scikit-allel Documentation

Release 1.1.0

Alistair Miles

Jun 21, 2017

Contents

This package provides utilities for exploratory analysis of large scale genetic variation data. It is based on numpy, scipy and other general-purpose Python scientific libraries.

- Source: https://github.com/cggh/scikit-allel
- Documentation: http://scikit-allel.readthedocs.org/
- Download: https://pypi.python.org/pypi/scikit-allel
- Gitter: https://gitter.im/cggh/pygenomics

Please feel free to ask questions via cggh/pygenomics on Gitter. Release announcements are posted to the cggh/pygenomics Gitter channel and the biovalidation mailing list. If you find a bug or would like to suggest a feature, please raise an issue on GitHub.

This site provides reference documentation for *scikit-allel*. For worked examples with real data, see the following articles:

- Extracting data from VCF files
- A tour of scikit-allel
-
- Fast PCA
- Mendelian transmission

If you would like to cite *scikit-allel* please use the DOI below.

CHAPTER 1

Why “scikit-allel”?

“SciKits” (short for SciPy Toolkits) are add-on packages for SciPy, hosted and developed separately and independently from the main SciPy distribution.

“Allel” (Greek $\lambda\lambda\lambda$) is the root of the word “allele” short for “allelomorph”, a word coined by William Bateson to mean variant forms of a gene. Today we use “allele” to mean any of the variant forms found at a site of genetic variation, such as the different nucleotides observed at a single nucleotide polymorphism (SNP).

CHAPTER 2

Installation

Pre-built binaries are available for Windows, Mac and Linux, and can be installed via conda:

```
$ conda install -c conda-forge scikit-allel
```

Alternatively, if you have a C compiler on your system, *scikit-allel* can be installed via pip:

```
$ pip install scikit-allel
```

N.B., *scikit-allel* requires numpy, scipy, matplotlib, seaborn, pandas, scikit-learn, h5py, numexpr, bcolz, zarr and dask. If installing via conda, these should be installed automatically. If installing via pip, please install these dependencies first, then use pip to install scikit-allel.

If you have never installed Python before, you might find the following article useful: Installing Python for data analysis

CHAPTER 3

Contributing

This is academic software, written in the cracks of free time between other commitments, by people who are often learning as we code. We greatly appreciate bug reports, pull requests, and any other feedback or advice. If you do find a bug, we'll do our best to fix it, but apologies in advance if we are not able to respond quickly. If you are doing any serious work with this package, please do not expect everything to work perfectly first time or be 100% correct. Treat everything with a healthy dose of suspicion, and don't be afraid to dive into the source code if you have to. Pull requests are always welcome.

CHAPTER 4

Contents

4.1 Data structures

4.1.1 In-memory data structures

GenotypeArray

class `allel.`**`GenotypeArray`**(*data*, *copy=False*, ***kwargs*)

Array of discrete genotype calls for a matrix of variants and samples.

Parameters **data** : array_like, int, shape (n_variants, n_samples, ploidy)

Genotype data.

copy : bool, optional

If True, make a copy of *data*.

****kwargs** : keyword arguments

All keyword arguments are passed through to `numpy.array()`.

See also:

`Genotypes`, `GenotypeVector`, `HaplotypeArray`, `GenotypeAlleleCountsArray`

Notes

This class represents data on discrete genotype calls as a 3-dimensional numpy array of integers. By convention, the first dimension corresponds to the variants genotyped, the second dimension corresponds to the samples genotyped, and the third dimension corresponds to the ploidy of the samples.

Each integer within the array corresponds to an **allele index**, where 0 is the reference allele, 1 is the first alternate allele, 2 is the second alternate allele, ... and -1 (or any other negative integer) is a missing allele call. A single byte integer dtype (int8) can represent up to 127 distinct alleles, which is usually sufficient. The actual alleles

(i.e., the alternate nucleotide sequences) and the physical positions of the variants within the genome of an organism are stored in separate arrays, discussed elsewhere.

Arrays of this class can store either **phased or unphased** genotype calls. If the genotypes are phased (i.e., haplotypes have been resolved) then individual haplotypes can be extracted by converting to a `HaplotypeArray` then indexing the second dimension. If the genotype calls are unphased then the ordering of alleles along the third (ploidy) dimension is arbitrary. N.B., this means that an unphased diploid heterozygous call could be stored as (0, 1) or equivalently as (1, 0).

A genotype array can store genotype calls with any ploidy > 1. For haploid calls, use a `HaplotypeArray`. Note that genotype arrays are not capable of storing calls for samples with differing or variable ploidy, see `GenotypeAlleleCountsArray` instead.

With genotype data on large numbers of variants and/or samples, storing the genotype calls in memory as an uncompressed numpy array if integers may be impractical. For working with large arrays of genotype data, see the *allel.model.chunked* and *allel.model.dask* modules.

Examples

Instantiate a genotype array:

```
>>> import allel
>>> g = allel.GenotypeArray([[[0, 0], [0, 1]],
...                          [[0, 1], [1, 1]],
...                          [[0, 2], [-1, -1]]], dtype='i1')
>>> g.dtype
dtype('int8')
>>> g.ndim
3
>>> g.shape
(3, 2, 2)
>>> g.n_variants
3
>>> g.n_samples
2
>>> g.ploidy
2
>>> g
<GenotypeArray shape=(3, 2, 2) dtype=int8>
0/0 0/1
0/1 1/1
0/2 ./.
```

Genotype calls for a single variant at all samples can be obtained by indexing the first dimension, e.g.:

```
>>> g[1]
<GenotypeVector shape=(2, 2) dtype=int8>
0/1 1/1
```

Genotype calls for a single sample at all variants can be obtained by indexing the second dimension, e.g.:

```
>>> g[:, 1]
<GenotypeVector shape=(3, 2) dtype=int8>
0/1 1/1 ./.
```

A genotype call for a single sample at a single variant can be obtained by indexing the first and second dimensions, e.g.:

```
>>> g[1, 0]
array([0, 1], dtype=int8)
```

A genotype array can store polyploid calls, e.g.:

```
>>> g = allel.GenotypeArray([[[0, 0, 0], [0, 0, 1]],
...                          [[0, 1, 1], [1, 1, 1]],
...                          [[0, 1, 2], [-1, -1, -1]]],
...                         dtype='i1')
>>> g.ploidy
3
>>> g
<GenotypeArray shape=(3, 2, 3) dtype=int8>
0/0/0 0/0/1
0/1/1 1/1/1
0/1/2 ././.
```

n_variants
Number of variants.

n_samples
Number of samples.

ploidy
Sample ploidy.

n_calls
Total number of genotype calls.

n_allele_calls
Total number of allele calls.

count_alleles(*max_allele=None*, *subpop=None*)
Count the number of calls of each allele per variant.

Parameters **max_allele** : int, optional

The highest allele index to count. Alleles above this will be ignored.

subpop : sequence of ints, optional

Indices of samples to include in count.

Returns **ac** : AlleleCountsArray

Examples

```
>>> import allel
>>> g = allel.GenotypeArray([[[0, 0], [0, 1]],
...                          [[0, 2], [1, 1]],
...                          [[2, 2], [-1, -1]]])
>>> g.count_alleles()
<AlleleCountsArray shape=(3, 3) dtype=int32>
3 1 0
1 2 1
0 0 2
>>> g.count_alleles(max_allele=1)
<AlleleCountsArray shape=(3, 2) dtype=int32>
3 1
```

```
1 2
0 0
```

count_alleles_subpops (*subpops*, *max_allele=None*)

Count alleles for multiple subpopulations simultaneously.

Parameters **subpops** : dict (string -> sequence of ints)

Mapping of subpopulation names to sample indices.

max_allele : int, optional

The highest allele index to count. Alleles above this will be ignored.

Returns **out** : dict (string -> AlleleCountsArray)

A mapping of subpopulation names to allele counts arrays.

to_packed (*boundscheck=True*)

Pack diploid genotypes into a single byte for each genotype, using the left-most 4 bits for the first allele and the right-most 4 bits for the second allele. Allows single byte encoding of diploid genotypes for variants with up to 15 alleles.

Parameters **boundscheck** : bool, optional

If False, do not check that minimum and maximum alleles are compatible with bit-packing.

Returns **packed** : ndarray, uint8, shape (n_variants, n_samples)

Bit-packed genotype array.

Notes

If a mask has been set, it is ignored by this function.

Examples

```
>>> import allel
>>> g = allel.GenotypeArray([[[0, 0], [0, 1]],
...                          [[0, 2], [1, 1]],
...                          [[2, 2], [-1, -1]]], dtype='i1')
>>> g.to_packed()
array([[  0,   1],
       [  2,  17],
       [ 34, 239]], dtype=uint8)
```

classmethod from_packed (*packed*)

Unpack diploid genotypes that have been bit-packed into single bytes.

Parameters **packed** : ndarray, uint8, shape (n_variants, n_samples)

Bit-packed diploid genotype array.

Returns **g** : GenotypeArray, shape (n_variants, n_samples, 2)

Genotype array.

Examples

```
>>> import allel
>>> import numpy as np
>>> packed = np.array([[0, 1],
...                    [2, 17],
...                    [34, 239]], dtype='u1')
>>> allel.GenotypeArray.from_packed(packed)
<GenotypeArray shape=(3, 2, 2) dtype=int8>
0/0 0/1
0/2 1/1
2/2 ./.
```

to_sparse (*format='csr'*, ***kwargs*)

Convert into a sparse matrix.

Parameters **format** : {'coo', 'csc', 'csr', 'dia', 'dok', 'lil'}

Sparse matrix format.

kwargs : keyword arguments

Passed through to sparse matrix constructor.

Returns **m** : scipy.sparse.spmatrix

Sparse matrix

Notes

If a mask has been set, it is ignored by this function.

Examples

```
>>> import allel
>>> g = allel.GenotypeArray([[[0, 0], [0, 0]],
...                          [[0, 1], [0, 1]],
...                          [[1, 1], [0, 0]],
...                          [[0, 0], [-1, -1]]], dtype='i1')
>>> m = g.to_sparse(format='csr')
>>> m
<4x4 sparse matrix of type '<class 'numpy.int8'>'
    with 6 stored elements in Compressed Sparse Row format>
>>> m.data
array([ 1,  1,  1,  1, -1, -1], dtype=int8)
>>> m.indices
array([1, 3, 0, 1, 2, 3], dtype=int32)
>>> m.indptr
array([0, 0, 2, 4, 6], dtype=int32)
```

static from_sparse (*m*, *ploidy*, *order=None*, *out=None*)

Construct a genotype array from a sparse matrix.

Parameters **m** : scipy.sparse.spmatrix

Sparse matrix

ploidy : int

The sample ploidy.

order : {'C', 'F'}, optional

Whether to store data in C (row-major) or Fortran (column-major) order in memory.

out : ndarray, shape (n_variants, n_samples), optional

Use this array as the output buffer.

Returns **g** : GenotypeArray, shape (n_variants, n_samples, ploidy)

Genotype array.

Examples

```
>>> import allel
>>> import numpy as np
>>> import scipy.sparse
>>> data = np.array([ 1,  1,  1,  1, -1, -1], dtype=np.int8)
>>> indices = np.array([1, 3, 0, 1, 2, 3], dtype=np.int32)
>>> indptr = np.array([0, 0, 2, 4, 6], dtype=np.int32)
>>> m = scipy.sparse.csr_matrix((data, indices, indptr))
>>> g = allel.GenotypeArray.from_sparse(m, ploidy=2)
>>> g
<GenotypeArray shape=(4, 2, 2) dtype=int8>
0/0 0/0
0/1 0/1
1/1 0/0
0/0 ./.
```

haploidify_samples()

Construct a pseudo-haplotype for each sample by randomly selecting an allele from each genotype call.

Returns **h** : HaplotypeArray

Notes

If a mask has been set, it is ignored by this function.

Examples

```
>>> import allel
>>> import numpy as np
>>> np.random.seed(42)
>>> g = allel.GenotypeArray([[[0, 0], [0, 1]],
...                          [[0, 2], [1, 1]],
...                          [[1, 2], [2, 1]],
...                          [[2, 2], [-1, -1]]])
>>> g.haploidify_samples()
<HaplotypeArray shape=(4, 2) dtype=int64>
0 1
0 1
1 1
2 .
>>> g = allel.GenotypeArray([[[0, 0, 0], [0, 0, 1]],
```

```
...                             [[0, 1, 1], [1, 1, 1]],
...                             [[0, 1, 2], [-1, -1, -1]]])
>>> g.haploidify_samples()
<HaplotypeArray shape=(3, 2) dtype=int64>
0 0
1 1
2 .
```

subset(*sel0=None*, *sel1=None*)

Make a sub-selection of variants and samples.

Parameters **sel0** : array_like

Boolean array or array of indices selecting variants.

sel1 : array_like

Boolean array or array of indices selecting samples.

Returns **out** : GenotypeArray

See also:

Genotypes.take, *Genotypes.compress*

Examples

```
>>> import allel
>>> g = allel.GenotypeArray([[[0, 0], [0, 1], [1, 1]],
...                          [[0, 1], [1, 1], [1, 2]],
...                          [[0, 2], [-1, -1], [-1, -1]]])
>>> g.subset([0, 1], [0, 2])
<GenotypeArray shape=(2, 2, 2) dtype=int64>
0/0 1/1
0/1 1/2
```

GenotypeVector

class allel.**GenotypeVector**(*data*, *copy=False*, ***kwargs*)

Array of genotype calls for a sequence of variants or samples.

Parameters **data** : array_like, int, shape (n_variants, n_samples, ploidy)

Genotype data.

copy : bool, optional

If True, make a copy of *data*.

****kwargs** : keyword arguments

All keyword arguments are passed through to `numpy.array()`.

See also:

`Genotypes`, `GenotypeArray`, `HaplotypeArray`, `GenotypeAlleleCountsArray`

n_calls

Total number of genotype calls.

ploidy
Sample ploidy.

n_allele_calls
Total number of allele calls.

Genotypes

Methods available on both `GenotypeArray` and `GenotypeVector` classes:

class `allel.`**Genotypes**(*data*, *copy=False*, ***kwargs*)
Base class for wrapping a NumPy array of genotype calls.

See also:

`GenotypeArray`, `GenotypeVector`

mask
A boolean mask, indicating genotype calls that should be filtered (i.e., excluded) from genotype and allele counting operations.

Notes

This is a lightweight genotype call mask and **not** a mask in the sense of a numpy masked array. This means that the mask will only be taken into account by the genotype and allele counting methods of this class, and is ignored by any of the generic methods on the ndarray class or by any numpy ufuncs.

Examples

```
>>> import allel
>>> g = allel.GenotypeArray([[[0, 0], [0, 1]],
...                          [[0, 1], [1, 1]],
...                          [[0, 2], [-1, -1]]], dtype='i1')
>>> g
<GenotypeArray shape=(3, 2, 2) dtype=int8>
0/0 0/1
0/1 1/1
0/2 ./.
>>> g.count_called()
5
>>> g.count_alleles()
<AlleleCountsArray shape=(3, 3) dtype=int32>
3 1 0
1 3 0
1 0 1
>>> v = g[:, 1]
>>> v
<GenotypeVector shape=(3, 2) dtype=int8>
0/1 1/1 ./.
>>> v.is_called()
array([ True,  True, False], dtype=bool)
>>> mask = [[True, False], [False, True], [False, False]]
>>> g.mask = mask
>>> g
<GenotypeArray shape=(3, 2, 2) dtype=int8>
```

```
./. 0/1
0/1 ./.
0/2 ./.
>>> g.count_called()
3
>>> g.count_alleles()
<AlleleCountsArray shape=(3, 3) dtype=int32>
1 1 0
1 1 0
1 0 1
>>> v = g[:, 1]
>>> v
<GenotypeVector shape=(3, 2) dtype=int8>
0/1 ./. ./.
>>> v.is_called()
array([ True, False, False], dtype=bool)
```

is_phased
A Boolean array indicating which genotype calls are phased and which are not.

Examples

```
>>> import allel
>>> g = allel.GenotypeArray([[[0, 0], [0, 1]],
...                          [[0, 1], [1, 1]],
...                          [[0, 2], [-1, -1]]], dtype='i1')
>>> g
<GenotypeArray shape=(3, 2, 2) dtype=int8>
0/0 0/1
0/1 1/1
0/2 ./.
>>> g.is_phased = [[True, True], [False, True], [False, False]]
>>> g
<GenotypeArray shape=(3, 2, 2) dtype=int8>
0|0 0|1
0/1 1|1
0/2 ./.
```

fill_masked(*value=-1*, *copy=True*)
Fill masked genotype calls with a given value.

Parameters **value** : int, optional

The fill value.

copy : bool, optional

If False, modify the array in place.

Returns **g** : GenotypeArray

Examples

```
>>> import allel
>>> g = allel.GenotypeArray([[[0, 0], [0, 1]],
...                          [[0, 1], [1, 1]],
```

```
...                             [[0, 2], [-1, -1]]], dtype='i1')
>>> mask = [[True, False], [False, True], [False, False]]
>>> g.mask = mask
>>> g.fill_masked().values
array([[[-1, -1],
        [ 0,  1]],
       [[ 0,  1],
        [-1, -1]],
       [[ 0,  2],
        [-1, -1]]], dtype=int8)
```

is_called()

Find non-missing genotype calls.

Returns **out** : ndarray, bool, shape (n_variants, n_samples)

Array where elements are True if the genotype call matches the condition.

Examples

```
>>> import allel
>>> g = allel.GenotypeArray([[[0, 0], [0, 1]],
...                          [[0, 1], [1, 1]],
...                          [[0, 2], [-1, -1]]])
>>> g.is_called()
array([[ True,  True],
       [ True,  True],
       [ True, False]], dtype=bool)
>>> v = g[:, 1]
>>> v
<GenotypeVector shape=(3, 2) dtype=int64>
0/1 1/1 ./.
>>> v.is_called()
array([ True,  True, False], dtype=bool)
```

is_missing()

Find missing genotype calls.

Returns **out** : ndarray, bool, shape (n_variants, n_samples)

Array where elements are True if the genotype call matches the condition.

Examples

```
>>> import allel
>>> g = allel.GenotypeArray([[[0, 0], [0, 1]],
...                          [[0, 1], [1, 1]],
...                          [[0, 2], [-1, -1]]])
>>> g.is_missing()
array([[False, False],
       [False, False],
       [False,  True]], dtype=bool)
>>> v = g[:, 1]
>>> v
<GenotypeVector shape=(3, 2) dtype=int64>
0/1 1/1 ./.
```

```
>>> v.is_missing()
array([False, False,  True], dtype=bool)
```

is_hom(*allele=None*)

Find genotype calls that are homozygous.

Parameters **allele** : int, optional

Allele index.

Returns **out** : ndarray, bool, shape (n_variants, n_samples)

Array where elements are True if the genotype call matches the condition.

Examples

```
>>> import allel
>>> g = allel.GenotypeArray([[[0, 0], [0, 1]],
...                          [[0, 1], [1, 1]],
...                          [[2, 2], [-1, -1]]])
>>> g.is_hom()
array([[ True, False],
       [False,  True],
       [ True, False]], dtype=bool)
>>> g.is_hom(allele=1)
array([[False, False],
       [False,  True],
       [False, False]], dtype=bool)
>>> v = g[:, 0]
>>> v
<GenotypeVector shape=(3, 2) dtype=int64>
0/0 0/1 2/2
>>> v.is_hom()
array([ True, False,  True], dtype=bool)
```

is_hom_ref()

Find genotype calls that are homozygous for the reference allele.

Returns **out** : ndarray, bool, shape (n_variants, n_samples)

Array where elements are True if the genotype call matches the condition.

Examples

```
>>> import allel
>>> g = allel.GenotypeArray([[[0, 0], [0, 1]],
...                          [[0, 1], [1, 1]],
...                          [[0, 2], [-1, -1]]])
>>> g.is_hom_ref()
array([[ True, False],
       [False, False],
       [False, False]], dtype=bool)
>>> v = g[:, 0]
>>> v
<GenotypeVector shape=(3, 2) dtype=int64>
0/0 0/1 0/2
```

```
>>> v.is_hom_ref()
array([ True, False, False], dtype=bool)
```

is_hom_alt()

Find genotype calls that are homozygous for any alternate (i.e., non-reference) allele.

Returns **out** : ndarray, bool, shape (n_variants, n_samples)

Array where elements are True if the genotype call matches the condition.

Examples

```
>>> import allel
>>> g = allel.GenotypeArray([[[0, 0], [0, 1]],
...                          [[0, 1], [1, 1]],
...                          [[2, 2], [-1, -1]]])
>>> g.is_hom_alt()
array([[False, False],
       [False,  True],
       [ True, False]], dtype=bool)
>>> v = g[:, 1]
>>> v
<GenotypeVector shape=(3, 2) dtype=int64>
0/1 1/1 ./.
>>> v.is_hom_alt()
array([False,  True, False], dtype=bool)
```

is_het(*allele=None*)

Find genotype calls that are heterozygous.

Returns **out** : ndarray, bool, shape (n_variants, n_samples)

Array where elements are True if the genotype call matches the condition.

allele : int, optional

Heterozygous allele.

Examples

```
>>> import allel
>>> g = allel.GenotypeArray([[[0, 0], [0, 1]],
...                          [[0, 1], [1, 1]],
...                          [[0, 2], [-1, -1]]])
>>> g.is_het()
array([[False,  True],
       [ True, False],
       [ True, False]], dtype=bool)
>>> g.is_het(2)
array([[False, False],
       [False, False],
       [ True, False]], dtype=bool)
>>> v = g[:, 0]
>>> v
<GenotypeVector shape=(3, 2) dtype=int64>
0/0 0/1 0/2
```

```
>>> v.is_het()
array([False,  True,  True], dtype=bool)
```

is_call(*call*)
Locate genotypes with a given call.

Parameters **call** : array_like, int, shape (ploidy,)

The genotype call to find.

Returns **out** : ndarray, bool, shape (n_variants, n_samples)

Array where elements are True if the genotype is *call*.

Examples

```
>>> import allel
>>> g = allel.GenotypeArray([[[0, 0], [0, 1]],
...                          [[0, 1], [1, 1]],
...                          [[0, 2], [-1, -1]]])
>>> g.is_call((0, 2))
array([[False, False],
       [False, False],
       [ True, False]], dtype=bool)
>>> v = g[:, 0]
>>> v
<GenotypeVector shape=(3, 2) dtype=int64>
0/0 0/1 0/2
>>> v.is_call((0, 2))
array([False, False,  True], dtype=bool)
```

count_called(*axis=None*)
Count called genotypes.

Parameters **axis** : int, optional

Axis over which to count, or None to perform overall count.

count_missing(*axis=None*)
Count missing genotypes.

Parameters **axis** : int, optional

Axis over which to count, or None to perform overall count.

count_hom(*allele=None*, *axis=None*)
Count homozygous genotypes.

Parameters **allele** : int, optional

Allele index.

axis : int, optional

Axis over which to count, or None to perform overall count.

count_hom_ref(*axis=None*)
Count homozygous reference genotypes.

Parameters **axis** : int, optional

Axis over which to count, or None to perform overall count.

count_hom_alt (*axis=None*)
Count homozygous alternate genotypes.

Parameters **axis** : int, optional

Axis over which to count, or None to perform overall count.

count_het (*allele=None*, *axis=None*)
Count heterozygous genotypes.

Parameters **allele** : int, optional

Allele index.

axis : int, optional

Axis over which to count, or None to perform overall count.

count_call (*call*, *axis=None*)
Count genotypes with a given call.

Parameters **call** : array_like, int, shape (ploidy,)

The genotype call to find.

axis : int, optional

Axis over which to count, or None to perform overall count.

to_n_ref (*fill=0*, *dtype='i1'*)
Transform each genotype call into the number of reference alleles.

Parameters **fill** : int, optional

Use this value to represent missing calls.

dtype : dtype, optional

Output dtype.

Returns **out** : ndarray, int8, shape (n_variants, n_samples)

Array of ref alleles per genotype call.

Notes

By default this function returns 0 for missing genotype calls **and** for homozygous non-reference genotype calls. Use the *fill* argument to change how missing calls are represented.

Examples

```
>>> import allel
>>> g = allel.GenotypeArray([[[0, 0], [0, 1]],
...                          [[0, 2], [1, 1]],
...                          [[2, 2], [-1, -1]]])
>>> g.to_n_ref()
array([[2, 1],
       [1, 0],
       [0, 0]], dtype=int8)
>>> g.to_n_ref(fill=-1)
array([[ 2,  1],
       [ 1,  0],
```

```
       [ 0, -1]], dtype=int8)
>>> v = g[:, 0]
>>> v
<GenotypeVector shape=(3, 2) dtype=int64>
0/0 0/2 2/2
>>> v.to_n_ref()
array([2, 1, 0], dtype=int8)
```

to_n_alt (*fill=0*, *dtype='i1'*)

Transform each genotype call into the number of non-reference alleles.

Parameters **fill** : int, optional

Use this value to represent missing calls.

dtype : dtype, optional

Output dtype.

Returns **out** : ndarray, int, shape (n_variants, n_samples)

Array of non-ref alleles per genotype call.

Notes

This function simply counts the number of non-reference alleles, it makes no distinction between different alternate alleles.

By default this function returns 0 for missing genotype calls **and** for homozygous reference genotype calls. Use the *fill* argument to change how missing calls are represented.

Examples

```
>>> import allel
>>> g = allel.GenotypeArray([[[0, 0], [0, 1]],
...                          [[0, 2], [1, 1]],
...                          [[2, 2], [-1, -1]]])
>>> g.to_n_alt()
array([[0, 1],
       [1, 2],
       [2, 0]], dtype=int8)
>>> g.to_n_alt(fill=-1)
array([[ 0,  1],
       [ 1,  2],
       [ 2, -1]], dtype=int8)
>>> v = g[:, 0]
>>> v
<GenotypeVector shape=(3, 2) dtype=int64>
0/0 0/2 2/2
>>> v.to_n_alt()
array([0, 1, 2], dtype=int8)
```

to_allele_counts (*max_allele=None*, *dtype='u1'*)

Transform genotype calls into allele counts per call.

Parameters **max_allele** : int, optional

Highest allele index. Provide this value to speed up computation.

dtype : dtype, optional

Output dtype.

Returns **out** : ndarray, uint8, shape (n_variants, n_samples, len(alleles))

Array of allele counts per call.

Examples

```
>>> import allel
>>> g = allel.GenotypeArray([[[0, 0], [0, 1]],
...                          [[0, 2], [1, 1]],
...                          [[2, 2], [-1, -1]]])
>>> g.to_allele_counts()
<GenotypeAlleleCountsArray shape=(3, 2, 3) dtype=uint8>
2:0:0 1:1:0
1:0:1 0:2:0
0:0:2 0:0:0
>>> v = g[:, 0]
>>> v
<GenotypeVector shape=(3, 2) dtype=int64>
0/0 0/2 2/2
>>> v.to_allele_counts()
<GenotypeAlleleCountsVector shape=(3, 3) dtype=uint8>
2:0:0 1:0:1 0:0:2
```

to_gt(*max_allele=None*)

Convert genotype calls to VCF-style string representation.

Returns **gt** : ndarray, string, shape (n_variants, n_samples)

Examples

```
>>> import allel
>>> g = allel.GenotypeArray([[[0, 0], [0, 1]],
...                          [[0, 2], [1, 1]],
...                          [[1, 2], [2, 1]],
...                          [[2, 2], [-1, -1]]])
>>> g.to_gt()
chararray([[b'0/0', b'0/1'],
       [b'0/2', b'1/1'],
       [b'1/2', b'2/1'],
       [b'2/2', b'./.']],
      dtype='|S3')
>>> v = g[:, 0]
>>> v
<GenotypeVector shape=(4, 2) dtype=int64>
0/0 0/2 1/2 2/2
>>> v.to_gt()
chararray([b'0/0', b'0/2', b'1/2', b'2/2'],
      dtype='|S3')
>>> g.is_phased = np.ones(g.shape[:-1], dtype=bool)
>>> g.to_gt()
chararray([[b'0|0', b'0|1'],
       [b'0|2', b'1|1'],
```

```
       [b'1|2', b'2|1'],
       [b'2|2', b'.|.']],
      dtype='|S3')
>>> v = g[:, 0]
>>> v
<GenotypeVector shape=(4, 2) dtype=int64>
0|0 0|2 1|2 2|2
>>> v.to_gt()
chararray([b'0|0', b'0|2', b'1|2', b'2|2'],
      dtype='|S3')
```

map_alleles(*mapping*, *copy=True*)

Transform alleles via a mapping.

Parameters **mapping** : ndarray, int8, shape (n_variants, max_allele)

An array defining the allele mapping for each variant.

copy : bool, optional

If True, return a new array; if False, apply mapping in place (only applies for arrays with dtype int8; all other dtypes require a copy).

Returns **gm** : GenotypeArray

See also:

`create_allele_mapping`

Notes

If a mask has been set, it is ignored by this function.

For arrays with dtype int8 an optimised implementation is used which is faster and uses far less memory. It is recommended to convert arrays to dtype int8 where possible before calling this method.

Examples

```
>>> import allel
>>> import numpy as np
>>> g = allel.GenotypeArray([[[0, 0], [0, 1]],
...                          [[0, 2], [1, 1]],
...                          [[1, 2], [2, 1]],
...                          [[2, 2], [-1, -1]]], dtype='i1')
>>> mapping = np.array([[1, 2, 0],
...                     [2, 0, 1],
...                     [2, 1, 0],
...                     [0, 2, 1]], dtype='i1')
>>> g.map_alleles(mapping)
<GenotypeArray shape=(4, 2, 2) dtype=int8>
1/1 1/2
2/1 0/0
1/0 0/1
1/1 ./.
>>> v = g[:, 0]
>>> v
<GenotypeVector shape=(4, 2) dtype=int8>
```

```
0/0 0/2 1/2 2/2
>>> v.map_alleles(mapping)
<GenotypeVector shape=(4, 2) dtype=int8>
1/1 2/1 1/0 1/1
```

compress(*condition*, *axis=0*, *out=None*)

Return selected slices of an array along given axis.

Parameters **condition** : array_like, bool

Array that selects which entries to return. N.B., if len(condition) is less than the size of the given axis, then output is truncated to the length of the condition array.

axis : int, optional

Axis along which to take slices. If None, work on the flattened array.

out : ndarray, optional

Output array. Its type is preserved and it must be of the right shape to hold the output.

Returns **out** : Genotypes

A copy of the array without the slices along axis for which *condition* is false.

Examples

```
>>> import allel
>>> g = allel.GenotypeArray([[[0, 0], [0, 1]],
...                          [[0, 1], [1, 1]],
...                          [[0, 2], [-1, -1]]])
>>> g.compress([True, False, True], axis=0)
<GenotypeArray shape=(2, 2, 2) dtype=int64>
0/0 0/1
0/2 ./.
>>> v = g[:, 0]
>>> v
<GenotypeVector shape=(3, 2) dtype=int64>
0/0 0/1 0/2
>>> v.compress([True, False, True], axis=0)
<GenotypeVector shape=(2, 2) dtype=int64>
0/0 0/2
```

take(*indices*, *axis=0*, *out=None*, *mode='raise'*)

Take elements from an array along an axis.

This function does the same thing as "fancy" indexing (indexing arrays using arrays); however, it can be easier to use if you need elements along a given axis.

Parameters **indices** : array_like

The indices of the values to extract.

axis : int, optional

The axis over which to select values.

out : ndarray, optional

If provided, the result will be placed in this array. It should be of the appropriate shape and dtype.

mode : {'raise', 'wrap', 'clip'}, optional

Specifies how out-of-bounds indices will behave.

- 'raise' – raise an error (default)
- 'wrap' – wrap around
- 'clip' – clip to the range

'clip' mode means that all indices that are too large are replaced by the index that addresses the last element along that axis. Note that this disables indexing with negative numbers.

Returns **subarray** : ndarray

Examples

```
>>> import allel
>>> g = allel.GenotypeArray([[[0, 0], [0, 1]],
...                          [[0, 1], [1, 1]],
...                          [[0, 2], [-1, -1]]])
>>> g.take([0, 2], axis=0)
<GenotypeArray shape=(2, 2, 2) dtype=int64>
0/0 0/1
0/2 ./.
>>> v = g[:, 0]
>>> v
<GenotypeVector shape=(3, 2) dtype=int64>
0/0 0/1 0/2
>>> v.take([0, 2], axis=0)
<GenotypeVector shape=(2, 2) dtype=int64>
0/0 0/2
```

concatenate(*others*, *axis=0*)

Join a sequence of arrays along an existing axis.

Parameters **others** : sequence of array_like

The arrays must have the same shape, except in the dimension corresponding to *axis* (the first, by default).

axis : int, optional

The axis along which the arrays will be joined. Default is 0.

Returns **res** : ndarray

The concatenated array.

Examples

```
>>> import allel
>>> g = allel.GenotypeArray([[[0, 0], [0, 1]],
...                          [[0, 1], [1, 1]],
...                          [[0, 2], [-1, -1]]])
>>> g.concatenate([g], axis=0)
<GenotypeArray shape=(6, 2, 2) dtype=int64>
0/0 0/1
```

```
0/1 1/1
0/2 ./.
0/0 0/1
0/1 1/1
0/2 ./.
>>> g.concatenate([g], axis=1)
<GenotypeArray shape=(3, 4, 2) dtype=int64>
0/0 0/1 0/0 0/1
0/1 1/1 0/1 1/1
0/2 ./. 0/2 ./.
>>> v1 = g[:, 0]
>>> v2 = g[:, 1]
>>> v1.concatenate([v2], axis=0)
<GenotypeVector shape=(6, 2) dtype=int64>
0/0 0/1 0/2 0/1 1/1 ./.
```

HaplotypeArray

class allel.**HaplotypeArray**(*data*, *copy=False*, ***kwargs*)

Array of haplotypes.

Parameters **data** : array_like, int, shape (n_variants, n_haplotypes)

Haplotype data.

****kwargs** : keyword arguments

All keyword arguments are passed through to `numpy.array()`.

See also:

`Genotypes`, `GenotypeArray`, `GenotypeVector`

Notes

This class represents haplotype data as a 2-dimensional numpy array of integers. By convention the first dimension corresponds to the variants genotyped, the second dimension corresponds to the haplotypes.

Each integer within the array corresponds to an **allele index**, where 0 is the reference allele, 1 is the first alternate allele, 2 is the second alternate allele, ... and -1 (or any other negative integer) is a missing allele call.

If adjacent haplotypes originate from the same sample, then a haplotype array can also be viewed as a genotype array. However, this is not a requirement.

With data on large numbers of variants and/or haplotypes, storing the data in memory as an uncompressed numpy array if integers may be impractical. For working with large arrays of haplotype data, see the *allel.model.chunked* and *allel.model.dask* modules.

Examples

Instantiate a haplotype array:

```
>>> import allel
>>> h = allel.HaplotypeArray([[0, 0, 0, 1],
...                           [0, 1, 1, 1],
...                           [0, 2, -1, -1]], dtype='i1')
```

```
>>> h.dtype
dtype('int8')
>>> h.ndim
2
>>> h.shape
(3, 4)
>>> h.n_variants
3
>>> h.n_haplotypes
4
>>> h
<HaplotypeArray shape=(3, 4) dtype=int8>
0 0 0 1
0 1 1 1
0 2 . .
```

Allele calls for a single variant at all haplotypes can be obtained by indexing the first dimension, e.g.:

```
>>> h[1]
array([0, 1, 1, 1], dtype=int8)
```

A single haplotype can be obtained by indexing the second dimension, e.g.:

```
>>> h[:, 1]
array([0, 1, 2], dtype=int8)
```

An allele call for a single haplotype at a single variant can be obtained by indexing the first and second dimensions, e.g.:

```
>>> h[1, 0]
0
```

View haplotypes as diploid genotypes:

```
>>> h.to_genotypes(ploidy=2)
<GenotypeArray shape=(3, 2, 2) dtype=int8>
0/0 0/1
0/1 1/1
0/2 ./.
```

n_variants
Number of variants.

n_haplotypes
Number of haplotypes.

subset (*sel0=None*, *sel1=None*)
Make a sub-selection of variants and haplotypes.

Parameters **sel0** : array_like

Boolean array or array of indices selecting variants.

sel1 : array_like

Boolean array or array of indices selecting haplotypes.

Returns **out** : HaplotypeArray

See also:

HaplotypeArray.take, *HaplotypeArray.compress*

is_called()

is_missing()

is_ref()

is_alt(*allele=None*)

is_call(*allele*)

count_called(*axis=None*)

count_missing(*axis=None*)

count_ref(*axis=None*)

count_alt(*axis=None*)

count_call(*allele*, *axis=None*)

count_alleles(*max_allele=None*, *subpop=None*)

Count the number of calls of each allele per variant.

Parameters **max_allele** : int, optional

The highest allele index to count. Alleles greater than this index will be ignored.

subpop : array_like, int, optional

Indices of haplotypes to include.

Returns **ac** : AlleleCountsArray, int, shape (n_variants, n_alleles)

Examples

```
>>> import allel
>>> h = allel.HaplotypeArray([[0, 0, 0, 1],
...                           [0, 1, 1, 1],
...                           [0, 2, -1, -1]], dtype='i1')
>>> ac = h.count_alleles()
>>> ac
<AlleleCountsArray shape=(3, 3) dtype=int32>
3 1 0
1 3 0
1 0 1
```

count_alleles_subpops(*subpops*, *max_allele=None*)

Count alleles for multiple subpopulations simultaneously.

Parameters **subpops** : dict (string -> sequence of ints)

Mapping of subpopulation names to sample indices.

max_allele : int, optional

The highest allele index to count. Alleles above this will be ignored.

Returns **out** : dict (string -> AlleleCountsArray)

A mapping of subpopulation names to allele counts arrays.

map_alleles(*mapping*, *copy=True*)

Transform alleles via a mapping.

Parameters **mapping** : ndarray, int8, shape (n_variants, max_allele)

An array defining the allele mapping for each variant.

copy : bool, optional

If True, return a new array; if False, apply mapping in place (only applies for arrays with dtype int8; all other dtypes require a copy).

Returns **hm** : HaplotypeArray

See also:

`allel.model.util.create_allele_mapping`

Notes

For arrays with dtype int8 an optimised implementation is used which is faster and uses far less memory. It is recommended to convert arrays to dtype int8 where possible before calling this method.

Examples

```
>>> import allel
>>> import numpy as np
>>> h = allel.HaplotypeArray([[0, 0, 0, 1],
...                           [0, 1, 1, 1],
...                           [0, 2, -1, -1]], dtype='i1')
>>> mapping = np.array([[1, 2, 0],
...                     [2, 0, 1],
...                     [2, 1, 0]], dtype='i1')
>>> h.map_alleles(mapping)
<HaplotypeArray shape=(3, 4) dtype=int8>
1 1 1 2
2 0 0 0
2 0 . .
```

to_genotypes(*ploidy*, *copy=False*)

Reshape a haplotype array to view it as genotypes by restoring the ploidy dimension.

Parameters **ploidy** : int

The sample ploidy.

copy : bool, optional

If True, make a copy of data.

Returns **g** : ndarray, int, shape (n_variants, n_samples, ploidy)

Genotype array (sharing same underlying buffer).

copy : bool, optional

If True, copy the data.

Examples

```
>>> import allel
>>> h = allel.HaplotypeArray([[0, 0, 0, 1],
...                           [0, 1, 1, 1],
...                           [0, 2, -1, -1]], dtype='i1')
>>> h.to_genotypes(ploidy=2)
<GenotypeArray shape=(3, 2, 2) dtype=int8>
0/0 0/1
0/1 1/1
0/2 ./.
```

to_sparse (*format='csr'*, ***kwargs*)

Convert into a sparse matrix.

Parameters **format** : {'coo', 'csc', 'csr', 'dia', 'dok', 'lil'}

Sparse matrix format.

kwargs : keyword arguments

Passed through to sparse matrix constructor.

Returns **m** : scipy.sparse.spmatrix

Sparse matrix

Examples

```
>>> import allel
>>> h = allel.HaplotypeArray([[0, 0, 0, 0],
...                           [0, 1, 0, 1],
...                           [1, 1, 0, 0],
...                           [0, 0, -1, -1]], dtype='i1')
>>> m = h.to_sparse(format='csr')
>>> m
<4x4 sparse matrix of type '<class 'numpy.int8'>'
    with 6 stored elements in Compressed Sparse Row format>
>>> m.data
array([ 1,  1,  1,  1, -1, -1], dtype=int8)
>>> m.indices
array([1, 3, 0, 1, 2, 3], dtype=int32)
>>> m.indptr
array([0, 0, 2, 4, 6], dtype=int32)
```

static **from_sparse** (*m*, *order=None*, *out=None*)

Construct a haplotype array from a sparse matrix.

Parameters **m** : scipy.sparse.spmatrix

Sparse matrix

order : {'C', 'F'}, optional

Whether to store data in C (row-major) or Fortran (column-major) order in memory.

out : ndarray, shape (n_variants, n_samples), optional

Use this array as the output buffer.

Returns **h** : HaplotypeArray, shape (n_variants, n_haplotypes)

Haplotype array.

Examples

```
>>> import allel
>>> import numpy as np
>>> import scipy.sparse
>>> data = np.array([ 1,  1,  1,  1, -1, -1], dtype=np.int8)
>>> indices = np.array([1, 3, 0, 1, 2, 3], dtype=np.int32)
>>> indptr = np.array([0, 0, 2, 4, 6], dtype=np.int32)
>>> m = scipy.sparse.csr_matrix((data, indices, indptr))
>>> h = allel.HaplotypeArray.from_sparse(m)
>>> h
<HaplotypeArray shape=(4, 4) dtype=int8>
0 0 0 0
0 1 0 1
1 1 0 0
0 0 . .
```

prefix_argsort()
Return indices that would sort the haplotypes by prefix.

distinct()
Return sets of indices for each distinct haplotype.

distinct_counts()
Return counts for each distinct haplotype.

distinct_frequencies()
Return frequencies for each distinct haplotype.

compress(*condition*, *axis=0*, *out=None*)
Return selected slices of an array along given axis.

Parameters **condition** : array_like, bool

Array that selects which entries to return. N.B., if len(condition) is less than the size of the given axis, then output is truncated to the length of the condition array.

axis : int, optional

Axis along which to take slices. If None, work on the flattened array.

out : ndarray, optional

Output array. Its type is preserved and it must be of the right shape to hold the output.

Returns **out** : HaplotypeArray

A copy of the array without the slices along axis for which *condition* is false.

Examples

```
>>> import allel
>>> h = allel.HaplotypeArray([[0, 0, 0, 1],
...                           [0, 1, 1, 1],
...                           [0, 2, -1, -1]], dtype='i1')
>>> h.compress([True, False, True], axis=0)
<HaplotypeArray shape=(2, 4) dtype=int8>
```

```
0 0 0 1
0 2 . .
>>> h.compress([True, False, True, False], axis=1)
<HaplotypeArray shape=(3, 2) dtype=int8>
0 0
0 1
0 .
```

take(*indices*, *axis=0*, *out=None*, *mode='raise'*)

Take elements from an array along an axis.

This function does the same thing as "fancy" indexing (indexing arrays using arrays); however, it can be easier to use if you need elements along a given axis.

Parameters **indices** : array_like

The indices of the values to extract.

axis : int, optional

The axis over which to select values.

out : ndarray, optional

If provided, the result will be placed in this array. It should be of the appropriate shape and dtype.

mode : {'raise', 'wrap', 'clip'}, optional

Specifies how out-of-bounds indices will behave.

- 'raise' – raise an error (default)
- 'wrap' – wrap around
- 'clip' – clip to the range

'clip' mode means that all indices that are too large are replaced by the index that addresses the last element along that axis. Note that this disables indexing with negative numbers.

Returns **subarray** : ndarray

Examples

```
>>> import allel
>>> h = allel.HaplotypeArray([[0, 0, 0, 1],
...                           [0, 1, 1, 1],
...                           [0, 2, -1, -1]], dtype='i1')
>>> h.take([0, 2], axis=0)
<HaplotypeArray shape=(2, 4) dtype=int8>
0 0 0 1
0 2 . .
>>> h.take([0, 2], axis=1)
<HaplotypeArray shape=(3, 2) dtype=int8>
0 0
0 1
0 .
```

subset(*sel0=None*, *sel1=None*)

Make a sub-selection of variants and haplotypes.

Parameters **sel0** : array_like

Boolean array or array of indices selecting variants.

sel1 : array_like

Boolean array or array of indices selecting haplotypes.

Returns **out** : HaplotypeArray

See also:

HaplotypeArray.take, *HaplotypeArray.compress*

concatenate(*others*, *axis=0*)

Join a sequence of arrays along an existing axis.

Parameters **others** : sequence of array_like

The arrays must have the same shape, except in the dimension corresponding to *axis* (the first, by default).

axis : int, optional

The axis along which the arrays will be joined. Default is 0.

Returns **res** : ndarray

The concatenated array.

Examples

```
>>> import allel
>>> h = allel.HaplotypeArray([[0, 0, 0, 1],
...                           [0, 1, 1, 1],
...                           [0, 2, -1, -1]], dtype='i1')
>>> h.concatenate([h], axis=0)
<HaplotypeArray shape=(6, 4) dtype=int8>
0 0 0 1
0 1 1 1
0 2 . .
0 0 0 1
0 1 1 1
0 2 . .
>>> h.concatenate([h], axis=1)
<HaplotypeArray shape=(3, 8) dtype=int8>
0 0 0 1 0 0 0 1
0 1 1 1 0 1 1 1
0 2 . . 0 2 . .
```

AlleleCountsArray

class allel.**AlleleCountsArray**(*data*, *copy=False*, ***kwargs*)

Array of allele counts.

Parameters **data** : array_like, int, shape (n_variants, n_alleles)

Allele counts data.

copy : bool, optional

If True, make a copy of *data*.

****kwargs** : keyword arguments

All keyword arguments are passed through to `numpy.array()`.

Notes

This class represents allele counts as a 2-dimensional numpy array of integers. By convention the first dimension corresponds to the variants genotyped, the second dimension corresponds to the alleles counted.

Examples

Obtain allele counts from a genotype array:

```
>>> import allel
>>> g = allel.GenotypeArray([[[0, 0], [0, 1]],
...                          [[0, 1], [1, 1]],
...                          [[0, 2], [-1, -1]]], dtype='i1')
>>> ac = g.count_alleles()
>>> ac
<AlleleCountsArray shape=(3, 3) dtype=int32>
3 1 0
1 3 0
1 0 1
>>> ac.dtype
dtype('int32')
>>> ac.shape
(3, 3)
>>> ac.n_variants
3
>>> ac.n_alleles
3
```

Allele counts for a single variant can be obtained by indexing the first dimension, e.g.:

```
>>> ac[1]
array([1, 3, 0], dtype=int32)
```

Allele counts for a specific allele can be obtained by indexing the second dimension, e.g., reference allele counts:

```
>>> ac[:, 0]
array([3, 1, 1], dtype=int32)
```

Calculate the total number of alleles called for each variant:

```
>>> import numpy as np
>>> an = np.sum(ac, axis=1)
>>> an
array([4, 4, 2])
```

Add allele counts from two populations:

```
>>> ac + ac
<AlleleCountsArray shape=(3, 3) dtype=int32>
6 2 0
```

```
2 6 0
2 0 2
```

n_variants
Number of variants.

n_alleles
Number of alleles.

max_allele()
Return the highest allele index for each variant.

Returns **n** : ndarray, int, shape (n_variants,)

Allele index array.

Examples

```
>>> import allel
>>> g = allel.GenotypeArray([[[0, 0], [0, 1]],
...                          [[0, 2], [1, 1]],
...                          [[2, 2], [-1, -1]]])
>>> ac = g.count_alleles()
>>> ac.max_allele()
array([1, 2, 2], dtype=int8)
```

allelism()
Determine the number of distinct alleles observed for each variant.

Returns **n** : ndarray, int, shape (n_variants,)

Allelism array.

Examples

```
>>> import allel
>>> g = allel.GenotypeArray([[[0, 0], [0, 1]],
...                          [[0, 2], [1, 1]],
...                          [[2, 2], [-1, -1]]])
>>> ac = g.count_alleles()
>>> ac.allelism()
array([2, 3, 1])
```

is_variant()
Find variants with at least one non-reference allele call.

Returns **out** : ndarray, bool, shape (n_variants,)

Boolean array where elements are True if variant matches the condition.

Examples

```
>>> import allel
>>> g = allel.GenotypeArray([[[0, 0], [0, 0]],
...                          [[0, 0], [0, 1]],
```

```
...                             [[0, 2], [1, 1]],
...                             [[2, 2], [-1, -1]]])
>>> ac = g.count_alleles()
>>> ac.is_variant()
array([False,  True,  True,  True], dtype=bool)
```

is_non_variant()
Find variants with no non-reference allele calls.

Returns **out** : ndarray, bool, shape (n_variants,)

Boolean array where elements are True if variant matches the condition.

Examples

```
>>> import allel
>>> g = allel.GenotypeArray([[[0, 0], [0, 0]],
...                          [[0, 0], [0, 1]],
...                          [[0, 2], [1, 1]],
...                          [[2, 2], [-1, -1]]])
>>> ac = g.count_alleles()
>>> ac.is_non_variant()
array([ True, False, False, False], dtype=bool)
```

is_segregating()
Find segregating variants (where more than one allele is observed).

Returns **out** : ndarray, bool, shape (n_variants,)

Boolean array where elements are True if variant matches the condition.

Examples

```
>>> import allel
>>> g = allel.GenotypeArray([[[0, 0], [0, 0]],
...                          [[0, 0], [0, 1]],
...                          [[0, 2], [1, 1]],
...                          [[2, 2], [-1, -1]]])
>>> ac = g.count_alleles()
>>> ac.is_segregating()
array([False,  True,  True, False], dtype=bool)
```

is_non_segregating(*allele=None*)
Find non-segregating variants (where at most one allele is observed).

Parameters **allele** : int, optional

Allele index.

Returns **out** : ndarray, bool, shape (n_variants,)

Boolean array where elements are True if variant matches the condition.

Examples

```
>>> import allel
>>> g = allel.GenotypeArray([[[0, 0], [0, 0]],
...                          [[0, 0], [0, 1]],
...                          [[0, 2], [1, 1]],
...                          [[2, 2], [-1, -1]]])
>>> ac = g.count_alleles()
>>> ac.is_non_segregating()
array([ True, False, False,  True], dtype=bool)
>>> ac.is_non_segregating(allele=2)
array([False, False, False,  True], dtype=bool)
```

is_singleton(*allele*)

Find variants with a single call for the given allele.

Parameters **allele** : int, optional

Allele index.

Returns **out** : ndarray, bool, shape (n_variants,)

Boolean array where elements are True if variant matches the condition.

Examples

```
>>> import allel
>>> g = allel.GenotypeArray([[[0, 0], [0, 0]],
...                          [[0, 0], [0, 1]],
...                          [[1, 1], [1, 2]],
...                          [[2, 2], [-1, -1]]])
>>> ac = g.count_alleles()
>>> ac.is_singleton(allele=1)
array([False,  True, False, False], dtype=bool)
>>> ac.is_singleton(allele=2)
array([False, False,  True, False], dtype=bool)
```

is_doubleton(*allele*)

Find variants with exactly two calls for the given allele.

Parameters **allele** : int, optional

Allele index.

Returns **out** : ndarray, bool, shape (n_variants,)

Boolean array where elements are True if variant matches the condition.

Examples

```
>>> import allel
>>> g = allel.GenotypeArray([[[0, 0], [0, 0]],
...                          [[0, 0], [1, 1]],
...                          [[1, 1], [1, 2]],
...                          [[2, 2], [-1, -1]]])
>>> ac = g.count_alleles()
>>> ac.is_doubleton(allele=1)
```

```
array([False,  True, False, False], dtype=bool)
>>> ac.is_doubleton(allele=2)
array([False, False, False,  True], dtype=bool)
```

is_biallelic()
Find biallelic variants.

Returns **out** : ndarray, bool, shape (n_variants,)

Boolean array where elements are True if variant matches the condition.

is_biallelic_01(*min_mac=None*)
Find variants biallelic for the reference (0) and first alternate (1) allele.

Parameters **min_mac** : int, optional

Minimum minor allele count.

Returns **out** : ndarray, bool, shape (n_variants,)

Boolean array where elements are True if variant matches the condition.

count_variant()

count_non_variant()

count_segregating()

count_non_segregating(*allele=None*)

count_singleton(*allele=1*)

count_doubleton(*allele=1*)

to_frequencies(*fill=nan*)
Compute allele frequencies.

Parameters **fill** : float, optional

Value to use when number of allele calls is 0.

Returns **af** : ndarray, float, shape (n_variants, n_alleles)

Examples

```
>>> import allel
>>> g = allel.GenotypeArray([[[0, 0], [0, 1]],
...                          [[0, 2], [1, 1]],
...                          [[2, 2], [-1, -1]]])
>>> ac = g.count_alleles()
>>> ac.to_frequencies()
array([[ 0.75,  0.25,  0.  ],
       [ 0.25,  0.5 ,  0.25],
       [ 0.  ,  0.  ,  1.  ]])
```

map_alleles(*mapping*)
Transform alleles via a mapping.

Parameters **mapping** : ndarray, int8, shape (n_variants, max_allele)

An array defining the allele mapping for each variant.

Returns **ac** : AlleleCountsArray

See also:

`create_allele_mapping`

Examples

```
>>> import allel
>>> g = allel.GenotypeArray([[[0, 0], [0, 0]],
...                          [[0, 0], [0, 1]],
...                          [[0, 2], [1, 1]],
...                          [[2, 2], [-1, -1]]])
>>> ac = g.count_alleles()
>>> ac
<AlleleCountsArray shape=(4, 3) dtype=int32>
4 0 0
3 1 0
1 2 1
0 0 2
>>> mapping = [[1, 0, 2],
...            [1, 0, 2],
...            [2, 1, 0],
...            [1, 2, 0]]
>>> ac.map_alleles(mapping)
<AlleleCountsArray shape=(4, 3) dtype=int64>
0 4 0
1 3 0
1 2 1
2 0 0
```

compress(*condition*, *axis=0*, *out=None*)

take(*indices*, *axis=0*, *out=None*, *mode='raise'*)

concatenate(*others*, *axis=0*)

GenotypeAlleleCountsArray

class allel.**GenotypeAlleleCountsArray**(*data*, *copy=False*, ***kwargs*)

Array of genotype calls for a matrix of variants and samples, stored as allele counts per call.

Parameters **data** : array_like, int, shape (n_variants, n_samples, ploidy)

Genotype data.

copy : bool, optional

If True, make a copy of *data*.

****kwargs** : keyword arguments

All keyword arguments are passed through to `numpy.array()`.

See also:

`GenotypeAlleleCounts`, `GenotypeAlleleCountsVector`, `GenotypeArray`

Notes

This class provides an alternative representation of genotype calls, allowing for variable copy number (effective ploidy) between chromosomes and/or genome regions. Genotype calls are represented as a 3-dimensional array of integers. By convention, the first dimension corresponds to the variants genotyped, the second dimension corresponds to the samples genotyped, and the third dimension corresponds to the alleles genotyped in index order. Each integer in the array records the **count** for the given in allele in the given variant and sample.

Examples

Instantiate an array:

```
>>> import allel
>>> g = allel.GenotypeAlleleCountsArray([[[2, 0, 0], [0, 2, 0]],
...                                      [[1, 1, 0], [0, 1, 1]],
...                                      [[0, 0, 4], [0, 0, 0]]],
...                                     dtype='u1')
>>> g.dtype
dtype('uint8')
>>> g.ndim
3
>>> g.shape
(3, 2, 3)
>>> g.n_variants
3
>>> g.n_samples
2
>>> g.n_alleles
3
>>> g
<GenotypeAlleleCountsArray shape=(3, 2, 3) dtype=uint8>
2:0:0 0:2:0
1:1:0 0:1:1
0:0:4 0:0:0
```

Genotype calls for a single variant at all samples can be obtained by indexing the first dimension, e.g.:

```
>>> g[1]
<GenotypeAlleleCountsVector shape=(2, 3) dtype=uint8>
1:1:0 0:1:1
```

Genotype calls for a single sample at all variants can be obtained by indexing the second dimension, e.g.:

```
>>> g[:, 1]
<GenotypeAlleleCountsVector shape=(3, 3) dtype=uint8>
0:2:0 0:1:1 0:0:0
```

A genotype call for a single sample at a single variant can be obtained by indexing the first and second dimensions, e.g.:

```
>>> g[1, 0]
array([1, 1, 0], dtype=uint8)
```

Copy number (effective ploidy) may vary between calls:

```
>>> cn = g.sum(axis=2)
>>> cn
array([[2, 2],
       [2, 2],
       [4, 0]], dtype=uint64)
```

n_variants
Number of variants.

n_samples
Number of samples.

n_alleles
Number of alleles.

count_alleles(*subpop=None*)

subset(*sel0=None*, *sel1=None*)

GenotypeAlleleCountsVector

class allel.**GenotypeAlleleCountsVector**(*data*, *copy=False*, ***kwargs*)
Array of genotype calls for a sequence of variants or samples, stored as allele counts per call.

Parameters **data** : array_like, int, shape (n_variants, n_samples, ploidy)

Genotype data.

copy : bool, optional

If True, make a copy of *data*.

****kwargs** : keyword arguments

All keyword arguments are passed through to `numpy.array()`.

See also:

`GenotypeAlleleCounts`, `GenotypeAlleleCountsArray`, `GenotypeVector`

n_calls
Number of variants.

n_alleles
Number of alleles.

GenotypeAlleleCounts

Methods available on both `GenotypeAlleleCountsArray` and `GenotypeAlleleCountsVector` classes:

class allel.**GenotypeAlleleCounts**(*data*, *copy=False*, ***kwargs*)
Base class for wrapping a NumPy array of genotype calls, stored as allele counts per call.

See also:

`GenotypeAlleleCountsArray`, `GenotypeAlleleCountsVector`

is_called()

is_missing()

is_hom(*allele=None*)

is_hom_ref()

is_hom_alt()

is_het(*allele=None*)

compress(*condition*, *axis=0*, *out=None*)

take(*indices*, *axis=0*, *out=None*, *mode='raise'*)

concatenate(*others*, *axis=0*)

VariantTable

class allel.**VariantTable**(*data*, *index=None*, *copy=False*, ***kwargs*)

Table (catalogue) of variants.

Parameters **data** : array_like, structured, shape (n_variants,)

Variant records.

index : string or pair of strings, optional

Names of columns to use for positional index, e.g., 'POS' if table contains a 'POS' column and records from a single chromosome/contig, or ('CHROM', 'POS') if table contains records from multiple chromosomes/contigs.

****kwargs** : keyword arguments, optional

Further keyword arguments are passed through to `numpy.rec.array()`.

Examples

Instantiate a table from existing data:

```
>>> import allel
>>> records = [[b'chr1', 2, 35, 4.5, (1, 2)],
...            [b'chr1', 7, 12, 6.7, (3, 4)],
...            [b'chr2', 3, 78, 1.2, (5, 6)],
...            [b'chr2', 9, 22, 4.4, (7, 8)],
...            [b'chr3', 6, 99, 2.8, (9, 10)]]
>>> dtype = [('CHROM', 'S4'),
...          ('POS', 'u4'),
...          ('DP', int),
...          ('QD', float),
...          ('AC', (int, 2))]
>>> vt = allel.VariantTable(records, dtype=dtype,
...                         index=('CHROM', 'POS'))
>>> vt.names
('CHROM', 'POS', 'DP', 'QD', 'AC')
>>> vt.n_variants
5
```

Access a column:

```
>>> vt['DP']
array([35, 12, 78, 22, 99])
```

Access multiple columns:

```
>>> vt[['DP', 'QD']]
<VariantTable shape=(5,) dtype=(numpy.record, [('DP', '<i8'), ('QD', '<f8')])>
[(35, 4.5) (12, 6.7) (78, 1.2) (22, 4.4) (99, 2.8)]
```

Access a row:

```
>>> vt[2]
(b'chr2', 3, 78, 1.2, array([5, 6]))
```

Access multiple rows:

```
>>> vt[2:4]
<VariantTable shape=(2,) dtype=(numpy.record, [('CHROM', 'S4'), ('POS', '<u4'), ..
↪.
[(b'chr2', 3, 78, 1.2, [5, 6]) (b'chr2', 9, 22, 4.4, ...
```

Evaluate expressions against the table:

```
>>> vt.eval('DP > 30')
array([ True, False,  True, False,  True], dtype=bool)
>>> vt.eval('(DP > 30) & (QD > 4)')
array([ True, False, False, False, False], dtype=bool)
>>> vt.eval('DP * 2')
array([ 70,  24, 156,  44, 198])
```

Query the table:

```
>>> vt.query('DP > 30')
<VariantTable shape=(3,) dtype=(numpy.record, [('CHROM', 'S4'), ('POS', '<u4'), ..
↪.
[(b'chr1', 2, 35,  4.5, [ 1,  2]) (b'chr2', 3, 78,  1.2, ...
 (b'chr3', 6, 99,  2.8, [ 9, 10])]
>>> vt.query('(DP > 30) & (QD > 4)')
<VariantTable shape=(1,) dtype=(numpy.record, [('CHROM', 'S4'), ('POS', '<u4'), ..
↪.
[(b'chr1', 2, 35, 4.5, [1, 2])]
```

Use the index to query variants:

```
>>> vt.query_region(b'chr2', 1, 10)
<VariantTable shape=(2,) dtype=(numpy.record, [('CHROM', 'S4'), ('POS', '<u4'), ..
↪.
[(b'chr2', 3, 78,  1.2, [5, 6]) (b'chr2', 9, 22,  4.4, ...
```

n_variants
Number of variants (length of first dimension).

names

eval(*expression*, *vm='python'*)
Evaluate an expression against the table columns.

Parameters **expression** : string

Expression to evaluate.

vm : {'numexpr', 'python'}

Virtual machine to use.

Returns **result** : ndarray

query (*expression*, *vm='python'*)
Evaluate expression and then use it to extract rows from the table.

Parameters **expression** : string

Expression to evaluate.

vm : {'numexpr', 'python'}

Virtual machine to use.

Returns **result** : structured array

query_position (*chrom=None*, *position=None*)
Query the table, returning row or rows matching the given genomic position.

Parameters **chrom** : string, optional

Chromosome/contig.

position : int, optional

Position (1-based).

Returns **result** : row or VariantTable

query_region (*chrom=None*, *start=None*, *stop=None*)
Query the table, returning row or rows within the given genomic region.

Parameters **chrom** : string, optional

Chromosome/contig.

start : int, optional

Region start position (1-based).

stop : int, optional

Region stop position (1-based).

Returns **result** : VariantTable

to_vcf (*path*, *rename=None*, *number=None*, *description=None*, *fill=None*, *write_header=True*)
Write to a variant call format (VCF) file.

Parameters **path** : string

File path.

rename : dict, optional

Rename these columns in the VCF.

number : dict, optional

Override the number specified in INFO headers.

description : dict, optional

Descriptions for the INFO and FILTER headers.

fill : dict, optional

Fill values used for missing data in the table.

write_header : bool, optional

If True write VCF header.

Examples

Setup a variant table to write out:

```
>>> import allel
>>> chrom = [b'chr1', b'chr1', b'chr2', b'chr2', b'chr3']
>>> pos = [2, 6, 3, 8, 1]
>>> ids = ['a', 'b', 'c', 'd', 'e']
>>> ref = [b'A', b'C', b'T', b'G', b'N']
>>> alt = [(b'T', b'.'),
...        (b'G', b'.'),
...        (b'A', b'C'),
...        (b'C', b'A'),
...        (b'X', b'.')]
>>> qual = [1.2, 2.3, 3.4, 4.5, 5.6]
>>> filter_qd = [True, True, True, False, False]
>>> filter_dp = [True, False, True, False, False]
>>> dp = [12, 23, 34, 45, 56]
>>> qd = [12.3, 23.4, 34.5, 45.6, 56.7]
>>> flg = [True, False, True, False, True]
>>> ac = [(1, -1), (3, -1), (5, 6), (7, 8), (9, -1)]
>>> xx = [(1.2, 2.3), (3.4, 4.5), (5.6, 6.7), (7.8, 8.9),
...       (9.0, 9.9)]
>>> columns = [chrom, pos, ids, ref, alt, qual, filter_dp,
...            filter_qd, dp, qd, flg, ac, xx]
>>> records = list(zip(*columns))
>>> dtype = [('CHROM', 'S4'),
...          ('POS', 'u4'),
...          ('ID', 'S1'),
...          ('REF', 'S1'),
...          ('ALT', ('S1', 2)),
...          ('qual', 'f4'),
...          ('filter_dp', bool),
...          ('filter_qd', bool),
...          ('dp', int),
...          ('qd', float),
...          ('flg', bool),
...          ('ac', (int, 2)),
...          ('xx', (float, 2))]
>>> vt = allel.VariantTable(records, dtype=dtype)
```

Now write out to VCF and inspect the result:

```
>>> rename = {'dp': 'DP', 'qd': 'QD', 'filter_qd': 'QD'}
>>> fill = {'ALT': b'.', 'ac': -1}
>>> number = {'ac': 'A'}
>>> description = {'ac': 'Allele counts', 'filter_dp': 'Low depth'}
>>> vt.to_vcf('example.vcf', rename=rename, fill=fill,
...           number=number, description=description)
>>> print(open('example.vcf').read())
##fileformat=VCFv4.1
##fileDate=...
##source=...
##INFO=<ID=DP,Number=1,Type=Integer,Description="">
##INFO=<ID=QD,Number=1,Type=Float,Description="">
```

```
##INFO=<ID=ac,Number=A,Type=Integer,Description="Allele counts">
##INFO=<ID=flg,Number=0,Type=Flag,Description="">
##INFO=<ID=xx,Number=2,Type=Float,Description="">
##FILTER=<ID=QD,Description="">
##FILTER=<ID=dp,Description="Low depth">
#CHROM      POS     ID      REF     ALT     QUAL    FILTER  INFO
chr1        2       a       A       T       1.2     QD;dp   DP=12;QD=12.3;
↪ac=1;flg;xx=...
chr1        6       b       C       G       2.3     QD      DP=23;QD=23.4;
↪ac=3;xx=3.4,4.5
chr2        3       c       T       A,C     3.4     QD;dp   DP=34;QD=34.5;
↪ac=5,6;flg;x...
chr2        8       d       G       C,A     4.5     PASS    DP=45;QD=45.6;
↪ac=7,8;xx=7...
chr3        1       e       N       X       5.6     PASS    DP=56;QD=56.7;
↪ac=9;flg;xx=...
```

FeatureTable

class `allel.`**`FeatureTable`**(*data*, *copy=False*, ***kwargs*)

Table of genomic features (e.g., genes, exons, etc.).

Parameters **data** : array_like, structured, shape (n_variants,)

Variant records.

copy : bool, optional

If True, make a copy of *data*.

****kwargs** : keyword arguments, optional

Further keyword arguments are passed through to `numpy.rec.array()`.

`n_features`

Number of features (length of first dimension).

`names`

`eval`(*expression*, *vm='python'*)

Evaluate an expression against the table columns.

Parameters **expression** : string

Expression to evaluate.

vm : {'numexpr', 'python'}

Virtual machine to use.

Returns **result** : ndarray

`query`(*expression*, *vm='python'*)

Evaluate expression and then use it to extract rows from the table.

Parameters **expression** : string

Expression to evaluate.

vm : {'numexpr', 'python'}

Virtual machine to use.

Returns **result** : structured array

static **from_gff3**(*path*, *attributes=None*, *region=None*, *score_fill=-1*, *phase_fill=-1*, *attributes_fill='.'*, *dtype=None*)

Read a feature table from a GFF3 format file.

Parameters **path** : string

File path.

attributes : list of strings, optional

List of columns to extract from the "attributes" field.

region : string, optional

Genome region to extract. If given, file must be position sorted, bgzipped and tabix indexed. Tabix must also be installed and on the system path.

score_fill : int, optional

Value to use where score field has a missing value.

phase_fill : int, optional

Value to use where phase field has a missing value.

attributes_fill : object or list of objects, optional

Value(s) to use where attribute field(s) have a missing value.

dtype : numpy dtype, optional

Manually specify a dtype.

Returns **ft** : FeatureTable

to_mask(*size*, *start_name='start'*, *stop_name='end'*)

Construct a mask array where elements are True if the fall within features in the table.

Parameters **size** : int

Size of chromosome/contig.

start_name : string, optional

Name of column with start coordinates.

stop_name : string, optional

Name of column with stop coordinates.

Returns **mask** : ndarray, bool

SortedIndex

class allel.**SortedIndex**(*data*, *copy=False*, ***kwargs*)

Index of sorted values, e.g., positions from a single chromosome or contig.

Parameters **data** : array_like

Values in ascending order.

****kwargs** : keyword arguments

All keyword arguments are passed through to `numpy.array()`.

See also:

`SortedMultiIndex`, `UniqueIndex`

Notes

Values must be given in ascending order, although duplicate values may be present (i.e., values must be monotonically increasing).

Examples

```
>>> import allel
>>> idx = allel.SortedIndex([2, 5, 8, 14, 15, 23, 42, 42, 61, 77, 103], dtype='i4
↪')
>>> idx
<SortedIndex shape=(11,) dtype=int32>
[2, 5, 8, 14, 15, ..., 42, 42, 61, 77, 103]
>>> idx.dtype
dtype('int32')
>>> idx.ndim
1
>>> idx.shape
(11,)
>>> idx.is_unique
False
```

is_unique
True if no duplicate entries.

locate_key(*key*)
Get index location for the requested key.

Parameters **key** : object

Value to locate.

Returns **loc** : int or slice

Location of *key* (will be slice if there are duplicate entries).

Examples

```
>>> import allel
>>> idx = allel.SortedIndex([3, 6, 6, 11])
>>> idx.locate_key(3)
0
>>> idx.locate_key(11)
3
>>> idx.locate_key(6)
slice(1, 3, None)
>>> try:
...     idx.locate_key(2)
... except KeyError as e:
...     print(e)
...
2
```

locate_keys (*keys*, *strict=True*)
Get index locations for the requested keys.

Parameters **keys** : array_like

Array of keys to locate.

strict : bool, optional

If True, raise KeyError if any keys are not found in the index.

Returns **loc** : ndarray, bool

Boolean array with location of values.

Examples

```
>>> import allel
>>> idx1 = allel.SortedIndex([3, 6, 11, 20, 35])
>>> idx2 = allel.SortedIndex([4, 6, 20, 39])
>>> loc = idx1.locate_keys(idx2, strict=False)
>>> loc
array([False,  True, False,  True, False], dtype=bool)
>>> idx1[loc]
<SortedIndex shape=(2,) dtype=int64>
[6, 20]
```

locate_intersection (*other*)
Locate the intersection with another array.

Parameters **other** : array_like, int

Array of values to intersect.

Returns **loc** : ndarray, bool

Boolean array with location of intersection.

loc_other : ndarray, bool

Boolean array with location in *other* of intersection.

Examples

```
>>> import allel
>>> idx1 = allel.SortedIndex([3, 6, 11, 20, 35])
>>> idx2 = allel.SortedIndex([4, 6, 20, 39])
>>> loc1, loc2 = idx1.locate_intersection(idx2)
>>> loc1
array([False,  True, False,  True, False], dtype=bool)
>>> loc2
array([False,  True,  True, False], dtype=bool)
>>> idx1[loc1]
<SortedIndex shape=(2,) dtype=int64>
[6, 20]
>>> idx2[loc2]
<SortedIndex shape=(2,) dtype=int64>
[6, 20]
```

intersect (*other*)

Intersect with *other* sorted index.

Parameters **other** : array_like, int

Array of values to intersect with.

Returns **out** : SortedIndex

Values in common.

Examples

```
>>> import allel
>>> idx1 = allel.SortedIndex([3, 6, 11, 20, 35])
>>> idx2 = allel.SortedIndex([4, 6, 20, 39])
>>> idx1.intersect(idx2)
<SortedIndex shape=(2,) dtype=int64>
[6, 20]
```

locate_range (*start=None*, *stop=None*)

Locate slice of index containing all entries within *start* and *stop* values **inclusive**.

Parameters **start** : int, optional

Start value.

stop : int, optional

Stop value.

Returns **loc** : slice

Slice object.

Examples

```
>>> import allel
>>> idx = allel.SortedIndex([3, 6, 11, 20, 35])
>>> loc = idx.locate_range(4, 32)
>>> loc
slice(1, 4, None)
>>> idx[loc]
<SortedIndex shape=(3,) dtype=int64>
[6, 11, 20]
```

intersect_range (*start=None*, *stop=None*)

Intersect with range defined by *start* and *stop* values **inclusive**.

Parameters **start** : int, optional

Start value.

stop : int, optional

Stop value.

Returns **idx** : SortedIndex

Examples

```
>>> import allel
>>> idx = allel.SortedIndex([3, 6, 11, 20, 35])
>>> idx.intersect_range(4, 32)
<SortedIndex shape=(3,) dtype=int64>
[6, 11, 20]
```

locate_ranges(*starts*, *stops*, *strict=True*)

Locate items within the given ranges.

Parameters **starts** : array_like, int

Range start values.

stops : array_like, int

Range stop values.

strict : bool, optional

If True, raise KeyError if any ranges contain no entries.

Returns **loc** : ndarray, bool

Boolean array with location of entries found.

Examples

```
>>> import allel
>>> import numpy as np
>>> idx = allel.SortedIndex([3, 6, 11, 20, 35])
>>> ranges = np.array([[0, 2], [6, 17], [12, 15], [31, 35],
...                    [100, 120]])
>>> starts = ranges[:, 0]
>>> stops = ranges[:, 1]
>>> loc = idx.locate_ranges(starts, stops, strict=False)
>>> loc
array([False,  True,  True, False,  True], dtype=bool)
>>> idx[loc]
<SortedIndex shape=(3,) dtype=int64>
[6, 11, 35]
```

locate_intersection_ranges(*starts*, *stops*)

Locate the intersection with a set of ranges.

Parameters **starts** : array_like, int

Range start values.

stops : array_like, int

Range stop values.

Returns **loc** : ndarray, bool

Boolean array with location of entries found.

loc_ranges : ndarray, bool

Boolean array with location of ranges containing one or more entries.

Examples

```
>>> import allel
>>> import numpy as np
>>> idx = allel.SortedIndex([3, 6, 11, 20, 35])
>>> ranges = np.array([[0, 2], [6, 17], [12, 15], [31, 35],
...                    [100, 120]])
>>> starts = ranges[:, 0]
>>> stops = ranges[:, 1]
>>> loc, loc_ranges = idx.locate_intersection_ranges(starts, stops)
>>> loc
array([False,  True,  True, False,  True], dtype=bool)
>>> loc_ranges
array([False,  True, False,  True, False], dtype=bool)
>>> idx[loc]
<SortedIndex shape=(3,) dtype=int64>
[6, 11, 35]
>>> ranges[loc_ranges]
array([[ 6, 17],
       [31, 35]])
```

intersect_ranges(*starts*, *stops*)

Intersect with a set of ranges.

Parameters **starts** : array_like, int

Range start values.

stops : array_like, int

Range stop values.

Returns **idx** : SortedIndex

Examples

```
>>> import allel
>>> import numpy as np
>>> idx = allel.SortedIndex([3, 6, 11, 20, 35])
>>> ranges = np.array([[0, 2], [6, 17], [12, 15], [31, 35],
...                    [100, 120]])
>>> starts = ranges[:, 0]
>>> stops = ranges[:, 1]
>>> idx.intersect_ranges(starts, stops)
<SortedIndex shape=(3,) dtype=int64>
[6, 11, 35]
```

SortedMultiIndex

class allel.**SortedMultiIndex**(*l1*, *l2*, *copy=False*)

Two-level index of sorted values, e.g., variant positions from two or more chromosomes/contigs.

Parameters **l1** : array_like

First level values in ascending order.

l2 : array_like

Second level values, in ascending order within each sub-level.

copy : bool, optional

If True, inputs will be copied into new arrays.

See also:

`SortedIndex`, `UniqueIndex`

Examples

```
>>> import allel
>>> chrom = ['chr1', 'chr1', 'chr2', 'chr2', 'chr2', 'chr3']
>>> pos = [1, 4, 2, 5, 5, 3]
>>> idx = allel.SortedMultiIndex(chrom, pos)
>>> idx
<SortedMultiIndex shape=(6,), dtype=<U4/int64>
chr1:1 chr1:4 chr2:2 chr2:5 chr2:5 chr3:3
>>> len(idx)
6
```

locate_key(*k1*, *k2=None*)

Get index location for the requested key.

Parameters **k1** : object

Level 1 key.

k2 : object, optional

Level 2 key.

Returns **loc** : int or slice

Location of requested key (will be slice if there are duplicate entries).

Examples

```
>>> import allel
>>> chrom = ['chr1', 'chr1', 'chr2', 'chr2', 'chr2', 'chr3']
>>> pos = [1, 4, 2, 5, 5, 3]
>>> idx = allel.SortedMultiIndex(chrom, pos)
>>> idx.locate_key('chr1')
slice(0, 2, None)
>>> idx.locate_key('chr1', 4)
1
>>> idx.locate_key('chr2', 5)
slice(3, 5, None)
>>> try:
...     idx.locate_key('chr3', 4)
... except KeyError as e:
...     print(e)
...
('chr3', 4)
```

locate_range(*key*, *start=None*, *stop=None*)

Locate slice of index containing all entries within the range *key*:*start*-*stop* **inclusive**.

Parameters **key** : object

Level 1 key value.

start : object, optional

Level 2 start value.

stop : object, optional

Level 2 stop value.

Returns **loc** : slice

Slice object.

Examples

```
>>> import allel
>>> chrom = ['chr1', 'chr1', 'chr2', 'chr2', 'chr2', 'chr3']
>>> pos = [1, 4, 2, 5, 5, 3]
>>> idx = allel.SortedMultiIndex(chrom, pos)
>>> idx.locate_range('chr1')
slice(0, 2, None)
>>> idx.locate_range('chr1', 1, 4)
slice(0, 2, None)
>>> idx.locate_range('chr2', 3, 7)
slice(3, 5, None)
>>> try:
...     idx.locate_range('chr3', 4, 9)
... except KeyError as e:
...     print(e)
('chr3', 4, 9)
```

UniqueIndex

class `allel.`**`UniqueIndex`**(*data*, *copy=False*, *dtype=<type 'object'>*, ***kwargs*)

Array of unique values (e.g., variant or sample identifiers).

Parameters **data** : array_like

Values.

****kwargs** : keyword arguments

All keyword arguments are passed through to `numpy.array()`.

See also:

`SortedIndex`, `SortedMultiIndex`

Notes

This class represents an arbitrary set of unique values, e.g., sample or variant identifiers.

There is no need for values to be sorted. However, all values must be unique within the array, and must be hashable objects.

Examples

```
>>> import allel
>>> idx = allel.UniqueIndex(['A', 'C', 'B', 'F'])
>>> idx
<UniqueIndex shape=(4,) dtype=object>
['A', 'C', 'B', 'F']
>>> idx.dtype
dtype('O')
>>> idx.ndim
1
>>> idx.shape
(4,)
```

locate_key (*key*)

Get index location for the requested key.

Parameters **key** : object

Key to locate.

Returns **loc** : int

Location of *key*.

Examples

```
>>> import allel
>>> idx = allel.UniqueIndex(['A', 'C', 'B', 'F'])
>>> idx.locate_key('A')
0
>>> idx.locate_key('B')
2
>>> try:
...     idx.locate_key('X')
... except KeyError as e:
...     print(e)
...
'X'
```

locate_keys (*keys*, *strict=True*)

Get index locations for the requested keys.

Parameters **keys** : array_like

Array of keys to locate.

strict : bool, optional

If True, raise KeyError if any keys are not found in the index.

Returns **loc** : ndarray, bool

Boolean array with location of keys.

Examples

```
>>> import allel
>>> idx = allel.UniqueIndex(['A', 'C', 'B', 'F'])
>>> idx.locate_keys(['F', 'C'])
array([False,  True, False,  True], dtype=bool)
>>> idx.locate_keys(['X', 'F', 'G', 'C', 'Z'], strict=False)
array([False,  True, False,  True], dtype=bool)
```

locate_intersection(*other*)

Locate the intersection with another array.

Parameters **other** : array_like

Array to intersect.

Returns **loc** : ndarray, bool

Boolean array with location of intersection.

loc_other : ndarray, bool

Boolean array with location in *other* of intersection.

Examples

```
>>> import allel
>>> idx1 = allel.UniqueIndex(['A', 'C', 'B', 'F'], dtype=object)
>>> idx2 = allel.UniqueIndex(['X', 'F', 'G', 'C', 'Z'], dtype=object)
>>> loc1, loc2 = idx1.locate_intersection(idx2)
>>> loc1
array([False,  True, False,  True], dtype=bool)
>>> loc2
array([False,  True, False,  True, False], dtype=bool)
>>> idx1[loc1]
<UniqueIndex shape=(2,) dtype=object>
['C', 'F']
>>> idx2[loc2]
<UniqueIndex shape=(2,) dtype=object>
['F', 'C']
```

intersect(*other*)

Intersect with *other*.

Parameters **other** : array_like

Array to intersect.

Returns **out** : UniqueIndex

Examples

```
>>> import allel
>>> idx1 = allel.UniqueIndex(['A', 'C', 'B', 'F'], dtype=object)
>>> idx2 = allel.UniqueIndex(['X', 'F', 'G', 'C', 'Z'], dtype=object)
>>> idx1.intersect(idx2)
<UniqueIndex shape=(2,) dtype=object>
```

```
['C', 'F']
>>> idx2.intersect(idx1)
<UniqueIndex shape=(2,) dtype=object>
['F', 'C']
```

4.1.2 Chunked arrays

This module is maintained for backwards-compatibility, however it is recommended to migrate to use the equivalent classes from the *allel.model.dask* module where possible.

This module provides alternative implementations of array and table classes defined in the *allel.model.ndarray* module, using chunked arrays for data storage. Chunked arrays can be compressed and optionally stored on disk, providing a means for working with data too large to fit uncompressed in main memory.

Either Zarr, HDF5 (via h5py) or bcolz can be used as the underlying storage layer. Choice of storage layer can be made via the *storage* keyword argument which all class methods accept. This argument can either be a string identifying one of the predefined storage layer configurations, or an object implementing the chunked storage API. For more information about controlling storage see the *allel.chunked* module.

GenotypeChunkedArray

class allel.**GenotypeChunkedArray**(*data*)

Alternative implementation of the allel.model.ndarray.GenotypeArray class, wrapping a chunked array as the backing store.

Parameters **data** : array_like, int, shape (n_variants, n_samples, ploidy)

Genotype data to be wrapped. May be a bcolz carray, h5py dataset, or anything providing a similar interface.

Examples

Wrap an HDF5 dataset:

```
>>> import h5py
>>> with h5py.File('callset.h5', mode='w') as h5f:
...     h5g = h5f.create_group('/3L/calldata')
...     h5g.create_dataset('genotype',
...                        data=[[[0, 0], [0, 1]],
...                              [[0, 1], [1, 1]],
...                              [[0, 2], [-1, -1]]],
...                        dtype='i1', chunks=(2, 2, 2),
...                        compression='gzip', compression_opts=1)
...
<HDF5 dataset "genotype": shape (3, 2, 2), type "|i1">
>>> import allel
>>> callset = h5py.File('callset.h5', mode='r')
>>> g = allel.GenotypeChunkedArray(callset['/3L/calldata/genotype'])
>>> g
<GenotypeChunkedArray shape=(3, 2, 2) dtype=int8 chunks=(2, 2, 2)
   nbytes=12 cbytes=30 cratio=0.4
   compression=gzip compression_opts=1
   values=h5py._hl.dataset.Dataset>
```

```
>>> g.values
<HDF5 dataset "genotype": shape (3, 2, 2), type "|i1">
```

Obtain a numpy array by slicing, e.g.:

```
>>> g[:]
<GenotypeArray shape=(3, 2, 2) dtype=int8>
0/0 0/1
0/1 1/1
0/2 ./.
```

Note that most methods will return a chunked array, using whatever chunked storage is set as default (bcolz carray) or specified directly via the *storage* keyword argument. E.g.:

```
>>> g.copy()
<GenotypeChunkedArray shape=(3, 2, 2) dtype=int8 chunks=(3, 2, 2)
   ...
>>> g.copy(storage='bcolzmem')
<GenotypeChunkedArray shape=(3, 2, 2) dtype=int8 chunks=(4096, 2, 2)
   ...
>>> g.copy(storage='hdf5mem_zlib1')
<GenotypeChunkedArray shape=(3, 2, 2) dtype=int8 chunks=(3, 2, 2)
   ...
```

HaplotypeChunkedArray

class allel.**HaplotypeChunkedArray**(*data*)

Alternative implementation of the allel.model.ndarray.HaplotypeArray class, using a chunked array as the backing store.

Parameters **data** : array_like, int, shape (n_variants, n_haplotypes)

Haplotype data to be wrapped. May be a bcolz carray, h5py dataset, or anything providing a similar interface.

AlleleCountsChunkedArray

class allel.**AlleleCountsChunkedArray**(*data*)

Alternative implementation of the allel.model.ndarray.AlleleCountsArray class, using a chunked array as the backing store.

Parameters **data** : array_like, int, shape (n_variants, n_alleles)

Allele counts data to be wrapped. May be a bcolz carray, h5py dataset, or anything providing a similar interface.

VariantChunkedTable

class allel.**VariantChunkedTable**(*data*, *names=None*, *index=None*)

Alternative implementation of the allel.model.ndarray.VariantTable class, using a chunked table as the backing store.

Parameters **data: table_like**

Data to be wrapped. May be a tuple or list of columns (array-like), a dict mapping names to columns, a bcolz ctable, h5py group, numpy recarray, or anything providing a similar interface.

names : sequence of strings

Column names.

Examples

Wrap columns stored as datasets within an HDF5 group:

```
>>> import h5py
>>> chrom = [b'chr1', b'chr1', b'chr2', b'chr2', b'chr3']
>>> pos = [2, 7, 3, 9, 6]
>>> dp = [35, 12, 78, 22, 99]
>>> qd = [4.5, 6.7, 1.2, 4.4, 2.8]
>>> ac = [(1, 2), (3, 4), (5, 6), (7, 8), (9, 10)]
>>> with h5py.File('callset.h5', mode='w') as h5f:
...     h5g = h5f.create_group('/3L/variants')
...     h5g.create_dataset('CHROM', data=chrom, chunks=True)
...     h5g.create_dataset('POS', data=pos, chunks=True)
...     h5g.create_dataset('DP', data=dp, chunks=True)
...     h5g.create_dataset('QD', data=qd, chunks=True)
...     h5g.create_dataset('AC', data=ac, chunks=True)
...
<HDF5 dataset "CHROM": shape (5,), type "|S4">
<HDF5 dataset "POS": shape (5,), type "<i8">
<HDF5 dataset "DP": shape (5,), type "<i8">
<HDF5 dataset "QD": shape (5,), type "<f8">
<HDF5 dataset "AC": shape (5, 2), type "<i8">
>>> import allel
>>> callset = h5py.File('callset.h5', mode='r')
>>> vt = allel.VariantChunkedTable(callset['/3L/variants'],
...                                names=['CHROM', 'POS', 'AC', 'QD', 'DP'])
>>> vt
<VariantChunkedTable shape=(5,) dtype=[('CHROM', 'S4'), ('POS', '<i8'), ('AC', ...
   ...
```

Obtain a single row:

```
>>> vt[0]
row(CHROM=b'chr1', POS=2, AC=array([1, 2]), QD=4.5, DP=35)
```

Obtain a numpy array by slicing:

```
>>> vt[:]
<VariantTable shape=(5,) dtype=(numpy.record, [('CHROM', 'S4'), ('POS', '<i8'), ..
↪.
[(b'chr1', 2, [ 1,  2], 4.5, 35) (b'chr1', 7, [ 3,  4], 6.7, 12)
 (b'chr2', 3, [ 5,  6], 1.2, 78) (b'chr2', 9, [ 7,  8], 4.4, 22)
 (b'chr3', 6, [ 9, 10], 2.8, 99)]
```

Access a subset of columns:

```
>>> vt[['CHROM', 'POS']]
<VariantChunkedTable shape=(5,) dtype=[('CHROM', 'S4'), ('POS', '<i8')]
```

```
   nbytes=60 cbytes=60 cratio=1.0
   values=builtins.list>
```

Note that most methods will return a chunked table, using whatever chunked storage is set as default (bcolz ctable) or specified directly via the *storage* keyword argument. E.g.:

```
>>> vt.copy()
<VariantChunkedTable shape=(5,) dtype=[('CHROM', 'S4'), ('POS', '<i8'), ('AC', ...
   nbytes=220 cbytes=1.7K cratio=0.1
   values=allel.chunked.storage_zarr.ZarrTable>
>>> vt.copy(storage='bcolzmem')
<VariantChunkedTable shape=(5,) dtype=[('CHROM', 'S4'), ('POS', '<i8'), ('AC', ...
   nbytes=220 cbytes=80.0K cratio=0.0
   values=bcolz.ctable.ctable>
>>> vt.copy(storage='hdf5mem_zlib1')
<VariantChunkedTable shape=(5,) dtype=[('CHROM', 'S4'), ('POS', '<i8'), ('AC', ...
   nbytes=220 cbytes=131 cratio=1.7
   values=h5py._hl.files.File>
```

AlleleCountsChunkedTable

class allel.**AlleleCountsChunkedTable**(*data*, *names=None*)

4.1.3 Dask arrays

This module provides alternative implementations of array classes defined in the *allel.model.ndarray* module, using dask.array as the computational engine.

Dask uses blocked algorithms and task scheduling to break up work into smaller pieces, allowing computation over large datasets. It also uses lazy evaluation, meaning that multiple operations can be chained together into a task graph, reducing total memory requirements for intermediate results, and only the tasks required to generate the requested part of the final data set will be executed.

This module is experimental, if you find a bug please raise an issue on GitHub.

This module requires dask >= 0.11.1.

GenotypeDaskArray

class allel.**GenotypeDaskArray**(*data*, *chunks=None*, *name=None*, *lock=False*)

HaplotypeDaskArray

class allel.**HaplotypeDaskArray**(*data*, *chunks=None*, *name=None*, *lock=False*)

AlleleCountsDaskArray

class allel.**AlleleCountsDaskArray**(*data*, *chunks=None*, *name=None*, *lock=False*)

4.1.4 Utility functions

allel.**create_allele_mapping**(*ref*, *alt*, *alleles*, *dtype='i1'*)

Create an array mapping variant alleles into a different allele index system.

Parameters **ref** : array_like, S1, shape (n_variants,)

Reference alleles.

alt : array_like, S1, shape (n_variants, n_alt_alleles)

Alternate alleles.

alleles : array_like, S1, shape (n_variants, n_alleles)

Alleles defining the new allele indexing.

dtype : dtype, optional

Output dtype.

Returns **mapping** : ndarray, int8, shape (n_variants, n_alt_alleles + 1)

See also:

`GenotypeArray.map_alleles`, `HaplotypeArray.map_alleles`, `AlleleCountsArray.map_alleles`

Examples

Example with biallelic variants:

```
>>> import allel
>>> ref = [b'A', b'C', b'T', b'G']
>>> alt = [b'T', b'G', b'C', b'A']
>>> alleles = [[b'A', b'T'],  # no transformation
...            [b'G', b'C'],  # swap
...            [b'T', b'A'],  # 1 missing
...            [b'A', b'C']]  # 1 missing
>>> mapping = allel.create_allele_mapping(ref, alt, alleles)
>>> mapping
array([[ 0,  1],
       [ 1,  0],
       [ 0, -1],
       [-1,  0]], dtype=int8)
```

Example with multiallelic variants:

```
>>> ref = [b'A', b'C', b'T']
>>> alt = [[b'T', b'G'],
...        [b'A', b'T'],
...        [b'G', b'.']]
>>> alleles = [[b'A', b'T'],
...            [b'C', b'T'],
...            [b'G', b'A']]
>>> mapping = create_allele_mapping(ref, alt, alleles)
>>> mapping
array([[ 0,  1, -1],
       [ 0, -1,  1],
       [-1,  0, -1]], dtype=int8)
```

allel.**locate_fixed_differences**(*ac1*, *ac2*)

Locate variants with no shared alleles between two populations.

Parameters **ac1** : array_like, int, shape (n_variants, n_alleles)

Allele counts array from the first population.

ac2 : array_like, int, shape (n_variants, n_alleles)

Allele counts array from the second population.

Returns **loc** : ndarray, bool, shape (n_variants,)

See also:

allel.stats.diversity.windowed_df

Examples

```
>>> import allel
>>> g = allel.GenotypeArray([[[0, 0], [0, 0], [1, 1], [1, 1]],
...                          [[0, 1], [0, 1], [0, 1], [0, 1]],
...                          [[0, 1], [0, 1], [1, 1], [1, 1]],
...                          [[0, 0], [0, 0], [1, 1], [2, 2]],
...                          [[0, 0], [-1, -1], [1, 1], [-1, -1]]])
>>> ac1 = g.count_alleles(subpop=[0, 1])
>>> ac2 = g.count_alleles(subpop=[2, 3])
>>> loc_df = allel.locate_fixed_differences(ac1, ac2)
>>> loc_df
array([ True, False, False,  True,  True], dtype=bool)
```

allel.**locate_private_alleles**(**acs*)

Locate alleles that are found only in a single population.

Parameters ***acs** : array_like, int, shape (n_variants, n_alleles)

Allele counts arrays from each population.

Returns **loc** : ndarray, bool, shape (n_variants, n_alleles)

Boolean array where elements are True if allele is private to a single population.

Examples

```
>>> import allel
>>> g = allel.GenotypeArray([[[0, 0], [0, 0], [1, 1], [1, 1]],
...                          [[0, 1], [0, 1], [0, 1], [0, 1]],
...                          [[0, 1], [0, 1], [1, 1], [1, 1]],
...                          [[0, 0], [0, 0], [1, 1], [2, 2]],
...                          [[0, 0], [-1, -1], [1, 1], [-1, -1]]])
>>> ac1 = g.count_alleles(subpop=[0, 1])
>>> ac2 = g.count_alleles(subpop=[2])
>>> ac3 = g.count_alleles(subpop=[3])
>>> loc_private_alleles = allel.locate_private_alleles(ac1, ac2, ac3)
>>> loc_private_alleles
array([[ True, False, False],
       [False, False, False],
       [ True, False, False],
       [ True,  True,  True],
```

```
       [ True,  True, False]], dtype=bool)
>>> loc_private_variants = np.any(loc_private_alleles, axis=1)
>>> loc_private_variants
array([ True, False,  True,  True,  True], dtype=bool)
```

allel.**sample_to_haplotype_selection**(*indices*, *ploidy*)

4.2 Statistics and plotting

4.2.1 Diversity & divergence

allel.**mean_pairwise_difference**(*ac*, *an=None*, *fill=nan*)

Calculate for each variant the mean number of pairwise differences between chromosomes sampled from within a single population.

Parameters **ac** : array_like, int, shape (n_variants, n_alleles)

Allele counts array.

an : array_like, int, shape (n_variants,), optional

Allele numbers. If not provided, will be calculated from *ac*.

fill : float

Use this value where there are no pairs to compare (e.g., all allele calls are missing).

Returns **mpd** : ndarray, float, shape (n_variants,)

See also:

sequence_diversity, windowed_diversity

Notes

The values returned by this function can be summed over a genome region and divided by the number of accessible bases to estimate nucleotide diversity, a.k.a. *pi*.

Examples

```
>>> import allel
>>> h = allel.HaplotypeArray([[0, 0, 0, 0],
...                           [0, 0, 0, 1],
...                           [0, 0, 1, 1],
...                           [0, 1, 1, 1],
...                           [1, 1, 1, 1],
...                           [0, 0, 1, 2],
...                           [0, 1, 1, 2],
...                           [0, 1, -1, -1]])
>>> ac = h.count_alleles()
>>> allel.stats.mean_pairwise_difference(ac)
array([ 0.        ,  0.5       ,  0.66666667,  0.5       ,  0.        ,
        0.83333333,  0.83333333,  1.        ])
```

allel.**sequence_diversity**(*pos*, *ac*, *start=None*, *stop=None*, *is_accessible=None*)

Estimate nucleotide diversity within a given region.

Parameters **pos** : array_like, int, shape (n_items,)

Variant positions, using 1-based coordinates, in ascending order.

ac : array_like, int, shape (n_variants, n_alleles)

Allele counts array.

start : int, optional

The position at which to start (1-based).

stop : int, optional

The position at which to stop (1-based).

is_accessible : array_like, bool, shape (len(contig),), optional

Boolean array indicating accessibility status for all positions in the chromosome/contig.

Returns **pi** : ndarray, float, shape (n_windows,)

Nucleotide diversity.

Examples

```
>>> import allel
>>> g = allel.GenotypeArray([[[0, 0], [0, 0]],
...                          [[0, 0], [0, 1]],
...                          [[0, 0], [1, 1]],
...                          [[0, 1], [1, 1]],
...                          [[1, 1], [1, 1]],
...                          [[0, 0], [1, 2]],
...                          [[0, 1], [1, 2]],
...                          [[0, 1], [-1, -1]],
...                          [[-1, -1], [-1, -1]]])
>>> ac = g.count_alleles()
>>> pos = [2, 4, 7, 14, 15, 18, 19, 25, 27]
>>> pi = allel.stats.sequence_diversity(pos, ac, start=1, stop=31)
>>> pi
0.13978494623655915
```

allel.**windowed_diversity**(*pos*, *ac*, *size=None*, *start=None*, *stop=None*, *step=None*, *windows=None*, *is_accessible=None*, *fill=nan*)

Estimate nucleotide diversity in windows over a single chromosome/contig.

Parameters **pos** : array_like, int, shape (n_items,)

Variant positions, using 1-based coordinates, in ascending order.

ac : array_like, int, shape (n_variants, n_alleles)

Allele counts array.

size : int, optional

The window size (number of bases).

start : int, optional

The position at which to start (1-based).

stop : int, optional

The position at which to stop (1-based).

step : int, optional

The distance between start positions of windows. If not given, defaults to the window size, i.e., non-overlapping windows.

windows : array_like, int, shape (n_windows, 2), optional

Manually specify the windows to use as a sequence of (window_start, window_stop) positions, using 1-based coordinates. Overrides the size/start/stop/step parameters.

is_accessible : array_like, bool, shape (len(contig),), optional

Boolean array indicating accessibility status for all positions in the chromosome/contig.

fill : object, optional

The value to use where a window is completely inaccessible.

Returns **pi** : ndarray, float, shape (n_windows,)

Nucleotide diversity in each window.

windows : ndarray, int, shape (n_windows, 2)

The windows used, as an array of (window_start, window_stop) positions, using 1-based coordinates.

n_bases : ndarray, int, shape (n_windows,)

Number of (accessible) bases in each window.

counts : ndarray, int, shape (n_windows,)

Number of variants in each window.

Examples

```
>>> import allel
>>> g = allel.GenotypeArray([[[0, 0], [0, 0]],
...                          [[0, 0], [0, 1]],
...                          [[0, 0], [1, 1]],
...                          [[0, 1], [1, 1]],
...                          [[1, 1], [1, 1]],
...                          [[0, 0], [1, 2]],
...                          [[0, 1], [1, 2]],
...                          [[0, 1], [-1, -1]],
...                          [[-1, -1], [-1, -1]]])
>>> ac = g.count_alleles()
>>> pos = [2, 4, 7, 14, 15, 18, 19, 25, 27]
>>> pi, windows, n_bases, counts = allel.stats.windowed_diversity(
...     pos, ac, size=10, start=1, stop=31
... )
>>> pi
array([ 0.11666667,  0.21666667,  0.09090909])
>>> windows
array([[ 1, 10],
       [11, 20],
       [21, 31]])
>>> n_bases
```

```
array([10, 10, 11])
>>> counts
array([3, 4, 2])
```

allel.**mean_pairwise_difference_between**(*ac1*, *ac2*, *an1=None*, *an2=None*, *fill=nan*)

Calculate for each variant the mean number of pairwise differences between chromosomes sampled from two different populations.

Parameters **ac1** : array_like, int, shape (n_variants, n_alleles)

Allele counts array from the first population.

ac2 : array_like, int, shape (n_variants, n_alleles)

Allele counts array from the second population.

an1 : array_like, int, shape (n_variants,), optional

Allele numbers for the first population. If not provided, will be calculated from *ac1*.

an2 : array_like, int, shape (n_variants,), optional

Allele numbers for the second population. If not provided, will be calculated from *ac2*.

fill : float

Use this value where there are no pairs to compare (e.g., all allele calls are missing).

Returns **mpd** : ndarray, float, shape (n_variants,)

See also:

sequence_divergence, windowed_divergence

Notes

The values returned by this function can be summed over a genome region and divided by the number of accessible bases to estimate nucleotide divergence between two populations, a.k.a. *Dxy*.

Examples

```
>>> import allel
>>> h = allel.HaplotypeArray([[0, 0, 0, 0],
...                           [0, 0, 0, 1],
...                           [0, 0, 1, 1],
...                           [0, 1, 1, 1],
...                           [1, 1, 1, 1],
...                           [0, 0, 1, 2],
...                           [0, 1, 1, 2],
...                           [0, 1, -1, -1]])
>>> ac1 = h.count_alleles(subpop=[0, 1])
>>> ac2 = h.count_alleles(subpop=[2, 3])
>>> allel.stats.mean_pairwise_difference_between(ac1, ac2)
array([ 0.  ,  0.5 ,  1.  ,  0.5 ,  0.  ,  1.  ,  0.75,  nan])
```

allel.**sequence_divergence**(*pos*, *ac1*, *ac2*, *an1=None*, *an2=None*, *start=None*, *stop=None*, *is_accessible=None*)

Estimate nucleotide divergence between two populations within a given region.

Parameters **pos** : array_like, int, shape (n_items,)

Variant positions, using 1-based coordinates, in ascending order.

ac1 : array_like, int, shape (n_variants, n_alleles)

Allele counts array for the first population.

ac2 : array_like, int, shape (n_variants, n_alleles)

Allele counts array for the second population.

an1 : array_like, int, shape (n_variants,), optional

Allele numbers for the first population. If not provided, will be calculated from *ac1*.

an2 : array_like, int, shape (n_variants,), optional

Allele numbers for the second population. If not provided, will be calculated from *ac2*.

start : int, optional

The position at which to start (1-based).

stop : int, optional

The position at which to stop (1-based).

is_accessible : array_like, bool, shape (len(contig),), optional

Boolean array indicating accessibility status for all positions in the chromosome/contig.

Returns **Dxy** : ndarray, float, shape (n_windows,)

Nucleotide divergence.

Examples

Simplest case, two haplotypes in each population:

```
>>> import allel
>>> h = allel.HaplotypeArray([[0, 0, 0, 0],
...                           [0, 0, 0, 1],
...                           [0, 0, 1, 1],
...                           [0, 1, 1, 1],
...                           [1, 1, 1, 1],
...                           [0, 0, 1, 2],
...                           [0, 1, 1, 2],
...                           [0, 1, -1, -1],
...                           [-1, -1, -1, -1]])
>>> ac1 = h.count_alleles(subpop=[0, 1])
>>> ac2 = h.count_alleles(subpop=[2, 3])
>>> pos = [2, 4, 7, 14, 15, 18, 19, 25, 27]
>>> dxy = sequence_divergence(pos, ac1, ac2, start=1, stop=31)
>>> dxy
0.12096774193548387
```

allel.**windowed_divergence**(*pos*, *ac1*, *ac2*, *size=None*, *start=None*, *stop=None*, *step=None*, *windows=None*, *is_accessible=None*, *fill=nan*)

Estimate nucleotide divergence between two populations in windows over a single chromosome/contig.

Parameters **pos** : array_like, int, shape (n_items,)

Variant positions, using 1-based coordinates, in ascending order.

ac1 : array_like, int, shape (n_variants, n_alleles)

Allele counts array for the first population.

ac2 : array_like, int, shape (n_variants, n_alleles)

Allele counts array for the second population.

size : int, optional

The window size (number of bases).

start : int, optional

The position at which to start (1-based).

stop : int, optional

The position at which to stop (1-based).

step : int, optional

The distance between start positions of windows. If not given, defaults to the window size, i.e., non-overlapping windows.

windows : array_like, int, shape (n_windows, 2), optional

Manually specify the windows to use as a sequence of (window_start, window_stop) positions, using 1-based coordinates. Overrides the size/start/stop/step parameters.

is_accessible : array_like, bool, shape (len(contig),), optional

Boolean array indicating accessibility status for all positions in the chromosome/contig.

fill : object, optional

The value to use where a window is completely inaccessible.

Returns **Dxy** : ndarray, float, shape (n_windows,)

Nucleotide divergence in each window.

windows : ndarray, int, shape (n_windows, 2)

The windows used, as an array of (window_start, window_stop) positions, using 1-based coordinates.

n_bases : ndarray, int, shape (n_windows,)

Number of (accessible) bases in each window.

counts : ndarray, int, shape (n_windows,)

Number of variants in each window.

Examples

Simplest case, two haplotypes in each population:

```
>>> import allel
>>> h = allel.HaplotypeArray([[0, 0, 0, 0],
...                           [0, 0, 0, 1],
...                           [0, 0, 1, 1],
...                           [0, 1, 1, 1],
...                           [1, 1, 1, 1],
...                           [0, 0, 1, 2],
...                           [0, 1, 1, 2],
...                           [0, 1, -1, -1],
```

```
...                             [-1, -1, -1, -1]])
>>> ac1 = h.count_alleles(subpop=[0, 1])
>>> ac2 = h.count_alleles(subpop=[2, 3])
>>> pos = [2, 4, 7, 14, 15, 18, 19, 25, 27]
>>> dxy, windows, n_bases, counts = windowed_divergence(
...     pos, ac1, ac2, size=10, start=1, stop=31
... )
>>> dxy
array([ 0.15 ,  0.225,  0.   ])
>>> windows
array([[ 1, 10],
       [11, 20],
       [21, 31]])
>>> n_bases
array([10, 10, 11])
>>> counts
array([3, 4, 2])
```

allel.**watterson_theta**(*pos*, *ac*, *start=None*, *stop=None*, *is_accessible=None*)

Calculate the value of Watterson's estimator over a given region.

Parameters **pos** : array_like, int, shape (n_items,)

Variant positions, using 1-based coordinates, in ascending order.

ac : array_like, int, shape (n_variants, n_alleles)

Allele counts array.

start : int, optional

The position at which to start (1-based).

stop : int, optional

The position at which to stop (1-based).

is_accessible : array_like, bool, shape (len(contig),), optional

Boolean array indicating accessibility status for all positions in the chromosome/contig.

Returns **theta_hat_w** : float

Watterson's estimator (theta hat per base).

Examples

```
>>> import allel
>>> g = allel.GenotypeArray([[[0, 0], [0, 0]],
...                          [[0, 0], [0, 1]],
...                          [[0, 0], [1, 1]],
...                          [[0, 1], [1, 1]],
...                          [[1, 1], [1, 1]],
...                          [[0, 0], [1, 2]],
...                          [[0, 1], [1, 2]],
...                          [[0, 1], [-1, -1]],
...                          [[-1, -1], [-1, -1]]])
>>> ac = g.count_alleles()
>>> pos = [2, 4, 7, 14, 15, 18, 19, 25, 27]
>>> theta_hat_w = allel.stats.watterson_theta(pos, ac, start=1, stop=31)
```

```
>>> theta_hat_w
0.10557184750733138
```

allel.**windowed_watterson_theta**(*pos*, *ac*, *size=None*, *start=None*, *stop=None*, *step=None*, *windows=None*, *is_accessible=None*, *fill=nan*)

Calculate the value of Watterson's estimator in windows over a single chromosome/contig.

Parameters **pos** : array_like, int, shape (n_items,)

Variant positions, using 1-based coordinates, in ascending order.

ac : array_like, int, shape (n_variants, n_alleles)

Allele counts array.

size : int, optional

The window size (number of bases).

start : int, optional

The position at which to start (1-based).

stop : int, optional

The position at which to stop (1-based).

step : int, optional

The distance between start positions of windows. If not given, defaults to the window size, i.e., non-overlapping windows.

windows : array_like, int, shape (n_windows, 2), optional

Manually specify the windows to use as a sequence of (window_start, window_stop) positions, using 1-based coordinates. Overrides the size/start/stop/step parameters.

is_accessible : array_like, bool, shape (len(contig),), optional

Boolean array indicating accessibility status for all positions in the chromosome/contig.

fill : object, optional

The value to use where a window is completely inaccessible.

Returns **theta_hat_w** : ndarray, float, shape (n_windows,)

Watterson's estimator (theta hat per base).

windows : ndarray, int, shape (n_windows, 2)

The windows used, as an array of (window_start, window_stop) positions, using 1-based coordinates.

n_bases : ndarray, int, shape (n_windows,)

Number of (accessible) bases in each window.

counts : ndarray, int, shape (n_windows,)

Number of variants in each window.

Examples

```
>>> import allel
>>> g = allel.GenotypeArray([[[0, 0], [0, 0]],
...                          [[0, 0], [0, 1]],
...                          [[0, 0], [1, 1]],
...                          [[0, 1], [1, 1]],
...                          [[1, 1], [1, 1]],
...                          [[0, 0], [1, 2]],
...                          [[0, 1], [1, 2]],
...                          [[0, 1], [-1, -1]],
...                          [[-1, -1], [-1, -1]]])
>>> ac = g.count_alleles()
>>> pos = [2, 4, 7, 14, 15, 18, 19, 25, 27]
>>> theta_hat_w, windows, n_bases, counts = allel.stats.windowed_watterson_theta(
...     pos, ac, size=10, start=1, stop=31
... )
>>> theta_hat_w
array([ 0.10909091,  0.16363636,  0.04958678])
>>> windows
array([[ 1, 10],
       [11, 20],
       [21, 31]])
>>> n_bases
array([10, 10, 11])
>>> counts
array([3, 4, 2])
```

allel.**tajima_d**(*ac*, *pos=None*, *start=None*, *stop=None*)

Calculate the value of Tajima's D over a given region.

Parameters **ac** : array_like, int, shape (n_variants, n_alleles)

Allele counts array.

pos : array_like, int, shape (n_items,), optional

Variant positions, using 1-based coordinates, in ascending order.

start : int, optional

The position at which to start (1-based).

stop : int, optional

The position at which to stop (1-based).

Returns **D** : float

Examples

```
>>> import allel
>>> g = allel.GenotypeArray([[[0, 0], [0, 0]],
...                          [[0, 0], [0, 1]],
...                          [[0, 0], [1, 1]],
...                          [[0, 1], [1, 1]],
...                          [[1, 1], [1, 1]],
...                          [[0, 0], [1, 2]],
...                          [[0, 1], [1, 2]],
```

```
...                             [[0, 1], [-1, -1]],
...                             [[-1, -1], [-1, -1]]])
>>> ac = g.count_alleles()
>>> allel.stats.tajima_d(ac)
3.1445848780213814
>>> pos = [2, 4, 7, 14, 15, 18, 19, 25, 27]
>>> allel.stats.tajima_d(ac, pos=pos, start=7, stop=25)
3.8779735196179366
```

allel.**windowed_tajima_d**(*pos*, *ac*, *size=None*, *start=None*, *stop=None*, *step=None*, *windows=None*, *fill=nan*)

Calculate the value of Tajima's D in windows over a single chromosome/contig.

Parameters **pos** : array_like, int, shape (n_items,)

Variant positions, using 1-based coordinates, in ascending order.

ac : array_like, int, shape (n_variants, n_alleles)

Allele counts array.

size : int, optional

The window size (number of bases).

start : int, optional

The position at which to start (1-based).

stop : int, optional

The position at which to stop (1-based).

step : int, optional

The distance between start positions of windows. If not given, defaults to the window size, i.e., non-overlapping windows.

windows : array_like, int, shape (n_windows, 2), optional

Manually specify the windows to use as a sequence of (window_start, window_stop) positions, using 1-based coordinates. Overrides the size/start/stop/step parameters.

fill : object, optional

The value to use where a window is completely inaccessible.

Returns **D** : ndarray, float, shape (n_windows,)

Tajima's D.

windows : ndarray, int, shape (n_windows, 2)

The windows used, as an array of (window_start, window_stop) positions, using 1-based coordinates.

counts : ndarray, int, shape (n_windows,)

Number of variants in each window.

Examples

```
>>> import allel
>>> g = allel.GenotypeArray([[[0, 0], [0, 0]],
...                          [[0, 0], [0, 1]],
...                          [[0, 0], [1, 1]],
...                          [[0, 1], [1, 1]],
...                          [[1, 1], [1, 1]],
...                          [[0, 0], [1, 2]],
...                          [[0, 1], [1, 2]],
...                          [[0, 1], [-1, -1]],
...                          [[-1, -1], [-1, -1]]])
>>> ac = g.count_alleles()
>>> pos = [2, 4, 7, 14, 15, 18, 19, 25, 27]
>>> D, windows, counts = allel.stats.windowed_tajima_d(
...     pos, ac, size=10, start=1, stop=31
... )
>>> D
array([ 0.59158014,  2.93397641,  6.12372436])
>>> windows
array([[ 1, 10],
       [11, 20],
       [21, 31]])
>>> counts
array([3, 4, 2])
```

allel.**moving_tajima_d**(*ac*, *size*, *start=0*, *stop=None*, *step=None*)

Calculate the value of Tajima's D in moving windows of *size* variants.

Parameters **ac** : array_like, int, shape (n_variants, n_alleles)

Allele counts array.

size : int

The window size (number of variants).

start : int, optional

The index at which to start.

stop : int, optional

The index at which to stop.

step : int, optional

The number of variants between start positions of windows. If not given, defaults to the window size, i.e., non-overlapping windows.

Returns **d** : ndarray, float, shape (n_windows,)

Tajima's D.

Examples

```
>>> import allel
>>> g = allel.GenotypeArray([[[0, 0], [0, 0]],
...                          [[0, 0], [0, 1]],
...                          [[0, 0], [1, 1]],
...                          [[0, 1], [1, 1]],
...                          [[1, 1], [1, 1]],
```

```
...                             [[0, 0], [1, 2]],
...                             [[0, 1], [1, 2]],
...                             [[0, 1], [-1, -1]],
...                             [[-1, -1], [-1, -1]]])
>>> ac = g.count_alleles()
>>> D = allel.stats.moving_tajima_d(ac, size=3)
>>> D
array([ 0.59158014,  1.89305645,  5.79748537])
```

allel.**windowed_df**(*pos*, *ac1*, *ac2*, *size=None*, *start=None*, *stop=None*, *step=None*, *windows=None*, *is_accessible=None*, *fill=nan*)

Calculate the density of fixed differences between two populations in windows over a single chromosome/contig.

Parameters **pos** : array_like, int, shape (n_items,)

Variant positions, using 1-based coordinates, in ascending order.

ac1 : array_like, int, shape (n_variants, n_alleles)

Allele counts array for the first population.

ac2 : array_like, int, shape (n_variants, n_alleles)

Allele counts array for the second population.

size : int, optional

The window size (number of bases).

start : int, optional

The position at which to start (1-based).

stop : int, optional

The position at which to stop (1-based).

step : int, optional

The distance between start positions of windows. If not given, defaults to the window size, i.e., non-overlapping windows.

windows : array_like, int, shape (n_windows, 2), optional

Manually specify the windows to use as a sequence of (window_start, window_stop) positions, using 1-based coordinates. Overrides the size/start/stop/step parameters.

is_accessible : array_like, bool, shape (len(contig),), optional

Boolean array indicating accessibility status for all positions in the chromosome/contig.

fill : object, optional

The value to use where a window is completely inaccessible.

Returns **df** : ndarray, float, shape (n_windows,)

Per-base density of fixed differences in each window.

windows : ndarray, int, shape (n_windows, 2)

The windows used, as an array of (window_start, window_stop) positions, using 1-based coordinates.

n_bases : ndarray, int, shape (n_windows,)

Number of (accessible) bases in each window.

counts : ndarray, int, shape (n_windows,)

Number of variants in each window.

See also:

`allel.model.locate_fixed_differences`

4.2.2 F-statistics

`allel.`**`weir_cockerham_fst`**(*g*, *subpops*, *max_allele=None*, *blen=None*)

Compute the variance components from the analyses of variance of allele frequencies according to Weir and Cockerham (1984).

Parameters **g** : array_like, int, shape (n_variants, n_samples, ploidy)

Genotype array.

subpops : sequence of sequences of ints

Sample indices for each subpopulation.

max_allele : int, optional

The highest allele index to consider.

blen : int, optional

Block length to use for chunked computation.

Returns **a** : ndarray, float, shape (n_variants, n_alleles)

Component of variance between populations.

b : ndarray, float, shape (n_variants, n_alleles)

Component of variance between individuals within populations.

c : ndarray, float, shape (n_variants, n_alleles)

Component of variance between gametes within individuals.

Examples

Calculate variance components from some genotype data:

```
>>> import allel
>>> g = [[[0, 0], [0, 0], [1, 1], [1, 1]],
...      [[0, 1], [0, 1], [0, 1], [0, 1]],
...      [[0, 0], [0, 0], [0, 0], [0, 0]],
...      [[0, 1], [1, 2], [1, 1], [2, 2]],
...      [[0, 0], [1, 1], [0, 1], [-1, -1]]]
>>> subpops = [[0, 1], [2, 3]]
>>> a, b, c = allel.stats.weir_cockerham_fst(g, subpops)
>>> a
array([[ 0.5  ,  0.5  ,  0.   ],
       [ 0.   ,  0.   ,  0.   ],
       [ 0.   ,  0.   ,  0.   ],
       [ 0.   , -0.125, -0.125],
       [-0.375, -0.375,  0.   ]])
>>> b
array([[ 0.        ,  0.        ,  0.        ],
```

```
       [-0.25      , -0.25      ,  0.        ],
       [ 0.        ,  0.        ,  0.        ],
       [ 0.        ,  0.125     ,  0.25      ],
       [ 0.41666667,  0.41666667,  0.        ]])
>>> c
array([[ 0.        ,  0.        ,  0.        ],
       [ 0.5       ,  0.5       ,  0.        ],
       [ 0.        ,  0.        ,  0.        ],
       [ 0.125     ,  0.25      ,  0.125     ],
       [ 0.16666667,  0.16666667,  0.        ]])
```

Estimate the parameter theta (a.k.a., Fst) for each variant and each allele individually:

```
>>> fst = a / (a + b + c)
>>> fst
array([[ 1. ,  1. ,  nan],
       [ 0. ,  0. ,  nan],
       [ nan,  nan,  nan],
       [ 0. , -0.5, -0.5],
       [-1.8, -1.8,  nan]])
```

Estimate Fst for each variant individually (averaging over alleles):

```
>>> fst = (np.sum(a, axis=1) /
...        (np.sum(a, axis=1) + np.sum(b, axis=1) + np.sum(c, axis=1)))
>>> fst
array([ 1. ,  0. ,  nan, -0.4, -1.8])
```

Estimate Fst averaging over all variants and alleles:

```
>>> fst = np.sum(a) / (np.sum(a) + np.sum(b) + np.sum(c))
>>> fst
-4.3680905886891398e-17
```

Note that estimated Fst values may be negative.

allel.**hudson_fst**(*ac1*, *ac2*, *fill=nan*)

Calculate the numerator and denominator for Fst estimation using the method of Hudson (1992) elaborated by Bhatia et al. (2013).

Parameters **ac1** : array_like, int, shape (n_variants, n_alleles)

Allele counts array from the first population.

ac2 : array_like, int, shape (n_variants, n_alleles)

Allele counts array from the second population.

fill : float

Use this value where there are no pairs to compare (e.g., all allele calls are missing).

Returns **num** : ndarray, float, shape (n_variants,)

Divergence between the two populations minus average of diversity within each population.

den : ndarray, float, shape (n_variants,)

Divergence between the two populations.

Examples

Calculate numerator and denominator for Fst estimation:

```
>>> import allel
>>> g = allel.GenotypeArray([[[0, 0], [0, 0], [1, 1], [1, 1]],
...                          [[0, 1], [0, 1], [0, 1], [0, 1]],
...                          [[0, 0], [0, 0], [0, 0], [0, 0]],
...                          [[0, 1], [1, 2], [1, 1], [2, 2]],
...                          [[0, 0], [1, 1], [0, 1], [-1, -1]]])
>>> subpops = [[0, 1], [2, 3]]
>>> ac1 = g.count_alleles(subpop=subpops[0])
>>> ac2 = g.count_alleles(subpop=subpops[1])
>>> num, den = allel.stats.hudson_fst(ac1, ac2)
>>> num
array([ 1.        , -0.16666667,  0.        , -0.125     , -0.33333333])
>>> den
array([ 1.   ,  0.5  ,  0.   ,  0.625,  0.5  ])
```

Estimate Fst for each variant individually:

```
>>> fst = num / den
>>> fst
array([ 1.        , -0.33333333,         nan, -0.2       , -0.66666667])
```

Estimate Fst averaging over variants:

```
>>> fst = np.sum(num) / np.sum(den)
>>> fst
0.1428571428571429
```

allel.**patterson_fst**(*aca*, *acb*)

Estimator of differentiation between populations A and B based on the F2 parameter.

Parameters **aca** : array_like, int, shape (n_variants, 2)

Allele counts for population A.

acb : array_like, int, shape (n_variants, 2)

Allele counts for population B.

Returns **num** : ndarray, shape (n_variants,), float

Numerator.

den : ndarray, shape (n_variants,), float

Denominator.

Notes

See Patterson (2012), Appendix A.

TODO check if this is numerically equivalent to Hudson's estimator.

allel.**average_weir_cockerham_fst**(*g*, *subpops*, *blen*, *max_allele=None*)

Estimate average Fst and standard error using the block-jackknife.

Parameters **g** : array_like, int, shape (n_variants, n_samples, ploidy)

Genotype array.

subpops : sequence of sequences of ints

Sample indices for each subpopulation.

blen : int

Block size (number of variants).

max_allele : int, optional

The highest allele index to consider.

Returns **fst** : float

Estimated value of the statistic using all data.

se : float

Estimated standard error.

vb : ndarray, float, shape (n_blocks,)

Value of the statistic in each block.

vj : ndarray, float, shape (n_blocks,)

Values of the statistic from block-jackknife resampling.

allel.**average_hudson_fst**(*ac1*, *ac2*, *blen*)

Estimate average Fst between two populations and standard error using the block-jackknife.

Parameters **ac1** : array_like, int, shape (n_variants, n_alleles)

Allele counts array from the first population.

ac2 : array_like, int, shape (n_variants, n_alleles)

Allele counts array from the second population.

blen : int

Block size (number of variants).

Returns **fst** : float

Estimated value of the statistic using all data.

se : float

Estimated standard error.

vb : ndarray, float, shape (n_blocks,)

Value of the statistic in each block.

vj : ndarray, float, shape (n_blocks,)

Values of the statistic from block-jackknife resampling.

allel.**average_patterson_fst**(*ac1*, *ac2*, *blen*)

Estimate average Fst between two populations and standard error using the block-jackknife.

Parameters **ac1** : array_like, int, shape (n_variants, n_alleles)

Allele counts array from the first population.

ac2 : array_like, int, shape (n_variants, n_alleles)

Allele counts array from the second population.

blen : int

Block size (number of variants).

Returns **fst** : float

Estimated value of the statistic using all data.

se : float

Estimated standard error.

vb : ndarray, float, shape (n_blocks,)

Value of the statistic in each block.

vj : ndarray, float, shape (n_blocks,)

Values of the statistic from block-jackknife resampling.

allel.**moving_weir_cockerham_fst**(*g*, *subpops*, *size*, *start=0*, *stop=None*, *step=None*, *max_allele=None*)

Estimate average Fst in moving windows over a single chromosome/contig, following the method of Weir and Cockerham (1984).

Parameters **g** : array_like, int, shape (n_variants, n_samples, ploidy)

Genotype array.

subpops : sequence of sequences of ints

Sample indices for each subpopulation.

size : int

The window size (number of variants).

start : int, optional

The index at which to start.

stop : int, optional

The index at which to stop.

step : int, optional

The number of variants between start positions of windows. If not given, defaults to the window size, i.e., non-overlapping windows.

max_allele : int, optional

The highest allele index to consider.

Returns **fst** : ndarray, float, shape (n_windows,)

Average Fst in each window.

allel.**moving_hudson_fst**(*ac1*, *ac2*, *size*, *start=0*, *stop=None*, *step=None*)

Estimate average Fst in moving windows over a single chromosome/contig, following the method of Hudson (1992) elaborated by Bhatia et al. (2013).

Parameters **ac1** : array_like, int, shape (n_variants, n_alleles)

Allele counts array from the first population.

ac2 : array_like, int, shape (n_variants, n_alleles)

Allele counts array from the second population.

size : int

The window size (number of variants).

start : int, optional

The index at which to start.

stop : int, optional

The index at which to stop.

step : int, optional

The number of variants between start positions of windows. If not given, defaults to the window size, i.e., non-overlapping windows.

Returns **fst** : ndarray, float, shape (n_windows,)

Average Fst in each window.

allel.**moving_patterson_fst**(*ac1*, *ac2*, *size*, *start=0*, *stop=None*, *step=None*)

Estimate average Fst in moving windows over a single chromosome/contig, following the method of Patterson (2012).

Parameters **ac1** : array_like, int, shape (n_variants, n_alleles)

Allele counts array from the first population.

ac2 : array_like, int, shape (n_variants, n_alleles)

Allele counts array from the second population.

size : int

The window size (number of variants).

start : int, optional

The index at which to start.

stop : int, optional

The index at which to stop.

step : int, optional

The number of variants between start positions of windows. If not given, defaults to the window size, i.e., non-overlapping windows.

Returns **fst** : ndarray, float, shape (n_windows,)

Average Fst in each window.

allel.**windowed_weir_cockerham_fst**(*pos*, *g*, *subpops*, *size=None*, *start=None*, *stop=None*, *step=None*, *windows=None*, *fill=nan*, *max_allele=None*)

Estimate average Fst in windows over a single chromosome/contig, following the method of Weir and Cockerham (1984).

Parameters **pos** : array_like, int, shape (n_items,)

Variant positions, using 1-based coordinates, in ascending order.

g : array_like, int, shape (n_variants, n_samples, ploidy)

Genotype array.

subpops : sequence of sequences of ints

Sample indices for each subpopulation.

size : int

The window size (number of bases).

start : int, optional

The position at which to start (1-based).

stop : int, optional

The position at which to stop (1-based).

step : int, optional

The distance between start positions of windows. If not given, defaults to the window size, i.e., non-overlapping windows.

windows : array_like, int, shape (n_windows, 2), optional

Manually specify the windows to use as a sequence of (window_start, window_stop) positions, using 1-based coordinates. Overrides the size/start/stop/step parameters.

fill : object, optional

The value to use where there are no variants within a window.

max_allele : int, optional

The highest allele index to consider.

Returns **fst** : ndarray, float, shape (n_windows,)

Average Fst in each window.

windows : ndarray, int, shape (n_windows, 2)

The windows used, as an array of (window_start, window_stop) positions, using 1-based coordinates.

counts : ndarray, int, shape (n_windows,)

Number of variants in each window.

allel.**windowed_hudson_fst**(*pos*, *ac1*, *ac2*, *size=None*, *start=None*, *stop=None*, *step=None*, *windows=None*, *fill=nan*)

Estimate average Fst in windows over a single chromosome/contig, following the method of Hudson (1992) elaborated by Bhatia et al. (2013).

Parameters **pos** : array_like, int, shape (n_items,)

Variant positions, using 1-based coordinates, in ascending order.

ac1 : array_like, int, shape (n_variants, n_alleles)

Allele counts array from the first population.

ac2 : array_like, int, shape (n_variants, n_alleles)

Allele counts array from the second population.

size : int, optional

The window size (number of bases).

start : int, optional

The position at which to start (1-based).

stop : int, optional

The position at which to stop (1-based).

step : int, optional

The distance between start positions of windows. If not given, defaults to the window size, i.e., non-overlapping windows.

windows : array_like, int, shape (n_windows, 2), optional

Manually specify the windows to use as a sequence of (window_start, window_stop) positions, using 1-based coordinates. Overrides the size/start/stop/step parameters.

fill : object, optional

The value to use where there are no variants within a window.

Returns **fst** : ndarray, float, shape (n_windows,)

Average Fst in each window.

windows : ndarray, int, shape (n_windows, 2)

The windows used, as an array of (window_start, window_stop) positions, using 1-based coordinates.

counts : ndarray, int, shape (n_windows,)

Number of variants in each window.

allel.**windowed_patterson_fst**(*pos*, *ac1*, *ac2*, *size=None*, *start=None*, *stop=None*, *step=None*, *windows=None*, *fill=nan*)

Estimate average Fst in windows over a single chromosome/contig, following the method of Patterson (2012).

Parameters **pos** : array_like, int, shape (n_items,)

Variant positions, using 1-based coordinates, in ascending order.

ac1 : array_like, int, shape (n_variants, n_alleles)

Allele counts array from the first population.

ac2 : array_like, int, shape (n_variants, n_alleles)

Allele counts array from the second population.

size : int, optional

The window size (number of bases).

start : int, optional

The position at which to start (1-based).

stop : int, optional

The position at which to stop (1-based).

step : int, optional

The distance between start positions of windows. If not given, defaults to the window size, i.e., non-overlapping windows.

windows : array_like, int, shape (n_windows, 2), optional

Manually specify the windows to use as a sequence of (window_start, window_stop) positions, using 1-based coordinates. Overrides the size/start/stop/step parameters.

fill : object, optional

The value to use where there are no variants within a window.

Returns **fst** : ndarray, float, shape (n_windows,)

Average Fst in each window.

windows : ndarray, int, shape (n_windows, 2)

The windows used, as an array of (window_start, window_stop) positions, using 1-based coordinates.

counts : ndarray, int, shape (n_windows,)

Number of variants in each window.

4.2.3 Hardy-Weinberg equilibrium

allel.**heterozygosity_observed**(*g*, *fill=nan*)

Calculate the rate of observed heterozygosity for each variant.

Parameters **g** : array_like, int, shape (n_variants, n_samples, ploidy)

Genotype array.

fill : float, optional

Use this value for variants where all calls are missing.

Returns **ho** : ndarray, float, shape (n_variants,)

Observed heterozygosity

Examples

```
>>> import allel
>>> g = allel.GenotypeArray([[[0, 0], [0, 0], [0, 0]],
...                          [[0, 0], [0, 1], [1, 1]],
...                          [[0, 0], [1, 1], [2, 2]],
...                          [[1, 1], [1, 2], [-1, -1]]])
>>> allel.stats.heterozygosity_observed(g)
array([ 0.        ,  0.33333333,  0.        ,  0.5       ])
```

allel.**heterozygosity_expected**(*af*, *ploidy*, *fill=nan*)

Calculate the expected rate of heterozygosity for each variant under Hardy-Weinberg equilibrium.

Parameters **af** : array_like, float, shape (n_variants, n_alleles)

Allele frequencies array.

ploidy : int

Sample ploidy.

fill : float, optional

Use this value for variants where allele frequencies do not sum to 1.

Returns **he** : ndarray, float, shape (n_variants,)

Expected heterozygosity

Examples

```
>>> import allel
>>> g = allel.GenotypeArray([[[0, 0], [0, 0], [0, 0]],
...                          [[0, 0], [0, 1], [1, 1]],
...                          [[0, 0], [1, 1], [2, 2]],
...                          [[1, 1], [1, 2], [-1, -1]]])
>>> af = g.count_alleles().to_frequencies()
>>> allel.stats.heterozygosity_expected(af, ploidy=2)
array([ 0.        ,  0.5       ,  0.66666667,  0.375     ])
```

allel.**inbreeding_coefficient**(*g*, *fill=nan*)

Calculate the inbreeding coefficient for each variant.

Parameters **g** : array_like, int, shape (n_variants, n_samples, ploidy)

Genotype array.

fill : float, optional

Use this value for variants where the expected heterozygosity is zero.

Returns **f** : ndarray, float, shape (n_variants,)

Inbreeding coefficient.

Notes

The inbreeding coefficient is calculated as *1 - (Ho/He)* where *Ho* is the observed heterozygosity and *He* is the expected heterozygosity.

Examples

```
>>> import allel
>>> g = allel.GenotypeArray([[[0, 0], [0, 0], [0, 0]],
...                          [[0, 0], [0, 1], [1, 1]],
...                          [[0, 0], [1, 1], [2, 2]],
...                          [[1, 1], [1, 2], [-1, -1]]])
>>> allel.stats.inbreeding_coefficient(g)
array([        nan,  0.33333333,  1.        , -0.33333333])
```

4.2.4 Linkage disequilibrium

allel.**rogers_huff_r**(*gn*, *fill=nan*)

Estimate the linkage disequilibrium parameter *r* for each pair of variants using the method of Rogers and Huff (2008).

Parameters **gn** : array_like, int8, shape (n_variants, n_samples)

Diploid genotypes at biallelic variants, coded as the number of alternate alleles per call (i.e., 0 = hom ref, 1 = het, 2 = hom alt).

fill : float, optional

Value to use where r cannot be calculated.

Returns **r** : ndarray, float, shape (n_variants * (n_variants - 1) // 2,)

Matrix in condensed form.

Examples

```
>>> import allel
>>> g = allel.GenotypeArray([[[0, 0], [1, 1], [0, 0]],
...                          [[0, 0], [1, 1], [0, 0]],
...                          [[1, 1], [0, 0], [1, 1]],
...                          [[0, 0], [0, 1], [-1, -1]]], dtype='i1')
>>> gn = g.to_n_alt(fill=-1)
>>> gn
array([[ 0,  2,  0],
       [ 0,  2,  0],
       [ 2,  0,  2],
       [ 0,  1, -1]], dtype=int8)
>>> r = allel.stats.rogers_huff_r(gn)
>>> r
array([ 1.        , -1.00000012,  1.        , -1.00000012,  1.        , -1.
↪ ], ...
>>> r ** 2
array([ 1.        ,  1.00000024,  1.        ,  1.00000024,  1.        ,  1.
↪ ], ...
>>> from scipy.spatial.distance import squareform
>>> squareform(r ** 2)
array([[ 0.        ,  1.        ,  1.00000024,  1.        ],
       [ 1.        ,  0.        ,  1.00000024,  1.        ],
       [ 1.00000024,  1.00000024,  0.        ,  1.        ],
       [ 1.        ,  1.        ,  1.        ,  0.        ]], dtype=float32)
```

allel.**rogers_huff_r_between**(*gna*, *gnb*, *fill=nan*)

Estimate the linkage disequilibrium parameter *r* for each pair of variants between the two input arrays, using the method of Rogers and Huff (2008).

Parameters **gna, gnb** : array_like, int8, shape (n_variants, n_samples)

Diploid genotypes at biallelic variants, coded as the number of alternate alleles per call (i.e., 0 = hom ref, 1 = het, 2 = hom alt).

fill : float, optional

Value to use where r cannot be calculated.

Returns **r** : ndarray, float, shape (m_variants, n_variants)

Matrix in rectangular form.

allel.**windowed_r_squared**(*pos*, *gn*, *size=None*, *start=None*, *stop=None*, *step=None*, *windows=None*, *fill=nan*, *percentile=50*)

Summarise linkage disequilibrium in windows over a single chromosome/contig.

Parameters **pos** : array_like, int, shape (n_items,)

The item positions in ascending order, using 1-based coordinates..

gn : array_like, int8, shape (n_variants, n_samples)

Diploid genotypes at biallelic variants, coded as the number of alternate alleles per call (i.e., 0 = hom ref, 1 = het, 2 = hom alt).

size : int, optional

The window size (number of bases).

start : int, optional

The position at which to start (1-based).

stop : int, optional

The position at which to stop (1-based).

step : int, optional

The distance between start positions of windows. If not given, defaults to the window size, i.e., non-overlapping windows.

windows : array_like, int, shape (n_windows, 2), optional

Manually specify the windows to use as a sequence of (window_start, window_stop) positions, using 1-based coordinates. Overrides the size/start/stop/step parameters.

fill : object, optional

The value to use where a window is empty, i.e., contains no items.

percentile : int or sequence of ints, optional

The percentile or percentiles to calculate within each window.

Returns **out** : ndarray, shape (n_windows,)

The value of the statistic for each window.

windows : ndarray, int, shape (n_windows, 2)

The windows used, as an array of (window_start, window_stop) positions, using 1-based coordinates.

counts : ndarray, int, shape (n_windows,)

The number of items in each window.

See also:

`allel.stats.window.windowed_statistic`

Notes

Linkage disequilibrium (r**2) is calculated using the method of Rogers and Huff (2008).

`allel.`**`locate_unlinked`**(*gn*, *size=100*, *step=20*, *threshold=0.1*, *blen=None*)

Locate variants in approximate linkage equilibrium, where r**2 is below the given *threshold*.

Parameters **gn** : array_like, int8, shape (n_variants, n_samples)

Diploid genotypes at biallelic variants, coded as the number of alternate alleles per call (i.e., 0 = hom ref, 1 = het, 2 = hom alt).

size : int

Window size (number of variants).

step : int

Number of variants to advance to the next window.

threshold : float

Maximum value of r**2 to include variants.

blen : int, optional

Block length to use for chunked computation.

Returns **loc** : ndarray, bool, shape (n_variants)

Boolean array where True items locate variants in approximate linkage equilibrium.

Notes

The value of r**2 between each pair of variants is calculated using the method of Rogers and Huff (2008).

allel.**plot_pairwise_ld**(*m*, *colorbar=True*, *ax=None*, *imshow_kwargs=None*)

Plot a matrix of genotype linkage disequilibrium values between all pairs of variants.

Parameters **m** : array_like

Array of linkage disequilibrium values in condensed form.

colorbar : bool, optional

If True, add a colorbar to the current figure.

ax : axes, optional

The axes on which to draw. If not provided, a new figure will be created.

imshow_kwargs : dict-like, optional

Additional keyword arguments passed through to `matplotlib.pyplot.imshow()`.

Returns **ax** : axes

The axes on which the plot was drawn.

4.2.5 Site frequency spectra

allel.**sfs**(*dac*)

Compute the site frequency spectrum given derived allele counts at a set of biallelic variants.

Parameters **dac** : array_like, int, shape (n_variants,)

Array of derived allele counts.

Returns **sfs** : ndarray, int, shape (n_chromosomes,)

Array where the kth element is the number of variant sites with k derived alleles.

allel.**sfs_folded**(*ac*)

Compute the folded site frequency spectrum given reference and alternate allele counts at a set of biallelic variants.

Parameters **ac** : array_like, int, shape (n_variants, 2)

Allele counts array.

Returns **sfs_folded** : ndarray, int, shape (n_chromosomes//2,)

Array where the kth element is the number of variant sites with a minor allele count of k.

allel.**sfs_scaled**(*dac*)

Compute the site frequency spectrum scaled such that a constant value is expected across the spectrum for neutral variation and constant population size.

Parameters **dac** : array_like, int, shape (n_variants,)

Array of derived allele counts.

Returns **sfs_scaled** : ndarray, int, shape (n_chromosomes,)

An array where the value of the kth element is the number of variants with k derived alleles, multiplied by k.

allel.**sfs_folded_scaled**(*ac*, *n=None*)

Compute the folded site frequency spectrum scaled such that a constant value is expected across the spectrum for neutral variation and constant population size.

Parameters **ac** : array_like, int, shape (n_variants, 2)

Allele counts array.

n : int, optional

The total number of chromosomes called at each variant site. Equal to the number of samples multiplied by the ploidy. If not provided, will be inferred to be the maximum value of the sum of reference and alternate allele counts present in *ac*.

Returns **sfs_folded_scaled** : ndarray, int, shape (n_chromosomes//2,)

An array where the value of the kth element is the number of variants with minor allele count k, multiplied by the scaling factor (k * (n - k) / n).

allel.**joint_sfs**(*dac1*, *dac2*)

Compute the joint site frequency spectrum between two populations.

Parameters **dac1** : array_like, int, shape (n_variants,)

Derived allele counts for the first population.

dac2 : array_like, int, shape (n_variants,)

Derived allele counts for the second population.

Returns **joint_sfs** : ndarray, int, shape (m_chromosomes, n_chromosomes)

Array where the (i, j)th element is the number of variant sites with i derived alleles in the first population and j derived alleles in the second population.

allel.**joint_sfs_folded**(*ac1*, *ac2*)

Compute the joint folded site frequency spectrum between two populations.

Parameters **ac1** : array_like, int, shape (n_variants, 2)

Allele counts for the first population.

ac2 : array_like, int, shape (n_variants, 2)

Allele counts for the second population.

Returns **joint_sfs_folded** : ndarray, int, shape (m_chromosomes//2, n_chromosomes//2)

Array where the (i, j)th element is the number of variant sites with a minor allele count of i in the first population and j in the second population.

allel.**joint_sfs_scaled**(*dac1*, *dac2*)

Compute the joint site frequency spectrum between two populations, scaled such that a constant value is expected across the spectrum for neutral variation, constant population size and unrelated populations.

Parameters **dac1** : array_like, int, shape (n_variants,)

Derived allele counts for the first population.

dac2 : array_like, int, shape (n_variants,)

Derived allele counts for the second population.

Returns **joint_sfs_scaled** : ndarray, int, shape (m_chromosomes, n_chromosomes)

Array where the (i, j)th element is the scaled frequency of variant sites with i derived alleles in the first population and j derived alleles in the second population.

allel.**joint_sfs_folded_scaled**(*ac1*, *ac2*, *m=None*, *n=None*)

Compute the joint folded site frequency spectrum between two populations, scaled such that a constant value is expected across the spectrum for neutral variation, constant population size and unrelated populations.

Parameters **ac1** : array_like, int, shape (n_variants, 2)

Allele counts for the first population.

ac2 : array_like, int, shape (n_variants, 2)

Allele counts for the second population.

m : int, optional

Number of chromosomes called in the first population.

n : int, optional

Number of chromosomes called in the second population.

Returns **joint_sfs_folded_scaled** : ndarray, int, shape (m_chromosomes//2, n_chromosomes//2)

Array where the (i, j)th element is the scaled frequency of variant sites with a minor allele count of i in the first population and j in the second population.

allel.**fold_sfs**(*s*, *n*)

Fold a site frequency spectrum.

Parameters **s** : array_like, int, shape (n_chromosomes,)

Site frequency spectrum

n : int

Total number of chromosomes called.

Returns **sfs_folded** : ndarray, int

Folded site frequency spectrum

allel.**fold_joint_sfs**(*s*, *m*, *n*)

Fold a joint site frequency spectrum.

Parameters **s** : array_like, int, shape (m_chromosomes, n_chromosomes)

Joint site frequency spectrum.

m : int

Number of chromosomes called in the first population.

n : int

Number of chromosomes called in the second population.

Returns **joint_sfs_folded** : ndarray, int

Folded joint site frequency spectrum.

allel.**scale_sfs**(*s*)
Scale a site frequency spectrum.

Parameters **s** : array_like, int, shape (n_chromosomes,)

Site frequency spectrum.

Returns **sfs_scaled** : ndarray, int, shape (n_chromosomes,)

Scaled site frequency spectrum.

allel.**scale_sfs_folded**(*s*, *n*)
Scale a folded site frequency spectrum.

Parameters **s** : array_like, int, shape (n_chromosomes//2,)

Folded site frequency spectrum.

n : int

Number of chromosomes called.

Returns **sfs_folded_scaled** : ndarray, int, shape (n_chromosomes//2,)

Scaled folded site frequency spectrum.

allel.**scale_joint_sfs**(*s*)
Scale a joint site frequency spectrum.

Parameters **s** : array_like, int, shape (m_chromosomes, n_chromosomes)

Joint site frequency spectrum.

Returns **joint_sfs_scaled** : ndarray, int, shape (m_chromosomes, n_chromosomes)

Scaled joint site frequency spectrum.

allel.**scale_joint_sfs_folded**(*s*, *m*, *n*)
Scale a folded joint site frequency spectrum.

Parameters **s** : array_like, int, shape (m_chromosomes//2, n_chromosomes//2)

Folded joint site frequency spectrum.

m : int

Number of chromosomes called in the first population.

n : int

Number of chromosomes called in the second population.

Returns **joint_sfs_folded_scaled** : ndarray, int, shape (m_chromosomes//2, n_chromosomes//2)

Scaled folded joint site frequency spectrum.

allel.**plot_sfs**(*s*, *yscale='log'*, *bins=None*, *n=None*, *clip_endpoints=True*, *label=None*, *plot_kwargs=None*, *ax=None*)
Plot a site frequency spectrum.

Parameters **s** : array_like, int, shape (n_chromosomes,)

Site frequency spectrum.

yscale : string, optional

Y axis scale.

bins : int or array_like, int, optional

Allele count bins.

n : int, optional

Number of chromosomes sampled. If provided, X axis will be plotted as allele frequency, otherwise as allele count.

clip_endpoints : bool, optional

If True, do not plot first and last values from frequency spectrum.

label : string, optional

Label for data series in plot.

plot_kwargs : dict-like

Additional keyword arguments, passed through to ax.plot().

ax : axes, optional

Axes on which to draw. If not provided, a new figure will be created.

Returns **ax** : axes

The axes on which the plot was drawn.

allel.**plot_sfs_folded**(**args*, ***kwargs*)

Plot a folded site frequency spectrum.

Parameters **s** : array_like, int, shape (n_chromosomes/2,)

Site frequency spectrum.

yscale : string, optional

Y axis scale.

bins : int or array_like, int, optional

Allele count bins.

n : int, optional

Number of chromosomes sampled. If provided, X axis will be plotted as allele frequency, otherwise as allele count.

clip_endpoints : bool, optional

If True, do not plot first and last values from frequency spectrum.

label : string, optional

Label for data series in plot.

plot_kwargs : dict-like

Additional keyword arguments, passed through to ax.plot().

ax : axes, optional

Axes on which to draw. If not provided, a new figure will be created.

Returns **ax** : axes

The axes on which the plot was drawn.

allel.**plot_sfs_scaled**(**args*, ***kwargs*)

Plot a scaled site frequency spectrum.

Parameters **s** : array_like, int, shape (n_chromosomes,)

Site frequency spectrum.

yscale : string, optional

Y axis scale.

bins : int or array_like, int, optional

Allele count bins.

n : int, optional

Number of chromosomes sampled. If provided, X axis will be plotted as allele frequency, otherwise as allele count.

clip_endpoints : bool, optional

If True, do not plot first and last values from frequency spectrum.

label : string, optional

Label for data series in plot.

plot_kwargs : dict-like

Additional keyword arguments, passed through to ax.plot().

ax : axes, optional

Axes on which to draw. If not provided, a new figure will be created.

Returns **ax** : axes

The axes on which the plot was drawn.

allel.**plot_sfs_folded_scaled**(**args*, ***kwargs*)

Plot a folded scaled site frequency spectrum.

Parameters **s** : array_like, int, shape (n_chromosomes/2,)

Site frequency spectrum.

yscale : string, optional

Y axis scale.

bins : int or array_like, int, optional

Allele count bins.

n : int, optional

Number of chromosomes sampled. If provided, X axis will be plotted as allele frequency, otherwise as allele count.

clip_endpoints : bool, optional

If True, do not plot first and last values from frequency spectrum.

label : string, optional

Label for data series in plot.

plot_kwargs : dict-like

Additional keyword arguments, passed through to ax.plot().

ax : axes, optional

Axes on which to draw. If not provided, a new figure will be created.

Returns **ax** : axes

The axes on which the plot was drawn.

allel.**plot_joint_sfs**(*s*, *ax=None*, *imshow_kwargs=None*)

Plot a joint site frequency spectrum.

Parameters **s** : array_like, int, shape (n_chromosomes_pop1, n_chromosomes_pop2)

Joint site frequency spectrum.

ax : axes, optional

Axes on which to draw. If not provided, a new figure will be created.

imshow_kwargs : dict-like

Additional keyword arguments, passed through to ax.imshow().

Returns **ax** : axes

The axes on which the plot was drawn.

allel.**plot_joint_sfs_folded**(**args*, ***kwargs*)

Plot a joint site frequency spectrum.

Parameters **s** : array_like, int, shape (n_chromosomes_pop1/2, n_chromosomes_pop2/2)

Joint site frequency spectrum.

ax : axes, optional

Axes on which to draw. If not provided, a new figure will be created.

imshow_kwargs : dict-like

Additional keyword arguments, passed through to ax.imshow().

Returns **ax** : axes

The axes on which the plot was drawn.

allel.**plot_joint_sfs_scaled**(**args*, ***kwargs*)

Plot a scaled joint site frequency spectrum.

Parameters **s** : array_like, int, shape (n_chromosomes_pop1, n_chromosomes_pop2)

Joint site frequency spectrum.

ax : axes, optional

Axes on which to draw. If not provided, a new figure will be created.

imshow_kwargs : dict-like

Additional keyword arguments, passed through to ax.imshow().

Returns **ax** : axes

The axes on which the plot was drawn.

allel.**plot_joint_sfs_folded_scaled**(**args*, ***kwargs*)

Plot a scaled folded joint site frequency spectrum.

Parameters **s** : array_like, int, shape (n_chromosomes_pop1/2, n_chromosomes_pop2/2)

Joint site frequency spectrum.

ax : axes, optional

Axes on which to draw. If not provided, a new figure will be created.

imshow_kwargs : dict-like

Additional keyword arguments, passed through to ax.imshow().

Returns **ax** : axes

The axes on which the plot was drawn.

4.2.6 Pairwise distance and ordination

`allel.`**`pairwise_distance`**(*x*, *metric*, *chunked=False*, *blen=None*)

Compute pairwise distance between individuals (e.g., samples or haplotypes).

Parameters **x** : array_like, shape (n, m, ...)

Array of m observations (e.g., samples or haplotypes) in a space with n dimensions (e.g., variants). Note that the order of the first two dimensions is **swapped** compared to what is expected by scipy.spatial.distance.pdist.

metric : string or function

Distance metric. See documentation for the function `scipy.spatial.distance.pdist()` for a list of built-in distance metrics.

chunked : bool, optional

If True, use a block-wise implementation to avoid loading the entire input array into memory. This means that a distance matrix will be calculated for each block of the input array, and the results will be summed to produce the final output. For some distance metrics this will return a different result from the standard implementation.

blen : int, optional

Block length to use for chunked implementation.

Returns **dist** : ndarray, shape (m * (m - 1) / 2,)

Distance matrix in condensed form.

Examples

```
>>> import allel
>>> g = allel.GenotypeArray([[[0, 0], [0, 1], [1, 1]],
...                          [[0, 1], [1, 1], [1, 2]],
...                          [[0, 2], [2, 2], [-1, -1]]])
>>> d = allel.stats.pairwise_distance(g.to_n_alt(), metric='cityblock')
>>> d
array([ 3.,  4.,  3.])
>>> import scipy.spatial
>>> scipy.spatial.distance.squareform(d)
array([[ 0.,  3.,  4.],
       [ 3.,  0.,  3.],
       [ 4.,  3.,  0.]])
```

allel.**plot_pairwise_distance**(*dist*, *labels=None*, *colorbar=True*, *ax=None*, *imshow_kwargs=None*)

Plot a pairwise distance matrix.

Parameters **dist** : array_like

The distance matrix in condensed form.

labels : sequence of strings, optional

Sample labels for the axes.

colorbar : bool, optional

If True, add a colorbar to the current figure.

ax : axes, optional

The axes on which to draw. If not provided, a new figure will be created.

imshow_kwargs : dict-like, optional

Additional keyword arguments passed through to `matplotlib.pyplot.imshow()`.

Returns **ax** : axes

The axes on which the plot was drawn

allel.**pairwise_dxy**(*pos*, *gac*, *start=None*, *stop=None*, *is_accessible=None*)

Convenience function to calculate a pairwise distance matrix using nucleotide divergence (a.k.a. Dxy) as the distance metric.

Parameters **pos** : array_like, int, shape (n_variants,)

Variant positions.

gac : array_like, int, shape (n_variants, n_samples, n_alleles)

Per-genotype allele counts.

start : int, optional

Start position of region to use.

stop : int, optional

Stop position of region to use.

is_accessible : array_like, bool, shape (len(contig),), optional

Boolean array indicating accessibility status for all positions in the chromosome/contig.

Returns **dist** : ndarray

Distance matrix in condensed form.

See also:

`allel.model.ndarray.GenotypeArray.to_allele_counts`

allel.**pcoa**(*dist*)

Perform principal coordinate analysis of a distance matrix, a.k.a. classical multi-dimensional scaling.

Parameters **dist** : array_like

Distance matrix in condensed form.

Returns **coords** : ndarray, shape (n_samples, n_dimensions)

Transformed coordinates for the samples.

explained_ratio : ndarray, shape (n_dimensions)

Variance explained by each dimension.

`allel.`**`condensed_coords`**(*i*, *j*, *n*)

Transform square distance matrix coordinates to the corresponding index into a condensed, 1D form of the matrix.

Parameters **i** : int

Row index.

j : int

Column index.

n : int

Size of the square matrix (length of first or second dimension).

Returns **ix** : int

`allel.`**`condensed_coords_within`**(*pop*, *n*)

Return indices into a condensed distance matrix for all pairwise comparisons within the given population.

Parameters **pop** : array_like, int

Indices of samples or haplotypes within the population.

n : int

Size of the square matrix (length of first or second dimension).

Returns **indices** : ndarray, int

`allel.`**`condensed_coords_between`**(*pop1*, *pop2*, *n*)

Return indices into a condensed distance matrix for all pairwise comparisons between two populations.

Parameters **pop1** : array_like, int

Indices of samples or haplotypes within the first population.

pop2 : array_like, int

Indices of samples or haplotypes within the second population.

n : int

Size of the square matrix (length of first or second dimension).

Returns **indices** : ndarray, int

4.2.7 Principal components analysis

`allel.`**`pca`**(*gn*, *n_components=10*, *copy=True*, *scaler='patterson'*, *ploidy=2*)

Perform principal components analysis of genotype data, via singular value decomposition.

Parameters **gn** : array_like, float, shape (n_variants, n_samples)

Genotypes at biallelic variants, coded as the number of alternate alleles per call (i.e., 0 = hom ref, 1 = het, 2 = hom alt).

n_components : int, optional

Number of components to keep.

copy : bool, optional

If False, data passed to fit are overwritten.

scaler : {'patterson', 'standard', None}

Scaling method; 'patterson' applies the method of Patterson et al 2006; 'standard' scales to unit variance; None centers the data only.

ploidy : int, optional

Sample ploidy, only relevant if 'patterson' scaler is used.

Returns **coords** : ndarray, float, shape (n_samples, n_components)

Transformed coordinates for the samples.

model : GenotypePCA

Model instance containing the variance ratio explained and the stored components (a.k.a., loadings). Can be used to project further data into the same principal components space via the transform() method.

See also:

`randomized_pca`, `allel.stats.ld.locate_unlinked`

Notes

Genotype data should be filtered prior to using this function to remove variants in linkage disequilibrium.

`allel.`**`randomized_pca`**(*gn*, *n_components=10*, *copy=True*, *iterated_power=3*, *random_state=None*, *scaler='patterson'*, *ploidy=2*)

Perform principal components analysis of genotype data, via an approximate truncated singular value decomposition using randomization to speed up the computation.

Parameters **gn** : array_like, float, shape (n_variants, n_samples)

Genotypes at biallelic variants, coded as the number of alternate alleles per call (i.e., 0 = hom ref, 1 = het, 2 = hom alt).

n_components : int, optional

Number of components to keep.

copy : bool, optional

If False, data passed to fit are overwritten.

iterated_power : int, optional

Number of iterations for the power method.

random_state : int or RandomState instance or None (default)

Pseudo Random Number generator seed control. If None, use the numpy.random singleton.

scaler : {'patterson', 'standard', None}

Scaling method; 'patterson' applies the method of Patterson et al 2006; 'standard' scales to unit variance; None centers the data only.

ploidy : int, optional

Sample ploidy, only relevant if 'patterson' scaler is used.

Returns **coords** : ndarray, float, shape (n_samples, n_components)

Transformed coordinates for the samples.

model : GenotypeRandomizedPCA

Model instance containing the variance ratio explained and the stored components (a.k.a., loadings). Can be used to project further data into the same principal components space via the transform() method.

See also:

`pca`, `allel.stats.ld.locate_unlinked`

Notes

Genotype data should be filtered prior to using this function to remove variants in linkage disequilibrium.

Based on the `sklearn.decomposition.RandomizedPCA` implementation.

4.2.8 Admixture

`allel.`**`patterson_f2`**(*aca*, *acb*)

Unbiased estimator for F2(A, B), the branch length between populations A and B.

Parameters **aca** : array_like, int, shape (n_variants, 2)

Allele counts for population A.

acb : array_like, int, shape (n_variants, 2)

Allele counts for population B.

Returns **f2** : ndarray, float, shape (n_variants,)

Notes

See Patterson (2012), Appendix A.

`allel.`**`patterson_f3`**(*acc*, *aca*, *acb*)

Unbiased estimator for F3(C; A, B), the three-population test for admixture in population C.

Parameters **acc** : array_like, int, shape (n_variants, 2)

Allele counts for the test population (C).

aca : array_like, int, shape (n_variants, 2)

Allele counts for the first source population (A).

acb : array_like, int, shape (n_variants, 2)

Allele counts for the second source population (B).

Returns **T** : ndarray, float, shape (n_variants,)

Un-normalized f3 estimates per variant.

B : ndarray, float, shape (n_variants,)

Estimates for heterozygosity in population C.

Notes

See Patterson (2012), main text and Appendix A.

For un-normalized f3 statistics, ignore the *B* return value.

To compute the f3* statistic, which is normalized by heterozygosity in population C to remove numerical dependence on the allele frequency spectrum, compute `np.sum(T) / np.sum(B)`.

allel.**patterson_d**(*aca*, *acb*, *acc*, *acd*)

Unbiased estimator for D(A, B; C, D), the normalised four-population test for admixture between (A or B) and (C or D), also known as the "ABBA BABA" test.

Parameters **aca** : array_like, int, shape (n_variants, 2),

Allele counts for population A.

acb : array_like, int, shape (n_variants, 2)

Allele counts for population B.

acc : array_like, int, shape (n_variants, 2)

Allele counts for population C.

acd : array_like, int, shape (n_variants, 2)

Allele counts for population D.

Returns **num** : ndarray, float, shape (n_variants,)

Numerator (un-normalised f4 estimates).

den : ndarray, float, shape (n_variants,)

Denominator.

Notes

See Patterson (2012), main text and Appendix A.

For un-normalized f4 statistics, ignore the *den* return value.

allel.**average_patterson_f3**(*acc*, *aca*, *acb*, *blen*, *normed=True*)

Estimate F3(C; A, B) and standard error using the block-jackknife.

Parameters **acc** : array_like, int, shape (n_variants, 2)

Allele counts for the test population (C).

aca : array_like, int, shape (n_variants, 2)

Allele counts for the first source population (A).

acb : array_like, int, shape (n_variants, 2)

Allele counts for the second source population (B).

blen : int

Block size (number of variants).

normed : bool, optional

If False, use un-normalised f3 values.

Returns **f3** : float

Estimated value of the statistic using all data.

se : float

Estimated standard error.

z : float

Z-score (number of standard errors from zero).

vb : ndarray, float, shape (n_blocks,)

Value of the statistic in each block.

vj : ndarray, float, shape (n_blocks,)

Values of the statistic from block-jackknife resampling.

See also:

`allel.stats.admixture.patterson_f3`

Notes

See Patterson (2012), main text and Appendix A.

`allel.`**`average_patterson_d`**(*aca*, *acb*, *acc*, *acd*, *blen*)

Estimate D(A, B; C, D) and standard error using the block-jackknife.

Parameters **aca** : array_like, int, shape (n_variants, 2),

Allele counts for population A.

acb : array_like, int, shape (n_variants, 2)

Allele counts for population B.

acc : array_like, int, shape (n_variants, 2)

Allele counts for population C.

acd : array_like, int, shape (n_variants, 2)

Allele counts for population D.

blen : int

Block size (number of variants).

Returns **d** : float

Estimated value of the statistic using all data.

se : float

Estimated standard error.

z : float

Z-score (number of standard errors from zero).

vb : ndarray, float, shape (n_blocks,)

Value of the statistic in each block.

vj : ndarray, float, shape (n_blocks,)

Values of the statistic from block-jackknife resampling.

See also:

`allel.stats.admixture.patterson_d`

Notes

See Patterson (2012), main text and Appendix A.

`allel.`**`moving_patterson_f3`**(*acc*, *aca*, *acb*, *size*, *start=0*, *stop=None*, *step=None*, *normed=True*)
Estimate F3(C; A, B) in moving windows.

Parameters **acc** : array_like, int, shape (n_variants, 2)

Allele counts for the test population (C).

aca : array_like, int, shape (n_variants, 2)

Allele counts for the first source population (A).

acb : array_like, int, shape (n_variants, 2)

Allele counts for the second source population (B).

size : int

The window size (number of variants).

start : int, optional

The index at which to start.

stop : int, optional

The index at which to stop.

step : int, optional

The number of variants between start positions of windows. If not given, defaults to the window size, i.e., non-overlapping windows.

normed : bool, optional

If False, use un-normalised f3 values.

Returns **f3** : ndarray, float, shape (n_windows,)

Estimated value of the statistic in each window.

`allel.`**`moving_patterson_d`**(*aca*, *acb*, *acc*, *acd*, *size*, *start=0*, *stop=None*, *step=None*)
Estimate D(A, B; C, D) in moving windows.

Parameters **aca** : array_like, int, shape (n_variants, 2),

Allele counts for population A.

acb : array_like, int, shape (n_variants, 2)

Allele counts for population B.

acc : array_like, int, shape (n_variants, 2)

Allele counts for population C.

acd : array_like, int, shape (n_variants, 2)

Allele counts for population D.

size : int

The window size (number of variants).

start : int, optional

The index at which to start.

stop : int, optional

The index at which to stop.

step : int, optional

The number of variants between start positions of windows. If not given, defaults to the window size, i.e., non-overlapping windows.

Returns **d** : ndarray, float, shape (n_windows,)

Estimated value of the statistic in each window.

4.2.9 Selection

allel.**ihs**(*h*, *pos*, *map_pos=None*, *min_ehh=0.05*, *min_maf=0.05*, *include_edges=False*, *gap_scale=20000*, *max_gap=200000*, *is_accessible=None*, *use_threads=True*)

Compute the unstandardized integrated haplotype score (IHS) for each variant, comparing integrated haplotype homozygosity between the reference (0) and alternate (1) alleles.

Parameters **h** : array_like, int, shape (n_variants, n_haplotypes)

Haplotype array.

pos : array_like, int, shape (n_variants,)

Variant positions (physical distance).

map_pos : array_like, float, shape (n_variants,)

Variant positions (genetic map distance).

min_ehh: float, optional

Minimum EHH beyond which to truncate integrated haplotype homozygosity calculation.

min_maf : float, optional

Do not compute integrated haplotype homozogysity for variants with minor allele frequency below this value.

include_edges : bool, optional

If True, report scores even if EHH does not decay below *min_ehh* before reaching the edge of the data.

gap_scale : int, optional

Rescale distance between variants if gap is larger than this value.

max_gap : int, optional

Do not report scores if EHH spans a gap larger than this number of base pairs.

is_accessible : array_like, bool, optional

Genome accessibility array. If provided, distance between variants will be computed as the number of accessible bases between them.

use_threads : bool, optional

If True use multiple threads to compute.

Returns **score** : ndarray, float, shape (n_variants,)

Unstandardized IHS scores.

See also:

`standardize_by_allele_count`

Notes

This function will calculate IHS for all variants. To exclude variants below a given minor allele frequency, filter the input haplotype array before passing to this function.

This function computes IHS comparing the reference and alternate alleles. These can be polarised by switching the sign for any variant where the reference allele is derived.

This function returns NaN for any IHS calculations where haplotype homozygosity does not decay below *min_ehh* before reaching the first or last variant. To disable this behaviour, set *include_edges* to True.

Note that the unstandardized score is returned. Usually these scores are then standardized in different allele frequency bins.

`allel.`**`xpehh`**(*h1*, *h2*, *pos*, *map_pos=None*, *min_ehh=0.05*, *include_edges=False*, *gap_scale=20000*, *max_gap=200000*, *is_accessible=None*, *use_threads=True*)

Compute the unstandardized cross-population extended haplotype homozygosity score (XPEHH) for each variant.

Parameters **h1** : array_like, int, shape (n_variants, n_haplotypes)

Haplotype array for the first population.

h2 : array_like, int, shape (n_variants, n_haplotypes)

Haplotype array for the second population.

pos : array_like, int, shape (n_variants,)

Variant positions on physical or genetic map.

map_pos : array_like, float, shape (n_variants,)

Variant positions (genetic map distance).

min_ehh: float, optional

Minimum EHH beyond which to truncate integrated haplotype homozygosity calculation.

include_edges : bool, optional

If True, report scores even if EHH does not decay below *min_ehh* before reaching the edge of the data.

gap_scale : int, optional

Rescale distance between variants if gap is larger than this value.

max_gap : int, optional

Do not report scores if EHH spans a gap larger than this number of base pairs.

is_accessible : array_like, bool, optional

Genome accessibility array. If provided, distance between variants will be computed as the number of accessible bases between them.

use_threads : bool, optional

If True use multiple threads to compute.

Returns **score** : ndarray, float, shape (n_variants,)

Unstandardized XPEHH scores.

See also:

`standardize`

Notes

This function will calculate XPEHH for all variants. To exclude variants below a given minor allele frequency, filter the input haplotype arrays before passing to this function.

This function returns NaN for any EHH calculations where haplotype homozygosity does not decay below *min_ehh* before reaching the first or last variant. To disable this behaviour, set *include_edges* to True.

Note that the unstandardized score is returned. Usually these scores are then standardized genome-wide.

Haplotype arrays from the two populations may have different numbers of haplotypes.

`allel.`**`nsl`**`(`*h*, *use_threads=True*`)`

Compute the unstandardized number of segregating sites by length (nSl) for each variant, comparing the reference and alternate alleles, after Ferrer-Admetlla et al. (2014).

Parameters **h** : array_like, int, shape (n_variants, n_haplotypes)

Haplotype array.

use_threads : bool, optional

If True use multiple threads to compute.

Returns **score** : ndarray, float, shape (n_variants,)

See also:

`standardize_by_allele_count`

Notes

This function will calculate nSl for all variants. To exclude variants below a given minor allele frequency, filter the input haplotype array before passing to this function.

This function computes nSl by comparing the reference and alternate alleles. These can be polarised by switching the sign for any variant where the reference allele is derived.

This function does nothing about nSl calculations where haplotype homozygosity extends up to the first or last variant. There may be edge effects.

Note that the unstandardized score is returned. Usually these scores are then standardized in different allele frequency bins.

`allel.`**`xpnsl`**`(`*h1*, *h2*, *use_threads=True*`)`

Cross-population version of the NSL statistic.

Parameters **h1** : array_like, int, shape (n_variants, n_haplotypes)

Haplotype array for the first population.

h2 : array_like, int, shape (n_variants, n_haplotypes)

Haplotype array for the second population.

use_threads : bool, optional

If True use multiple threads to compute.

Returns **score** : ndarray, float, shape (n_variants,)

Unstandardized XPNSL scores.

allel.**standardize**(*score*)

Centre and scale to unit variance.

allel.**standardize_by_allele_count**(*score*, *aac*, *bins=None*, *n_bins=None*, *diagnostics=True*)

Standardize *score* within allele frequency bins.

Parameters **score** : array_like, float

The score to be standardized, e.g., IHS or NSL.

aac : array_like, int

An array of alternate allele counts.

bins : array_like, int, optional

Allele count bins, overrides *n_bins*.

n_bins : int, optional

Number of allele count bins to use.

diagnostics : bool, optional

If True, plot some diagnostic information about the standardization.

Returns **score_standardized** : ndarray, float

Standardized scores.

bins : ndarray, int

Allele count bins used for standardization.

allel.**ehh_decay**(*h*, *truncate=False*)

Compute the decay of extended haplotype homozygosity (EHH) moving away from the first variant.

Parameters **h** : array_like, int, shape (n_variants, n_haplotypes)

Haplotype array.

truncate : bool, optional

If True, the return array will exclude trailing zeros.

Returns **ehh** : ndarray, float, shape (n_variants,)

EHH at successive variants from the first variant.

allel.**voight_painting**(*h*)

Paint haplotypes, assigning a unique integer to each shared haplotype prefix.

Parameters **h** : array_like, int, shape (n_variants, n_haplotypes)

Haplotype array.

Returns **painting** : ndarray, int, shape (n_variants, n_haplotypes)

Painting array.

indices : ndarray, int, shape (n_hapotypes,)

Haplotype indices after sorting by prefix.

`allel.`**`plot_voight_painting`**(*painting*, *palette='colorblind'*, *flank='right'*, *ax=None*, *height_factor=0.01*)

Plot a painting of shared haplotype prefixes.

Parameters **painting** : array_like, int, shape (n_variants, n_haplotypes)

Painting array.

ax : axes, optional

The axes on which to draw. If not provided, a new figure will be created.

palette : string, optional

A Seaborn palette name.

flank : {'right', 'left'}, optional

If left, painting will be reversed along first axis.

height_factor : float, optional

If no axes provided, determine height of figure by multiplying height of painting array by this number.

Returns **ax** : axes

`allel.`**`fig_voight_painting`**(*h*, *index=None*, *palette='colorblind'*, *height_factor=0.01*, *fig=None*)

Make a figure of shared haplotype prefixes for both left and right flanks, centred on some variant of choice.

Parameters **h** : array_like, int, shape (n_variants, n_haplotypes)

Haplotype array.

index : int, optional

Index of the variant within the haplotype array to centre on. If not provided, the middle variant will be used.

palette : string, optional

A Seaborn palette name.

height_factor : float, optional

If no axes provided, determine height of figure by multiplying height of painting array by this number.

fig : figure

The figure on which to draw. If not provided, a new figure will be created.

Returns **fig** : figure

Notes

N.B., the ordering of haplotypes on the left and right flanks will be different. This means that haplotypes on the right flank **will not** correspond to haplotypes on the left flank at the same vertical position.

allel.**haplotype_diversity**(*h*)
Estimate haplotype diversity.

Parameters **h** : array_like, int, shape (n_variants, n_haplotypes)

Haplotype array.

Returns **hd** : float

Haplotype diversity.

allel.**moving_haplotype_diversity**(*h*, *size*, *start=0*, *stop=None*, *step=None*)
Estimate haplotype diversity in moving windows.

Parameters **h** : array_like, int, shape (n_variants, n_haplotypes)

Haplotype array.

size : int

The window size (number of variants).

start : int, optional

The index at which to start.

stop : int, optional

The index at which to stop.

step : int, optional

The number of variants between start positions of windows. If not given, defaults to the window size, i.e., non-overlapping windows.

Returns **hd** : ndarray, float, shape (n_windows,)

Haplotype diversity.

allel.**garud_h**(*h*)
Compute the H1, H12, H123 and H2/H1 statistics for detecting signatures of soft sweeps, as defined in Garud et al. (2015).

Parameters **h** : array_like, int, shape (n_variants, n_haplotypes)

Haplotype array.

Returns **h1** : float

H1 statistic (sum of squares of haplotype frequencies).

h12 : float

H12 statistic (sum of squares of haplotype frequencies, combining the two most common haplotypes into a single frequency).

h123 : float

H123 statistic (sum of squares of haplotype frequencies, combining the three most common haplotypes into a single frequency).

h2_h1 : float

H2/H1 statistic, indicating the "softness" of a sweep.

allel.**moving_garud_h**(*h*, *size*, *start=0*, *stop=None*, *step=None*)
Compute the H1, H12, H123 and H2/H1 statistics for detecting signatures of soft sweeps, as defined in Garud et al. (2015), in moving windows,

Parameters **h** : array_like, int, shape (n_variants, n_haplotypes)

Haplotype array.

size : int

The window size (number of variants).

start : int, optional

The index at which to start.

stop : int, optional

The index at which to stop.

step : int, optional

The number of variants between start positions of windows. If not given, defaults to the window size, i.e., non-overlapping windows.

Returns **h1** : ndarray, float, shape (n_windows,)

H1 statistics (sum of squares of haplotype frequencies).

h12 : ndarray, float, shape (n_windows,)

H12 statistics (sum of squares of haplotype frequencies, combining the two most common haplotypes into a single frequency).

h123 : ndarray, float, shape (n_windows,)

H123 statistics (sum of squares of haplotype frequencies, combining the three most common haplotypes into a single frequency).

h2_h1 : ndarray, float, shape (n_windows,)

H2/H1 statistics, indicating the "softness" of a sweep.

allel.**plot_haplotype_frequencies**(*h*, *palette='Paired'*, *singleton_color='w'*, *ax=None*)

Plot haplotype frequencies.

Parameters **h** : array_like, int, shape (n_variants, n_haplotypes)

Haplotype array.

palette : string, optional

A Seaborn palette name.

singleton_color : string, optional

Color to paint singleton haplotypes.

ax : axes, optional

The axes on which to draw. If not provided, a new figure will be created.

Returns **ax** : axes

allel.**plot_moving_haplotype_frequencies**(*pos*, *h*, *size*, *start=0*, *stop=None*, *n=None*, *palette='Paired'*, *singleton_color='w'*, *ax=None*)

Plot haplotype frequencies in moving windows over the genome.

Parameters **pos** : array_like, int, shape (n_items,)

Variant positions, using 1-based coordinates, in ascending order.

h : array_like, int, shape (n_variants, n_haplotypes)

Haplotype array.

size : int

The window size (number of variants).

start : int, optional

The index at which to start.

stop : int, optional

The index at which to stop.

n : int, optional

Color only the *n* most frequent haplotypes (by default, all non-singleton haplotypes are colored).

palette : string, optional

A Seaborn palette name.

singleton_color : string, optional

Color to paint singleton haplotypes.

ax : axes, optional

The axes on which to draw. If not provided, a new figure will be created.

Returns **ax** : axes

allel.**moving_delta_tajima_d**(*ac1*, *ac2*, *size*, *start=0*, *stop=None*, *step=None*)

Compute the difference in Tajima's D between two populations in moving windows.

Parameters **ac1** : array_like, int, shape (n_variants, n_alleles)

Allele counts array for the first population.

ac2 : array_like, int, shape (n_variants, n_alleles)

Allele counts array for the second population.

size : int

The window size (number of variants).

start : int, optional

The index at which to start.

stop : int, optional

The index at which to stop.

step : int, optional

The number of variants between start positions of windows. If not given, defaults to the window size, i.e., non-overlapping windows.

Returns **delta_d** : ndarray, float, shape (n_windows,)

Standardized delta Tajima's D.

See also:

allel.stats.diversity.moving_tajima_d

4.2.10 Mendelian inheritance

allel.**mendel_errors**(*parent_genotypes*, *progeny_genotypes*)

Locate genotype calls not consistent with Mendelian transmission of alleles.

Parameters **parent_genotypes** : array_like, int, shape (n_variants, 2, 2)

Genotype calls for the two parents.

progeny_genotypes : array_like, int, shape (n_variants, n_progeny, 2)

Genotype calls for the progeny.

Returns **me** : ndarray, int, shape (n_variants, n_progeny)

Count of Mendel errors for each progeny genotype call.

Examples

The following are all consistent with Mendelian transmission. Note that a value of 0 is returned for missing calls:

```
>>> import allel
>>> import numpy as np
>>> genotypes = np.array([
...     # aa x aa -> aa
...     [[0, 0], [0, 0], [0, 0], [-1, -1], [-1, -1], [-1, -1]],
...     [[1, 1], [1, 1], [1, 1], [-1, -1], [-1, -1], [-1, -1]],
...     [[2, 2], [2, 2], [2, 2], [-1, -1], [-1, -1], [-1, -1]],
...     # aa x ab -> aa or ab
...     [[0, 0], [0, 1], [0, 0], [0, 1], [-1, -1], [-1, -1]],
...     [[0, 0], [0, 2], [0, 0], [0, 2], [-1, -1], [-1, -1]],
...     [[1, 1], [0, 1], [1, 1], [0, 1], [-1, -1], [-1, -1]],
...     # aa x bb -> ab
...     [[0, 0], [1, 1], [0, 1], [-1, -1], [-1, -1], [-1, -1]],
...     [[0, 0], [2, 2], [0, 2], [-1, -1], [-1, -1], [-1, -1]],
...     [[1, 1], [2, 2], [1, 2], [-1, -1], [-1, -1], [-1, -1]],
...     # aa x bc -> ab or ac
...     [[0, 0], [1, 2], [0, 1], [0, 2], [-1, -1], [-1, -1]],
...     [[1, 1], [0, 2], [0, 1], [1, 2], [-1, -1], [-1, -1]],
...     # ab x ab -> aa or ab or bb
...     [[0, 1], [0, 1], [0, 0], [0, 1], [1, 1], [-1, -1]],
...     [[1, 2], [1, 2], [1, 1], [1, 2], [2, 2], [-1, -1]],
...     [[0, 2], [0, 2], [0, 0], [0, 2], [2, 2], [-1, -1]],
...     # ab x bc -> ab or ac or bb or bc
...     [[0, 1], [1, 2], [0, 1], [0, 2], [1, 1], [1, 2]],
...     [[0, 1], [0, 2], [0, 0], [0, 1], [0, 1], [1, 2]],
...     # ab x cd -> ac or ad or bc or bd
...     [[0, 1], [2, 3], [0, 2], [0, 3], [1, 2], [1, 3]],
... ])
>>> me = allel.stats.mendel_errors(genotypes[:, :2], genotypes[:, 2:])
>>> me
array([[0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0],
```

```
       [0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]])
```

The following are cases of 'non-parental' inheritance where one or two alleles are found in the progeny that are not present in either parent. Note that the number of errors may be 1 or 2 depending on the number of non-parental alleles:

```
>>> genotypes = np.array([
...     # aa x aa -> ab or ac or bb or cc
...     [[0, 0], [0, 0], [0, 1], [0, 2], [1, 1], [2, 2]],
...     [[1, 1], [1, 1], [0, 1], [1, 2], [0, 0], [2, 2]],
...     [[2, 2], [2, 2], [0, 2], [1, 2], [0, 0], [1, 1]],
...     # aa x ab -> ac or bc or cc
...     [[0, 0], [0, 1], [0, 2], [1, 2], [2, 2], [2, 2]],
...     [[0, 0], [0, 2], [0, 1], [1, 2], [1, 1], [1, 1]],
...     [[1, 1], [0, 1], [1, 2], [0, 2], [2, 2], [2, 2]],
...     # aa x bb -> ac or bc or cc
...     [[0, 0], [1, 1], [0, 2], [1, 2], [2, 2], [2, 2]],
...     [[0, 0], [2, 2], [0, 1], [1, 2], [1, 1], [1, 1]],
...     [[1, 1], [2, 2], [0, 1], [0, 2], [0, 0], [0, 0]],
...     # ab x ab -> ac or bc or cc
...     [[0, 1], [0, 1], [0, 2], [1, 2], [2, 2], [2, 2]],
...     [[0, 2], [0, 2], [0, 1], [1, 2], [1, 1], [1, 1]],
...     [[1, 2], [1, 2], [0, 1], [0, 2], [0, 0], [0, 0]],
...     # ab x bc -> ad or bd or cd or dd
...     [[0, 1], [1, 2], [0, 3], [1, 3], [2, 3], [3, 3]],
...     [[0, 1], [0, 2], [0, 3], [1, 3], [2, 3], [3, 3]],
...     [[0, 2], [1, 2], [0, 3], [1, 3], [2, 3], [3, 3]],
...     # ab x cd -> ae or be or ce or de
...     [[0, 1], [2, 3], [0, 4], [1, 4], [2, 4], [3, 4]],
... ])
>>> me = allel.stats.mendel_errors(genotypes[:, :2], genotypes[:, 2:])
>>> me
array([[1, 1, 2, 2],
       [1, 1, 2, 2],
       [1, 1, 2, 2],
       [1, 1, 2, 2],
       [1, 1, 2, 2],
       [1, 1, 2, 2],
       [1, 1, 2, 2],
       [1, 1, 2, 2],
       [1, 1, 2, 2],
       [1, 1, 2, 2],
       [1, 1, 2, 2],
       [1, 1, 1, 2],
       [1, 1, 1, 2],
       [1, 1, 1, 2],
       [1, 1, 1, 1]])
```

The following are cases of 'hemi-parental' inheritance, where progeny appear to have inherited two copies of an allele found only once in one of the parents:

```
>>> genotypes = np.array([
...     # aa x ab -> bb
...     [[0, 0], [0, 1], [1, 1], [-1, -1]],
...     [[0, 0], [0, 2], [2, 2], [-1, -1]],
...     [[1, 1], [0, 1], [0, 0], [-1, -1]],
...     # ab x bc -> aa or cc
...     [[0, 1], [1, 2], [0, 0], [2, 2]],
...     [[0, 1], [0, 2], [1, 1], [2, 2]],
...     [[0, 2], [1, 2], [0, 0], [1, 1]],
...     # ab x cd -> aa or bb or cc or dd
...     [[0, 1], [2, 3], [0, 0], [1, 1]],
...     [[0, 1], [2, 3], [2, 2], [3, 3]],
... ])
>>> me = allel.stats.mendel_errors(genotypes[:, :2], genotypes[:, 2:])
>>> me
array([[1, 0],
       [1, 0],
       [1, 0],
       [1, 1],
       [1, 1],
       [1, 1],
       [1, 1],
       [1, 1]])
```

The following are cases of 'uni-parental' inheritance, where progeny appear to have inherited both alleles from a single parent:

```
>>> genotypes = np.array([
...     # aa x bb -> aa or bb
...     [[0, 0], [1, 1], [0, 0], [1, 1]],
...     [[0, 0], [2, 2], [0, 0], [2, 2]],
...     [[1, 1], [2, 2], [1, 1], [2, 2]],
...     # aa x bc -> aa or bc
...     [[0, 0], [1, 2], [0, 0], [1, 2]],
...     [[1, 1], [0, 2], [1, 1], [0, 2]],
...     # ab x cd -> ab or cd
...     [[0, 1], [2, 3], [0, 1], [2, 3]],
... ])
>>> me = allel.stats.mendel_errors(genotypes[:, :2], genotypes[:, 2:])
>>> me
array([[1, 1],
       [1, 1],
       [1, 1],
       [1, 1],
       [1, 1],
       [1, 1]])
```

allel.**paint_transmission**(*parent_haplotypes*, *progeny_haplotypes*)

Paint haplotypes inherited from a single diploid parent according to their allelic inheritance.

Parameters **parent_haplotypes** : array_like, int, shape (n_variants, 2)

Both haplotypes from a single diploid parent.

progeny_haplotypes : array_like, int, shape (n_variants, n_progeny)

Haplotypes found in progeny of the given parent, inherited from the given parent. I.e.,

haplotypes from gametes of the given parent.

Returns **painting** : ndarray, uint8, shape (n_variants, n_progeny)

An array of integers coded as follows: 1 = allele inherited from first parental haplotype; 2 = allele inherited from second parental haplotype; 3 = reference allele, also carried by both parental haplotypes; 4 = non-reference allele, also carried by both parental haplotypes; 5 = non-parental allele; 6 = either or both parental alleles missing; 7 = missing allele; 0 = undetermined.

Examples

```
>>> import allel
>>> haplotypes = allel.HaplotypeArray([
...     [0, 0, 0, 1, 2, -1],
...     [0, 1, 0, 1, 2, -1],
...     [1, 0, 0, 1, 2, -1],
...     [1, 1, 0, 1, 2, -1],
...     [0, 2, 0, 1, 2, -1],
...     [0, -1, 0, 1, 2, -1],
...     [-1, 1, 0, 1, 2, -1],
...     [-1, -1, 0, 1, 2, -1],
... ], dtype='i1')
>>> painting = allel.stats.paint_transmission(haplotypes[:, :2],
...                                           haplotypes[:, 2:])
>>> painting
array([[3, 5, 5, 7],
       [1, 2, 5, 7],
       [2, 1, 5, 7],
       [5, 4, 5, 7],
       [1, 5, 2, 7],
       [6, 6, 6, 7],
       [6, 6, 6, 7],
       [6, 6, 6, 7]], dtype=uint8)
```

allel.**phase_by_transmission**(*g*, *window_size*, *copy=True*)

Phase genotypes in a trio or cross where possible using Mendelian transmission.

Parameters **g** : array_like, int, shape (n_variants, n_samples, 2)

Genotype array, with parents as first two columns and progeny as remaining columns.

window_size : int

Number of previous heterozygous sites to include when phasing each parent. A number somewhere between 10 and 100 may be appropriate, depending on levels of heterozygosity and quality of data.

copy : bool, optional

If False, attempt to phase genotypes in-place. Note that this is only possible if the input array has int8 dtype, otherwise a copy is always made regardless of this parameter.

Returns **g** : GenotypeArray

Genotype array with progeny phased where possible.

allel.**phase_progeny_by_transmission**(*g*)

Phase progeny genotypes from a trio or cross using Mendelian transmission.

Parameters **g** : array_like, int, shape (n_variants, n_samples, 2)

Genotype array, with parents as first two columns and progeny as remaining columns.

Returns **g** : ndarray, int8, shape (n_variants, n_samples, 2)

Genotype array with progeny phased where possible.

Examples

```
>>> import allel
>>> g = allel.GenotypeArray([
...      [[0, 0], [0, 0], [0, 0]],
...      [[1, 1], [1, 1], [1, 1]],
...      [[0, 0], [1, 1], [0, 1]],
...      [[1, 1], [0, 0], [0, 1]],
...      [[0, 0], [0, 1], [0, 0]],
...      [[0, 0], [0, 1], [0, 1]],
...      [[0, 1], [0, 0], [0, 1]],
...      [[0, 1], [0, 1], [0, 1]],
...      [[0, 1], [1, 2], [0, 1]],
...      [[1, 2], [0, 1], [1, 2]],
...      [[0, 1], [2, 3], [0, 2]],
...      [[2, 3], [0, 1], [1, 3]],
...      [[0, 0], [0, 0], [-1, -1]],
...      [[0, 0], [0, 0], [1, 1]],
... ], dtype='i1')
>>> g = allel.stats.phase_progeny_by_transmission(g)
>>> print(g.to_str(row_threshold=None))
0/0 0/0 0|0
1/1 1/1 1|1
0/0 1/1 0|1
1/1 0/0 1|0
0/0 0/1 0|0
0/0 0/1 0|1
0/1 0/0 1|0
0/1 0/1 0/1
0/1 1/2 0|1
1/2 0/1 2|1
0/1 2/3 0|2
2/3 0/1 3|1
0/0 0/0 ./.
0/0 0/0 1/1
>>> g.is_phased
array([[False, False,  True],
       [False, False,  True],
       [False, False,  True],
       [False, False,  True],
       [False, False,  True],
       [False, False,  True],
       [False, False,  True],
       [False, False, False],
       [False, False,  True],
       [False, False,  True],
       [False, False,  True],
       [False, False,  True],
       [False, False, False],
       [False, False, False]], dtype=bool)
```

allel.**phase_parents_by_transmission**(*g*, *window_size*)

Phase parent genotypes from a trio or cross, given progeny genotypes already phased by Mendelian transmission.

Parameters **g** : GenotypeArray

Genotype array, with parents as first two columns and progeny as remaining columns, where progeny genotypes are already phased.

window_size : int

Number of previous heterozygous sites to include when phasing each parent. A number somewhere between 10 and 100 may be appropriate, depending on levels of heterozygosity and quality of data.

Returns **g** : GenotypeArray

Genotype array with parents phased where possible.

4.2.11 Runs of homozygosity (ROH)

allel.**roh_mhmm**(*gv*, *pos*, *phet_roh=0.001*, *phet_nonroh=(0.0025, 0.01)*, *transition=1e-06*, *min_roh=0*, *is_accessible=None*, *contig_size=None*)

Call ROH (runs of homozygosity) in a single individual given a genotype vector.

This function computes the likely ROH using a Multinomial HMM model. There are 3 observable states at each position in a chromosome/contig: 0 = Hom, 1 = Het, 2 = inaccessible (i.e., unobserved).

The model is provided with a probability of observing a het in a ROH (*phet_roh*) and one or more probabilities of observing a het in a non-ROH, as this probability may not be constant across the genome (*phet_nonroh*).

Parameters **gv** : array_like, int, shape (n_variants, ploidy)

Genotype vector.

pos: array_like, int, shape (n_variants,)

Positions of variants, same 0th dimension as *gv*.

phet_roh: float, optional

Probability of observing a heterozygote in a ROH. Appropriate values will depend on de novo mutation rate and genotype error rate.

phet_nonroh: tuple of floats, optional

One or more probabilites of observing a heterozygote outside of ROH. Appropriate values will depend primarily on nucleotide diversity within the population, but also on mutation rate and genotype error rate.

transition: float, optional

Probability of moving between states.

min_roh: integer, optional

Minimum size (bp) to condsider as a ROH. Will depend on contig size and recombination rate.

is_accessible: array_like, bool, shape ('contig_size',), optional

Boolean array for each position in contig describing whether accessible or not.

contig_size: int, optional

If is_accessible not known/not provided, allows specification of total length of contig.

Returns df_roh: DataFrame

Data frame where each row describes a run of homozygosity. Columns are 'start', 'stop', 'length' and 'is_marginal'. Start and stop are 1-based, stop-inclusive.

froh: float

Proportion of genome in a ROH.

Notes

This function requires hmmlearn to be installed.

This function currently requires around 4GB memory for a contig size of ~50Mbp.

4.2.12 Window utilities

allel.**moving_statistic**(*values*, *statistic*, *size*, *start=0*, *stop=None*, *step=None*)

Calculate a statistic in a moving window over *values*.

Parameters **values** : array_like

The data to summarise.

statistic : function

The statistic to compute within each window.

size : int

The window size (number of values).

start : int, optional

The index at which to start.

stop : int, optional

The index at which to stop.

step : int, optional

The distance between start positions of windows. If not given, defaults to the window size, i.e., non-overlapping windows.

Returns **out** : ndarray, shape (n_windows,)

Examples

```
>>> import allel
>>> values = [2, 5, 8, 16]
>>> allel.stats.moving_statistic(values, np.sum, size=2)
array([ 7, 24])
>>> allel.stats.moving_statistic(values, np.sum, size=2, step=1)
array([ 7, 13, 24])
```

allel.**windowed_statistic**(*pos*, *values*, *statistic*, *size=None*, *start=None*, *stop=None*, *step=None*, *windows=None*, *fill=nan*)

Calculate a statistic from items in windows over a single chromosome/contig.

Parameters **pos** : array_like, int, shape (n_items,)

The item positions in ascending order, using 1-based coordinates..

values : array_like, int, shape (n_items,)

The values to summarise. May also be a tuple of values arrays, in which case each array will be sliced and passed through to the statistic function as separate arguments.

statistic : function

The statistic to compute.

size : int, optional

The window size (number of bases).

start : int, optional

The position at which to start (1-based).

stop : int, optional

The position at which to stop (1-based).

step : int, optional

The distance between start positions of windows. If not given, defaults to the window size, i.e., non-overlapping windows.

windows : array_like, int, shape (n_windows, 2), optional

Manually specify the windows to use as a sequence of (window_start, window_stop) positions, using 1-based coordinates. Overrides the size/start/stop/step parameters.

fill : object, optional

The value to use where a window is empty, i.e., contains no items.

Returns **out** : ndarray, shape (n_windows,)

The value of the statistic for each window.

windows : ndarray, int, shape (n_windows, 2)

The windows used, as an array of (window_start, window_stop) positions, using 1-based coordinates.

counts : ndarray, int, shape (n_windows,)

The number of items in each window.

Notes

The window stop positions are included within a window.

The final window will be truncated to the specified stop position, and so may be smaller than the other windows.

Examples

Count non-zero (i.e., True) items in non-overlapping windows:

```
>>> import allel
>>> pos = [1, 7, 12, 15, 28]
>>> values = [True, False, True, False, False]
>>> nnz, windows, counts = allel.stats.windowed_statistic(
...     pos, values, statistic=np.count_nonzero, size=10
... )
>>> nnz
array([1, 1, 0])
>>> windows
array([[ 1, 10],
       [11, 20],
       [21, 28]])
>>> counts
array([2, 2, 1])
```

Compute a sum over items in half-overlapping windows:

```
>>> values = [3, 4, 2, 6, 9]
>>> x, windows, counts = allel.stats.windowed_statistic(
...     pos, values, statistic=np.sum, size=10, step=5, fill=0
... )
>>> x
array([ 7, 12,  8,  0,  9])
>>> windows
array([[ 1, 10],
       [ 6, 15],
       [11, 20],
       [16, 25],
       [21, 28]])
>>> counts
array([2, 3, 2, 0, 1])
```

allel.**windowed_count**(*pos*, *size=None*, *start=None*, *stop=None*, *step=None*, *windows=None*)

Count the number of items in windows over a single chromosome/contig.

Parameters **pos** : array_like, int, shape (n_items,)

The item positions in ascending order, using 1-based coordinates..

size : int, optional

The window size (number of bases).

start : int, optional

The position at which to start (1-based).

stop : int, optional

The position at which to stop (1-based).

step : int, optional

The distance between start positions of windows. If not given, defaults to the window size, i.e., non-overlapping windows.

windows : array_like, int, shape (n_windows, 2), optional

Manually specify the windows to use as a sequence of (window_start, window_stop) positions, using 1-based coordinates. Overrides the size/start/stop/step parameters.

Returns **counts** : ndarray, int, shape (n_windows,)

The number of items in each window.

windows : ndarray, int, shape (n_windows, 2)

The windows used, as an array of (window_start, window_stop) positions, using 1-based coordinates.

Notes

The window stop positions are included within a window.

The final window will be truncated to the specified stop position, and so may be smaller than the other windows.

Examples

Non-overlapping windows:

```
>>> import allel
>>> pos = [1, 7, 12, 15, 28]
>>> counts, windows = allel.stats.windowed_count(pos, size=10)
>>> counts
array([2, 2, 1])
>>> windows
array([[ 1, 10],
       [11, 20],
       [21, 28]])
```

Half-overlapping windows:

```
>>> counts, windows = allel.stats.windowed_count(pos, size=10, step=5)
>>> counts
array([2, 3, 2, 0, 1])
>>> windows
array([[ 1, 10],
       [ 6, 15],
       [11, 20],
       [16, 25],
       [21, 28]])
```

allel.**per_base**(*x*, *windows*, *is_accessible=None*, *fill=nan*)

Calculate the per-base value of a windowed statistic.

Parameters **x** : array_like, shape (n_windows,)

The statistic to average per-base.

windows : array_like, int, shape (n_windows, 2)

The windows used, as an array of (window_start, window_stop) positions using 1-based coordinates.

is_accessible : array_like, bool, shape (len(contig),), optional

Boolean array indicating accessibility status for all positions in the chromosome/contig.

fill : object, optional

Use this value where there are no accessible bases in a window.

Returns **y** : ndarray, float, shape (n_windows,)

The input array divided by the number of (accessible) bases in each window.

n_bases : ndarray, int, shape (n_windows,)

The number of (accessible) bases in each window

allel.**equally_accessible_windows**(*is_accessible*, *size*)

Create windows each containing the same number of accessible bases.

Parameters **is_accessible** : array_like, bool, shape (n_bases,)

Array defining accessible status of all bases on a contig/chromosome.

size : int

Window size (number of accessible bases).

Returns **windows** : ndarray, int, shape (n_windows, 2)

Window start/stop positions (1-based).

4.2.13 Preprocessing utilities

allel.**get_scaler**(*scaler*, *copy*, *ploidy*)

class allel.**CenterScaler**(*copy=True*)

class allel.**StandardScaler**(*copy=True*)

class allel.**PattersonScaler**(*copy=True*, *ploidy=2*)

4.2.14 Miscellanea

allel.**plot_variant_locator**(*pos*, *step=None*, *ax=None*, *start=None*, *stop=None*, *flip=False*, *line_kwargs=None*)

Plot lines indicating the physical genome location of variants from a single chromosome/contig. By default the top x axis is in variant index space, and the bottom x axis is in genome position space.

Parameters **pos** : array_like

A sorted 1-dimensional array of genomic positions from a single chromosome/contig.

step : int, optional

Plot a line for every *step* variants.

ax : axes, optional

The axes on which to draw. If not provided, a new figure will be created.

start : int, optional

The start position for the region to draw.

stop : int, optional

The stop position for the region to draw.

flip : bool, optional

Flip the plot upside down.

line_kwargs : dict-like

Additional keyword arguments passed through to *plt.Line2D*.

Returns **ax** : axes

The axes on which the plot was drawn

allel.**tabulate_state_transitions**(*x*, *states*, *pos=None*)

Construct a dataframe where each row provides information about a state transition.

Parameters **x** : array_like, int

1-dimensional array of state values.

states : set

Set of states of interest. Any state value not in this set will be ignored.

pos : array_like, int, optional

Array of positions corresponding to values in *x*.

Returns **df** : DataFrame

Notes

The resulting dataframe includes one row at the start representing the first state observation and one row at the end representing the last state observation.

Examples

```
>>> import allel
>>> x = [1, 1, 0, 1, 1, 2, 2, 0, 2, 1, 1]
>>> df = allel.tabulate_state_transitions(x, states={1, 2})
>>> df
   lstate  rstate  lidx  ridx
0      -1       1    -1     0
1       1       2     4     5
2       2       1     8     9
3       1      -1    10    -1
>>> pos = [2, 4, 7, 8, 10, 14, 19, 23, 28, 30, 31]
>>> df = allel.tabulate_state_transitions(x, states={1, 2}, pos=pos)
>>> df
   lstate  rstate  lidx  ridx  lpos  rpos
0      -1       1    -1     0    -1     2
1       1       2     4     5    10    14
2       2       1     8     9    28    30
3       1      -1    10    -1    31    -1
```

allel.**tabulate_state_blocks**(*x*, *states*, *pos=None*)

Construct a dataframe where each row provides information about continuous state blocks.

Parameters **x** : array_like, int

1-dimensional array of state values.

states : set

Set of states of interest. Any state value not in this set will be ignored.

pos : array_like, int, optional

Array of positions corresponding to values in *x*.

Returns **df** : DataFrame

Examples

```
>>> import allel
>>> x = [1, 1, 0, 1, 1, 2, 2, 0, 2, 1, 1]
>>> df = allel.tabulate_state_blocks(x, states={1, 2})
>>> df
   state  support  start_lidx  start_ridx  stop_lidx  stop_ridx  size_min  \
0      1        4          -1           0          4          5         5
1      2        3           4           5          8          9         4
2      1        2           8           9         10         -1         2
   size_max is_marginal
0        -1        True
1         4       False
2        -1        True
>>> pos = [2, 4, 7, 8, 10, 14, 19, 23, 28, 30, 31]
>>> df = allel.tabulate_state_blocks(x, states={1, 2}, pos=pos)
>>> df
   state  support  start_lidx  start_ridx  stop_lidx  stop_ridx  size_min  \
0      1        4          -1           0          4          5         5
1      2        3           4           5          8          9         4
2      1        2           8           9         10         -1         2
   size_max is_marginal  start_lpos  start_rpos  stop_lpos  stop_rpos  \
0        -1        True          -1           2         10         14
1         4       False          10          14         28         30
2        -1        True          28          30         31         -1
   length_min  length_max
0           9          -1
1          15          19
2           2          -1
```

4.3 Input/output utilities

4.3.1 Variant Call Format (VCF)

`allel.`**`read_vcf`**(*input*, *fields=None*, *types=None*, *numbers=None*, *alt_number=3*, *fills=None*, *region=None*, *tabix='tabix'*, *samples=None*, *transformers=None*, *buffer_size=16384*, *chunk_length=65536*, *log=None*)

Read data from a VCF file into NumPy arrays.

Parameters **input** : string or file-like

Path to VCF file on the local file system. May be uncompressed or gzip-compatible compressed file. May also be a file-like object (e.g., *io.BytesIO*).

fields : list of strings, optional

Fields to extract data for. Should be a list of strings, e.g., `['variants/CHROM', 'variants/POS', 'variants/DP', 'calldata/GT']`. If you are feeling lazy, you can drop the 'variants/' and 'calldata/' prefixes, in which case the fields will be matched against fields declared in the VCF header, with variants taking priority over calldata if a field with the same ID exists both in INFO and FORMAT headers. I.e., `['CHROM', 'POS', 'DP', 'GT']` will work, although watch out for fields like 'DP' which can be both INFO and FORMAT. For convenience, some special string

values are also recognized. To extract all fields, provide just the string `'*'`. To extract all variants fields (including all INFO fields) provide `'variants/*'`. To extract all calldata fields (i.e., defined in FORMAT headers) provide `'calldata/*'`.

types : dict, optional

Overide data types. Should be a dictionary mapping field names to NumPy data types. E.g., providing the dictionary `{'variants/DP': 'i8', 'calldata/GQ': 'i2'}` will mean the 'variants/DP' field is stored in a 64-bit integer array, and the 'calldata/GQ' field is stored in a 16-bit integer array.

numbers : dict, optional

Override the expected number of values. Should be a dictionary mapping field names to integers. E.g., providing the dictionary `{'variants/ALT': 5, 'variants/AC': 5, 'calldata/HQ': 2}` will mean that, for each variant, 5 values are stored for the 'variants/ALT' field, 5 values are stored for the 'variants/AC' field, and for each sample, 2 values are stored for the 'calldata/HQ' field.

alt_number : int, optional

Assume this number of alternate alleles and set expected number of values accordingly for any field declared with number 'A' or 'R' in the VCF meta-information.

fills : dict, optional

Override the fill value used for empty values. Should be a dictionary mapping field names to fill values.

region : string, optional

Genomic region to extract variants for. If provided, should be a tabix-style region string, which can be either just a chromosome name (e.g., '2L'), or a chromosome name followed by 1-based beginning and end coordinates (e.g., '2L:100000-200000'). Note that only variants whose start position (POS) is within the requested range will be included. This is slightly different from the default tabix behaviour, where a variant (e.g., deletion) may be included if its position (POS) occurs before the requested region but its reference allele overlaps the region - such a variant will *not* be included in the data returned by this function.

tabix : string, optional

Name or path to tabix executable. Only required if *region* is given. Setting *tabix* to *None* will cause a fall-back to scanning through the VCF file from the beginning, which may be much slower than tabix but the only option if tabix is not available on your system and/or the VCF file has not been tabix-indexed.

samples : list of strings

Selection of samples to extract calldata for. If provided, should be a list of strings giving sample identifiers. May also be a list of integers giving indices of selected samples.

transformers : list of transformer objects, optional

Transformers for post-processing data. If provided, should be a list of Transformer objects, each of which must implement a "transform()" method that accepts a dict containing the chunk of data to be transformed. See also the `ANNTransformer` class which implements post-processing of data from SNPEFF.

buffer_size : int, optional

Size in bytes of the I/O buffer used when reading data from the underlying file or tabix

stream.

chunk_length : int, optional

Length (number of variants) of chunks in which data are processed.

log : file-like, optional

A file-like object (e.g., *sys.stderr*) to print progress information.

Returns **data** : dict[str, ndarray]

A dictionary holding arrays.

allel.**vcf_to_npz**(*input*, *output*, *compressed=True*, *overwrite=False*, *fields=None*, *types=None*, *numbers=None*, *alt_number=3*, *fills=None*, *region=None*, *tabix=True*, *samples=None*, *transformers=None*, *buffer_size=16384*, *chunk_length=65536*, *log=None*)

Read data from a VCF file into NumPy arrays and save as a .npz file.

Parameters **input** : string

Path to VCF file on the local file system. May be uncompressed or gzip-compatible compressed file. May also be a file-like object (e.g., *io.BytesIO*).

output : string

File-system path to write output to.

compressed : bool, optional

If True (default), save with compression.

overwrite : bool, optional

If False (default), do not overwrite an existing file.

fields : list of strings, optional

Fields to extract data for. Should be a list of strings, e.g., `['variants/CHROM', 'variants/POS', 'variants/DP', 'calldata/GT']`. If you are feeling lazy, you can drop the 'variants/' and 'calldata/' prefixes, in which case the fields will be matched against fields declared in the VCF header, with variants taking priority over calldata if a field with the same ID exists both in INFO and FORMAT headers. I.e., `['CHROM', 'POS', 'DP', 'GT']` will work, although watch out for fields like 'DP' which can be both INFO and FORMAT. For convenience, some special string values are also recognized. To extract all fields, provide just the string `'*'`. To extract all variants fields (including all INFO fields) provide `'variants/*'`. To extract all calldata fields (i.e., defined in FORMAT headers) provide `'calldata/*'`.

types : dict, optional

Overide data types. Should be a dictionary mapping field names to NumPy data types. E.g., providing the dictionary `{'variants/DP': 'i8', 'calldata/GQ': 'i2'}` will mean the 'variants/DP' field is stored in a 64-bit integer array, and the 'calldata/GQ' field is stored in a 16-bit integer array.

numbers : dict, optional

Override the expected number of values. Should be a dictionary mapping field names to integers. E.g., providing the dictionary `{'variants/ALT': 5, 'variants/AC': 5, 'calldata/HQ': 2}` will mean that, for each variant, 5 values are stored for the 'variants/ALT' field, 5 values are stored for the 'variants/AC' field, and for each sample, 2 values are stored for the 'calldata/HQ' field.

alt_number : int, optional

Assume this number of alternate alleles and set expected number of values accordingly for any field declared with number 'A' or 'R' in the VCF meta-information.

fills : dict, optional

Override the fill value used for empty values. Should be a dictionary mapping field names to fill values.

region : string, optional

Genomic region to extract variants for. If provided, should be a tabix-style region string, which can be either just a chromosome name (e.g., '2L'), or a chromosome name followed by 1-based beginning and end coordinates (e.g., '2L:100000-200000'). Note that only variants whose start position (POS) is within the requested range will be included. This is slightly different from the default tabix behaviour, where a variant (e.g., deletion) may be included if its position (POS) occurs before the requested region but its reference allele overlaps the region - such a variant will *not* be included in the data returned by this function.

tabix : string, optional

Name or path to tabix executable. Only required if *region* is given. Setting *tabix* to *None* will cause a fall-back to scanning through the VCF file from the beginning, which may be much slower than tabix but the only option if tabix is not available on your system and/or the VCF file has not been tabix-indexed.

samples : list of strings

Selection of samples to extract calldata for. If provided, should be a list of strings giving sample identifiers. May also be a list of integers giving indices of selected samples.

transformers : list of transformer objects, optional

Transformers for post-processing data. If provided, should be a list of Transformer objects, each of which must implement a "transform()" method that accepts a dict containing the chunk of data to be transformed. See also the `ANNTransformer` class which implements post-processing of data from SNPEFF.

buffer_size : int, optional

Size in bytes of the I/O buffer used when reading data from the underlying file or tabix

stream.

chunk_length : int, optional

Length (number of variants) of chunks in which data are processed.

log : file-like, optional

A file-like object (e.g., *sys.stderr*) to print progress information.

`allel.`**`vcf_to_hdf5`**(*input*, *output*, *group='/'*, *compression='gzip'*, *compression_opts=1*, *shuffle=False*, *overwrite=False*, *fields=None*, *types=None*, *numbers=None*, *alt_number=3*, *fills=None*, *region=None*, *tabix='tabix'*, *samples=None*, *transformers=None*, *buffer_size=16384*, *chunk_length=65536*, *chunk_width=64*, *log=None*)

Read data from a VCF file and load into an HDF5 file.

Parameters **input** : string

Path to VCF file on the local file system. May be uncompressed or gzip-compatible compressed file. May also be a file-like object (e.g., *io.BytesIO*).

output : string

File-system path to write output to.

group : string

Group within destination HDF5 file to store data in.

compression : string

Compression algorithm, e.g., 'gzip' (default).

compression_opts : int

Compression level, e.g., 1 (default).

shuffle : bool

Use byte shuffling, which may improve compression (default is False).

overwrite : bool

If False (default), do not overwrite an existing file.

fields : list of strings, optional

Fields to extract data for. Should be a list of strings, e.g., `['variants/CHROM', 'variants/POS', 'variants/DP', 'calldata/GT']`. If you are feeling lazy, you can drop the 'variants/' and 'calldata/' prefixes, in which case the fields will be matched against fields declared in the VCF header, with variants taking priority over calldata if a field with the same ID exists both in INFO and FORMAT headers. I.e., `['CHROM', 'POS', 'DP', 'GT']` will work, although watch out for fields like 'DP' which can be both INFO and FORMAT. For convenience, some special string values are also recognized. To extract all fields, provide just the string `'*'`. To extract all variants fields (including all INFO fields) provide `'variants/*'`. To extract all calldata fields (i.e., defined in FORMAT headers) provide `'calldata/*'`.

types : dict, optional

Overide data types. Should be a dictionary mapping field names to NumPy data types. E.g., providing the dictionary `{'variants/DP': 'i8', 'calldata/GQ': 'i2'}` will mean the 'variants/DP' field is stored in a 64-bit integer array, and the 'calldata/GQ' field is stored in a 16-bit integer array.

numbers : dict, optional

Override the expected number of values. Should be a dictionary mapping field names to integers. E.g., providing the dictionary `{'variants/ALT': 5, 'variants/AC': 5, 'calldata/HQ': 2}` will mean that, for each variant, 5 values are stored for the 'variants/ALT' field, 5 values are stored for the 'variants/AC' field, and for each sample, 2 values are stored for the 'calldata/HQ' field.

alt_number : int, optional

Assume this number of alternate alleles and set expected number of values accordingly for any field declared with number 'A' or 'R' in the VCF meta-information.

fills : dict, optional

Override the fill value used for empty values. Should be a dictionary mapping field names to fill values.

region : string, optional

Genomic region to extract variants for. If provided, should be a tabix-style region string, which can be either just a chromosome name (e.g., '2L'), or a chromosome name followed by 1-based beginning and end coordinates (e.g., '2L:100000-200000'). Note that

only variants whose start position (POS) is within the requested range will be included. This is slightly different from the default tabix behaviour, where a variant (e.g., deletion) may be included if its position (POS) occurs before the requested region but its reference allele overlaps the region - such a variant will *not* be included in the data returned by this function.

tabix : string, optional

Name or path to tabix executable. Only required if *region* is given. Setting *tabix* to *None* will cause a fall-back to scanning through the VCF file from the beginning, which may be much slower than tabix but the only option if tabix is not available on your system and/or the VCF file has not been tabix-indexed.

samples : list of strings

Selection of samples to extract calldata for. If provided, should be a list of strings giving sample identifiers. May also be a list of integers giving indices of selected samples.

transformers : list of transformer objects, optional

Transformers for post-processing data. If provided, should be a list of Transformer objects, each of which must implement a "transform()" method that accepts a dict containing the chunk of data to be transformed. See also the `ANNTransformer` class which implements post-processing of data from SNPEFF.

buffer_size : int, optional

Size in bytes of the I/O buffer used when reading data from the underlying file or tabix

stream.

chunk_length : int, optional

Length (number of variants) of chunks in which data are processed.

chunk_width : int, optional

Width (number of samples) to use when storing chunks in output.

log : file-like, optional

A file-like object (e.g., *sys.stderr*) to print progress information.

allel.**vcf_to_zarr**(*input*, *output*, *group='/'*, *compressor='default'*, *overwrite=False*, *fields=None*, *types=None*, *numbers=None*, *alt_number=3*, *fills=None*, *region=None*, *tabix='tabix'*, *samples=None*, *transformers=None*, *buffer_size=16384*, *chunk_length=65536*, *chunk_width=64*, *log=None*)

Read data from a VCF file and load into a Zarr on-disk store.

Parameters **input** : string

Path to VCF file on the local file system. May be uncompressed or gzip-compatible compressed file. May also be a file-like object (e.g., *io.BytesIO*).

output : string

File-system path to write output to.

group : string

Group within destination Zarr hierarchy to store data in.

compressor : compressor

Compression algorithm, e.g., zarr.Blosc(cname='zstd', clevel=1, shuffle=1).

overwrite : bool

If False (default), do not overwrite an existing file.

fields : list of strings, optional

Fields to extract data for. Should be a list of strings, e.g., `['variants/CHROM', 'variants/POS', 'variants/DP', 'calldata/GT']`. If you are feeling lazy, you can drop the 'variants/' and 'calldata/' prefixes, in which case the fields will be matched against fields declared in the VCF header, with variants taking priority over calldata if a field with the same ID exists both in INFO and FORMAT headers. I.e., `['CHROM', 'POS', 'DP', 'GT']` will work, although watch out for fields like 'DP' which can be both INFO and FORMAT. For convenience, some special string values are also recognized. To extract all fields, provide just the string `'*'`. To extract all variants fields (including all INFO fields) provide `'variants/*'`. To extract all calldata fields (i.e., defined in FORMAT headers) provide `'calldata/*'`.

types : dict, optional

Overide data types. Should be a dictionary mapping field names to NumPy data types. E.g., providing the dictionary `{'variants/DP': 'i8', 'calldata/GQ': 'i2'}` will mean the 'variants/DP' field is stored in a 64-bit integer array, and the 'calldata/GQ' field is stored in a 16-bit integer array.

numbers : dict, optional

Override the expected number of values. Should be a dictionary mapping field names to integers. E.g., providing the dictionary `{'variants/ALT': 5, 'variants/AC': 5, 'calldata/HQ': 2}` will mean that, for each variant, 5 values are stored for the 'variants/ALT' field, 5 values are stored for the 'variants/AC' field, and for each sample, 2 values are stored for the 'calldata/HQ' field.

alt_number : int, optional

Assume this number of alternate alleles and set expected number of values accordingly for any field declared with number 'A' or 'R' in the VCF meta-information.

fills : dict, optional

Override the fill value used for empty values. Should be a dictionary mapping field names to fill values.

region : string, optional

Genomic region to extract variants for. If provided, should be a tabix-style region string, which can be either just a chromosome name (e.g., '2L'), or a chromosome name followed by 1-based beginning and end coordinates (e.g., '2L:100000-200000'). Note that only variants whose start position (POS) is within the requested range will be included. This is slightly different from the default tabix behaviour, where a variant (e.g., deletion) may be included if its position (POS) occurs before the requested region but its reference allele overlaps the region - such a variant will *not* be included in the data returned by this function.

tabix : string, optional

Name or path to tabix executable. Only required if *region* is given. Setting *tabix* to *None* will cause a fall-back to scanning through the VCF file from the beginning, which may be much slower than tabix but the only option if tabix is not available on your system and/or the VCF file has not been tabix-indexed.

samples : list of strings

Selection of samples to extract calldata for. If provided, should be a list of strings giving sample identifiers. May also be a list of integers giving indices of selected samples.

transformers : list of transformer objects, optional

Transformers for post-processing data. If provided, should be a list of Transformer objects, each of which must implement a "transform()" method that accepts a dict containing the chunk of data to be transformed. See also the `ANNTransformer` class which implements post-processing of data from SNPEFF.

buffer_size : int, optional

Size in bytes of the I/O buffer used when reading data from the underlying file or tabix

stream.

chunk_length : int, optional

Length (number of variants) of chunks in which data are processed.

chunk_width : int, optional

Width (number of samples) to use when storing chunks in output.

log : file-like, optional

A file-like object (e.g., *sys.stderr*) to print progress information.

`allel.`**`vcf_to_dataframe`**(*input*, *fields=None*, *types=None*, *numbers=None*, *alt_number=3*, *fills=None*, *region=None*, *tabix='tabix'*, *transformers=None*, *buffer_size=16384*, *chunk_length=65536*, *log=None*)

Read data from a VCF file into a pandas DataFrame.

Parameters **input** : string

Path to VCF file on the local file system. May be uncompressed or gzip-compatible compressed file. May also be a file-like object (e.g., *io.BytesIO*).

fields : list of strings, optional

Fields to extract data for. Should be a list of strings, e.g., `['variants/CHROM', 'variants/POS', 'variants/DP', 'calldata/GT']`. If you are feeling lazy, you can drop the 'variants/' and 'calldata/' prefixes, in which case the fields will be matched against fields declared in the VCF header, with variants taking priority over calldata if a field with the same ID exists both in INFO and FORMAT headers. I.e., `['CHROM', 'POS', 'DP', 'GT']` will work, although watch out for fields like 'DP' which can be both INFO and FORMAT. For convenience, some special string values are also recognized. To extract all fields, provide just the string `'*'`. To extract all variants fields (including all INFO fields) provide `'variants/*'`. To extract all calldata fields (i.e., defined in FORMAT headers) provide `'calldata/*'`.

types : dict, optional

Overide data types. Should be a dictionary mapping field names to NumPy data types. E.g., providing the dictionary `{'variants/DP': 'i8', 'calldata/GQ': 'i2'}` will mean the 'variants/DP' field is stored in a 64-bit integer array, and the 'calldata/GQ' field is stored in a 16-bit integer array.

numbers : dict, optional

Override the expected number of values. Should be a dictionary mapping field names to integers. E.g., providing the dictionary `{'variants/ALT': 5, 'variants/AC': 5, 'calldata/HQ': 2}` will mean that, for each variant, 5 values are stored for the 'variants/ALT' field, 5 values are stored for the 'variants/AC' field, and for each sample, 2 values are stored for the 'calldata/HQ' field.

alt_number : int, optional

Assume this number of alternate alleles and set expected number of values accordingly for any field declared with number 'A' or 'R' in the VCF meta-information.

fills : dict, optional

Override the fill value used for empty values. Should be a dictionary mapping field names to fill values.

region : string, optional

Genomic region to extract variants for. If provided, should be a tabix-style region string, which can be either just a chromosome name (e.g., '2L'), or a chromosome name followed by 1-based beginning and end coordinates (e.g., '2L:100000-200000'). Note that only variants whose start position (POS) is within the requested range will be included. This is slightly different from the default tabix behaviour, where a variant (e.g., deletion) may be included if its position (POS) occurs before the requested region but its reference allele overlaps the region - such a variant will *not* be included in the data returned by this function.

tabix : string, optional

Name or path to tabix executable. Only required if *region* is given. Setting *tabix* to *None* will cause a fall-back to scanning through the VCF file from the beginning, which may be much slower than tabix but the only option if tabix is not available on your system and/or the VCF file has not been tabix-indexed.

transformers : list of transformer objects, optional

Transformers for post-processing data. If provided, should be a list of Transformer objects, each of which must implement a "transform()" method that accepts a dict containing the chunk of data to be transformed. See also the `ANNTransformer` class which implements post-processing of data from SNPEFF.

buffer_size : int, optional

Size in bytes of the I/O buffer used when reading data from the underlying file or tabix

stream.

chunk_length : int, optional

Length (number of variants) of chunks in which data are processed.

log : file-like, optional

A file-like object (e.g., *sys.stderr*) to print progress information.

Returns **df** : pandas.DataFrame

`allel.`**`vcf_to_csv`**(*input*, *output*, *fields=None*, *types=None*, *numbers=None*, *alt_number=3*, *fills=None*, *region=None*, *tabix='tabix'*, *transformers=None*, *buffer_size=16384*, *chunk_length=65536*, *log=None*, ***kwargs*)

Read data from a VCF file and write out to a comma-separated values (CSV) file.

Parameters **input** : string

Path to VCF file on the local file system. May be uncompressed or gzip-compatible compressed file. May also be a file-like object (e.g., *io.BytesIO*).

output : string

File-system path to write output to.

fields : list of strings, optional

Fields to extract data for. Should be a list of strings, e.g., `['variants/CHROM', 'variants/POS', 'variants/DP', 'calldata/GT']`. If you are feeling lazy, you can drop the 'variants/' and 'calldata/' prefixes, in which case the fields will be matched against fields declared in the VCF header, with variants taking priority over calldata if a field with the same ID exists both in INFO and FORMAT headers. I.e., `['CHROM', 'POS', 'DP', 'GT']` will work, although watch out for fields like 'DP' which can be both INFO and FORMAT. For convenience, some special string values are also recognized. To extract all fields, provide just the string `'*'`. To extract all variants fields (including all INFO fields) provide `'variants/*'`. To extract all calldata fields (i.e., defined in FORMAT headers) provide `'calldata/*'`.

types : dict, optional

Overide data types. Should be a dictionary mapping field names to NumPy data types. E.g., providing the dictionary `{'variants/DP': 'i8', 'calldata/GQ': 'i2'}` will mean the 'variants/DP' field is stored in a 64-bit integer array, and the 'calldata/GQ' field is stored in a 16-bit integer array.

numbers : dict, optional

Override the expected number of values. Should be a dictionary mapping field names to integers. E.g., providing the dictionary `{'variants/ALT': 5, 'variants/AC': 5, 'calldata/HQ': 2}` will mean that, for each variant, 5 values are stored for the 'variants/ALT' field, 5 values are stored for the 'variants/AC' field, and for each sample, 2 values are stored for the 'calldata/HQ' field.

alt_number : int, optional

Assume this number of alternate alleles and set expected number of values accordingly for any field declared with number 'A' or 'R' in the VCF meta-information.

fills : dict, optional

Override the fill value used for empty values. Should be a dictionary mapping field names to fill values.

region : string, optional

Genomic region to extract variants for. If provided, should be a tabix-style region string, which can be either just a chromosome name (e.g., '2L'), or a chromosome name followed by 1-based beginning and end coordinates (e.g., '2L:100000-200000'). Note that only variants whose start position (POS) is within the requested range will be included. This is slightly different from the default tabix behaviour, where a variant (e.g., deletion) may be included if its position (POS) occurs before the requested region but its reference allele overlaps the region - such a variant will *not* be included in the data returned by this function.

tabix : string, optional

Name or path to tabix executable. Only required if *region* is given. Setting *tabix* to *None* will cause a fall-back to scanning through the VCF file from the beginning, which may be much slower than tabix but the only option if tabix is not available on your system and/or the VCF file has not been tabix-indexed.

transformers : list of transformer objects, optional

Transformers for post-processing data. If provided, should be a list of Transformer objects, each of which must implement a "transform()" method that accepts a dict containing the chunk of data to be transformed. See also the `ANNTransformer` class which implements post-processing of data from SNPEFF.

buffer_size : int, optional

Size in bytes of the I/O buffer used when reading data from the underlying file or tabix

stream.

chunk_length : int, optional

Length (number of variants) of chunks in which data are processed.

log : file-like, optional

A file-like object (e.g., *sys.stderr*) to print progress information.

kwargs : keyword arguments

All remaining keyword arguments are passed through to pandas.DataFrame.to_csv(). E.g., to write a tab-delimited file, provide *sep=' '*.

allel.**vcf_to_recarray**(*input*, *fields=None*, *types=None*, *numbers=None*, *alt_number=3*, *fills=None*, *region=None*, *tabix='tabix'*, *transformers=None*, *buffer_size=16384*, *chunk_length=65536*, *log=None*)

Read data from a VCF file into a NumPy recarray.

Parameters **input** : string

Path to VCF file on the local file system. May be uncompressed or gzip-compatible compressed file. May also be a file-like object (e.g., *io.BytesIO*).

fields : list of strings, optional

Fields to extract data for. Should be a list of strings, e.g., `['variants/CHROM', 'variants/POS', 'variants/DP', 'calldata/GT']`. If you are feeling lazy, you can drop the 'variants/' and 'calldata/' prefixes, in which case the fields will be matched against fields declared in the VCF header, with variants taking priority over calldata if a field with the same ID exists both in INFO and FORMAT headers. I.e., `['CHROM', 'POS', 'DP', 'GT']` will work, although watch out for fields like 'DP' which can be both INFO and FORMAT. For convenience, some special string values are also recognized. To extract all fields, provide just the string `'*'`. To extract all variants fields (including all INFO fields) provide `'variants/*'`. To extract all calldata fields (i.e., defined in FORMAT headers) provide `'calldata/*'`.

types : dict, optional

Overide data types. Should be a dictionary mapping field names to NumPy data types. E.g., providing the dictionary `{'variants/DP': 'i8', 'calldata/GQ': 'i2'}` will mean the 'variants/DP' field is stored in a 64-bit integer array, and the 'calldata/GQ' field is stored in a 16-bit integer array.

numbers : dict, optional

Override the expected number of values. Should be a dictionary mapping field names to integers. E.g., providing the dictionary `{'variants/ALT': 5, 'variants/AC': 5, 'calldata/HQ': 2}` will mean that, for each variant, 5 values are stored for the 'variants/ALT' field, 5 values are stored for the 'variants/AC' field, and for each sample, 2 values are stored for the 'calldata/HQ' field.

alt_number : int, optional

Assume this number of alternate alleles and set expected number of values accordingly for any field declared with number 'A' or 'R' in the VCF meta-information.

fills : dict, optional

Override the fill value used for empty values. Should be a dictionary mapping field names to fill values.

region : string, optional

Genomic region to extract variants for. If provided, should be a tabix-style region string, which can be either just a chromosome name (e.g., '2L'), or a chromosome name followed by 1-based beginning and end coordinates (e.g., '2L:100000-200000'). Note that only variants whose start position (POS) is within the requested range will be included. This is slightly different from the default tabix behaviour, where a variant (e.g., deletion) may be included if its position (POS) occurs before the requested region but its reference allele overlaps the region - such a variant will *not* be included in the data returned by this function.

tabix : string, optional

Name or path to tabix executable. Only required if *region* is given. Setting *tabix* to *None* will cause a fall-back to scanning through the VCF file from the beginning, which may be much slower than tabix but the only option if tabix is not available on your system and/or the VCF file has not been tabix-indexed.

transformers : list of transformer objects, optional

Transformers for post-processing data. If provided, should be a list of Transformer objects, each of which must implement a "transform()" method that accepts a dict containing the chunk of data to be transformed. See also the `ANNTransformer` class which implements post-processing of data from SNPEFF.

buffer_size : int, optional

Size in bytes of the I/O buffer used when reading data from the underlying file or tabix

stream.

chunk_length : int, optional

Length (number of variants) of chunks in which data are processed.

log : file-like, optional

A file-like object (e.g., *sys.stderr*) to print progress information.

Returns **ra** : np.rec.array

`allel.`**`iter_vcf_chunks`**(*input*, *fields=None*, *types=None*, *numbers=None*, *alt_number=3*, *fills=None*, *region=None*, *tabix='tabix'*, *samples=None*, *transformers=None*, *buffer_size=16384*, *chunk_length=65536*)

Iterate over chunks of data from a VCF file as NumPy arrays.

Parameters **input** : string

Path to VCF file on the local file system. May be uncompressed or gzip-compatible compressed file. May also be a file-like object (e.g., *io.BytesIO*).

fields : list of strings, optional

Fields to extract data for. Should be a list of strings, e.g., `['variants/CHROM', 'variants/POS', 'variants/DP', 'calldata/GT']`. If you are feeling lazy, you can drop the 'variants/' and 'calldata/' prefixes, in which case the fields will be matched against fields declared in the VCF header, with variants taking priority over calldata if a field with the same ID exists both in INFO and FORMAT headers. I.e., `['CHROM', 'POS', 'DP', 'GT']` will work, although watch out for fields like 'DP' which can be both INFO and FORMAT. For convenience, some special string values are also recognized. To extract all fields, provide just the string `'*'`. To extract

all variants fields (including all INFO fields) provide `'variants/*'`. To extract all calldata fields (i.e., defined in FORMAT headers) provide `'calldata/*'`.

types : dict, optional

Overide data types. Should be a dictionary mapping field names to NumPy data types. E.g., providing the dictionary `{'variants/DP': 'i8', 'calldata/GQ': 'i2'}` will mean the 'variants/DP' field is stored in a 64-bit integer array, and the 'calldata/GQ' field is stored in a 16-bit integer array.

numbers : dict, optional

Override the expected number of values. Should be a dictionary mapping field names to integers. E.g., providing the dictionary `{'variants/ALT': 5, 'variants/AC': 5, 'calldata/HQ': 2}` will mean that, for each variant, 5 values are stored for the 'variants/ALT' field, 5 values are stored for the 'variants/AC' field, and for each sample, 2 values are stored for the 'calldata/HQ' field.

alt_number : int, optional

Assume this number of alternate alleles and set expected number of values accordingly for any field declared with number 'A' or 'R' in the VCF meta-information.

fills : dict, optional

Override the fill value used for empty values. Should be a dictionary mapping field names to fill values.

region : string, optional

Genomic region to extract variants for. If provided, should be a tabix-style region string, which can be either just a chromosome name (e.g., '2L'), or a chromosome name followed by 1-based beginning and end coordinates (e.g., '2L:100000-200000'). Note that only variants whose start position (POS) is within the requested range will be included. This is slightly different from the default tabix behaviour, where a variant (e.g., deletion) may be included if its position (POS) occurs before the requested region but its reference allele overlaps the region - such a variant will *not* be included in the data returned by this function.

tabix : string, optional

Name or path to tabix executable. Only required if *region* is given. Setting *tabix* to *None* will cause a fall-back to scanning through the VCF file from the beginning, which may be much slower than tabix but the only option if tabix is not available on your system and/or the VCF file has not been tabix-indexed.

samples : list of strings

Selection of samples to extract calldata for. If provided, should be a list of strings giving sample identifiers. May also be a list of integers giving indices of selected samples.

transformers : list of transformer objects, optional

Transformers for post-processing data. If provided, should be a list of Transformer objects, each of which must implement a "transform()" method that accepts a dict containing the chunk of data to be transformed. See also the `ANNTransformer` class which implements post-processing of data from SNPEFF.

buffer_size : int, optional

Size in bytes of the I/O buffer used when reading data from the underlying file or tabix

stream.

chunk_length : int, optional

Length (number of variants) of chunks in which data are processed.

Returns **fields** : list of strings

Normalised names of fields that will be extracted.

samples : ndarray

Samples for which data will be extracted.

headers : VCFHeaders

Tuple of metadata extracted from VCF headers.

it : iterator

Chunk iterator.

class allel.**ANNTransformer**

allel.**write_vcf**(*path*, *callset*, *rename=None*, *number=None*, *description=None*, *fill=None*, *write_header=True*)

Preliminary support for writing a VCF file. Currently does not support sample data. Needs further work.

4.3.2 GFF3

allel.**gff3_to_dataframe**(*path*, *attributes=None*, *region=None*, *score_fill=-1*, *phase_fill=-1*, *attributes_fill='.'*, *tabix='tabix'*, ***kwargs*)

Load data from a GFF3 into a pandas DataFrame.

Parameters **path** : string

Path to input file.

attributes : list of strings, optional

List of columns to extract from the "attributes" field.

region : string, optional

Genome region to extract. If given, file must be position sorted, bgzipped and tabix indexed. Tabix must also be installed and on the system path.

score_fill : int, optional

Value to use where score field has a missing value.

phase_fill : int, optional

Value to use where phase field has a missing value.

attributes_fill : object or list of objects, optional

Value(s) to use where attribute field(s) have a missing value.

tabix : string, optional

Tabix command.

Returns pandas.DataFrame

allel.**gff3_to_recarray**(*path*, *attributes=None*, *region=None*, *score_fill=-1*, *phase_fill=-1*, *attributes_fill='.'*, *tabix='tabix'*, *dtype=None*)

Load data from a GFF3 into a NumPy recarray.

Parameters **path** : string

Path to input file.

attributes : list of strings, optional

List of columns to extract from the "attributes" field.

region : string, optional

Genome region to extract. If given, file must be position sorted, bgzipped and tabix indexed. Tabix must also be installed and on the system path.

score_fill : int, optional

Value to use where score field has a missing value.

phase_fill : int, optional

Value to use where phase field has a missing value.

attributes_fill : object or list of objects, optional

Value(s) to use where attribute field(s) have a missing value.

tabix : string, optional

Tabix command.

dtype : dtype, optional

Override dtype.

Returns np.recarray

allel.**iter_gff3**(*path*, *attributes=None*, *region=None*, *score_fill=-1*, *phase_fill=-1*, *attributes_fill='.'*, *tabix='tabix'*)

Iterate over records in a GFF3 file.

Parameters **path** : string

Path to input file.

attributes : list of strings, optional

List of columns to extract from the "attributes" field.

region : string, optional

Genome region to extract. If given, file must be position sorted, bgzipped and tabix indexed. Tabix must also be installed and on the system path.

score_fill : int, optional

Value to use where score field has a missing value.

phase_fill : int, optional

Value to use where phase field has a missing value.

attributes_fill : object or list of objects, optional

Value(s) to use where attribute field(s) have a missing value.

tabix : string

Tabix command.

Returns Iterator

4.3.3 Fasta

allel.**write_fasta**(*path*, *sequences*, *names*, *mode='w'*, *width=80*)
Write nucleotide sequences stored as numpy arrays to a FASTA file.

Parameters **path** : string

File path.

sequences : sequence of arrays

One or more ndarrays of dtype 'S1' containing the sequences.

names : sequence of strings

Names of the sequences.

mode : string, optional

Use 'a' to append to an existing file.

width : int, optional

Maximum line width.

4.4 Chunked storage utilities

This module provides an abstraction layer over generic chunked array storage libraries. Currently HDF5 (via h5py), bcolz and zarr are supported storage layers.

Different storage configurations can be used with the functions and classes defined below. Wherever a function or method takes a *storage* keyword argument, the value of the argument will determine the storage used for the output.

If *storage* is a string, it will be used to look up one of several predefined storage configurations via the storage registry, which is a dictionary located at *allel.chunked.storage_registry*. The default storage can be changed globally by setting the value of the 'default' key in the storage registry.

Alternatively, *storage* may be an instance of one of the storage classes defined below, e.g., *allel.chunked.storage_bcolz.BcolzMemStorage* or *allel.chunked.storage_hdf5.HDF5TmpStorage*, which allows custom configuration of storage parameters such as compression type and level.

For example:

```
>>> from allel import chunked
>>> import numpy as np
>>> a = np.arange(10000000)
>>> chunked.copy(a)
Array((10000000,), int64, chunks=(39063,), order=C)
  ...
>>> chunked.copy(a, storage='bcolzmem')
carray((10000000,), int64)
  ...
>>> chunked.copy(a, storage='bcolztmp')
carray((10000000,), int64)
  ...
>>> chunked.copy(a, storage='zarrmem')
Array((10000000,), int64, chunks=(39063,), order=C)
  ...
>>> chunked.copy(a, storage='zarrtmp')
Array((10000000,), int64, chunks=(39063,), order=C)
  ...
```

```
>>> chunked.copy(a, storage=chunked.BcolzStorage(cparams=bcolz.cparams(cname='lz4')))
carray((10000000,), int64)
  ...
>>> chunked.copy(a, storage='hdf5mem_zlib1')
<HDF5 dataset "data": shape (10000000,), type "<i8">
>>> chunked.copy(a, storage='hdf5tmp_zlib1')
<HDF5 dataset "data": shape (10000000,), type "<i8">
>>> import h5py
>>> h5f = h5py.File('example.h5', mode='w')
>>> h5g = h5f.create_group('test')
>>> chunked.copy(a, storage='hdf5', group=h5g, name='data')
<HDF5 dataset "data": shape (10000000,), type "<i8">
>>> h5f['test/data']
<HDF5 dataset "data": shape (10000000,), type "<i8">
```

4.4.1 Storage

bcolz

class allel.chunked.storage_bcolz.**BcolzStorage**(***kwargs*)
Storage layer using bcolz carray and ctable.

class allel.chunked.storage_bcolz.**BcolzMemStorage**(***kwargs*)

class allel.chunked.storage_bcolz.**BcolzTmpStorage**(***kwargs*)

allel.chunked.storage_bcolz.**bcolz_storage = 'bcolz'**
Storage layer using bcolz carray and ctable.

allel.chunked.storage_bcolz.**bcolzmem_storage = 'bcolzmem'**

allel.chunked.storage_bcolz.**bcolztmp_storage = 'bcolztmp'**

allel.chunked.storage_bcolz.**bcolz_zlib1_storage = 'bcolz_zlib1'**
Storage layer using bcolz carray and ctable.

allel.chunked.storage_bcolz.**bcolzmem_zlib1_storage = 'bcolzmem_zlib1'**

allel.chunked.storage_bcolz.**bcolztmp_zlib1_storage = 'bcolztmp_zlib1'**

HDF5 (h5py)

class allel.chunked.storage_hdf5.**HDF5Storage**(***kwargs*)
Storage layer using HDF5 dataset and group.

class allel.chunked.storage_hdf5.**HDF5MemStorage**(***kwargs*)

class allel.chunked.storage_hdf5.**HDF5TmpStorage**(***kwargs*)

allel.chunked.storage_hdf5.**hdf5_storage = 'hdf5'**
Storage layer using HDF5 dataset and group.

allel.chunked.storage_hdf5.**hdf5mem_storage = 'hdf5mem'**

allel.chunked.storage_hdf5.**hdf5tmp_storage = 'hdf5tmp'**

allel.chunked.storage_hdf5.**hdf5_zlib1_storage = 'hdf5_zlib1'**
Storage layer using HDF5 dataset and group.

allel.chunked.storage_hdf5.**hdf5mem_zlib1_storage = 'hdf5mem_zlib1'**

allel.chunked.storage_hdf5.**hdf5tmp_zlib1_storage = 'hdf5tmp_zlib1'**

allel.chunked.storage_hdf5.**hdf5_lzf_storage = 'hdf5_lzf'**
Storage layer using HDF5 dataset and group.

allel.chunked.storage_hdf5.**hdf5mem_lzf_storage = 'hdf5mem_lzf'**

allel.chunked.storage_hdf5.**hdf5tmp_lzf_storage = 'hdf5tmp_lzf'**

allel.chunked.storage_hdf5.**h5fmem**(***kwargs*)
Create an in-memory HDF5 file.

allel.chunked.storage_hdf5.**h5ftmp**(***kwargs*)
Create an HDF5 file backed by a temporary file.

4.4.2 Functions

allel.chunked.core.**store**(*data*, *arr*, *start=0*, *stop=None*, *offset=0*, *blen=None*)
Copy *data* block-wise into *arr*.

allel.chunked.core.**copy**(*data*, *start=0*, *stop=None*, *blen=None*, *storage=None*, *create='array'*, ***kwargs*)
Copy *data* block-wise into a new array.

allel.chunked.core.**map_blocks**(*data*, *f*, *blen=None*, *storage=None*, *create='array'*, ***kwargs*)
Apply function *f* block-wise over *data*.

allel.chunked.core.**reduce_axis**(*data*, *reducer*, *block_reducer*, *mapper=None*, *axis=None*, *blen=None*, *storage=None*, *create='array'*, ***kwargs*)
Apply an operation to *data* that reduces over one or more axes.

allel.chunked.core.**amax**(*data*, *axis=None*, *mapper=None*, *blen=None*, *storage=None*, *create='array'*, ***kwargs*)
Compute the maximum value.

allel.chunked.core.**amin**(*data*, *axis=None*, *mapper=None*, *blen=None*, *storage=None*, *create='array'*, ***kwargs*)
Compute the minimum value.

allel.chunked.core.**asum**(*data*, *axis=None*, *mapper=None*, *blen=None*, *storage=None*, *create='array'*, ***kwargs*)
Compute the sum.

allel.chunked.core.**count_nonzero**(*data*, *mapper=None*, *blen=None*, *storage=None*, *create='array'*, ***kwargs*)
Count the number of non-zero elements.

allel.chunked.core.**compress**(*condition*, *data*, *axis=0*, *out=None*, *blen=None*, *storage=None*, *create='array'*, ***kwargs*)
Return selected slices of an array along given axis.

allel.chunked.core.**take**(*data*, *indices*, *axis=0*, *out=None*, *mode='raise'*, *blen=None*, *storage=None*, *create='array'*, ***kwargs*)
Take elements from an array along an axis.

allel.chunked.core.**subset**(*data*, *sel0=None*, *sel1=None*, *blen=None*, *storage=None*, *create='array'*, ***kwargs*)
Return selected rows and columns of an array.

allel.chunked.core.**concatenate**(*tup*, *axis=0*, *blen=None*, *storage=None*, *create='array'*, ***kwargs*)
Concatenate arrays.

allel.chunked.core.**binary_op**(*data*, *op*, *other*, *blen=None*, *storage=None*, *create='array'*, ***kwargs*)
Compute a binary operation block-wise over *data*.

allel.chunked.core.**copy_table**(*tbl*, *start=0*, *stop=None*, *blen=None*, *storage=None*, *create='table'*, ***kwargs*)
Copy *tbl* block-wise into a new table.

allel.chunked.core.**compress_table**(*condition*, *tbl*, *axis=None*, *out=None*, *blen=None*, *storage=None*, *create='table'*, ***kwargs*)
Return selected rows of a table.

allel.chunked.core.**take_table**(*tbl*, *indices*, *axis=None*, *out=None*, *mode='raise'*, *blen=None*, *storage=None*, *create='table'*, ***kwargs*)
Return selected rows of a table.

allel.chunked.core.**concatenate_table**(*tup*, *blen=None*, *storage=None*, *create='table'*, ***kwargs*)
Stack tables in sequence vertically (row-wise).

allel.chunked.core.**eval_table**(*tbl*, *expression*, *vm='python'*, *blen=None*, *storage=None*, *create='array'*, *vm_kwargs=None*, ***kwargs*)
Evaluate *expression* against columns of a table.

4.4.3 Classes

class allel.chunked.core.**ChunkedArrayWrapper**(*data*)
Wrapper class for chunked array-like data.

Parameters **data** : array_like

Data to be wrapped. May be a bcolz carray, h5py dataset, or anything providing a similar interface.

class allel.chunked.core.**ChunkedTableWrapper**(*data*, *names=None*)
Wrapper class for chunked table-like data.

Parameters **data: table_like**

Data to be wrapped. May be a tuple or list of columns (array-like), a dict mapping names to columns, a bcolz ctable, h5py group, numpy recarray, or anything providing a similar interface.

names : sequence of strings

Column names.

4.5 Miscellaneous utilities

allel.util.**hdf5_cache**(*filepath=None*, *parent=None*, *group=None*, *names=None*, *typed=False*, *hashed_key=False*, ***h5dcreate_kwargs*)
HDF5 cache decorator.

Parameters **filepath** : string, optional

Path to HDF5 file. If None a temporary file name will be used.

parent : string, optional

Path to group within HDF5 file to use as parent. If None the root group will be used.

group : string, optional

Path to group within HDF5 file, relative to parent, to use as container for cached data. If None the name of the wrapped function will be used.

names : sequence of strings, optional

Name(s) of dataset(s). If None, default names will be 'f00', 'f01', etc.

typed : bool, optional

If True, arguments of different types will be cached separately. For example, f(3.0) and f(3) will be treated as distinct calls with distinct results.

hashed_key : bool, optional

If False (default) the key will not be hashed, which makes for readable cache group names. If True the key will be hashed, however note that on Python >= 3.3 the hash value will not be the same between sessions unless the environment variable PYTHONHASHSEED has been set to the same value.

Returns **decorator** : function

Examples

Without any arguments, will cache using a temporary HDF5 file:

```
>>> import allel
>>> @allel.util.hdf5_cache()
... def foo(n):
...     print('executing foo')
...     return np.arange(n)
...
>>> foo(3)
executing foo
array([0, 1, 2])
>>> foo(3)
array([0, 1, 2])
>>> foo.cache_filepath
'/tmp/tmp_jwtwgjz'
```

Supports multiple return values, including scalars, e.g.:

```
>>> @allel.util.hdf5_cache()
... def bar(n):
...     print('executing bar')
...     a = np.arange(n)
...     return a, a**2, n**2
...
>>> bar(3)
executing bar
(array([0, 1, 2]), array([0, 1, 4]), 9)
>>> bar(3)
(array([0, 1, 2]), array([0, 1, 4]), 9)
```

Names can also be specified for the datasets, e.g.:

```
>>> @allel.util.hdf5_cache(names=['z', 'x', 'y'])
... def baz(n):
...     print('executing baz')
```

```
...     a = np.arange(n)
...     return a, a**2, n**2
...
>>> baz(3)
executing baz
(array([0, 1, 2]), array([0, 1, 4]), 9)
>>> baz(3)
(array([0, 1, 2]), array([0, 1, 4]), 9)
```

4.6 Release notes

4.6.1 v1.1.0

Reading Variant Call Format (VCF) files

This release includes new functions for extracting data from VCF files and loading into NumPy arrays, HDF5 files and other storage containers. These functions are backed by VCF parsing code implemented in Cython, so should be reasonably fast. This is new code so there may be bugs, please report any issues via GitHub.

For a tutorial and worked examples, see the following article: Extracting data from VCF.

For API documentation, see the following functions: *allel.read_vcf()*, *allel.vcf_to_npz()*, *allel.vcf_to_hdf5()*, *allel.vcf_to_zarr()*, *allel.vcf_to_dataframe()*, *allel.vcf_to_csv()*, *allel.vcf_to_recarray()*, *allel.iter_vcf_chunks()*.

Reading GFF3 files

Added convenience functions *allel.gff3_to_dataframe()* and *allel.gff3_to_recarray()*.

Maintenance work

- scikit-allel is now compatible with Dask versions 0.12 and later (#148#).
- Fixed issue within functions *allel.joint_sfs()* and *allel.joint_sfs_folded()* relating to data types (#144#).
- Fixed regression in functions *allel.ehh_decay()* and *allel.voight_painting()* following refactoring of array data structures in version 1.0.0 (#142#).
- HTML representations of arrays have been tweaked to look better in Jupyter notebooks (#141#).

End of support for Python 2

Important: This is the last version of scikit-allel that will support Python 2. The next version of scikit-allel will support Python versions 3.5 and later only.

4.6.2 v1.0.3

Fix test compatibility with numpy 1.10.

4.6.3 v1.0.2

Move cython function imports outside of functions to work around bug found when using scikit-allel with dask.

4.6.4 v1.0.1

Add missing test packages so full test suite can be run to verify install.

4.6.5 v1.0.0

This release includes some subtle but important changes to the architecture of the data structures modules (*allel.model.ndarray*, *allel.model.chunked*, *allel.model.dask*). These changes are mostly backwards-compatible but in some cases could break existing code, hence the major version number has been incremented. Also included in this release are some new functions related to Mendelian inheritance and calling runs of homozygosity, further details below.

Mendelian errors and phasing by transmission

This release includes a new *allel.stats.mendel* module with functions to help with analysis of related individuals. The function *allel.mendel_errors()* locates genotype calls within a trio or cross that are not consistent with Mendelian segregation of alleles. The function *allel.phase_by_transmission()* will resolve unphased diploid genotypes into phased haplotypes for a trio or cross using Mendelian transmission rules. The function *allel.paint_transmission()* can help with evaluating and visualizing the results of phasing a trio or cross.

Runs of homozygosity

A new *allel.roh_mhmm()* function provides support for locating long runs of homozygosity within a single sample. The function uses a multinomial hidden Markov model to predict runs of homozygosity based on the rate of heterozygosity over the genome. The function can also incorporate information about which positions in the genome are not accessible to variant calling and hence where there is no information about heterozygosity, to reduce false calling of ROH in regions where there is patchy data. We've run this on data from the Ag1000G project but have not performed a comprehensive evaluation with other species, feedback is very welcome.

Changes to data structures

The *allel.model.ndarray* module includes a new `allel.model.ndarray.GenotypeVector` class. This class represents an array of genotype calls for a single variant in multiple samples, or for a single sample at multiple variants. This class makes it easier, for example, to locate all variants which are heterozygous in a single sample.

Also in the same module are two new classes `allel.model.ndarray.GenotypeAlleleCountsArray` and `allel.model.ndarray.GenotypeAlleleCountsVector`. These classes provide support for an alternative encoding of genotype calls, where each call is stored as the counts of each allele observed. This allows encoding of genotype calls where samples may have different ploidy for a given chromosome (e.g., *Leishmania*) and/or where samples carry structural variation within some genome regions, altering copy number (and hence effective ploidy) with respect to the reference sequence.

There have also been architectural changes to all data structures modules. The most important change is that all classes in the *allel.model.ndarray* module now **wrap** numpy arrays and are no longer direct sub-classes of the numpy `numpy.ndarray` class. These classes still **behave** like numpy arrays in most respects, and so in most

cases this change should not impact existing code. If you need a plain numpy array for any reason you can always use `numpy.asarray()` or access the `.values` property, e.g.:

```
>>> import allel
>>> import numpy as np
>>> g = allel.GenotypeArray([[[0, 1], [0, 0]], [[0, 2], [1, 1]]])
>>> isinstance(g, np.ndarray)
False
>>> a = np.asarray(g)
>>> isinstance(a, np.ndarray)
True
>>> isinstance(g.values, np.ndarray)
True
```

This change was made because there are a number of complexities that arise when sub-classing class:*numpy.ndarray* and these were proving tricky to manage and maintain.

The *allel.model.chunked* and *allel.model.dask* modules also follow the same wrapper pattern. For the *allel.model.dask* module this means a change in the way that classes are instantiated. For example, to create a `allel.model.dask.GenotypeDaskArray`, pass the underlying data directly into the class constructor, e.g.:

```
>>> import allel
>>> import h5py
>>> h5f = h5py.File('callset.h5', mode='r')
>>> h5d = h5f['3R/calldata/genotype']
>>> genotypes = allel.GenotypeDaskArray(h5d)
```

If the underlying data is chunked then there is no need to specify the chunks manually when instantiating a dask array, the native chunk shape will be used.

Finally, the *allel.model.bcolz* module has been removed, use either the *allel.model.chunked* or *allel.model.dask* module instead.

4.6.6 v0.21.2

This release resolves compatibility issues with Zarr version 2.1.

4.6.7 v0.21.1

- Added parameter *min_maf* to *allel.ihs()* to skip IHS calculation for variants below a given minor allele frequency.
- Minor change to calculation of integrated haplotype homozygosity to enable values to be reported for first and last variants if *include_edges* is *True*.
- Minor change to *allel.standardize_by_allele_count()* to better handle missing values.

4.6.8 v0.21.0

In this release the implementations of *allel.ihs()* and *allel.xpehh()* selection statistics have been reworked to address a number of issues:

- Both functions can now integrate over either a genetic map (via the *map_pos* parameter) or a physical map.

- Both functions now accept *max_gap* and *gap_scale* parameters to perform adjustments to integrated haplotype homozygosity where there are large gaps between variants, following the standard approach. Alternatively, if a map of genome accessibility is available, it may be provided via the *is_accessible* parameter, in which case the distance between variants will be scaled by the fraction of accessible bases between them.
- Both functions are now faster and can make use of multiple threads to further accelerate computation.
- Several bugs in the previous implementations of these functions have been fixed (#91).
- New utility functions are provided for standardising selection scores, see `allel.standardize_by_allele_count()` (for use with IHS and NSL) and `allel.standardize()` (for use with XPEHH).

Other changes:

- Added functions `allel.moving_tajima_d()` and `allel.moving_delta_tajima_d()` (#81, #70).
- Added functions `allel.moving_weir_cockerham_fst()`, `allel.moving_hudson_fst()`, `allel.moving_patterson_fst()`.
- Added functions `allel.moving_patterson_f3()` and `allel.moving_patterson_d()`.
- Renamed "blockwise" to "average" in function names in `allel.stats.fst` and `allel.stats.admixture` for clarity.
- Added convenience methods `allel.AlleleCountsArray.is_biallelic()` and `allel.AlleleCountsArray.is_biallelic_01()` for locating biallelic variants.
- Added support for zarr in the `allel.chunked` module (#101).
- Changed HDF5 default chunked storage to use gzip level 1 compression instead of no compression (#100).
- Fixed bug in `allel.sequence_divergence()` (#75).
- Added workaround for chunked arrays if passed as arguments into numpy aggregation functions (#66).
- Protect against invalid coordinates when mapping from square to condensed coords (#83).
- Fixed bug in `allel.plot_sfs_folded()` and added docstrings for all plotting functions in `allel.stats.sf` (#80).
- Fixed bug related to taking views of genotype and haplotype arrays (#77).

4.6.9 v0.20.3

- Fixed a bug in the *count_alleles()* methods on genotype and haplotype array classes that manifested if the *max_allele* argument was provided (#59).
- Fixed a bug in Jupyter notebook *display* method for chunked tables (#57).
- Fixed a bug in site frequency spectrum scaling functions (#54).
- Changed behaviour of *subset* method on genotype and haplotype arrays to better infer argument types and handle None argument values (#55).
- Changed table *eval* and *query* methods to make python the default for expression evaluation, because it is more expressive than numexpr (#58).

4.6.10 v0.20.2

- Changed `allel.util.hdf5_cache()` to resolve issues with hashing and argument order (#51, #52).

4.6.11 v0.20.1

- Changed functions `allel.weir_cockerham_fst()` and `allel.locate_unlinked()` such that chunked implementations are now used by default, to avoid accidentally and unnecessarily loading very large arrays into memory (#50).

4.6.12 v0.20.0

- Added new `allel.model.dask` module, providing implementations of the genotype, haplotype and allele counts classes backed by dask.array (#32).
- Released the GIL where possible in Cython optimised functions (#43).
- Changed functions in `allel.stats.selection` that accept *min_ehh* argument, such that *min_ehh* = *None* should now be used to indicate that no minimum EHH threshold should be applied.

4.6.13 v0.19.0

The major change in v0.19.0 is the addition of the new `allel.model.chunked` module, which provides classes for variant call data backed by chunked array storage (#31). This is a generalisation of the previously available `allel.model.bcolz` to enable the use of both bcolz and HDF5 (via h5py) as backing storage. The `allel.model.bcolz` module is now deprecated but will be retained for backwargs compatibility until the next major release.

Other changes:

- Added function for computing the number of segregating sites by length (nSl), a summary statistic comparing haplotype homozygosity between different alleles (similar to IHS), see `allel.nsl()` (#40).
- Added functions for computing haplotype diversity, see `allel.haplotype_diversity()` and `allel.moving_haplotype_diversity()` (#29).
- Added function `allel.plot_moving_haplotype_frequencies()` for visualising haplotype frequency spectra in moving windows over the genome (#30).
- Added *vstack()* and *hstack()* methods to genotype and haplotype arrays to enable combining data from multiple arrays (#21).
- Added convenience function `allel.equally_accessible_windows()` (#16).
- Added methods *from_hdf5_group()* and *to_hdf5_group()* to `allel.model.ndarray.VariantTable` (#26).
- Added `allel.util.hdf5_cache()` utility function.
- Modified functions in the `allel.stats.selection` module that depend on calculation of integrated haplotype homozygosity to return NaN when haplotypes do not decay below a specified threshold (#39).
- Fixed missing return value in `allel.plot_voight_painting()` (#23).
- Fixed return type from array reshape() (#34).

Contributors: alimanfoo, hardingnj

4.6.14 v0.18.1

- Minor change to the Garud H statistics to avoid raising an exception when the number of distinct haplotypes is very low (#20).

4.6.15 v0.18.0

- Added functions for computing H statistics for detecting signatures of soft sweeps, see *allel.garud_h()*, *allel.moving_garud_h()*, *allel.plot_haplotype_frequencies()* (#19).
- Added function *allel.fig_voight_painting()* to paint both flanks either side of some variant under selection in a single figure (#17).
- Changed return values from *allel.voight_painting()* to also return the indices used for sorting haplotypes by prefix (#18).

4.6.16 v0.17.0

- Added new module for computing and plotting site frequency spectra, see *allel.stats.sf* (#12).
- All plotting functions have been moved into the appropriate stats module that they naturally correspond to. The `allel.plot` module is deprecated (#13).
- Improved performance of carray and ctable loading from HDF5 with a condition (#11).

4.6.17 v0.16.2

- Fixed behaviour of take() method on compressed arrays when indices are not in increasing order (#6).
- Minor change to scaler argument to PCA functions in *allel.stats.decomposition* to avoid confusion about when to fall back to default scaler (#7).

4.6.18 v0.16.1

- Added block-wise implementation to *allel.locate_unlinked()* so it can be used with compressed arrays as input.

4.6.19 v0.16.0

- Added new selection module with functions for haplotype-based analyses of recent selection, see *allel.stats.selection*.

4.6.20 v0.15.2

- Improved performance of `allel.model.bcolz.carray_block_compress()`, `allel.model.bcolz.ctable_block_compress()` and `allel.model.bcolz.carray_block_subset()` for very sparse selections.
- Fix bug in IPython HTML table captions.
- Fix bug in addcol() method on bcolz ctable wrappers.

4.6.21 v0.15.1

- Fix missing package in setup.py.

4.6.22 v0.15

- Added functions to estimate Fst with standard error via a block-jackknife: `allel.blockwise_weir_cockerham_fst()`, `allel.blockwise_hudson_fst()`, `allel.blockwise_patterson_fst()`.
- Fixed a serious bug in *allel.weir_cockerham_fst()* related to incorrect estimation of heterozygosity, which manifested if the subpopulations being compared were not a partition of the total population (i.e., there were one or more samples in the genotype array that were not included in the subpopulations to compare).
- Added method `allel.AlleleCountsArray.max_allele()` to determine highest allele index for each variant.
- Changed first return value from admixture functions `allel.blockwise_patterson_f3()` and `allel.blockwise_patterson_d()` to return the estimator from the whole dataset.
- Added utility functions to the *allel.stats.distance* module for transforming coordinates between condensed and uncondensed forms of a distance matrix.
- Classes previously available from the *allel.model* and *allel.bcolz* modules are now aliased from the root *allel* module for convenience. These modules have been reorganised into an `allel.model` package with submodules *allel.model.ndarray* and `allel.model.bcolz`.
- All functions in the `allel.model.bcolz` module use cparams from input carray as default for output carray (convenient if you, e.g., want to use zlib level 1 throughout).
- All classes in the *allel.model.ndarray* and `allel.model.bcolz` modules have changed the default value for the *copy* keyword argument to *False*. This means that **not** copying the input data, just wrapping it, is now the default behaviour.
- Fixed bug in `GenotypeArray.to_gt()` where maximum allele index is zero.

4.6.23 v0.14

- Added a new module *allel.stats.admixture* with statistical tests for admixture between populations, implementing the f2, f3 and D statistics from Patterson (2012). Functions include `allel.blockwise_patterson_f3()` and `allel.blockwise_patterson_d()` which compute the f3 and D statistics respectively in blocks of a given number of variants and perform a block-jackknife to estimate the standard error.

4.6.24 v0.12

- Added functions for principal components analysis of genotype data. Functions in the new module *allel.stats.decomposition* include *allel.pca()* to perform a PCA via full singular value decomposition, and *allel.randomized_pca()* which uses an approximate truncated singular value decomposition to speed up computation. In tests with real data the randomized PCA is around 5 times faster and uses half as much memory as the conventional PCA, producing highly similar results.
- Added function *allel.pcoa()* for principal coordinate analysis (a.k.a. classical multi-dimensional scaling) of a distance matrix.
- Added new utility module *allel.stats.preprocessing* with classes for scaling genotype data prior to use as input for PCA or PCoA. By default the scaling (i.e., normalization) of Patterson (2006) is used with principal components analysis functions in the *allel.stats.decomposition* module. Scaling functions can improve the ability to resolve population structure via PCA or PCoA.

- Added method `allel.GenotypeArray.to_n_ref()`. Also added `dtype` argument to `allel.GenotypeArray.to_n_ref()` and `allel.GenotypeArray.to_n_alt()` methods to enable direct output as float arrays, which can be convenient if these arrays are then going to be scaled for use in PCA or PCoA.
- Added `allel.GenotypeArray.mask` property which can be set with a Boolean mask to filter genotype calls from genotype and allele counting operations. A similar property is available on the `allel.GenotypeCArray` class. Also added method `allel.GenotypeArray.fill_masked()` and similar method on the `allel.GenotypeCArray` class to fill masked genotype calls with a value (e.g., -1).

4.6.25 v0.11

- Added functions for calculating Watterson's theta (proportional to the number of segregating variants): *allel.watterson_theta()* for calculating over a given region, and *allel.windowed_watterson_theta()* for calculating in windows over a chromosome/contig.
- Added functions for calculating Tajima's D statistic (balance between nucleotide diversity and number of segregating sites): *allel.tajima_d()* for calculating over a given region and *allel.windowed_tajima_d()* for calculating in windows over a chromosome/contig.
- Added *allel.windowed_df()* for calculating the rate of fixed differences between two populations.
- Added function *allel.locate_fixed_differences()* for locating variants that are fixed for different alleles in two different populations.
- Added function *allel.locate_private_alleles()* for locating alleles and variants that are private to a single population.

4.6.26 v0.10

- Added functions implementing the Weir and Cockerham (1984) estimators for F-statistics: *allel.weir_cockerham_fst()* and *allel.windowed_weir_cockerham_fst()*.
- Added functions implementing the Hudson (1992) estimator for Fst: *allel.hudson_fst()* and *allel.windowed_hudson_fst()*.
- Added new module *allel.stats.ld* with functions for calculating linkage disequilibrium estimators, including *allel.rogers_huff_r()* for pairwise variant LD calculation, *allel.windowed_r_squared()* for windowed LD calculations, and *allel.locate_unlinked()* for locating variants in approximate linkage equilibrium.
- Added function *allel.plot_pairwise_ld()* for visualising a matrix of linkage disequilbrium values between pairs of variants.
- Added function *allel.create_allele_mapping()* for creating a mapping of alleles into a different index system, i.e., if you want 0 and 1 to represent something other than REF and ALT, e.g., ancestral and derived. Also added methods `allel.GenotypeArray.map_alleles()`, `allel.HaplotypeArray.map_alleles()` and `allel.AlleleCountsArray.map_alleles()` which will perform an allele transformation given an allele mapping.
- Added function *allel.plot_variant_locator()* ported across from anhima.
- Refactored the `allel.stats` module into a package with sub-modules for easier maintenance.

4.6.27 v0.9

- Added documentation for the functions `allel.bcolz.carray_from_hdf5()`, `allel.bcolz.carray_to_hdf5()`, `allel.bcolz.ctable_from_hdf5_group()`, `allel.bcolz.ctable_to_hdf5_group()`.
- Refactoring of internals within the `allel.bcolz` module.

4.6.28 v0.8

- Added *subpop* argument to `allel.GenotypeArray.count_alleles()` and `allel.HaplotypeArray.count_alleles()` to enable count alleles within a sub-population without subsetting the array.
- Added functions `allel.GenotypeArray.count_alleles_subpops()` and `allel.HaplotypeArray.count_alleles_subpops()` to enable counting alleles in multiple sub-populations in a single pass over the array, without sub-setting.
- Added classes `allel.model.FeatureTable` and `allel.bcolz.FeatureCTable` for storing and querying data on genomic features (genes, etc.), with functions for parsing from a GFF3 file.
- Added convenience function *allel.pairwise_dxy()* for computing a distance matrix using Dxy as the metric.

4.6.29 v0.7

- Added function *allel.write_fasta()* for writing a nucleotide sequence stored as a NumPy array out to a FASTA format file.

4.6.30 v0.6

- Added method `allel.VariantTable.to_vcf()` for writing a variant table to a VCF format file.

CHAPTER 5

Acknowledgments

Development of this package is supported by the MRC Centre for Genomics and Global Health.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index

D

E

F

G

H

I

J

L

M

N

P

Q

R

S

T

U

V

W

X