scikit-allel Documentation

Release 0.9.1

Alistair Miles

March 12, 2015

Contents

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

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

- GitHub repository: https://github.com/cggh/scikit-allel

CHAPTER 1

Installation

This package requires numpy, scipy, matplotlib, pandas, h5py, numexpr and bcolz. Install dependencies first, then:

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

CHAPTER 2

Contents

2.1 Data structures

This module defines NumPy array classes for variant call data.

2.1.1 GenotypeArray

class `allel.model.`**`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.

In many cases the number of distinct alleles for each variant is small, e.g., less than 10, or even 2 (all variants are biallelic). In these cases a genotype array is not the most compact way of storing genotype data in memory. This class defines functions for bit-packing diploid genotype calls into single bytes, and for transforming genotype arrays into sparse matrices, which can assist in cases where memory usage needs to be minimised. Note however that these more compact representations do not allow the same flexibility in terms of using numpy universal functions to access and manipulate data.

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.

Examples

Instantiate a genotype array:

```
>>> import allel
>>> g = allel.model.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.model.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).

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.model.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.model.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.model.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.model.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.model.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.model.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()

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.

Examples

```
>>> import allel
>>> g = allel.model.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)
```

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.model.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(*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.model.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.

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.model.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_alt(*fill=0*)

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.model.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.model.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.

Examples

```
>>> import allel
>>> g = allel.model.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.model.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

Examples

```
>>> import allel
>>> g = allel.model.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.model.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)

Examples

```
>>> import allel
>>> g = allel.model.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

Examples

```
>>> import allel
>>> import numpy as np
>>> np.random.seed(42)
>>> g = allel.model.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.model.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]]
```

2.1.2 HaplotypeArray

class allel.model.**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.model.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.model.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.

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.model.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.model.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.model.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]]
```

2.1.3 AlleleCountsArray

class `allel.model.`**`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.model.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).

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.model.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.model.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.model.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.model.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.model.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.model.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.model.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.model.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.  ]])
```

2.1.4 VariantTable

class `allel.model.`**`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.model.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’, ’<i8’), (’QD’, ’<f8’), (’AC’,
[(b’chr2’, 3, 78, 1.2, array([5, 6])) (b’chr2’, 9, 22, 4.4, array([7, 8]))]
```

Use the index to query variants:

```
>>> vt.query_region(b’chr2’, 1, 10)
VariantTable((2,), dtype=[(’CHROM’, ’S4’), (’POS’, ’<u4’), (’DP’, ’<i8’), (’QD’, ’<f8’), (’AC’,
[(b’chr2’, 3, 78, 1.2, array([5, 6])) (b’chr2’, 9, 22, 4.4, array([7, 8]))]
```

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.model.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.model.VariantTable(records, dtype=dtype)
>>> vt.query('DP > 30')
VariantTable((3,), dtype=[('CHROM', 'S4'), ('POS', '<u4'), ('DP', '<i8'), ('QD', '<f8'), ('A
[(b'chr1', 2, 35, 4.5, array([1, 2])) (b'chr2', 3, 78, 1.2, array([5, 6]))
 (b'chr3', 6, 99, 2.8, array([ 9, 10]))]
>>> vt.query('(DP > 30) & (QD > 4)')
VariantTable((1,), dtype=[('CHROM', 'S4'), ('POS', '<u4'), ('DP', '<i8'), ('QD', '<f8'), ('A
[(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.model.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=1.2,2.
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;xx=5.6,
chr2        8       d       G       C,A     4.5     PASS    DP=45;QD=45.6;ac=7,8;xx=7.8,8.9
chr3        1       e       N       X       5.6     PASS    DP=56;QD=56.7;ac=9;flg;xx=9.0,9.
```

2.1.5 FeatureTable

class `allel.model.`**`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

2.1.6 SortedIndex

class `allel.model.`**`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.model.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.model.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.model.SortedIndex([3, 6, 11, 20, 35])
>>> idx2 = allel.model.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.model.SortedIndex([3, 6, 11, 20, 35])
>>> idx2 = allel.model.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.model.SortedIndex([3, 6, 11, 20, 35])
>>> idx2 = allel.model.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.model.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.model.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.model.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.model.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.model.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]
```

2.1.7 UniqueIndex

class allel.model.**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.model.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.model.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.model.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.model.UniqueIndex([’A’, ’C’, ’B’, ’F’])
>>> idx2 = allel.model.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.model.UniqueIndex([’A’, ’C’, ’B’, ’F’])
>>> idx2 = allel.model.UniqueIndex([’X’, ’F’, ’G’, ’C’, ’Z’])
>>> idx1.intersect(idx2)
UniqueIndex(2, dtype=<U1)
[’C’ ’F’]
```

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

2.1.8 SortedMultiIndex

class `allel.model.`**`SortedMultiIndex`**(*l1*, *l2*, *copy=True*)

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.model.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.model.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.model.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)
```

2.2 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.

2.2.1 GenotypeCArray

class `allel.bcolz.`**`GenotypeCArray`**(*data=None*, *copy=True*, ***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.bcolz.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.bcolz.GenotypeCArray(data, copy=False)
>>> 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.bcolz.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()
carray((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]]
```

2.2.2 HaplotypeCArray

class `allel.bcolz.`**`HaplotypeCArray`**(*data=None*, *copy=True*, ***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.

2.2.3 AlleleCountsCArray

class `allel.bcolz.`**`AlleleCountsCArray`**(*data=None*, *copy=True*, ***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.

2.2.4 VariantCTable

class `allel.bcolz.`**`VariantCTable`**(*data=None*, *copy=True*, *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]))]
```

2.2.5 FeatureCTable

class `allel.bcolz.`**`FeatureCTable`**(*data=None*, *copy=True*, ***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.

2.2.6 Utility functions

`allel.bcolz.`**`carray_block_map`**(*carr*, *f*, *out=None*, *blen=None*, ***kwargs*)

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

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

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

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

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

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

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

`allel.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.

2.3 Statistics

This module provides statistical functions for use with variant call data.

2.3.1 Diversity & divergence

`allel.stats.`**`mean_pairwise_diversity`**(*ac*, *fill=nan*)

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

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

Allele counts array.

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_diversity(ac)
array([ 0.        ,  0.5       ,  0.66666667,  0.5       ,  0.        ,
        0.83333333,  0.83333333,  1.        ])
```

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

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

Calculate 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

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.`**`mean_pairwise_divergence`**(*ac1*, *ac2*, *an1=None*, *an2=None*, *fill=nan*)

Calculate for each variant the mean number of pairwise differences between haplotypes 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.take([0, 1], axis=1).count_alleles()
>>> ac2 = h.take([2, 3], axis=1).count_alleles()
>>> allel.stats.mean_pairwise_divergence(ac1, ac2)
array([ 0.  ,  0.5 ,  1.  ,  0.5 ,  0.  ,  1.  ,  0.75,   nan])
```

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

Calculate 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]])
>>> h1 = h.subset(haplotypes=[0, 1])
>>> h2 = h.subset(haplotypes=[2, 3])
>>> ac1 = h1.count_alleles()
>>> ac2 = h2.count_alleles()
>>> 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.`**`windowed_divergence`**(*pos*, *ac1*, *ac2*, *size*, *start=None*, *stop=None*, *step=None*, *is_accessible=None*, *fill=nan*)

Calculate 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

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]])
>>> h1 = h.subset(haplotypes=[0, 1])
>>> h2 = h.subset(haplotypes=[2, 3])
>>> ac1 = h1.count_alleles()
>>> ac2 = h2.count_alleles()
>>> 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])
```

2.3.2 Pairwise distance

allel.stats.**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 (n_individuals * (n_individuals - 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.`**`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`

2.3.3 Hardy-Weinberg equilibrium

`allel.stats.`**`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.`**`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.**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])
```

2.3.4 Window utilities

allel.stats.**moving_statistic**(*values*, *statistic*, *size=None*, *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.**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

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.`**`windowed_statistic`**(*pos*, *values*, *statistic*, *size*, *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.

statistic : function

The statistic to compute.

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 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.**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

2.4 Plotting functions

Plotting functions for variant call data.

2.4.1 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

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*

Python Module Index

Index