pyobs Documentation

Release 1.0.0

Mattia Bruno

Sep 18, 2020

CHAPTER

ONE

THE OBS CLASS

class `pyobs.obs`(*orig=None*, *desc='unknown'*)

Class defining an observable

Parameters

- `orig` (`obs`, *`optional`*) – creates a copy of orig
- `desc` (*`str`*, *`optional`*) – description of the observable

Examples

```
>>> from pyobs import obs
>>> a = obs(desc='test')
```

`add_syst_err`(*name*, *err*)

Add a systematic error to the observable

Parameters

- `name` (*`str`*) – label that uniquely identifies the syst. error
- `err` (*`array`*) – array with the same dimensions of the observable with the systematic error

Examples

```
>>> data = [0.198638, 0.403983, 1.215960, 1.607684, 0.199049, ... ]
>>> vec = pyobs.obs(desc='vector')
>>> vec.create('A',data,shape=(4,))
>>> print(vec)
0.201(13)    0.399(26)    1.199(24)    1.603(47)
>>> vec.add_syst_err('syst.err',[0.05,0.05,0,0])
>>> print(vec)
0.201(52)    0.399(56)    1.199(24)    1.603(47)
```

`create`(*ename*, *data*, *icnfg=None*, *rname=None*, *shape=1*, *lat=None*)

Create an observable

Parameters

- `ename` (*`str`*) – label of the ensemble
- `data` (*`array`*, *`list of arrays`*) – the data generated from a single or multiple replica
- `icnfg` (*`array of ints or list of arrays of ints`*, *`optional`*) – indices of the configurations corresponding to data; if not passed the measurements are assumed to be contiguous

- `rname` (*str or list of str, optional*) – identifier of the replica; if not passed integers from 0 are automatically assigned
- `shape` (*tuple, optional*) – shape of the observable, data must be passed accordingly
- `lat` (*list of ints, optional*) – the size of each dimension of the master-field; if passed data is assumed to be obtained from observables measured at different sites and *icnfg* is re-interpreted as the index labeling the sites; if *icnfg* is not passed data is assumed to be contiguous on all sites.

Note: For data and icnfg array can mean either a list or a 1-D numpy.array. If the observable has already been created, calling create again will add a new replica to the same ensemble.

Examples

```
>>> data = [0.43, 0.42, ... ] # a scalar observable
>>> a = pyobs.obs(desc='test')
>>> a.create('EnsembleA',data)
```

```
>>> data0 = [0.43,0.42, ... ] # replica 0
>>> data1 = [0.40,0.41, ... ] # replica 1
>>> a = pyobs.obs(desc='test')
>>> a.create('EnsembleA',[data0,data1],rname=['r0','r1'])
```

```
>>> data = [0.43, 0.42, 0.44, ... ]
>>> icnfg= [  10,   11,   13, ... ]
>>> a = pyobs.obs(desc='test')
>>> a.create('EnsembleA',data,icnfg=icnfg)
```

```
>>> data = [1.0, 2.0, 3.0, 4.0, ... ]
>>> a = pyobs.obs(desc='matrix')
>>> a.create('EnsembleA',data,shape=(2,2))
```

Examples

```
>>> data = [0.43, 0.42, 0.44, ... ]
>>> lat = [64,32,32,32]
>>> a = pyobs.obs(desc='test-mf')
>>> a.create('EnsembleA',data,lat=lat)
```

```
>>> data = [0.43, 0.42, 0.44, ... ]
>>> idx = [0, 2, 4, 6, ...] # measurements on all even points of time-slice
>>> lat = [32, 32, 32]
>>> a = pyobs.obs(desc='test-mf')
>>> a.create('EnsembleA',data,lat=lat,icnfg=idx)
```

`create_from_cov`(*cname*, *value*, *covariance*)

Create observables based on covariance matrices

Parameters

- `cname` (*str*) – label that uniquely identifies the data set
- `value` (*array*) – central value of the observable; only 1-D arrays are supported
- `covariance` (*array*) – a 2-D covariance matrix; if *covariance* is a 1-D array of same length as *value*, a diagonal covariance matrix is assumed.

Examples

```
>>> mpi = pyobs.obs(desc='pion masses, charged and neutral')
>>> mpi.create_cd('mpi-pdg18',[139.57061,134.9770],[0.00023**2,0.0005**2])
>>> print(mpi)
139.57061(23)    134.97700(50)
```

`error`(*errinfo={}*, *plot=False*, *pfile=None*)

Estimate the error of the observable, by summing in quadrature the systematic errors with the statistical errors computed from all ensembles and master fields.

Parameters

- `errinfo` (*dict, optional*) – dictionary containg one instance of the errinfo class for each ensemble/master-field. The errinfo class provides additional details for the automatic or manual windowing procedure in the Gamma method. If not passed default parameters are assumed.
- `plot` (*bool, optional*) – if specified a plot is produced, for every element of the observable, and for every ensemble/master-field where the corresponding element has fluctuations. In addition one piechart plot is produced for every element, showing the contributions to the error from the various sources, only if there are multiple sources, ie several ensembles. It is recommended to use the plotting function only for observables with small dimensions.
- `pfile` (*str, optional*) – if specified all plots produced with the flag *plot* are saved to disk, using *pfile* as base name with an additional suffix.

Returns the central value and error of the observable.

Return type list of two arrays

Note: By default, the errors are computed with the Gamma method, with the *Stau* parameter equal to 1.5. Additionally the jackknife method can be used by passing the appropriate *errinfo* dictionary with argument *bs* set to a non-zero integer value. For master fields the error is computed using the master-field approach and the automatic windowing procedure requires the additional argument *k* (see main documentation), which by default is zero, but can be specified via the errinfo dictionary. Through the *errinfo* dictionary the user can treat every ensemble differently, as explained in the examples below.

Examples

```
>>> obsA = pyobs.obs('obsA')
>>> obsA.create('A',dataA) # create the observable A from ensemble A
>>> [v,e] = obsA.error() # collect central value and error in v,e
>>> einfo = {'A': errinfo(Stau=3.0)} # specify non-standard Stau for ensemble A
>>> [_,e1] = obsA.error(errinfo=einfo)
>>> print(e,e1) # check difference in error estimation
```

```
>>> obsB = pyobs.obs('obsB')
>>> obsB.create('B',dataB) # create the observable B from ensemble B
>>> obsC = obsA * obsB # derived observable with fluctuations from ensembles A,B
>>> einfo = {'A': errinfo(Stau=3.0), 'B': errinfo(W=30)}
>>> [v,e] = obsC.error(errinfo=einfo,plot=True)
```

`error_of_error`(*errinfo={}*)

Returns the error of the error based on the analytic prediction obtained by U. Wolff.

Parameters `errinfo` (*dict, optional*) – see the documentation of the *error* method.

Returns the error of the error

Return type array

`peek()`

Display a brief summary of the content of the observable, including its memory footprint and requirements (for error computation), its description and ensemble/replica content

Example

```
>>> obs.peek()
Observable with shape = (1, 4)
 - description: vector-test
 - size: 82 KB
 - mean: [[0.20007161 0.40085252 1.19902686 1.60184989]]
 - Ensemble A
    - Replica 0 with mask [0, 1, 2, 3] and ncnfg 500
         temporary additional memory required 0.015 MB
```

`tauint()`

Estimates the integrated autocorrelation time and its error for every ensemble, with the automatic windowing procedure.

CHAPTER

TWO

MANIPULATION OF OBSERVABLES

reshape(x, new_shape)	Change the shape of the observable
concatenate(x, y[, axis])	Join two arrays along an existing axis
transpose(x[, axes])	Transpose a tensor along specific axes.
sort(x[, axis])	Sort a tensor along a specific axis.
diag(x)	Extract the diagonal of 2-D array or construct a diagonal matrix from a 1-D array

2.1 pyobs.reshape

`pyobs.reshape`(*x*, *new_shape*)

Change the shape of the observable

Parameters

- `x` (`obs`) – observables to be reshaped
- `new_shape` (*tuple*) – the new shape of the observable

Returns reshaped observable

Return type *obs*

Notes

This function acts exclusively on the mean value.

2.2 pyobs.concatenate

`pyobs.concatenate`(*x*, *y*, *axis=0*)

Join two arrays along an existing axis

Parameters

- `y` (*x*,) – the two observable to concatenate
- `axis` (*int*, *optional*) – the axis along which the observables will be joined. Default is 0.

Returns the concatenated observable

Return type *obs*

Notes

If x and y contain information from separate ensembles, they are merged accordingly by keeping only the minimal amount of data in memory.

2.3 pyobs.transpose

`pyobs.transpose(x, axes=None)`
Transpose a tensor along specific axes. For an array a with two axes, gives the matrix transpose.

Parameters

- `x` (`obs`) – input observable
- `axes` (*tuple or list of ints, optional*) – If specified, it must be a tuple or list which contains a permutation of [0,1,..,N-1] where N is the number of axes of x. For more details read the documentation of *numpy.transpose*

Returns the transposed observable

Return type *obs*

2.4 pyobs.sort

`pyobs.sort(x, axis=- 1)`
Sort a tensor along a specific axis.

Parameters

- `x` (`obs`) – input observable
- `axis` (*int, optional*) – the axis which is sorted. Default is -1, the
- `axis.` (*last*) –

Returns the sorted observable

Return type *obs*

2.5 pyobs.diag

`pyobs.diag(x)`
Extract the diagonal of 2-D array or construct a diagonal matrix from a 1-D array

Parameters `x` (`obs`) – input observable

Returns the diagonally projected or extended observable

Return type *obs*

CHAPTER

THREE

UNARY OPERATIONS

3.1 Linear functions

sum(x[, axis])	Sum of array elements over a given axis.
trace(x[, offset, axis1, axis2])	Return the sum along diagonals of the array.

3.1.1 pyobs.sum

`pyobs.sum`(*x*, *axis=None*)

Sum of array elements over a given axis.

Parameters

- `x` (obs) – array with elements to sum
- `axis` (*None or int or tuple of ints, optional*) – Axis or axes along which a sum is performed. The default, axis=None, will sum all elements of the input array.

Returns sum along the axis

Return type *obs*

Examples

```
>>> import pyobs
>>> pyobs.sum(a)
>>> pyobs.sum(a,axis=0)
```

3.1.2 pyobs.trace

`pyobs.trace`(*x*, *offset=0*, *axis1=0*, *axis2=1*)

Return the sum along diagonals of the array.

Parameters

- `x` (obs) – observable whose diagonal elements are taken
- `offset` (*int, optional*) – offset of the diagonal from the main diagonal; can be both positive and negative. Defaults to 0.
- `axis2` (*axis1,*) – axes to be used as the first and second axis of the 2-D sub-arrays whose diagonals are taken; defaults are the first two axes of *x*.

Returns the sum of the diagonal elements

Return type *obs*

Notes

If x is 2-D, the sum along its diagonal with the given offset is returned, i.e., the sum of elements *x[i,i+offset]* for all i. If x has more than two dimensions, then the axes specified by *axis1* and *axis2* are used to determine the 2-D sub-arrays whose traces are returned. The shape of the resulting array is the same as that of a with *axis1* and *axis2* removed.

Examples

```
>>> tr = pyobs.trace(mat)
```

3.2 Exponential functions

log (x)	Return the Natural logarithm element-wise.
exp (x)	Return the exponential element-wise.
cosh (x)	Return the Hyperbolic cosine element-wise.
sinh (x)	Return the Hyperbolic sine element-wise.
arccosh (x)	Return the inverse Hyperbolic cosine element-wise.

3.2.1 pyobs.log

`pyobs.log(`*x*`)`

Return the Natural logarithm element-wise.

Parameters `x` (`obs`) – input observable

Returns the logarithm of the input observable, element-wise.

Return type *obs*

Examples

```
>>> logA = pyobs.log(obsA)
```

3.2.2 pyobs.exp

`pyobs.exp(`*x*`)`

Return the exponential element-wise.

Parameters `x` (`obs`) – input observable

Returns the exponential of the input observable, element-wise.

Return type *obs*

Examples

```
>>> expA = pyobs.exp(obsA)
```

3.2.3 pyobs.cosh

pyobs.cosh(*x*)

Return the Hyperbolic cosine element-wise.

Parameters x (obs) – input observable

Returns the hyperbolic cosine of the input observable, element-wise.

Return type *obs*

Examples

```
>>> B = pyobs.cosh(obsA)
```

3.2.4 pyobs.sinh

pyobs.sinh(*x*)

Return the Hyperbolic sine element-wise.

Parameters x (obs) – input observable

Returns the hyperbolic sine of the input observable, element-wise.

Return type *obs*

Examples

```
>>> B = pyobs.sinh(obsA)
```

3.2.5 pyobs.arccosh

pyobs.arccosh(*x*)

Return the inverse Hyperbolic cosine element-wise.

Parameters x (obs) – input observable

Returns the inverse hyperbolic cosine of the input observable, element-wise.

Return type *obs*

Examples

```
>>> B = pyobs.arccosh(obsA)
```

3.3 Special functions

besselk(v, x)	Modified Bessel function of the second kind of real order v, element-wise.

3.3.1 pyobs.besselk

`pyobs.besselk(`v, x`)`

Modified Bessel function of the second kind of real order v, element-wise.

Parameters

- `v` (*float*) – order of the Bessel function
- `x` (`obs`) – real observable where to evaluate the Bessel function

Returns the modified bessel function computed for the input observable

Return type *obs*

CHAPTER

FOUR

LINEAR ALGEBRA ROUTINES

`pyobs.linalg.inv(x)`

Compute the inverse of a square matrix

Parameters `x` (`obs`) – Matrix to be inverted

Returns (Multiplicative) inverse of x

Return type *obs*

Examples

```
>>> from pyobs.linalg import inv
>>> a = pyobs.obs()
>>> a.create('A',data,shape=(2,2))
>>> ainv = pyobs.inv(a)
```

Notes

If the number of dimensions is bigger than 2, x is treated as a stack of matrices residing in the last two indexes and broadcast accordingly.

`pyobs.linalg.eig(x)`

Computes the eigenvalues and eigenvectors of a square matrix observable. The central values are computed using the *numpy.linalg.eig* routine.

Parameters `x` (`obs`) – a symmetric square matrix (observable) with dimensions *NxN*

Returns a vector observable with the eigenvalues and a matrix observable whose columns correspond to the eigenvectors

Return type list of obs

Notes

The error on the eigenvectors is based on the assumption that the input matrix is symmetric. If this not respected, the returned eigenvectors will have under or over-estimated errors.

Examples

```
>>> [w,v] = pyobs.linalg.eig(mat)
>>> for i in range(N):
>>>     # check eigenvalue equation
>>>     print(mat @ v[:,i] - v[:,i] * w[i])
```

`pyobs.linalg.eigLR(x)`

Computes the eigenvalues and the left and right eigenvectors of a square matrix observable. The central values are computed using the *numpy.linalg.eig* routine.

Parameters `x` (`obs`) – a square matrix (observable) with dimensions *NxN*;

Returns a vector observable with the eigenvalues and two matrix observables whose columns correspond to the right and left eigenvectors respectively.

Return type list of obs

Notes

This input matrix is not expected to be symmetric. If it is the usage of *eig* is recommended for better performance.

Examples

```
>>> [l,v,w] = pyobs.linalg.eigLR(mat)
>>> for i in range(N):
>>>     # check eigenvalue equation
>>>     print(mat @ v[:,i] - v[:,i] * l[i])
>>>     print(w[:,i] @ mat - w[:,i] * l[i])
```

`pyobs.linalg.matrix_power(x, a)`

Raises a square symmetric matrix to any non-integer power *a*.

Parameters

- `x` (`obs`) – a symmetric square matrix (observable) with dimensions *NxN*
- `a` (*float*) – the power

Returns the matrix raised to the power *a*

Return type *obs*

Notes

The calculation is based on the eigenvalue decomposition of the matrix.

Examples

```
>>> matsq = pyobs.linalg.matrix_power(mat, 2) # identical to mat @ mat
>>> matsqrt = pyobs.linalg.matrix_power(mat, -0.5)
>>> matsqrt @ mat @ matsqrt # return the identity
```

CHAPTER

FIVE

THE MFIT CLASS

class `pyobs.mfit`(*x*, *W*, *f*, *df*, *v='x'*)

Class to perform fits to multiple observables, via the minimization of the χ^2 function

$$\chi^2 = r^T W r \quad , \quad r_i = y_i - \phi(\{p\}, \{x_i\})$$

with ϕ the model function, p_α the model parameters ($\alpha = 1, \ldots, N_\alpha$) and x_i^μ the kinematic coordinates ($i = 1, \ldots, N$ corresponds to the i-th kinematical point, and μ labels the dimensions). The matrix W defines the metric of the χ^2 function.

The ideal choice for the matrix W is the inverse of the covariance matrix, which defines a correlated fit. Instead a more practical choice for cases where a reliable estimate of covariances is not available, is given by the inverse of the diagonal part of the covariance matrix, which instead defines an uncorrelated fit.

Parameters

- `x` (*array*) – array with the values of the x-axis; 2-D arrays are accepted and the second dimension is intepreted with the index μ
- `W` (*array*) – 2-D array with the matrix W; if a 1-D array is passed, the program inteprets it as the inverse of the diagonal entries of the covariance matrix
- `f` (*function*) – callable function or lambda function defining ϕ; the program assumes x_i^μ correspond to the first arguments
- `df` (*function*) – callable function or lambda function returning an array that contains the gradient of ϕ, namely $\partial\phi(\{p\}, \{x_i\})/\partial p_\alpha$
- `v` (*str, optional*) – a string with the list of variables used in f as the kinematic coordinates. Default value corresponds to x, which implies that f must be defined using x as first and unique kinematic variable.

Notes

Once created the class must be called with at least one argument given by the observables corresponding to the data points y_i. See examples below.

Examples

```
>>> xax=[1,2,3,4]
>>> f=lambda x,p0,p1: p0 + p1*x
>>> df=lambda x,p0,p1: [1, x]
>>> [y,dy] = yobs1.error()
>>> W=1./dy**2
>>> fit1 = mfit(xax,W,f,df)
>>> pars = fit1(yobs1)
>>> print(pars)
0.925(35)    2.050(19)
```

`eval(xax, pars)`

Evaluates the function on a list of coordinates using the parameters obtained from a χ^2 minimization.

Parameters

- `xax` (*array , list of arrays*) – the coordinates x_i^μ where the function must be evaluated. For combined fits, a list of arrays must be passed, one for each fit.
- `pars` (obs) – the observable returned by calling this class

Returns

observables corresponding to the functions evaluated at the coordinates *xax*.

Return type *obs*, list of obs

Examples

```
>>> fit1 = mfit(xax,W,f,df)
>>> pars = fit1(yobs1)
>>> print(pars)
0.925(35)    2.050(19)
>>> xax = numpy.arange(0,10,0.2)
>>> yeval = fit1.eval(xax, pars)
```

`pars()`

Prints the list of parameters

5.1 Additional functionalities

`pyobs.symbolic.diff(f, x, dx)`

Utility function to compute the gradient and hessian of a function using symbolic calculus

Parameters

- `f` (*string*) – the reference function
- `x` (*string*) – variables that are not differentiated; different variables must be separated by a comma
- `dx` (*string*) – variables that are differentiated; different variables must be separated by a comma

Returns (scalar) function f lambda : (vector) function of the gradient of f lambda : (matrix) function of the hessian of f

Return type lambda

Notes

The symbolic manipulation is based on the library *sympy* and the user must follow the *sympy* syntax when passing the argument *f*. The analytic form of the gradient and hessian can be printed by activating the 'diff' verbose flag.

Examples

```
>>> res = diff('a+b*x','x','a,b') # differentiate wrt to a and b
a + b*x
[1, x]
[[0, 0], [0, 0]]
>>> for i in range(3):
>>>     print(res[i](0.4,1.,2.))
1.8
[1, 0.4]
[[0, 0], [0, 0]]
```

CHAPTER

SIX

ADDITIONAL UTILITY FUNCTIONS

`pyobs.random.acrand(`*mu*, *sigma*, *tau*, *N*`)`

Create synthetic autocorrelated Monte Carlo data

Parameters

- `mu` (*int or float*) – the central value
- `sigma` (*float*) – the square root of the variance of the observable in absence of autocorrelations
- `tau` (*float*) – the integrated autocorrelation time
- `N` (*int*) – the number of configurations/measurements

Returns the synthetic data

Return type list

Note: The expected error from a proper autocorrelation analysis is the product of *sigma* with the square root of *tau* divided by *N*.

Examples

```
>>> data = pyobs.random.acrand(0.1234,0.0001,4.0,1000)
>>> obs = pyobs.obs(desc='test-acrand')
>>> obs.create('A',data)
>>> print(obs)
0.12341(26)
```

`pyobs.random.acrandn(`*mu*, *cov*, *tau*, *N*`)`

Create synthetic correlated Monte Carlo 1-D data

Parameters

- `mu` (*list of array*) – the central values of corresponding observable; a 1-D array is expected
- `cov` (*array*) – the covariance matrix of the observable (in absence of autocorrelations); if a 1-D array is passed, a diagonal covariance matrix is assumed
- `tau` (*float*) – the integrated autocorrelation time
- `N` (*int*) – the number of configurations/measurements

Returns

2-D array with the synthetic data, such that each row corresponds to a configuration

Return type numpy.ndarray

CHAPTER SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p