Modular Toolkit for Data Processing (MDP)

This document is also available online: http://mdp-toolkit.sourceforge.net/tutorial.html

Tutorial

Author: Pietro Berkes, Niko Wilbert, and Tiziano Zito

Homepage: http://mdp-toolkit.sourceforge.net

Version: 2.4

This is a guide to basic and some more advanced features of the MDP library. Besides the present tutorial, you can learn more about MDP by using the standard Python tools. All MDP nodes have doc-strings, the public attributes and methods have telling names: All information about a node can be obtained using the `help` and `dir` functions within the Python interpreter. In addition to that, an automatically generated API is available.

Note

Code snippets throughout the script will be denoted by:

```
>>> print "Hello world!"
Hello world!
```

To run the following code examples don't forget to import mdp in your Python session with:

```
>>> import mdp
```

You'll find all the code of this tutorial within the `demo` directory in the MDP installation path.

Contents

Introduction

The use of the Python programming language in computational neuroscience has been growing steadily during the past few years. The maturation of two important open source projects, the scientific libraries NumPy and SciPy, gives access to a large collection of scientific functions that rivals in size and speed with well known commercial alternatives like The MathWorks™ Matlab©. Furthermore, the flexible and dynamic nature of Python offers the scientific programmer the opportunity to quickly develop efficient and structured software while maximizing prototyping and reusability capabilities.

The Modular toolkit for Data Processing (MDP) package contributes to this growing community a library of widely used data processing algorithms, and the possibility to combine them according to a pipeline analogy to build more complex data processing software.

MDP has been designed to be used as-is and as a framework for scientific data processing development.

From the user's perspective, MDP consists of a collection of supervised and unsupervised learning algorithms, and other data processing units (*nodes*) that can be combined into data processing sequences (*flows*) and more complex feed-forward network architectures. Given a set of input data, MDP takes care of successively training or executing all nodes in the network. This allows the user to specify complex algorithms as a series of simpler data processing steps in a natural way.

The base of available algorithms is steadily increasing and includes, to name but the most common, Principal Component Analysis (PCA and NIPALS), several Independent Component Analysis

algorithms (CuBICA, FastICA, TDSEP, and JADE), Slow Feature Analysis, Gaussian Classifiers, Restricted Boltzmann Machine, and Locally Linear Embedding (see the Node List section for a more exhaustive list and references).

Particular care has been taken to make computations efficient in terms of speed and memory. To reduce memory requirements, it is possible to perform learning using batches of data, and to define the internal parameters of the nodes to be single precision, which makes the usage of very large data sets possible. Moreover, the `parallel` subpackage offers a parallel implementation of the basic nodes and flows.

From the developer's perspective, MDP is a framework that makes the implementation of new supervised and unsupervised learning algorithms easy and straightforward. The basic class, `Node`, takes care of tedious tasks like numerical type and dimensionality checking, leaving the developer free to concentrate on the implementation of the learning and execution phases. Because of the common interface, the node then automatically integrates with the rest of the library and can be used in a network together with other nodes. A node can have multiple training phases and even an undetermined number of phases. This allows the implementation of algorithms that need to collect some statistics on the whole input before proceeding with the actual training, and others that need to iterate over a training phase until a convergence criterion is satisfied. The ability to train each phase using chunks of input data is maintained if the chunks are generated with iterators. Moreover, crash recovery is optionally available: in case of failure, the current state of the flow is saved for later inspection.

MDP is distributed under the open source LGPL license. It has been written in the context of theoretical research in neuroscience, but it has been designed to be helpful in any context where trainable data processing algorithms are used. Its simplicity on the user's side, the variety of readily available algorithms, and the reusability of the implemented nodes make it also a useful educational tool.

With over 10,000 downloads since its first public release in 2004, MDP has become a widely used Python scientific software. It has minimal dependencies, requiring only the NumPy numerical extension, is completely platform-independent, and is available as a package in the GNU/Linux Debian distribution and the Python(x,y) scientific Python distribution.

As the number of its users and contributors is increasing, MDP appears to be a good candidate for becoming a community-driven common repository of user-supplied, freely available, Python implemented data processing algorithms.

Quick Start

Using MDP is as easy as:

```
>>> import mdp
>>> # perform pca on some data x
...
>>> y = mdp.pca(x)
>>> # perform ica on some data x using single precision
...
>>> y = mdp.fastica(x, dtype='float32')
```

MDP requires the numerical Python extensions NumPy or SciPy. At import time MDP will select `scipy` if available, otherwise `numpy` will be loaded. You can force the use of a numerical extension by setting the environment variable `MDPNUMX=numpy` or `MDPNUMX=scipy`.

An important remark

Input array data is typically assumed to be two-dimensional and ordered such that observations of the same variable are stored on rows and different variables are stored on columns.

Nodes

A *node* is the basic building block of an MDP application. It represents a data processing element, like for example a learning algorithm, a data filter, or a visualization step (see the Node List section for an exhaustive list and references).

Each node can have one or more training phases, during which the internal structures are learned from training data (e.g. the weights of a neural network are adapted or the covariance matrix is estimated) and an execution phase, where new data can be processed forwards (by processing the data through the node) or backwards (by applying the inverse of the transformation computed by the node if defined).

Nodes have been designed to be applied to arbitrarily long sets of data: if the underlying algorithms supports it, the internal structures can be updated incrementally by sending multiple batches of data (this is equivalent to online learning if the chunks consists of single observations, or to batch learning if the whole data is sent in a single chunk). It is thus possible to perform computations on amounts of data that would not fit into memory or to generate data on-the-fly.

A `Node` also defines some utility methods, like for example `copy` and `save`, that return an exact copy of a node and save it in a file, respectively. Additional methods may be present, depending on the algorithm.

Node Instantiation

Nodes can be obtained by creating an instance of the `Node` class.

Each node is characterized by an input dimension (i.e., the dimensionality of the input vectors), an output dimension, and a `dtype`, which determines the numerical type of the internal structures and of the output signal. By default, these attributes are inherited from the input data if left unspecified. The constructor of each node class can require other task-specific arguments. The full documentation is always available in the doc-string of the node's class.

Some examples of node instantiation:

- Create a node that performs Principal Component Analysis (PCA) whose input dimension and `dtype` are inherited from the input data during training. Output dimensions default to input dimensions.

  ```
  >>> pcanode1 = mdp.nodes.PCANode()
  >>> pcanode1
  PCANode(input_dim=None, output_dim=None, dtype=None)
  ```

- Setting `output_dim = 10` means that the node will keep only the first 10 principal components of the input.

  ```
  >>> pcanode2 = mdp.nodes.PCANode(output_dim = 10)
  >>> pcanode2
  PCANode(input_dim=None, output_dim=10, dtype=None)
  ```

  The output dimensionality can also be specified in terms of the explained variance. If we want to keep the number of principal components which can account for 80% of the input variance, we set:

  ```
  >>> pcanode3 = mdp.nodes.PCANode(output_dim = 0.8)
  >>> pcanode3.desired_variance
  0.80000000000000004
  ```

- If `dtype` is set to `float32` (32-bit float), the input data is cast to single precision when received and the internal structures are also stored as `float32`. `dtype` influences the memory space necessary for a node and the precision with which the computations are performed.

```
>>> pcanode4 = mdp.nodes.PCANode(dtype = 'float32')
>>> pcanode4
PCANode(input_dim=None, output_dim=None, dtype='float32')
```

You can obtain a list of the numerical types supported by a node lookng at its `supported_dtypes` property:

```
>>> pcanode4.supported_dtypes
[dtype('float32'), dtype('float64')]
```

This attribute is a list of `numpy.dtype` objects.

- A `PolynomialExpansionNode` expands its input in the space of polynomals of a given degree by computing all monomials up to the specified degree. Its constructor needs as first argument the degree of the polynomials space (3 in this case).

```
>>> expnode = mdp.nodes.PolynomialExpansionNode(3)
```

Node Training

Some nodes need to be trained to perform their task. For example, the Principal Component Analysis (PCA) algorithm requires the computation of the mean and covariance matrix of a set of training data from which the principal eigenvectors of the data distribution are estimated.

This can be done during a training phases by calling the `train` method. MDP supports both supervised and unsupervised training, and algorithms with multiple training phases.

Some examples of node training:

- Create some random data to train the node

```
>>> x = mdp.numx_rand.random((100, 25))  # 25 variables, 100 observations
```

- Analyzes the batch of data `x` and update the estimation of mean and covariance matrix:

```
>>> pcanode1.train(x)
```

At this point the input dimension and the `dtype` have been inherited from `x`:

```
>>> pcanode1
PCANode(input_dim=25, output_dim=None, dtype='float64')
```

- We can train our node with more than one chunk of data. This is especially useful when the input data is too long to be stored in memory or when it has to be created on-the-fly. (See also the Iterators section):

```
>>> for i in range(100):
...     x = mdp.numx_rand.random((100, 25))
...     pcanode1.train(x)
>>>
```

- Some nodes don't need to or cannot be trained:

```
>>> expnode.is_trainable()
False
```

Trying to train them anyway would raise an `IsNotTrainableException`.

- The training phase ends when the `stop_training`, `execute`, `inverse`, and possibly some other node-specific methods are called. For example we can finalyze the PCA algorithm by computing and selecting the principal eigenvectors

```
>>> pcanode1.stop_training()
```

- If the `PCANode` was declared to have a number of output components dependent on the input variance to be explained, we can check after training the number of output components and the actually explained variance:

```
>>> pcanode3.train(x)
>>> pcanode3.stop_training()
>>> pcanode3.output_dim
16
>>> pcanode3.explained_variance
0.85261144755506446
```

  It is now possible to access the trained internal data. In general, a list of the interesting internal attributes can be found in the class documentation.

```
>>> avg = pcanode1.avg            # mean of the input data
>>> v = pcanode1.get_projmatrix() # projection matrix
```

- Some nodes, namely the one corresponding to supervised algorithms, e.g. Fisher Discriminant Analysis (FDA), may need some labels or other supervised signals to be passed during training. Detailed information about the signature of the `train` method can be read in its doc-string.

```
>>> fdanode = mdp.nodes.FDANode()
>>> for label in ['a', 'b', 'c']:
...     x = mdp.numx_rand.random((100, 25))
...     fdanode.train(x, label)
>>>
```

- A node could also require multiple training phases. For example, the training of `fdanode` is not complete yet, since it has two training phases: The first one computing the mean of the data conditioned on the labels, and the second one computing the overall and within-class covariance matrices and solving the FDA problem. The first phase must be stopped and the second one trained:

```
>>> fdanode.stop_training()
>>> for label in ['a', 'b', 'c']:
...     x = mdp.numx_rand.random((100, 25))
...     fdanode.train(x, label)
>>>
```

  The easiest way to train multiple phase nodes is using flows, which automatically handle multiple phases (see the Flows section).

Node Execution

Once the training is finished, it is possible to execute the node:

- The input data is projected on the principal components learned in the training phase:

```
>>> x = mdp.numx_rand.random((100, 25))
>>> y_pca = pcanode1.execute(x)
```

- Calling a node instance is equivalent to executing it:

```
>>> y_pca = pcanode1(x)
```

- The input data is expanded in the space of polynomials of degree 3:

```
>>> x = mdp.numx_rand.random((100, 5))
>>> y_exp = expnode(x)
```

- The input data is projected to the directions learned by FDA:

```
>>> x = mdp.numx_rand.random((100, 25))
>>> y_fda = fdanode(x)
```

- Some nodes may allow for optional arguments in the `execute` method. As always the complete information can be found in the doc-string.

Node Inversion

If the operation computed by the node is invertible, the node can also be executed *backwards*, thus computing the inverse transformation:

- In the case of PCA, for example, this corresponds to projecting a vector in the principal components space back to the original data space:

```
>>> pcanode1.is_invertible()
True
>>> x = pcanode1.inverse(y_pca)
```

- The expansion node in not invertible:

```
>>> expnode.is_invertible()
False
```

  Trying to compute the inverse would raise an `IsNotInvertibleException`.

Writing your own nodes: subclassing Node

MDP tries to make it easy to write new nodes that interface with the existing data processing elements.

The `Node` class is designed to make the implementation of new algorithms easy and intuitive. This base class takes care of setting input and output dimension and casting the data to match the numerical type (e.g. float or double) of the internal variables, and offers utility methods that can be used by the developer.

To expand the MDP library of implemented nodes with user-made nodes, it is sufficient to subclass `Node`, overriding some of the methods according to the algorithm one wants to implement, typically the `_train`, `_stop_training`, and `_execute` methods.

In its namespace MDP offers references to the main modules `numpy` or `scipy`, and the subpackages `linalg`, `random`, and `fft` as `mdp.numx`, `mdp.numx_linalg`, `mdp.numx_rand`, and `mdp.numx_fft`. This is done to possibly support additional numerical extensions in the future. For this reason it is recommended to refer to the `numpy` or `scipy` numerical extensions through the MDP aliases `mdp.numx`, `mdp.numx_linalg`, `mdp.numx_fft`, and `mdp.numx_rand` when writing `Node` subclasses. This shall ensure that your nodes can be used without modifications should MDP support alternative numerical extensions in the future.

We'll illustrate all this with some toy examples.

- We start by defining a node that multiplies its input by 2.

  Define the class as a subclass of `Node`:

```
>>> class TimesTwoNode(mdp.Node):
```

This node cannot be trained. To specify this, one has to overwrite the `is_trainable` method to return False:

```
...     def is_trainable(self):
...         return False
```

Execute only needs to multiply `x` by 2:

```
...     def _execute(self, x):
...         return 2*x
```

Note that the `execute` method, which should never be overwritten and which is inherited from the `Node` parent class, will perform some tests, for example to make sure that `x` has the right rank, dimensionality and casts it to have the right `dtype`. After that the user-supplied `_execute` method is called. Each subclass has to handle the `dtype` defined by the user or inherited by the input data, and make sure that internal structures are stored consistently. To help with this the `Node` base class has a method called `_refcast(array)` that casts the input `array` only when its `dtype` is different from the `Node` instance's `dtype`.

The inverse of the multiplication by 2 is of course the division by 2:

```
...     def _inverse(self, y):
...         return y/2
...
>>>
```

Test the new node:

```
>>> node = TimesTwoNode(dtype = 'int32')
>>> x = mdp.numx.array([[1.0, 2.0, 3.0]])
>>> y = node(x)
>>> print x, '* 2 =  ', y
[ [ 1.  2.  3.]] * 2 =   [ [2 4 6]]
>>> print y, '/ 2 =', node.inverse(y)
[ [2 4 6]] / 2 = [ [1 2 3]]
```

- We then define a node that raises the input to the power specified in the initializer:

```
>>> class PowerNode(mdp.Node):
```

We redefine the init method to take the power as first argument. In general one should always give the possibility to set the `dtype` and the input dimensions. The default value is `None`, which means that the exact value is going to be inherited from the input data:

```
...     def __init__(self, power, input_dim=None, dtype=None):
```

Initialize the parent class:

```
...         super(PowerNode, self).__init__(input_dim=input_dim, dtype=dtype)
```

Store the power:

```
...         self.power = power
```

`PowerNode` is not trainable...

```
...     def is_trainable(self):
...         return False
```

... nor invertible:

```
...     def is_invertible(self):
...         return False
```

It is possible to overwrite the function `_get_supported_dtypes` to return a list of `dtype` supported by the node:

```
...     def _get_supported_dtypes(self):
...         return ['float32', 'float64']
```

The supported types can be specified in any format allowed by the `numpy.dtype` constructor. The interface method `get_supported_dtypes` converts them and sets the property `supported_dtypes`, which is a list of `numpy.dtype` objects.

The `_execute` method:

```
...     def _execute(self, x):
...         return self._refcast(x**self.power)
...
>>>
```

Test the new node:

```
>>> node = PowerNode(3)
>>> x = mdp.numx.array([[1.0, 2.0, 3.0]])
>>> y = node(x)
>>> print x, '**', node.power, '=', node(x)
[ [ 1.  2.  3.]] ** 3 = [ [  1.   8.  27.]]
```

- We now define a node that needs to be trained. The `MeanFreeNode` computes the mean of its training data and subtracts it from the input during execution:

```
>>> class MeanFreeNode(mdp.Node):
...     def __init__(self, input_dim=None, dtype=None):
...         super(MeanFreeNode, self).__init__(input_dim=input_dim,
...                                            dtype=dtype)
```

We store the mean of the input data in an attribute. We initialize it to `None` since we still don't know how large is an input vector:

```
...         self.avg = None
```

Same for the number of training points:

```
...         self.tlen = 0
```

The subclass only needs to overwrite the `_train` method, which will be called by the parent `train` after some testing and casting has been done:

```
...     def _train(self, x):
...         # Initialize the mean vector with the right
...         # size and dtype if necessary:
...         if self.avg is None:
...             self.avg = mdp.numx.zeros(self.input_dim,
...                                       dtype=self.dtype)
```

Update the mean with the sum of the new data:

```
...         self.avg += mdp.numx.sum(x, axis=0)
```

Count the number of points processed:

```
...         self.tlen += x.shape[0]
```

Note that the `train` method can have further arguments, which might be useful to implement algorithms that require supervised learning. For example, if you want to define a node that performs some form of classification you can define a `_train(self, data, labels)` method. The parent `train` checks `data` and takes care to pass the `labels` on (cf. for example `mdp.nodes.FDANode`).

The `_stop_training` function is called by the parent `stop_training` method when the training phase is over. We divide the sum of the training data by the number of training vectors to obtain the mean:

```
...     def _stop_training(self):
...         self.avg /= self.tlen
...         if self.output_dim is None:
...             self.output_dim = self.input_dim
```

Note that we `input_dim` are set automatically by the `train` method, and we want to ensure that the node has `output_dim` set after training. For nodes that do not need training, the setting is performed automatically upon execution. The `_execute` and `_inverse` methods:

```
...     def _execute(self, x):
...         return x - self.avg
...     def _inverse(self, y):
...         return y + self.avg
...
>>>
```

Test the new node:

```
>>> node = MeanFreeNode()
>>> x = mdp.numx_rand.random((10,4))
>>> node.train(x)
>>> y = node(x)
>>> print 'Mean of y (should be zero): ', mdp.numx.mean(y, 0)
Mean of y (should be zero):  [  0.00000000e+00   2.22044605e-17
-2.22044605e-17   1.11022302e-17]
```

- It is also possible to define nodes with multiple training phases. In such a case, calling the `train` and `stop_training` functions multiple times is going to execute successive training phases (this kind of node is much easier to train using Flows). Here we'll define a node that returns a meanfree, unit variance signal. We define two training phases: first we compute the mean of the signal and next we sum the squared, meanfree input to compute the standard deviation (of course it is possible to solve this problem in one single step - remeber this is just a toy example).

```
>>> class UnitVarianceNode(mdp.Node):
...     def __init__(self, input_dim=None, dtype=None):
...         super(UnitVarianceNode, self).__init__(input_dim=input_dim,
...                                                dtype=dtype)
...         self.avg = None # average
...         self.std = None # standard deviation
...         self.tlen = 0
```

The training sequence is defined by the user-supplied method `_get_train_seq`, that returns a list of tuples, one for each training phase. The tuples contain references to the training and stop-training methods of each of them. The default output of this method is `[(_train, _stop_training)]`, which explains the standard behavior illustrated above. We overwrite the method to return the list of our training/stop_training methods:

```
...     def _get_train_seq(self):
...         return [(self._train_mean, self._stop_mean),
...                 (self._train_std, self._stop_std)]
```

Next we define the training methods. The first phase is identical to the one in the previous example:

```
...     def _train_mean(self, x):
...         if self.avg is None:
...             self.avg = mdp.numx.zeros(self.input_dim,
...                                       dtype=self.dtype)
...         self.avg += mdp.numx.sum(x, 0)
...         self.tlen += x.shape[0]
...     def _stop_mean(self):
...         self.avg /= self.tlen
```

The second one is only marginally different and does not require many explanations:

```
...     def _train_std(self, x):
...         if self.std is None:
...             self.tlen = 0
...             self.std = mdp.numx.zeros(self.input_dim,
...                                       dtype=self.dtype)
...         self.std += mdp.numx.sum((x - self.avg)**2., 0)
...         self.tlen += x.shape[0]
...     def _stop_std(self):
...         # compute the standard deviation
...         self.std = mdp.numx.sqrt(self.std/(self.tlen-1))
```

The `_execute` and `_inverse` methods are not surprising, either:

```
...     def _execute(self, x):
...         return (x - self.avg)/self.std
...     def _inverse(self, y):
...         return y*self.std + self.avg
>>>
```

Test the new node:

```
>>> node = UnitVarianceNode()
>>> x = mdp.numx_rand.random((10,4))
>>> # loop over phases
... for phase in range(2):
...     node.train(x)
...     node.stop_training()
...
...
>>> # execute
... y = node(x)
>>> print 'Standard devia-
tion of y (should be one): ', mdp.numx.std(y, axis=0)
Standard deviation of y (should be one):  [ 1.  1.  1.  1.]
```

- In our last example we'll define a node that returns two copies of its input. The output is going to have twice as many dimensions.

```
>>> class TwiceNode(mdp.Node):
...     def is_trainable(self): return False
...     def is_invertible(self): return False
```

When `Node` inherits the input dimension, output dimension, and `dtype` from the input data, it calls the methods `set_input_dim`, `set_output_dim`, and `set_dtype`. Those are the setters for `input_dim`, `output_dim` and `dtype`, which are Python properties. If a subclass needs to change the default behaviour, the internal methods `_set_input_dim`, `_set_output_dim` and `_set_dtype` can be overwritten. The property setter will call the internal method after some basic testing and internal settings. The private methods `_set_input_dim`, `_set_output_dim` and `_set_dtype` are responsible for setting the private attributes `_input_dim`, `_output_dim`, and `_dtype` that contain the actual value.

Here we overwrite `_set_input_dim` to automatically set the output dimension to be twice the input one, and `_set_output_dim` to raise an exception, since the output dimension should not be set explicitly.

```
...     def _set_input_dim(self, n):
...         self._input_dim = n
...         self._output_dim = 2*n
...     def _set_output_dim(self, n):
...         raise mdp.NodeException, "Output dim can not be set explicitly!"
```

The `_execute` method:

```
...     def _execute(self, x):
...         return mdp.numx.concatenate((x, x), 1)
...
>>>
```

Test the new node

```
>>> node = TwiceNode()
>>> x = mdp.numx.zeros((5,2))
>>> x
array([[0, 0],
       [0, 0],
       [0, 0],
       [0, 0],
       [0, 0]])
>>> node.execute(x)
array([[0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]])
```

Flows

A *flow* is a sequence of nodes that are trained and executed together to form a more complex algorithm. Input data is sent to the first node and is successively processed by the subsequent nodes along the sequence.

Using a flow as opposed to handling manually a set of nodes has a clear advantage: The general flow implementation automatizes the training (including supervised training and multiple training phases), execution, and inverse execution (if defined) of the whole sequence.

Crash recovery is optionally available: in case of failure the current state of the flow is saved for later inspection. A subclass of the basic flow class (`CheckpointFlow`) allows user-supplied checkpoint functions to be executed at the end of each phase, for example to save the internal structures of a node for later analysis. Flow objects are Python containers. Most of the builtin `list` methods are available. A `Flow` can be saved or copied using the corresponding `save` and `copy` methods.

Flow instantiation, training and execution

For example, suppose we need to analyze a very high-dimensional input signal using Independent Component Analysis (ICA). To reduce the computational load, we would like to reduce the input dimensionality of the data using PCA. Moreover, we would like to find the data that produces local maxima in the output of the ICA components on a new test set (this information could be used for instance to characterize the ICA filters).

We start by generating some input signal at random (which makes the example useless, but it's just for illustration...). Generate 1000 observations of 20 independent source signals:

```
>>> inp = mdp.numx_rand.random((1000, 20))
```

Rescale x to have zero mean and unit variance:

```
>>> inp = (inp - mdp.numx.mean(inp, 0))/mdp.numx.std(inp, 0)
```

We reduce the variance of the last 15 components, so that they are going to be eliminated by PCA:

```
>>> inp[:,5:] /= 10.0
```

Mix the input signals linearly:

```
>>> x = mdp.utils.mult(inp,mdp.numx_rand.random((20, 20)))
```

`x` is now the training data for our simulation. In the same way we also create a test set `x_test`.

```
>>> inp_test = mdp.numx_rand.random((1000, 20))
>>> inp_test = (inp_test -
mdp.numx.mean(inp_test, 0))/mdp.numx.std(inp_test, 0)
>>> inp_test[:,5:] /= 10.0
>>> x_test = mdp.utils.mult(inp_test, mdp.numx_rand.random((20, 20)))
```

- We could now perform our analysis using only nodes, that's the lenghty way...

  1. Perform PCA:

     ```
     >>> pca = mdp.nodes.PCANode(output_dim=5)
     >>> pca.train(x)
     >>> out1 = pca(x)
     ```

  2. Perform ICA using CuBICA algorithm:

     ```
     >>> ica = mdp.nodes.CuBICANode()
     >>> ica.train(out1)
     >>> out2 = ica(out1)
     ```

  3. Find the three largest local maxima in the output of the ICA node when applied to the test data, using a `HitParadeNode`:

     ```
     >>> out1_test = pca(x_test)
     >>> out2_test = ica(out1_test)
     >>> hitnode = mdp.nodes.HitParadeNode(3)
     >>> hitnode.train(out2_test)
     >>> maxima, indices = hitnode.get_maxima()
     ```

- ... or we could use flows, which is the best way:

```
>>> flow = mdp.Flow([mdp.nodes.PCANode(output_dim=5), mdp.nodes.CuBICANode()])
```

Note that flows can be built simply by concatenating nodes:

```
>>> flow = mdp.nodes.PCANode(output_dim=5) + mdp.nodes.CuBICANode()
```

Train the resulting flow:

```
>>> flow.train(x)
```

Now the training phase of PCA and ICA are completed. Next we append a `HitParadeNode` which we want to train on the test data:

```
>>> flow.append(mdp.nodes.HitParadeNode(3))
```

As before, new nodes can be appended to an existing flow by adding them ot it:

```
>>> flow += mdp.nodes.HitParadeNode(3)
```

Train the `HitParadeNode` on the test data:

```
>>> flow.train(x_test)
>>> maxima, indices = flow[2].get_maxima()
```

A single call to the `flow`'s `train` method will automatically take care of training nodes with multiple training phases, if such nodes are present.

Just to check that everything works properly, we can calculate covariance between the generated sources and the output (should be approximately 1):

```
>>> out = flow.execute(x)
>>> cov = mdp.numx.amax(abs(mdp.utils.cov2(inp[:,:5], out)))
>>> print cov
[ 0.98992083  0.99244511  0.99227319  0.99663185  0.9871812 ]
```

The `HitParadeNode` is an analysis node and as such does not interfere with the data flow.

Note that flows can be executed by calling the `Flow` instance directly:

```
>>> out = flow(x)
```

Flow inversion

Flows can be inverted by calling their `inverse` method. In the case where the flow contains non-invertible nodes, trying to invert it would raise an exception. In this case, however, all nodes are invertible. We can reconstruct the mix by inverting the flow:

```
>>> rec = flow.inverse(out)
```

Calculate covariance between input mix and reconstructed mix: (should be approximately 1)

```
>>> cov = mdp.numx.amax(abs(mdp.utils.cov2(x/mdp.numx.std(x,axis=0),
...                                        rec/mdp.numx.std(rec,axis=0))))
>>> print cov
[ 0.99839606  0.99744461  0.99616208  0.99772863  0.99690947
  0.99864056  0.99734378  0.98722502  0.98118101  0.99407939
  0.99683096  0.99756988  0.99664384  0.99723419  0.9985529
  0.99829763  0.9982712   0.99721741  0.99682906  0.98858858]
```

Flows are container type objects

`Flow` objects are defined as Python containers, and thus are endowed with most of the methods of Python lists.

You can loop through a `Flow`:

```
>>> for node in flow:
...     print repr(node)
...
PCANode(input_dim=20, output_dim=5, dtype='float64')
CuBICANode(input_dim=5, output_dim=5, dtype='float64')
HitParadeNode(input_dim=5, output_dim=5, dtype='float64')
HitParadeNode(input_dim=5, output_dim=5, dtype='float64')
>>>
```

You can get slices, `pop`, `insert`, and `append` nodes:

```
>>> len(flow)
4
>>> print flow[::2]
[PCANode, HitParadeNode]
>>> nodetoberemoved = flow.pop(-1)
>>> nodetoberemoved
HitParadeNode(input_dim=5, output_dim=5, dtype='float64')
>>> len(flow)
3
```

Finally, you can concatenate flows:

```
>>> dummyflow = flow[1:].copy()
>>> longflow = flow + dummyflow
>>> len(longflow)
4
```

The returned flow must always be consistent, i.e. input and output dimensions of successive nodes always have to match. If you try to create an inconsistent flow you'll get an exception.

Crash recovery

If a node in a flow fails, you'll get a traceback that tells you which node has failed. You can also switch the crash recovery capability on. If something goes wrong you'll end up with a pickle dump of the flow, that can be later inspected.

To see how it works let's define a bogus node that always throws an `Exception` and put it into a flow:

```
>>> class BogusExceptNode(mdp.Node):
...    def train(self,x):
...        self.bogus_attr = 1
...        raise Exception, "Bogus Exception"
...    def execute(self,x):
...        raise Exception, "Bogus Exception"
...
>>> flow = mdp.Flow([BogusExceptNode()])
```

Switch on crash recovery:

```
>>> flow.set_crash_recovery(1)
```

Attempt to train the flow:

```
>>> flow.train(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  [...]
mdp.linear_flows.FlowExceptionCR:
----------------------------------------
! Exception in node #0 (BogusExceptNode):
Node Traceback:
Traceback (most recent call last):
  [...]
Exception: Bogus Exception
----------------------------------------
A crash dump is available on: "/tmp/MDPcrash_LmISO_.pic"
```

You can give a file name to tell the flow where to save the dump:

```
>>> flow.set_crash_recovery('/home/myself/mydumps/MDPdump.pic')
```

Iterators

Python allows user-defined classes to support iteration, as described in the Python docs. A convenient implementation of the iterator protocol is provided by generators: see this article for an introduction, and the official PEP for a complete description.

Let us define two bogus node classes to be used as examples of nodes:

```
>>> class BogusNode(mdp.Node):
...     """This node does nothing."""
...     def _train(self, x):
...         pass
...
>>> class BogusNode2(mdp.Node):
...     """This node does nothing. But it's not trainable nor invertible.
...     """
...     def is_trainable(self): return False
...     def is_invertible(self): return False
...
>>>
```

This generator generates `blocks` input blocks to be used as training set. In this example one block is a 2-dimensional time series. The first variable is [2,4,6,....,1000] and the second one [0,1,3,5,...,999]. All blocks are equal, this of course would not be the case in a real-life example.

In this example we use a progress bar to get progress information.

```
>>> def gen_data(blocks):
...     for i in mdp.utils.progressinfo(xrange(blocks)):
...         block_x = mdp.numx.atleast_2d(mdp.numx.arange(2,1001,2))
...         block_y = mdp.numx.atleast_2d(mdp.numx.arange(1,1001,2))
...         # put variables on columns and observations on rows
...         block = mdp.numx.transpose(mdp.numx.concatenate([block_x,block_y]))
...         yield block
...
>>>
```

The `progressinfo` function is a fully configurable text-mode progress info box tailored to the command-line die-hards. Have a look at its doc-string and prepare to be amazed!

Let's define a bogus flow consisting of 2 `BogusNode`:

```
>>> flow = mdp.Flow([BogusNode(),BogusNode()], verbose=1)
```

Train the first node with 5000 blocks and the second node with 3000 blocks. Note that the only allowed argument to `train` is a sequence (list or tuple) of iterators. In case you don't want or need to use incremental learning and want to do a one-shot training, you can use as argument to `train` a single array of data:

block-mode training

```
>>> flow.train([gen_data(5000),gen_data(3000)])
Training node #0 (BogusNode)
[==================================100%==================================>]

Training finished
Training node #1 (BogusNode)
[==================================100%==================================>]

Training finished
Close the training phase of the last node
```

one-shot training using one single set of data for both nodes

```
>>> flow = BogusNode() + BogusNode()
>>> block_x = mdp.numx.atleast_2d(mdp.numx.arange(2,1001,2))
>>> block_y = mdp.numx.atleast_2d(mdp.numx.arange(1,1001,2))
>>> sin-
gle_block = mdp.numx.transpose(mdp.numx.concatenate([block_x,block_y]))
>>> flow.train(single_block)
```

If your flow contains non-trainable nodes, you must specify a `None` iterator for the non-trainable nodes:

```
>>> flow = mdp.Flow([BogusNode2(),BogusNode()], verbose=1)
>>> flow.train([None, gen_data(5000)])
Training node #0 (BogusNode2)
Training finished
Training node #1 (BogusNode)
[==================================100%==================================>]

Training finished
Close the training phase of the last node
```

You can use the one-shot training:

```
>>> flow = mdp.Flow([BogusNode2(),BogusNode()], verbose=1)
>>> flow.train(single_block)
Training node #0 (BogusNode2)
Training finished
Training node #1 (BogusNode)
Training finished
Close the training phase of the last node
```

Iterators can be used also for execution (and inversion):

```
>>> flow = mdp.Flow([BogusNode(),BogusNode()], verbose=1)
>>> flow.train([gen_data(1), gen_data(1)])
Training node #0 (BogusNode)
Training finished
Training node #1 (BosgusNode)
[==================================100%==================================>]

Training finished
Close the training phase of the last node
>>> output = flow.execute(gen_data(1000))
[==================================100%==================================>]
>>> output = flow.inverse(gen_data(1000))
[==================================100%==================================>]
```

Execution and inversion can be done in one-shot mode also. Note that since training is finished you are not going to get a warning

```
>>> output = flow(single_block)
>>> output = flow.inverse(single_block)
```

If a node requires multiple training phases (e.g., `GaussianClassifierNode`), `Flow` automatically takes care of reusing the iterator multiple times. In this case generators are not allowed, since they *expire* after yielding the last data block. If you try to restart them, they raise a `StopIteration` exception. General iterators, instead, can always be restarted. For example, you can loop over a list as many times as you need.

However, it is fairly easy to wrap a generator in a simple iterator if you need to:

```
>>> class SimpleIterator(object):
...     def __init__(self, blocks):
...         self.blocks = blocks
...     def __iter__(self):
...         # this is a generator
...         for i in range(self.blocks):
...             yield generate_some_data()
>>>
```

Note that if you use random numbers within the iterator, you usually would like to reset the random number generator to produce the same sequence every time:

```
>>> class RandomIterator(object):
...     def __init__(self):
...         self.state = None
...     def __iter__(self):
...         if self.state is None:
...             self.state = mdp.numx_rand.get_state()
...         else:
...             mdp.numx_rand.set_state(self.state)
...         for i in range(2):
...             yield mdp.numx_rand.random((1,4))
>>> iterator = RandomIterator()
>>> for x in iterator:
...     print x
...
[[ 0.99586495  0.53463386  0.6306412   0.09679571]]
[[ 0.51117469  0.46647448  0.95089738  0.94837122]]
```

```
>>> for x in iterator:
...     print x
...
[[ 0.99586495  0.53463386  0.6306412   0.09679571]]
[[ 0.51117469  0.46647448  0.95089738  0.94837122]]
```

Checkpoints

It can sometimes be useful to execute arbitrary functions at the end of the training or execution phase, for example to save the internal structures of a node for later analysis. This can easily be done by defining a `CheckpointFlow`. As an example imagine the following situation: you want to perform Principal Component Analysis (PCA) on your data to reduce the dimensionality. After this you want to expand the signals into a nonlinear space and then perform Slow Feature Analysis to extract slowly varying signals. As the expansion will increase the number of components, you don't want to run out of memory, but at the same time you want to keep as much information as possible after the dimensionality reduction. You could do that by specifying the percentage of the total input variance that has to be conserved in the dimensionality reduction. As the number of output components of the PCA node now can become as large as the that of the input components, you want to check, after training the PCA node, that this number is below a certain threshold. If this is not the case you want to abort the execution and maybe start again requesting less variance to be kept.

Let start defining a generator to be used through the whole example:

```
>>> def gen_data(blocks,dims):
...     mat = mdp.numx_rand.random((dims,dims))-0.5
...     for i in xrange(blocks):
...         # put variables on columns and observations on rows
...         block = mdp.utils.mult(mdp.numx_rand.random((1000,dims)), mat)
...         yield block
...
>>>
```

Define a `PCANode` which reduces dimensionality of the input, a `PolynomialExpansionNode` to expand the signals in the space of polynomials of degree 2 and a `SFANode` to perform SFA:

```
>>> pca = mdp.nodes.PCANode(output_dim=0.9)
>>> exp = mdp.nodes.PolynomialExpansionNode(2)
>>> sfa = mdp.nodes.SFANode()
```

As you see we have set the output dimension of the `PCANode` to be `0.9`. This means that we want to keep at least 90% of the variance of the original signal. We define a `PCADimensionExceededException` that has to be thrown when the number of output components exceeds a certain threshold:

```
>>> class PCADimensionExceededException(Exception):
...     """Exception base class for PCA exceeded dimensions case."""
...     pass
...
>>>
```

Then, write a `CheckpointFunction` that checks the number of output dimensions of the `PCANode` and aborts if this number is larger than `max_dim`:

```
>>> class CheckPCA(mdp.CheckpointFunction):
...     def __init__(self,max_dim):
...         self.max_dim = max_dim
...     def __call__(self,node):
```

```
...         node.stop_training()
...         act_dim = node.get_output_dim()
...         if act_dim > self.max_dim:
...             errstr = 'PCA output dimensions exceeded maximum '+\
...                      '(%d > %d)'%(act_dim,self.max_dim)
...             raise PCADimensionExceededException, errstr
...         else:
...             print 'PCA output dimensions = %d'%(act_dim)
...
>>>
```

Define the CheckpointFlow:

```
>>> flow = mdp.CheckpointFlow([pca, exp, sfa])
```

To train it we have to supply 3 generators and 3 checkpoint functions:

```
>>> flow.train([gen_data(10, 50), None, gen_data(10, 50)],
...            [CheckPCA(10), None, None])
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
  [...]
__main__.PCADimensionExceededException: PCA output dimensions exceeded maxi-
mum (25 > 10)
```

The training fails with a `PCADimensionExceededException`. If we only had 12 input dimensions instead of 50 we would have passed the checkpoint:

```
>>> flow[0] = mdp.nodes.PCANode(output_dim=0.9)
>>> flow.train([gen_data(10, 12), None, gen_data(10, 12)],
...            [CheckPCA(10), None, None])
PCA output dimensions = 6
```

We could use the built-in `CheckpoinSaveFunction` to save the `SFANode` and analyze the results later :

```
>>> pca = mdp.nodes.PCANode(output_dim=0.9)
>>> exp = mdp.nodes.PolynomialExpansionNode(2)
>>> sfa = mdp.nodes.SFANode()
>>> flow = mdp.CheckpointFlow([pca, exp, sfa])
>>> flow.train([gen_data(10, 12), None, gen_data(10, 12)],
...            [CheckPCA(10),
...             None,
...             mdp.CheckpointSaveFunction('dummy.pic',
...                                        stop_training = 1,
...                                        protocol = 0)])
...
PCA output dimensions = 7
```

We can now reload and analyze the `SFANode`:

```
>>> fl = file('dummy.pic')
>>> import cPickle
>>> sfa_reloaded = cPickle.load(fl)
>>> sfa_reloaded
SFANode(input_dim=35, output_dim=35, dtype='d')
```

Don't forget to clean the rubbish:

```
>>> fl.close()
>>> import os
>>> os.remove('dummy.pic')
```

Hierarchical Networks

In case the desired data processing application cannot be defined as a sequence of nodes, the `hinet` subpackage makes it possible to construct arbitrary feed-forward architectures, and in particular hierarchical networks.

Building blocks

The `hinet` package contains three basic buiding blocks (which are all nodes themselves) to construct hierarchical node networks: `Layer`, `FlowNode`, `Switchboard`.

- The first building block is the `Layer` node, which works like a horizontal version of flow. It acts as a wrapper for a set of nodes that are trained and executed in parallel. For example, we can combine two nodes with 100 dimensional input to construct a layer with a 200-dimensional input

  ```
  >>> node1 = mdp.nodes.PCANode(input_dim=100, output_dim=10)
  >>> node2 = mdp.nodes.SFANode(input_dim=100, output_dim=20)
  >>> layer = mdp.hinet.Layer([node1, node2])
  >>> layer
  Layer(input_dim=200, output_dim=30, dtype=None)
  ```

  The first half of the 200 dimensional input data is then automatically assigned to `node1` and the second half to `node2`. We can train and execute a `Layer` just like any other node. Note that the dimensions of the nodes must be already set when the layer is constructed.

- In order to be able to build arbitrary feed-forward node structures, `hinet` provides a wrapper class for flows (i.e., vertical stacks of nodes) called `FlowNode`. For example, we can replace `node1` in the above example with a `FlowNode`:

  ```
  >>> node1_1 = mdp.nodes.PCANode(input_dim=100, output_dim=50)
  >>> node1_2 = mdp.nodes.SFANode(input_dim=50, output_dim=10)
  >>> node1_flow = mdp.Flow([node1_1, node1_2])
  >>> node1 = mdp.hinet.FlowNode(node1_flow)
  >>> layer = mdp.hinet.Layer([node1, node2])
  >>> layer
  Layer(input_dim=200, output_dim=30, dtype=None)
  ```

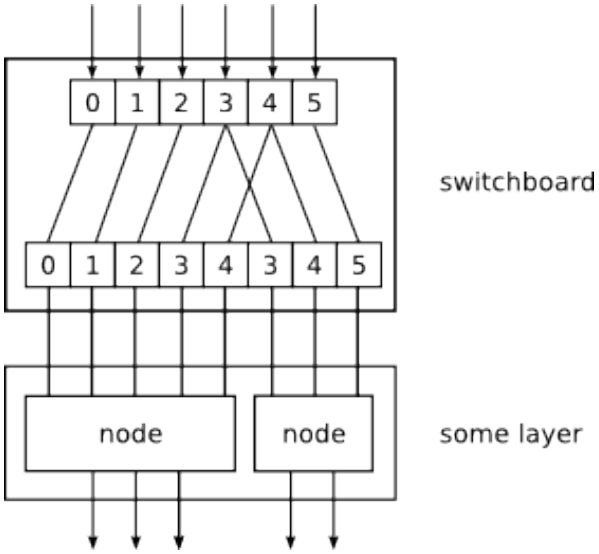
  in this example `node1` has two training phases (one for each internal node). Therefore `layer` now has two training phases as well and behaves like any other node with two training phases. By combining and nesting `FlowNode` and `Layer`, it is thus possible to build complex node structures.

- When implementing networks one might have to route different parts of the data to different nodes in a layer in complex ways. This is done by the `Switchboard` node, which can handle such the routing. A `Switchboard` is initialized with a 1-D Array with one entry for each output connection, containing the corresponding index of the input connection that it receives its input from, e.g.:

  ```
  >>> switchboard = mdp.hinet.Switchboard(input_dim=6, connec-
  tions=[0,1,2,3,4,3,4,5])
  >>> switchboard
  Switchboard(input_dim=3, output_dim=2, dtype=None)
  ```

```
>>> x = mdp.numx.array([[2,4,6,8,10,12]])
>>> switchboard.execute(x)
array([[ 2,  4,  6,  8, 10,  8, 10, 12]])
```

The switchboard can then be followed by a layer that splits the routed input to the appropriate nodes, as illustrated in following picture:



By combining layers with switchboards one can realize any feed-forward network topology. Defining the switchboard routing manually can be quite tedious. One way to automatize this is by defining switchboard subclasses for special routing situations. The `Rectangular2dSwitchboard` class is one such example and will be briefly described in a later example.

HTML representation

Since hierarchical networks can be quite complicated, `hinet` includes the class `HiNetHTML` that translates an MDP flow into a graphical visualization in an HTML file. After instantiating the class with a given HTML file one can pass any flow to it (we use the layer from above):

```
>>> file = open("test.html")
>>> file.write('<html>\n<head>\n<title>HiNetHTML Test</title>\n</head>\n<body>\n')
>>> hinet_html = mdp.hinet.HiNetHTML(file)
>>> flow = mdp.Flow([layer])
>>> hinet_html.parse_flow(flow)
>>> file.write('</body>\n</html>')
>>> file.close()
```

`file` now includes the HTML representation for the flow consisting of the layer. In the example below we will show such a representation for a more complicated example.

It is possible to include some internal node parameters in the representation (especially for newly defined custom nodes). This is actually very easy, the source code of this module contains more instructions on how to do this. It is also possible to modify the HTML presentation by providing a custom CSS string.

Parallelization

The `parallel` adds the ability to parallelize the training and execution of MPD flows. This package is split into two decoupled parts:

- The first part consists of parallel versions of the familiar MDP structures of nodes and flows. The first basic building block is the abstact base class `ParallelNode` for nodes which can be trained in a parallelized way. Secondly there is the `ParallelFlow` class, which splits the training or execution into tasks which can then be processes in parallel.

- The second part consists of the schedulers. A scheduker takes tasks and processes them in a more or less parallel way (e.g. in multiple python processes). A scheduler deals with the more technical aspects of the parallelization, but does not need to know anything about nodes and flows.

Practically all the complexities of parallelization are hidden in this package. There are also helper functions so that the parallel trainign or execuition can be done in one line of code.

Basic Examples

In the following example we parallelize a simple Flow consisting of PCA and quadratic SFA, so that it uses two cores on a modern CPU:

```
>>> node1 = mdp.nodes.PCANode(input_dim=100, output_dim=10)
>>> node2 = mdp.nodes.SFA2Node(input_dim=10, output_dim=10)
>>> flow = node1 + node2
>>> data_iterable = mdp.numx_rand.random((6, 200, 100))
>>> scheduler = parallel.ProcessScheduler(n_processes=2)
>>> parallel_flow = parallel.make_parallel(flow)
>>> parallel_flow.train(data_iterable, scheduler=scheduler)
```

So only two additional lines were needed to parallelize the training of the flow. The `make_parallel` function tries to find parallel versions of the nodes in the given flow. It then modifies the nodes such that they make use of the parallel version. A new `ParallelFlow` is then constructed and returned. One can also reverse this with the function `unmake_parallel`.

We can also implement this example by directly constructing a parallel flow:

```
>>> node1 = mdp.parallel.ParallelPCANode(input_dim=100, output_dim=10)
>>> node2 = mdp.parallel.ParallelSFA2Node(input_dim=10, output_dim=10)
>>> parallel_flow = mdp.parallel.ParallelFlow([node1, node2])
>>> data_iterable = mdp.numx_rand.random((6, 200, 100))
>>> scheduler = parallel.ProcessScheduler(n_processes=2)
>>> parallel_flow.train(data_iterable, scheduler=scheduler)
```

This gives you more control over which parallel node classes are used. In the following sections we will go into more details. As long as you only want to use the already existing classes for parallelization you can actually skip large parts of the follwing sections.

Scheduler

A scheduler is an instance of one of the scheduler classes we provide. They are all derived from the `Scheduler` base class. Currently we provide the `Scheduler` base class (which does not do any real parallelization) and the `ProcessScheduler` class which can distributes the incoming tasks over mutltiple python processes (circumventing the global interpreter lock). There is also experimental support for the Parallel Python library in the `mdp.parallel.pp_support` package.

The first important method of the scheduler class is `add_task`. This method takes two arguments: `data` and `task_callable`, which can be a function or an object with a `__call__` method. The return value of the `task_callable` is the result of the task. If `task_callable` is None then the last provided `task_callable` will be used. This splitting of callable and data in principle makes it possible to implement caching of the `task_callable` in the scheduler (but so far none of our schedulers actually implement this feature).

After submitteing all the tasks with `add_task` you can then call the `get_results` method. This method returns all the task results, normally in a list. If there are open tasks in the scheduler `get_results` will wait until all the tasks are finished. You can also check the status of the scheduler by looking at the `n_open_tasks` attribute, which tells you the number of open tasks.

Internally an instance of the base class `mdp.parallel.ResultContainer` is used for the storage of the results in the scheduler. By providing your own result container to the scheduler you modify the storage. For example the default result container is an instance of `OrderedResultContainer`

Parallel Nodes

The parallel package introduces the new base class `ParallelNode` for nodes that support parallel training. Derived from this are parallel node versions like `ParallelPCANode` for the `PCANode`. Parallel execution on the other hand works with any normal node as well (only the `copy` method of the node is needed for this).

If you want to write your own parallel nodes you have to derive your node from `ParallelNode`. `ParallelNode` has the new template methods `fork` and `join`. `fork` should return a new parallel node instance. This new instance can then be trained somewhere else (e.g. in a different process) with the usual `train` method. Afterwards one calls `join` on the original node, with the forked node as the argument. This is effectively the same as calling `train` directly on the original node.

In your own parallel nodes you should only overwrite the `_fork` and `_join` methods, which are automatically called by `fork` and `join`. The `fork` and `join` take care of the standard node attritbutes like the dimensions. You should also look at the source code of a parallel node like `ParallelPCANode` to get a better idea about how to write parallel nodes.

Currently we provide the following parallel nodes: `ParallelPCANode`, `ParallelWhiteningNode`, `ParallelSFANode`, `ParallelSFA2Node`, `ParallelFlowNode`, `ParallelLayer`, `ParallelCloneLayer` (the last three are from the `hinet` package).

Parallel Flows

As shown earlier in the basic example a parallel flow implements the parallel training (and execution) using a provided scheduler. The scheduler is simplyprovided as an additional argumet for the train or execute method of the parallel flow. If no scheduler is provided the node behaves just like a normal flow.

You can also do the parallel training in a more custom way by manually fetching tasks and assigning them to a scheduler. However, this should rarely be required.

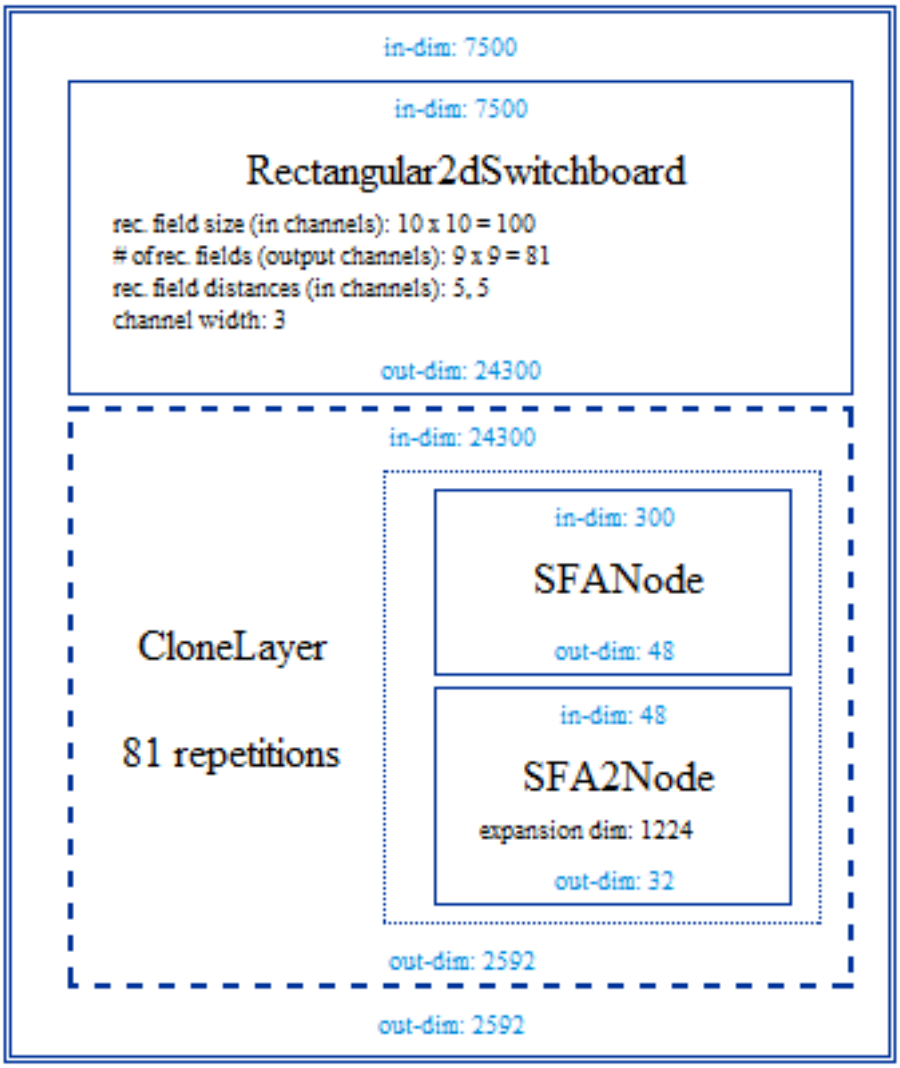
Example application (2-D image data)

As promised we now present a more complicated example. We define the lowest layer for some kind of image processing system. The input data is assumed to consist of image sequences, with each image having a size of 50 by 50 pixels. We take color images, so after converting the images to one dimensional numpy arrays each pixel corresponds to three numeric values in the array, which the values just next to each other (one for each color channel).

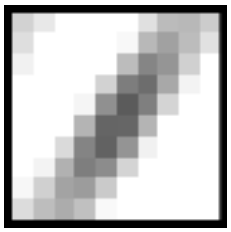
The processing layer consists of many parallel units, which only see a small image region with a size of 10 by 10 pixels. These so called receptive fields cover the whole image and have an overlap of five pixels. Note that the image data is represented as an 1-D array. Therefore we need the `Rectangular2dSwitchboard` class to correctly route the data for each receptive field to the corresponding unit in the following layer. We also call the switchboard output for a single receptive field an output channel and the three RGB values for a single pixel form an input channel. Each processing unit is a flow consisting of an `SFANode` (to somewhat reduce the dimensionality) that is followed by an `SFA2Node`. Since we assume that the statistics are similar in each receptive filed we actually use the same nodes for each receptive field. Therefore we use a `CloneLayer` instead of the standard `Layer`. Here is the actual code:

```
>>> switchboard = mdp.hinet.Rectangular2dSwitchboard(x_in_channels=50,
...                                                   y_in_channels=50,
...                                                   x_field_channels=10,
...                                                   y_field_channels=10,
...                                                   x_field_spacing=5,
...                                                   y_field_spacing=5,
...                                                   in_channel_dim=3)
>>> sfa_dim = 48
>>> sfa_node = mdp.nodes.SFANode(input_dim=switchboard.out_channel_dim,
...                              output_dim=sfa_dim)
>>> sfa2_dim = 32
>>> sfa2_node = mdp.nodes.SFA2Node(input_dim=sfa_dim,
...                                output_dim=sfa2_dim)
>>> flownode = mdp.hinet.FlowNode(mdp.Flow([sfa_node, sfa2_node]))
>>> sfa_layer = mdp.hinet.CloneLayer(flownode,
...                                  n_nodes=switchboard.output_channels)
>>> flow = mdp.Flow([switchboard, sfa_layer])
```

The HTML representation of the the constructed flow looks like this in your browser:



Now one can train this flow for example with image sequences from a movie. After the training phase one can compute the image pattern that produces the highest response in a given output coordinate (use `mdp.utils.QuadraticForm`). One such optimal image pattern may look like this (only a grayscale version is shown):



So the network units have developed some kind of primitive line detector. More on this topic can be found in: Berkes, P. and Wiskott, L., *Slow feature analysis yields a rich repertoire of complex cell properties*. Journal of Vision, 5(6):579-602.

One could also add more layers on top of this first layer to do more complicated stuff. Note that the `in_channel_dim` in the next `Rectangular2dSwitchboard` would be 32, since this is the output dimension of one unit in the `CloneLayer` (instead of 3 in the first switchboard, corresponding to the three RGB colors).

Real life examples

Logistic maps

In this section we show a complete example of MDP usage in a machine learning application, and use non-linear Slow Feature Analysis for processing of non-stationary time series. We consider a chaotic time series derived by a logistic map (a demographic model of the population biomass of species in the presence of limiting factors such as food supply or disease) that is non-stationary in the sense that the underlying parameter is not fixed but is varying smoothly in time.

The goal is to extract the slowly varying parameter that is hidden in the observed time series. This example reproduces some of the results reported in Laurenz Wiskott, *Estimating Driving Forces of Nonstationary Time Series with Slow Feature Analysis*. arXiv.org e-Print archive.

Generate the slowly varying driving force, a combination of three sine waves (freqs: 5, 11, 13 Hz), and define a function to generate the logistic map

```
>>> p2 = mdp.numx.pi*2
>>> t = mdp.numx.linspace(0,1,10000,endpoint=0) # time axis 1s, sampler-
ate 10KHz
>>> dforce = mdp.numx.sin(p2*5*t) + mdp.numx.sin(p2*11*t) + mdp.numx.sin(p2*13*t)
>>> def logistic_map(x,r):
...     return r*x*(1-x)
...
>>>
```

Note that we define `series` to be a two-dimensional array. Inputs to MDP must be two-dimensional arrays with variables on columns and observations on rows. In this case we have only one variable:

```
>>> series = mdp.numx.zeros((10000,1),'d')
```

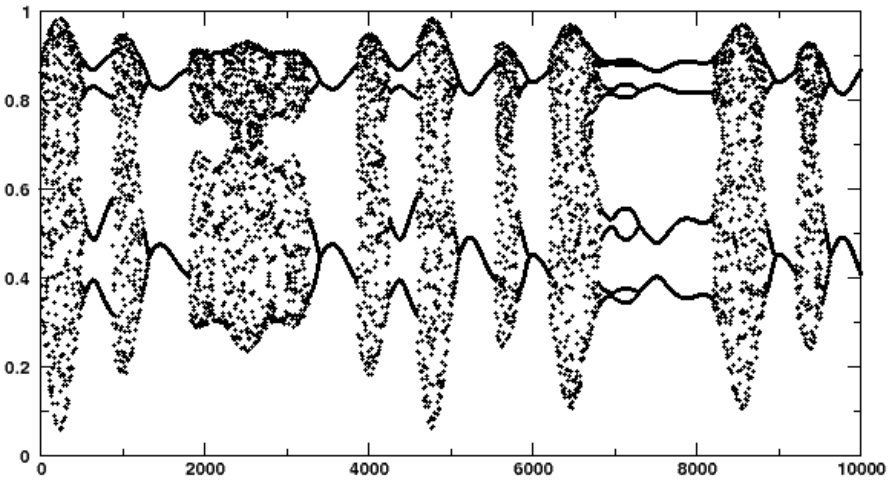
Fix the initial condition:

```
>>> series[0] = 0.6
```

Generate the time series using the logistic equation. The driving force modifies the logistic equation parameter `r`:

```
>>> for i in range(1,10000):
...     series[i] = logistic_map(series[i-1],3.6+0.13*dforce[i])
...
>>>
```

If you have a plotting package `series` should look like this:



To reconstruct the underlying parameter, we define a `Flow` to perform SFA in the space of polynomials of degree 3. We first use a node that embeds the 1-dimensional time series in a 10 dimensional space using a sliding temporal window of size 10 (`TimeFramesNode(10)`). Second, we expand the signal in the space of polynomials of degree 3 using a `PolynomialExpansionNode(3)`. Finally, we perform SFA on the expanded signal and keep the slowest feature using the `SFANode(output_dim=1)`.

In order to measure the slowness of the input time series before and after processing, we put at the beginning and at the end of the node sequence a node that computes the eta-value (a measure of slowness) of its input (`EtaComputerNode()`):

```
>>> flow = (mdp.nodes.EtaComputerNode() +
...         mdp.nodes.TimeFramesNode(10) +
...         mdp.nodes.PolynomialExpansionNode(3) +
...         mdp.nodes.SFANode(output_dim=1) +
...         mdp.nodes.EtaComputerNode() )
...
>>>
```

Since the time series is short enough to be kept in memory we don't need to define generators and we can feed the flow directly with the whole signal:

```
>>> flow.train(series)
```

Since the second and the third nodes are not trainable we are going to get two warnings (`Training Interrupted`). We can safely ignore them. Execute the flow to get the slow feature

```
>>> slow = flow(series)
```

The slow feature should match the driving force up to a scaling factor, a constant offset and the sign. To allow a comparison we rescale the driving force to have zero mean and unit variance:

```
>>> resc_dforce = (dforce - mdp.numx.mean(dforce,0))/mdp.numx.std(dforce,0)
```

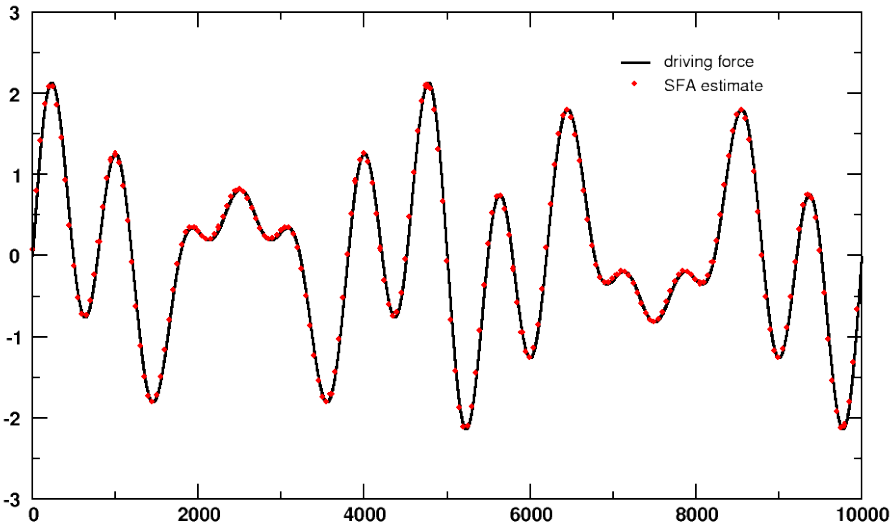
Print covariance between the rescaled driving force and the slow feature. Note that embedding the time series with 10 time frames leads to a time series with 9 observations less:

```
>>> mdp.utils.cov2(resc_dforce[:-9],slow)
0.99992501533859179
```

Print the *eta-values* of the chaotic time series and of the slow feature

```
>>> print 'Eta value (time series): ', flow[0].get_eta(t=10000)
Eta value (time series):  [ 3002.53380245]
>>> print 'Eta value (slow feature): ', flow[-1].get_eta(t=9996)
Eta value (slow feature):  [ 10.2185087]
```

If you have a plotting package you could plot the real driving force is plotted together with the driving force estimated by SFA and see that they match perfectly:



Growing neural gas

We generate uniformly distributed random data points confined on different 2-D geometrical objects. The Growing Neural Gas Node builds a graph with the same topological structure.

Fix the random seed to obtain reproducible results:

```
>>> mdp.numx_rand.seed(1266090063)
```

Some functions to generate uniform probability distributions on different geometrical objects:

```
>>> def uniform(min_, max_, dims):
...     """Return a random number between min_ and max_ ."""
...     return mdp.numx_rand.random(dims)*(max_-min_)+min_
...
>>> def circumference_distr(center, radius, n):
```

```
...     """Return n random points uniformly distributed on a circumfer-
ence."""
...     phi = uniform(0, 2*mdp.numx.pi, (n,1))
...     x = radius*mdp.numx.cos(phi)+center[0]
...     y = radius*mdp.numx.sin(phi)+center[1]
...     return mdp.numx.concatenate((x,y), axis=1)
...
>>> def circle_distr(center, radius, n):
...     """Return n random points uniformly distributed on a circle."""
...     phi = uniform(0, 2*mdp.numx.pi, (n,1))
...     sqrt_r = mdp.numx.sqrt(uniform(0, radius*radius, (n,1)))
...     x = sqrt_r*mdp.numx.cos(phi)+center[0]
...     y = sqrt_r*mdp.numx.sin(phi)+center[1]
...     return mdp.numx.concatenate((x,y), axis=1)
...
>>> def rectangle_distr(center, w, h, n):
...     """Return n random points uniformly distributed on a rectangle."""
...     x = uniform(-w/2., w/2., (n,1))+center[0]
...     y = uniform(-h/2., h/2., (n,1))+center[1]
...     return mdp.numx.concatenate((x,y), axis=1)
...
>>> N = 2000
```

Explicitly collect random points from some distributions:

- Circumferences:

```
>>> cf1 = circumference_distr([6,-0.5], 2, N)
>>> cf2 = circumference_distr([3,-2], 0.3, N)
```

- Circles:

```
>>> cl1 = circle_distr([-5,3], 0.5, N/2)
>>> cl2 = circle_distr([3.5,2.5], 0.7, N)
```

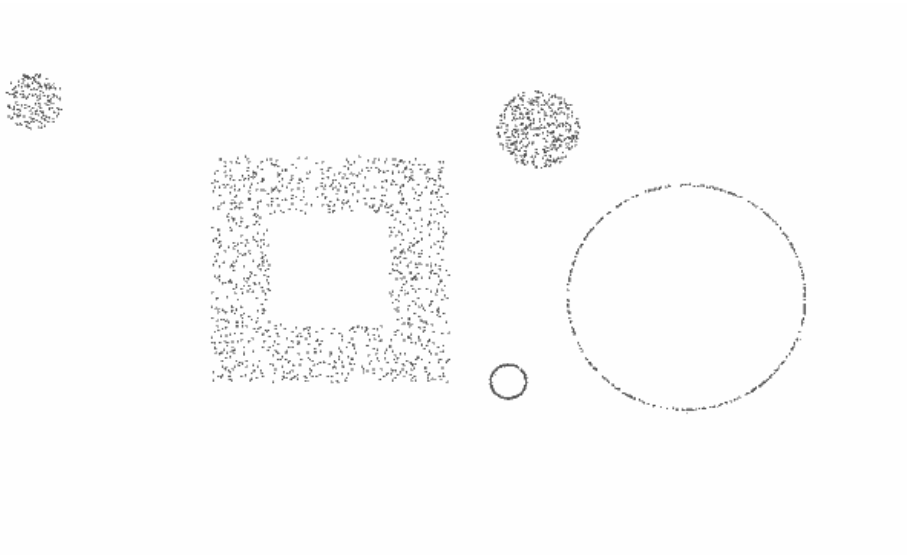
- Rectangles:

```
>>> r1 = rectangle_distr([-1.5,0], 1, 4, N)
>>> r2 = rectangle_distr([+1.5,0], 1, 4, N)
>>> r3 = rectangle_distr([0,+1.5], 2, 1, N/2)
>>> r4 = rectangle_distr([0,-1.5], 2, 1, N/2)
```

Shuffle the points to make the statistics stationary

```
>>> x = mdp.numx.concatenate([cf1, cf2, cl1, cl2, r1,r2,r3,r4], axis=0)
>>> x = mdp.numx.take(x,mdp.numx_rand.permutation(x.shape[0]), axis=0)
```

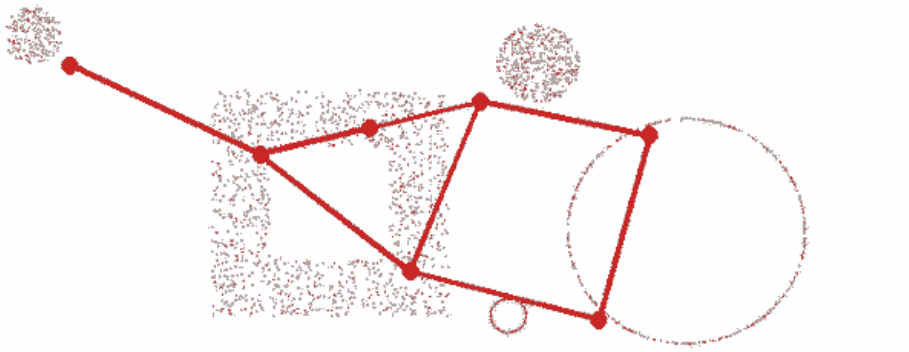
If you have a plotting package `x` should look like this:



Create a `GrowingNeuralGasNode` and train it:

```
>>> gng = mdp.nodes.GrowingNeuralGasNode(max_nodes=75)
```

The initial distribution of nodes is randomly chosen:

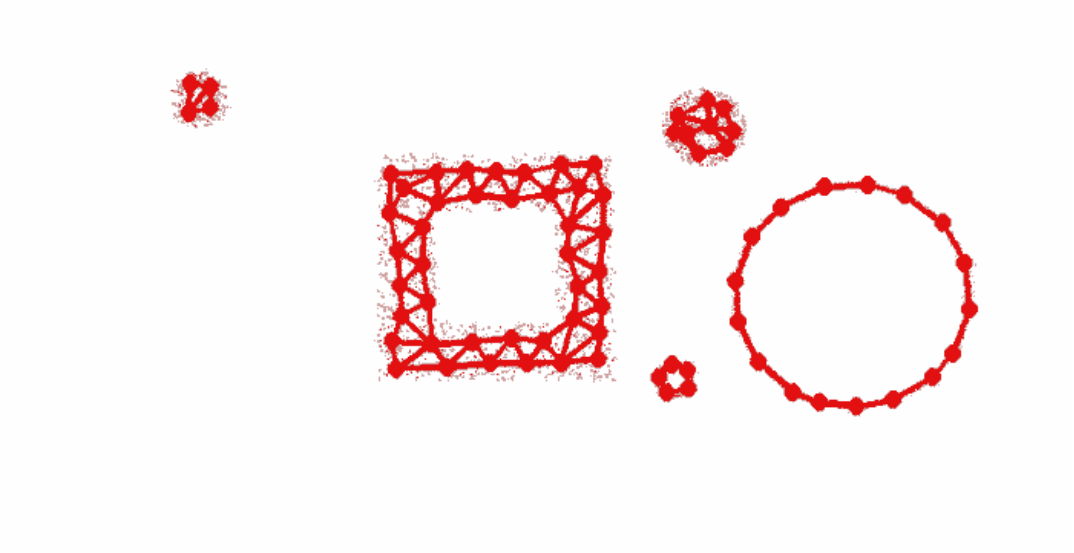


The training is performed in small chunks in order to visualize the evolution of the graph:

```
>>> STEP = 500
>>> for i in range(0,x.shape[0],STEP):
...     gng.train(x[i:i+STEP])
...     # [...] plotting instructions
...
>>> gng.stop_training()
```

See here the animation of training.

Visualizing the neural gas network, we'll see that it is adapted to the topological structure of the data distribution:

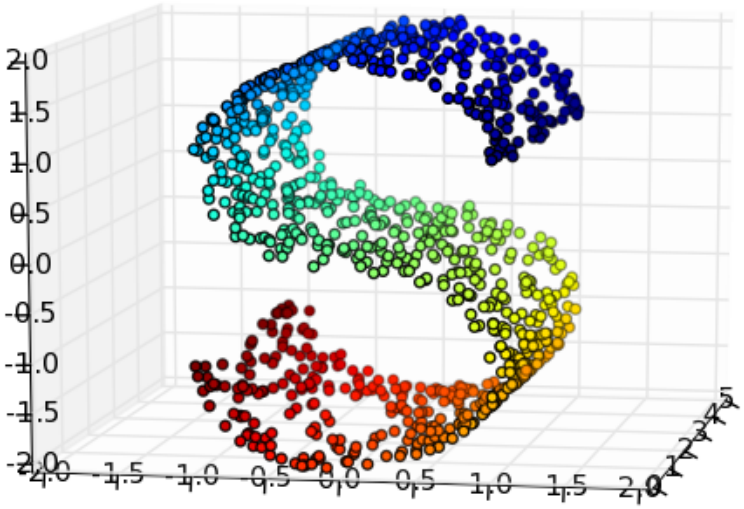


Calculate the number of connected components:

```
>>> n_obj = len(gng.graph.connected_components())
>>> print n_obj
5
```

Locally linear embedding

Locally linear embedding (LLE) approximates the input data with a low-dimensional surface and reduces its dimensionality by learning a mapping to the surface. Here we consider data generated randomly on an S-shaped 2D surface embedded in a 3D space:



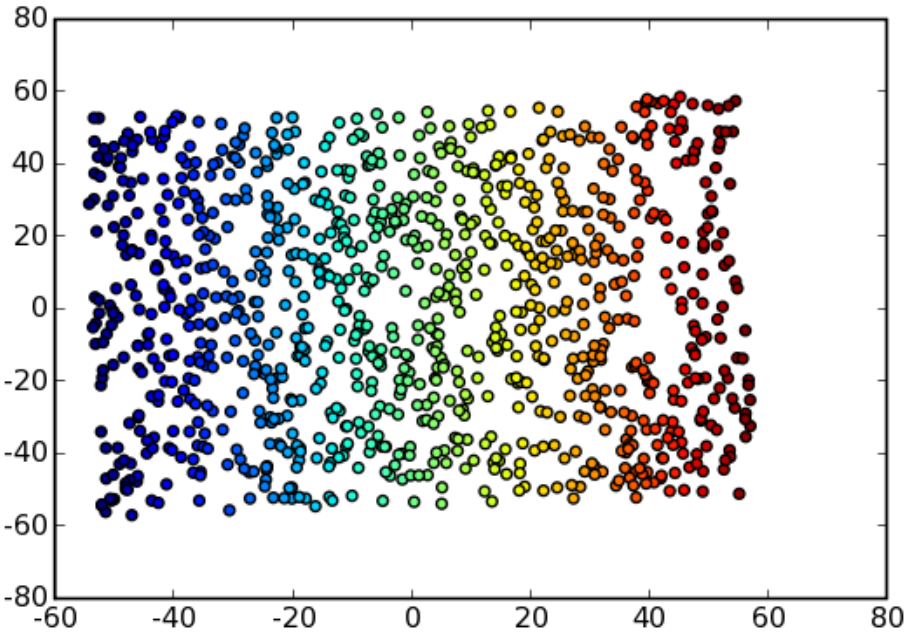
The surface is defined by the function

```
>>> def s_distr(npoints, hole=False):
...     """Return a 3D S-shaped surface. If hole is True, the surface has
...     a hole in the middle."""
...     t = mdp.numx_rand.random(npoints)
...     y = mdp.numx_rand.random(npoints)*5.
...     theta = 3.*mdp.numx.pi*(t-0.5)
...     x = mdp.numx.sin(theta)
...     z = mdp.numx.sign(theta)*(mdp.numx.cos(theta) - 1.)
...     if hole:
...         in-
dices = mdp.numx.where(((0.3>t) | (0.7<t)) | ((1.>y) | (4.<y)))
...         return x[indices], y[indices], z[indices], t[indices]
...     else:
...         return x, y, z, t
```

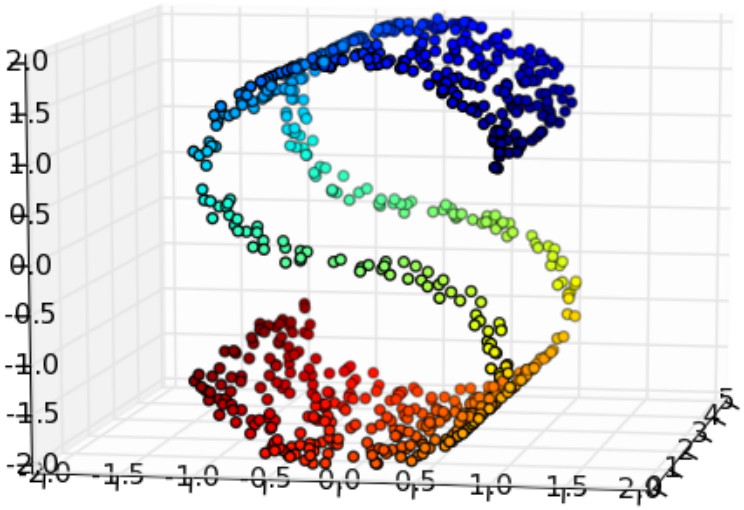
We generate 1000 points on this surface, then define an LLENode with parameters k=15 (number of neighbors) and output_dim=2 (the number of dimensions of the reduced representation), then train and execute the node to obtain the projected data:

```
>>> n, k = 1000, 15
>>> x, y, z, t = s_distr(n, hole=False)
>>> data = mdp.numx.array([x,y,z]).T
>>> lle_projected_data = mdp.nodes.LLENode(k, output_dim=2)(data)
```

The projected data forms a nice parameteric representation of the S-shaped surface:



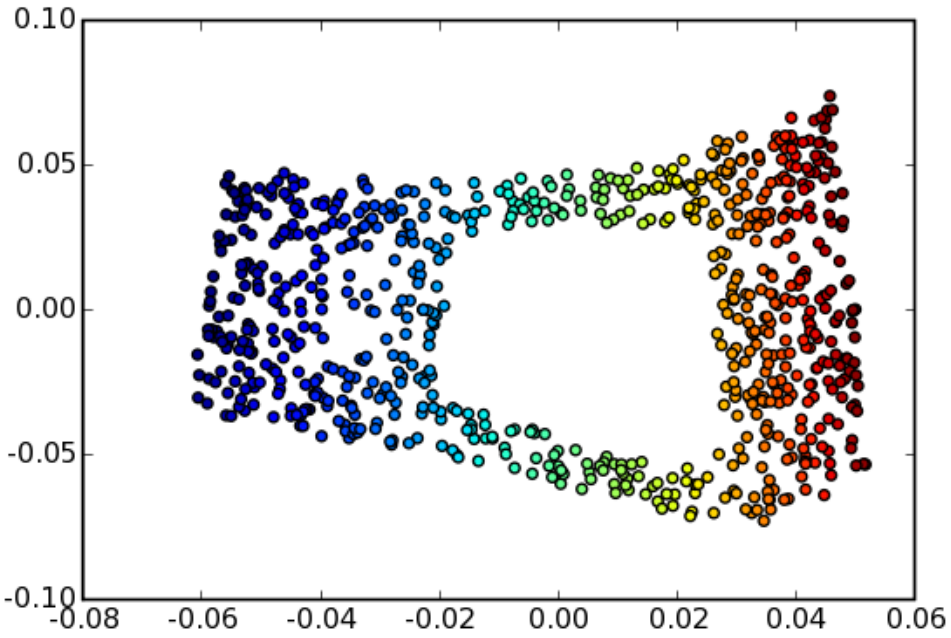
The problem becomes more difficult if the surface has a hole in the middle:



In this case, the LLE algorithm has some difficulty finding the correct representation. The lines

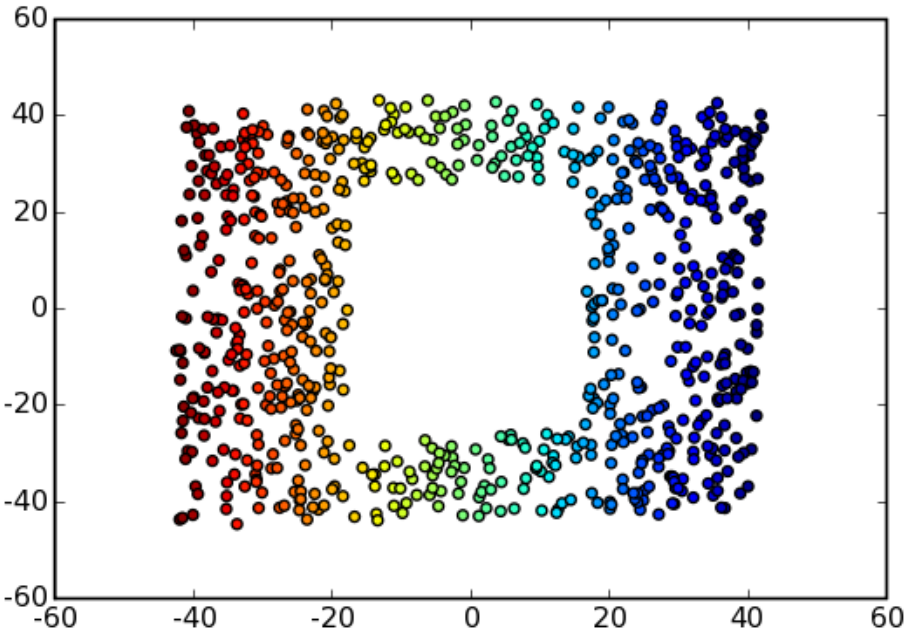
```
>>> x, y, z, t = s_distr(n, hole=True)
>>> data = mdp.numx.array([x,y,z]).T
>>> lle_projected_data = mdp.nodes.LLENode(k, output_dim=2)(data)
```

return a distorted mapping:



The Hessian LLE Node takes the local curvature of the surface into account, and is able to find a better representation:

```
>>> hlle_projected_data = mdp.nodes.HLLENode(k, output_dim=2)(data)
```



Node List

Here is the complete list of implemented nodes. Refer to the API for the full documentation and interface description.

- **CuBICANode** Perform Independent Component Analysis using the CuBICA algorithm.

  Reference: Blaschke, T. and Wiskott, L. (2003). *CuBICA: Independent Component Analysis by Simultaneous Third- and Fourth-Order Cumulant Diagonalization.* IEEE Transactions on Signal Processing, 52(5), pp. 1250-1256. More information about ICA can be found among others in Hyvarinen A., Karhunen J., Oja E. (2001). *Independent Component Analysis*, Wiley.

- **EtaComputerNode** Compute the eta values of the normalized training data. The delta value of a signal is a measure of its temporal variation, and is defined as the mean of the derivative squared, i.e. `delta(x) = mean(dx/dt(t)^2)`. `delta(x)` is zero if 'x' is a constant signal, and increases if the temporal variation of the signal is bigger. The eta value is a more intuitive measure of temporal variation, defined as `eta(x) = T/(2*pi) * sqrt(delta(x))`. If 'x' is a signal of length 'T' which consists of a sine function that accomplishes exactly 'N' oscillations, then `eta(x) = N`.

  Reference: Wiskott, L. and Sejnowski, T.J. (2002). *Slow Feature Analysis: Unsupervised Learning of Invariances*, Neural Computation, 14(4):715-770.

- **FANode** Perform Factor Analysis. The current implementation should be most efficient for long data sets: the sufficient statistics are collected in the training phase, and all EM-cycles are performed at its end. More information about Factor Analysis can be found in Max Welling's classnotes in the chapter "Linear Models".

- **FastICANode** Perform Independent Component Analysis using the FastICA algorithm.

  Reference: Aapo Hyvarinen (1999). *Fast and Robust Fixed-Point Algorithms for Independent Component Analysis*, IEEE Transactions on Neural Networks, 10(3):626-634. More information about ICA can be found among others in Hyvarinen A., Karhunen J., Oja E. (2001). *Independent Component Analysis*, Wiley.

- **FDANode** Perform a (generalized) Fisher Discriminant Analysis of its input. It is a supervised node that implements FDA using a generalized eigenvalue approach.

  More information on Fisher Discriminant Analysis can be found for example in C. Bishop, *Neural Networks for Pattern Recognition*, Oxford Press, pp. 105-112.

- **GaussianClassifierNode** Perform a supervised Gaussian classification. Given a set of labelled data, the node fits a gaussian distribution to each class.

- **GrowingNeuralGasNode** Learn the topological structure of the input data by building a corresponding graph approximation.

  More information about the Growing Neural Gas algorithm can be found in B. Fritzke, *A Growing Neural Gas Network Learns Topologies*, in G. Tesauro, D. S. Touretzky, and T. K. Leen (editors), *Advances in Neural Information Processing Systems 7*, pages 625-632. MIT Press, Cambridge MA, 1995.

- **HitParadeNode** Collect the first 'n' local maxima and minima of the training signal which are separated by a minimum gap 'd'.

- **HLLENode** Original code contributed by Jake VanderPlas.

  Perform a Hessian Locally Linear Embedding analysis on the data.

  Implementation based on algorithm outlined in *Hessian Eigenmaps: new locally linear embedding techniques for high-dimensional data* by C. Grimes and D. Donoho, 2003

- **ISFANode** Perform Independent Slow Feature Analysis on the input data.

  More information about ISFA can be found in: Blaschke, T. , Zito, T., and Wiskott, L. *Independent Slow Feature Analysis and Nonlinear Blind Source Separation.* Neural Computation 19(4):994-1021 (2007).

- **JADENode** Original code contributed by Gabriel Beckers.

  Perform Independent Component Analysis using the JADE algorithm.

  References: Cardoso, J.-F, and Souloumiac, A. *Blind beamforming for non Gaussian signals.* Radar and Signal Processing, IEE Proceedings F, 140(6): 362-370 (1993), and Cardoso, J.-F. *High-order contrasts for independent component analysis.* Neural Computation, 11(1): 157-192 (1999). More information about ICA can be found among others in Hyvarinen A., Karhunen J., Oja E. (2001). *Independent Component Analysis*, Wiley.

- **LLENode** Original code contributed by Jake VanderPlas.

  Perform a Locally Linear Embedding analysis on the data.

  Based on the algorithm outlined in *An Introduction to Locally Linear Embedding* by L. Saul and S. Roweis, using improvements suggested in *Locally Linear Embedding for Classification* by D. deRidder and R.P.W. Duin.

- **NIPALSNode** Original code contributed by Michael Schmuker, Susanne Lezius, and Farzad Farkhooi.

  Perform Principal Component Analysis using the NIPALS algorithm. This algorithm is particularyl useful if you have more variable than observations, or in general when the number of variables is huge and calculating a full covariance matrix may be unfeasable. It's also more efficient of the standard PCANode if you expect the number of significant principal

components to be a small. In this case setting output_dim to be a certain fraction of the total variance, say 90%, may be of some help.

Reference for NIPALS (Nonlinear Iterative Partial Least Squares): Wold, H. *Nonlinear estimation by iterative least squares procedures.* in David, F. (Editor), Research Papers in Statistics, Wiley, New York, pp 411-444 (1966).

More information about Principal Component Analysis, a.k.a. discrete Karhunen-Loeve transform can be found among others in I.T. Jolliffe, *Principal Component Analysis*, Springer-Verlag (1986).

- **NoiseNode** Original code contributed by Mathias Franzius.

  Inject multiplicative or additive noise into the input data.

- **PCANode** Filter the input data throug the most significatives of its principal components.

  More information about Principal Component Analysis, a.k.a. discrete Karhunen-Loeve transform can be found among others in I.T. Jolliffe, *Principal Component Analysis*, Springer-Verlag (1986).

- **PolynomialExpansionNode** Perform expansion in a polynomial space.

- **QuadraticExpansionNode** Perform expansion in the space formed by all linear and quadratic monomials

- **RBMNode** Implementation of a Restricted Boltzmann Machine.

  For more information on RBMs, see Geoffrey E. Hinton (2007) Boltzmann machine. Scholarpedia, 2(5):1668

- **RBMWithLabelsNode** Implementation of a Restricted Boltzmann Machine with softmax labels.

  For more information on RBMs, see Geoffrey E. Hinton (2007) Boltzmann machine Scholarpedia, 2(5):1668

  Hinton, G. E, Osindero, S., and Teh, Y. W. *A fast learning algorithm for deep belief nets*, Neural Computation, 18:1527-1554 (2006).

- **SFANode** Extract the slowly varying components from the input data.

  More information about Slow Feature Analysis can be found in Wiskott, L. and Sejnowski, T.J., *Slow Feature Analysis: Unsupervised Learning of Invariances*, Neural Computation, 14(4):715-770 (2002).

- **SFA2Node** Get an input signal, expand it in the space of inhomogeneous polynomials of degree 2 and extract its slowly varying components. The `get_quadratic_form` method returns the input-output function of one of the learned unit as a `mdp.utils.QuadraticForm` object.

  More information about Slow Feature Analysis can be found in Wiskott, L. and Sejnowski, T.J., *Slow Feature Analysis: Unsupervised Learning of Invariances*, Neural Computation, 14(4):715-770 (2002).

- **TDSEPNode** Perform Independent Component Analysis using the TDSEP algorithm. Note that TDSEP, as implemented in this Node, is an online algorithm, i.e. it is suited to be trained on huge data sets, provided that the training is done sending small chunks of data for each time.

  Reference: Ziehe, Andreas and Muller, Klaus-Robert (1998). *TDSEP an efficient algorithm for blind separation using time structure.* in Niklasson, L, Boden, M, and Ziemke, T (Editors), Proc. 8th Int. Conf. Artificial Neural Networks (ICANN 1998).

- **TimeFramesNode** Copy delayed version of the input signal on the space dimensions.

```
For example, for time_frames=3 and gap=2:

[ X(1) Y(1)          [ X(1) Y(1) X(3) Y(3) X(5) Y(5)
  X(2) Y(2)            X(2) Y(2) X(4) Y(4) X(6) Y(6)
  X(3) Y(3)   -->      X(3) Y(3) X(5) Y(5) X(7) Y(7)
  X(4) Y(4)            X(4) Y(4) X(6) Y(6) X(8) Y(8)
  X(5) Y(5)            ...  ...  ...  ...  ...  ... ]
  X(6) Y(6)
  X(7) Y(7)
  X(8) Y(8)
  ...  ...  ]
```

- **WhiteningNode** 'Whiten' the input data by filtering it through the most significatives of its principal components. All output signals have zero mean, unit variance and are decorrelated.

Didn't you find what you were looking for?

If you want to contribute some code or a new algorithm, please do not hesitate to submit it!

Additional utilities

MDP offers some additional utilities of general interest in the `mdp.utils` module. Refer to the API for the full documentation and interface description.

CovarianceMatrix This class stores an empirical covariance matrix that can be updated incrementally. A call to the `fix` method returns the current state of the covariance matrix, the average and the number of observations, and resets the internal data.

Note that the internal sum is a standard `__add__` operation. We are not using any of the fancy sum algorithms to avoid round off errors when adding many numbers. If you want to contribute a `CovarianceMatrix` class that uses such algorithms we would be happy to include it in MDP. For a start see the Python recipe by Raymond Hettinger. For a review about floating point arithmetic and its pitfalls see this interesting article.

DelayCovarianceMatrix This class stores an empirical covariance matrix between the signal and time delayed signal that can be updated incrementally.

MultipleCovarianceMatrices Container class for multiple covariance matrices to easily execute operations on all matrices at the same time.

dig_node(node) Crawl recursively an MDP `Node` looking for arrays. Return (dictionary, string), where the dictionary is: { attribute_name: (size_in_bytes, array_reference)} and string is a nice string representation of it.

get_node_size(node) Get 'node' total byte-size using `cPickle` with protocol=2. (The byte-size is related the memory needed by the node).

progressinfo(sequence, length, style, custom) A fully configurable text-mode progress info box tailored to the command-line die-hards. To get a progress info box for your loops use it like this:

```
>>> for i in progressinfo(sequence):
...     do_something(i)
```

You can also use it with generators, files or any other iterable object, but in this case you have to specify the total length of the sequence:

```
>>> for line in progressinfo(open_file, nlines):
...     do_something(line)
```

A few examples of the available layouts:

```
[==================================73%=============>.................]

Progress:  67%[====================================>                  ]

23% [02:01:28] - [00:12:37]
```

QuadraticForm Define an inhomogeneous quadratic form as `1/2 x'Hx + f'x + c`. This class implements the quadratic form analysis methods presented in: Berkes, P. and Wiskott, L. On the analysis and interpretation of inhomogeneous quadratic forms as receptive fields. *Neural Computation*, 18(8): 1868-1895. (2006).

refcast(array, dtype) Cast the array to 'dtype' only if necessary, otherwise return a reference.

rotate(mat, angle, columns, units) Rotate in-place a NxM data matrix in the plane defined by the 'columns' when observation are stored on rows. Observations are rotated counterclockwise. This corresponds to the following matrix-multiplication for each data-point (unchanged elements omitted):

```
[  cos(angle) -sin(angle)     [ x_i ]
   sin(angle)  cos(angle) ] * [ x_j ]
```

random_rot(dim, dtype) Return a random rotation matrix, drawn from the Haar distribution (the only uniform distribution on SO(n)). The algorithm is described in the paper Stewart, G.W., *The efficient generation of random orthogonal matrices with an application to condition estimators*, SIAM Journal on Numerical Analysis, 17(3), pp. 403-409, 1980. For more information see this Wikipedia entry.

symrand(dim_or_eigv, dtype) Return a random symmetric (Hermitian) matrix with eigenvalues uniformly distributed on (0,1].

Graph module

MDP contains `mdp.graph`, a lightweight package to handle directed graphs.

Graph Represent a directed graph. This class contains several methods to create graph structures and manipulate them, among which

- `add_tree`**: Add a tree to the graph.** The tree is specified with a nested list of tuple, in a LISP-like notation. The values specified in the list become the values of the single nodes. Return an equivalent nested list with the nodes instead of the values.
  Example:

```
>>> a=b=c=d=e=None
>>> g.add_tree( (a, b, (c, d ,e)) )
# corresponds to this tree structure, with all node val-
ues set to None:

    a
   / \
  b   c
     / \
    d   e
```

- `topological_sort`: Perform a topological sort of the nodes.
- `dfs`, `undirected_dfs`: Perform Depth First sort.
- `bfs`, `undirected_bfs`: Perform Breadth First sort.
- `connected_components`**: Return a list of lists containing** the nodes of all connected components of the graph.
- `is_weakly_connected`: Return True if the graph is weakly connected.

GraphEdge Represent a graph edge and all information attached to it.

GraphNode Represent a graph node and all information attached to it.

recursive_map(func, seq) Apply a function recursively on a sequence and all subsequences.

recursive_reduce(func, seq, *argv) Apply `reduce(func, seq)` recursively to a sequence and all its subsequences.

Future Development

MDP is currently maintained by a core team of 3 developers, but it is open to user contributions. Users have already contributed some of the nodes, and more contributions are currently being reviewed for inclusion in future releases of the package. The package development can be followed on the public subversion code repository. Questions, bug reports, and feature requests are typically handled by the user mailing list

A new, large MDP package is currently under development that will extend MDP with more complex data flows, including back-propagation and loops. This framework will be integrated with both the `parallel` and the `hinet` packages to allow for large and complex data processing networks.

Finally, some more ToDos:

- Add more data processing algorithms.
- Advanced usage of the hinet package will be possible only in presence of an easy and intuitive GUI :)
- Wait for a good guy who wants to contribute a `CovarianceMatrix` class that uses some of the fancy sum algorithms to avoid round off errors when adding many numbers.

Contributors

In this final section we want to thank all users who have contributed code to the MDP project. Strictly in alphabetical order:

- Gabriel Beckers
- Farzad Farkhooi
- Mathias Franzius
- Michael Hanke
- Susanne Lezius
- Michael Schmuker
- Jake VanderPlas

Generated on: 2008-10-17 17:32 UTC. Generated by Docutils from reStructuredText source.