] * dh_lines.shape[1], *dh_lines.shape[2:]),
    k=20,
    f_index=index_functions.visual_selection(simulator, seed=7)
)
pyt_next_cycle = simulator.select(
    headrows,
    k=20,
    f_index=index_functions.phenotype_index(simulator, envs[0])
)
next_cycle_f0 = np.concatenate(
    (pyt_next_cycle, ayt, hdrw_next_cycle),
    axis=0
)
```

And then repeating the breeding scheme using the *next_cycle_f0* as founder population. The code for the breeding scheme can be found here.

2.6 Distributed computation

We present here how to perform computation on multiple devices.

Consider a scenario where you have access to four GPUs and aim to distribute the workload effectively among them. To achieve this, we employ the JAX pmap function, which allows seamless distribution of functions across multiple accelerators.

```
from chromax import Simulator
from chromax.sample_data import genome, genetic_map
import jax

simulator = Simulator(genetic_map=genetic_map)
# load 200 individuals
population = simulator.load_population(genome)[:200]
# divide them in 4 groups
population = population.reshape(4, -1, *population.shape[1:])

# prepare a parallelized function over groups
pmap_dh = jax.pmap(
    simulator.double_haploid,
    in_axes=(0, None),
    static_broadcasted_argnums=1
)
# perform distributed computation
dh_pop = pmap_dh(population, 10)
# reshape to an ungrouped population
dh_pop = dh_pop.reshape(-1, *dh_pop.shape[2:])
```

If you want to perform random crosses or full diallel, grouping the population will change the semantics (the random crosses or the full diallel will be performed by group independently). In this case, you should use the function `cross` after generating the proper array of parents. For example, to perform random crosses:

```
from chromax import Simulator
from chromax.sample_data import genome, genetic_map
import numpy as np
import jax
simulator = Simulator(genetic_map=genetic_map)
population = simulator.load_population(genome)

random_indices = np.random.random_integers(0, len(population) - 1, size=(200, 2))
parents = population[random_indices]
parents = parents.reshape(4, -1, *parents.shape[1:])
pmap_cross = jax.pmap(simulator.cross,)
new_pop = pmap_cross(parents)
new_pop = new_pop.reshape(-1, *new_pop.shape[2:])
```

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

C

INDEX