Journal of Statistical Software

October 2014, Volume 61, Issue 5. *http://www.jstatsoft.org/*

Natter: A Python Natural Image Statistics Toolbox

Fabian H. Sinz
University Tübingen

Jörn-Philipp Lies
University Tübingen

Sebastian Gerwinn
OFFIS, Oldenburg

Matthias Bethge
University Tübingen

Abstract

The statistical analysis and modeling of natural images is an important branch of statistics with applications in image signaling, image compression, computer vision, and human perception. Because the space of all possible images is too large to be sampled exhaustively, natural image models must inevitably make assumptions in order to stay tractable. Subsequent model comparison can then filter out those models that best capture the statistical regularities in natural images. Proper model comparison, however, often requires that the models and the preprocessing of the data match down to the implementation details. Here we present the **Natter**, a statistical software toolbox for natural images models, that can provide such consistency. The **Natter** includes powerful but tractable baseline model as well as standardized data preprocessing steps. It has an extensive test suite to ensure correctness of its algorithms, it interfaces to the modular toolkit for data processing toolbox **MDP**, and provides simple ways to log the results of numerical experiments. Most importantly, its modular structure can be extended by new models with minimal coding effort, thereby providing a platform for the development and comparison of probabilistic models for natural image data.

Keywords: natural image statistics, L_p-spherically symmetric distributions, L_p-nested symmetric distributions, parameter estimation, inference, sampling, Python.

1. Introduction

The statistics of natural images are studied from many perspectives and for many possible applications. Early investigations, motivated by the need for an efficient transmission of television signals, noticed that generic natural images exhibit robust statistical properties like a $1/f$ power spectrum (Kretzmer 1952; Schreiber 1956; Deriugin 1956). In the context of vision science the statistics of natural images has become the subject of increasing interest in the late eighties and early nineties (Buchsbaum and Gottschalk 1983; Field 1987; Burton and Moorhead 1987; Ruderman and Bialek 1994). Since then, the field of natural image statistics has produced a variety of models and approaches ranging from modeling whole images by Markov random fields (Besag 1986; Winkler 1995; Hosseini, Sinz, and Bethge 2010), scale mixtures thereof (Lyu and Simoncelli 2007), or infinitely divisible random processes and dead leaf models (Serra 1982; Jeulin, Villalobos, and Dubus 1995; Zhu, Wu, and Mumford 1997; Mumford and Gidas 2001; Lee, Mumford, and Huang 2001; Chainais 2007), to modeling small patches sampled from natural images using linear generative models with factorial or group factorial hidden sources (Olshausen and Field 1996; Bell and Sejnowski 1997; Hyvärinen and Hoyer 2000), Gaussian scale mixtures (Wainwright and Simoncelli 2000), or Boltzmann machines, deep-belief networks, product of experts and more general models with more involved hidden structures (Hyvärinen, Hurri, and Vaeyrynen 2003; Roth and Black 2005; Osindero, Welling, and Hinton 2006; Welling and Gehler 2005; Osindero and Hinton 2008; Karklin and Lewicki 2008; Zoran and Weiss 2009; Ranzato and Hinton 2010; Ranzato, Krizhevsky, and Hinton 2010).

Independent of whether the motivation for studying natural image statistics is image compression, image processing, image content inference, or a better understanding of biological sensory systems, the challenge in developing good probabilistic models stems from the fact that even small patches of gray-scale images are too high-dimensional to be sampled exhaustively: for example, the histogram for an 8 bit 3×3 gray-scale image has already 2^{72} possible states which is clearly beyond the reach of any brute-force histogram estimation. To tackle this problem, it is therefore necessary to assume a certain structure in the design of statistical models. This raises the problem of how to evaluate and compare the performance of different models which implement these assumptions. In terms of probabilistic modeling, a natural choice is to use the log-likelihood of different models for unseen test data. This approach is equivalent to measuring the Kullback-Leibler divergence between the true and the model distribution (Kullback and Leibler 1951). Unfortunately, many statistical models proposed for natural images have an intractable normalization constant which makes likelihood evaluation hard or intractable. A common approach to avoid these difficulties is to use related evaluation measures like denoising performance or – a less objective – visual inspection of samples from the models.

An important step towards building better probabilistic models for natural images is to devise simple tractable models that can serve as building blocks for more involved models and set lower bounds on their performance at the same time. Possible choices for such models are the class of L_p-spherically symmetric models and the class of L_p-nested symmetric models (Gupta and Song 1997; Sinz and Bethge 2009; Sinz, Simoncelli, and Bethge 2009b). These distribution classes contain common models like the Gaussian, independent component analysis (ICA) (Hyvärinen and Oja 1997; Comon 1994), and some independent subspace analysis (ISA) models (Hyvärinen and Hoyer 2000; Hyvärinen and Koester 2007). However, they also offer enough flexibility to outperform these special cases while still being tractable (Sinz and Bethge 2009). Apart from choosing a benchmark class of distributions, reliable model comparison requires that preprocessing steps and models match down to the implementation details in order to avoid occasional discrepancies in the evaluation across labs. While many authors publish the code for their specific model, there is no general toolbox devoted to natural image statistics that could provide such a consistency. Apart from facilitating model comparison, such a toolbox would also minimize the additional coding effort during the development of new models.

Here, we describe the **Natter** toolbox for natural image statistics written in Python (van Rossum *et al.* 2014). In short, the main goals of the **Natter** are to

- provide an easy to use framework for model comparison of natural image models;
- design a framework that facilitates the development of new models.

To that end, the **Natter** provides

- a library of implemented models for natural image statistics that provide a firm baseline for model comparisons;
- standard operations which are typically used in a data preprocessing pipeline for natural image modeling.

The **Natter** is not meant to be a comprehensive collection of all existing natural image models but provides a simple framework for integrating new models and compare it to a number of baseline models such as independent component analysis (ICA), independent subspace analysis (ISA), L_p-spherically symmetric models, and L_p-nested symmetric models that are provided by the **Natter**. These classes of models are quite general so that the currently implemented distributions of the **Natter** cover many spherically symmetric distributions such as Gaussians and Gaussian scale mixtures (GSM), elliptically contoured distributions, L_p-spherically symmetric models such as the p-generalized normal distribution, L_p-nested symmetric distributions, independent component analysis (ICA), independent subspace analysis (ISA), and mixtures of the aforementioned models (see Figure 1). The key to this wide variety of models is their modular structure. For example, each L_p-spherically symmetric distribution is characterized by its radial distribution and its particular value of p which controls the contour lines of the joint density. When setting $p = 2$, one obtains all spherically symmetric distributions which contain GSMs. Setting the radial distribution to a χ-distribution, one obtains a Gaussian. Any other combination of parameters yields a different probability density. The **Natter** incorporates this modular structure such that new models can be explored with minimal coding effort. To our knowledge, the **Natter** is the only toolbox that provides fitting of L_p-spherically symmetric distributions in that generality, and it is the only toolbox that allows to fit L_p-nested symmetric distribution. Finally, extensive tests ensure the correctness and robustness of the quantitative results, and a comprehensive API documentation generated with **sphinx** (Brandl 2014) makes it easy to use.

The paper is structured as follows: We start with a general description of the class architecture. Then we describe how to use the **Natter** at a typical toolchain that compares different models on natural image patches. We start with loading, sampling, and preprocessing data from images in Section 3 and demonstrate how to train and evaluate models using examples of increasing complexity in Section 4. Finally, we show how to add new models in Section 5, describe nonlinear transformations and log-determinants in Section 6 and describe auxiliary features in Section 7.

All code examples can be found in the `Examples` directory of the **Natter**. The **Natter** comes with a comprehensive API documentation found under `doc` in the **Natter** directory. We suggest to open it before proceeding to the examples below.

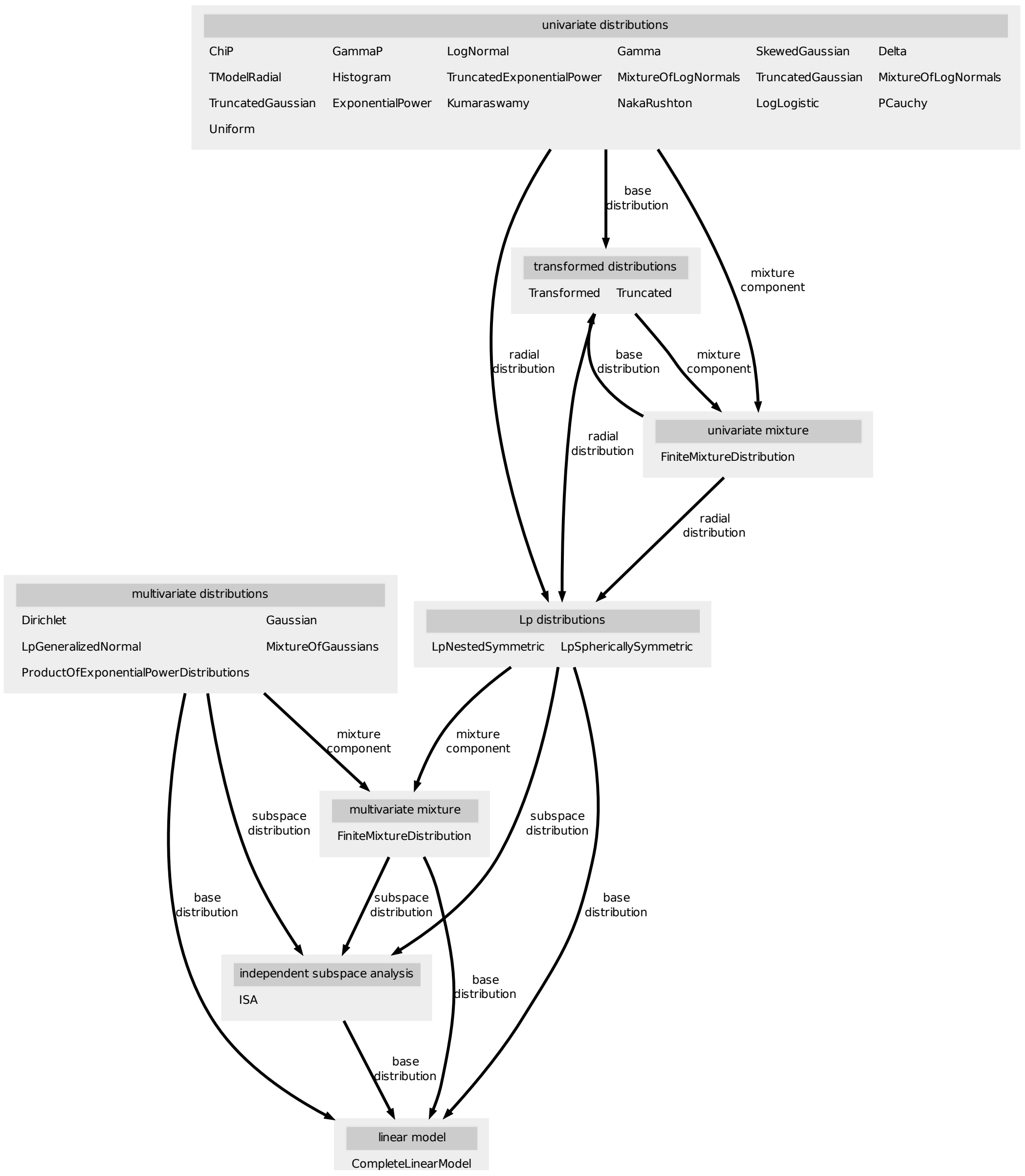

Figure 1: Distribution modularity in the **Natter**: Listed objects are classes contained in the **Natter**. Most implemented distributions are univariate distributions. New univariate distributions can be generated by transforming them (e.g., a Gaussian into a log-normal distribution) or using them as components in univariate mixture distributions. Univariate distributions are radial distributions of the L_p-spherically symmetric and L_p-nested distributions, which are multivariate distributions. Multivariate distributions can be combined in a mixture distribution, as distributions on subspaces in an ISA model, or as distributions on the filter responses in a complete linear model. Using this modularity, the distributions in the **Natter** cover many distributions that are frequently used in natural image modeling like spherically symmetric models (including Gaussians and Gaussian scale mixtures), elliptically contoured distributions and mixtures thereof, many L_p-spherically symmetric distributions such as the p-generalized Gaussian (which becomes ICA for natural images when combined with the complete linear model), and L_p-nested distributions.

2. General design and class hierarchy

As can be seen in Figure 2, most of the functionality of the **Natter** is based on three object types: '`natter.DataModule.Data`', '`natter.Transforms.Transform`', and '`natter.Distributions.Distribution`'.

The object '`natter.DataModule.Data`' holds a two-dimensional **NumPy** (Oliphant 2006) array that contains the single examples as columns. We encapsulated the array in an extra object to provide additional functions and to overload the multiplication operator such that filters ('`Transform`' objects) can be conveniently applied to data. In addition, the '`natter.DataModule.Data`' object maintains a history of processing steps applied to it and can also provide different representations of itself since it inherits from '`natter.Logging.LogToken`' (see below).

All children of the '`natter.Transforms.Transform`' implement transformations of data. '`LinearTransform`' objects contain a matrix of filters while '`NonlinearTransform`' objects implement nonlinear functions applied to single data points. Both objects have corresponding factories, that generate common transforms. The '`natter.Transforms.Transform`' class defines several abstract functions that need to be implemented by its children. New transforms can be added without implementing these functions. However, if they are needed by the internal mechanisms of the **Natter**, an exception will be raised telling the user to implement this method.

'`Transform`' objects can be applied to data or concatenated with other transforms via the multiplication operator. Apart from a convenient user interface, the main reason why we encapsulated transforms in an extra object is that the '`Transform`' objects can automatically keep track of the log-determinant of transforms' Jacobian. This quantity is important when likelihoods for the original data need to be computed when a probability distribution was fitted on transformed data. The same applies to entropy computations. While the log-determinant of the Jacobian can still easily be computed for linear transforms, it becomes more intricate for nonlinear transforms, where the average log-determinant of the Jacobian is needed. When implementing new '`NonlinearTransform`' objects, the user needs to specify the Jacobian of the single transform. The **Natter** then computes the log-determinant of the Jacobian for concatenations by using the chain rule. For '`LinearTransform`' a specification of the Jacobian is not necessary since it is simply the filter matrix.

'`Data`' and '`LinearTransform`' objects both support access to elements via slicing (see the example in Section 3).

The core of the **Natter** are objects in `natter.Distributions` which all inherit from '`natter.Distributions.Distribution`'. These objects implement probability densities on data and their respective fitting routines. Like the '`Transform`' class, the '`Distribution`' class defines several abstract methods like `pdf`, `cdf`, etc. which need to be implemented by its children. However, like for the '`Transform`' class, new children do not need to implement these methods. If an internal mechanism of the **Natter** needs them, an exception will be raised telling the user to implement it. This means that new distributions can be generated with minimal effort since the user implements only those methods she needs. The abstract methods are inherited and provide meaningful error messages when the (missing) method is called.

Finally, almost all object in the **Natter** inherit from '`natter.Logging.LogToken`'. This class defines abstract methods that (when implemented) yield representations of the objects in

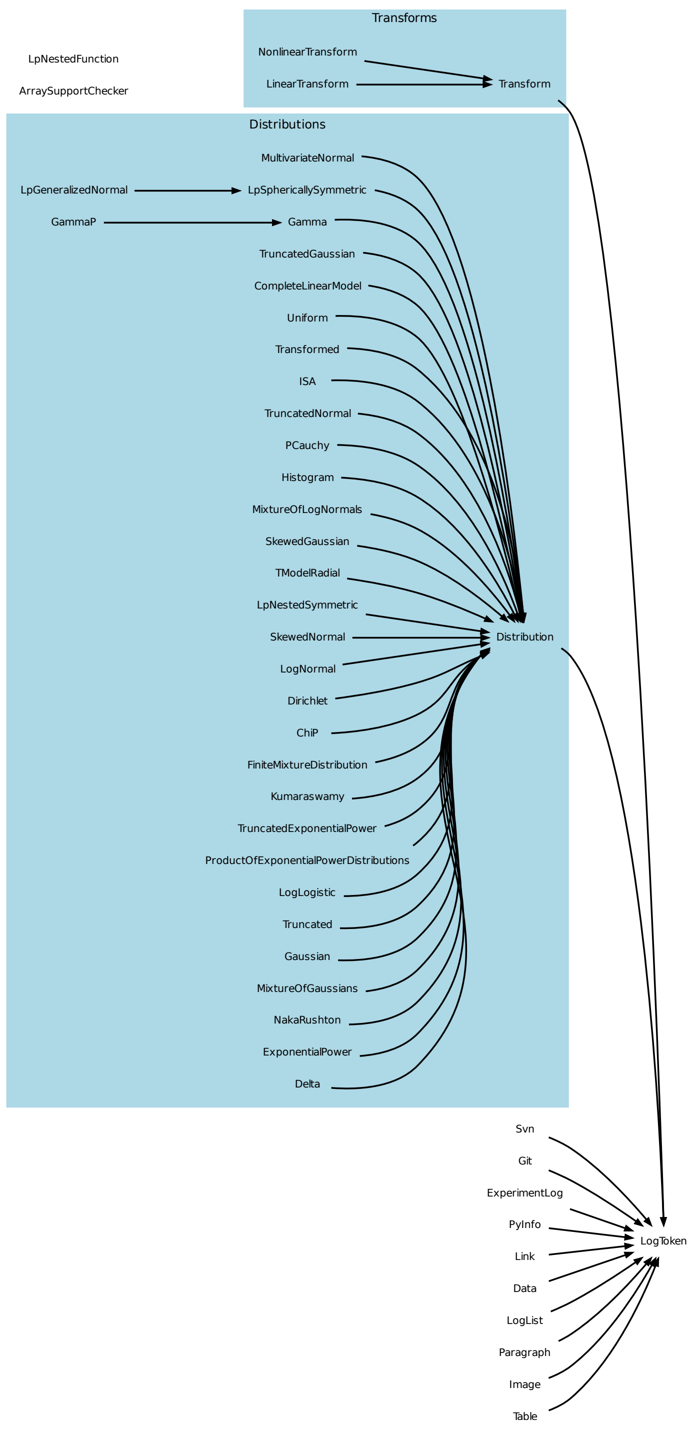

Figure 2: Class hierarchy of the **Natter**: Almost all objects inherit from '`LogToken`' which is used for semi-automatic generation of reports for numerical experiments (see Section 7.1). All distributions inherit from '`Distribution`' which defines several abstract methods implemented by the children. Similarly, the '`Transform`' object defines abstract methods for linear and nonlinear transforms.

HTML, TXT, or LaTeX. These methods are used by a semi-automatic logging module that can generate reports for numerical experiments (see Section 7.1).

3. Data handling

3.1. Data loading

There are four ways to create a '`Data`' object in the **Natter**:

1. load it from a file;
2. sample it from a collection of images;
3. sample it from an existing generative model;
4. or initialize it from a **NumPy** array.

In this section, we will focus on the first two options. The necessary functions are provided by the modules `DataModule.DataLoader` and `DataModule.DataSampler` which must be imported in Python.

The simplest way to load data is from an ASCII file with whitespace-separated numbers. As a convention, each column is treated as a data sample. The `DataLoader` module provides a generic loading function that tries to guess the right data format from the file's extension. The datafile in the following example can be downloaded from `http://www.bethgelab.org/software/natter/`.

```
>>> from natter.DataModule import DataLoader
>>> dat = DataLoader.load('hateren8x8_train_No1.dat')
```

Custom data loading routines can easily be implemented by adding the respective function to the `DataModule.DataLoader`.

Instead of loading the data from a file, the patches can directly be sampled from images via the `DataSampler` module. In our example, we sample 50,000 patches from the van Hateren database (Van Hateren and Van Der Schaaf 1998). The following example needs the `*.iml` images from the van Hateren database which is hosted at `http://www.bethgelab.org/datasets/vanhateren/`.

```
>>> from natter.DataModule import DataSampler
>>> from natter.Auxiliary import ImageUtils
>>> DATADIR = './'
>>> loadFunc = ImageUtils.loadHaterenImage
>>> sampleFunc = DataSampler.img2PatchRand
>>> noSamples = 50000
>>> patchSize = 8
>>> myIter = DataSampler.directoryIterator(DATADIR, noSamples, patchSize,
...   loadFunc, sampleFunc)
>>> dat = DataSampler.sample(myIter, noSamples)
```

`DataSampler.directoryIterator` creates an iterator, that samples `noSamples` patches evenly from all images in the specified directory. Each image is loaded with the `loadFunc` function. The patches are selected from the image via the `sampleFunc` function. For instance, `sampleFunc` could select images randomly or from regions with a specified contrast. By replacing one of those functions by another function with different behavior, the data acquisition can easily be adapted to the specific needs. The module `natter.DataModule.DataSampler` contains several other ways of fetching data from images.

Apart from their memory efficiency, this flexibility is the reason why we implemented sampling from data sources via iterators in Python. We cannot foresee from which image database or from which format future users might want to import data: For example, they may want to draw random samples from images, simulate eye movement on images, fetch data from internet resources, databases, or a camera. Iterators can cope with this great demand of flexibility. The iterator simply needs to return one data sample at a time when its `next()` function is called. The creation of the '`Data`' object is taken over by the **Natter**.

3.2. Preprocessing

The first preprocessing step after a '`Data`' object has been created is usually to subtract the pixel mean or median, to remove the DC component, and to whiten the data.

```
>>> from natter.Transforms import LinearTransformFactory
>>> mu = dat.center()
>>> D = dat.makeWhiteningVolumeConserving()
>>> FDCAC = LinearTransformFactory.DCAC(dat)
>>> FAC = FDCAC[1:, :]
>>> dat = FAC * dat
>>> FwPCA = LinearTransformFactory.wPCA(dat)
>>> dat = FwPCA * dat
```

The center method returns the mean (or median) and can also be called with a prescribed center value, for example when the mean computed on training data should also be used to center the test data.

The rescaling step is motivated by entropy computations which play an important role for model comparisons. One drawback of the joint differential entropy is that it is not invariant under rescaling of the data. In order to be able to compare entropies between different datasets one needs to fix the scale in some way. We propose to globally rescale the data by a single scalar such that whitening has determinant one. While the choice of rescaling is arbitrary, this particular choice is convenient for entropy and log-likelihood computations because it puts the joint differential entropies on a common scale. The **Natter** provides a method to rescale the data such that whitening has determinant one (for further details on the rescaling see Eichhorn, Sinz, and Bethge 2009). The method returns the singular values of the data which can also be passed to it, for example to rescale test data with the same factor as the training data.

While many statistical properties of natural images are quite stable, the constant part of the image patches (the DC component) can vary drastically from image to image. Therefore, a common preprocessing step is to remove the DC component from the patches and either ignore it or treat it separately in further analyses. One possible way to accomplish this, is to simply subtract the constant part from each patch. However, in this case the data-covariance matrix becomes rank-deficient. This can be undesirable e.g., if one wants to apply whitening as another preprocessing step. Oftentimes, a more convenient way to remove the DC component is to project the data onto the (DC-)orthogonal space. The **Natter** provides an orthogonal matrix whose first row projects the data on the DC component such that the $n-1$ last rows live in the orthogonal complement of the DC component (see Eichhorn *et al.* 2009). The advantage of projecting out the DC component with an orthogonal matrix is that the determinant of an orthogonal matrix is one which means that it does not need to be taken into account for likelihood evaluations on data: if $\mathbf{x}$ is a vectorized input patch, $\mathbf{y} = W\mathbf{x}$ with orthogonal W, and ρ_Y is a density on $\mathbf{y}$, then $\rho_X(\mathbf{x}) = \rho_Y(W\mathbf{x})$ since $|\det W| = 1$. For the same reason, an orthogonal matrix does not change the joint entropy. The **Natter** provides such a filter object via `LinearTransformFactory.DCAC(dat)`. The first row of the filter corresponds to the DC component, the remaining rows to the AC components. In the example, we first created the filter, then extract the AC components, and apply the resulting filter to data via the multiplication operator.

Linear filter objects in the **Natter** are '`numpy.ndarray`' matrices that are encapsulated as member variable `W` in a '`LinearTransform`' object. The main reason why the matrix `W` is stored in an object is to provide convenience methods on the filters and to keep track of the log-determinant of the Jacobian (see Section 6). Like '`Data`' objects, '`LinearTransform`'s support slicing to access rows and columns of `W`. The multiplication operator for '`LinearTransform`'s is overloaded such that they can be multiplied with other '`LinearTransform`', '`NonlinearTransform`', and '`Data`' objects. Depending on the participating objects, multiplication either simply multiplies the arrays or concatenates the nonlinear function with the results form previous multiplications.

The last preprocessing step in our example is whitening. Like the DC-AC separation, the whitening transform can be generated with the `LinearTransformFactory` which offers several whitening transforms. Here, we simply choose whitening based on principal component analysis (PCA). The **Natter** also offers symmetric whitening. Other whitening transforms can easily be added by implementing a new factory method in `LinearTransformFactory`.

Apart from the examples above, the `LinearTransformFactory` module can generate many other frequently used linear transforms. It also provides an interface to the **MDP** library (see Section 7.3, Zito, Wilbert, Wiskott, and Berkes 2008). Additionally, a custom linear filter can simply be created from a '`numpy.ndarray`' by passing it to the constructor of '`LinearTransform`'.

4. Models

All statistical models of natural images included in the **Natter** can be stated as parametric distributions. This section presents the core of the **Natter**, the `Distributions` module which contains the parametric models.

As mentioned in the introduction and as can be seen in Figure 1, the **Natter** uses the modular structure of L_p-spherically symmetric and L_p-nested distributions to cover a wide range of common parametric distributions for natural images. Here, we demonstrate at a model comparison for 8×8 image patches, how distribution objects can be created and estimated, and how their quality can be assessed. We will also explain other features of data objects and filters as we go along with the example.

We first state the whole example and then go through its elements step by step. We do not comment on the data loading and preprocessing commands as they have been discussed above. We also omit standard commands for plotting etc. in order to avoid clutter. The full code can be found in the example directory.

```
>>> from natter.DataModule import DataLoader
>>> from natter.Transforms import LinearTransformFactory
>>> from natter.Distributions import \
...   ProductOfExponentialPowerDistributions, LpSphericallySymmetric, \
...   CompleteLinearModel, LpGeneralizedNormal
>>> from collections import OrderedDict, defaultdict
>>>
>>> FILENAME_TRAIN = 'hateren4x4_train_No1.dat.gz'
>>> FILENAME_TEST = 'hateren4x4_test_No1.dat.gz'
>>>
>>> dat_train = DataLoader.load(FILENAME_TRAIN)
>>> dat_test = DataLoader.load(FILENAME_TEST)
>>>
>>> mu_train = dat_train.center()
>>> dat_test.center(mu_train)
>>>
>>> s = dat_train.makeWhiteningVolumeConserving()
>>> dat_test.makeWhiteningVolumeConserving(D = s)
>>>
>>> FDCAC = LinearTransformFactory.DCAC(dat_train)
>>> FDC = FDCAC[0, :]
>>> FAC = FDCAC[1:, :]
>>> FwPCA = LinearTransformFactory.wPCA(FAC * dat_train)
>>>
>>> dat_train = FwPCA * FAC * dat_train
>>> dat_test = FwPCA * FAC * dat_test
>>>
>>> n = dat_train.dim()
>>>
>>> q = LpSphericallySymmetric(n = n, p = 1.3)
>>> q.primary.remove('p')
>>>
>>> models = OrderedDict([
...   ('factorial exponential power',
...   ProductOfExponentialPowerDistributions(n = n)),
...   (r'$p$-generalized Normal', LpGeneralizedNormal(n = n)),
...   (r'$L_p$-spherical', LpSphericallySymmetric(n = n)),
...   ('complete linear model', CompleteLinearModel(n = n, q = q,
...   W = LinearTransformFactory.fastICA(dat_train)))])
>>>
>>> avg_log_loss = defaultdict(list)
```

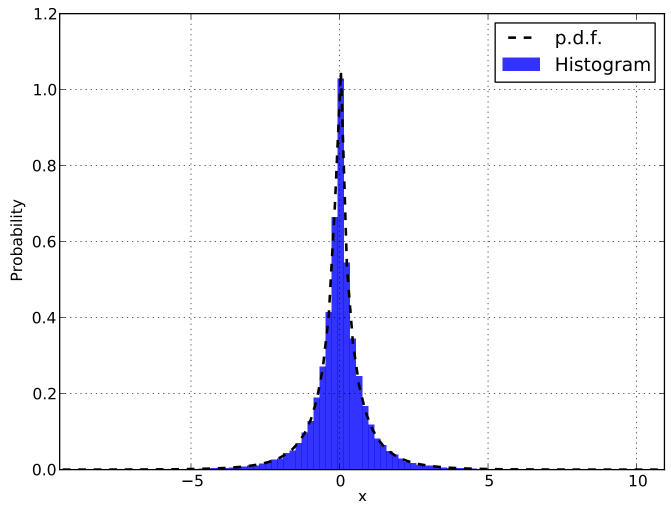

Figure 3: Univariate marginal of a power exponential distribution fitted to DC-zero filter responses of natural image patches.

```
>>>
>>> for model, p in models.items():
...   p.estimate(dat_train[:, :5000])
...   for xtest in dat_test.bootstrap(50, dat_test.numex()):
...     avg_log_loss[model].append(p.all(xtest))
>>> ...
>>> K = ['factorial exponential power', r'$p$-generalized Normal',
...   r'$L_p$-spherical', 'complete linear model' ]
>>> bp = ax.boxplot([avg_log_loss[k] for k in K], positions = arange(len(K)))
>>> ...
>>> for i, (model, p) in enumerate(models.items()):
...   ...
...   dat2 = p.sample(300000)
...   dat_train[:2, :].plot(ax = ax, plottype = 'loghist', colors = 'b',
...     label = 'true data')
...   dat2[:2, :].plot(ax = ax, plottype = 'loghist', colors = 'r',
...     label = 'sampled data')
...   ...
... ...
>>> for i, model in enumerate([r'$p$-generalized Normal',
...   r'$L_p$-spherical']):
...   p = models[model]
...   ...
...   p['rp'].histogram(dat_test.norm(p['p']), bins = 100, ax = ax)
>>> ...
>>> F = FDC.stack(models['complete linear model']['W'] * FwPCA * FAC)
>>> F.plotFilters(ax = ax)
>>> ...
```

4.1. Creating the distribution objects

We use four common distributions for our comparison which we introduce briefly.

1. **Factorial power exponential distribution:** Single filter responses to whitening filters on natural images are well captured by exponential power distributions

$$\rho(y) = \frac{p}{2s^{\frac{1}{p}}\Gamma\left[\frac{1}{p}\right]} \exp\left(-\frac{|y|^p}{s}\right)$$

   (Bethge 2006; Srivastava, Lee, Simoncelli, and Zhu 2003, see also Figure 3). The '`ProductOfExponentialPowerDistributions`' object implements a factorial model for a set of filter responses, where filter response is assumed to be independent of the other.

$$\rho_Y(\mathbf{y}) = \prod_i \frac{p_i}{2s_i^{\frac{1}{p_i}}\Gamma\left(\frac{1}{p_i}\right)} \exp\left(-\frac{|y_i|^{p_i}}{s_i}\right).$$

2. ***p*-generalized Normal distribution:** The p-generalized normal distribution (Goodman and Kotz 1973)

$$\rho_Y(\mathbf{y}) = \prod_i \frac{p}{2s^{\frac{1}{p}}\Gamma\left(\frac{1}{p}\right)} \exp\left(-\frac{|y_i|^p}{s}\right)$$

   has power exponentially distributed marginals and is therefore tightly linked to the above distribution. The difference is that each marginal has to have the same value for the exponent p. It belongs to the class of L_p-spherically symmetric distributions and is therefore parametrized over a radial distribution on the L_p-norm in the **Natter** (see next distribution). We include it here to demonstrate how a distribution can be parametrized in various ways and how the radial distribution of an L_p-spherically symmetric distribution can be used to visually assess the goodness-of-fit of a model.

3. **L_p-spherically symmetric distribution:** The joint distribution of whitened natural images is star-shaped. Therefore, an ideal class of distributions are the spherically symmetric distributions (Gupta and Song 1997; Hyvärinen and Koester 2007; Sinz and Bethge 2009).

   Due to their special statistical structure, L_p-spherically symmetric distributions have a generic density of the form (Gupta and Song 1997)

$$\rho(\mathbf{y}) = \frac{p^{n-1}\Gamma\left(\frac{n}{p}\right))\varrho(\|\mathbf{y}\|_p)}{\|\mathbf{y}\|_p^{n-1}2^n\Gamma\left(\frac{1}{p}\right)^n}. \tag{1}$$

   The function $\|\mathbf{y}\|_p = \left(\sum_i |y_i|^p\right)^{1/p}$ is called L_p-norm. It determines the shape of the contour lines of the distribution. Note that for $\|\mathbf{y}\|_p$ to be a proper norm p is required to be at least one. However, in the context of L_p-spherically symmetric distributions $\|\mathbf{y}\|_p$ does not need to be a proper norm. In fact, since L_p-spherically symmetric distributions are a special case of ν-spherical distributions (Fernández, Osiewalski, and Steel 1995) it is sufficient that $\|\mathbf{y}\|_p$ is positively homogeneous of degree one. This means that p merely needs to be positive.

Therefore, the generic parameters of an L_p-spherically symmetric distribution are the parameter p for the L_p-norm and the radial distribution ϱ. This is exactly how the **Natter** implements them.

4. **Complete linear model:** In Section 3, we demonstrated how to whiten data with the **Natter**. A property of whitened data is that is stays white under orthogonal transforms. Therefore, we can extend the above models by adding an orthogonal transform to the parameters such that our model becomes $\rho_X(\mathbf{x}) = \rho(A\mathbf{x})$ where A can be decomposed into $A = QW$ where W is a whitening matrix and $Q \in SO(n)$. As before, we assume that the data has been whitened and rescaled such that whitening has determinant one. Therefore, our model becomes $\rho_X(\mathbf{x}) = \rho_Y(Q\mathbf{y})$ where $\mathbf{y} = W\mathbf{x}$ is the whitened data. ρ_Y is a base distribution that needs to be chosen. In the current example, we will use an L_p-spherically symmetric distribution. Q can be estimated by gradient ascent on the log-likelihood of the model. However, since $Q \in SO(n)$ the optimization is more involved. The **Natter** adapts a scheme by Manton (2002) where the gradient with respect to $\mathbb{R}^{n\times n}$ is projected onto the tangent space of $SO(n)$ and used to perform a line search with back-projections onto $SO(n)$.

   The advantage of the '`CompleteLinearModel`' is its generality. Depending on the base distribution `q` one obtains different models that have been proposed for natural image statistics. If `q` is a '`ProductOfExponentialPowerDistributions`', then one obtains a density model for ICA. If `q` is chosen such that $\rho(\mathbf{y}) = \prod_I \rho_I(\mathbf{y}_I)$ where I are index sets that form a partition of $1, \dots, n$ and the single ρ_I are L_p-spherically symmetric one obtains a density model for ISA (Hyvärinen and Koester 2007). Setting it to an L_p-spherically symmetric distribution leads to the model in Sinz and Bethge (2009). Using an L_p-nested symmetric distribution yields the model in Sinz *et al.* (2009b). Future distributions of that form can easily be integrated by implementing new base distributions. Since the derivative needed for the optimization can be decomposed via the chain rule, the base distribution only needs to provide the necessary derivative functions to be used in the '`CompleteLinearModel`'.

Now, let us look at the example in more detail. Since all distributions in the example are multivariate distributions, we need to specify the dimensionality when creating the distribution objects. We extract it from the data via `n = dat_train.dim()` beforehand.

The line `q = LpSphericallySymmetric(n = n, p = 1.3)` creates an L_p-spherically symmetric distribution in n dimensions and sets the p for the L_p-norm to $p = 1.3$.

If we printed `q` to the prompt, the output would be

```
------------------------------
Lp-Spherically Symmetric Distribution
        p: 2.0
        n: 63
        rp:
        ------------------------------
        Gamma Distribution
                s: 1.0
                u: 1.0
                Primary Parameters:[u, s]
        ------------------------------


        Primary Parameters:[rp, p]
------------------------------
```

It lists the name of the '`Distribution`' object and a set of parameters that define a particular distribution. All parameters are stored in a member dictionary called `param`. The keys are the parameter names, the values are the parameters. Note that the parameters can also be transforms or distributions like the radial distribution `rp`. In this case, the L_p-spherically distribution has a radial γ-distribution since this is the default option. However, one can specify any other density model on $\mathbb{R}^+$ as a radial distribution. Once it is implemented as a child of '`Distribution`' and provides the necessary methods it can serve as a radial distribution of an '`LpSphericallySymmetric`' object (see Section 5).

When a '`Distribution`' object is created parameters can be passed to the constructor. The standard way to do so is with named parameters as in the example before. The syntax is identical to function calls with named parameters in Python. Another option is to call the constructor with a dictionary that parameter-name:parameter-value key-value pairs. Therefore, a simple way to clone a '`Distribution`' object is to call the constructor with the `param` array of the original '`Distribution`' object. However, each distribution also has a `copy` member function that clones it.

Parameters can be obtained and set by treating the distribution like a dictionary. For example, one could obtain the dimensionality and the value of p from `q` via

```
>>> n = q['n']
>>> p = q['p']
```

It is possible to change any parameter this way although only some changes are sensible. For example, if the dimensionality of `models['factorial exponential power']` is set to 80 via `models['factorial exponential power']['n'] = 80`, the list which holds the marginal distribution would still be 63 dimensional. The '`Distribution`' objects leave the responsibility to keep the parameters consistent to the user.

Printing a distribution to the prompt also lists its primary parameters, which are the ones that can be fitted to data. In situations, where the optimization can be computationally expensive, it sometimes is a good idea to exclude certain parameters from the optimization. For example, it is known from previous work, that $p = 1.3$ is a good choice for L_p-spherically symmetric distributions fitted to filter responses of natural image patches. The **Natter** allows to exclude parameters from the optimization via the `primary` member list of '`Distribution`' objects. By default, this list contains all parameters that can be estimated with the distribution object. When deleting parameter names from that list, they will not be estimated. In the example, we exclude p from the estimation by simply deleting p from the array of primary parameters via `q.primary.remove('p')`.

Primary parameters are also helpful for a quick implementation of maximum likelihood estimation. The '`Distribution`' class, from which every '`Distribution`' object inherits, defines the abstract methods `primary2array`, `array2primary`, `primaryBounds`, `loglik`, and `dldtheta`.

When implemented, the former two functions convert the primary parameters of an object into a **NumPy** array and vice versa. The function `primaryBounds` provides a list of possible bounds on the parameters, for example, when a parameter is restricted to be positive. The function `dldtheta` must yield the derivative of the log-likelihood with respect to the primary parameters at the data points passed to `dldtheta`. If those five functions are implemented by a new '`Distribution`' object, the generic `estimate` method of '`Distribution`' tries to perform a gradient ascent on the log-likelihood. Since these functions are usually implemented quickly, the **Natter** infrastructure minimizes coding effort which makes it a useful tool for exploring new types of distributions.

Note that an interval restriction on the parameters does not handle all possible constraints: For example, for matrices there are often no natural bounds in terms of an interval. Therefore, the default `estimate` method of a '`Distribution`' object with raise a warning to point to that fact.

The remaining code for generating the distributions creates a dictionary with the other distributions with the dimensionality set to n and default parameters otherwise. As mentioned above, we use the L_p-spherically symmetric distribution `q` as base distribution for the complete linear model. The orthogonal matrix from the complete linear model is initialized from a complete set of independent component analysis (ICA) filters which is a '`LinearTransform`' object generated with the `LinearTransformFactory` module via `W = LinearTransformFactory.fastICA(dat_train)`. The function `fastICA` interfaces to the **MDP** library for obtaining the filters.

4.2. Fitting and testing the distributions

Once the '`Distribution`' object is setup, it is very simple to fit it to data: In our example we loop through all distributions from the `models` dictionary and fit them to the first 5000 examples in the data object `dat_train`. Note that the data object supports slicing just like a normal '`numpy.ndarray`'.

```
for model, p in models.items():
  p.estimate(dat_train[:, :5000])
  for xtest in dat_test.bootstrap(50, dat_test.numex()):
    avg_log_loss[model].append(p.all(xtest))
```

By default, each object that inherits from '`Distribution`' has an `estimate` method which is called with a '`Data`' object as first argument. Other parameters can be specified for particular distributions in order to control the estimation behavior. As mentioned before, the `estimate` method tries to use the `primary2array`, `array2primary`, `dldtheta`, `primaryBounds` functions to perform a gradient ascent on the log-likelihood. This default option is very convenient for a fast implementation of new distributions. However, in certain cases, a simple gradient ascent cannot be performed, or there are faster ways to estimate the distribution. For those cases, the children of '`Distribution`' have their own `estimate` methods.

For the complete linear model, the p of the base distribution will not be estimated since we excluded it from the primary parameters. If that was not the case, the `estimate` method would also fit p. Of course, the estimation method is computationally more expensive in this case.

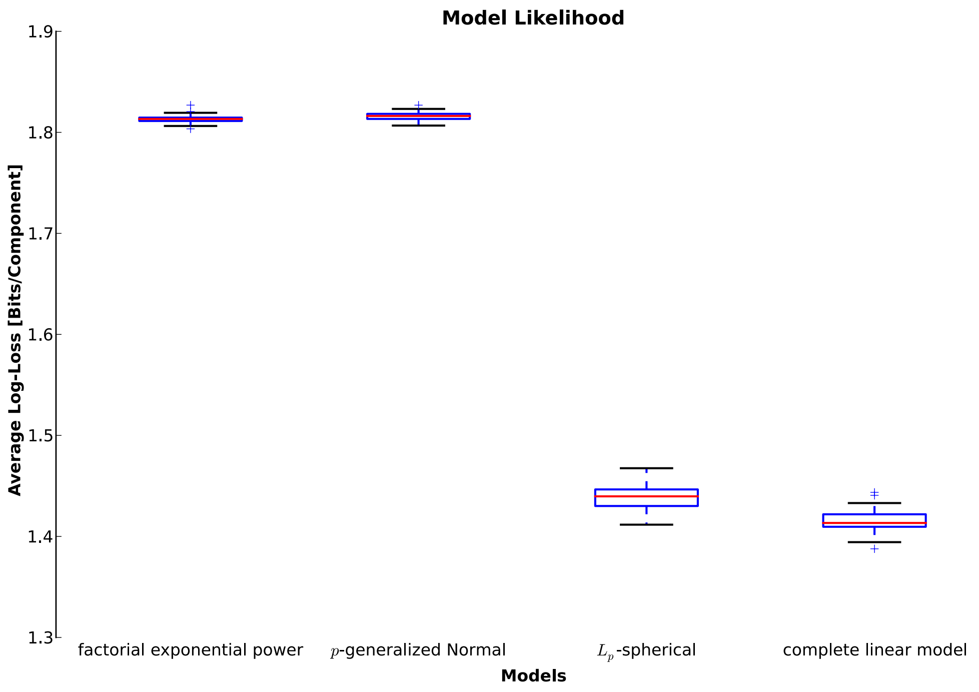

Figure 4: Average log-loss of the different models in the example. Not surprisingly, the factorial exponential power distribution and the p-generalized Normal perform similarly. The L_p-spherically symmetric model and the complete linear model perform better by almost half a bit per dimension.

After the model has been estimated from the data, we compute an estimate of the average log loss (ALL), which is the negative expected log likelihood

$$\mathrm{ALL}[p] = \langle -\log p\left(\mathbf{x}\right)\rangle \approx \frac{1}{m}\sum_{i=1}^{m} -\log p\left(\mathbf{x}_i\right).$$

The **Natter** additionally normalizes the ALL by the dimensionality of the data and converts the results to bits.

It is easy to show that the ALL is the entropy of the true distribution plus the Kullback-Leibler divergence between the true distribution and the model. Since the latter is always positive, the ALL is a true loss function that allows us to compare different models with each other on an absolute scale.

In order to provide errorbars on the ALL we use the bootstrap iterator implemented in the data object. Here, it samples 50 datasets from `dat_test` with the same number of data points, computes the ALL for each test set, and stores the ALL in the `avg_log_loss` dictionary.

The next part of the example puts these values into a box plot (Figure 4). Not surprisingly, the factorial exponential power distribution and the p-generalized Normal perform similarly. The L_p-spherically symmetric model and the complete linear model perform better by almost half a bit per dimension. The next parts of the example now demonstrate ways to find reasons for the difference in performance.

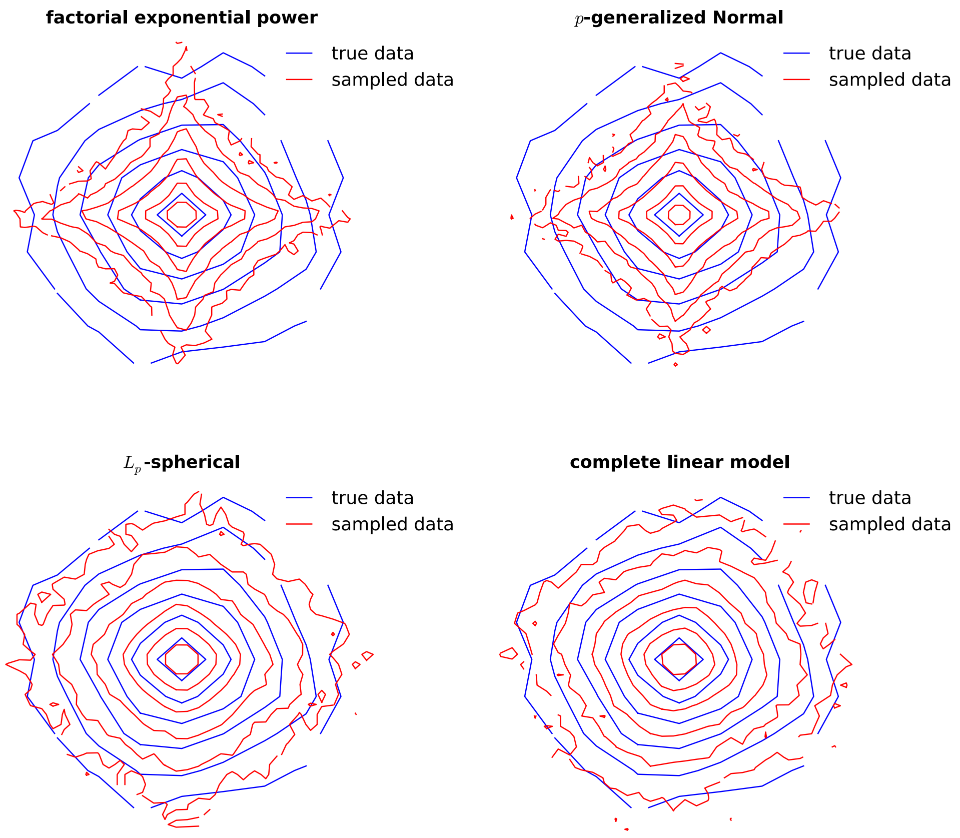

Figure 5: Contour plots of the log-histogram for samples from the various models in the example and the training data. Not surprisingly, the contours of the factorial exponential power distribution and the p-generalized Normal are almost identical. Due to the independence assumption on the marginals, the resulting iso-density contours are too star-shaped (Simoncelli 1997; Bethge 2006; Eichhorn *et al.* 2009). The L_p-spherically symmetric distribution and the complete linear model capture the contour shape much better (Sinz and Bethge 2009).

4.3. Plotting two-dimensional log-contours

As a visual assessment of the goodness-of-fit, one can look at two-dimensional log-contour plots of the original data and samples from the fitted distributions. In our example, the relevant code to do this in the **Natter** is

```
>>> for i, (model, p) in enumerate(models.items()):
...     ...
...     dat2 = p.sample(300000)
...     dat_train[:2, :].plot(ax = ax, plottype = 'loghist', colors = 'b',
...       label = 'true data')
...     dat2[:2, :].plot(ax = ax, plottype = 'loghist', colors = 'r',
...       label = 'sampled data')
...     ...
```

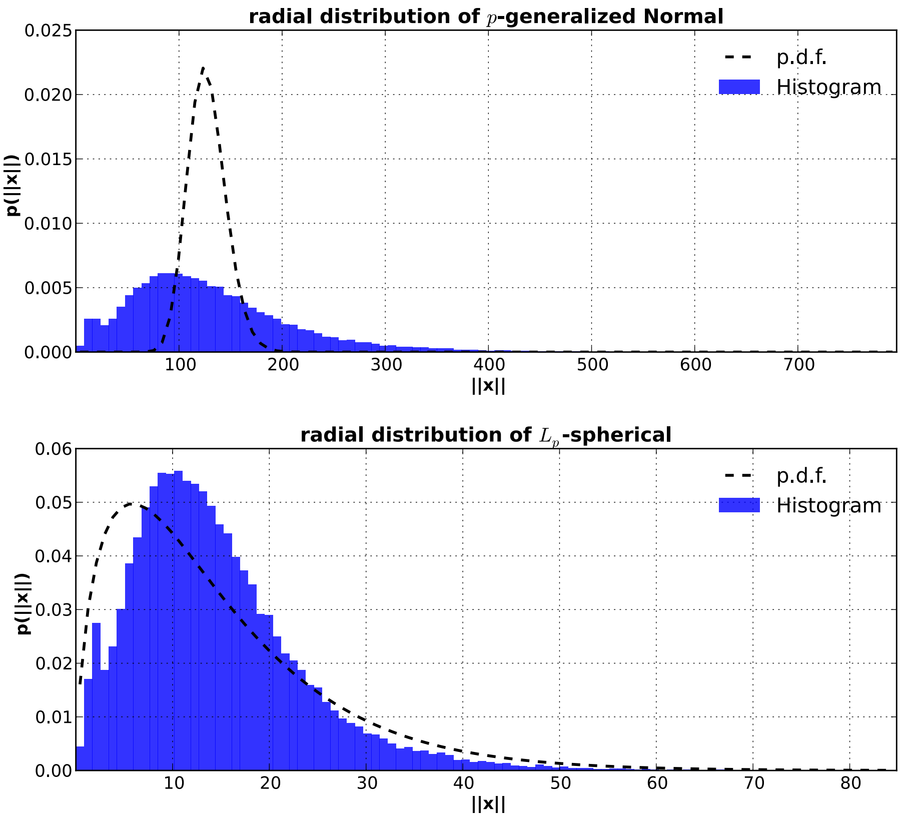

Figure 6: Radial distributions of a p-generalized Normal (top) and a L_p-spherically symmetric model with a radial γ-distribution. Note that the p for the L_p-norm in both cases is different. The radial distribution of the L_p-generalized Normal (a generalized χ-distribution) fits less well than the γ-distribution.

The '`Data`' object provides a plotting methods that either generates a scatter plot or a contour plot of the log-histogram. In the example, we first sample 300k samples from the fitted distribution and then use these data points and the training data to generate contour plots of the log-histogram for the first two data dimensions.

4.4. Plotting radial distributions

Apart from the contour lines, an important feature of L_p-spherically symmetric models is the radial distribution. In our example above, we visually assess the goodness-of-fit via a histogram. To this end, we extract the radial distribution via `p['rp']` as mentioned above and feed the L_p-norms of the data points with the estimated p to its histogram function.

```
>>> for i, model in enumerate([r'$p$-generalized Normal',
...    r'$L_p$-spherical']):
...    p = models[model]
...    ...
...    p['rp'].histogram(dat_test.norm(p['p']), bins = 100, ax = ax)
...    ...
```

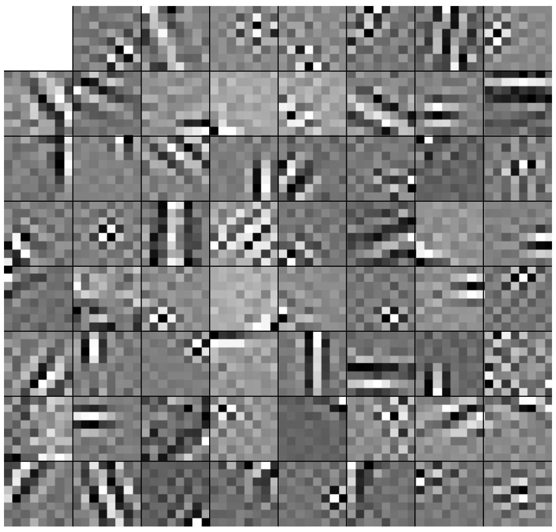

Figure 7: Combined DC, AC, whitening, and complete linear model filters. Since the filter values are normalized for better contrast, the constant DC filter appears as a white patch. The filters are similar to the Gabor like filter found with ICA (Sinz and Bethge 2009).

Figure 6 shows the resulting plots. We can see that the radial distribution of the p-generalized Normal (a generalized χ-distribution) fits much less well than the radial γ-distribution of the L_p-spherically symmetric model. The L_p-spherically symmetric model could probably be improved further by choosing another radial distribution from the `Distributions` module of the **Natter**. Usually, a mixture distribution yields good fits. The **Natter** contains a generic mixture distribution object which can be used for that purpose. In that case, the L_p-spherically symmetric model would be generated via

```
>>> rp = FiniteMixtureDistribution(P = [Gamma(u = random.rand() * 3.,
...    s = random.rand() * 3.) for _ in xrange(5)])
>>> p = LpSphericallySymmetric(n = n, rp = rp)
```

4.5. Plotting filters

Finally, a common visual inspection method in natural image modeling is to look at the resulting filters. One parameter of the complete linear model is the additional rotation matrix W. Since the data has been projected on the AC components and whitened before, we need to combine that filter matrix with the AC filters and the whitening filters and we need to combine all these matrices to plot the net filters. The relevant code of the example is

```
>>> F = FDC.stack(models['complete linear model']['W'] * FwPCA * FAC)
>>> F.plotFilters(ax = ax)
```

Here, we extract the orthogonal matrix W from the complete linear model via `models['complete linear model']['W']`, multiply it with the other filters via the overloaded multiplication operator, and finally stack the result with the DC filter.

'`LinearTransform`' objects provide a plotting function for filter that takes care of reshaping the vectors into patches and normalizing the patches for better contrast. The resulting filters can be seen in Figure 7.

5. Implementation of new models

In the beginning, we emphasized that the modular structure of the distributions implemented in the **Natter** allows the user to quickly explore many common distributions for natural images. In this section, we describe the basic structure of a '`Distribution`' object and demonstrate how they can be implemented.

To this end, assume that we want to implement a uniform distribution within an L_p-unit ball. The corresponding radial distribution is a β-distribution

$$\varrho(r) = \frac{r^{\alpha-1}(1-r)^{\beta-1}}{B(\alpha,\beta)}$$

with parameters $\alpha = n$ and $\beta = 1$ (Sinz and Bethge 2010). Since the **Natter** does not contain a β-distribution so far, we need to implement it in order to get the uniform distribution via the '`LpSphericallySymmetric`' distribution. We first state the class, discuss the single components, and then demonstrate how to integrate it into the L_p-spherically symmetric distribution below.

```
from natter.Auxiliary.Utils import parseParameters
from scipy import stats
from scipy.special import digamma, beta
from natter.Distributions import Distribution
from natter.DataModule import Data
from numpy import *

class Beta(Distribution):

  def __init__(self, *args, **kwargs):
    param = parseParameters(args, kwargs)

    self.name = 'Beta Distribution'
    self.param = {'alpha':1.0, 'beta':1.0}

    if param is not None:
      for k in param.keys():
        self.param[k] = float(param[k])
      self.primary = ['alpha', 'beta']

  def pdf(self, dat):
    return squeeze(stats.beta.pdf(dat.X, self['alpha'], self['beta'] ))

  def loglik(self,dat):
    return log(self.pdf(dat))
```

```
def sample(self,m):
  return Data(stats.beta.rvs(self['alpha'], self['beta'], size = (m, )))

def primary2array(self):
  ret = zeros(len(self.primary))
  for ind,key in enumerate(self.primary):
    ret[ind] = self.param[key]
  return ret

 def array2primary(self, arr):
   ind = 0
   for ind, key in enumerate(self.primary):
     self.param[key] = arr[ind]
   return self

 def primaryBounds(self):
   return len(self.primary) * [(1e-6, None)]

 def dldtheta(self, dat):
   ret = zeros((len(self.primary), dat.numex()))
   x = dat.X[0]
   a = self['alpha']
   b = self['beta']
   p = self.pdf(dat)
   for ind, key in enumerate(self.primary):
     if key is 'alpha':
       ret[ind, :] = p * (digamma(a + b) - digamma(a) + log(x))
     elif key is 'beta':
       ret[ind, :] = p * (digamma(a + b) - digamma(b) + log(1 - x))
   return ret
```

5.1. Class definition and constructor

Since, the β-distribution should be a child of 'Distribution' it inherits from it. We already mentioned that each 'Distribution' object holds a member dictionary called param which holds the necessary parameters. This dictionary is built up in the constructor via that function parseParameters. Apart from that, the object gets a name and default values for the parameters. Parameters passed to the distribution by the user replace the default parameters in the following for loop. Finally, we declare both parameters of the β-distribution to be primary parameters via self.primary = ['alpha', 'beta'].

5.2. Member functions

By inheriting from 'Distribution', each distribution provides a number of functions, like pdf, cdf, sample, loglik, ... Initially, these function are not implemented. When called, they raise an exception which notifies the user that this member function has not been implemented yet. Depending on how a particular distribution is supposed to be used, many of these functions are not needed. In the current case, we would like to sample from the distribution, compute (log)-likelihoods, and estimate its parameters from data.

In order to provide the first two functionalities, we implement the functions `pdf`, `loglik`, and `sample`, where we basically borrow the functionality from the `scipy.stats` module (Jones, Oliphant, and Peterson 2001). By convention, the `sample` method returns a '`Data`' object which always holds a two-dimensional '`numpy.ndarray`'. The `loglik`, `pdf`, and `cdf` functions always return one-dimensional arrays.

In order to provide the necessary methods for fitting the distribution, we need to implement the functions `primary2array`, `array2primary`, `primaryBounds`, and `dldtheta`. The first, converts the primary parameters into a '`numpy.ndarray`', while the second reverses that operation. The third provides bounds for the primary parameters as a list of tuples. In our case, the parameters need to be positive. Therefore, we return `(1e-6, None)` as bounds, where `None` indicates that there is no upper bound and where we use `1e-6` instead of `0` for numerical stability. Finally, the function `dldtheta` returns the derivatives of the log-likelihood with respect to the primary parameters at each data point in `dat`. The return value is an '`numpy.ndarray`' with as many rows as primary parameters and as many columns as data points. These four functions will be used by the `estimate` method inherited from the '`Distribution`' class, which will attempt a gradient ascent on the log-likelihood with respect to the primary parameters.

5.3. Using the β-distribution as radial distribution

We now demonstrate how to use the above '`Beta`' distribution object as a radial distribution in the '`LpSphericallySymmetric`' object. Omitting the code for creating figures, axes, and showing the plot, the relevant code is:

```
>>> p_true = LpSphericallySymmetric(n = 2, rp = Beta(alpha = 2., beta = 1.),
...   p = .5)
>>>
>>> p_est = LpSphericallySymmetric(n = 2, rp = Beta(alpha = 2, beta = 2),
...   p = .5)
>>> p_est.primary.remove('p')
>>> p_unest = p_est.copy()
>>>
>>> dat = p_true.sample(5000)
>>> p_est.estimate(dat)
>>>
>>> dat_false = p_unest.sample(5000)
>>> dat_est = p_est.sample(5000)
>>> ...
>>> dat.plot(ax = ax)
>>> ax.axis([-1, 1, -1, 1])
>>> ...
>>> dat_false.plot(ax = ax)
>>> ...
>>> dat_est.plot(ax = ax)
```

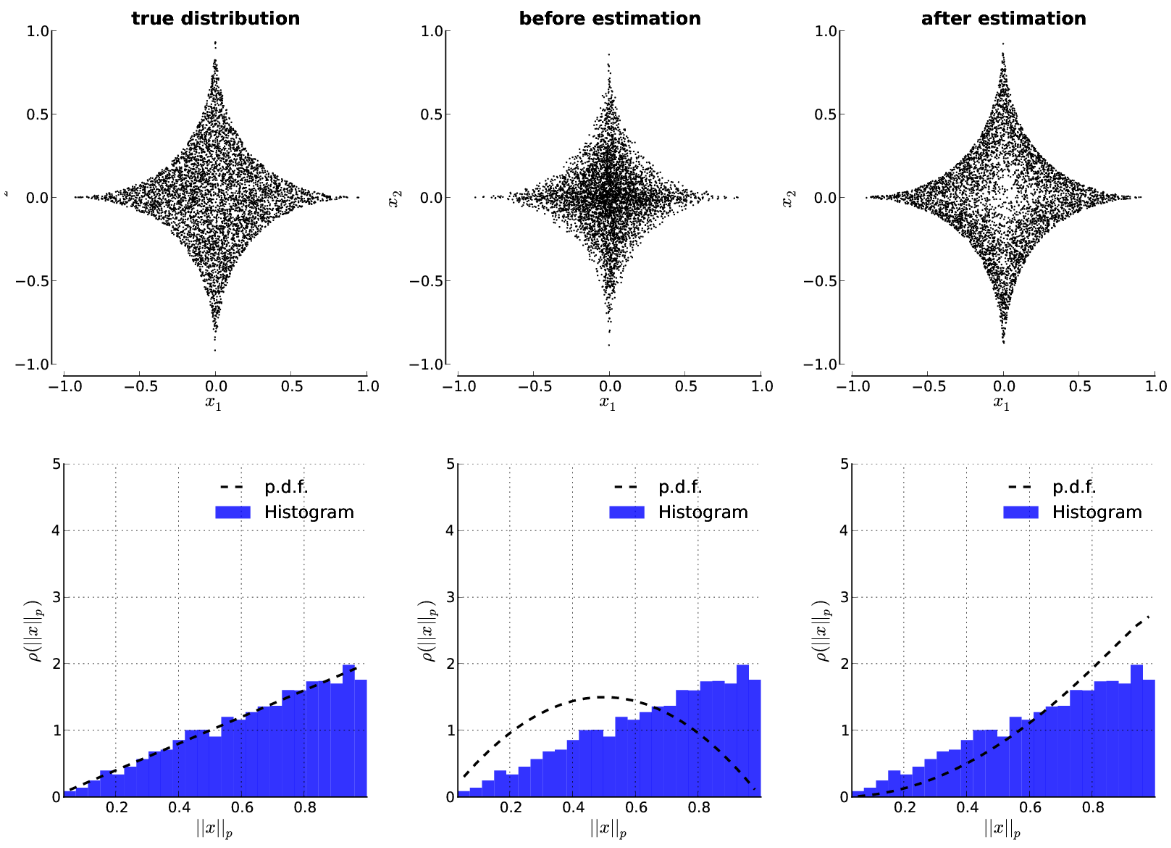

Figure 8: Uniform distribution within the $L_{\frac{1}{2}}$-unit ball. The radial distribution is a $\beta(n, 1)$ distribution (Sinz and Bethge 2010). The top row shows samples from the joint distribution for the true uniform distribution, an $L_{\frac{1}{2}}$-spherically symmetric distribution with a different β-distribution as radial distribution, and the joint distribution after optimizing the parameters of the former distribution on data from the uniform distribution. The bottom row shows the histograms of the true data with the respective radial distributions.

```
>>> ...
>>> p_true['rp'].histogram(dat.norm(p = p_true['p']), ax = ax)
>>> ...
>>> p_unest['rp'].histogram(dat.norm(p = p_unest['p']), ax = ax)
>>> ...
>>> p_est['rp'].histogram(dat.norm(p = p_est['p']), ax = ax)
>>> ...
```

As you can see, we generate a uniform distribution within the $L_{\frac{1}{2}}$-unit ball by initializing an '`LpSphericallySymmetric`' object with the proper '`Beta`' object as radial distribution. We set `p = .5` to determine the shape of the sphere, and exclude p from estimation by removing it from the primary parameters. We also create another L_p-spherically symmetric object with a different radial β-distribution. We then sample data from the true model, and fit the second L_p-spherically symmetric distribution to it. Its `estimate` method internally calls the `estimate` method of '`Beta`' which calls the inherited `estimate` method of the '`Distribution`' object which uses the methods we implemented above to fit find the right parameters. The resulting distributions can be seen in Figure 8. As one can see, the fit is sufficiently close but not perfect. In the case of the β-distribution this is caused by a shallow likelihood function for particular combinations of parameters α and β. A dedicated `estimate` method might yield better results in this case. This demonstrates that a simple likelihood optimization quickly yields acceptable, and in many cases sufficient, results which might be improved in their accuracy and computational efficiency by replacing them with more specialized fitting routines. The **Natter** offers both possibilities.

6. Nonlinear transforms and log-determinants

A factorial model does not provide a very good fit to whitening filter responses of natural image patches (see Figures 4 and 5). This means that the filter responses still contain strong higher order dependencies. For some applications, however, it might be desirable to have independent features of an image. Recent research in natural image statistics, have produced a nonlinear transform, called radial factorization or radial Gaussianization (Sinz and Bethge 2009; Lyu and Simoncelli 2009), which strongly reduces the statistical dependencies between the single marginals. This transform can be thought of as a kind of nonlinear ICA which first fits an arbitrary L_p-spherically symmetric distribution to the joint filter responses and subsequently rescales the radii in the L_p-norm with a nonlinear function such that the new radial distribution corresponds to a joint factorial model. It is based on the fact that for L_p-spherically symmetric distributions and a fixed value of p, there is a single type of radial distribution (up to scale) that has independent marginals (Sinz, Gerwinn, and Bethge 2009a). For $p = 2$ this distribution is isotropic multivariate Normal. For an arbitrary value of p, the distribution is the factorial distribution with generalized Normal marginals which was already discussed above.

The **Natter** implements the radial factorization as '`NonlinearTransform`'. As for the '`LinearTransform`', the multiplication operator is overloaded. '`NonlinearTransform`' objects can be multiplied with other '`NonlinearTransform`' or '`LinearTransform`' objects. In this case, multiplication simply means concatenation. When multiplied with a '`Data`' object, the '`NonlinearTransform`' is applied to the data.

Apart from radial factorization, there are several other nonlinear transforms that can be generated by the `NonlinearTransformFactory`. As an example, we now demonstrate how to visualize a two-dimensional copula for whitening responses before radial factorization and afterwards. If the responses are independent, the copula should look uniform (see Figure 9). This can be seen as a visual assessment of how well a given distribution fits the data.

```
>>> from natter.Transforms import NonlinearTransformFactory
>>> from natter.Distributions import ISA, ExponentialPower, ISA, Uniform
>>> from natter.Logging.LogTokens import Table
>>>
>>> print "Loading a simple Data module from an ascii file"
>>> FILENAME_TRAIN = 'hateren8x8_train_No1.dat.gz'
>>>
```

```
>>> dat = DataLoader.load(FILENAME_TRAIN)
>>>
>>> dat.center()
>>> dat.makeWhiteningVolumeConserving()
>>>
>>> FDCAC = LinearTransformFactory.DCAC(dat)
>>> FDC = FDCAC[0, :]
>>> FAC = FDCAC[1:, :]
>>> FwPCA = LinearTransformFactory.wPCA(FAC * dat)
>>> dat = FwPCA * FAC * dat
>>>
>>> n = dat.dim()
>>>
>>> p = LpSphericallySymmetric(n = n)
>>> pT = ISA(P = [Uniform() for _ in xrange(n)], n = n,
...   S = [(i, ) for i in xrange(63)])
>>> pISA_before = ISA(P = [Histogram() for _ in xrange(n)], n = n,
...   S = [(i, ) for i in xrange(63)])
>>> pISA_after = ISA(P = [Histogram() for _ in xrange(n)], n = n,
...   S = [(i, ) for i in xrange(63)])
>>>
>>> p.estimate(dat)
>>> pISA_before.estimate(dat, bins = 5000)
>>>
>>> FHE = NonlinearTransformFactory.MarginalHistogramEqualization(pISA_before,
...   pT)
>>> dat_no_rf = FHE * dat
>>>
>>> FRF = NonlinearTransformFactory.RadialFactorization(p)
>>> dat2 = FRF * dat
>
>>> pISA_after.estimate(dat2, bins = 5000)
>>> FHE = NonlinearTransformFactory.MarginalHistogramEqualization(pISA_after,
...   pT)
>>> dat_rf = FHE * dat2
>>> ...
>>> dat_no_rf[:2, :].plot(ax = ax, color = 'k')
>>> ...
>
>>> dat_rf[:2, :].plot(ax = ax, color = 'k')
>>> ...
>>> T = Table(['before rad. fac.', 'after rad. fac.'],
...   ['untransf. data', 'transf. data with log-det'])
>>> T['after rad. fac.', 'untransf. data'] = pISA_after.all(dat2)
>>> T['after rad. fac.', 'transf. data with log-det'] = pT.all(dat_rf) \
...   - mean(FHE.logDetJacobian(dat2)) / log(2) / dat2.dim()
>>> T['before rad. fac.', 'transf. data with log-det'] = pT.all(dat_rf) \
...    - mean((FHE * FRF).logDetJacobian(dat)) / log(2) / dat.dim()
>>> T['before rad. fac.', 'untransf. data'] = p.all(dat)
>>> print T
>>> show()
```

After the data has been loaded, we generate an L_p-spherically symmetric model as before and fit it to the filter responses. Additionally, we create an independent subspace (`ISA`) model with one-dimensional histogram distributions on the marginals. This model is equivalent to a density model of ICA on natural images.

The single histogram distributions will fit the marginals well, but not the joint distribution due to dependencies between the filter responses (see Figure 5). If radial factorization makes the marginals more independent, the ICA (`ISA`) model should fit better which will be reflected in a more uniform copula.

After the L_p-spherically symmetric model has been estimated, we generate a radial factorization object from that model via `NonlinearTransformFactory.RadialFactorization(p)`. The object `FRF` is a '`NonlinearTransform`' and can be multiplied with '`Data`' and '`LinearTransform`' objects.

In addition to the radial factorization object, we create two marginal histogram equalization '`NonlinearTransform`' objects that map random variables from the ISA models into uniformly distributed random variables.

We plot the first two marginals of the resulting data objects. Although both data sets will have uniform marginals, the one without radial factorization shows more statistical dependencies than the one with (see Figure 9).

'`NonlinearTransform`' objects represent functions that are applied to each data point in a dataset. The overloaded multiplication operator makes it very convenient to apply nonlinear functions to data and outputs of linear filters. Most importantly, however, it also tracks the determinant of the Jacobian of a concatenated transformation. These log-determinants are frequently needed for likelihood or entropy computations. For instance, when fitting a probability density to transformed data one needs to correct with the average log-determinant of the Jacobian in order to compute the likelihood of the untransformed data with the estimated model. The only information, the user needs to specify is the log-determinant of the Jacobian of each single '`NonlinearTransform`'.

In the final part of the example, we compute likelihoods for the data in two different ways: Either by using the original probability density or by using a probability density on the transformed data and accounting for the transformation with the average log-determinant of the Jacobian. Note that the two values should approximately be equal. For storing the values we use a `Table` from the `Logging` module (see Section 7.1). The above code yields the following output.

```
+------------------+-----------------+-------------------------+
|                  | untransf. data  | transf. data with log-det |
+------------------+-----------------+-------------------------+
| before rad. fac. | 1.446           | 1.443                     |
+------------------+-----------------+-------------------------+
| after rad. fac.  | 2.026           | 2.026                     |
+------------------+-----------------+-------------------------+
```

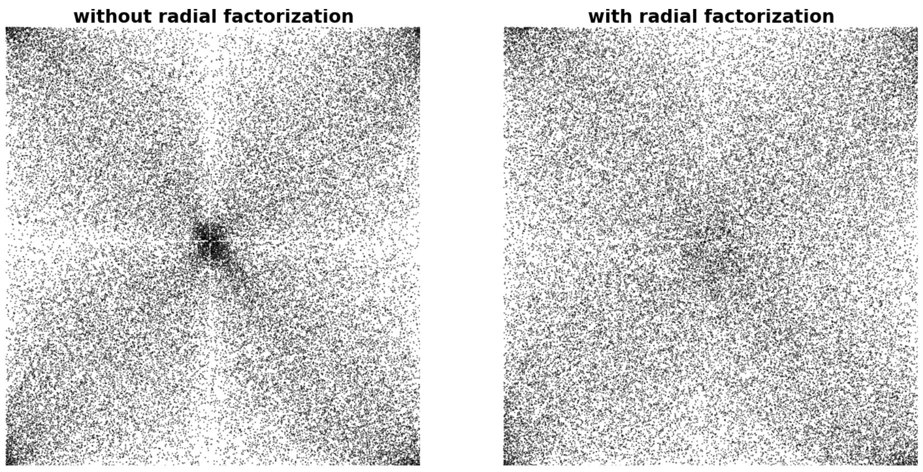

Figure 9: Two marginal distributions of whitening filter responses that have been transformed into a uniform distribution before (left) and after (right) radial factorization was applied. Even though, the responses are not completely independent, one can easily see that radial factorization strongly reduces the dependencies.

7. Additional features

The **Natter** implements a number of additional features that are useful in natural image numerical experiments. These include logging of experiments and results, non-parametric entropy estimation methods, and a wrapper for the modular toolkit for data processing toolkit **MDP**.

7.1. Logging

For displaying the table above we used the `Table` '`LogToken`' from the `Logging` module. In general, each **Natter** object is able to print a representation of itself to the Python prompt. However, for documenting the results of a specific numerical experiment, this output might not be the best option. For this purpose, the **Natter** provides the `Logging` module. This module has a member called '`ExperimentLog`'. Objects of that type can store information and results from a numerical experiment and save it to different formats. Information is simply added by adding strings with information to it. Hyperlinks can be added, in case the '`ExperimentLog`' is saved as HTML file.

```
>>> from natter.Logging import ExperimentLog
>>> p = ExperimentLog('My fancy experiment')
>>> p += 'We sampled of data we found on the website:'
>>> p *= ('http://dataparadise.com','data paradise')
>>> p.write('results.html', format = 'html')
```

Most importantly, all objects that inherit from '`LogToken`' can be added to '`ExperimentLog`'. By inheriting from '`LogToken`'s, the object has to provide the method `html` and `ascii`. When implemented, these methods must return a representation of themselves in the respective format. While the `LogTokens` module additionally provides certain objects like the `Table` used in the example above, all ‘`Distribution`’ objects are also `LogTokens` (see Figure 2). This means that they, too, can be simply added to the ‘`ExperimentLog`’ object.

7.2. Entropy estimation

Many numerical experiments with natural image statistics involve the estimation of Shannon entropy. The estimation of the joint entropy for multivariate sources is a challenging estimation problem (Paninski 2003) and existing estimators can be rather involved (e.g., Hosseini *et al.* 2010; Chandler and Field 2007). Therefore, the **Natter** only implements estimators for univariate marginal entropies in the `natter.Auxiliary.Entropy` module (with one exception; see below). Estimation of marginal entropies involves choosing bin sizes, regularization, and conversion from discrete to differential entropy (Paninski 2003; Eichhorn *et al.* 2009). The **Natter** implements different regularization methods which are described in Paninski (2003). When the bin size is not specified, it uses the heuristic of Scott (1979) to estimate it properly.

While non-parametric joint entropy estimation is hard, the **Natter** provides a more robust possibility to estimate the entropy with parametric models by fitting a model from the **Natter** to a training set and then compute the average log-loss over a test set. It can be shown that the average log-loss equals the entropy of the data plus the Kullback-Leibler divergence between the true and the parametric model distribution. Therefore, the average log-loss is a robust and conservative estimate of the true joint entropy.

Using the structure of L_p-spherically symmetric distributions, the **Natter** additionally provides a semi-parametric Monte Carlo estimator for the joint entropy in the function `natter.Auxiliary.Entropy.LpEntropy`. It estimates the p of the L_p-norm from the ratio distribution of the marginals, which is independent of the radial distribution (Szabłowski 1998), and uses the special form of the density of L_p-spherically symmetric distributions (see Equation 1) for a Monte Carlo estimator of the joint entropy.

7.3. Integration of the modular toolkit for data processing toolbox MDP

For comparability of results, it is often desirable to reuse exactly the same preprocessing chain down to the implementation details. Therefore, it makes sense to provide a simple integration of standard toolkits to avoid the need of reimplementing methods or transforming data from one structure to another. To simplify the (pre-)processing of data, the **Natter** provides an intuitive wrapper for the linear **MDP** nodes. The following example shows how to use the wrapper to apply FastICA (Hyvärinen and Oja 1997) to the whitened data:

```
>>> from natter.Transforms import LinearTransformFactory
>>> U = LinearTransformFactory.mdpWrapper(dat, 'FastICA', output = 'filters',
...    verbose = True, whitened = True)
>>> dat_ica = U * dat
```

The `mdpWrapper` passes all named parameters but `output` on to the **MDP** node. The parameter `output` defines which member of the node will be read out and returned as ‘`LinearTransform`’ object.

7.4. Test generators for distributions

Software verification is a demanding task, especially for probabilistic methods. The **Natter** comes with a test suite that (i) tests already defined modules and models and (ii) provides a simple mechanism to test these basic properties for newly implemented probabilistic models. These tests employ the standardized **Natter** framework to assess whether the functions provided by newly implemented distributions are correct or at least consistent with each other. New user-defined distributions, can easily be included in this generic test suite by adding them into the dictionary in the file `config_dictionaries.py` inside the **Natter**'s test directory. A full syntax example is provided within the file.

The generic test suite checks the following items.

Consistency between `loglik` and `sample` method. This test checks whether the defined likelihood is consistent with the sampling method via importance sampling. First, samples are generated from a distribution, which is known to be consistent in its definition of sampling and likelihood methods. Then the partition function of the distribution under test is estimated via importance sampling. If the defined likelihood function is properly normalized, the partition function and hence the estimate thereof should be 1. Similarly, the partition function of the known distribution is estimated via importance sampling but this time by using samples generated from the distribution which is tested. Note that in both directions the likelihood function is needed for calculating the importance weights. Hence this test is not a verification in a strict sense, but nevertheless indicates the consistency of the defined methods. Also note that the estimate of the partition function varies between tests due to its statistical nature. Therefore, this test might fail occasionally if the required tolerance of the estimate is not met. The tolerance level can be adjusted for each model in the file `config_dictionaries.py` in the **Natter**'s test directory.

Gradient checks. The gradient of the likelihood function with respect to the data and the primary parameters is tested against the finite difference approximation to the gradient, if these functions are provided. Again, the tolerance level can be specified individually.

Definition of a default distribution. This test ensures that the '`Distribution`' object generates a default distribution (generated with no arguments) which has the basic properties as tested by the other generic tests.

Setting and getting of primary parameters. A generic consistency check between the `primary2array` and `array2primary` functions for the primary parameters is performed. First, the export method is called, then the obtained array is slightly perturbed and fed into the import function. The primary parameters, which are now obtained via the export function, are compared to the original ones. The generic test passes, if these are indeed the same and hence the import and export functions are consistent with each other.

Each probabilistic model provided by the **Natter** is included in this generic test suite. However, to test the specific requirements for these models, there is an additional unit-test file for these models, also located in the **Natter**'s test directory.

8. Conclusion

We presented the natural image statistics toolbox **Natter** as a flexible and extensible framework for preprocessing, model estimation and model comparison on natural image patches. The **Natter** strongly relies on object orientation and inheritance to modularize and outsource computational building blocks in these models. This minimizes coding and testing efforts when integrating new models in the **Natter** framework. The **Natter** is designed such that models can easily be added, because only those methods that are needed by the user and other **Natter** features she wishes to use, must be implemented for a new object. All other features can be added later when required. Finally, the **Natter** comes with an extensive battery of tests that ensure the correct functionality and also allow to test the correctness of future models.

The toolbox described here is only the first step towards a comprehensive toolbox for natural images model comparison. So far, we mainly focused on providing a flexible framework and good baseline models against which more involved models can be compared. Since the Gaussian distribution is not a good baseline for natural image patches, we advocate L_p-spherically symmetric and L_p-nested models as such a baseline.

Now that these models are established, future extensions will focus on more complex models. Unfortunately, most of those models are trained without explicitly knowing their normalization constant using techniques like sampling, score matching, or noise contrastive divergence (Hyvärinen 2005; Gutmann and Hyvärinen 2012). Apart from a more difficult estimation procedure, these models also pose a strong challenge for model comparison. Since the normalization constant is unknown, the likelihood of test data must be estimated with sampling or other techniques, adding another layer of complexity. Getting good estimates for even small models is hard and a subject of active research (see, e.g., Theis, Gerwinn, Sinz, and Bethge 2011). However, since the aforementioned estimation methods are generic, a modularized toolbox like the **Natter** is a good platform to thoroughly implemented those techniques and use them for model comparison in order to foster advancement in natural image statistics research.

References

Bell AJ, Sejnowski TJ (1997). “The ‘Independent Components’ of Natural Scenes Are Edge Filters.” *Vision Research*, **37**(23), 3327–3338.

Besag J (1986). “On the Statistical Analysis of Dirty Pictures.” *Journal of the Royal Statistical Society B*, **48**(3), 259–302.

Bethge M (2006). “Factorial Coding of Natural Images: How Effective are Linear Models in Removing Higher-Order Dependencies?” *Journal of the Optical Society of America A*, **23**(6), 1253–1268.

Brandl G (2014). ***Sphinx**: Python Documentation Generator*. Python package version 1.2.2, URL http://sphinx-doc.org/.

Buchsbaum G, Gottschalk A (1983). “Trichromacy, Opponent Colours Coding and Optimum Colour Information Transmission in the Retina.” *Proceedings of the Royal Society B: Biological Sciences*, **220**(1218), 89–113.

Burton GJ, Moorhead IR (1987). "Color and Spatial Structure in Natural Scenes." *Applied Optics*, **26**(1), 157–170.

Chainais P (2007). "Infinitely Divisible Cascades to Model the Statistics of Natural Images." *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **29**(12), 2105–2119.

Chandler DM, Field DJ (2007). "Estimates of the Information Content and Dimensionality of Natural Scenes from Proximity Distributions." *Journal of the Optical Society of America A*, **24**(4), 922–941.

Comon P (1994). "Independent Component Analysis, A New Concept?" *Signal Processing*, **36**(3), 287–314.

Deriugin NG (1956). "The Power Spectrum and the Correlation Function of the Television Signal." *Telecommunications*, **1**, 1–12.

Eichhorn J, Sinz F, Bethge M (2009). "Natural Image Coding in V1: How Much Use Is Orientation Selectivity?" *PLoS Computational Biology*, **5**(4), e1000336.

Fernández C, Osiewalski J, Steel MFJ (1995). "Modeling and Inference with v-Spherical Distributions." *Journal of the American Statistical Association*, **90**(432), 1331–1340.

Field DJ (1987). "Relations Between the Statistics of Natural Images and the Response Properties of Cortical Cells." *Journal of the Optical Society of America A*, **4**(12), 2379–2394.

Goodman IR, Kotz S (1973). "Multivariate θ-Generalized Normal Distributions." *Journal of Multivariate Analysis*, **3**(2), 204–219.

Gupta AK, Song D (1997). "L_p-Norm Spherical Distribution." *Journal of Statistical Planning and Inference*, **60**(2), 241–260.

Gutmann MU, Hyvärinen A (2012). "Noise-Contrastive Estimation of Unnormalized Statistical Models, with Applications to Natural Image Statistics." *Journal of Machine Learning Research*, **13**(Feb), 307–361.

Hosseini R, Sinz F, Bethge M (2010). "Lower Bounds on the Redundancy of Natural Images." *Vision Research*, **50**(22), 2213–2222.

Hyvärinen A (2005). "Estimation of Non-Normalized Statistical Models by Score Matching." *Journal of Machine Learning Research*, **6**(Apr), 695–709.

Hyvärinen A, Hoyer P (2000). "Emergence of Phase- and Shift-Invariant Features by Decomposition of Natural Images into Independent Feature Subspaces." *Neural Computation*, **12**(7), 1705–1720.

Hyvärinen A, Hurri J, Vaeyrynen J (2003). "Bubbles: A Unifying Framework for Low-Level Statistical Properties of Natural Image Sequences." *Journal of the Optical Society of America A*, **20**(7), 1237–1252.

Hyvärinen A, Koester U (2007). "Complex Cell Pooling and the Statistics of Natural Images." *Network: Computation in Neural Systems*, **18**(2), 81–100.

Hyvärinen A, Oja E (1997). "A Fast Fixed-Point Algorithm for Independent Component Analysis." *Neural Computation*, **9**(7), 1483–1492.

Jeulin D, Villalobos IT, Dubus A (1995). "Morphological Analysis of UO 2 Powder Using a Dead Leaves Model." *Microscopy Microanalysis Microstructures*, **6**(4), 371–384.

Jones E, Oliphant TE, Peterson P (2001). ***SciPy**: Open Source Scientific Tools for Python.* URL http://www.scipy.org/.

Karklin Y, Lewicki MS (2008). "Emergence of Complex Cell Properties by Learning to Generalize in Natural Scenes." *Nature*, **457**(7225), 83–86.

Kretzmer ER (1952). "Statistics of Television Signals." *Bell System Technical Journal*, **31**(2), 751–763.

Kullback S, Leibler RA (1951). "On Information and Sufficiency." *The Annals of Mathematical Statistics*, **22**(1), 79–86.

Lee AB, Mumford DB, Huang J (2001). "Occlusion Models for Natural Images: A Statistical Study of a Scale-Invariant Dead Leaves Model." *International Journal of Computer Vision*, **41**(1), 35–59.

Lyu S, Simoncelli EP (2007). "Statistical Modeling of Images with Fields of Gaussian Scale Mixtures." In *Advances in Neural Information Processing Systems 19*, pp. 945–952. Curran Associates, Inc.

Lyu S, Simoncelli EP (2009). "Nonlinear Extraction of Independent Components of Natural Images Using Radial Gaussianization." *Neural Computation*, **21**(6), 1485–1519.

Manton JH (2002). "Optimization Algorithms Exploiting Unitary Constraints." *IEEE Transactions on Signal Processing*, **50**(3), 635–650.

Mumford DB, Gidas B (2001). "Stochastic Models for Generic Images." *Quarterly of Applied Mathematics*, **59**(1), 85–112.

Oliphant TE (2006). *Guide to **NumPy***. Provo. URL http://www.tramy.us/.

Olshausen BA, Field DJ (1996). "Emergence of Simple-Cell Receptive Field Properties by Learning a Sparse Code for Natural Images." *Nature*, **381**(6583), 607–609.

Osindero S, Hinton G (2008). "Modeling Image Patches with a Directed Hierarchy of Markov Random Fields." In JC Platt, D Koller, Y Singer, S Roweis (eds.), *Advances in Neural Information Processing Systems 20*, pp. 1121–1128. Curran Associates, Inc., Cambridge.

Osindero S, Welling M, Hinton GE (2006). "Topographic Product Models Applied to Natural Scene Statistics." *Neural Computation*, **18**(2), 381–414.

Paninski L (2003). "Estimation of Entropy and Mutual Information." *Neural Computation*, **15**(6), 1191–1253.

Ranzato MA, Hinton GE (2010). "Modeling Pixel Means and Covariances Using Factorized Third-Order Boltzmann Machines." In *IEEE Conference on Computer Vision and Pattern Recognition*, volume 9.

Ranzato MA, Krizhevsky A, Hinton GE (2010). "Factored 3-Way Restricted Boltzmann Machines For Modeling Natural Images." In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9, pp. 621–628.

Roth S, Black MJ (2005). "Fields of Experts: A Framework for Learning Image Priors." In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, volume 2, pp. 860–867.

Ruderman DL, Bialek W (1994). "Statistics of Natural Images: Scaling in the Woods." *Physical Review Letters*, **73**(6), 814.

Schreiber W (1956). "The Measurement of Third Order Probability Distributions of Television Signals." *IRE Transactions on Information Theory*, **2**(3), 94–105.

Scott DW (1979). "On Optimal and Data-Based Histograms." *Biometrika*, **66**(3), 605–610.

Serra J (1982). *Image Analysis and Mathematical Morphology*, volume 1. Academic Press.

Simoncelli EP (1997). "Statistical Models for Images: Compression, Restoration and Synthesis." In *Conference Record of the Thirty-First Asilomar Conference on Signals, Systems & Computers*, volume 1, pp. 673–678.

Sinz F, Bethge M (2009). "The Conjoint Effect of Divisive Normalization and Orientation Selectivity on Redundancy Reduction." In D Koller, D Schuurmans, Y Bengio, L Bottou (eds.), *Advances in Neural Information Processing Systems 21*, pp. 1521–1528. Curran Associates, Inc.

Sinz F, Bethge M (2010). "L_p-Nested Symmetric Distributions." *Journal of Machine Learning Research*, **11**(Dec), 3409–3451.

Sinz F, Gerwinn S, Bethge M (2009a). "Characterization of the p-Generalized Normal Distribution." *Journal of Multivariate Analysis*, **100**(5), 817–820.

Sinz F, Simoncelli EP, Bethge M (2009b). "Hierarchical Modeling of Local Image Features through L_p-Nested Symmetric Distributions." In Y Bengio, D Schuurmans, J Lafferty, C Williams, A Culotta (eds.), *Advances in Neural Information Processing Systems 22*, pp. 1696–1704. Curran Associates, Inc.

Srivastava A, Lee AB, Simoncelli EP, Zhu SC (2003). "On Advances in Statistical Modeling of Natural Images." *Journal of Mathematical Imaging and Vision*, **18**(1), 17–33.

Szabłowski PJ (1998). "Uniform Distributions on Spheres in Finite Dimensional L_α and Their Generalizations." *Journal of Multivariate Analysis*, **64**(2), 103–117.

Theis L, Gerwinn S, Sinz F, Bethge M (2011). "In All Likelihood, Deep Belief Is Not Enough." *Journal of Machine Learning Research*, **12**(Nov), 3071–3096.

Van Hateren JH, Van Der Schaaf A (1998). "Independent Component Filters of Natural Images Compared with Simple Cells in Primary Visual Cortex." *Proceedings of the Royal Society B: Biological Sciences*, **265**(1394), 359–366.

van Rossum G, *et al.* (2014). *Python Programming Language*. Python Software Foundation. URL http://www.python.org/.

Wainwright MJ, Simoncelli EP (2000). "Scale Mixtures of Gaussians and the Statistics of Natural Images." *Neural Information Processing Systems*, **12**(1), 855–861.

Welling M, Gehler PV (2005). "Products of "Edge-Perts"." In Y Weiss, B Schölkopf, JC Platt (eds.), *Advances in Neural Information Processing Systems 18*, pp. 419–426. MIT Press.

Winkler G (1995). *Image Analysis, Random Fields and Dynamic Monte Carlo Methods*, volume 771. Springer-Verlag.

Zhu SC, Wu YN, Mumford D (1997). "Minimax Entropy Principle and Its Application to Texture Modeling." *Neural Computation*, **9**(8), 1627–1660.

Zito T, Wilbert N, Wiskott L, Berkes P (2008). "Modular Toolkit for Data Processing (**MDP**): A Python Data Processing Framework." *Frontiers in Neuroinformatics*, **2**(8), 1–7.

Zoran D, Weiss Y (2009). "The "Tree-Dependent Components" of Natural Images are Edge Filters." *Vision Research*, **37**(23), 3327–38.

Affiliation:

Fabian Sinz
Department for Neuroethology, Universität Tübingen
Bernstein Center for Computational Neuroscience, Tübingen
72076 Tübingen, Germany
E-mail: fabee@bethgelab.org
URL: http://www.bethgelab.org/

Matthias Bethge
Werner Reichardt Centre for Integrative Neuroscience, University Tübingen
Bernstein Center for Computational, Neuroscience, Tübingen
Max Planck Institute for Biological Cybernetics, Tübingen
Otfried-Müller-Str. 25
72076 Tübingen, Germany
E-mail: matthias@bethgelab.org
URL: http://www.bethgelab.org/

Journal of Statistical Software http://www.jstatsoft.org/
published by the American Statistical Association http://www.amstat.org/
Volume 61, Issue 5 *Submitted:* 2013-04-21
October 2014 *Accepted:* 2014-05-23