<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Basic Statistics</td>
<td>1</td>
</tr>
<tr>
<td>2 C Int Lists</td>
<td>11</td>
</tr>
<tr>
<td>3 Hidden Markov Models</td>
<td>17</td>
</tr>
<tr>
<td>4 Continuous Emission Hidden Markov Models</td>
<td>31</td>
</tr>
<tr>
<td>5 Distributions used in implementing Hidden Markov Models</td>
<td>45</td>
</tr>
<tr>
<td>6 Hidden Markov Models – Utility functions</td>
<td>53</td>
</tr>
<tr>
<td>7 Discrete Gaussian Samplers over the Integers</td>
<td>57</td>
</tr>
<tr>
<td>8 Discrete Gaussian Samplers for $\mathbb{Z}[x]$</td>
<td>63</td>
</tr>
<tr>
<td>9 Discrete Gaussian Samplers over Lattices</td>
<td>67</td>
</tr>
<tr>
<td>10 T-test using R</td>
<td>79</td>
</tr>
<tr>
<td>11 Indices and Tables</td>
<td>81</td>
</tr>
<tr>
<td>Python Module Index</td>
<td>83</td>
</tr>
<tr>
<td>Index</td>
<td>85</td>
</tr>
</tbody>
</table>
This file contains basic descriptive functions. Included are the mean, median, mode, moving average, standard deviation, and the variance. When calling a function on data, there are checks for functions already defined for that data type.

The `mean()` function returns the arithmetic mean (the sum of all the members of a list, divided by the number of members). Further revisions may include the geometric and harmonic mean. The `median()` function returns the number separating the higher half of a sample from the lower half. The `mode()` returns the most common occurring member of a sample, plus the number of times it occurs. If entries occur equally common, the smallest of a list of the most common entries is returned. The `moving_average()` is a finite impulse response filter, creating a series of averages using a user-defined number of subsets of the full data set. The `std()` and the `variance()` return a measurement of how far data points tend to be from the arithmetic mean.

Functions are available in the namespace `stats`, i.e. you can use them by typing `stats.mean`, `stats.median`, etc.

REMARK: If all the data you are working with are floating point numbers, you may find `stats.TimeSeries` helpful, since it is extremely fast and offers many of the same descriptive statistics as in the module.

AUTHOR:

- Andrew Hou (11/06/2009)

```python
sage.stats.basic_stats.mean(v)
```

Return the mean of the elements of `v`.

We define the mean of the empty list to be the (symbolic) NaN, following the convention of MATLAB, Scipy, and R.

This function is deprecated. Use `numpy.mean()` or `numpy.nanmean()` instead.

INPUT:

- `v` – a list of numbers

OUTPUT:

- a number

EXAMPLES:

```python
sage: mean([pi, e])  #... -> needs sage.symbolic
doctest:warning...
DeprecationWarning: sage.stats.basic_stats.mean is deprecated; use numpy.mean or numpy.nanmean instead
See https://github.com/sagemath/sage/issues/29662 for details.
1/2*pi + 1/2*e
sage: mean([])  #... (continues on next page)```
Return the median (middle value) of the elements of \( v \).

If \( v \) is empty, we define the median to be NaN, which is consistent with NumPy (note that R returns NULL). If \( v \) is comprised of strings, TypeError occurs. For elements other than numbers, the median is a result of sorted().

This function is deprecated. Use \texttt{numpy.median()} or \texttt{numpy.nanmedian()} instead.

**INPUT:**
- \( v \) – a list

**OUTPUT:**
- median element of \( v \)

**EXAMPLES:**
sage: median([1,2,3,4,5])
doctest:warning...
DeprecationWarning: sage.stats.basic_stats.median is deprecated;
use numpy.median or numpy.nanmedian instead
See https://github.com/sagemath/sage/issues/29662 for details.
3
sage: median([e, pi])
^needs sage.symbolic
1/2*pi + 1/2*e
sage: median(['sage', 'linux', 'python'])
'python'
sage: median([])
^needs sage.symbolic
NaN
sage: class MyClass:
    ....: def median(self):
    ....:     return 1
sage: stats.median(MyClass())
1

>>> from sage.all import *
>>> median([Integer(1),Integer(2),Integer(3),Integer(4),Integer(5)])
doctest:warning...
DeprecationWarning: sage.stats.basic_stats.median is deprecated;
use numpy.median or numpy.nanmedian instead
See https://github.com/sagemath/sage/issues/29662 for details.
3
>>> median([e, pi])
^needs sage.symbolic
1/2*pi + 1/2*e
>>> median(['sage', 'linux', 'python'])
'python'
>>> median([])
^needs sage.symbolic
NaN
>>> class MyClass:
...    def median(self):
...        return Integer(1)
>>> stats.median(MyClass())
1

sage.stats.basic_stats.mode(v)

Return the mode of v.

The mode is the list of the most frequently occurring elements in v. If \( n \) is the most times that any element occurs in v, then the mode is the list of elements of v that occur \( n \) times. The list is sorted if possible.

This function is deprecated. Use scipy.stats.mode() or statistics.mode() instead.

Note: The elements of v must be hashable.

INPUT:

- v – a list

OUTPUT:

- a list (sorted if possible)
EXAMPLES:

```
sage: v = [1,2,4,1,6,2,6,7,1]
sage: mode(v)

doctest:warning...
DeprecationWarning: sage.stats.basic_stats.mode is deprecated;
use scipy.stats.mode or statistics.mode instead
See https://github.com/sagemath/sage/issues/29662 for details.
[1]
sage: v.count(1)
3
sage: mode([])
[]
sage: mode([1,2,3,4,5])
[1, 2, 3, 4, 5]
sage: mode([3,1,2,1,2,3])
[1, 2, 3]
sage: mode([0, 2, 7, 7, 13, 20, 2, 13])
[2, 7, 13]
sage: mode(['sage', 'four', 'I', 'three', 'sage', 'pi'])
['sage']
sage: class MyClass:
....:     def mode(self):
....:         return [1]
sage: stats.mode(MyClass())
[1]
```

```
>>> from sage.all import *
>>> v = [Integer(1),Integer(2),Integer(4),Integer(1),Integer(6),Integer(2),
->Integer(6),Integer(7),Integer(1)]
>>> mode(v)

doctest:warning...
DeprecationWarning: sage.stats.basic_stats.mode is deprecated;
use scipy.stats.mode or statistics.mode instead
See https://github.com/sagemath/sage/issues/29662 for details.
[1]
>>> v.count(Integer(1))
3
>>> mode([])
[]
>>> mode([Integer(1),Integer(2),Integer(3),Integer(4),Integer(5)])
[1, 2, 3, 4, 5]
>>> mode([Integer(3),Integer(1),Integer(2),Integer(1),Integer(2),Integer(3)])
[1, 2, 3]
>>> mode([Integer(0), Integer(2), Integer(7), Integer(7), Integer(13),
->Integer(20), Integer(2), Integer(13)])
[2, 7, 13]
>>> mode(['sage', 'four', 'I', 'three', 'sage', 'pi'])
['sage']
>>> class MyClass:
...     def mode(self):
...         return [Integer(1)]
```

(continues on next page)
sage.stats.basic_stats.moving_average(v, n)

Return the moving average of a list \( v \).

The moving average of a list is often used to smooth out noisy data.

If \( v \) is empty, we define the entries of the moving average to be NaN.

This method is deprecated. Use pandas.Series.rolling() instead.

INPUT:

- \( v \) – a list
- \( n \) – the number of values used in computing each average.

OUTPUT:

- a list of length \( \text{len}(v) - n + 1 \), since we do not fabric any values

EXAMPLES:

```python
sage: moving_average([1..10], 1)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

sage: moving_average([1..10], 4)
[5/2, 7/2, 9/2, 11/2, 13/2, 15/2, 17/2]

sage: moving_average([], 1)
[]

sage: moving_average([pi, e, I, sqrt(2), 3/5], 2)
[1/2*pi + 1/2*e, 1/2*e + 1/2*I, 1/2*sqrt(2) + 1/2*I, 1/2*sqrt(2) + 3/10]
```

We check if the input is a time series, and if so use the optimized simple_moving_average() method, but with (slightly different) meaning as defined above (the point is that the simple_moving_average() on time series returns \( n \) values:

```python
>>> from sage.all import *

>>> moving_average(ellipsis_range(Integer(1), Ellipsis, Integer(10)), Integer(1))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

>>> moving_average(ellipsis_range(Integer(1), Ellipsis, Integer(10)), Integer(4))
[5/2, 7/2, 9/2, 11/2, 13/2, 15/2, 17/2]

>>> moving_average([], Integer(1))
[]

>>> moving_average([pi, e, I, sqrt(Integer(2)), Integer(3)/Integer(5)], Integer(2))
[1/2*pi + 1/2*e, 1/2*e + 1/2*I, 1/2*sqrt(2) + 1/2*I, 1/2*sqrt(2) + 3/10]
```

\* needs sage.symbolic

\# needs sage.symbolic
sage: a = stats.TimeSeries([1..10])  # needs numpy
sage: stats.moving_average(a, 3)  # needs numpy
[2.0000, 3.0000, 4.0000, 5.0000, 6.0000, 7.0000, 8.0000, 9.0000]
sage: stats.moving_average(list(a), 3)  # needs numpy
[2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]

```python
def main():
    from sage.all import *
    a = stats.TimeSeries((ellipsis_range(Integer(1),Ellipsis,Integer(10))))  # needs numpy
    stats.moving_average(a, Integer(3))  # needs numpy
    stats.moving_average(list(a), Integer(3))  # needs numpy
```

sage.stats.basic_stats.std(v, bias=False)

Return the standard deviation of the elements of \(v\).

We define the standard deviation of the empty list to be NaN, following the convention of MATLAB, Scipy, and R.

This function is deprecated. Use `numpy.std()` or `numpy.nanstd()` instead.

INPUT:

- \(v\) – a list of numbers
- \(bias\) – bool (default: False); if False, divide by \(\text{len}(v) - 1\) instead of \(\text{len}(v)\) to give a less biased estimator (sample) for the standard deviation.

OUTPUT:

- a number

EXAMPLES:

```makefile
sage: # needs sage.symbolic
sage: std([1..6], bias=True)
1/2*sqrt(35/3)
sage: std([1..6], bias=False)
sqrt(7/2)
sage: std([e, pi])
sqrt(1/2)*abs(pi - e)
```
sage: std([])
NaN
sage: std([I, sqrt(2), 3/5])
1/15*sqrt(1/2)*sqrt((10*sqrt(2) - 5*I - 3)^2
+ (5*sqrt(2) - 10*I + 3)^2 + (5*sqrt(2) + 5*I - 6)^2)
sage: std([RIF(1.0103, 1.0103), RIF(2)])
0.69982358134032617

sage: # needs numpy
sage: import numpy
sage: x = numpy.array([1,2,3,4,5])
sage: std(x, bias=False)
1.5811388300841898
sage: x = stats.TimeSeries([1..100])
sage: std(x)
29.011491975882016

>>> from sage.all import *
>>> # needs sage.symbolic
>>> std((ellipsis_range(Integer(1),Ellipsis,Integer(6))), bias=True)
doctest:warning...
DeprecationWarning: sage.stats.basic_stats.std is deprecated;
use numpy.std or numpy.nanstd instead
See https://github.com/sagemath/sage/issues/29662 for details.
doctest:warning...
DeprecationWarning: sage.stats.basic_stats.variance is deprecated;
use numpy.var or numpy.nanvar instead
See https://github.com/sagemath/sage/issues/29662 for details.
doctest:warning...
DeprecationWarning: sage.stats.basic_stats.mean is deprecated;
use numpy.mean or numpy.nanmean instead
See https://github.com/sagemath/sage/issues/29662 for details.
1/2*sqrt(35/3)
>>> std((ellipsis_range(Integer(1),Ellipsis,Integer(6))), bias=False)
sqrt(7/2)
>>> std([e, pi])
sqrt(1/2)*abs(pi - e)
>>> std([])
NaN
>>> std([I, sqrt(Integer(2)), Integer(3)/Integer(5)])
1/15*sqrt(1/2)*sqrt((10*sqrt(2) - 5*I - 3)^2
+ (5*sqrt(2) - 10*I + 3)^2 + (5*sqrt(2) + 5*I - 6)^2)
>>> std([RIF(RealNumber('1.0103'), RealNumber('1.0103')), RIF(Integer(2))])
0.69982358134032617

>>> # needs numpy
>>> import numpy
>>> x = numpy.array([Integer(1),Integer(2),Integer(3),Integer(4),Integer(5)])
>>> std(x, bias=False)
1.5811388300841898
>>> x = stats.TimeSeries((ellipsis_range(Integer(1),Ellipsis,Integer(100))))
>>> std(x)
29.011491975882016

sage.stats.basic_stats.variance(v, bias=False)

Return the variance of the elements of v.
We define the variance of the empty list to be NaN, following the convention of MATLAB, Scipy, and R.

This function is deprecated. Use `numpy.var()` or `numpy.nanvar()` instead.

**INPUT:**

- `v` – a list of numbers
- `bias` – bool (default: `False`); if `False`, divide by `len(v) - 1` instead of `len(v)` to give a less biased estimator (sample) for the standard deviation.

**OUTPUT:**

- a number

**EXAMPLES:**

```python
sage: variance([1..6])
7/2
sage: variance([1..6], bias=True)
35/12
sage: variance([e, pi])
1/2*(pi - e)^2
sage: variance([])
NaN
sage: variance([I, sqrt(2), 3/5])
1/450*(10*sqrt(2) - 5*I - 3)^2 + 1/450*(5*sqrt(2) - 10*I + 3)^2 + 1/450*(5*sqrt(2) + 5*I - 6)^2
sage: variance(RIF(1.0103, 1.0103), RIF(2))
0.4897530450000000?
```

See https://github.com/sagemath/sage/issues/29662 for details.
....:    def mean(self):
....:        return 3

def mean(self):
    return 3

sage: R = SillyPythonList()
sage: variance(R)
2
sage: variance(R, bias=True)
1

>>> from sage.all import *
>>> variance((ellipsis_range(Integer(1), Ellipsis, Integer(6))))
    doctest:warning...
DeprecationWarning: sage.stats.basic_stats.variance is deprecated; use numpy.var or numpy.nanvar instead
See https://github.com/sagemath/sage/issues/29662 for details.
7/2
>>> variance((ellipsis_range(Integer(1), Ellipsis, Integer(6))), bias=True)  
35/12
>>> variance([e, pi])
    # needs sage.symbolic
1/2*(pi - e)^2
>>> variance([])
NaN
>>> variance([I, sqrt(Integer(2)), Integer(3)/Integer(5)])
    # needs sage.symbolic
1/450*(10*sqrt(2) - 5*I - 3)^2 + 1/450*(5*sqrt(2) - 10*I + 3)^2
    + 1/450*(5*sqrt(2) + 5*I - 6)^2
>>> variance([RIF(RealNumber('1.0103'), RealNumber('1.0103')), RIF(Integer(2))])
    # needs numpy
0.4897530450000000?
>>> import numpy
    # needs numpy
>>> x = numpy.array([Integer(1), Integer(2), Integer(3), Integer(4), Integer(5)])
    # needs numpy
>>> variance(x, bias=False)
2.5
>>> x = stats.TimeSeries((ellipsis_range(Integer(1), Ellipsis, Integer(100))))
>>> variance(x)
841.6666666666666
>>> variance(x, bias=True)
833.25

>>> class MyClass:
...    def variance(self, bias=False):
...        return Integer(1)

>>> stats.variance(MyClass())
1

>>> class SillyPythonList:
...    def __init__(self):
...        self.__list = [Integer(2), Integer(4)]
...    def __len__(self):
...        return len(self.__list)
...    def __iter__(self):
...        return self.__list.__iter__()
...    def mean(self):
...        return Integer(3)

>>> R = SillyPythonList()
>>> variance(R)
2

```python
>>> variance(R, bias=True)
```

1
This is a class for fast basic operations with lists of C ints. It is similar to the double precision TimeSeries class. It has all the standard C int semantics, of course, including overflow. It is also similar to the Python list class, except all elements are C ints, which makes some operations much, much faster. For example, concatenating two IntLists can be over 10 times faster than concatenating the corresponding Python lists of ints, and taking slices is also much faster.

AUTHOR:

• William Stein, 2010-03

class sage.stats.intlist.IntList

   Bases: object

   A list of C int's.

   list()

      Return Python list version of self with Python ints as entries.

   EXAMPLES:

   sage: a = stats.IntList([1..15]); a
   [1, 2, 3, 4, 5 ... 11, 12, 13, 14, 15]
   sage: a.list()
   [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
   sage: list(a) == a.list()
   True
   sage: type(a.list()[0])
   <... 'int'>

   >>> from sage.all import *
   >>> a = stats.IntList((ellipsis_range(Integer(1),Ellipsis,Integer(15)))); a
   [1, 2, 3, 4, 5 ... 11, 12, 13, 14, 15]
   >>> a.list()
   [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
   >>> list(a) == a.list()
   True
   >>> type(a.list()[Integer(0)])
   <... 'int'>

max(index=False)

   Return the largest value in this time series. If this series has length 0 we raise a ValueError

   INPUT:

   • index – bool (default: False); if True, also return index of maximum entry.

   OUTPUT:
• int – largest value
• int – index of largest value; only returned if index=True

EXAMPLES:

```python
sage: v = stats.IntList([1,-4,3,-2,-4,3])
sage: v.max()
3
sage: v.max(index=True)
(3, 2)
```

```python
>>> from sage.all import *
>>> v = stats.IntList([Integer(1),-Integer(4),Integer(3),-Integer(2),-
˓→Integer(4),Integer(3)])
>>> v.max()
3
>>> v.max(index=True)
(-4, 1)
```

**min (index=False)**

Return the smallest value in this integer list. If this series has length 0 we raise a **ValueError**.

**INPUT:**

• index – bool (default: False); if True, also return index of minimal entry.

**OUTPUT:**

• float – smallest value
• integer – index of smallest value; only returned if index=True

**EXAMPLES:**

```python
sage: v = stats.IntList([1,-4,3,-2,-4])
sage: v.min()
-4
sage: v.min(index=True)
(-4, 1)
```

```python
>>> from sage.all import *
>>> v = stats.IntList([Integer(1),-Integer(4),Integer(3),-Integer(2),-
˓→Integer(4)])
>>> v.min()
(-4, 1)
```

**plot (**args, **kwds)**

Return a plot of this **IntList**.

This just constructs the corresponding double-precision floating point **TimeSeries** object, passing on all arguments.

**EXAMPLES:**

```python
sage: stats.IntList([3,7,19,-2]).plot()  # needs sage.plot
```

Graphics object consisting of 1 graphics primitive
sage: stats.IntList([3,7,19,-2]).plot(color='red',
                   needs_sage.plot, #
                   ....: points=50, points=True)
Graphics object consisting of 1 graphics primitive

>>> from sage.all import *
>>> stats.IntList([Integer(3),Integer(7),Integer(19),-Integer(2)]).plot() # needs sage.plot
Graphics object consisting of 1 graphics primitive

>>> stats.IntList([Integer(3),Integer(7),Integer(19),-Integer(2)]).plot(color=
                   'red', # needs sage.plot
                   ....: points=Integer(50), points=True)
Graphics object consisting of 1 graphics primitive

plot_histogram(*args, **kwds)

Return a histogram plot of this IntList.

This just constructs the corresponding double-precision floating point TimeSeries object, and plots it, passing on all arguments.

EXAMPLES:

sage: stats.IntList([1..15]).plot_histogram() # needs sage.plot
Graphics object consisting of 50 graphics primitives

>>> from sage.all import *

>>> stats.IntList((ellipsis_range(Integer(1),Ellipsis,Integer(15))).plot_histogram() # needs sage.plot
Graphics object consisting of 50 graphics primitives

prod()

Return the product of the entries of self.

EXAMPLES:

sage: a = stats.IntList([1..10]); a
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
sage: a.prod()
3628800
sage: factorial(10)
3628800

>>> from sage.all import *

>>> a = stats.IntList((ellipsis_range(Integer(1),Ellipsis,Integer(10))).prod() # needs sage.plot
>>> a.prod()
3628800
>>> factorial(Integer(10))
3628800

Note that there can be overflow:

sage: a = stats.IntList([2^30, 2]); a
[1073741824, 2]
sage: a.prod()
-2147483648

```python
>>> from sage.all import *

>>> a = stats.IntList([Integer(2)**Integer(30), Integer(2)]); a
[1073741824, 2]

>>> a.prod()
-2147483648
```

### sum()

Return the sum of the entries of self.

**EXAMPLES:**

```python
sage: stats.IntList([1..100]).sum()
5050

>>> from sage.all import *

>>> stats.IntList((ellipsis_range(Integer(1),Ellipsis,Integer(100)))).sum()
5050
```

Note that there can be overflow, since the entries are C ints:

```python
sage: a = stats.IntList([2^30,2^30]); a
[1073741824, 1073741824]

sage: a.sum()
-2147483648

>>> from sage.all import *

>>> a = stats.IntList([Integer(2)**Integer(30),Integer(2)**Integer(30)]); a
[1073741824, 1073741824]

>>> a.sum()
-2147483648
```

### time_series()

Return TimeSeries version of self, which involves changing each entry to a double.

**EXAMPLES:**

```python
sage: T = stats.IntList([-2,3,5]).time_series(); T
[-2.0000, 3.0000, 5.0000]

sage: type(T)
<... sage.stats.time_series.TimeSeries>

>>> from sage.all import *

>>> T = stats.IntList([-Integer(2),Integer(3),Integer(5)]).time_series(); T
[-2.0000, 3.0000, 5.0000]

>>> type(T)
<... 'sage.stats.time_series.TimeSeries'>
```

`sage.stats.intlist.unpickle_intlist_v1(v, n)`

Version 1 unpickle method.

**INPUT:**

- v – a raw char buffer
EXAMPLES:

```python
sage: v = stats.IntList([1, 2, 3])
sage: s = v.__reduce__()[1][0]
sage: type(s) == type(b'')
True
sage: sage.stats.intlist.unpickle_intlist_v1(s, 3)
[1, 2, 3]
sage: sage.stats.intlist.unpickle_intlist_v1(s+s, 6)
[1, 2, 3, 1, 2, 3]
sage: sage.stats.intlist.unpickle_intlist_v1(b'', 0)
[]
```

```python
>>> from sage.all import *
>>> v = stats.IntList([Integer(1), Integer(2), Integer(3)])
>>> s = v.__reduce__()[Integer(1)][Integer(0)]
>>> type(s) == type(b''
True
>>> sage.stats.intlist.unpickle_intlist_v1(s, Integer(3))
[1, 2, 3]
>>> sage.stats.intlist.unpickle_intlist_v1(s+s, Integer(6))
[1, 2, 3, 1, 2, 3]
>>> sage.stats.intlist.unpickle_intlist_v1(b'', Integer(0))
[]
```
This is a complete pure-Cython optimized implementation of Hidden Markov Models. It fully supports Discrete, Gaussian, and Mixed Gaussian emissions.

The best references for the basic HMM algorithms implemented here are:

- Tapas Kanungo’s “Hidden Markov Models”
- Jackson's HMM tutorial: 
  http://personal.ee.surrey.ac.uk/Personal/P.Jackson/tutorial/

LICENSE: Some of the code in this file is based on reading Kanungo’s GPLv2+ implementation of discrete HMM’s, hence the present code must be licensed with a GPLv2+ compatible license.

AUTHOR:
- William Stein, 2010-03

```
class sage.stats.hmm.hmm.DiscreteHiddenMarkovModel
    Bases: HiddenMarkovModel

    A discrete Hidden Markov model implemented using double precision floating point arithmetic.

    INPUT:

    - A – a list of lists or a square \( N \times N \) matrix, whose \((i, j)\) entry gives the probability of transitioning from state \( i \) to state \( j \).
    - B – a list of \( N \) lists or a matrix with \( N \) rows, such that \( B[i, k] \) gives the probability of emitting symbol \( k \) while in state \( i \).
    - \( \pi \) – the probabilities of starting in each initial state, i.e., \( \pi[i] \) is the probability of starting in state \( i \).
    - \text{emission\_symbols} – None or list (default: None); if None, the \text{emission\_symbols} are the ints \([0..N-1]\), where \( N \) is the number of states. Otherwise, they are the entries of the list \text{emissions\_symbols}, which must all be hashable.
    - \text{normalize} – bool (default: True); if given, input is normalized to define valid probability distributions, e.g., the entries of \( A \) are made nonnegative and the rows sum to 1, and the probabilities in \( \pi \) are normalized.

    EXAMPLES:

    sage: m = hmm.DiscreteHiddenMarkovModel([[0.4,0.6],[0.1,0.9]],
        ....: [[0.1,0.9],[0.5,0.5]],
        ....: [.5,.5]); m
    Discrete Hidden Markov Model with 2 States and 2 Emissions
    Transition matrix:
    [0.4  0.6]
    [0.1  0.9]
```

(continues on next page)
Emission matrix:
\[
\begin{bmatrix}
0.1 & 0.9 \\
0.5 & 0.5
\end{bmatrix}
\]

Initial probabilities: [0.5000, 0.5000]
```
sage: m.log_likelihood([0,1,0,1,0,1])
-4.66693474691329...
```
```
sage: m.viterbi([0,1,0,1,0,1])
([1, 1, 1, 1, 1, 1], -5.378832842208748)
```
```
sage: m.baum_welch([0,1,0,1,0,1])
(0.0, 22)
```
```
sage: m  # rel tol 1e-10
Discrete Hidden Markov Model with 2 States and 2 Emissions
Transition matrix:
\[
\begin{bmatrix}
1.0134345614745788e-70 & 1.0 \\
1.0 & 3.9974352713558623e-19
\end{bmatrix}
\]
Emission matrix:
\[
\begin{bmatrix}
7.380221566254936e-54 & 1.0 \\
1.0 & 3.997435262602193e-19
\end{bmatrix}
\]
Initial probabilities: [0.0000, 1.0000]
```
```
sage: m.sample(10)
[0, 1, 0, 1, 0, 1, 0, 1, 0, 1]
```
```
sage: m.graph().plot()
# needs sage.plot
```
Graphics object consisting of 6 graphics primitives

```python
>>> from sage.all import *
>>> m = hmm.DiscreteHiddenMarkovModel([[RealNumber('0.4'), RealNumber('0.6')],
    ...
    [RealNumber('0.1'), RealNumber('0.9')]],
    ...
    [RealNumber('0.5'), RealNumber('0.5')]); m
```
```
Discrete Hidden Markov Model with 2 States and 2 Emissions
Transition matrix:
\[
\begin{bmatrix}
0.4 & 0.6 \\
0.1 & 0.9
\end{bmatrix}
\]
Emission matrix:
\[
\begin{bmatrix}
0.1 & 0.9 \\
0.5 & 0.5
\end{bmatrix}
\]
Initial probabilities: [0.5000, 0.5000]
```
```
>>> m.log_likelihood([Integer(0), Integer(1), Integer(0), Integer(1), Integer(0),
    ...
    Integer(1)])
-4.66693474691329...
```
```
>>> m.viterbi([Integer(0), Integer(1), Integer(0), Integer(1), Integer(0), Integer(1)])
([1, 1, 1, 1, 1, 1], -5.378832842208748)
```
```
>>> m.baum_welch([Integer(0), Integer(1), Integer(0), Integer(1), Integer(0),
    ...
    Integer(1)])
(0.0, 22)
```
```
>>> m  # rel tol 1e-10
Discrete Hidden Markov Model with 2 States and 2 Emissions
Transition matrix:
\[
\begin{bmatrix}
1.0134345614745788e-70 & 1.0 \\
1.0 & 3.9974352713558623e-19
\end{bmatrix}
\]
Emission matrix:
\[
\begin{bmatrix}
7.380221566254936e-54 & 1.0 \\
1.0 & 3.997435262602193e-19
\end{bmatrix}
\]
Initial probabilities: [0.0000, 1.0000]
```
A 3-state model that happens to always outputs 'b':

```
sage: m = hmm.DiscreteHiddenMarkovModel([[1/3]*3]*3, [[0,1,0]]*3, [1/3]*3, ['a','b', 'c'])
sage: m.sample(10)
['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b']
```
The following illustrates how Baum-Welch is only a local optimizer, i.e., the above model is far more likely to produce the sequence [1,0]*20 than the one we get below:

```sage
m = hmm.DiscreteHiddenMarkovModel([[RealNumber(0.5), RealNumber(0.5)],
    ...
    [RealNumber(0.5), RealNumber(0.5)]],
    ...
    [RealNumber(0.5), RealNumber(0.5)])
>>> m.baumwelch([Integer(1), Integer(0)]*Integer(20), log_likelihood_cutoff=0)
(-27.725887222397784, 1)
```
```python
sage: m = hmm.DiscreteHiddenMarkovModel([[0.1,0.9],[0.9,0.1]],
......: [[0.5,0.5],[0.2,0.8]],
......:
[0.2,0.8])
sage: set_random_seed(0); v = m.sample(100)
sage: m.baum_welch(v,fix_emissions=True)
(-66.98630856918774, 100)
sage: m.emission_matrix()
[0.5 0.5]
[0.2 0.8]
sage: m = hmm.DiscreteHiddenMarkovModel([[0.1,0.9],[0.9,0.1]],
......: [[0.5,0.5],[0.2,0.8]],
......:
[0.2,0.8])
sage: m.baum_welch(v)
(-66.782360659293..., 100)
sage: m.emission_matrix()
# rel tol 1e-14
[ 0.5303085748626447 0.46969142513735535]
[ 0.2909775550173978 0.709022449826023]
```

```python
>>> from sage.all import *
>>> m = hmm.DiscreteHiddenMarkovModel([[RealNumber('0.1'),RealNumber('0.9')],
˓→[RealNumber('0.9'),RealNumber('0.1')]],
...
[[RealNumber('0.5'),RealNumber('0.5')],
...
[RealNumber('0.2'),RealNumber('0.8')]])
>>> set_random_seed(Integer(0)); v = m.sample(Integer(100))
>>> m.baum_welch(v,fix_emissions=True)
(-66.98630856918774, 100)
>>> m.emission_matrix()
[0.5 0.5]
[0.2 0.8]
>>> m = hmm.DiscreteHiddenMarkovModel([[RealNumber('0.1'),RealNumber('0.9')],
˓→[RealNumber('0.9'),RealNumber('0.1')]],
...
[[RealNumber('0.5'),RealNumber('0.5')],
...
[RealNumber('0.2'),RealNumber('0.8')]])
>>> m.baum_welch(v)
(-66.782360659293..., 100)
>>> m.emission_matrix()
# rel tol 1e-14
[ 0.5303085748626447 0.46969142513735535]
[ 0.2909775550173978 0.709022449826023]
```

**emission_matrix()**

Return the matrix whose \(i\)-th row specifies the emission probability distribution for the \(i\)-th state.

More precisely, the \(i,j\) entry of the matrix is the probability of the Markov model outputting the \(j\)-th symbol when it is in the \(i\)-th state.

**OUTPUT:** a Sage matrix with real double precision (RDF) entries.

**EXAMPLES:**
Statistics, Release 10.4

```python
>>> from sage.all import *
>>> m = hmm.DiscreteHiddenMarkovModel([[RealNumber('0.4'), RealNumber('0.6')],
                                           [RealNumber('0.1'), RealNumber('0.9')]],
                                           ...
                                           [[RealNumber('0.1'), RealNumber('0.9')],
                                           [RealNumber('0.5'), RealNumber('0.5')]])
>>> E = m.emission_matrix(); E
[[0.1 0.9]
 [0.5 0.5]]
```

The returned matrix is mutable, but changing it does not change the transition matrix for the model:

```python
sage: E[0,0] = 0; E[0,1] = 1
sage: m.emission_matrix()
[0.1 0.9]
[0.5 0.5]
```

```python
>>> from sage.all import *
>>> E[Integer(0),Integer(0)] = Integer(0); E[Integer(0),Integer(1)] = Integer(1)
>>> m.emission_matrix()
[[0.1 0.9]
 [0.5 0.5]]
```

**generate_sequence**(length, starting_state=None)

Return a sample of the given length from this HMM.

**INPUT:**

- **length** – positive integer
- **starting_state** – int (or None); if specified, generate a sequence using this model starting with the given state instead of the initial probabilities to determine the starting state.

**OUTPUT:**

- an IntList or list of emission symbols
- IntList of the actual states the model was in when emitting the corresponding symbols

**EXAMPLES:**

In this example, the emission symbols are not set:

```python
sage: set_random_seed(0)
sage: a = hmm.DiscreteHiddenMarkovModel([[0.1,0.9],[0.1,0.9]],
                                           [[1,0],[0,1]],
                                           [[0,1]])
sage: a.generate_sequence(5)
([1, 0, 1, 1, 1], [1, 0, 1, 1, 1])
sage: list(a.generate_sequence(1000)[0]).count(0)
90
```

```python
>>> from sage.all import *
>>> set_random_seed(Integer(0))
>>> a = hmm.DiscreteHiddenMarkovModel([[RealNumber('0.1'), RealNumber('0.9')],
                                           [RealNumber('0.1'), RealNumber('0.9')]],
                                           ...
                                           [Integer(1),Integer(0),Integer(0),
                                           Integer(1)])
```

(continues on next page)
Here the emission symbols are set:

```python
sage: set_random_seed(0)
sage: a = hmm.DiscreteHiddenMarkovModel([[0.5,0.5],[0.1,0.9]],
                                           [[1,0],[0,1]],
                                           [0,1], ['up', 'down'])
sage: a.generate_sequence(5)
(['down', 'up', 'down', 'down', 'down'], [1, 0, 1, 1, 1])
```

Specify the starting state:

```python
sage: set_random_seed(0); a.generate_sequence(5, starting_state=0)
(['up', 'up', 'down', 'down', 'down'], [0, 0, 1, 1, 1])
```

**log_likelihood**(obs, scale=True)

Return the logarithm of the probability that this model produced the given observation sequence. Thus the output is a non-positive number.

**INPUT:**

- obs = sequence of observations
- scale = boolean (default: True); if True, use rescaling to overid loss of precision due to the very limited dynamic range of floats. You should leave this as True unless the obs sequence is very small.

**EXAMPLES:**

```python
sage: m = hmm.DiscreteHiddenMarkovModel([[0.4,0.6],[0.1,0.9]],
                                          [[0.1,0.9],[0.5,0.5]],
                                          [.2,.8])
sage: m.log_likelihood([0, 1, 0, 1, 0, 1, 0, 0, 0])
-7.3301308009370825
sage: m.log_likelihood([0, 1, 0, 1, 0, 1, 0, 0, 0], scale=False)
-7.3301308009370825
sage: m.log_likelihood([])
0.0
```
sage: m = hmm.DiscreteHiddenMarkovModel([[0.4,0.6],[0.1,0.9]],
....:                                         [0.1,0.9],[0.5,0.5]),
....:                                         [.2,.8], ['happy','sad'])
sage: m.log_likelihood(['happy','happy'])
-1.6565295199679506
sage: m.log_likelihood(['happy','sad'])
-1.4731602941415523

>>> from sage.all import *
>>> m = hmm.DiscreteHiddenMarkovModel([[RealNumber('0.4'),RealNumber('0.6')],
                                      →[RealNumber('0.1'),RealNumber('0.9')],
                                      ...
                                      →[RealNumber('0.5'),RealNumber('0.5')]],
                                      ...
                                      →[RealNumber('.2'),RealNumber('.8')])
>>> m.log_likelihood([Integer(0), Integer(1), Integer(0), Integer(1),...
                      →Integer(1), Integer(0), Integer(1), Integer(0), Integer(0), Integer(0)])
-7.3301308009370825
>>> m.log_likelihood([Integer(0), Integer(1), Integer(0), Integer(1),...
                      →Integer(1), Integer(0), Integer(1), Integer(0), Integer(0), Integer(0)],...
                      →scale=False)
-7.330130800937082
>>> m.log_likelihood([])
0.0

>>> m = hmm.DiscreteHiddenMarkovModel([[RealNumber('0.4'),RealNumber('0.6')],
                                      →[RealNumber('0.1'),RealNumber('0.9')],
                                      ...
                                      →[RealNumber('0.5'),RealNumber('0.5')]],
                                      ...
                                      →'happy','sad'))
>>> m.log_likelihood(['happy','happy'])
-1.6565295199679506
>>> m.log_likelihood(['happy','sad'])
-1.4731602941415523

Overflow from not using the scale option:

sage: m = hmm.DiscreteHiddenMarkovModel([[0.4,0.6],[0.1,0.9]],
....:                                         [0.1,0.9],[0.5,0.5]),
....:                                         [.2,.8])
sage: m.log_likelihood([0,1]*1000, scale=True)
1433.82066652728
sage: m.log_likelihood([0,1]*1000, scale=False)
-inf

>>> from sage.all import *
>>> m = hmm.DiscreteHiddenMarkovModel([[RealNumber('0.4'),RealNumber('0.6')],
                                      →[RealNumber('0.1'),RealNumber('0.9')],
                                      ...
                                      →[RealNumber('0.5'),RealNumber('0.5')]],
                                      ...
                                      →['0.2','0.8'])
>>> m.log_likelihood([Integer(0),Integer(1)]*Integer(1000), scale=True)
1433.82066652728
>>> m.log_likelihood([Integer(0),Integer(1)]*Integer(1000), scale=False)
-inf
**viterbi**(obs, log_scale=True)

Determine “the” hidden sequence of states that is most likely to produce the given sequence seq of observations, along with the probability that this hidden sequence actually produced the observation.

**INPUT:**

- seq – sequence of emitted ints or symbols
- log_scale – bool (default: True) whether to scale the sequence in order to avoid numerical overflow.

**OUTPUT:**

- list – “the” most probable sequence of hidden states, i.e., the Viterbi path.
- float – log of probability that the observed sequence was produced by the Viterbi sequence of states.

**EXAMPLES:**

```python
sage: a = hmm.DiscreteHiddenMarkovModel([[0.1,0.9],[0.1,0.9]],
                                      [[0.9,0.1],[0.1,0.9]],
                                      [0.5,0.5])

sage: a.viterbi([1,0,0,1,0,0,1,1])
([1, 0, 0, 1, ..., 0, 1, 1], -11.06245322477221...)

>>> from sage.all import *

>>> a = hmm.DiscreteHiddenMarkovModel([[RealNumber(0.1),RealNumber(0.9)],
                                     [RealNumber(0.1),RealNumber(0.9)],
                                     [RealNumber(0.9),RealNumber(0.1)],
                                     [RealNumber(0.5),RealNumber(0.5)]])

>>> a.viterbi([Integer(1),Integer(0),Integer(0),Integer(1),Integer(0),
              Integer(0),Integer(1),Integer(1)])
([1, 0, 0, 1, ..., 0, 1, 1], -11.06245322477221...)
```

We predict the state sequence when the emissions are 3/4 and ‘abc’:

```python
sage: a = hmm.DiscreteHiddenMarkovModel([[0.1,0.9],[0.1,0.9]],
                                      [[0.9,0.1],[0.1,0.9]],
                                      [0.5,0.5], [3/4, 'abc'])

>>> from sage.all import *

>>> a.viterbi([3/4, 'abc', 'abc'] + [3/4]*10)
([0, 1, 1, 0, 0 ... 0, 0, 0, 0, 0], -25.299405845367794)
```

Note that state 0 is common below, despite the model trying hard to switch to state 1:

```python
([0, 1, 1, 0, 0 ... 0, 0, 0, 0, 0], -25.299405845367794)
```

```python
>>> from sage.all import *

>>> a.viterbi([Integer(3)/Integer(4), 'abc', 'abc'] + [Integer(3)/
                                     Integer(4)]*Integer(10))
([0, 1, 1, 0 ... 0, 0, 0, 0, 0], -25.299405845367794)
```
class sage.stats.hmm.HiddenMarkovModel
Bases: object

Abstract base class for all Hidden Markov Models.

graph (eps=0.001)
Create a weighted directed graph from the transition matrix, not including any edge with a probability less than eps.

INPUT:
• eps – nonnegative real number

OUTPUT: a DiGraph

EXAMPLES:

```python
sage: m = hmm.DiscreteHiddenMarkovModel([[.3,0,.7],[0,0,1],[.5,.5,0]],
....:  [[.5,.5,.2]]*3,
....:  [1/3]*3)

sage: G = m.graph(); G
Looped digraph on 3 vertices

sage: G.edges(sort=True)
[(0, 0, 0.3), (0, 2, 0.7), (1, 2, 1.0), (2, 0, 0.5), (2, 1, 0.5)]

sage: G.plot()
```

initial_probabilities()

Return the initial probabilities as a TimeSeries of length N, where N is the number of states of the Markov model.

EXAMPLES:

```python
sage: m = hmm.DiscreteHiddenMarkovModel([[0.4,0.6],[0.1,0.9]],
....:  [[0.1,0.9],[0.5,0.5]],
....:  [.2,.8])

sage: pi = m.initial_probabilities(); pi
[0.2000, 0.8000]
```

(continues on next page)
Statistics, Release 10.4

(continued from previous page)

```python
sage: type(pi)
<... 'sage.stats.time_series.TimeSeries'>
```

```python
>>> from sage.all import *

```python
>>> m = hmm.DiscreteHiddenMarkovModel([[RealNumber('0.4'),RealNumber('0.6')],
    →[RealNumber('0.1'),RealNumber('0.9')],
    ...
    [[RealNumber('0.1'),RealNumber('0.9')],
    →[RealNumber('0.5'),RealNumber('0.5')]],
    ...
    [RealNumber('0.2'),RealNumber('0.8')])
>>> pi = m.initial_probabilities(); pi
[0.2000, 0.8000]
>>> type(pi)
<... 'sage.stats.time_series.TimeSeries'>
```

The returned time series is a copy, so changing it does not change the model:

```python
sage: pi[0] = .1; pi[1] = .9
sage: m.initial_probabilities()
[0.2000, 0.8000]
```

```python
>>> from sage.all import *

```python
>>> pi[Integer(0)] = RealNumber('.1'); pi[Integer(1)] = RealNumber('.9')
>>> m.initial_probabilities()
[0.2000, 0.8000]
```

Some other models:

```python
sage: m = hmm.GaussianHiddenMarkovModel([[.1,.9],[.5,.5]],
    ....:[(1,1), (-1,1)],
    ....:[.1,.9])
sage: m.initial_probabilities()
[0.1000, 0.9000]
```

```python
sage: m = hmm.GaussianMixtureHiddenMarkovModel(
    ....:[[.9,.1],[.4,.6]],
    ....:[[.4,(0,1)), (.6,(1,0.1))], [(1,(0,1))]],
    ....:[.7,.3])
```

```python
sage: m.initial_probabilities()
[0.7000, 0.3000]
```

```python
>>> from sage.all import *

```python
>>> m = hmm.GaussianHiddenMarkovModel([[RealNumber('.1'),RealNumber('.9')],
    →[RealNumber('.5'),RealNumber('.5')]],
    ...
    [(Integer(1),Integer(1)), (-Integer(1),
    →Integer(1))],
    ...
    [RealNumber('.1'),RealNumber('.9')])
```

```python
>>> m = hmm.GaussianMixtureHiddenMarkovModel(
    ...
    [[RealNumber('.9'),RealNumber('.1')],[RealNumber('.4'),RealNumber{
    →'6')]],
    ...
    [[(RealNumber('.4'),(Integer(0),Integer(1))),(RealNumber('.6'),
    →(Integer(1),RealNumber('0.1')))], [(Integer(1), (Integer(0),Integer(1))))],
    ...
    [RealNumber('.7'),RealNumber('.3')])
```

```python
>>> m.initial_probabilities()
[0.7000, 0.3000]
```
sample (length, number=None, starting_state=None)

Return number samples from this HMM of given length.

INPUT:

- **length** – positive integer
- **number** – (default: None) if given, compute list of this many sample sequences
- **starting_state** – int (or None); if specified, generate a sequence using this model starting with the given state instead of the initial probabilities to determine the starting state.

OUTPUT:

- if number is not given, return a single TimeSeries.
- if number is given, return a list of TimeSeries.

EXAMPLES:
```
>>> from sage.all import *
>>> set_random_seed(Integer(0))
>>> a = hmm.DiscreteHiddenMarkovModel(
[RealNumber('0.5'),RealNumber('0.5')],
˓→[RealNumber('0.1'),RealNumber('0.9')]
...
[Integer(1),Integer(0)], [Integer(0),
˓→Integer(1)]), [Integer(0),Integer(1)],
...
[up', 'down'])
>>> a.sample(Integer(10))
['down', 'up', 'down', 'down', 'down', 'down', 'up', 'up', 'up', 'up']
```

Force a starting state:

```
sage: set_random_seed(0); a.sample(10, starting_state=0)
['up', 'up', 'down', 'down', 'down', 'down', 'up', 'up', 'up', 'up']
```

```
transition_matrix()
Return the state transition matrix.

OUTPUT: a Sage matrix with real double precision (RDF) entries.

EXAMPLES:
```
sage: M = hmm.DiscreteHiddenMarkovModel([[0.7,0.3],[0.9,0.1]],
˓→[[0.5,.5],[.1,.9]],
˓→[0.3,0.7])
sage: T = M.transition_matrix(); T
[0.7 0.3]
[0.9 0.1]
```

The returned matrix is mutable, but changing it does not change the transition matrix for the model:

```
sage: T[0,0] = .1; T[0,1] = .9
sage: M.transition_matrix()
[0.7 0.3]
[0.9 0.1]
```

```
Transition matrices for other types of models:

```python
sage: M = hmm.GaussianHiddenMarkovModel([[.1, .9], [.5, .5]],
                               [(1,1), (-1,1)],
                               [.5, .5])
sage: M.transition_matrix()
[0.1 0.9]
[0.5 0.5]
sage: M = hmm.GaussianMixtureHiddenMarkovModel(
                               [[.9, .1], [.4, .6]],
                               [[(0.4,(0,1)), (0.6,(0,0.1))], [(1,(0,1))]],
                               [.7, .3])
sage: M.transition_matrix()
[0.9 0.1]
[0.4 0.6]
```

```python
from sage.all import *

>>> M = hmm.GaussianHiddenMarkovModel([[RealNumber('1'),RealNumber('9')],
                                    [RealNumber('5'),RealNumber('5')]],
                                    [(Integer(1),Integer(1)), (-Integer(1),
                                    Integer(1))],
                                    [RealNumber('5'),RealNumber('5')])
>>> M.transition_matrix()
[0.1 0.9]
[0.5 0.5]

>>> M = hmm.GaussianMixtureHiddenMarkovModel(
                                    [[RealNumber('9'),RealNumber('1')],
                                    [RealNumber('4'),RealNumber('>6')]],
                                    [(RealNumber('4'),Integer(0),Integer(1))], (RealNumber('6'),
                                    (Integer(1),RealNumber('0.1'))),[(Integer(1),Integer(0),Integer(1))]],
                                    [RealNumber('7'),RealNumber('3')])
>>> M.transition_matrix()
[0.9 0.1]
[0.4 0.6]
```

`sage.stats.hmm.unpickle_discrete_hmm_v0 (A, B, pi, emission_symbols, name)`

`sage.stats.hmm.unpickle_discrete_hmm_v1 (A, B, pi, n_out, emission_symbols, emission_symbols_dict)`

Return a `DiscreteHiddenMarkovModel`, restored from the arguments.

This function is used internally for unpickling.
CHAPTER
FOUR

CONTINUOUS EMISSION HIDDEN MARKOV MODELS

AUTHOR:

• William Stein, 2010-03

class sage.stats.hmm.chmm.GaussianHiddenMarkovModel

Bases: HiddenMarkovModel

Gaussian emissions Hidden Markov Model.

INPUT:

• A – matrix; the $N \times N$ transition matrix
• B – list of pairs (mu, sigma) that define the distributions
• pi – initial state probabilities
• normalize – bool (default: True)

EXAMPLES:

We illustrate the primary functions with an example 2-state Gaussian HMM:

\[
\begin{pmatrix}
0.1 & 0.9 \\
0.5 & 0.5
\end{pmatrix}
\]

Emission parameters:

\[(1.0, 1.0), (-1.0, 1.0)\]

Initial probabilities: [0.5000, 0.5000]
We query the defining transition matrix, emission parameters, and initial state probabilities:

```python
sage: m.transition_matrix()
[0.1 0.9]
[0.5 0.5]
sage: m.emission_parameters()
[(1.0, 1.0), (-1.0, 1.0)]
sage: m.initial_probabilities()
[0.5000, 0.5000]
```

We obtain a sample sequence with 10 entries in it, and compute the logarithm of the probability of obtaining this sequence, given the model:

```python
sage: obs = m.sample(5); obs  # random
[-1.6835, 0.0635, -2.1688, 0.3043, -0.3188]
sage: log_likelihood = m.log_likelihood(obs)
sage: counter = 0
sage: n = 0
sage: def add_samples(i):
...    global counter, n
...    for _ in range(i):
...        n += 1
...        obs2 = m.sample(5)
...        if all(abs(obs2[i] - obs[i]) < 0.25 for i in range(5)):
...            counter += 1
sage: add_samples(10000)
```

```python
>>> from sage.all import *

>>> obs = m.sample(Integer(5)); obs  # random
[-1.6835, 0.0635, -2.1688, 0.3043, -0.3188]

>>> log_likelihood = m.log_likelihood(obs)

>>> counter = Integer(0)

>>> n = Integer(0)

>>> def add_samples(i):
...    global counter, n
...    for _ in range(Integer(5)):
...        n += Integer(1)
...        obs2 = m.sample(Integer(5))
...        if all(abs(obs2[i] - obs[i]) < RealNumber('0.25') for i in range(Integer(5))):
...            counter += Integer(1)

>>> add_samples(Integer(10000))
```

```python
>>> from sage.all import *

>>> obs = m.sample(Integer(5)); obs  # random
[-1.6835, 0.0635, -2.1688, 0.3043, -0.3188]

>>> log_likelihood = m.log_likelihood(obs)

>>> counter = Integer(0)

>>> n = Integer(0)

>>> def add_samples(i):
...    global counter, n
...    for _ in range(Integer(5)):
...        n += Integer(1)
...        obs2 = m.sample(Integer(5))
...        if all(abs(obs2[i] - obs[i]) < RealNumber('0.25') for i in range(Integer(5))):
...            counter += Integer(1)

>>> add_samples(Integer(10000))
```
We compute the Viterbi path, and probability that the given path of states produced obs:

```python
sage: m.viterbi(obs)  # random
([(1, 0, 1, 0, 1), -8.714092684611794])
```

```python
>>> from sage.all import *
>>> m.viterbi(obs)  # random
([(1, 0, 1, 0, 1), -8.714092684611794])
```

We use the Baum-Welch iterative algorithm to find another model for which our observation sequence is more likely:

```python
sage: try:
   ...:     p, s = m.baum_welch(obs)
   ...:     assert p > log_likelihood
   ...:     assert (1 <= s <= 500)
   ...: except RuntimeError:
   ...:     pass
```

```python
>>> from sage.all import *
>>> try:
...     p, s = m.baum_welch(obs)
...     assert p > log_likelihood
...     assert (Integer(1) <= s <= Integer(500))
... except RuntimeError:
...     pass
```

Notice that running Baum-Welch changed our model:

```python
sage: m  # random
Gaussian Hidden Markov Model with 2 States
Transition matrix:
[ 0.4154981366185841 0.584501863381416]
[ 0.9999993174253741 6.825746258991804e-07]
Emission parameters:
[(0.4178882427119503, 0.5173109664360919),
 (-1.502520831331122, 0.5085512836055119)]
Initial probabilities: [0.0000, 1.0000]
```

```python
>>> from sage.all import *
>>> m  # random
Gaussian Hidden Markov Model with 2 States
Transition matrix:
[ 0.4154981366185841 0.584501863381416]
[ 0.9999993174253741 6.825746258991804e-07]
Emission parameters:
[(0.4178882427119503, 0.5173109664360919),
 (-1.502520831331122, 0.5085512836055119)]
Initial probabilities: [0.0000, 1.0000]
```

**baum_welch** (obs, max_iter=500, log_likelihood_cutoff=0.0001, min_sd=0.01, fix_emissions=False, v=False)

Given an observation sequence obs, improve this HMM using the Baum-Welch algorithm to increase the probability of observing obs.

**INPUT:**

- **obs** – a time series of emissions
• `max_iter` – integer (default: 500) maximum number of Baum-Welch steps to take
• `log_likelihood_cutoff` – positive float (default: 1e-4); the minimal improvement in likelihood with respect to the last iteration required to continue. Relative value to log likelihood.
• `min_sd` – positive float (default: 0.01); when reestimating, the standard deviation of emissions is not allowed to be less than `min_sd`.
• `fix_emissions` – bool (default: False); if True, do not change emissions when updating

OUTPUT:
changes the model in place, and returns the log likelihood and number of iterations.

EXAMPLES:

```python
sage: m = hmm.GaussianHiddenMarkovModel([[.1,.9],[.5,.5]],
...: [(1,.5), (-1,3)],
...: [.1,.9])

sage: m.log_likelihood([-2,-1,.1,0.1])
-8.858282215986275
sage: m.baum_welch([-2,-1,.1,0.1])
(4.534646052182..., 7)
```

We illustrate bounding the standard deviation below. Note that above we had different emission parameters when the `min_sd` was the default of 0.01:
We watch the log likelihoods of the model converge, step by step:

```python
sage: m = hmm.GaussianHiddenMarkovModel([[.1,.9],[.5,.5]],
                      [(1,5), (-1,3)],
                      [.1,.9])
sage: v = m.sample(10)
sage: l = stats.TimeSeries([m.baum_welch(v, max_iter=1)[0]
                        for _ in range(len(v))])
sage: all(l[i] <= l[i+1] + 0.0001 for i in range(9))
True
sage: l  # random
[-20.1167, -17.7611, -16.9814, -16.9364, -16.9314,
 -16.9309, -16.9309, -16.9309, -16.9309, -16.9309]
```

We illustrate fixing emissions:

```python
sage: m = hmm.GaussianHiddenMarkovModel([[.1,.9],[.9,.1]],
                      [(1,2),(-1,.5)],
                      [.3,.7])
sage: set_random_seed(0); v = m.sample(100)
sage: m.baum_welch(v, fix_emissions=True)
```
emission_parameters()

Return the parameters that define the normal distributions associated to all of the states.

OUTPUT:

A list $B[i] = (\mu, \sigma)$, such that the distribution associated to state $i$ is normal with mean $\mu$ and standard deviation $\sigma$.

EXAMPLES:

```python
sage: M = hmm.GaussianHiddenMarkovModel([[.1,.9],[.5,.5]],
                      [(1,.5), (-1,3)],
                      [.1,.9])
sage: M.emission_parameters()
[(1.0, 0.5), (-1.0, 3.0)]
```
generate_sequence \( (\text{length, starting\_state}=\text{None}) \)

Return a sample of the given length from this HMM.

**INPUT:**
- \( \text{length} \) – positive integer
- \( \text{starting\_state} \) – int (or None); if specified then generate a sequence using this model starting with the given state instead of the initial probabilities to determine the starting state.

**OUTPUT:**
- an \texttt{IntList} or list of emission symbols
- \texttt{TimeSeries} of emissions

**EXAMPLES:**

```
sage: m = hmm.GaussianHiddenMarkovModel([[.1,.9],[.5,.5]],
                                            [[1,.5], (-1,3)],
                                            [.1,.9])
sage: m.generate_sequence(5)  # random
([[-3.0505, 0.5317, -4.5065, 0.6521, 1.0435], [1, 0, 1, 0, 1])
```

```
sage: m.generate_sequence(0)
((), (])
```

```
sage: m.generate_sequence(-1)
Traceback (most recent call last):
  ...
ValueError: length must be nonnegative
```

```
>>> from sage.all import *

>>> m = hmm.GaussianHiddenMarkovModel([[RealNumber('.1'),RealNumber('.9')],
                                           [RealNumber('.5'),RealNumber('.5')]])
>>> m.generate_sequence(Integer(5))  # random
(([-3.0505, 0.5317, -4.5065, 0.6521, 1.0435], [1, 0, 1, 0, 1])
```

```
>>> m.generate_sequence(Integer(0))
((), (])
```

```
>>> m.generate_sequence(-Integer(1))
Traceback (most recent call last):
  ...
ValueError: length must be nonnegative
```

Verify numerically that the starting state is 0 with probability about 0.1:

```
sage: counter = 0
sage: n = 0
sage: def add_samples(i):
    ...:     global counter, n
    ...:     for i in range(i):
    ...:         n += 1
    ...:     if m.generate_sequence(1)[1][0] == 0:
    ...:         counter += 1
```
sage: add_samples(10^5)
sage: while abs(counter*1.0 / n - 0.1) > 0.01: add_samples(10^5)

```python
>>> from sage.all import *

>>> counter = Integer(0)
>>> n = Integer(0)
>>> def add_samples(i):
...     global counter, n
...     for i in range(i):
...         n += Integer(1)
...     if m.generate_sequence(Integer(1))[Integer(1)][Integer(0)] ==
...         Integer(0):
...         counter += Integer(1)

>>> add_samples(Integer(10)**Integer(5))
>>> while abs(counter*RealNumber('1.0') / n - RealNumber('0.1')) > RealNumber('0.01'):
...     add_samples(Integer(10)**Integer(5))
```

Example in which the starting state is 0 (see Issue #11452):

```
sage: set_random_seed(23); m.generate_sequence(2)
([0.6501, -2.0151], [0, 1])
```

```python
>>> from sage.all import *

>>> set_random_seed(Integer(23)); m.generate_sequence(Integer(2))
([0.6501, -2.0151], [0, 1])
```

Force a starting state of 1 even though as we saw above it would be 0:

```
sage: set_random_seed(23); m.generate_sequence(2, starting_state=1)
([-3.1491, -1.0244], [1, 1])
```

```python
>>> set_random_seed(Integer(23)); m.generate_sequence(Integer(2), starting_state=Integer(1))
([-3.1491, -1.0244], [1, 1])
```

**log_likelihood**(obs)

Return the logarithm of a continuous analogue of the probability that this model produced the given observation sequence.

Note that the “continuous analogue of the probability” above can be bigger than 1, hence the logarithm can be positive.

**INPUT:**

- obs – sequence of observations

**OUTPUT:**

float

**EXAMPLES:**

```
sage: m = hmm.GaussianHiddenMarkovModel([[.1,.9],[.5,.5]],
....:                        [(1,.5), (-1,3)],
...)
```
.. source:: sage: m.log_likelihood([1,1,1])
-4.297880766072486
sage: s = m.sample(20)
sage: -80 < m.log_likelihood(s) < -20
True

```python
>>> from sage.all import *

>>> m = hmm.GaussianHiddenMarkovModel([[RealNumber('.1'), RealNumber('.9')],
    ... [RealNumber('.5'), RealNumber('.5')]],
    ... [[Integer(1), RealNumber('.5')],
    ... [RealNumber('.1'), RealNumber('.9')]])

>>> m.log_likelihood([Integer(1), Integer(1), Integer(1)])
-4.297880766072486
>>> s = m.sample(Integer(20))
>>> -Integer(80) < m.log_likelihood(s) < -Integer(20)
True
```

\textbf{viterbi} \textit{(obs)}

Determine “the” hidden sequence of states that is most likely to produce the given sequence \textit{obs} of observations, along with the probability that this hidden sequence actually produced the observation.

\textbf{INPUT:}

\begin{itemize}
    \item \textit{obs} – sequence of emitted ints or symbols
\end{itemize}

\textbf{OUTPUT:}

\begin{itemize}
    \item list – “the” most probable sequence of hidden states, i.e., the Viterbi path.
    \item float – log of probability that the observed sequence was produced by the Viterbi sequence of states.
\end{itemize}

\textbf{EXAMPLES:}

We find the optimal state sequence for a given model:

```python
sage: m = hmm.GaussianHiddenMarkovModel([[0.5, 0.5], [0.5, 0.5]],
    ... [(0, 1), (10, 1)],
    ... [0.5, 0.5])
sage: m.viterbi([0, 1, 10, 10, 1])
([0, 0, 1, 1, 0], -9.0604285688230...)
```

```python
>>> from sage.all import *

>>> m = hmm.GaussianHiddenMarkovModel([[RealNumber('0.5'), RealNumber('0.5')],
    ... [RealNumber('0.5'), RealNumber('0.5')]],
    ... [[Integer(0), Integer(1)],
    ... [RealNumber('0.5'), RealNumber('0.5')]])

>>> m.viterbi([Integer(0), Integer(1), Integer(10), Integer(10), Integer(1)])
([0, 0, 1, 1, 0], -9.0604285688230...)
```

Another example in which the most likely states change based on the last observation:

```python
sage: m = hmm.GaussianHiddenMarkovModel([[.1,.9],[.5,.5]],
    ... [(1,.5), (-1,3)],
    ... [.1,.9])
sage: m.viterbi([-2,-1,.1,0.1])
```

(continues on next page)
Statistics, Release 10.4

((continued from previous page))

```python
sage: m.viterbi([-2,-1,1,0.3])
([(1, 1, 1, 0), -9.566023653378513])
```

```python
>>> from sage.all import *

>>> m = hmm.GaussianHiddenMarkovModel([[RealNumber('.1'),RealNumber('.9')],
       [RealNumber('.5'),RealNumber('.5')]],
       [(Integer(1),RealNumber('.5'))],
       [(RealNumber('.1'),RealNumber('.9'))])

>>> m.viterbi([-Integer(2),-Integer(1),RealNumber('.1'),RealNumber(0.3)])
([(1, 1, 1, 0), -9.566023653378513])
```

```python
class sage.stats.hmm.chmm.GaussianMixtureHiddenMarkovModel

Bases: GaussianHiddenMarkovModel

Gaussian mixture Hidden Markov Model.

INPUT:

- A – matrix; the \(N \times N\) transition matrix
- B – list of mixture definitions for each state. Each state may have a varying number of gaussians with selection probabilities that sum to 1 and encoded as \((p, (mu, sigma))\)
- pi – initial state probabilities
- normalize – bool (default: True); if given, input is normalized to define valid probability distributions, e.g., the entries of \(A\) are made nonnegative and the rows sum to 1, and the probabilities in \(pi\) are normalized.

EXAMPLES:

```python
sage: A = [[0.5,0.5],[0.5,0.5]]
sage: B = [[(0.9,(0.0,1.0)), (0.1,(1,10000))],[(1,(1,1)), (0,(0,0.1))]]
sage: hmm.GaussianMixtureHiddenMarkovModel(A, B, [1,0])
Gaussian Mixture Hidden Markov Model with 2 States
Transition matrix:
[0.5 0.5]
[0.5 0.5]
Emission parameters:
[0.9*N(0,0,1.0) + 0.1*N(1.0,10000.0), 1.0*N(1.0,1.0) + 0.0*N(0.0,0.1)]
Initial probabilities: [1.0000, 0.0000]
```

```python
>>> A = [[RealNumber('0.5'),RealNumber('0.5')],
       [RealNumber('0.5'),RealNumber('0.5')]]

>>> B = [[(RealNumber('0.9'), (RealNumber('0.0'),RealNumber('1.0'))), (RealNumber('0.1'), (Integer(1),Integer(10000)))]
       ,[(Integer(1),Integer(1)), (Integer(0),RealNumber('0.1'))]]

>>> hmm.GaussianMixtureHiddenMarkovModel(A, B, [Integer(1),Integer(0)])
Gaussian Mixture Hidden Markov Model with 2 States
Transition matrix:
[0.5 0.5]
[0.5 0.5]
Emission parameters:
```

(continues on next page)

40 Chapter 4. Continuous Emission Hidden Markov Models
\[0.9*N(0.0,1.0) + 0.1*N(1.0,10000.0), 1.0*N(1.0,1.0) + 0.0*N(0.0,0.1)]\]

Initial probabilities: [1.0000, 0.0000]

**baum_welch** *(obs, max_iter=1000, log_likelihood_cutoff=1e-12, min_sd=0.01, fix_emissions=False)*

Given an observation sequence `obs`, improve this HMM using the Baum-Welch algorithm to increase the probability of observing `obs`.

**INPUT:**

- `obs` – a time series of emissions
- `max_iter` – integer (default: 1000) maximum number of Baum-Welch steps to take
- `log_likelihood_cutoff` – positive float (default: 1e-12); the minimal improvement in likelihood with respect to the last iteration required to continue. Relative value to log likelihood.
- `min_sd` – positive float (default: 0.01); when reestimating, the standard deviation of emissions is not allowed to be less than `min_sd`.
- `fix_emissions` – bool (default: False); if True, do not change emissions when updating

**OUTPUT:**

changes the model in place, and returns the log likelihood and number of iterations.

**EXAMPLES:**

```python
sage: m = hmm.GaussianMixtureHiddenMarkovModel(
    ...: [[.9,.1], [.4,.6]],
    ...: [[.4,(0,1)], (.6,(1,0.1))], [(1,(0,1))],
    ...: [.7,.3])
sage: set_random_seed(0); v = m.sample(10); v
[0.3576, -0.9365, 0.9449, -0.6957, 1.0217,
  0.9644, 0.9987, -0.5950, -1.0219, 0.6477]
sage: m.log_likelihood(v)
-8.31408655939536...
sage: m.baum_welch(v)
(2.18905068682..., 15)
sage: m.log_likelihood(v)
2.18905068682...
sage: m  # rel tol 6e-12
Gaussian Mixture Hidden Markov Model with 2 States
Transition matrix:
[ 0.8746363339773399 0.12536366602266016]
[ 1.0 1.451685202290174e-40]
Emission parameters:
[0.50161629343*N(-0.812298726239,0.173329026744)
 + 0.499838370657*N(0.982433690378,0.029719932009),
  1.0*N(0.503260056832,0.145881515324)]
Initial probabilities: [0.0000, 1.0000]
```

```python
>>> from sage.all import *
>>> m = hmm.GaussianMixtureHiddenMarkovModel(
...    ...: [[RealNumber('.9'),RealNumber('.1')], [RealNumber('.4'),RealNumber('→'.6')]],
...    ...: [[RealNumber('.4'),(Integer(0),Integer(1))], (RealNumber('.6'),
...    →(Integer(1),RealNumber('→'.0.1)))]}, [(Integer(1),(Integer(0),Integer(1)))],
...    ...: [RealNumber('.7'),RealNumber('→'.3')]]
>>> set_random_seed(Integer(0)); v = m.sample(Integer(10)); v
```

(continues on next page)
We illustrate bounding the standard deviation below. Note that above we had different emission parameters when the \( \text{min\_sd} \) was the default of 0.01:

```python
sage: m = hmm.GaussianMixtureHiddenMarkovModel(
    ...: [[.9,.1],[.4,.6]],
    ...: [[(0,1),(0,1)], [(0,1),(0,1)]],
    ...: [.7,.3])
```

```python
sage: m.baum_welch(v, min_sd=1)
(-12.617885761692..., 1000)
```

```python
sage: m.emission_parameters()
# rel tol 6e-12
[0.50354563447*N(0.0,1.0) + 0.6*N(0.0,0.1), 1.0*N(0.0543433426535,1.0)]
```

We illustrate fixing all emissions:

```python
sage: m = hmm.GaussianMixtureHiddenMarkovModel(
    ...: [[.9,.1],[.4,.6]],
    ...: [[(0,1),(0,1)], [(0,1),(0,1)]],
    ...: [.7,.3])
```

```python
sage: set_random_seed(0); v = m.sample(10)
```

```python
sage: m.baum_welch(v, fix_emissions=True)
(-7.58656858997..., 36)
```

```python
sage: m.emission_parameters()
[0.4*N(0.0,1.0) + 0.6*N(1.0,0.1),
  1.0*N(0.0543433426535,1.0)]
```
>>> from sage.all import *

>>> m = hmm.GaussianMixtureHiddenMarkovModel(
...        [[RealNumber(.9), RealNumber(‘.1’)], [RealNumber(‘.4’), RealNumber(‘.6’)]],
...        [[(RealNumber(.4), (Integer(0), Integer(1))), (RealNumber(.6),
...             (Integer(1), RealNumber(‘.0’)))], [(Integer(1), (Integer(0), Integer(1)))],
...        [RealNumber(‘.7’), RealNumber(‘.3’)]

>>> set_random_seed(Integer(0)); v = m.sample(Integer(10))

>>> m.baum_welch(v, fix_emissions=True)

(-7.58656858997..., 36)

>>> m.emission_parameters()

[0.4*N(0.0,1.0) + 0.6*N(1.0,0.1), 1.0*N(0.0,1.0)]

emission_parameters()

Returns a list of all the emission distributions.

OUTPUT:

list of Gaussian mixtures

EXAMPLES:

sage: m = hmm.GaussianMixtureHiddenMarkovModel([[.9,.1], [.4,.6]],
...        [[(1, (0,1))]],
...        [0.4*N(0,0,1.0) + 0.6*N(1,0,0.1), 1.0*N(0,0,1.0)]

>>> from sage.all import *

>>> m = hmm.GaussianMixtureHiddenMarkovModel([[RealNumber(‘.9’), RealNumber(‘.1’)]],
...        [[(RealNumber(‘.4’), (Integer(0), Integer(1))), (RealNumber(‘.6’),
...             (Integer(1), RealNumber(‘.0’)))], [(Integer(1), (Integer(0), Integer(1)))],
...        [RealNumber(‘.7’), RealNumber(‘.3’)]

>>> m.emission_parameters()

[0.4*N(0,0,1.0) + 0.6*N(1,0,0.1), 1.0*N(0,0,1.0)]

sage.stats.hmm.chmm.unpickle_gaussian_hmm_v0(A, B, pi, name)

EXAMPLES:

sage: m = hmm.GaussianHiddenMarkovModel([[1]], [(0,1)], [1])
sage: m = sage.stats.hmm.chmm.unpickle_gaussian_hmm_v0(m.transition_matrix(), m.
...    emission_parameters(), m.initial_probabilities(), 'test')

Gaussian Hidden Markov Model with 1 States

Transition matrix:
[1.0]

Emission parameters:
[(0.0, 1.0)]

Initial probabilities: [1.0000]
```python
>>> from sage.all import *
>>> m = hmm.GaussianHiddenMarkovModel([[Integer(1)]], [(Integer(0),Integer(1))],
   ... [Integer(1)])
>>> sage.stats.hmm.chmm.unpickle_gaussian_hmm_v0(m.transition_matrix(), m.
   ... emission_parameters(), m.initial_probabilities(), 'test')
Gaussian Hidden Markov Model with 1 States
Transition matrix:
[1.0]
Emission parameters:
[(0.0, 1.0)]
Initial probabilities: [1.0000]
```

```python
sage.stats.hmm.chmm.unpickle_gaussian_hmm_v1(A, B, pi, prob, n_out)
EXAMPLES:
```sage: m = hmm.GaussianHiddenMarkovModel([[1]], [(0,1)], [1])
sage: loads(dumps(m)) == m # indirect test
True
```sage: m = hmm.GaussianHiddenMarkovModel([[Integer(1)]], [(Integer(0),Integer(1))],
   ... [Integer(1)])
>>> loads(dumps(m)) == m # indirect test
True
```sage.stats.hmm.chmm.unpickle_gaussian_mixture_hmm_v1(A, B, pi, mixture)
EXAMPLES:
```sage: m = hmm.GaussianMixtureHiddenMarkovModel([[1]], [[.4,(0,1)], (.6,(1,.0.
   ... 1))]], [1])
sage: loads(dumps(m)) == m # indirect test
True
```sage: m = hmm.GaussianMixtureHiddenMarkovModel([[Integer(1)]], [[RealNumber('0.4'),
   ... (Integer(0),Integer(1))], (RealNumber('0.6'),(Integer(1),RealNumber('0.1')))],
   ... [Integer(1)])
>>> loads(dumps(m)) == m # indirect test
True
```

Chapter 4. Continuous Emission Hidden Markov Models
DISTRIBUTIONS USED IN IMPLEMENTING HIDDEN MARKOV MODELS

These distribution classes are designed specifically for HMM's and not for general use in statistics. For example, they have fixed or non-fixed status, which only make sense relative to being used in a hidden Markov model.

AUTHOR:

• William Stein, 2010-03

```python
class sage.stats.hmm.distributions.DiscreteDistribution
    Bases: Distribution

class sage.stats.hmm.distributions.Distribution
    Bases: object
    A distribution.

    plot(*args, **kwds)
    Return a plot of the probability density function.

    INPUT:
    • args and kwds, passed to the Sage plot() function

    OUTPUT:
    • a Graphics object
```

EXAMPLES:

```python
sage: P = hmm.GaussianMixtureDistribution([(.2,-10,.5),(.6,1,1),(.2,20,.5)])
sage: P.plot(-10,30)  # needs sage.plot
Graphics object consisting of 1 graphics primitive

>>> from sage.all import *
>>> P = hmm.GaussianMixtureDistribution([(RealNumber('.2'),-Integer(10),
                                         RealNumber('.5')),(RealNumber('.6'),Integer(1),Integer(1)),(RealNumber('.2
                                         '),Integer(20),RealNumber('.5'))])
>>> P.plot(-Integer(10),Integer(30))  # needs sage.plot
Graphics object consisting of 1 graphics primitive
```

```python
prob(x)
    The probability density function evaluated at x.

    INPUT:
```
• x – object

OUTPUT:
• float

EXAMPLES:
This method must be defined in a derived class:

```python
sage: import sage.stats.hmm.distributions
sage: sage.stats.hmm.distributions.Distribution().prob(0)
Traceback (most recent call last):
...
NotImplementedError
```

```python
>>> from sage.all import *
>>> import sage.stats.hmm.distributions
>>> sage.stats.hmm.distributions.Distribution().prob(Integer(0))
Traceback (most recent call last):
...
NotImplementedError
```

```python
sage: import sage.stats.hmm.distributions
sage: sage.stats.hmm.distributions.Distribution().sample()
Traceback (most recent call last):
...
NotImplementedError
```

```python
>>> from sage.all import *
>>> import sage.stats.hmm.distributions
>>> sage.stats.hmm.distributions.Distribution().sample()
Traceback (most recent call last):
...
NotImplementedError
```

```python
class sage.stats.hmm.distributions.GaussianDistribution
Bases: Distribution
```

```python
class sage.stats.hmm.distributions.GaussianMixtureDistribution
Bases: Distribution
```

A probability distribution defined by taking a weighted linear combination of Gaussian distributions.

EXAMPLES:
Statistics, Release 10.4

sage: P = hmm.GaussianMixtureDistribution([(0.3, 1, 2), (0.7, -1, 1)]); P
0.3*N(1.0, 2.0) + 0.7*N(-1.0, 1.0)

sage: P[0]
(0.3, 1.0, 2.0)

sage: P.is_fixed()
False

sage: P.fix(0)

sage: P.is_fixed(0)
True

sage: P.is_fixed(1)
False

sage: P.unfix(1)

sage: P.is_fixed(1)
False

>>> from sage.all import *

>>> P = hmm.GaussianMixtureDistribution([(RealNumber('.3'), Integer(1), Integer(2)),
                                       (RealNumber('.7'), -Integer(1), Integer(1))]); P
0.3*N(1.0, 2.0) + 0.7*N(-1.0, 1.0)

>>> P[Integer(0)]
(0.3, 1.0, 2.0)

>>> P.is_fixed()
False

>>> P.fix(Integer(1))

>>> P.is_fixed(Integer(0))
False

>>> P.is_fixed(Integer(1))
True

>>> P.unfix(Integer(1))

fix(i=None)

Set that this GaussianMixtureDistribution (or its i-th component) is fixed when using Baum-Welch to update the corresponding HMM.

INPUT:

• i – None (default) or integer; if given, only fix the i-th component

EXAMPLES:

sage: P = hmm.GaussianMixtureDistribution([(0.2, -10, .5), (0.6, 1, 1), (0.2, 20, .5)])

sage: P.fix(1); P.is_fixed()
False

sage: P.is_fixed(1)
True

sage: P.is_fixed(0)
False

sage: P.unfix(1)

>>> from sage.all import *

>>> P = hmm.GaussianMixtureDistribution([(RealNumber('.2'), -Integer(10),
                                       RealNumber('.5')), (RealNumber('.6'), Integer(1), Integer(1)), (RealNumber('.2
                                       .5')), Integer(20), RealNumber('.5'))]); P
0.3*N(1.0, 2.0) + 0.7*N(-1.0, 1.0)

>>> P.fix(Integer(1)); P.is_fixed()
False

>>> P.is_fixed(Integer(1))
True

(continues on next page)
is_fixed \(i=\text{None}\)

Return whether or not this `GaussianMixtureDistribution` is fixed when using Baum-Welch to update the corresponding HMM.

INPUT:

- \(i\) – None (default) or integer; if given, only return whether the \(i\)-th component is fixed

EXAMPLES:

```python
sage: P = hmm.GaussianMixtureDistribution([(0.2,-10,0.5), (0.6,1,1), (0.2,20,0.5)])
sage: P.is_fixed()
False
sage: P.is_fixed(0)
False
sage: P.fix(0); P.is_fixed()
False
```

```python
from sage.all import *
```
The function `prob_m(x, m)` returns the probability of `x` using just the `m`-th summand.

**INPUT:**
- `x` – float
- `m` – integer

**OUTPUT:**
- float

**EXAMPLES:**

```
sage: P = hmm.GaussianMixtureDistribution([(0.2, -10, 0.5), (0.6, 1, 1), (0.2, 20, 0.5)])
sage: P.prob_m(0.5, 0)
2.7608117680508...e-97
sage: P.prob_m(0.5, 1)
0.21123919605857971
sage: P.prob_m(0.5, 2)
0.0
```

The function `sample(n=None)` returns a single sample from this distribution (by default), or if `n > 1`, return a `TimeSeries` of samples.

**INPUT:**
- `n` – integer or None (default: None)
OUTPUT:

- float if \( n \) is None (default); otherwise a TimeSeries

EXAMPLES:

```python
sage: P = hmm.GaussianMixtureDistribution([[.2,-10,.5],[.6,1,1],[.2,20,.5]])
sage: type(P.sample())
<class 'float'>
sage: l = P.sample(1)
sage: len(l)
1
sage: type(l)
<class 'sage.stats.time_series.TimeSeries'>
sage: l = P.sample(5)
sage: len(l)
5
sage: type(l)
<class 'sage.stats.time_series.TimeSeries'>
sage: l = P.sample(0)

Traceback (most recent call last):
... ValueError: n must be nonnegative

>>> from sage.all import *
>>> P = hmm.GaussianMixtureDistribution([(RealNumber('.2'),-Integer(10),
RealNumber('.5')),(RealNumber('.6'),Integer(1),Integer(1)),(RealNumber('.2
1'),Integer(20),RealNumber('.5'))])
>>> type(P.sample())
<class 'float'>
>>> l = P.sample(Integer(1))
>>> len(l)
1
>>> type(l)
<class 'sage.stats.time_series.TimeSeries'>
>>> l = P.sample(Integer(5))
>>> len(l)
5
>>> type(l)
<class 'sage.stats.time_series.TimeSeries'>
>>> l = P.sample(Integer(0))
>>> len(l)
0
>>> type(l)
<class 'sage.stats.time_series.TimeSeries'>
>>> P.sample(-Integer(3))
Traceback (most recent call last):
... ValueError: n must be nonnegative

unfix \((i=None)\)

Set that this GaussianMixtureDistribution (or its \(i\)-th component) is not fixed when using Baum-Welch to update the corresponding HMM.
INPUT:

- `i` — None (default) or integer; if given, only fix the `i`-th component

EXAMPLES:

```python
sage: P = hmm.GaussianMixtureDistribution([(.2,-10,.5),(.6,1,1),(.2,20,.5)])
sage: P.fix(1); P.is_fixed(1)
True
sage: P.unfix(1); P.is_fixed(1)
False
sage: P.fix(); P.is_fixed()
True
sage: P.unfix(); P.is_fixed()
False
```

```python
>>> from sage.all import *

>>> P = hmm.GaussianMixtureDistribution([RealNumber(.2),-Integer(10),
                                        RealNumber(.5)),
                                        RealNumber(.6),Integer(1),Integer(1)),
                                        RealNumber(.2 ,
                                        Integer(20),RealNumber(.5)])

>>> P.fix(Integer(1)); P.is_fixed(Integer(1))
True

>>> P.unfix(Integer(1)); P.is_fixed(Integer(1))
False

>>> P.fix(); P.is_fixed()
True

>>> P.unfix(); P.is_fixed()
False
```

```
sage.stats.hmm.distributions.unpickle_gaussian_mixture_distribution_v1(c0, cl, param, fixed)
```

Used in unpickling `GaussianMixtureDistribution` objects.

EXAMPLES:

```python
sage: P = hmm.GaussianMixtureDistribution([(.2,-10,.5),(.6,1,1),(.2,20,.5)])
sage: loads(dumps(P)) == P  # indirect doctest
True
```

```python
>>> from sage.all import *

>>> P = hmm.GaussianMixtureDistribution([RealNumber(.2),-Integer(10),
                                        RealNumber(.5)),
                                        RealNumber(.6),Integer(1),Integer(1)),
                                        RealNumber(.2 ,
                                        Integer(20),RealNumber(.5)])

>>> loads(dumps(P)) == P  # indirect doctest
True
```
HIDDEN MARKOV MODELS – UTILITY FUNCTIONS

AUTHOR:

- William Stein, 2010-03

```python
class sage.stats.hmm.util.HMM_Util
    Bases: object

    A class used in order to share cdef’s methods between different files.

    initial_probs_to_TimeSeries(pi, normalize)
        This function is used internally by the __init__ methods of various Hidden Markov Models.
        INPUT:
            • pi – vector, list, or TimeSeries
            • normalize – if True, replace negative entries by 0 and rescale to ensure that the sum of the entries
              in each row is equal to 1. If the sum of the entries in a row is 0, replace them all by 1/N.
        OUTPUT:
            • a TimeSeries of length N

    EXAMPLES:

    sage: import sage.stats.hmm.util
    sage: u = sage.stats.hmm.util.HMM_Util()
    sage: u.initial_probs_to_TimeSeries([0.1,0.2,0.9], True)
    [0.0833, 0.1667, 0.7500]
    sage: u.initial_probs_to_TimeSeries([0.1,0.2,0.9], False)
    [0.1000, 0.2000, 0.9000]

    normalize_probability_TimeSeries(T, i, j)
        This function is used internally by the Hidden Markov Models code.
```
Replace entries of $T[i:j]$ in place so that they are all nonnegative and sum to 1. Negative entries are replaced by 0 and $T[i:j]$ is then rescaled to ensure that the sum of the entries in each row is equal to 1. If all entries are 0, replace them by $1/(j-i)$.

**INPUT:**
- $T$ – a TimeSeries
- $i$ – nonnegative integer
- $j$ – nonnegative integer

**OUTPUT:**
- $T$ is modified

**EXAMPLES:**

```python
sage: import sage.stats.hmm.util
sage: T = stats.TimeSeries([.1, .3, .7, .5])
sage: u = sage.stats.hmm.util.HMM_Util()
sage: u.normalize_probability_TimeSeries(T,0,3)
sage: T
[0.0909, 0.2727, 0.6364, 0.5000]
sage: u.normalize_probability_TimeSeries(T,0,4)
sage: T
[0.0606, 0.1818, 0.4242, 0.3333]
sage: abs(T.sum()-1) < 1e-8  # might not exactly equal 1 due to rounding
True
```

```python
>>> from sage.all import *
>>> import sage.stats.hmm.util
>>> T = stats.TimeSeries([RealNumber(.1), RealNumber(.3), RealNumber(.7), RealNumber(.5)])
>>> u = sage.stats.hmm.util.HMM_Util()
>>> u.normalize_probability_TimeSeries(T,Integer(0),Integer(3))
>>> T
[0.0909, 0.2727, 0.6364, 0.5000]
>>> u.normalize_probability_TimeSeries(T,Integer(0),Integer(4))
>>> T
[0.0606, 0.1818, 0.4242, 0.3333]
>>> abs(T.sum()-Integer(1)) < RealNumber('1e-8')  # might not exactly equal
1 due to rounding
True
```

**state_matrix_to_TimeSeries** $(A, N, normalize)$

This function is used internally by the `__init__` methods of Hidden Markov Models to make a transition matrix from $A$.

**INPUT:**
- $A$ – matrix, list, list of lists, or TimeSeries
- $N$ – number of states
- $normalize$ – if True, replace negative entries by 0 and rescale to ensure that the sum of the entries in each row is equal to 1. If the sum of the entries in a row is 0, replace them all by $1/N$.

**OUTPUT:**
- a TimeSeries

**EXAMPLES:**
sage: import sage.stats.hmm.util
sage: u = sage.stats.hmm.util.HMM_Util()

sage: u.state_matrix_to_TimeSeries([[.1,.7],[3/7,4/7]], 2, True)
[0.1250, 0.8750, 0.4286, 0.5714]

sage: u.state_matrix_to_TimeSeries([[.1,.7],[3/7,4/7]], 2, False)
[0.1000, 0.7000, 0.4286, 0.5714]

>>> from sage.all import *

>>> import sage.stats.hmm.util

>>> u = sage.stats.hmm.util.HMM_Util()

>>> u.state_matrix_to_TimeSeries([[RealNumber('.1'),RealNumber('.7')],
                                [Integer(3)/Integer(7),Integer(4)/Integer(7)]], Integer(2), True)
[0.1250, 0.8750, 0.4286, 0.5714]

>>> u.state_matrix_to_TimeSeries([[RealNumber('.1'),RealNumber('.7')],
                                [Integer(3)/Integer(7),Integer(4)/Integer(7)]], Integer(2), False)
[0.1000, 0.7000, 0.4286, 0.5714]
CHAPTER
SEVEN

DISCRETE GAUSSIAN SAMPLERS OVER THE INTEGERS

This class realizes oracles which returns integers proportionally to \(\exp\left(-\frac{(x - c)^2}{2}\right)\). All oracles are implemented using rejection sampling. See \texttt{DiscreteGaussianDistributionIntegerSampler.__init__()} for which algorithms are available.

AUTHORS:

• Martin Albrecht (2014-06-28): initial version

EXAMPLES:

We construct a sampler for the distribution \(D_{3,c}\) with width \(3\) and center \(c = 0\):

\begin{verbatim}
from sage.stats.distributions.discrete_gaussian_integer import
...DiscreteGaussianDistributionIntegerSampler
sage: sigma = 3.0
sage: D = DiscreteGaussianDistributionIntegerSampler(sigma=sigma)
\end{verbatim}

We ask for 100000 samples:

\begin{verbatim}
sage: from collections import defaultdict
sage: counter = defaultdict(Integer)
sage: n = 0
sage: def add_samples(i):
... global counter, n
... for _ in range(i):
... counter[D()] += 1
... n += 1
sage: add_samples(100000)
\end{verbatim}

(continues on next page)
These are sampled with a probability proportional to $\exp(-x^2/18)$. More precisely we have to normalise by dividing by the overall probability over all integers. We use the fact that hitting anything more than 6 standard deviations away is very unlikely and compute:

```python
sage: bound = (6*sigma).floor()
sage: norm_factor = sum([exp(-x^2/(2*sigma^2)) for x in range(-bound,bound+1)])
```

With this normalisation factor, we can now test if our samples follow the expected distribution:

```python
sage: expected = lambda x : ZZ(round(n*exp(-x^2/(2*sigma^2))/norm_factor))
sage: observed = lambda x : counter[x]
```

We construct an instance with a larger width:

```python
sage: from sage.stats.distributions.discrete_gaussian_integer import DiscreteGaussianDistributionIntegerSampler
sage: sigma = 127
sage: D = DiscreteGaussianDistributionIntegerSampler(sigma=sigma, algorithm='uniform+online')
```

(continues on next page)
ask for 100000 samples:

```python
sage: from collections import defaultdict
sage: counter = defaultdict(Integer)
sage: n = 0
sage: def add_samples(i):
....:     global counter, n
....:     for _ in range(i):
....:         counter[D()] += 1
....:     n += 1

sage: add_samples(100000)
```

and check if the proportions fit:

```python
sage: expected = lambda x, y: (exp(-x^2/(2*sigma^2))/exp(-y^2/(2*sigma^2))).n()
sage: observed = lambda x, y: float(counter[x])/counter[y]

sage: while not all(v in counter for v in (0, 1, -100)): add_samples(Integer(10000))
```

We construct a sampler with $\sigma = 127$:

```python
>>> sigma = Integer(127)
>>> D = DiscreteGaussianDistributionIntegerSampler(sigma=sigma, algorithm='uniform+online')
```
sage: from sage.stats.distributions.discrete_gaussian_integer import \ ˓→DiscreteGaussianDistributionIntegerSampler
sage: sigma = 3
sage: D = DiscreteGaussianDistributionIntegerSampler(sigma=sigma, c=1/2)
sage: s = 0
sage: n = 0
sage: def add_samples(i):
       global s, n
       for _ in range(i):
           s += D()
           n += 1
sage: add_samples(100000)
sage: while abs(float(s)/n - 0.5) > 5e-2: add_samples(10000)

>>> from sage.all import *
>>> from sage.stats.distributions.discrete_gaussian_integer import \ ˓→DiscreteGaussianDistributionIntegerSampler
>>> sigma = Integer(3)
>>> D = DiscreteGaussianDistributionIntegerSampler(sigma=sigma, c=Integer(1)/\ ˓→Integer(2))
>>> s = Integer(0)
>>> n = Integer(0)
>>> def add_samples(i):
... global s, n
... for _ in range(i):
...     s += D()
...     n += Integer(1)
...>>> add_samples(Integer(100000))
>>> while abs(float(s)/n - RealNumber(0.5)) > RealNumber(5e-2):
...     add_˓→samples(Integer(10000))

REFERENCES:
• [DDLL2013]

class sage.stats.distributions.discrete_gaussian_integer.DiscreteGaussianDistributionIntegerSampler

Bases: sage.stats.distributions.discrete_gaussian_integer.DiscreteGaussianSampler

A Discrete Gaussian Sampler using rejection sampling.

__init__(sigma, c=0, tau=6, algorithm=None, precision='mp')

Construct a new sampler for a discrete Gaussian distribution.

INPUT:

- **sigma** – samples $x$ are accepted with probability proportional to $\exp(-(x-c)^2/(2\sigma^2))$
- **c** – the mean of the distribution. The value of $c$ does not have to be an integer. However, some algorithms only support integer-valued $c$ (default: 0)
- **tau** – samples outside the range $[Lc-l, Lc+l]$ are considered to have probability zero. This bound applies to algorithms which sample from the uniform distribution (default: 6)
- **algorithm** – see list below (default: "uniform+table" for $t$ bounded by DiscreteGaussianDistributionIntegerSampler.table_cutoff and "uniform+online" for bigger)
• **precision** – either "mp" for multi-precision where the actual precision used is taken from sigma or "dp" for double precision. In the latter case results are not reproducible. (default: "mp")

**ALGORITHMS:**

• "uniform+table" – classical rejection sampling, sampling from the uniform distribution and accepted with probability proportional to \(\exp(-(x-c)^2/(2^2))\) where \(\exp(-(x-c)^2/(2^2))\) is precomputed and stored in a table. Any real-valued \(c\) is supported.

• "uniform+logtable" – samples are drawn from a uniform distribution and accepted with probability proportional to \(\exp(-(x-c)^2/(2^2))\) where \(\exp(-(x-c)^2/(2^2))\) is computed using logarithmically many calls to Bernoulli distributions. See [DDLL2013] for details. Only integer-valued \(c\) are supported.

• "uniform+online" – samples are drawn from a uniform distribution and accepted with probability proportional to \(\exp(-(x-c)^2/(2^2))\) where \(\exp(-(x-c)^2/(2^2))\) is computed in each invocation. Typically this is very slow. See [DDLL2013] for details. Any real-valued \(c\) is accepted.

• "sigma2+logtable" – samples are drawn from an easily samplable distribution with \(\sigma^2 = kx^2\) with \(k = \sqrt{1/(2 \log 2)}\) and accepted with probability proportional to \(\exp(-(x-c)^2/(2^2))\) where \(\exp(-(x-c)^2/(2^2))\) is computed using logarithmically many calls to Bernoulli distributions (but no calls to exp). See [DDLL2013] for details. Note that this sampler adjusts to match \(kx^2\) for some integer \(k\). Only integer-valued \(c\) are supported.

**EXAMPLES:**

```python
sage: from sage.stats.distributions.discrete_gaussian_integer import DiscreteGaussianDistributionIntegerSampler
sage: DiscreteGaussianDistributionIntegerSampler(3.0, algorithm="uniform+online")
Discrete Gaussian sampler over the Integers with sigma = 3.000000 and c = 0.
sage: DiscreteGaussianDistributionIntegerSampler(3.0, algorithm="uniform+table")
Discrete Gaussian sampler over the Integers with sigma = 3.000000 and c = 0.
sage: DiscreteGaussianDistributionIntegerSampler(3.0, algorithm="uniform+logtable")
Discrete Gaussian sampler over the Integers with sigma = 3.000000 and c = 0.

>>> from sage.all import *
>>> from sage.stats.distributions.discrete_gaussian_integer import DiscreteGaussianDistributionIntegerSampler
>>> DiscreteGaussianDistributionIntegerSampler(RealNumber('3.0'), algorithm="uniform+online")
Discrete Gaussian sampler over the Integers with sigma = 3.000000 and c = 0.
>>> DiscreteGaussianDistributionIntegerSampler(RealNumber('3.0'), algorithm="uniform+table")
Discrete Gaussian sampler over the Integers with sigma = 3.000000 and c = 0.
>>> DiscreteGaussianDistributionIntegerSampler(RealNumber('3.0'), algorithm="uniform+logtable")
Discrete Gaussian sampler over the Integers with sigma = 3.000000 and c = 0.
```

Note that "sigma2+logtable" adjusts:
Discrete Gaussian sampler over the Integers with sigma = 3.397287 and c = 0.

```python
>>> from sage.all import *

**__call__**

Return a new sample.

**EXAMPLES:**

```python
sage: from sage.stats.distributions.discrete_gaussian_integer import DiscreteGaussianDistributionIntegerSampler
sage: DiscreteGaussianDistributionIntegerSampler(3.0, algorithm="uniform+online")() # random
-3

sage: DiscreteGaussianDistributionIntegerSampler(3.0, algorithm="uniform+table")() # random
3
```

algorithm
c
sigma
table cutoff = 1000000
tau
This class realizes oracles which returns polynomials in \( \mathbb{Z}[x] \) where each coefficient is sampled independently with a probability proportional to \( \exp(-|x - c|^2/(2s^2)) \).

AUTHORS:

- Martin Albrecht, Robert Fitzpatrick, Daniel Cabracas, Florian Göpfert, Michael Schneider: initial version

EXAMPLES:

```python
sage: from sage.stats.distributions.discrete_gaussian_polynomial import...
   ...DiscreteGaussianDistributionPolynomialSampler
sage: sigma = 3.0; n = 1000
sage: l = [DiscreteGaussianDistributionPolynomialSampler(ZZ['x'], 64, sigma)()
       ...for _ in range(n)]
sage: l = [vector(f).norm().n() for f in l]  # needs sage.symbolic
sage: from numpy import mean  # needs numpy
sage: mean(l), sqrt(64)*sigma  # abs tol 5e-1
(24.0, 24.0)
```

```python
>>> from sage.all import *
>>> from sage.stats.distributions.discrete_gaussian_polynomial import...
   ...DiscreteGaussianDistributionPolynomialSampler
>>> sigma = RealNumber('3.0'); n = Integer(1000)
>>> l = [DiscreteGaussianDistributionPolynomialSampler(ZZ['x'], Integer(64), sigma)()
       ...for _ in range(n)]
>>> l = [vector(f).norm().n() for f in l]  # needs sage.symbolic
>>> from numpy import mean  # needs numpy
>>> mean(l), sqrt(Integer(64))*sigma  # abs tol 5e-1
(24.0, 24.0)
```

**class** sage.stats.distributions.discrete_gaussian_polynomial.DiscreteGaussianDistributionPolynomialSampler

**Bases:** SageObject

Discrete Gaussian sampler for polynomials.

**EXAMPLES:**

...
```python
sage: from sage.stats.distributions.discrete_gaussian_polynomial import
    DiscreteGaussianDistributionPolynomialSampler
sage: p = DiscreteGaussianDistributionPolynomialSampler(ZZ['x'], 8, 3.0)()
sage: p.parent()
Univariate Polynomial Ring in x over Integer Ring
sage: p.degree() < 8
True
sage: gs = DiscreteGaussianDistributionPolynomialSampler(ZZ['x'], 8, 3.0)

sage: [gs() for __ in range(3)]
# random
[4*x^7 + 4*x^6 - 4*x^5 + 2*x^4 + x^3 - 4*x + 7, -5*x^6 + 4*x^5 - 3*x^3 + 4*x^2 +
  x, 2*x^7 + 2*x^6 + 2*x^5 - x^4 - 2*x^2 + 3*x + 1]
```

```python
>>> from sage.all import *
>>> from sage.stats.distributions.discrete_gaussian_polynomial import
    DiscreteGaussianDistributionPolynomialSampler

>>> p = DiscreteGaussianDistributionPolynomialSampler(ZZ['x'], Integer(8),
    RealNumber('3.0'))()

>>> p.parent()
Univariate Polynomial Ring in x over Integer Ring

>>> p.degree() < Integer(8)
True

>>> gs = DiscreteGaussianDistributionPolynomialSampler(ZZ['x'], Integer(8),
    RealNumber('3.0'))

>>> [gs() for __ in range(Integer(3))]
# random
[4*x^7 + 4*x^6 - 4*x^5 + 2*x^4 + x^3 - 4*x + 7, -5*x^6 + 4*x^5 - 3*x^3 + 4*x^2 +
  x, 2*x^7 + 2*x^6 + 2*x^5 - x^4 - 2*x^2 + 3*x + 1]
```

**__init__(P, n, sigma)***

Construct a sampler for univariate polynomials of degree \(n-1\) where coefficients are drawn independently with standard deviation \(\sigma\).

**INPUT:**

- \(P\) – a univariate polynomial ring over the Integers
- \(n\) – number of coefficients to be sampled
- \(\sigma\) – coefficients \(x\) are accepted with probability proportional to \(\exp(-x^2/(2\sigma^2))\). If an object of type `sage.stats.distributions.discrete_gaussian_integer.DiscreteGaussianDistributionIntegerSampler` is passed, then this sampler is used to sample coefficients.

**EXAMPLES:**

```python
sage: from sage.stats.distributions.discrete_gaussian_polynomial import
    DiscreteGaussianDistributionPolynomialSampler

sage: p = DiscreteGaussianDistributionPolynomialSampler(ZZ['x'], 8, 3.0)()

sage: p.parent()
Univariate Polynomial Ring in x over Integer Ring

sage: p.degree() < 8
True

sage: gs = DiscreteGaussianDistributionPolynomialSampler(ZZ['x'], 8, 3.0)

sage: [gs() for __ in range(3)]
# random
[4*x^7 + 4*x^6 - 4*x^5 + 2*x^4 + x^3 - 4*x + 7, -5*x^6 + 4*x^5 - 3*x^3 + 4*x^2 +
  x, 2*x^7 + 2*x^6 + 2*x^5 - x^4 - 2*x^2 + 3*x + 1]
```

```python
>>> from sage.all import *
>>> from sage.stats.distributions.discrete_gaussian_polynomial import

>>> p = DiscreteGaussianDistributionPolynomialSampler(ZZ['x'], Integer(8),
    RealNumber('3.0'))()

>>> p.parent()
Univariate Polynomial Ring in x over Integer Ring

>>> p.degree() < Integer(8)
True

>>> gs = DiscreteGaussianDistributionPolynomialSampler(ZZ['x'], Integer(8),
    RealNumber('3.0'))

>>> [gs() for __ in range(Integer(3))]
# random
[4*x^7 + 4*x^6 - 4*x^5 + 2*x^4 + x^3 - 4*x + 7, -5*x^6 + 4*x^5 - 3*x^3 + 4*x^2 +
  x, 2*x^7 + 2*x^6 + 2*x^5 - x^4 - 2*x^2 + 3*x + 1]
```

(continues on next page)
DiscreteGaussianDistributionPolynomialSampler

```python
>>> p = DiscreteGaussianDistributionPolynomialSampler(ZZ['x'], Integer(8),
RealNumber('3.0'))()
```

```python
>>> p.parent()
Univariate Polynomial Ring in x over Integer Ring
```

```python
>>> p.degree() < Integer(8)
True
```

```python
>>> gs = DiscreteGaussianDistributionPolynomialSampler(ZZ['x'], Integer(8),
RealNumber('3.0'))
```

```python
>>> [gs() for _ in range(Integer(3))]
# random
[4*x^7 + 4*x^6 - 4*x^5 + 2*x^4 + x^3 - 4*x + 7, -5*x^6 + 4*x^5 - 3*x^3 + 4*x^2 + x, 2*x^7 + 2*x^6 + 2*x^5 - x^4 - 2*x^2 + 3*x + 1]
```

___call__()

Return a new sample.

EXAMPLES:

```python
sage: from sage.stats.distributions.discrete_gaussian_polynomial import_
DiscreteGaussianDistributionPolynomialSampler
sage: sampler = DiscreteGaussianDistributionPolynomialSampler(ZZ['x'], 8, 12.0)
sage: sampler().parent()
Univariate Polynomial Ring in x over Integer Ring
sage: sampler().degree() <= 7
True
```

```python
>>> from sage.all import *
```

```python
>>> from sage.stats.distributions.discrete_gaussian_polynomial import_
DiscreteGaussianDistributionPolynomialSampler
```

```python
>>> sampler = DiscreteGaussianDistributionPolynomialSampler(ZZ['x'],
Integer(8), RealNumber('12.0'))
```

```python
>>> sampler().parent()
Univariate Polynomial Ring in x over Integer Ring
```

```python
>>> sampler().degree() <= Integer(7)
True
```
CHAPTER

NINE

DISCRETE GAUSSIAN SAMPLERS OVER LATTICES

This file implements oracles which return samples from a lattice following a discrete Gaussian distribution. That is, if \( \sigma \) is big enough relative to the provided basis, then vectors are returned with a probability proportional to \( \exp(-|x - c|^2/(2\sigma^2)) \). More precisely lattice vectors in \( x \in \Lambda \) are returned with probability:

\[
\frac{\exp(-|x - c|^2/(2\sigma^2))}{\sum_{v \in \Lambda} \exp(-|v|^2/(2\sigma^2))}.
\]

AUTHORS:

- Gareth Ma (2023-09-22): implement non-spherical sampling

EXAMPLES:

```python
sage: D = distributions.DiscreteGaussianDistributionLatticeSampler(ZZ^10, 3.0)
sage: D(), D(), D()  # random
((3, 0, -5, 0, -1, -3, 3, -7, 2), (4, 0, 1, -2, -4, -4, 4, 0, 1, -4), (-3, 0, 4, 5, 0, 1, 3, 2, 0, -1))
sage: a = D()
sage: a.parent()
Ambient free module of rank 10 over the principal ideal domain Integer Ring
```

```python
>>> from sage.all import *
>>> D = distributions.DiscreteGaussianDistributionLatticeSampler(ZZ**Integer(10), RealNumber(3.0))
>>> D(), D(), D()  # random
((3, 0, -5, 0, -1, -3, 3, -7, 2), (4, 0, 1, -2, -4, -4, 4, 0, 1, -4), (-3, 0, 4, 5, 0, 1, 3, 2, 0, -1))
>>> a = D()
>>> a.parent()
Ambient free module of rank 10 over the principal ideal domain Integer Ring
```

**class** **sage.stats.distributions.discrete_gaussian_lattice.DiscreteGaussianDistributionLatticeSampler**

**Bases:** `SageObject`

GPV sampler for Discrete Gaussians over Lattices.
EXAMPLES:

```python
sage: D = distributions.DiscreteGaussianDistributionLatticeSampler(ZZ^10, 3.0); D
Discrete Gaussian sampler with Gaussian parameter σ = 3.00000000000000, c=(0, 0, 0, 0, 0, 0, 0, 0, 0, 0) over lattice with basis

[[0 0 0 0 0 0 0 0 0 0]
 [0 1 0 0 0 0 0 0 0 0]
 [0 0 1 0 0 0 0 0 0 0]
 [0 0 0 1 0 0 0 0 0 0]
 [0 0 0 0 1 0 0 0 0 0]
 [0 0 0 0 0 1 0 0 0 0]
 [0 0 0 0 0 0 1 0 0 0]
 [0 0 0 0 0 0 0 1 0 0]
 [0 0 0 0 0 0 0 0 1 0]
 [0 0 0 0 0 0 0 0 0 1]]
```

We plot a histogram:

```python
sage: import warnings
sage: warnings.simplefilter('ignore', UserWarning)

sage: D = distributions.DiscreteGaussianDistributionLatticeSampler(identity_matrix(2), 3.0)

sage: S = [D() for _ in range(2^12)]

sage: l = [vector(v.list() + [S.count(v)]) for v in set(S)]

sage: list_plot3d(l, point_list=True, interpolation='nn')
```

REFERENCES:

Chapter 9. Discrete Gaussian Samplers over Lattices
• [GPV2008]

\[ \text{__init__}(B, \sigma=1, c=0, r=None, precision=None, \text{sigma\_basis}=False) \]

Construct a discrete Gaussian sampler over the lattice \( \Lambda(B) \) with parameter \( \sigma \) and center \( c \).

INPUT:

- \( B \) – a (row) basis for the lattice, one of the following:
  - an integer matrix,
  - an object with a \texttt{.matrix()} method, e.g. \( \mathbb{Z}^n \), or
  - an object where \texttt{matrix(B)} succeeds, e.g. a list of vectors
- \( \sigma \) – Gaussian parameter, one of the following:
  - a real number \( \sigma > 0 \) (spherical),
  - a positive definite matrix \( \Sigma \) (non-spherical), or
  - any matrix-like \( S \), equivalent to \( \Sigma = SS^T \), when \text{sigma\_basis} is set
- \( c \) – (default: 0) center \( c \), any vector in \( \mathbb{Z}^n \) is supported, but \( c \in \Lambda(B) \) is faster
- \( r \) – (default: None) rounding parameter \( r \) as defined in [Pei2010]; ignored for spherical Gaussian parameter; if not provided, set to be the maximal possible such that \( \Sigma - rBB^T \) is positive definite
- \text{precision} – bit precision \( \geq 53 \).
- \text{sigma\_basis} – (default: \text{False}) When set, \( \sigma \) is treated as a (row) basis, i.e. the covariance matrix is computed by \( \Sigma = SS^T \)

Todo: Rename class methods like \texttt{.f} and hide most of them (at least behind something like \texttt{.data}).

EXAMPLES:

```python
sage: n = 2; sigma = 3.0
sage: D = distributions.DiscreteGaussianDistributionLatticeSampler(ZZ^n, sigma)
```

```python
sage: f = D.f
sage: nf = D._normalisation_factor_zz(); nf
# needs sage.symbolic
56.548677646...
```

```python
sage: from collections import defaultdict
sage: counter = defaultdict(Integer); m = 0
sage: def add_samples(i):
....:     global counter, m
....:     for _ in range(i):
....:         counter[D()] += 1
....:         m += 1
```

```python
sage: v = vector(ZZ, n, (-3, -3))
sage: v.set_immutable()
sage: while v not in counter: add_samples(1000)
```

(continues on next page)
sage: v = vector(ZZ, n, (0, 0))
sage: v.set_immutable()
sage: while v not in counter:  
.....: add_samples(1000)
sage: while abs(m*f(v)*1.0/nf/counter[v] - 1.0) >= 0.1:  
..: needs sage.symbolic
.....: add_samples(1000)

>>> from sage.all import *
>>> n = Integer(2); sigma = RealNumber('3.0')
>>> D = distributions.DiscreteGaussianDistributionLatticeSampler(ZZ**n, sigma)
>>> f = D.f
>>> nf = D._normalisation_factor_zz(); nf  
..: needs sage.symbolic
56.5486677646...

>>> from collections import defaultdict
>>> counter = defaultdict(Integer); m = Integer(0)
>>> def add_samples(i):
... global counter, m
... for _ in range(i):
...     counter[D()] += Integer(1)
...     m += Integer(1)

>>> v = vector(ZZ, n, (-Integer(3), -Integer(3)))
>>> v.set_immutable()

Spherical covariance are automatically handled:

sage: distributions.DiscreteGaussianDistributionLatticeSampler(ZZ^3,  
..: sigma=Matrix(3, 3, 2))
Discrete Gaussian sampler with Gaussian parameter σ = 2.00000000000000, c=(0,  
..: 0, 0) over lattice with basis

[1 0 0]
[0 1 0]
[0 0 1]

>>> from sage.all import *
>>> distributions.DiscreteGaussianDistributionLatticeSampler(ZZ**Integer(3),  
..: sigma=Matrix(Integer(3), Integer(3), Integer(2)))
Discrete Gaussian sampler with Gaussian parameter σ = 2.00000000000000, c=(0,  
..: 0, 0) over lattice with basis
The sampler supports non-spherical covariance in the form of a Gram matrix:

```python
sage: n = 3
sage: Sigma = Matrix(ZZ, [[5, -2, 4], [-2, 10, -5], [4, -5, 5]])
sage: c = vector(ZZ, [7, 2, 5])
sage: D = distributions.DiscreteGaussianDistributionLatticeSampler(ZZ^n, Sigma, c)
sage: f = D.f
sage: nf = D._normalisation_factor_zz(); nf
```

This has not been properly implemented

```
78.6804...
```

We can compute the expected number of samples before sampling a vector:

```python
sage: v = vector(ZZ, n, (11, 4, 8))
sage: v.set_immutable()
sage: 1 / (f(v) / nf)
2553.9461...
```

```python
sage: counter = defaultdict(Integer); m = 0
sage: while v not in counter:
    ....:     add_samples(1000)
sage: sum(counter.values())
3000
```

```python
sage: while abs(m*f(v)*1.0/nf/counter[v] - 1.0) >= 0.1:  # needs sage.symbolic
    ....:     add_samples(1000)
```

```python
>>> from sage.all import *
>>> v = vector(ZZ, n, (Integer(11), Integer(4), Integer(8)))
>>> v.set_immutable()
>>> Integer(1) / (f(v) / nf)
2553.9461...
```

```python
>>> counter = defaultdict(Integer); m = Integer(0)
>>> while v not in counter:
    ...     add_samples(Integer(1000))
>>> sum(counter.values())  # random
3000
```
While

if the covariance provided is not positive definite, an error is thrown:

```
sage: Sigma = Matrix(ZZ, [[0, 1], [1, 0]])
sage: distributions.DiscreteGaussianDistributionLatticeSampler(ZZ^2, Sigma)
Traceback (most recent call last):
  ... 
RuntimeError: Sigma(=[0.000000000000000 1.00000000000000]
[ 1.00000000000000 0.000000000000000]) is not positive definite
```

The sampler supports passing a basis for the covariance:

```
sage: n = 3
sage: S = Matrix(ZZ, [[2, 0, 0], [-1, 3, 0], [2, -1, 1]])
S = 
[[2 0 0]
 [1 -3 0]
 [2 -1 1]]
sage: D = distributions.DiscreteGaussianDistributionLatticeSampler(ZZ^n, S, sigma_basis=True)
sage: D.sigma()
[[4.00000000000000 -2.00000000000000 4.00000000000000]
 [-2.00000000000000 10.00000000000000 -5.00000000000000]
 [4.00000000000000 -5.00000000000000 6.00000000000000]]
```

The non-spherical sampler supports offline computation to speed up sampling. This will be useful when changing the center \(c\) is supported. The difference is more significant for larger matrices. For 128x128 we observe a 4x speedup (86s -> 20s):

```
sage: D.offline_samples = []
sage: T = 2**12
sage: L = [D() for _ in range(T)] # 560ms
sage: D.add_offline_samples(T) # 150ms
sage: L = [D() for _ in range(T)] # 370ms
```

Chapter 9. Discrete Gaussian Samplers over Lattices
We can also initialise with matrix-like objects:

```
sage: qf = matrix(3, [2, 1, 1, 1, 2, 1, 1, 1, 2])  
sage: D = distributions.DiscreteGaussianDistributionLatticeSampler(qf, 3.0); D  
    Discrete Gaussian sampler with Gaussian parameter \( \sigma = 3.00000000000000 \), \( c=(0, \ldots, 0, 0) \) over lattice with basis
    
    \begin{bmatrix}
        2 & 1 & 1 \\
        1 & 2 & 1 \\
        1 & 1 & 2
    \end{bmatrix}
```

```
>>> from sage.all import *  
>>> qf = matrix(Integer(3), [Integer(2), Integer(1), Integer(1), Integer(1),  
                        Integer(2), Integer(1), Integer(1), Integer(1), Integer(2)])  
>>> D = distributions.DiscreteGaussianDistributionLatticeSampler(qf, RealNumber('3.0')); D  
    Discrete Gaussian sampler with Gaussian parameter \( \sigma = 3.00000000000000 \), \( c=(0, \ldots, 0, 0) \) over lattice with basis
    
    \begin{bmatrix}
        2 & 1 & 1 \\
        1 & 2 & 1 \\
        1 & 1 & 2
    \end{bmatrix}
```

__call__

Return a new sample.

EXAMPLES:

```
sage: D = distributions.DiscreteGaussianDistributionLatticeSampler(ZZ^3, 3.0,  
    c=(1,0,0))  
sage: L = [D() for _ in range(2^12)]  
sage: mean_L = sum(L) / len(L)  
sage: norm(mean_L.n() - D.c()) < 0.25  
    True
```

```
sage: D = distributions.DiscreteGaussianDistributionLatticeSampler(ZZ^3, 3.0,  
    c=(1/2,0,0))  
sage: L = [D() for _ in range(2^12)]  
    # long time  
sage: mean_L = sum(L) / len(L)  
    # long time  
sage: norm(mean_L.n() - D.c()) < 0.25  
    # long time  
    True
```

```
>>> from sage.all import *  
>>> D = distributions.  
    DiscreteGaussianDistributionLatticeSampler(ZZ^3, RealNumber('3.0'),  
    c=(Integer(1),Integer(0),Integer(0)))
```

(continues on next page)
```python
>>> L = [D() for _ in range(Integer(2)**Integer(12))]
>>> mean_L = sum(L) / len(L)
>>> norm(mean_L.n() - D.c()) < RealNumber('0.25')
True
```

```python
>>> D = distributions.DiscreteGaussianDistributionLatticeSampler(ZZ**Integer(3), RealNumber('3.0'), c=(Integer(1)/Integer(2), Integer(0), Integer(0)))
```

```python
>>> L = [D() for _ in range(Integer(2)**Integer(12))]  # long time
>>> mean_L = sum(L) / len(L)  # long time
>>> norm(mean_L.n() - D.c()) < RealNumber('0.25')  # long time
True
```

```python
add_offline_samples(cnt=1)
```

Precompute samples from $B^{-1}D_1$ to be used in _call_non_spherical().

**EXAMPLES:**

```python
sage: Sigma = Matrix([[5, -2, 4], [-2, 10, -5], [4, -5, 5]])
```

```python
sage: D = distributions.DiscreteGaussianDistributionLatticeSampler(ZZ^3, Sigma)
```

```python
sage: assert not D.is_spherical
```

```python
sage: D.add_offline_samples(2^12)
```

```python
sage: L = [D() for _ in range(2^12)]  # Takes less time
```

```python
>>> from sage.all import *
```

```python
>>> Sigma = Matrix([[Integer(5), -Integer(2), Integer(4)], [-Integer(2), Integer(10), -Integer(5)], [Integer(4), -Integer(5), Integer(5)]]
```

```python
>>> D = distributions.DiscreteGaussianDistributionLatticeSampler(ZZ**Integer(3), Sigma)
```

```python
>>> assert not D.is_spherical
```

```python
>>> D.add_offline_samples(Integer(2)**Integer(12))
```

```python
>>> L = [D() for _ in range(Integer(2)**Integer(12))]  # Takes less time
```

```python
Center c.

Samples from this sampler will be centered at c.

**EXAMPLES:**

```python
sage: D = distributions.DiscreteGaussianDistributionLatticeSampler(ZZ^3, 3.0, c=(1,0,0))
```

```python
Discrete Gaussian sampler with Gaussian parameter \( \sigma = 3.0 \), \( c=(1,0,0) \) over lattice with basis
```

```latex
\begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{bmatrix}
```

```python
sage: D.c()
```

```python
(1, 0, 0)
```

```python
>>> from sage.all import *
```

```python
>>> D = distributions.
```

```python
DiscreteGaussianDistributionLatticeSampler(ZZ**Integer(3), RealNumber('3.0')
```

(continues on next page)
Discrete Gaussian sampler with Gaussian parameter $\sigma = 3.00000000000000$, $c=(1,\rightarrow0, 0)$ over lattice with basis

$$
\begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{bmatrix}
$$

```python
>>> D.c()
(1, 0, 0)
```

**static compute_precision** *(precision, sigma)*

Compute precision to use.

**INPUT:**

- precision – an integer $\geq 53$ nor None.
- sigma – if precision is None then the precision of sigma is used.

**EXAMPLES:**

```python
sage: DGL = distributions.DiscreteGaussianDistributionLatticeSampler
sage: DGL.compute_precision(100, RR(3))
100
sage: DGL.compute_precision(100, RealField(200)(3))
100
sage: DGL.compute_precision(100, 3)
100
sage: DGL.compute_precision(None, RR(3))
53
sage: DGL.compute_precision(None, RealField(200)(3))
200
sage: DGL.compute_precision(None, 3)
53
```

```python
>>> from sage.all import *
>>> DGL = distributions.DiscreteGaussianDistributionLatticeSampler
>>> DGL.compute_precision(Integer(100), RR(Integer(3)))
100
>>> DGL.compute_precision(Integer(100), RealField(Integer(200))(Integer(3)))
100
>>> DGL.compute_precision(Integer(100), Integer(3))
100
>>> DGL.compute_precision(None, RR(Integer(3)))
53
>>> DGL.compute_precision(None, RealField(Integer(200))(Integer(3)))
200
>>> DGL.compute_precision(None, Integer(3))
53
```

$f(x)$

Compute the Gaussian $\rho_{A,c,\Sigma}$.

**EXAMPLES:**
sage: Sigma = Matrix(ZZ, [[5, -2, 4], [-2, 10, -5], [4, -5, 5]])
sage: D = distributions.DiscreteGaussianDistributionLatticeSampler(ZZ^3, Sigma)
sage: D.f([1, 0, 1])
0.802518797962478
sage: D.f([1, 0, 3])
0.00562800641440405
>>> from sage.all import *
>>> Sigma = Matrix(ZZ, [[Integer(5), -Integer(2), Integer(4)], [-Integer(2), Integer(10), -Integer(5)], [Integer(4), -Integer(5), Integer(5)]])
>>> D = distributions.DiscreteGaussianDistributionLatticeSampler(ZZ**Integer(3), Sigma)
>>> D.f([Integer(1), Integer(0), Integer(1)])
0.802518797962478
>>> D.f([Integer(1), Integer(0), Integer(3)])
0.00562800641440405

set_c(c)
Modifies center $c$.

EXAMPLES:

sage: D = distributions.DiscreteGaussianDistributionLatticeSampler(ZZ^3, 3.0, c=(1,0,0))
sage: D.set_c((2, 0, 0))
sage: D
Discrete Gaussian sampler with Gaussian parameter $\sigma = 3.00000000000000$, $c=(2, 0, 0)$ over lattice with basis

[[1 0 0]
 [0 1 0]
 [0 0 1]]

sigma()
Gaussian parameter $\sigma$.

If $\sigma$ is a real number, samples from this sampler will have expected norm $\sqrt{n}\sigma$ where $n$ is the dimension of the lattice.

EXAMPLES:

sage: D = distributions.DiscreteGaussianDistributionLatticeSampler(ZZ^3, 3.0, c=(1,0,0))
```python
sage: D.sigma()
3.00000000000000

>>> from sage.all import *
>>> D = distributions.
    →DiscreteGaussianDistributionLatticeSampler(ZZ**Integer(3), RealNumber('3.0 
    →'), c=(Integer(1),Integer(0),Integer(0)))
>>> D.sigma()
3.00000000000000
```
sage.stats.r.ttest(x, y, conf_level=0.95, **kw)

T-Test using R

INPUT:
- $x, y$ – vectors of same length
- $\text{conf\_level}$ – confidence level of the interval, $[0,1)$ in percent

OUTPUT:
Tuple: (p-value, R return object)

EXAMPLES:

```
sage: a, b = ttest([1,2,3,4,5],[1,2,3,3.5,5.121]); a
# abs tol 1e-12 # optional - rpy2
0.9410263720274274
```

```
from sage.all import *

>>> a, b = ttest([Integer(1),Integer(2),Integer(3),Integer(4),Integer(5)],
>>>              [Integer(1),Integer(2),Integer(3),RealNumber('3.5'),RealNumber('5.121')]); a
# abs tol 1e-12 # optional - rpy2
0.9410263720274274
```
INDICES AND TABLES

- Index
- Module Index
- Search Page
S
sage.stats.basic_stats, 1
sage.stats.distributions.discrete_gaussian_integer, 57
sage.stats.distributions.discrete_gaussian_lattice, 67
sage.stats.distributions.discrete_gaussian_polynomial, 63
sage.stats.hmm.chmm, 31
sage.stats.hmm.distributions, 45
sage.stats.hmm.hmm, 17
sage.stats.hmm.util, 53
sage.stats.intlist, 11
sage.stats.r, 79
Non-alphabetical
__call__()  (sage.stats.distributions.discrete_gaussian_integer.DiscreteGaussianDistributionIntegerSampler method), 62
__call__()  (sage.stats.distributions.discrete_gaussian_lattice.DiscreteGaussianDistributionLatticeSampler method), 73
__call__()  (sage.stats.distributions.discrete_gaussian_polynomial.DiscreteGaussianDistributionPolynomialSampler method), 65
__init__()  (sage.stats.distributions.discrete_gaussian_integer.DiscreteGaussianDistributionIntegerSampler method), 60
__init__()  (sage.stats.distributions.discrete_gaussian_lattice.DiscreteGaussianDistributionLatticeSampler method), 69
__init__()  (sage.stats.distributions.discrete_gaussian_polynomial.DiscreteGaussianDistributionPolynomialSampler method), 64
add_offline_samples()  (sage.stats.distributions.discrete_gaussian_lattice.DiscreteGaussianDistributionLatticeSampler method), 74
c()  (sage.stats.distributions.discrete_gaussian_lattice.DiscreteGaussianDistributionLatticeSampler method), 74
compute_precision()  (sage.stats.distributions.discrete_gaussian_lattice.DiscreteGaussianDistributionLatticeSampler static method), 75

D
DiscreteDistribution (class in sage.stats.distributions), 45
DiscreteGaussianDistributionIntegerSampler (class in sage.stats.distributions.discrete_gaussian_integer), 60
DiscreteGaussianDistributionLatticeSampler (class in sage.stats.distributions.discrete_gaussian_lattice), 67
DiscreteGaussianDistributionPolynomialSampler (class in sage.stats.distributions.discrete_gaussian_polynomial), 63
DiscreteHiddenMarkovModel (class in sage.stats.hmm), 17
Distribution (class in sage.stats.hmm.distributions), 45

E
emission_matrix()  (sage.stats.hmm.hmm.DiscreteHiddenMarkovModel method), 21
emission_parameters()  (sage.stats.hmm.hmm.DiscreteHiddenMarkovModel method), 36
emission_parameters()  (sage.stats.hmm.chmm.GaussianMixtureHiddenMarkovModel method), 43

F
f()  (sage.stats.distributions.discrete_gaussian_lattice.DiscreteGaussianDistributionLatticeSampler method), 75
fix()  (sage.stats.hmm.distributions.GaussianMixtureDistribution method), 47
G
GaussianDistribution (class in sage.stats.hmm.distributions), 46
GaussianHiddenMarkovModel (class in sage.stats.hmm.chmm), 31
GaussianMixtureDistribution (class in sage.stats.hmm.distributions), 46
GaussianMixtureHiddenMarkovModel (class in sage.stats.hmm.chmm), 40
generate_sequence() (sage.stats.hmm.chmm.GaussianHiddenMarkovModel method), 37
generate_sequence() (sage.stats.hmm.hmm.DiscreteHiddenMarkovModel method), 22
graph() (sage.stats.hmm.hmm.HiddenMarkovModel method), 26

H
HiddenMarkovModel (class in sage.stats.hmm.hmm), 25
HMM_Util (class in sage.stats.hmm.util), 53

I
initial_probabilities() (sage.stats.hmm.hmm.HiddenMarkovModel method), 26
initial_probs_to_TimeSeries() (sage.stats.hmm.util.HMM_Util method), 53
IntList (class in sage.stats.intlist), 11
is_fixed() (sage.stats.hmm.distributions.GaussianMixtureDistribution method), 48

L
list() (sage.stats.intlist.IntList method), 11
log_likelihood() (sage.stats.hmm.chmm.GaussianHiddenMarkovModel method), 38
log_likelihood() (sage.stats.hmm.hmm.DiscreteHiddenMarkovModel method), 23

M
max() (sage.stats.intlist.IntList method), 11
mean() (in module sage.stats.basic_stats), 1
median() (in module sage.stats.basic_stats), 2
min() (sage.stats.intlist.IntList method), 12
mode() (in module sage.stats.basic_stats), 3
module
sage.stats.basic_stats, 1
sage.stats.distributions.discrete_gaussian_integer, 57
sage.stats.distributions.discrete_gaussian_lattice, 67
sage.stats.distributions.discrete_gaussian_polynomial, 63
sage.stats.hmm.chmm, 31

sage.stats.hmm.distributions, 45
sage.stats.hmm.hmm, 17
sage.stats.hmm.util, 53
sage.stats.intlist, 11
sage.stats.r, 79
moving_average() (in module sage.stats.basic_stats), 5

N
normalize_probability_TimeSeries() (sage.stats.hmm.util.HMM_Util method), 53

P
plot() (sage.stats.hmm.distributions.Distribution method), 45
plot() (sage.stats.intlist.IntList method), 12
plot_histogram() (sage.stats.intlist.IntList method), 13
prob() (sage.stats.hmm.distributions.Distribution method), 45
prob() (sage.stats.hmm.distributions.GaussianMixtureDistribution method), 48
prob_m() (sage.stats.hmm.distributions.GaussianMixtureDistribution method), 49
prod() (sage.stats.intlist.IntList method), 13

S
sage.stats.basic_stats module, 1
sage.stats.distributions.discrete_gaussian_integer module, 57
sage.stats.distributions.discrete_gaussian_lattice module, 67
sage.stats.distributions.discrete_gaussian_polynomial module, 63
sage.stats.hmm.chmm module, 31
sage.stats.hmm.distributions module, 45
sage.stats.hmm.hmm module, 17
sage.stats.hmm.util module, 53
sage.stats.intlist module, 11
sage.stats.r module, 79
sample() (sage.stats.hmm.distributions.Distribution method), 46
sample() (sage.stats.hmm.distributions.GaussianMixtureDistribution method), 49
sample()  (sage.stats.hmm.hmm.HiddenMarkovModel method), 27
set_c()  (sage.stats.distributions.discrete_gaussian_lattice.DiscreteGaussianDistributionLatticeSampler method), 76
sigma  (sage.stats.distributions.discrete_gaussian_integer.DiscreteGaussianDistributionIntegerSampler attribute), 62
sigma()  (sage.stats.distributions.discrete_gaussian_lattice.DiscreteGaussianDistributionLatticeSampler method), 76
state_matrix_to_TimeSeries()  (sage.stats.hmm.util.HMM_Util method), 54
std()  (in module sage.stats.basic_stats), 6
sum()  (sage.stats.intlist.IntList method), 14

T
table_cutoff  (sage.stats.distributions.discrete_gaussian_integer.DiscreteGaussianDistributionIntegerSampler attribute), 62
tau  (sage.stats.distributions.discrete_gaussian_integer.DiscreteGaussianDistributionIntegerSampler attribute), 62
time_series()  (sage.stats.intlist.IntList method), 14
transition_matrix()  (sage.stats.hmm.hmm.HiddenMarkovModel method), 29
ttest()  (in module sage.stats.r), 79

U
unfix()  (sage.stats.hmm.distributions.GaussianMixtureDistribution method), 50
unpickle_discrete_hmm_v0()  (in module sage.stats.hmm), 30
unpickle_discrete_hmm_v1()  (in module sage.stats.hmm), 30
unpickle_gaussian_hmm_v0()  (in module sage.stats.hmm.chmm), 43
unpickle_gaussian_hmm_v1()  (in module sage.stats.hmm.chmm), 44
unpickle_gaussian_mixture_distribution_v1()  (in module sage.stats.hmm.distributions), 51
unpickle_gaussian_mixture_hmm_v1()  (in module sage.stats.hmm.chmm), 44
unpickle_intlist_v1()  (in module sage.stats.intlist), 14

V
variance()  (in module sage.stats.basic_stats), 7
viterbi()  (sage.stats.hmm.chmm.GaussianHiddenMarkovModel method), 39
viterbi()  (sage.stats.hmm.hmm.DiscreteHiddenMarkovModel method), 24