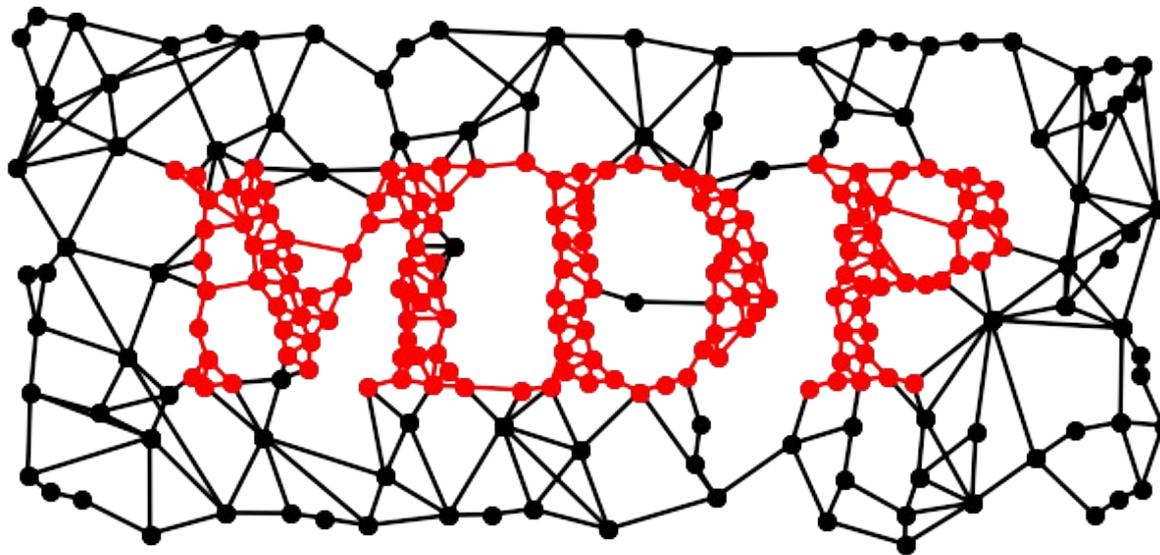


Modular toolkit for Data Processing

a Python data processing framework



Pietro Berkes, Niko Wilbert, Tiziano Zito



CNS 2009

Python in Neuroscience Workshop

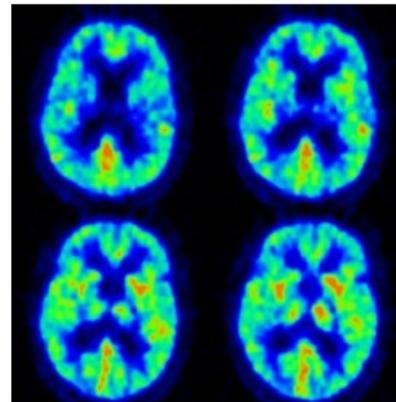
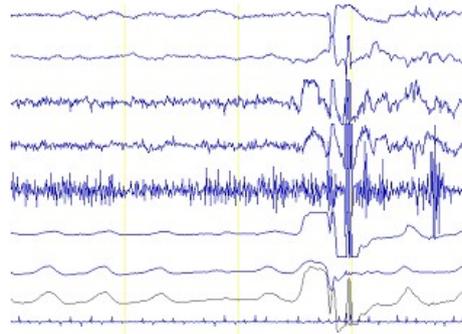
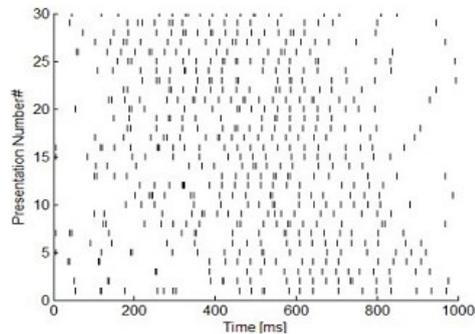
July 22th, 2009



<http://mdp-toolkit.sourceforge.net>

Background

- Python module, Object Oriented design
- Open Source (LGPL)
- first public release 2004
- 15k+ downloads, available in Debian, Ubuntu, MacPorts, Python(x,y)
- origin in Prof. Wiskott research group...



... but also used outside computational neuroscience

Contents

- **MDP basic building blocks:
nodes, flows, networks**
- **Extending MDP**
- **Parallelization**
- **Examples and design considerations**
- **Future development: the BiNet package**
- **Using and embedding MDP**
- **Future perspectives**

Building Blocks: Node

- **fundamental data processing element**
Node classes correspond to algorithms
- **API: train**
support for multiple phases, batch,
online, chunks, supervised, unsupervised
- **API: execute**
map n -dim input to m -dim output
- **API: inverse**
inverse of execute mapping
- **data format: 2-dim numpy arrays**
- **automatic consistency checks and conversions (dimensions, dtype, ...)**

Building Blocks: Node

- **example: Principal Component Analysis (PCA)**

```
>>> pca = PCANode(output_dim=0.9, dtype='float32')
>>> for x in data_stream:
...     pca.train(x)
>>> out = pca.execute(x)
>>> rec = pca.invert(out)
>>> pca.explained_variance
0.9012761929
>>> pca.output_dim
45
```

- **shortcut**

```
>>> proj_data = pca(x, output_dim=0.9, dtype='float32')
```

Building Blocks: Node

- **PCA (standard, NIPALS)**
- **ICA (FastICA, CuBICA, JADE, TDSEP, XSFA)**
- **Locally Linear Embedding**
- **Hessian Locally Linear Embedding**
- **Linear Regression**
- **Fisher Discriminant Analysis**
- **Slow Feature Analysis**
- **Independent Slow Feature Analysis**
- **Restricted Boltzmann Machine**
- **Growing Neural Gas**
- **Factor Analysis**
- **Gaussian Classifiers**
- **Polynomial Expansion**
- **Time Frames**
- **Hit Parades**
- **Noise**
- **...**

Building Blocks: Flow

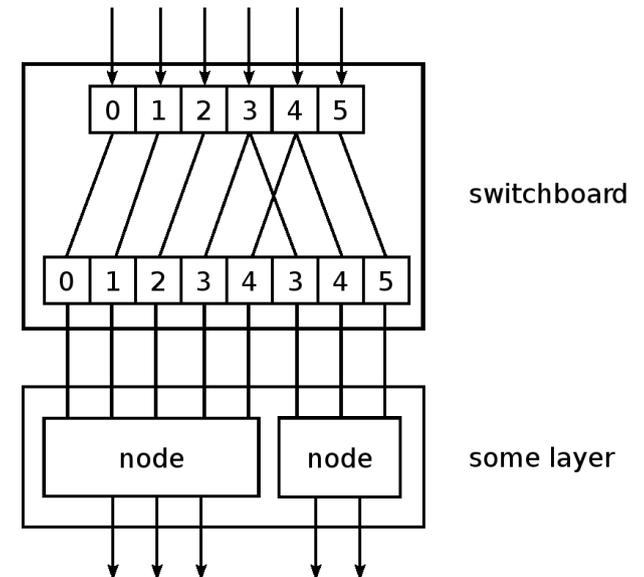
- combine nodes in a pipeline
- API: train, execute, inverse
- automatic training, execution, inversion
- automatic checks: dims and data formats
- Flow is a Python container (list)
- feed on arrays or iterators
- crash recovery, checkpoints

```
>>> flow = PCANode() + SFANode() + FastICANode()
>>> # DataStream is an iterable and returns
>>> # chunks of data (online or offline)
>>> flow.train([DataStream()]*3)
>>> out = flow.execute(x)
>>> rec = flow.inverse(out)
>>> flow += HitParadeNode()
```

Building Blocks: Network

- `mdp.hinet`: hierarchical networks
- `Layer`: combine nodes horizontally in parallel
- `Switchboard`: routing between layers
- `FlowNode`: encapsulate a Flow into a "super" Node
- everything is a Node: combine as you want
- all acyclic graphs can be emulated

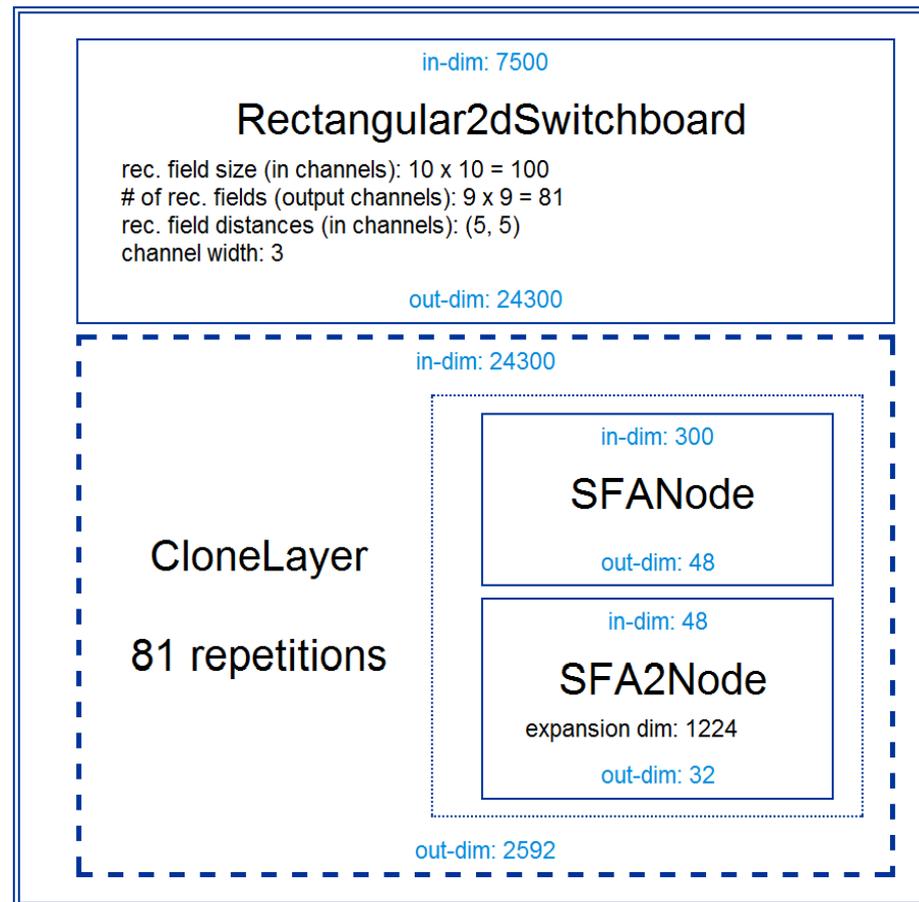
```
>>> layer = Layer([PCANode(),  
...               ICANode(), ...])  
>>> switch = Switchboard(input_dim=N,  
...                      connections=[0,3,1,...])  
>>> net = switch + layer  
>>> net.train(generator())  
>>> out = net.execute(x)
```



Building Blocks: Network

- **HTML visualization of your network**
- **Use it for debugging, reports or GUIs**

```
>>> mdp.hinet.show_flow(net)
```



Extending MDP: Writing Nodes

- concentrate on the algorithm, MDP takes care of the details
- use MDP utilities in your nodes
- immediately integrate your nodes with the existing library
- contribute your nodes to MDP!

```
>>> class MyNode(Node):
...     def _train(self, x):
...         ... training code ...
...     def _execute(self, x):
...         ... execution code ...
...     def _inverse(self, y):
...         ... inversion code ...
...
>>> flow = PCANode() + MyNode()
```

Parallelization

- for *embarrassingly parallel* problems
data chunks can be processed independently
- use multiple cores and multiple machines
Parallel Python support
- automatic parallelization of serial flows
- use abstract scheduler API
write your own adapter
- simple API: write your own parallel nodes
implement `_fork` and `_join` methods

```
>>> flow = PCANode() + SFANode()  
>>> scheduler = ProcessScheduler(n_processes=8)  
>>> pflow = make_flow_parallel(flow)  
>>> pflow.train(data, scheduler)
```

Real World Example

- **object recognition system**
working on 155x155 pixel image sequences
- **several GBs of training data for each training phase**
- **hierarchical network with nested nodes**
900 "super" nodes on lowest layer
- **training is distributed over network**



[Franzius, Wilbert, Wiskott, 2008]

Software Design Considerations

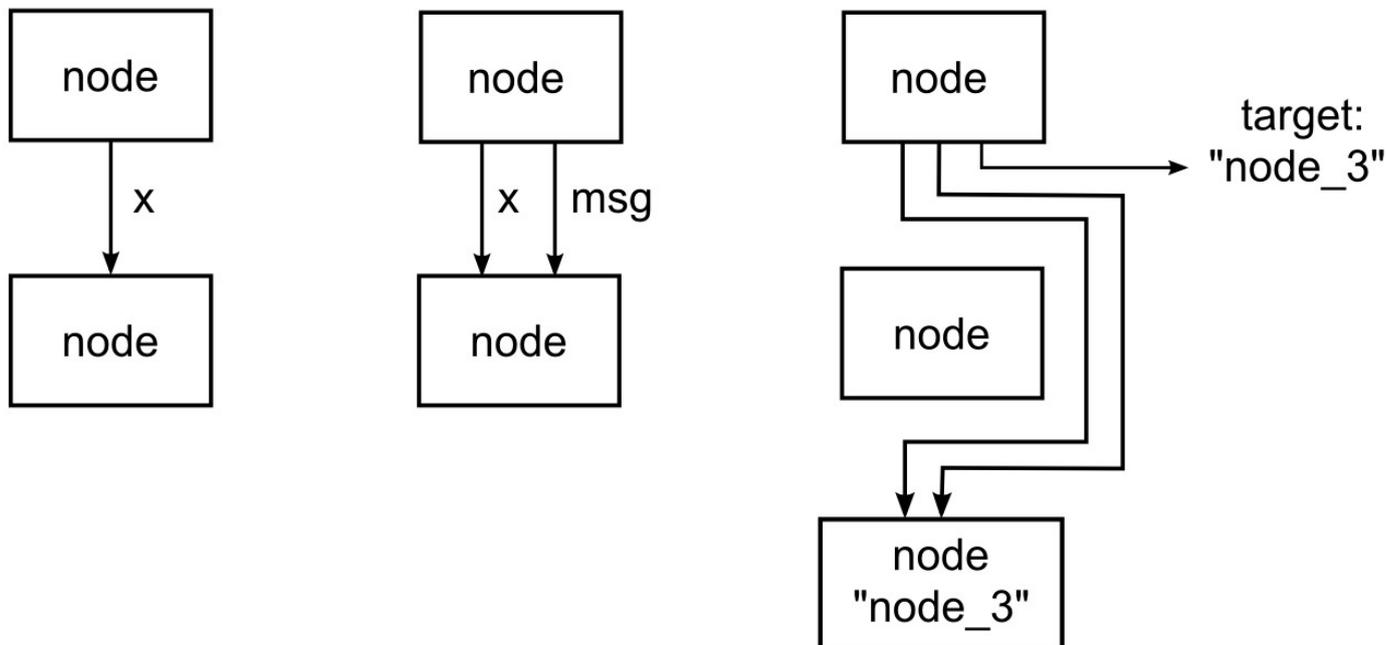
- **Networks in MDP are modular. Why?**
 - flexibility
 - distributed complexity = less complexity
 - extensions [e.g. parallelization] are independent from the specific structure of the network
 - higher but "once and for all" implementation cost
- **Comparison with monolithic approach:**
 - network structure is hard-coded
 - quick and dirty, not reusable
 - every extension must be hard-coded
 - network changes affect low-level code in different locations

Future Development: the BiNet Package

- `mdp.binnet` allows bi-directional data flow
feedback loops, error back-propagation, ...
- compatible with `mdp.parallel` and `mdp.hinet`
- HTML+JS inspector for debugging and
visualization
- scheduled for MDP 3.0 (end 2009?)

BiNet: Building Blocks

- **BiNode** and **BiFlow**: backward compatible
- **BiNode** has an ID string and can be accessed by this name: `biflow['PCA_node_3']`
- pass a "message" dictionary together with data
- **BiNode** can specify a target node [GOTO :)]



BiNet: Features

- message content addresses all or specific nodes:

```
>>> msg = {"status" : general_stuff,  
...       "fda_node=>labels" : fda_labels}
```

- message content accessed by argument names

```
>>> class BiFDANode(BiNode):  
...     def _train(self, x, labels):  
...         ... FDA training code ...
```

- several additional features, if you need it it's probably already there ;)

Using MDP

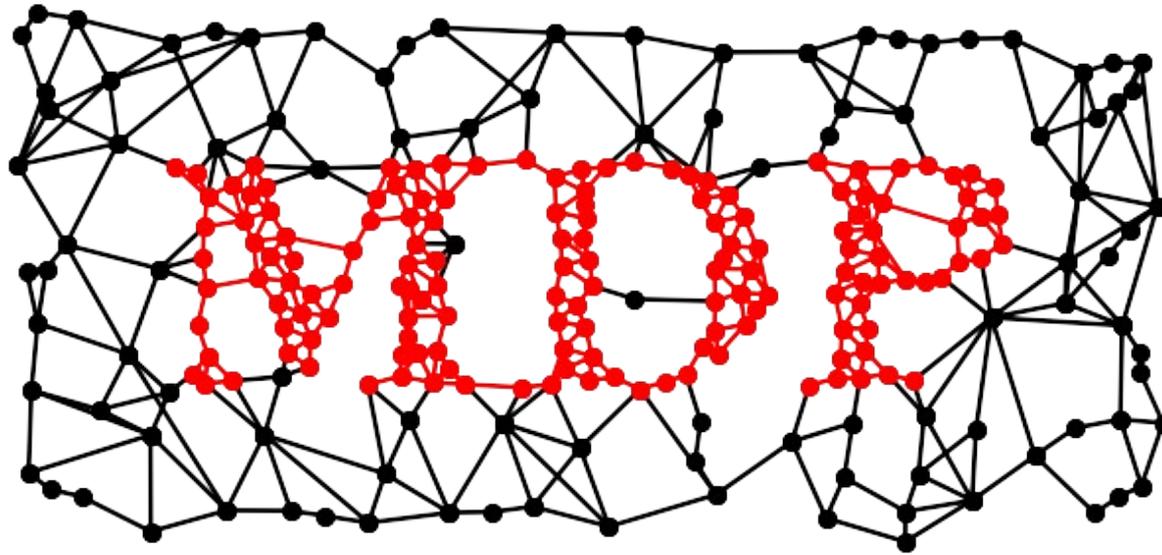
- **comprehensive documentation:**
 - tutorial covering basic and advanced usage
 - public objects have detailed doc-strings
 - PEP8 compliant, commented, and pylint-clean code
 - active and responsive user mailing list
- **collection of efficient and well tested (390+ unit tests) algorithms**
- **minimal dependencies: Python + NumPy**

Embedding MDP

- input and output just NumPy arrays
- API is stable and designed for straightforward embedding
- PyMCA: X-ray fluorescence mapping
- PyMVPA: ML framework for neuroimaging data analysis
- Chandler: personal organizer application

Future perspectives

- **Architecture:**
 - fully integrate, test, and document BiNet
 - Python 3
 - plugin system
 - GUI
 - automatic graph to BiNet translator
- **Algorithms:**
 - integrate widely used libraries (e.g. SVM)



- **Developers:**

- Pietro Berkes, Niko Wilbert, Tiziano Zito

- **Contributors:**

- Gabriel Beckers, Alberto Escalante, Farzad Farkhooi, Mathias Franzius, Yaroslav Halchenko, Michael Hanke, Konrad Hinsén, Samuel John, Susanne Lezius, Michael Schmuker, Henning Sprekeler, Jake VanderPlas