# multinorm Documentation

***Release 0.2***

**Christoph Deil**

**Oct 26, 2018**

# Contents:

*Multivariate Normal Distributions for Humans.*

A Python class to work with model fit results (parameters and the covariance matrix).

- Code: https://github.com/cdeil/multinorm
- Tutorial: multinorm.ipynb
- Documentation: https://multinorm.readthedocs.io
- License: BSD-3-Clause

CHAPTER 1

# Install

This package supports Python 3.6 or later. Python 3.5 or 2.7 or older versions are not supported.

There's nothing platform-specific; Linux, MacOS and Windows are supported.

The required dependencies are numpy, scipy and pandas.

To install `multinorm` use pip:

```
pip install multinorm
```

This will install the required dependencies if you don't have them already:

- numpy
- scipy
- pandas

There are some built-in methods for plotting using matplotlib. That optionally dependency has to be installed separately, *pip install multinorm* will not install matplotlib.

This package consists of a single Python file multinorm.py. Most users will not care about this implementation detail, but if you'd like to copy and "vendor" it for some reason, you can bundle a copy and avoid the extra dependency for just one file and class.

CHAPTER 2

# Getting started

**Note:** For a quick and hands-on introduction, start with the multinorm.ipynb Jupyter notebook tutorial, then continue reading here.

## 2.1 Import

The *multinorm* package offers a single class *MultiNorm*, so you always start like this:

```
from multinorm import MultiNorm
```

## 2.2 Create

To create a *MultiNorm* object, pass a `mean` vector, a `covariance` matrix (both as Numpy arrays) and optionally a list of parameter `names`:

```
from multinorm import MultiNorm
mean = [10, 20, 30]
covariance = [[1, 0, 0], [0, 4, 0], [0, 0, 9]]
names = ["a", "b", "c"]
mn = MultiNorm(mean, covariance, names)
```

Sometimes the mean and covariance are given directly, e.g. in a publication, and you would define them in Python code as shown here, or read them from a file.

However, often you obtain these values as the result of a fit of a parametrised model to data, or estimate them in some other way.

Further examples to create *MultiNorm* objects are here: *Create*

## 2.3 Read only

*MultiNorm* objects should be used read-only objects!

If you need to change something (`mean`, `covariance`, `names`), make a new object!

TODO: make read-only as much as possible, the document remaining caveats!

## 2.4 Analyse

Once you have a *MultiNorm* object representing a multivariate normal distribution, you can access the following properties and methods to analyse it.

The object repr only shows the number of dimensions (number of parameters) `n` of the distribution:

```
>>> mn
MultiNorm(n=3)
```

To see the contents, print the object:

```
>>> print(mn)
MultiNorm(n=3)
names: ['a', 'b', 'c']
mean: [10. 20. 30.]
err: [1. 2. 3.]
cov:
[[1. 0. 0.]
 [0. 4. 0.]
 [0. 0. 9.]]
```

You can access the attributes like this:

```
>>> mn.n
3
>>> mn.mean
array([10., 20., 30.])
>>> mn.cov
array([[1., 0., 0.],
       [0., 4., 0.],
       [0., 0., 9.]])
>>> mn.names
['a', 'b', 'c']
```

The `mean` and `covar` are *numpy.ndarray* objects. To be as accurate as possible, we always cast to 64-bit float on *MultiNorm* initialisation and do all computations with 64-bit floating point precision, even if 32-bit float or integer numbers are passed in.

```
>>> type(mn.mean)
numpy.ndarray
>>> mn.mean.dtype
dtype('float64')
```

The `mean` is a 1-dimensional array, and `cov` is a 2-dimensional array:

```
>>> mn.mean.shape
(3,)
>>> mn.cov.shape
(3, 3)
```

Parameter error vector *err()*:

```
>>> mn.err
array([1., 2., 3.])
```

Precision matrix (the inverse covariance) *precision()*:

```
>>> mn.precision
array([[1.        , 0.        , 0.        ],
       [0.        , 0.25      , 0.        ],
       [0.        , 0.        , 0.11111111]])
```

Correlation matrix *correlation()*:

```
>>> mn.correlation
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

These are just the basic attributes and properties.

We continue with this example on the *Analyse* page and show how to really do some analysis with *MultiNorm* objects and methods.

## 2.5 Plot

Plot ellipse using *to_matplotlib_ellipse()*:

```
import matplotlib.pyplot as plt
mn2 = mn.marginal()
mn2.plot()
```

Further examples to plot *MultiNorm* objects are here: *Plot*

## 2.6 What next?

The *Create*, *Analyse* and *Plot* tutorial pages contain further examples. The *Theory* and *References* pages contain background information and definitions, as well as links to other documents and codes.

The full API documentation is here: *MultiNorm*. Note that you can click on "source" on the right for any method or property, and read the implementation to see what exactly it does. It's usually a few lines of straightforward code using Python and Numpy, so reading the source is recommended.

CHAPTER 3

# Create

As we saw in *Getting started*, to create a *MultiNorm* object, you pass a `mean` vector, a `covariance` matrix (both as Numpy arrays) and optionally a list of parameter `names`:

```
from multinorm import MultiNorm
mean = [10, 20, 30]
covariance = [[1, 0, 0], [0, 4, 0], [0, 0, 9]]
names = ["a", "b", "c"]
mn = MultiNorm(mean, covariance, names)
```

But where do these things come from?

On this page, we look at the most common scenarios.

## 3.1 From fit

TODO: show example using scipy.optimise.curve_fit‘

To use `multinorm`, we first need to fit some parameterised model to obtain a best-fit parameter vector and covariance matrix.

Let’s use scipy.optimize_curve_fit to fit some data.

TODO: show example using iminuit

http://www.statsmodels.org/devel/examples/notebooks/generated/chi2_fitting.html https://github.com/cdeil/pyfit/blob/master/fitting_tutorial/src/tests/chi2_example.py https://lmfit.github.io https://iminuit.readthedocs.io https://sherpa.readthedocs.io

## 3.2 From points

A common way to analyse likelihood or in Bayesian analyses the posterior probability distributions is to use MCMC methods that sample the distribution. E.g. emcee or pymc are Python packages that generate this kind of output.

Estimating the multivariate normal distribution from samples well can be difficult, there are many methods with different trade-offs. We recommend using a different package for this task, e.g. sklearn.covariance.

That said, there is a method *MultiNorm.from_points()* that calls `numpy.std()` to compute the mean vector, and `numpy.cov()` to compute what's sometimes called the "empirical" multivariate normal estimate.

Points should always be given as 2-dimensional arrays with shape `(n_dim, n_points)`.

```
>>> points = mn.sample(size=100, random_state=0)
>>> MultiNorm.from_points(points)
MultiNorm(n=3)
names: ['par_0', 'par_1', 'par_2']
mean: [ 9.87581591 20.21250462 30.30156153]
err: [0.98090098 1.97394775 3.09360932]
cov:
[[ 0.96216674 -0.04439635  0.33802118]
 [-0.04439635  3.89646972 -0.45369176]
 [ 0.33802118 -0.45369176  9.57041861]]
```

## 3.3 From publication

TODO: show example how to take covar (or par errors) from a publication or blog post, i.e. as inputs.

## 3.4 From product

TODO: document *MultiNorm.from_product()*

## 3.5 Make example

TODO: document *MultiNorm.make_example()*

CHAPTER 4

# Analyse

## 4.1 Example

A basic example and properties of *MultiNorm* were shown in *Getting started*.

On this page we continue with analysis methods using same example:

```
from multinorm import MultiNorm
mean = [10, 20, 30]
covariance = [[1, 0, 0], [0, 4, 0], [0, 0, 9]]
names = ["a", "b", "c"]
mn = MultiNorm(mean, covariance, names)
```

## 4.2 Scipy

For most computations, *Multinorm* uses scipy. The *scipy()* property is a frozen scipy.stats.multivariate_normal object. It is cached, accessing it multiple times doesn't incur any extra computations. Note that scipy.stats.multivariate_normal has a `cov_info` object, which contains a covariance matrix decomposition which is computed once and cached. It is at this time undocumented, but it is a public property and is what powers most computations in the scipy and in this class.

```
>>> s = mn.scipy
>>> type(s)
scipy.stats._multivariate.multivariate_normal_frozen
```

To present a consistent and complete API, *MultiNorm* re-exposes the functionality of scipy.stats.multivariate_normal, it is a wrapper.

Draw random samples from the distribution using *sample()*:

```
>>> points = mn.sample(size=2, random_state=0)
>>> points
array([[10.97873798, 20.80031442, 35.29215704],
       [ 9.02272212, 23.73511598, 36.7226796 ]])
```

Points are always given as arrays with shape `(n_dim, n_points)`.

Evaluate the probability density function (PDF), call *`pdf()`*:

```
>>> mn.pdf(points)
array([1.27661616e-03, 9.31966590e-05])
```

For `log(pdf)` (natural logarithm), call *`logpdf()`*:

```
>>> mn.logpdf(points)
array([-6.66354232, -9.28079868])
```

There is also a `cdf` and `logcdf` method for the cumulative distribution function, as well as `entropy`. Since these are rarely needed, we didn't wrap them. But you can still access them via the *`scipy()`* property.

## 4.3 Marginal

TODO: marginal

## 4.4 Conditional

TODO: conditional

## 4.5 Error propagation

TODO: to_uncertainties, to_soerp, to_mcerp

## 4.6 Sigmas

TODO: sigma_distance

CHAPTER 5

# Plot

The `multinorm` package contains a few plot methods using matplotlib.

This page shows examples of those, as well as how to do some common plots by computing the relevant numpy arrays and passing them to matplotlib directly.

tbd

See https://stackoverflow.com/questions/29432629/correlation-matrix-using-pandas

# CHAPTER 6

# Theory

In this section, we give a bit of theory background concerning the methods used in `multinorm`. We give the formulae used, and a reference to where the formula and a derivation and discussion can be found.

**Note:** The multivariate normal distribution has very nice mathematical properties. Every derived quantity follows either again a multivariate normal distribution or a chi-squared distribution.

## 6.1 Marginal distribution

The marginal distribution can be obtained with the *`marginal()`* method.

You can think of the marginal distribution as the distribution obtained by simply ignoring some of the parameters, or by "projecting" the $N$-dimensional distribution onto the lower-dimensional subspace of parameters of interest.

The marginal distribution of the multivariate normal is again a multivariate normal distribution.

It can be obtained simply by keeping only the parameters of interest in the mean vector and covariance matrix (drop the parameters that are being marginalised out).

See here.

## 6.2 Conditional distribution

The conditional distribution can be obtained with the *`conditional()`* method.

The conditional distribution is given by the "slice" in the $N$-dimensional distribution when fixing some of the parameters.

The conditional distribution of the multivariate normal is again a multivariate normal distribution.

It can be obtained by partitioning the mean $\mu$ and covariance $\Sigma$ of the $N$-dimensional distribution into two part, corresponding to the parameters that are fixed, and that are kept free.

The formulae to obtain the mean and covariance of the conditional distribution are given here.

## 6.3 Fix parameters

This method is used e.g. in MINUIT, see Section 1.3.1 here: http://lmu.web.psi.ch/docu/manuals/software_manuals/minuit2/mnerror.pdf

As far as I can tell, it gives the same results as conditional (see *test_conditional_vs_fix*).

TODO: work out the math of why that is the case and document it here.

Add note that for MVN the covar matrix for conditional doesn't depend on parameter values.

TODO: document and make example in the analyse section using iminuit.

## 6.4 Product distribution

TODO: improve this section: https://github.com/cdeil/multinorm/issues/13

We should give the full equations, the ones below are the special case for distributions without correlations.

The approximation we will use can be found in many textbooks, e.g. Section 5.6.1 stats book. given $n$ Gaussian likelihood estimates with parameter estimates $x_i$ and known parameter errors $\sigma_i$:

$$p(\mu|x_i, \sigma_i),$$

if we define "weights" as inverse square of errors

$$w_i = 1/\sigma_i^2, \sigma_i = 1/\sqrt{w_i},$$

then the from_product maximum likelihood estimate error is given by (Equation 5.50):

$$\mu_0 = \frac{\sum w_i x_i}{\sum w_i}$$

and the from_product measurement parameter error is given by

$$w = \sum w_i.$$

## 6.5 Sigmas

For the one-dimensional normal distribution $N(\mu, \sigma)$ the probability content within $n * \sigma$ is given by roughly 68% for $n = 1$, 95% for $n = 2$ and 99.7% for $n = 3$.

What's the equivalent for the $N$-dimensional normal distribution?

For a given mean $\mu$ and covariance $\Sigma$ and point $p$ one can define a distance $d$ via

$$d = \sqrt{(p - \mu)^T \Sigma^{-1} (p - \mu)}.$$

The set of equal-distance points is an ellipsoidal surface and has the property that all points on it have equal probability density. It is the equivalent of the distance $d = (p - \mu)/\sigma$, i.e. the number of standard deviations $d = n * \sigma$ from the mean.

However, the probability content for a given $n * \sigma$ is lower for the $N$-dimensional distribution. It turns out that $d^2$ has a $\chi^2$ distribution with $N$ degrees of freedom:

$$P(d^2) = \chi^2(d^2, N)$$

That means you can compute the probability content using `scipy.stats.chi2` like this:

```
>>> import numpy as np
>>> from scipy.stats import chi2
>>> n_sigma = np.array([1, 2, 3])
>>> chi2.cdf(n_sigma ** 2, 1)
array([0.68268949, 0.95449974, 0.9973002 ])
>>> chi2.cdf(n_sigma ** 2, 2)
array([0.39346934, 0.86466472, 0.988891  ])
>>> chi2.cdf(n_sigma ** 2, 3)
array([0.19874804, 0.73853587, 0.97070911])
```

Note that the 1 sigma ellipse in 2D has probability content 39%, in 3D it's only 20%, and it gets smaller and smaller for higher dimensions.

Also see https://stats.stackexchange.com/questions/331283

For further information see the Wikipedia Mahalanobis distance page.

The *MultiNorm.to_matplotlib_ellipse* takes an `n_sigma` option, and will return an ellipse that matches the points with Mahalanobis distance $d^2 = n * \sigma$.

See also sigma in the corner.py docs.

# CHAPTER 7

# API

**class** multinorm.**MultiNorm**(*mean=None*, *cov=None*, *names=None*)

Multivariate normal distribution.

Given `n` parameters, the `mean` and `names` should be one-dimensional with size `n`, and `cov` should be a two-dimensional matrix of shape `(n, n)`.

Documentation for this class:

- Tutorial Jupyter notebook: multinorm.ipynb
- Documentation: *Getting started*, *Create*, *Analyse*
- Equations and statistics: *Theory*

Note that MultiNorm objects should be used read-only, almost all properties are cached. If you need to modify values, make a new *MultiNorm* object.

**Parameters**

- **mean** (*numpy.ndarray*) – Mean vector
- **cov** (*numpy.ndarray*) – Covariance matrix
- **names** (*list*) – Python list of parameter names (str). Default: use "par_i" with `i = 0 .. N - 1`.

**conditional**(*pars*, *values=None*)

Conditional *MultiNormal* distribution.

Resulting lower-dimensional distribution obtained by fixing `pars` to `values`. The output distribution is for the other parameters, the complement of `pars`.

See *Conditional distribution*.

**Parameters**

- **pars** (*list*) – Fixed parameters (indices or names)
- **values** (*list*) – Fixed parameters (values). Default is to use the values from `mean`.

**Returns** *MultiNorm* – Conditional distribution

**correlation**
Correlation matrix (*pandas.DataFrame*).

Correlation $C$ is related to covariance $\Sigma$ via:

$$C_{ij} = \frac{\Sigma_{ij}}{\sqrt{\Sigma_{ii}\Sigma_{jj}}}$$

**cov**
Covariance matrix (*pandas.DataFrame*).

**drop** (*pars*)
Drop parameters.

This simply removes the entry from the *mean* vector, and the corresponding column and row from the *cov* matrix.

The computation is the same as `MultiNorm.marginal()`, only here the parameters to drop are given, and there the parameters to keep are given.

**Parameters** **pars** (*list*) – Parameters to fix (indices or names)

**err**
Error vector (*pandas.DataFrame*).

Defined as $\sigma_i = \sqrt{\Sigma_{ii}}$.

**fix** (*pars*)
Fix parameters.

See *Fix parameters*.

**Parameters** **pars** (*list*) – Parameters to fix (indices or names)

**classmethod from_err** (*mean=None*, *err=None*, *correlation=None*, *names=None*)
Create *MultiNorm* from parameter errors.

With errors $\sigma_i$ this will create a diagonal covariance matrix with

$$\Sigma_{ii} = \sigma_i^2$$

For a given `correlation`, or in general: this will create a *MultiNormal* with a covariance matrix such that it's `err` and `correlation` match the one specified here (up to rounding errors).

**Parameters**

- **mean** (*numpy.ndarray*) – Mean vector
- **err** (*numpy.ndarray*) – Error vector
- **correlation** (*numpy.ndarray*) – Correlation matrix
- **names** (*list*) – Parameter names

**classmethod from_points** (*points*, *names=None*)
Create *MultiNorm* from parameter points.

Usually the points are samples from some distribution and creating this *MultiNorm* distribution is an estimate / approximation of that distribution of interest.

See: *From points*.

**Parameters** **points** (*numpy.ndarray*) – Array of data points with shape `(n, 2)`.

**classmethod from_product** (*distributions*)
Create *MultiNorm* as product distribution.

This represents the joint likelihood distribution, assuming the individual distributions are from independent measurements.

See *Product distribution* .

**Parameters** **distributions** (*list*) – Python list of *MultiNorm* distributions.

**Returns** *MultiNorm* – Product distribution

**logpdf** (*points*)
Natural log of PDF.

Calls scipy.stats.multivariate_normal.

**classmethod make_example** (*n_par=3*, *n_fix=0*, *random_state=42*)
Create example *MultiNorm* for testing.

This is a factory method that allows the quick creation of example *MultiNormal* with any number of parameters for testing.

See: *Make example*.

**Parameters**

- **n_par** (*int*) – Number of parameters
- **n_fix** (*int*) – Number of fixed parameters in addition to `n_par`.
- **random_state** – Seed (int) - default: 42 Put `None` to choose random seed. Can also pass *numpy.random.RandomState* object.

**marginal** (*pars*)
Marginal *MultiNormal* distribution.

See *Marginal distribution*.

**Parameters** **pars** (*list*) – List of parameters (integer indices)

**Returns** *MultiNorm* – Marginal distribution

**mean**
Mean vector (*pandas.Series*).

**n**
Number of dimensions of the distribution (int).

Given by the number of parameters.

**names**
Parameter names (*list* of *str*).

**parameters**
Parameter table (*pandas.DataFrame*).

Index is “name”, columns are “mean” and “err”

**pdf** (*points*)
Probability density function.

Calls scipy.stats.multivariate_normal.

**precision**
Precision matrix (*pandas.DataFrame*).

The inverse of the covariance matrix.

Sometimes called the "information matrix" or "Hesse matrix".

**sample**(*size=1*, *random_state=None*)
Draw random samples.

Calls scipy.stats.multivariate_normal.

**scipy**
Frozen scipy.stats.multivariate_normal distribution object.

A cached property. Used for many computations internally.

**sigma_distance**(*point*)
Number of standard deviations from the mean (float).

Also called the Mahalanobis distance. See *Sigmas*.

**to_matplotlib_ellipse**(*n_sigma=1*, ***kwargs*)
Create matplotlib.patches.Ellipse.

See examples in *Plot*.

**Parameters** **n_sigma** (*int*) – Number of standard deviations. See *Sigmas*.

**to_uncertainties**()
Convert to uncertainties objects.

A tuple of numbers with uncertainties (one for each parameter) is returned.

The uncertainties package makes it easy to do error propagation on derived quantities.

See examples in *Analyse*.

CHAPTER 8

# References

## 8.1 Definitions

In the `multinorm` package, we use the following variable names:

- `MultiNorm` - the multivariate normal (a.k.a. Gaussian) distribution
- `n` - the number of dimensions, i.e. number of parameters. Math: $n$
- `mean` - the vector of mean values of the distribution. Math: $\mu$
- `cov` - covariance matrix of the distribution. Math: $\Sigma$
- `precision` - precision matrix. Math: $\Sigma^{-1}$

## 8.2 Documents

Some useful references for multivariate normal distributions:

- Wikipedia multivariate normal
- Wikipedia Mahalanobis distance

## 8.3 Codes

Other codes related to multivariate normal distributions:

- Wolfram MultinormalDistribution
- numpy.random.multivariate_normal
- scipy.stats.multivariate_normal
- sklearn.covariance

- uncertainties
- statsmodels

CHAPTER 9

# Changes

This is the changelog for `multinorm`.

You can always find the latest release and all previous versions at https://pypi.org/project/multinorm/

## 9.1 0.3.dev

- In development, coming soon ...
- Goal: first complete version of the tutorial notebook and docs

## 9.2 0.2

- Released Oct 26, 2018
- API now mostly uses pandas objects, pandas is now a core dependency
- *MultiNorm* is now full of cached properties, must be used read-only
- Many methods added and improved, more tests and docs
- Added a tutorial notebook *multinorm.ipynb*
- Dropped Python 2 support, require Python 3.6 or later

## 9.3 0.1

- Released Oct 19, 2018
- First version

# CHAPTER 10

# Contribute

This package is very new, there hasn't been any user feedback or review yet. Very likely the API and implementation can be improved.

Please give feedback to help make it better!

## 10.1 Github

Contributions to *multinorm* are welcome any time on Github: https://github.com/cdeil/multinorm

- If you find an issue, please file a bug report.
- If you're missing a feature, please file a request.
- If you have the skills and time, please send a pull request.

Pull requests should be small and easy to review. If the work takes more than an hour, please open an issue describing what you plan to do first to get some feedback.

## 10.2 Develop

To work on `multinorm`, first get the latest version:

```
git clone https://github.com/cdeil/multinorm.git
cd multinorm
```

Everything is where you'd expect it, i.e. the files to edit are:

- Code: multinorm.py
- Tests: test_multinorm.py
- Docs: RST files in docs

## 10.3 Install

To hack on `multinorm`, you need to have a development environment with all packages and tools installed.

If you're using `conda`, use this:

```
conda env create -f environment.yml
conda activate multinorm
```

If you're using `pip`, you can use `pipenv` like this:

```
pip install pipenv
pipenv install
pipenv shell
```

With the virtual environment active, run this command:

```
pip install -e .
```

This installs `multinorm` in editable mode, meaning a pointer is put in your site-packages to the current source folder, so that after editing the code you only have to re-start python and re-run to get this new version, and not run an install command again.

## 10.4 Tests

Run all tests:

```
pytest -v
```

Run tests and check coverage:

```
pytest -v --cov=multinorm --cov-report=html
open htmlcov/index.html
```

## 10.5 Code style

We use the black code style. To apply it in-place to all files:

```
black .
```

## 10.6 Docs

To build the docs:

```
cd docs
make clean && make html
open _build/html/index.html
```

Then for any other tasks go back to the top level of the package:

```
cd ..
```

## 10.7 Release

To make a release for this package, follow the following steps

1. check that the tests and docs build are OK
2. check via `git tag` or at https://pypi.org/project/multinorm what the next version should be
3. `git clean -fdx`
4. `git tag v0.1` (substitute actual version number here and in the following steps)
5. `python setup.py build sdist`
6. check the package in `dist` (should automate somehow)
7. `twine upload dist/*`
8. `git push --tags`

We should automate this. I didn't have time yet to try them out, but these look interesting:

- https://github.com/pyscaffold/pyscaffold
- https://github.com/regro/rever
- https://github.com/noirbizarre/bumpr

# Python Module Index

## m

# Index