scikit-allel Documentation

Release 0.15.1

Alistair Miles

November 27, 2015

Contents

This package provides utilities for working with large scale genetic variation data using numpy, scipy and other established Python scientific libraries.

- GitHub repository: https://github.com/cggh/scikit-allel
- Documentation: http://scikit-allel.readthedocs.org/
- Download: https://pypi.python.org/pypi/scikit-allel

This package is in an early stage of development, if you have any questions please email Alistair Miles <alimanfoo@googlemail.com>.

CHAPTER 1

Installation

This package requires numpy, scipy, matplotlib, pandas, scikit-learn, h5py, numexpr, bcolz and petl. Install these dependencies first, then use pip to install scikit-allel:

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

CHAPTER 2

Contents

2.1 Data structures

2.1.1 In-memory data structures

This module defines NumPy array classes for variant call data.

Please note, functions and command line utilities for converting variant call data from the VCF file format into NumPy arrays and HDF5 files are available from the vcfnp package.

GenotypeArray

class `allel.model.ndarray.`**`GenotypeArray`**

Array of discrete genotype calls.

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

Genotype data.

****kwargs** : keyword arguments

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

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.

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.bcolz.GenotypeCArray` class, which provides an alternative implementation of this interface using compressed arrays.

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
```

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

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

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

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

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
```

n_variants
Number of variants (length of first array dimension).

n_samples
Number of samples (length of second array dimension).

ploidy
Sample ploidy (length of third array dimension).

mask
A boolean mask associated with this genotype array, 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.

Note also that the mask may not survive any slicing, indexing or other subsetting procedures (e.g., call to np.compress() or np.take()). I.e., the mask will have to be similarly indexed then reapplied. The only exceptions are simple slicing operations that preserve the dimensionality and ploidy of the array, and the subset() method, both of which **will** preserve the mask if present.

Examples

```
>>> import allel
>>> g = allel.GenotypeArray([[[0, 0], [0, 1]],
...                          [[0, 1], [1, 1]],
...                          [[0, 2], [-1, -1]]], dtype='i1')
>>> g.count_called()
5
>>> g.count_alleles()
AlleleCountsArray((3, 3), dtype=int32)
[[3 1 0]
 [1 3 0]
 [1 0 1]]
>>> mask = [[True, False], [False, True], [False, False]]
>>> g.mask = mask
>>> g.count_called()
3
>>> g.count_alleles()
AlleleCountsArray((3, 3), dtype=int32)
[[1 1 0]
 [1 1 0]
 [1 0 1]]
```

fill_masked(*value=-1*, *mask=None*, *copy=True*)
Fill masked genotype calls with a given value.

Parameters **value** : int, optional

The fill value.

mask : array_like, bool, shape (n_variants, n_samples), optional

A boolean array where True elements indicate genotype calls to be filled. If not provided, value of the *mask* property will be used.

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()
GenotypeArray((3, 2, 2), dtype=int8)
[[[-1 -1]
  [ 0  1]]
 [[ 0  1]
  [-1 -1]]
 [[ 0  2]
  [-1 -1]]]
```

subset(*variants=None*, *samples=None*)

Make a sub-selection of variants and/or samples.

Parameters **variants** : array_like

Boolean array or list of indices.

samples : array_like

Boolean array or list of indices.

Returns **out** : GenotypeArray

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(variants=[0, 1], samples=[0, 2])
GenotypeArray((2, 2, 2), dtype=int64)
[[[0 0]
  [1 1]]
 [[0 1]
  [1 2]]]
```

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)
```

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)
```

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)
```

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)
```

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)
```

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)
```

is_call(*call*)

Find 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)
```

count_called(*axis=None*)

count_missing(*axis=None*)

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

count_hom_ref(*axis=None*)

count_hom_alt(*axis=None*)

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

count_call(*call*, *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 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((3, 3), dtype=int32)
[[3 1 0]
```

```
 [1 2 1]
 [0 0 2]]
>>> g.count_alleles(max_allele=1)
AlleleCountsArray((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.

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

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((4, 2, 2), dtype=int8)
```

```
[[[ 1  1]
  [ 1  2]]
 [[ 2  1]
  [ 0  0]]
 [[ 1  0]
  [ 0  1]]
 [[ 1  1]
  [-1 -1]]]
```

to_haplotypes(*copy=False*)

Reshape a genotype array to view it as haplotypes by dropping the ploidy dimension.

Returns **h** : HaplotypeArray, shape (n_variants, n_samples * ploidy)

Haplotype array.

copy : bool, optional

If True, make a copy of the data.

Notes

If genotype calls are unphased, the haplotypes returned by this function will bear no resemblance to the true haplotypes.

Examples

```
>>> import allel
>>> g = allel.GenotypeArray([[[0, 0], [0, 1]],
...                          [[0, 1], [1, 1]],
...                          [[0, 2], [-1, -1]]])
>>> g.to_haplotypes()
HaplotypeArray((3, 4), dtype=int64)
[[ 0  0  0  1]
 [ 0  1  1  1]
 [ 0  2 -1 -1]]
```

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.

Returns **out** : ndarray, int, 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)
```

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.

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)
```

to_allele_counts (*alleles=None*)

Transform genotype calls into allele counts per call.

Parameters **alleles** : sequence of ints, optional

If not None, count only the given alleles. (By default, count all alleles.)

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()
array([[[2, 0, 0],
        [1, 1, 0]],
       [[1, 0, 1],
        [0, 2, 0]],
       [[0, 0, 2],
        [0, 0, 0]]], dtype=uint8)
>>> g.to_allele_counts(alleles=(0, 1))
array([[[2, 0],
        [1, 1]],
       [[1, 0],
        [0, 2]],
       [[0, 0],
        [0, 0]]], dtype=uint8)
```

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)
```

static **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((3, 2, 2), dtype=int8)
[[[ 0  0]
  [ 0  1]]
 [[ 0  2]
  [ 1  1]]
 [[ 2  2]
  [-1 -1]]]
```

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((4, 2, 2), dtype=int8)
[[[ 0  0]
  [ 0  0]]
 [[ 0  1]
  [ 0  1]]
 [[ 1  1]
  [ 0  0]]
 [[ 0  0]
  [-1 -1]]]
```

to_gt(*phased=False*, *max_allele=None*)

Convert genotype calls to VCF-style string representation.

Parameters **phased** : bool, optional

Determines separator.

max_allele : int, optional

Manually specify max allele index.

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

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]],
...                          [[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')
>>> g.to_gt(phased=True)
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')
```

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((4, 2), dtype=int64)
[[ 0  1]
 [ 0  1]
 [ 1  1]
 [ 2 -1]]
>>> 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((3, 2), dtype=int64)
[[ 0  0]
```

```
        [ 1  1]
        [ 2 -1]]
```

HaplotypeArray

class allel.model.ndarray.**HaplotypeArray**

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()`.

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.

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
```

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((3, 2, 2), dtype=int8)
[[[ 0  0]
  [ 0  1]]
 [[ 0  1]
  [ 1  1]]
 [[ 0  2]
  [-1 -1]]]
```

n_variants
Number of variants (length of first dimension).

n_haplotypes
Number of haplotypes (length of second dimension).

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

Parameters **variants** : array_like

Boolean array or list of indices.

haplotypes : array_like

Boolean array or list of indices.

Returns **out** : HaplotypeArray

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((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:

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((3, 4), dtype=int8)
[[ 1  1  1  2]
 [ 2  0  0  0]
 [ 2  0 -1 -1]]
```

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.

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((3, 2, 2), dtype=int8)
[[[ 0  0]
  [ 0  1]]
 [[ 0  1]
  [ 1  1]]
 [[ 0  2]
  [-1 -1]]]
```

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((4, 4), dtype=int8)
[[ 0  0  0  0]
 [ 0  1  0  1]
 [ 1  1  0  0]
 [ 0  0 -1 -1]]
```

AlleleCountsArray

class allel.model.ndarray.**AlleleCountsArray**

Array of allele counts.

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

Allele counts 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((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
>>> n = np.sum(ac, axis=1)
>>> n
array([4, 4, 2])
```

n_variants
Number of variants (length of first array dimension).

n_alleles
Number of alleles (length of second array dimension).

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)
```

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((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((4, 3), dtype=int64)
[[0 4 0]
 [1 3 0]
 [1 2 1]
 [2 0 0]]
```

VariantTable

class allel.model.ndarray.**VariantTable**
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 `np.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((5,), dtype=[('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((2,), dtype=[('CHROM', 'S4'), ('POS', '<u4'), ('DP', '...
[(b'chr2', 3, 78, 1.2, array([5, 6])) (b'chr2', 9, 22, 4.4, array([...
```

Use the index to query variants:

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

n_variants
Number of variants (length of first dimension).

names
Column names.

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

Parameters **expression** : string

Expression to evaluate.

vm : {'numexpr', 'python'}

Virtual machine to use.

Returns **result** : ndarray

Examples

```
>>> 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)
>>> 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], dtype=int64)
```

query(*expression*, *vm='numexpr'*)
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** : VariantTable

Examples

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

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.

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]
>>> id = ['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, id, 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.model.ndarray.`**`FeatureTable`**

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

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

Variant records.

index : pair or triplet of strings, optional

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

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

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

`n_features`

Number of features (length of first dimension).

`names`

Column names.

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

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='numexpr'*)

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** : FeatureTable

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 : object, optional

Value to use where score field has a missing value.

phase_fill : object, 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.model.ndarray.**SortedIndex**

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()`.

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, 14, 15, 42, 42, 77], dtype='i4')
>>> idx.dtype
dtype('int32')
>>> idx.ndim
1
>>> idx.shape
(7,)
>>> idx.is_unique
False
```

is_unique
True if no duplicate entries.

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

Parameters **key** : int

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, int

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((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((2,), dtype=int64)
[ 6 20]
>>> idx2[loc2]
SortedIndex((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((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((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((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((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((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((3,), dtype=int64)
[ 6 11 35]
```

UniqueIndex

class allel.model.ndarray.**UniqueIndex**

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()`.

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.dtype
dtype('<U1')
>>> 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'])
>>> idx2 = allel.UniqueIndex(['X', 'F', 'G', 'C', 'Z'])
>>> 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((2,), dtype=<U1)
['C' 'F']
>>> idx2[loc2]
UniqueIndex((2,), dtype=<U1)
['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'])
>>> idx2 = allel.UniqueIndex(['X', 'F', 'G', 'C', 'Z'])
>>> idx1.intersect(idx2)
UniqueIndex((2,), dtype=<U1)
['C' 'F']
```

```
>>> idx2.intersect(idx1)
UniqueIndex((2,), dtype=<U1)
['F' 'C']
```

SortedMultiIndex

class allel.model.ndarray.**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.

Examples

```
>>> import allel
>>> chrom = ['chr1', 'chr1', 'chr2', 'chr2', 'chr2', 'chr3']
>>> pos = [1, 4, 2, 5, 5, 3]
>>> idx = allel.SortedMultiIndex(chrom, pos)
>>> 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(*k1*, *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)
```

Utility functions

allel.model.ndarray.**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.

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.model.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 = allel.model.create_allele_mapping(ref, alt, alleles)
>>> mapping
array([[ 0,  1, -1],
       [ 0, -1,  1],
       [-1,  0, -1]], dtype=int8)
```

allel.model.ndarray.**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.model.locate_fixed_differences(ac1, ac2)
>>> loc_df
array([ True, False, False,  True,  True], dtype=bool)
```

allel.model.ndarray.**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.model.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.model.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)
```

2.1.2 Compressed arrays (bcolz)

This module provides alternative implementations of array interfaces defined in the `allel.model` module, using bcolz compressed arrays (`bcolz.carray`) instead of numpy arrays for data storage. Compressed arrays can use either main memory or be stored on disk. In either case, the use of compressed arrays enables analysis of data that are too large to fit uncompressed into main memory.

GenotypeCArray

class allel.model.bcolz.**GenotypeCArray**(*data=None*, *copy=False*, ***kwargs*)

Alternative implementation of the allel.model.GenotypeArray interface, using a bcolz.carray as the backing store.

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

Data to initialise the array with. May be a bcolz carray, which will not be copied if copy=False. May also be None, in which case rootdir must be provided (disk-based array).

copy : bool, optional

If True, copy the input data into a new bcolz carray.

****kwargs** : keyword arguments

Passed through to the bcolz carray constructor.

Examples

Instantiate a compressed genotype array from existing data:

```
>>> import allel
>>> g = allel.GenotypeCArray([[[0, 0], [0, 1]],
...                           [[0, 1], [1, 1]],
...                           [[0, 2], [-1, -1]]], dtype='i1')
>>> g
GenotypeCArray((3, 2, 2), int8)
  nbytes: 12; cbytes: 16.00 KB; ratio: 0.00
  cparams := cparams(clevel=5, shuffle=True, cname='blosclz')
[[[ 0  0]
  [ 0  1]]
 [[ 0  1]
  [ 1  1]]
 [[ 0  2]
  [-1 -1]]]
```

Obtain a numpy ndarray from a compressed array by slicing:

```
>>> g[:]
GenotypeArray((3, 2, 2), dtype=int8)
[[[ 0  0]
  [ 0  1]]
 [[ 0  1]
  [ 1  1]]
 [[ 0  2]
  [-1 -1]]]
```

Build incrementally:

```
>>> import bcolz
>>> data = bcolz.zeros((0, 2, 2), dtype='i1')
>>> data.append([[0, 0], [0, 1]])
>>> data.append([[0, 1], [1, 1]])
>>> data.append([[0, 2], [-1, -1]])
>>> g = allel.GenotypeCArray(data)
>>> g
GenotypeCArray((3, 2, 2), int8)
```

```
  nbytes: 12; cbytes: 16.00 KB; ratio: 0.00
  cparams := cparams(clevel=5, shuffle=True, cname='blosclz')
[[[ 0  0]
  [ 0  1]]
 [[ 0  1]
  [ 1  1]]
 [[ 0  2]
  [-1 -1]]]
```

Load from HDF5:

```
>>> import h5py
>>> with h5py.File('example.h5', mode='w') as h5f:
...     h5f.create_dataset('genotype',
...                        data=[[[0, 0], [0, 1]],
...                              [[0, 1], [1, 1]],
...                              [[0, 2], [-1, -1]]],
...                        dtype='i1',
...                        chunks=(2, 2, 2))
...
<HDF5 dataset "genotype": shape (3, 2, 2), type "|i1">
>>> g = allel.GenotypeCArray.from_hdf5('example.h5', 'genotype')
>>> g
GenotypeCArray((3, 2, 2), int8)
  nbytes: 12; cbytes: 16.00 KB; ratio: 0.00
  cparams := cparams(clevel=5, shuffle=True, cname='blosclz')
[[[ 0  0]
  [ 0  1]]
 [[ 0  1]
  [ 1  1]]
 [[ 0  2]
  [-1 -1]]]
```

Note that methods of this class will return bcolz carrays rather than numpy ndarrays where possible. E.g.:

```
>>> g.take([0, 2], axis=0)
GenotypeCArray((2, 2, 2), int8)
  nbytes: 8; cbytes: 16.00 KB; ratio: 0.00
  cparams := cparams(clevel=5, shuffle=True, cname='blosclz')
[[[ 0  0]
  [ 0  1]]
 [[ 0  2]
  [-1 -1]]]
>>> g.is_called()
CArrayWrapper((3, 2), bool)
  nbytes: 6; cbytes: 16.00 KB; ratio: 0.00
  cparams := cparams(clevel=5, shuffle=True, cname='blosclz')
[[ True  True]
 [ True  True]
 [ True False]]
>>> g.to_haplotypes()
HaplotypeCArray((3, 4), int8)
  nbytes: 12; cbytes: 16.00 KB; ratio: 0.00
  cparams := cparams(clevel=5, shuffle=True, cname='blosclz')
[[ 0  0  0  1]
 [ 0  1  1  1]
 [ 0  2 -1 -1]]
>>> g.count_alleles()
AlleleCountsCArray((3, 3), int32)
```

```
      nbytes: 36; cbytes: 16.00 KB; ratio: 0.00
      cparams := cparams(clevel=5, shuffle=True, cname='blosclz')
    [[3 1 0]
     [1 3 0]
     [1 0 1]]
```

HaplotypeCArray

class allel.model.bcolz.**HaplotypeCArray**(*data=None*, *copy=False*, ***kwargs*)

Alternative implementation of the allel.model.HaplotypeArray interface, using a bcolz.carray as the backing store.

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

Data to initialise the array with. May be a bcolz carray, which will not be copied if copy=False. May also be None, in which case rootdir must be provided (disk-based array).

copy : bool, optional

If True, copy the input data into a new bcolz carray.

****kwargs** : keyword arguments

Passed through to the bcolz carray constructor.

AlleleCountsCArray

class allel.model.bcolz.**AlleleCountsCArray**(*data=None*, *copy=False*, ***kwargs*)

Alternative implementation of the allel.model.AlleleCountsArray interface, using a bcolz.carray as the backing store.

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

Data to initialise the array with. May be a bcolz carray, which will not be copied if copy=False. May also be None, in which case rootdir must be provided (disk-based array).

copy : bool, optional

If True, copy the input data into a new bcolz carray.

****kwargs** : keyword arguments

Passed through to the bcolz carray constructor.

VariantCTable

class allel.model.bcolz.**VariantCTable**(*data=None*, *copy=False*, *index=None*, ***kwargs*)

Alternative implementation of the allel.model.VariantTable interface, using a bcolz.ctable as the backing store.

Parameters **data** : tuple or list of column objects, optional

The list of column data to build the ctable object. This can also be a pure NumPy structured array. May also be a bcolz ctable, which will not be copied if copy=False. May also be None, in which case rootdir must be provided (disk-based array).

copy : bool, optional

If True, copy the input data into a new bcolz ctable.

index : string or pair of strings, optional

If a single string, name of column to use for a sorted index. If a pair of strings, name of columns to use for a sorted multi-index.

****kwargs** : keyword arguments

Passed through to the bcolz ctable constructor.

Examples

Instantiate from existing data:

```
>>> import allel
>>> 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)]
>>> vt = allel.bcolz.VariantCTable([chrom, pos, dp, qd, ac],
...                                names=['CHROM', 'POS', 'DP', 'QD', 'AC'],
...                                index=('CHROM', 'POS'))
>>> vt
VariantCTable((5,), [('CHROM', 'S4'), ('POS', '<i8'), ('DP', '<i8'), ('QD', '<f8'), ('AC', '<i8'
  nbytes: 220; cbytes: 80.00 KB; ratio: 0.00
  cparams := cparams(clevel=5, shuffle=True, cname='blosclz')
[(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])]
```

Slicing rows returns `allel.model.VariantTable`:

```
>>> vt[:2]
VariantTable((2,), dtype=[('CHROM', 'S4'), ('POS', '<i8'), ('DP', '<i8'), ('QD', '<f8'), ('AC',
[(b'chr1', 2, 35, 4.5, array([1, 2])) (b'chr1', 7, 12, 6.7, array([3, 4]))]
```

Accessing columns returns `allel.bcolz.VariantCTable`:

```
>>> vt[['DP', 'QD']]
VariantCTable((5,), [('DP', '<i8'), ('QD', '<f8')])
  nbytes: 80; cbytes: 32.00 KB; ratio: 0.00
  cparams := cparams(clevel=5, shuffle=True, cname='blosclz')
[(35, 4.5) (12, 6.7) (78, 1.2) (22, 4.4) (99, 2.8)]
```

Use the index to locate variants:

```
>>> loc = vt.index.locate_range(b'chr2', 1, 10)
>>> vt[loc]
VariantTable((2,), dtype=[('CHROM', 'S4'), ('POS', '<i8'), ('DP', '<i8'), ('QD', '<f8'), ('AC',
[(b'chr2', 3, 78, 1.2, array([5, 6])) (b'chr2', 9, 22, 4.4, array([7, 8]))]
```

FeatureCTable

class `allel.model.bcolz.`**`FeatureCTable`**(*data=None*, *copy=False*, ***kwargs*)

Alternative implementation of the `allel.model.FeatureTable` interface, using a `bcolz.ctable` as the backing store.

Parameters **data** : tuple or list of column objects, optional

The list of column data to build the ctable object. This can also be a pure NumPy structured array. May also be a bcolz ctable, which will not be copied if copy=False. May also be None, in which case rootdir must be provided (disk-based array).

copy : bool, optional

If True, copy the input data into a new bcolz ctable.

index : pair or triplet of strings, optional

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

****kwargs** : keyword arguments

Passed through to the bcolz ctable constructor.

Utility functions

allel.model.bcolz.**carray_block_map**(*domain*, *f*, *out=None*, *blen=None*, *wrap=None*, ***kwargs*)

allel.model.bcolz.**carray_block_sum**(*carr*, *axis=None*, *blen=None*, *transform=None*)

allel.model.bcolz.**carray_block_max**(*carr*, *axis=None*, *blen=None*)

allel.model.bcolz.**carray_block_min**(*carr*, *axis=None*, *blen=None*)

allel.model.bcolz.**carray_block_compress**(*carr*, *condition*, *axis*, *blen=None*, ***kwargs*)

allel.model.bcolz.**carray_block_take**(*carr*, *indices*, *axis*, ***kwargs*)

allel.model.bcolz.**carray_from_hdf5**(**args*, ***kwargs*)

Load a bcolz carray from an HDF5 dataset.

Either provide an h5py dataset as a single positional argument, or provide two positional arguments giving the HDF5 file path and the dataset node path within the file.

All keyword arguments are passed through to the bcolz.carray constructor.

allel.model.bcolz.**carray_to_hdf5**(*carr*, *parent*, *name*, ***kwargs*)

Write a bcolz carray to an HDF5 dataset.

Parameters **carr** : bcolz.carray

Data to write.

parent : string or h5py group

Parent HDF5 file or group. If a string, will be treated as HDF5 file name.

name : string

Name or path of dataset to write data into.

kwargs : keyword arguments

Passed through to h5py require_dataset() function.

Returns **h5d** : h5py dataset

allel.model.bcolz.**ctable_block_compress**(*ctbl*, *condition*, *blen=None*, ***kwargs*)

allel.model.bcolz.**ctable_block_take**(*ctbl*, *indices*, ***kwargs*)

allel.model.bcolz.**ctable_from_hdf5_group**(**args*, ***kwargs*)
Load a bcolz ctable from columns stored as separate datasets with an HDF5 group.

Either provide an h5py group as a single positional argument, or provide two positional arguments giving the HDF5 file path and the group node path within the file.

All keyword arguments are passed through to the bcolz.ctable constructor.

allel.model.bcolz.**ctable_to_hdf5_group**(*ctbl*, *parent*, *name*, ***kwargs*)
Write each column in a bcolz ctable to a dataset in an HDF5 group.

Parameters **parent** : string or h5py group

Parent HDF5 file or group. If a string, will be treated as HDF5 file name.

name : string

Name or path of group to write data into.

kwargs : keyword arguments

Passed through to h5py require_dataset() function.

Returns **h5g** : h5py group

2.2 Statistics

Statistical functions for use with variant call data.

2.2.1 Diversity & divergence

allel.stats.diversity.**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.model.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.stats.diversity.**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.model.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.stats.diversity.**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.model.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.stats.diversity.**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.model.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.stats.diversity.**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.

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.model.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.stats.diversity.**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.model.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.stats.diversity.**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.model.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.stats.diversity.**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.model.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.stats.diversity.**tajima_d**(*pos*, *ac*, *start=None*, *stop=None*)

Calculate the value of Tajima's D 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).

Returns **D** : float

Examples

```
>>> import allel
>>> g = allel.model.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 = allel.stats.tajima_d(pos, ac, start=1, stop=31)
>>> D
3.1445848780213814
```

allel.stats.diversity.**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.model.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.stats.diversity.**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`

2.2.2 F-statistics

`allel.stats.fst.`**`weir_cockerham_fst`**(*g*, *subpops*, *max_allele=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.

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.stats.fst.**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.model.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.stats.fst.`**`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.stats.fst.**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.stats.fst.**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.stats.fst.**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.

allel.stats.fst.**blockwise_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.stats.fst.**blockwise_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.stats.fst.**blockwise_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.

2.2.3 Hardy-Weinberg equilibrium

allel.stats.hw.**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.model.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.stats.hw.**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.

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.model.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.stats.hw.**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.model.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])
```

2.2.4 Linkage disequilibrium

allel.stats.ld.**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).

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

Matrix in condensed form.

Examples

```
>>> import allel
>>> g = allel.model.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.        ], dtype=floa
>>> r ** 2
array([ 1.        ,  1.00000024,  1.        ,  1.00000024,  1.        ,  1.        ], dtype=floa
>>> 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.        ]])
```

allel.stats.ld.**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.stats.ld.**locate_unlinked**(*gn*, *size=100*, *step=20*, *threshold=0.1*)

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.

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).

2.2.5 Pairwise distance and ordination

allel.stats.distance.**pairwise_distance**(*x*, *metric*)

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.

Returns **dist** : ndarray, shape (m * (m - 1) / 2,)

Distance matrix in condensed form.

See also:

allel.plot.pairwise_distance

Notes

If *x* is a bcolz carray, a chunk-wise implementation will be used to avoid loading the entire input array into memory. This means that a distance matrix will be calculated for each chunk in 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, although the relative distances may be equivalent.

Examples

```
>>> import allel
>>> g = allel.model.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.stats.distance.**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.GenotypeArray.to_allele_counts

allel.stats.distance.**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.stats.distance.**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.stats.distance.**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.stats.distance.**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

2.2.6 Principal components analysis

allel.stats.decomposition.**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.stats.decomposition.**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.

2.2.7 Admixture tests

`allel.stats.admixture.`**`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.stats.admixture.`**`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.stats.admixture.`**`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.stats.admixture.`**`blockwise_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.stats.admixture.**blockwise_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.

2.2.8 Window utilities

allel.stats.window.**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.stats.window.**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.stats.window.**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.stats.window.**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

2.2.9 Preprocessing utilities

allel.stats.preprocessing.**get_scaler**(*scaler*, *copy*, *ploidy*)

class allel.stats.preprocessing.**CenterScaler**(*copy=True*)

class allel.stats.preprocessing.**StandardScaler**(*copy=True*)

class allel.stats.preprocessing.**PattersonScaler**(*copy=True*, *ploidy=2*)

2.3 Plotting functions

Plotting functions for variant call data.

2.3.1 Variant location

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

2.3.2 Pairwise distance

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

2.3.3 Linkage disequilibrium

`allel.plot.`**`pairwise_ld`**(*m*, *colorbar=True*, *ax=None*, *imshow_kwargs=None*)

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

Parameters **m** : array_like

LD matrix 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

2.4 Input/output utilities

`allel.io.`**`write_vcf`**(*path*, *variants*, *rename=None*, *number=None*, *description=None*, *fill=None*, *write_header=True*)

`allel.io.`**`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.

allel.io.**iter_gff3**(*path*, *attributes=None*, *region=None*, *score_fill=-1*, *phase_fill=-1*, *attributes_fill='.'*)

2.5 Release notes

2.5.1 v0.15

- Added functions to estimate Fst with standard error via a block-jackknife: *allel.stats.fst.blockwise_weir_cockerham_fst()*, *allel.stats.fst.blockwise_hudson_fst()*, *allel.stats.fst.blockwise_patterson_fst()*.
- Fixed a serious bug in *allel.stats.fst.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.model.AlleleCountsArray.max_allele()` to determine highest allele index for each variant.
- Changed first return value from admixture functions *allel.stats.admixture.blockwise_patterson_f3()* and *allel.stats.admixture.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.

2.5.2 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.stats.admixture.blockwise_patterson_f3()* and *allel.stats.admixture.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.

2.5.3 v0.12

- Added functions for principal components analysis of genotype data. Functions in the new module *allel.stats.decomposition* include *allel.stats.decomposition.pca()* to perform a PCA via full singular value decomposition, and *allel.stats.decomposition.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.stats.distance.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.model.GenotypeArray.to_n_ref()`. Also added `dtype` argument to `allel.model.GenotypeArray.to_n_ref()` and `allel.model.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.model.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.bcolz.GenotypeCArray` class. Also added method `allel.model.GenotypeArray.fill_masked()` and similar method on the `allel.bcolz.GenotypeCArray` class to fill masked genotype calls with a value (e.g., -1).

2.5.4 v0.11

- Added functions for calculating Watterson's theta (proportional to the number of segregating variants): *allel.stats.diversity.watterson_theta()* for calculating over a given region, and *allel.stats.diversity.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.stats.diversity.tajima_d()* for calculating over a given region and *allel.stats.diversity.windowed_tajima_d()* for calculating in windows over a chromosome/contig.
- Added *allel.stats.diversity.windowed_df()* for calculating the rate of fixed differences between two populations.
- Added function `allel.model.locate_fixed_differences()` for locating variants that are fixed for different alleles in two different populations.
- Added function `allel.model.locate_private_alleles()` for locating alleles and variants that are private to a single population.

2.5.5 v0.10

- Added functions implementing the Weir and Cockerham (1984) estimators for F-statistics: *allel.stats.fst.weir_cockerham_fst()* and *allel.stats.fst.windowed_weir_cockerham_fst()*.
- Added functions implementing the Hudson (1992) estimator for Fst: *allel.stats.fst.hudson_fst()* and *allel.stats.fst.windowed_hudson_fst()*.
- Added new module *allel.stats.ld* with functions for calculating linkage disequilibrium estimators, including *allel.stats.ld.rogers_huff_r()* for pairwise variant LD calculation, *allel.stats.ld.windowed_r_squared()* for windowed LD calculations, and *allel.stats.ld.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.model.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.model.GenotypeArray.map_alleles()`, `allel.model.HaplotypeArray.map_alleles()` and `allel.model.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.

2.5.6 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.

2.5.7 v0.8

- Added *subpop* argument to `allel.model.GenotypeArray.count_alleles()` and `allel.model.HaplotypeArray.count_alleles()` to enable count alleles within a sub-population without subsetting the array.
- Added functions `allel.model.GenotypeArray.count_alleles_subpops()` and `allel.model.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.stats.distance.pairwise_dxy()* for computing a distance matrix using Dxy as the metric.

2.5.8 v0.7

- Added function *allel.io.write_fasta()* for writing a nucleotide sequence stored as a NumPy array out to a FASTA format file.

2.5.9 v0.6

- Added method `allel.model.VariantTable.to_vcf()` for writing a variant table to a VCF format file.

CHAPTER 3

Acknowledgments

Development of this package is supported by the MRC Centre for Genomics and Global Health.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

a

A

B

C

E

F

G

H

I

L

M

N

P

Q

R

S

T

U

V

W