Bayesian networks with Mocapy++

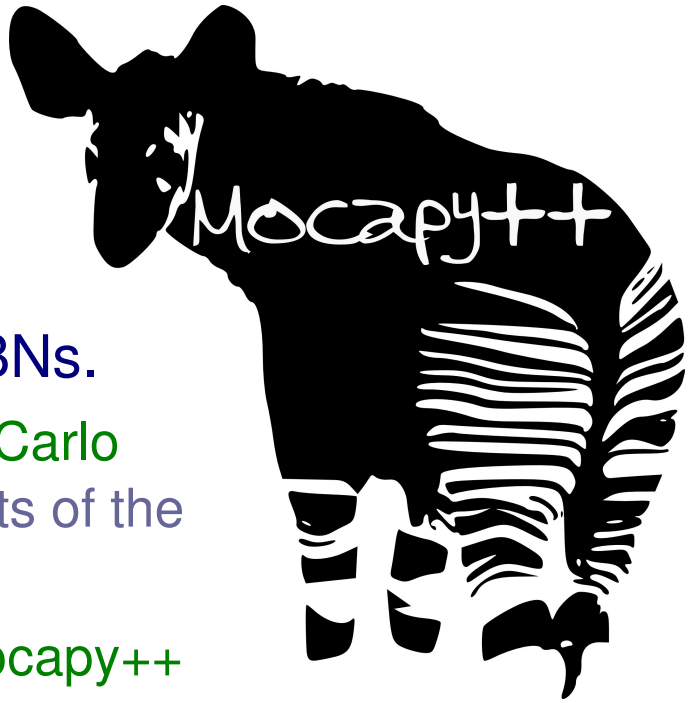
Thomas Hamelryck, May 2011

Bioinformatics center

Department of Biology

University of Copenhagen

Mocapy++



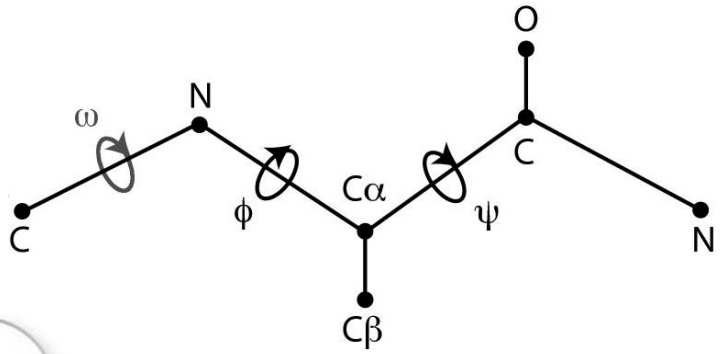
- A toolkit for learning and inference in DBNs.
  - Mocapy stands for Markov chain Monte Carlo inference and Python, two key ingredients of the original python toolkit
  - Now much faster C++ version, called Mocapy++
  - Freely available from Sourceforge
    - http://sourceforge.net/projects/mocapy/
- Learning & inference using stochastic EM
- Not just for discrete variables!
  - Categorical (=discrete, tabular), Gaussian, Dirichlet, Multinomial, Poisson
  - Directional statistics

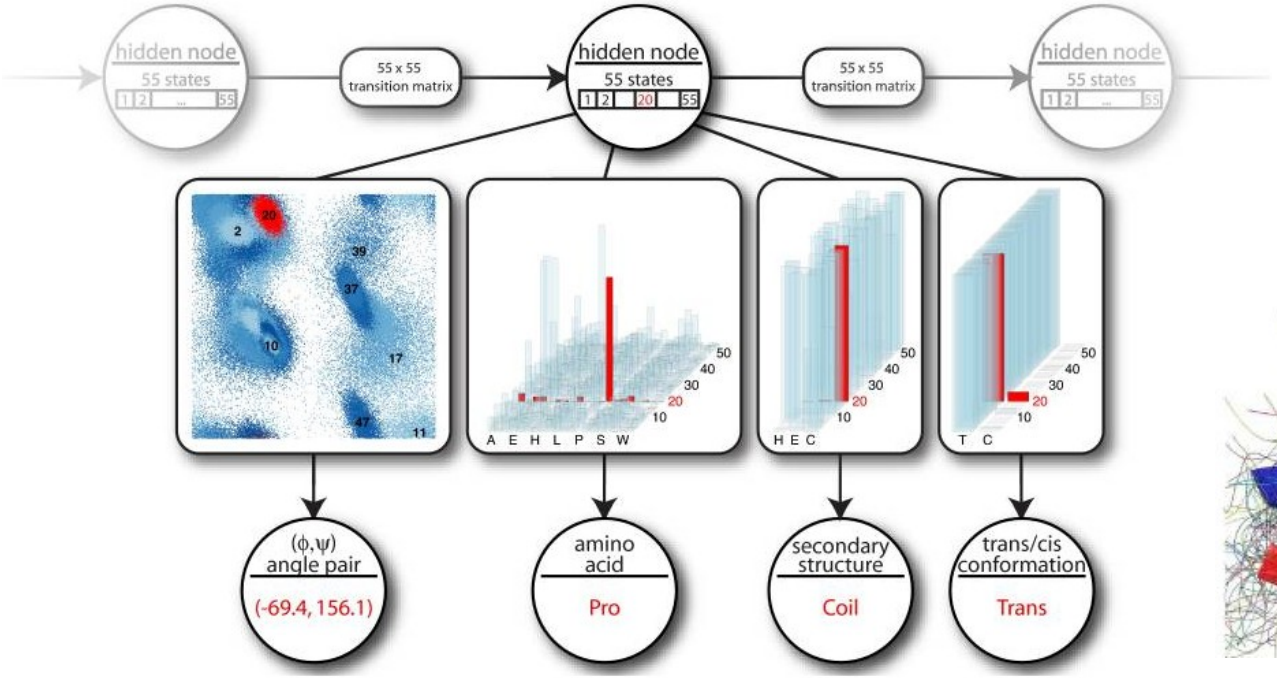
Mocapy++ in GSoC

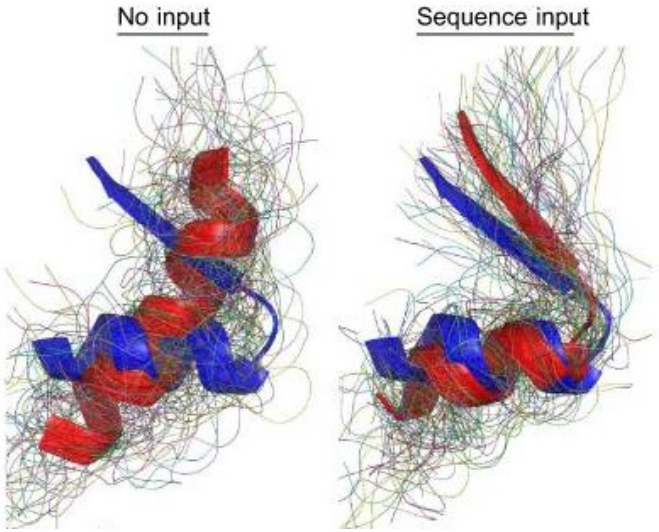
- Google summer of code
  - Offers student developers stipends
  - Open source software projects
  - Founded in 2005
    - 4500 students, 3000 mentors from over 100 countries
  - Benefits
    - Increased visibility for the projects
    - Students gain experience
    - Source code!
- Two students in 2011 through Biopython
  - Python bindings
  - Extending Mocapy++ with node classes in Python

Success stories: TorusDBN

- Local protein structure
  - PLoS Comp. Biol. (2006), 2(9), e131
  - PNAS (2009), 105(26)

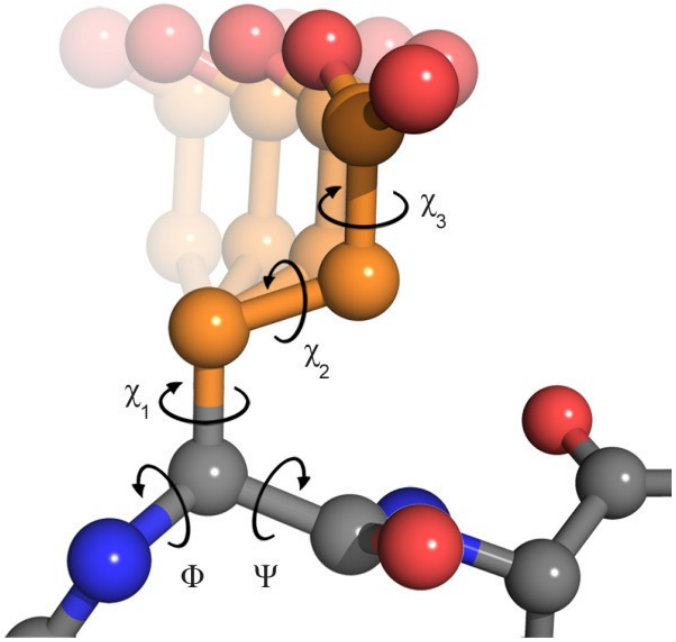


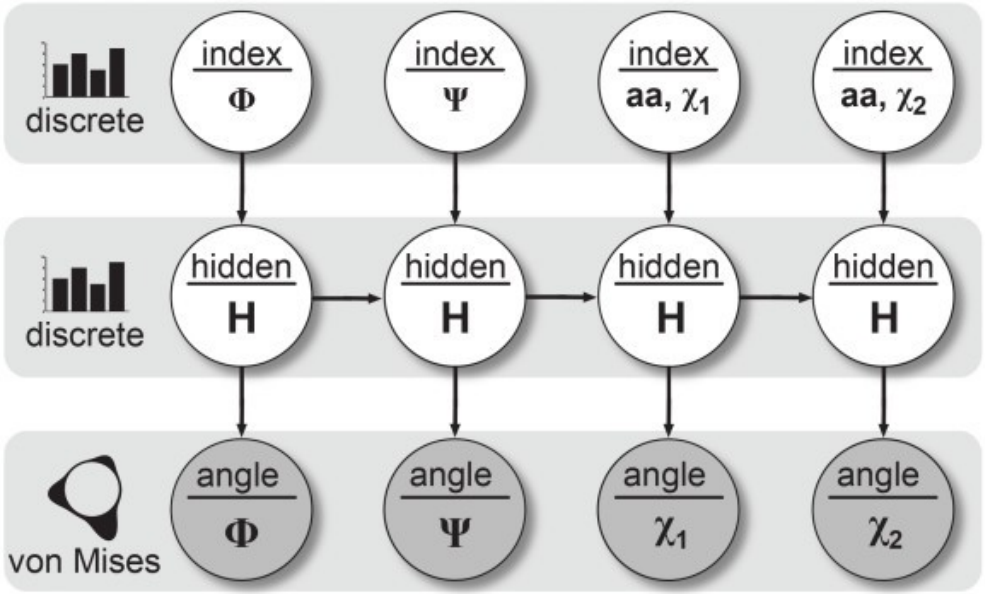




Success stories: Basilisk

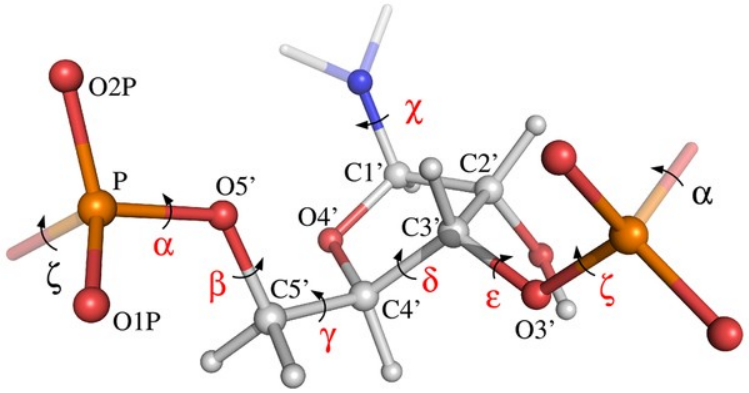
- Probabilistic model of amino acid side chains
  - BMC Bioinformatics (2010), 11, 306

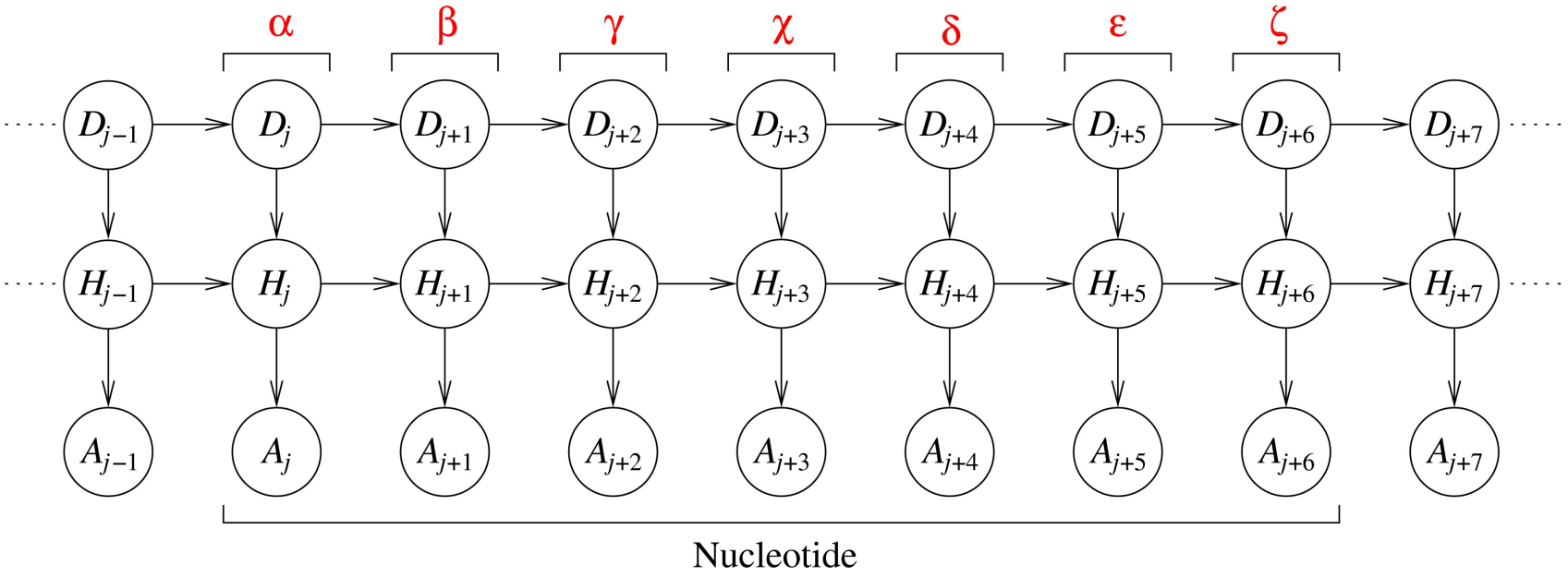




Success stories: Barnacle

- Probabilistic model of RNA structure
  - PLoS Comput Biol (2009), 5(6): e1000406





Mocapy prerequisites

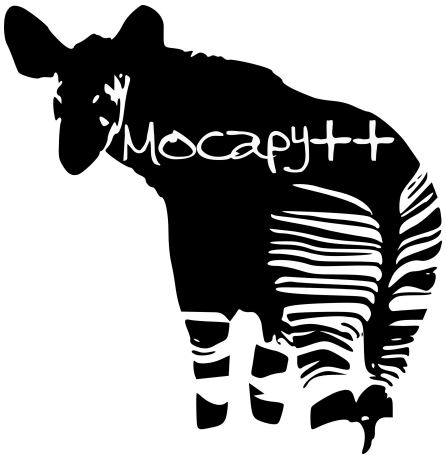
- GNU C compiler gcc (>version 4.1.2)
- Cmake (>version 2.6)
  - Configure the build system
- Boost (>version 1.36)
- LAPACK (>version 3.2)
- GNU FORTRAN compiler (>version 4.3)
- To compile, type in the root Mocapy++ directory:

  cmake .

  make

Use of Mocapy++

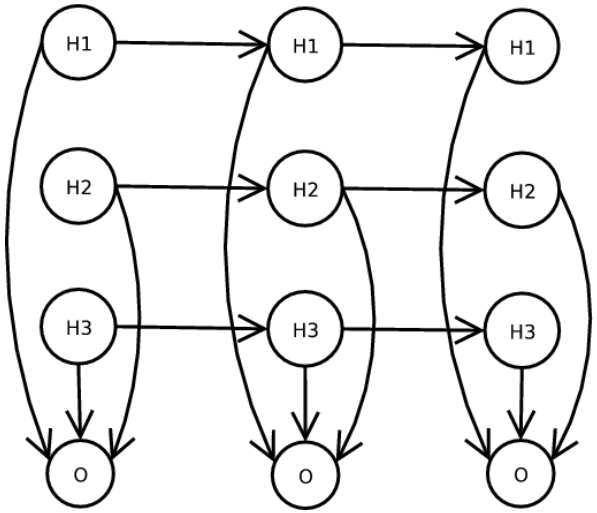
- Mocapy++ is a software library
  - Specify and use a DBN in a C++ program
  - Compile and use executable
- Compile your program
  - Put program in examples directory and use cmake
    - Add program name to the CmakeLists.txt file
    - Type “cmake .” in root directory
  - This will take care of linking with Boost, LAPACK etc.



Creating a DBN in Mocapy++

Defining a DBN in Mocapy

- Creating the node objects
  - Parameters can be picked at random
  - ...or can be specified
- Creating the DBN
  - Two-slice paradigm
    - Intra-slice connections
    - Inter-slice connections
  - Initialization of the data structures

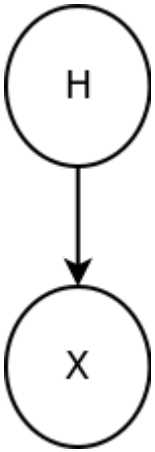


Node types (1)

- Discrete/tabular/categorical nodes
  - This node adopts a finite number of states (=node size)
  - Specified by a conditional probability table (CPT) that specifies P(x|Pa(x)) for each combination of parent/child values.
  - Shape of table: (parent 1 size, parent 2 size,..., child size)
- Example of a CPT with shape (2,2)

		Child	
	No car	Second hand car	New car
Parent: Low income	0.2	0.4	0.4
Parent: High income	0.1	0.3	0.6

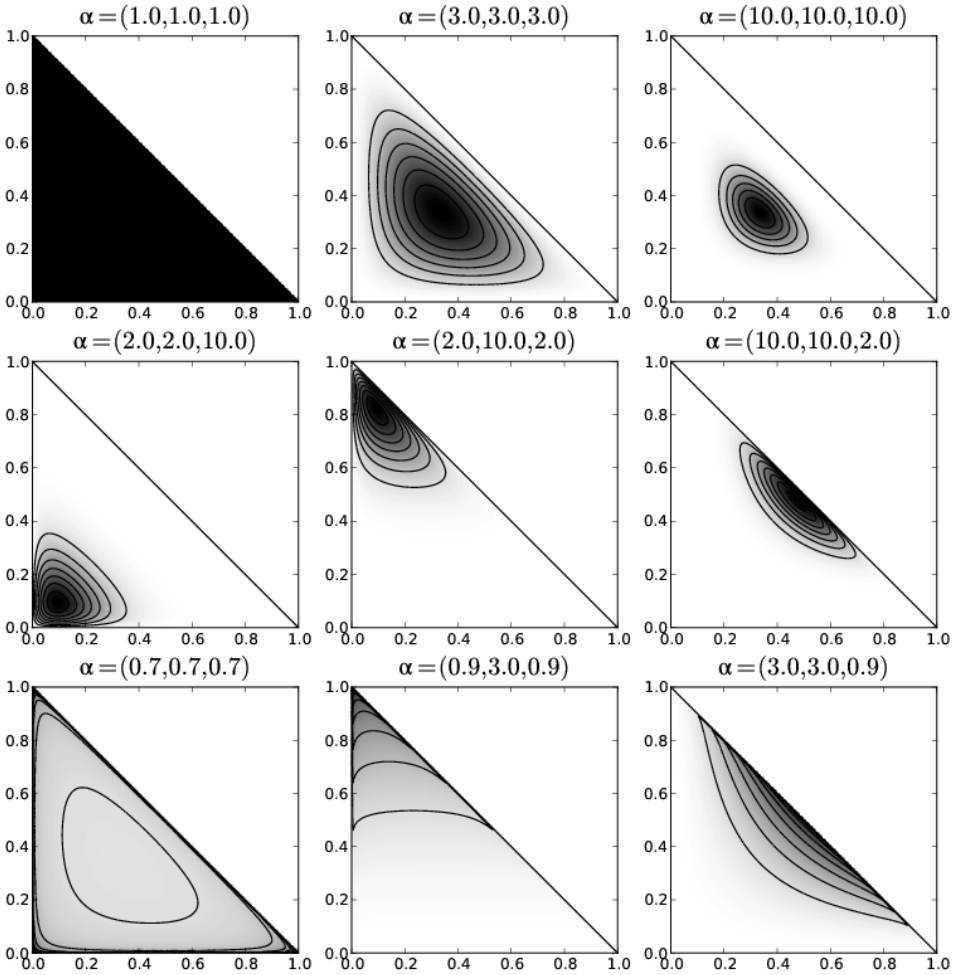
Node types (2)



- Discrete/tabular/categorical nodes
  - CPT
- All other nodes
  - They need to be leaf nodes
  - They need to have exactly one discrete node as parent
    - Each value of the discrete parent node determines a set of parameters for the conditional distribution of the child
    - Example
      - Gaussian mixture model
        - P(X|H=0) is Gaussian with mean m0, variance v0
        - P(X|H=1) is Gaussian with mean m1, variance v1
        - ...

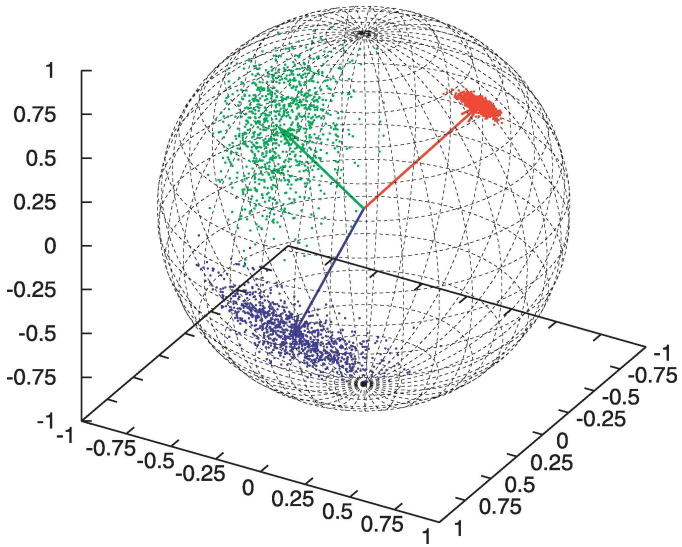
Node types (3)

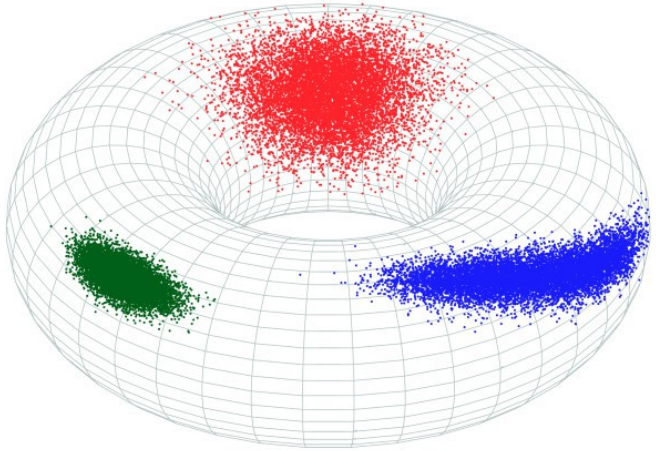
- Gaussian nodes
  - Uni- and multivariate
  - Shrinkage estimation
- Dirichlet nodes
  - Probability vectors
    - sum(X)=1 and $X_i>0$
- Poisson nodes
  - Positive integers
- Multinomial node
  - Count vectors



Node types (4)

- Directional statistics
  - Angles, directions, orientations,...
  - Molecules=bond angles
    - Modeling protein structure
- Kent and von Mises-Fisher nodes
  - Unit vectors
  - Points on the *N*-dimensional sphere
- von Mises nodes
  - Uni- and bivariate
    - Points on the circle/torus
  - Dihedral angles





Preliminaries

- Import the Mocapy header

```
#include “mocapy.h”
```

- Initialize Mocapy's random number generator, which makes everything easily reproducible

```
mocapy_seed(a);
```

Creating the nodes (1)

- A discrete node that can adopt five values
  - Size=5
  - No parents

```
CPD cpd(vec(5));
cpd.normalize();
Node *dnode1=Nodefactory::new_discrete_node(5, “X”, false, cpd);
```

  - “X”: arbitrary name of the node, used in output
  - “false”: random values, “true”: uniform probabilities

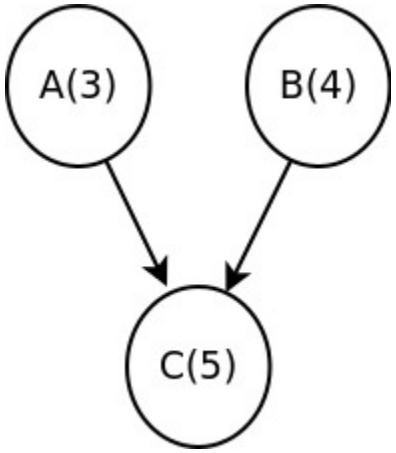
MDArray

- MDArray class
  - Mocapy++ work horse
  - Used to store data, specify CPD's, etc.

```
#define CPD MDArray<double>
```

  - Very similar to Numpy's array class

Creating the nodes (2)



- A discrete node C with size 5
  - ..and parents A,B with sizes 3 and 4

```
CPD cpd(vec(3,4,5));
cpd.normalize();
Node *dnode2=Nodefactory::new_discrete_node(5, “Y”, false, cpd);
```

  - “Y”: arbitrary name of the node, used in output
  - “false”: random values, “true”: uniform probabilities

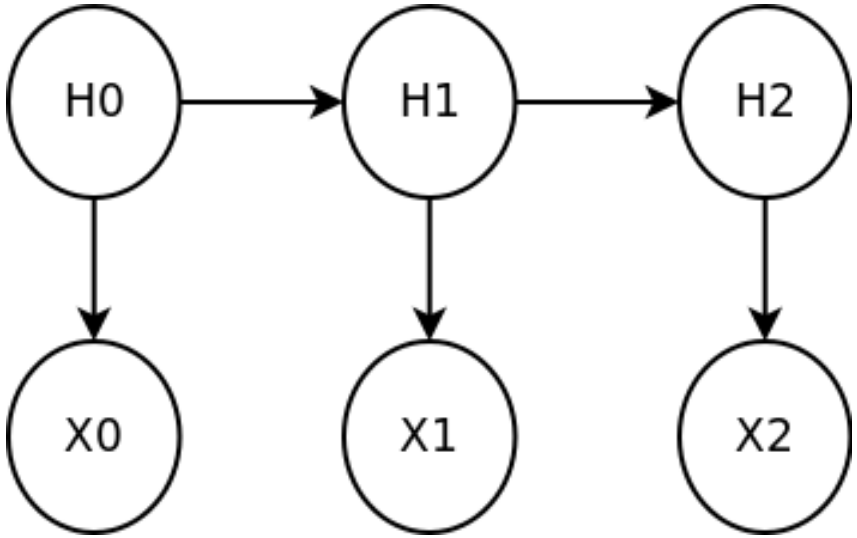
Creating the nodes (3)

- NodeFactory also creates other node types

```
Node* gnode = NodeFactory::new_gaussian_node(n);
Node* mnode = NodeFactory::new_multinomial_node(n);
Node* knode = NodeFactory::new_kent_node();
Node* vm1D = NodeFactory::new_vonmises_node();
Node* vm2D = NodeFactory::new_vonmises2d_node();
Node* pnode = NodeFactory::new_poisson_node();
```

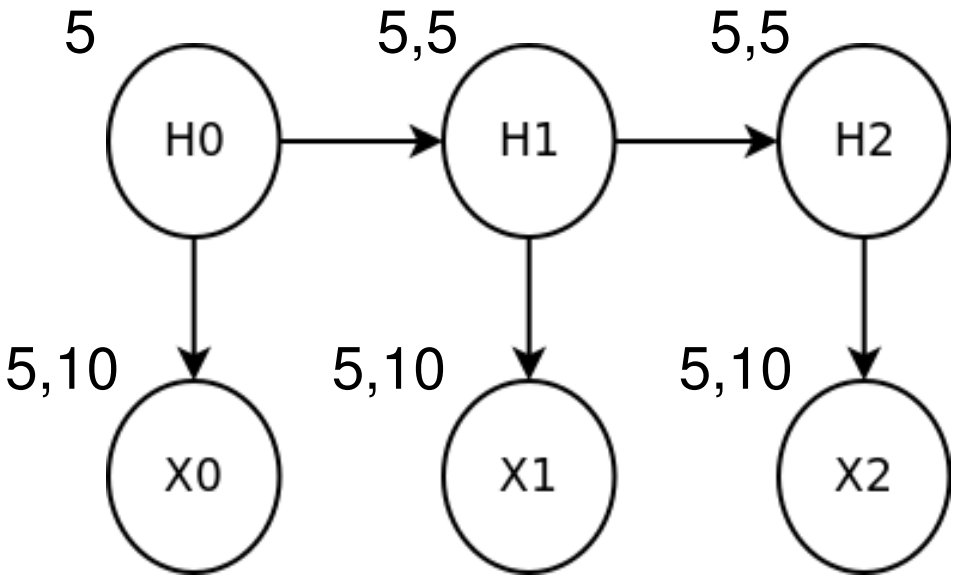
Example: an HMM

- HMM graph with 5 hidden states and 10 observed states
  - Why is the first hidden node H0 different from the next ones?
  - Is the same true for the first observed node X0?



Example: CPT shapes

- HMM
  - CPT shapes for 5 hidden states and 10 observed states



Example: creating the graph

- First a DBN object is created...
- ...which is then used to specify the graph's edges

```
// We assume that nodes h0, x0, h1 and x1 are created
DBN dbn;
vector<Node*> start_nodes = vec(h0, x0);
vector<Node*> end_nodes = vec(h1, x1);
dbn.set_slices(start_nodes, end_nodes);
dbn.add_intra(0, 1);        // You can also use the node names
dbn.add_inter(0, 0);
dbn.construct();
```

Example: tied nodes

- Let's now tie X0 and X1
  - Simply re-use the same node X

```
// We assume that nodes h0, h1 and x are created
DBN dbn;
vector<Node*> start_nodes = vec(h0, x);
vector<Node*> end_nodes = vec(h1, x);
dbn.set_slices(start_nodes, end_nodes);
dbn.add_intra(0, 1);
dbn.add_inter(0, 0);
dbn.construct();
```

Sampling from the model

- Let's try to sample a sequence from this DBN
  - Length 100

```
pair<Sequence, double> seq_ll = dbn.sample_sequence(100);
```

- Returns:
  - Sampled sequence
  - Log likelihood of the sequence
- Calculate LogLik from sequence

```
double ll=dbn.calc_ll(s);
```

Persistence

- DBN objects can be saved and retrieved
  - Makes use of Boost persistence
  - Saving:

```
dbn.save("my_dbn.pickle");
```

  - Retrieval:

```
DBN dbn;
dbn.load("my_dbn.pickle");
```

Inference and learning

Inference and learning

- We adopt a practical point-of-view: inference & learning
- Typically, some nodes in a BN will be observed (the evidence) others will be hidden. Some notation:
  - O=observed nodes (input)
  - H=hidden nodes (of interest)
  - S=nuisance hidden nodes (to be integrated out, typically discrete)
  - θ=parameters (CPT, (μ,σ) for Gaussian,...)
- Inference is assigning probabilities to the possible values of some hidden nodes, for fixed θ, given O
- Learning is estimating the parameters θ, given O

Inference

- In principle inference is easy
  - Let's assume discrete hidden nodes

$$P(\boldsymbol{H},\boldsymbol{O}) = P(\boldsymbol{H}|\boldsymbol{O})P(\boldsymbol{O})$$

$$P(\boldsymbol{H}|\boldsymbol{O}) = \frac{P(\boldsymbol{H},\boldsymbol{O})}{P(\boldsymbol{O})} = \frac{\sum_{\boldsymbol{S}} P(\boldsymbol{H},\boldsymbol{S},\boldsymbol{O})}{\sum_{\boldsymbol{H},\boldsymbol{S}} P(\boldsymbol{H},\boldsymbol{S},\boldsymbol{O})}$$

- The trick is to make inference efficient!
  - Deterministic or stochastic (sampling) methods

Stochastic inference (1)

- Gibbs sampling
  - We assume θ's are known, we sample hidden nodes
  - This is trivial to implement! (Bishop p. 542, p. 382)
  - First, fix values of the observed nodes and initialize values of hidden nodes (random, typically)
  - Repeat N times
    - Pick random hidden node
    - Sample its value conditional on the current values of all the other nodes
  - Calculate $P(h_{i...j}|\mathbf{O})$ based on the sampled values of $h_{i...j}$

Gibbs sampling

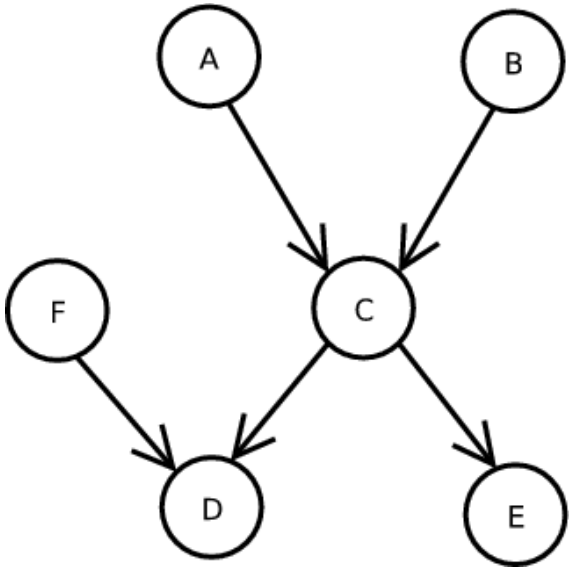
- We can sample from a high dimensional distribution P(**z**) by replacing one value for one variable at a time:

$$z_1 \sim P(z_1 | z_{2,} z_{3,} .., z_n)$$
$$z_2 \sim P(z_2 | z_{1,} z_{3,} .., z_n)$$
$$z_3 \sim P(z_3 | z_{1,} z_{2,} .., z_n)$$
$$...$$

- This clearly samples from P(**z**) (see Bishop, p. 544), since

$$P(\boldsymbol{z}) = P(z_i | z_{j \neq i}) P(z_{j \neq i})$$

Stochastic inference (2)



- Gibbs sampling in BNs
  - Based on Markov blanket of x_i.
  - Parents, children, and children's parents

$$x_i \sim P\left(x_i \middle| x_{j\neq i}\right)$$

$$P\left(x_i \middle| x_{j\neq i}\right) \propto P\left(x_i \middle| \mathrm{Pa}\left(x_i\right)\right) \prod_{c\in \mathrm{Ch}\left(x_i\right)} P\left(c \middle| \mathrm{Pa}\left(c\right)\right)$$

  - Essentially, one computes P(...) for each value of x_i and picks a value of x_i proportional to P(...)
    - C ~ P(C|A,B)P(E|C)P(D|C,F); A ~ P(A)P(C|A,B)
  - Observed nodes are fixed to their observed values

Markov blanket sampling

- We can easily see that Markov blanket sampling is indeed Gibbs sampling for BNs

$$P(x_i|x_{k\neq i}) = \frac{P(x_i, x_{k\neq i})}{P(x_{k\neq i})} = \frac{P(x_{1,}.., x_n)}{\sum_{x_i} P(x_{1,}..., x_n)} = \frac{\prod_k P(x_k|Pa(x_k))}{\sum_{x_i}\prod_k P(x_k|Pa(x_k))}$$

- Now any factor $P(x_k|Pa(x_k))$ that does NOT depend on x_i can be taken out of the sum in de denominator and cancels
- Only the Markov blanket terms remain in the numerator
  - Note that the denominator just enforces normalization

Learning

- We have assumed that the parameters θ of the BN are known
  - In most cases these need to be learned from training data.
- The training data can either be complete (all values of the nodes are known in the training data) or incomplete (there are missing node values)
  - A full Bayesian approach (with no difference between 'hidden nodes' & 'parameters') would be ideal, but since this is often intractable one often uses ML estimation
    - This can be viewed as an approximation

Learning with complete data (1)

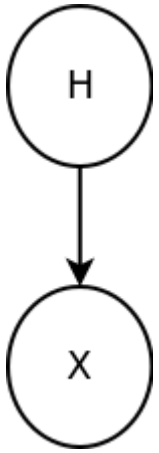
- Suppose we have a given BN graph and all data is observed
- We can simply calculate the parameters of the $P(x_i|Pa(x_i))$ distributions using ML
  - For CPTs, simply count how many times each parent/child combination occurred, and normalize for all parent values
    - Our income/car example involves 6 parent/child combinations, and hence 6 counts

Counts:

		No car	Second hand car	New car
(12,24,24)	Low income	0.2	0.4	0.4
(2,6,12)	High income	0.1	0.3	0.6

Learning with complete data (2)

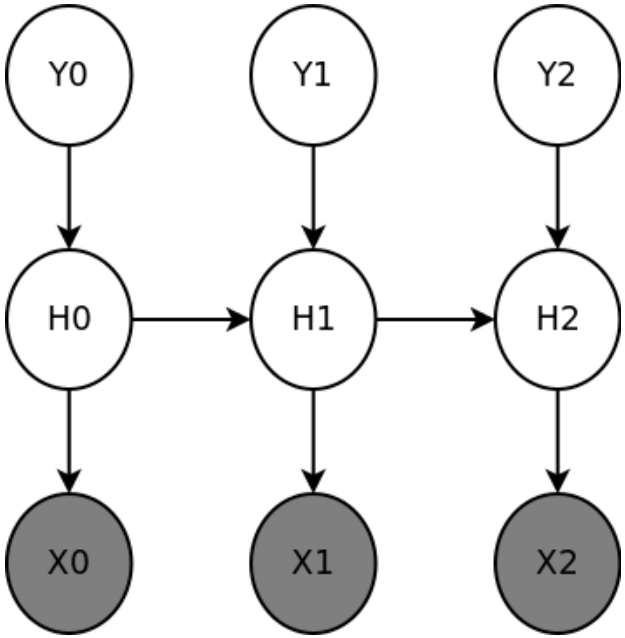
- A given BN graph and all data is observed
- Calculate the parameters of the $P(x_i|Pa(x_i))$ using ML
  - Gaussian mixture model
    - For each observation, the mixture component is given
    - Calculate mean and variance for each component
      - Example: two components
        - P(X|0) and P(X|1) are Gaussians
        - Component 0: mean=1.5
        - Component 1; mean=2.0
- Hence, the case of complete data is very simple



(H,X)
0,1.5
1,2.0
1.2.1
0,1.4
0,1.6
1,1.9

Learning with complete data (3)

- DBNs with several node types
  - Discrete nodes
    - Shown in white in example
    - Assemble parent/child counts
    - Calculate CPTs by normalizing
      - Note: not always a 2D table!
  - Other node types
    - Shown in gray
    - Always one discrete parent
    - Collect parent/child values
    - Do ML estimation for each parent value



Learning with missing data (1)

- Nearly always, some node values will be unobserved in your training set
- Missing values can be intrinsic...
  - In HMMs, the hidden nodes often do not have a clear cut interpretation, that is, they cannot be measured directly as a matter of principle
- ..or due to missing observations
  - For example, blood pressure was not measured in a diagnostic BN

Learning with missing data (2)

- Learning is done using Expectation-Maximization
  - Dempster, Laird, Rubin (1977)
  - We obtain a ML point estimate!
  - Iterative process: an E-step and an M-step
  - In the E-step, the hidden node values are inferred based on the observed node values
    - Inference can be done using any of the discussed inference methods
      - belief propagation, Gibbs sampling,...
  - In the M-step, the parameters of the BN are estimated using ML based on the filled in missing values
    - Like the “everything observed” case!

The EM algorithm

- First, the values of the hidden nodes are inferred, using the current setting of parameters θ

$$Q(\boldsymbol{H}) = P(\boldsymbol{H}|\boldsymbol{O}, \theta^{\text{old}})$$

- Then, the 'completed data' is used to update the parameters

$$\theta^{\text{new}} = \text{argmax}_{\theta} \text{E}_{\text{Q}(\boldsymbol{H})}\left\{\log P(\boldsymbol{H}, \boldsymbol{O}|\theta)\right\}$$

- Repeating this is guaranteed to end up at a (local) maximum of the log likelihood (LogLik) P(**O**|θ)
- Bishop, Chapter 9

EM as free energy minimization*

- EM can be understood as minimizing a function F(Q,θ), in terms of θ and Q(H). A local minimum of F(Q,θ) is a local maximum of the likelihood
  - Neal & Hinton, 1998

$$F(Q,\theta) = -\mathrm{E}_{Q(\boldsymbol{H})}\left\{\log P(\boldsymbol{H},\boldsymbol{O}|\theta)\right\} + \mathrm{E}_{Q(\boldsymbol{H})}\left\{Q(\boldsymbol{H})\right\}$$

- This function is equivalent to the Gibbs free energy F of statistical mechanics
  - F=U-TS
  - First term=energy term, Second term=entropy term

Monte Carlo EM

- The E-step can be done using Gibbs sampling
  - Monte Carlo EM (MC-EM)
    - A "large set" of samples approximates P(**H**|**O**,θ)
      - Wei & Tanner, 1990
- Of interest if P(**H**|**O**,θ) cannot be calculated analytically
- Suffers from the same drawbacks as classical EM
  - Depends on starting position
  - Slow convergence
  - Can get stuck at a saddle point/local maximum

Stochastic EM

- Surprisingly, it is also possible to do a single draw of Gibbs sampling for each hidden node, and proceed to the M-step
  - Stochastic EM (Diebolt & Ip, 1996)
    - Best suited for large datasets (bioinformatics!)
    - Solves many problems associated with EM/MC-EM:
      - Can escape local maxima/saddle points
      - Less dependence on starting position
      - Faster convergence, much faster than MC-EM
- In practice, it's like having complete data (M-step=ML)
- Mocapy is specifically designed for stochastic EM

Avoiding overfitting

- The size of a hidden node is a hyperparameter
  - The higher this number, the better the loglik will be
  - The loglik will not tell you when you start to overfit
- Ways to determine optimal hidden node size
  - Maximum of LogLik penalized by the number of parameters of the model, and/or taking into account the number of data points
    - Bayesian Information Criterion (BIC)
    - Akaike Information Criterion (AIC)
    - Integrated Completed Likelihood (ICL)
      - Use the completed log likelihood, cL

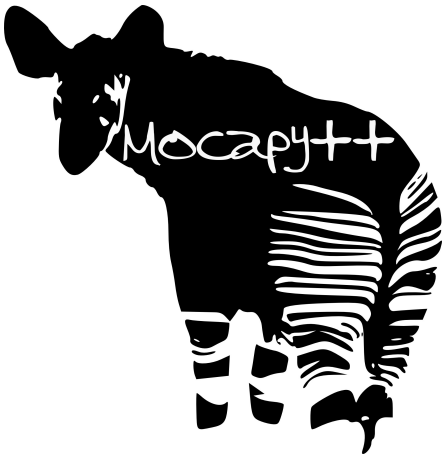
$$BIC = 2\mathrm{L} - p\ln(n)$$
$$AIC = 2\mathrm{L} - 2\mathrm{p}$$
$$ICL = 2\mathrm{cL} - p\ln(n)$$

DBN References

- *Pattern recognition and machine learning*, Christopher M. Bishop (2006), Springer verlag
  - Chapter 13, Sequential data
- *An introduction to hidden Markov models and Bayesian networks.* Ghahramani, Z. (2001), Int. J. Pattern Recogn. Art. Intell., 15, 9-42
- Kevin Murphy's PhD thesis
  - http://www.cs.ubc.ca/~murphyk/Thesis/thesis.pdf

MCMC/S-EM references

- *Markov chain Monte Carlo methods in practice*, Gilks, W., Richardson, S., Spiegelhalter, DJ. (1996), CRC Press
- *Markov chain Monte Carlo method and its applications.* Brooks, SP. (1998) The statistician, 47, 69-100
- *On stochastic versions of the EM algorithm.* (1995) Celeux, G., Chauveau, D., Diebolt, J. Technical Report 2514, INRIA Rhone-Alpes. http://citeseer.ist.psu.edu/celeux95stochastic.html
- *The stochastic EM algorithm. Estimation and asymptotic results.* Nielsen, SF. (2000) Bernoulli, 6, 457-489.



Learning and Inference with Mocapy

Overview (1)

- Inference of hidden nodes (E-step)
  - The hidden node values are filled in by Gibbs sampling
    - Markov blanket sampling
- Parameter learning (M-step)
  - Once the hidden nodes are filled in, the parameters can be estimated using ML
- Note that the main use of Mocapy is parameter learning

Overview (2)

- EM parameter learning (ML, MAP)
  - Stochastic EM
    - Diebolt & Ip, 1996
    - In the E-step, each sequence is completed by a single sweep of Gibbs sampling
    - Fast convergence, escapes local minima/saddle points, less dependence on initial conditions
  - Monte Carlo EM
    - Wei & Tanner, 1990
    - In the E-step, a large number of completed sequences are generated by Gibbs sampling

Data format (1)

- Usually you use a simple text file

  # This line is ignored

  0 5.4 0 1

  0 3.1 6 1

  0 4.0 1 8

  0 8.0 7 5

  0 3.5 4 0

  0 5.7 5 5

- Missing values can be set to zero

Data format (2)

- Load data file

```
MDArray<double> data = data_loader("data.dat");
```

- Format summary
  - #=comments
  - Empty line=new sequence
  - One line=one slice
    - Node values separated with spaces

Missing values (1)

- Missing values are indicated by a mismask array
  - 2D MDArray
  - First dimension is sequence position.
  - Second dimension is node index.
  - 0 is hidden, 1 is observed, 2 is absent nodes
- Example
  - HMM mismask file

```
0 1      # Hidden, observed
0 2      # Hidden, absent

0 1      # Second sequence
0 1
```

Missing values (2)

- Read the mismask data from the file

```
MDArray<double> mismask = data_loader("mismask.dat");
```

- So now we have
  - Specified the DBN
  - Data and mismask arrays
  - Next step is to start learning!

Inference: Sampler engine

- The sampler engine performs the Gibbs sampling.
  - Takes the DBN object as argument.
- Currently only Gibbs sampling/Random sweep

```
GibbsRandom sampler = GibbsRandom(dbn);
InfEngineMCMC inf = InfEngineMCMC(dbn, &sampler, seq,
    mismask);
```

Learning: EM engine

- The EM engine performs S-EM or MC-EM learning
  - It uses the inference engine object to do the inference of the hidden node values
- The data is passed to the EM engine
  - List of sequences and a list of mismasks

```
GibbsRandom mcmc = GibbsRandom(model_dbn);
EMEngine em = EMEngine(model_dbn, seq_list, mismask_list);
```

EM cycle (1)

- The (S-)EM cycle is done as follows:

```
for (uint i=0; i<100; i++) {
    em.do_E_step(50, 50, true);
    double ll = em.get_loglik();
    em.do_M_step();
    cout < < “LL=” < < ll < < endl;
}
```

EM cycle (2)

- Some things to note:
  - Save your DBN regularly!
  - Do a short burn in step before the first M-step
  - How many samples?
    - S-EM: number of sampling steps=1
    - MC-EM: number of sampling steps>>1
  - Returned LogLik is that of the completed data
    - Can be used to evaluate convergence

You are on your own

- The Mocapy++ root directory contains an example directory
  - MDArray
  - HMMs
    - Discrete output
    - Gaussian, von Mises, etc.
    - Factorial HMM

THE END