

BAYES-LIN: An object-oriented environment for Bayes linear local computation

Darren J. Wilkinson*
Copyright © 1997–2000

April 7, 2000

The latest version of the BAYES-LIN software and documentation (including the latest version of this document), can be obtained from the BAYES-LIN WWW page:

<http://www.staff.ncl.ac.uk/d.j.wilkinson/bayeslin/>

Abstract

BAYES-LIN is an extension of the LISP-STAT object-oriented statistical computing environment, which adds to LISP-STAT some object prototypes appropriate for carrying out Bayes linear analyses and local computation *via* message-passing between clique-tree nodes of Bayes linear belief networks. Currently the BAYES-LIN system represents a rather low-level set of tools for a back-end computational engine. A GUI front end, allowing interactive formulation of DAG models could be easily added, but is currently missing from the system. This document provides a very brief introduction to the system, by means of a work-through of two example computations, followed by a list of variables, functions, objects and methods provided by the system.

This document describes the *0.3 alpha* release. Make sure your documentation matches the version of the software that you are using.

*Department of Statistics, University of Newcastle, Newcastle upon Tyne, NE1 7RU, ENGLAND.
Email: d.j.wilkinson@ncl.ac.uk WWW: <http://www.staff.ncl.ac.uk/d.j.wilkinson/>

Contents

1	Introduction	3
1.1	Bayes linear methods	3
1.2	LISP-STAT	3
1.3	Local computation	3
1.4	Installing and running BAYES-LIN	3
2	Example from BLM 2	4
3	A simple DLM	6
3.1	A global analysis	6
3.2	Local computation	7
4	Other examples	9
5	Reference	12
5.1	Global functions	12
5.2	Object prototypes	13

1 Introduction

1.1 Bayes linear methods

Bayes linear methods are a form of Bayesian statistics, which acknowledge the difficulties associated with the full modelling, specification, and conditioning required by distributional Bayesian statistics, and instead try to make best possible use of partial specifications, based on means, variances and covariances. Unsurprisingly, much of the theory is formally identical to inference in multivariate Gaussian Bayesian networks, but interpretation of results is generally different. This document assumes a working knowledge of the basic tools of the Bayes linear methodology. An introduction to Bayes linear methods is given in [1]. An introduction to (non-local) computational issues can be found in [6]. The foundations of the theory are discussed in [5], [3], and [2]. On-line, an introduction to the theory can be found in [4], from the Bayes Linear Methods WWW home page: <http://www.maths.dur.ac.uk/stats/bayeslin/>

1.2 LISP-STAT

LISP-STAT is an interpreted, object-oriented environment for statistical computing, described in [9]. This document assumes a working knowledge of LISP-STAT, and the basics of object-oriented programming. On-line, LISP-STAT information is available from the LISP-STAT WWW home page: <http://www.stat.umn.edu/~luke/xls/xlsinfo/xlsinfo.html>

1.3 Local computation

BAYES-LIN carries out local computation *via* message-passing between adjacent nodes of a clique-tree representing the statistical model of interest. Again, local computation in Bayesian networks is a huge area, and this document assumes a working knowledge of graphical models, conditional independence and some of the ideas behind local computation. The best introduction to all of these areas is [8]. In particular, Chapter 3 of that volume deals with all of the relevant graph-theoretic concepts, and Section 7.2 gives an introduction to graphical Gaussian models. The algorithm currently used for carrying out local computation is loosely based on the pure Gaussian case of [7].

1.4 Installing and running BAYES-LIN

You need a working LISP-STAT system installed before you attempt to install BAYES-LIN. The following instructions are for a UNIX system with an XLISP-STAT installation, but installing on other systems should be similar. Note that the graphics work best on systems with at least a 16 bit colour display. If you only have an 8 bit display (256 colours), make sure that most are free for use by BAYES-LIN. The graphics will not work on displays poorer than 8 bit colour. Create a new directory for the BAYES-LIN system. Download the BAYES-LIN software from the BAYES-LIN WWW page: <http://www.staff.ncl.ac.uk/d.j.wilkinson/bayeslin/> and put into the new directory. In this new directory type:

```
% gunzip blin02a.tar.gz
% tar -xvf blin02a.tar
% gzip blin01a.tar
```

You should then be able to run LISP-STAT with the BAYES-LIN extensions simply by running

```
% xispstat
```

from within this directory. You can check that the extensions are loaded by typing in some of the following commands in the LISP-STAT listener window.

```
> (help 'var-mat)
> (send var-mat-proto :help)
> (send var-mat-proto :help :inverse)
```

Note that all BAYES-LIN global functions, object prototypes and methods have on-line documentation, which should be consulted frequently, as not all features are covered fully in this manual.

In general, to make sure the extensions are loaded, use the expression

```
> (require "bayeslin")
```

When you are satisfied that the extensions are loaded, exit BAYES-LIN.

```
> (exit)
```

In order to run the examples, simply call LISP-STAT with the example as first argument. *eg.*

```
% xispstat blm2ex.lsp
```

or

```
% xispstat ex-dlm
```

These two examples will be explained in the following sections.

2 Example from BLM 2

This example does not involve any local computation, but is useful for illustrating the basic concepts, in the context of a well known example from [11], where further details relating to the background of this example can be found. $B = (B_1, B_2)^T$ and $D = (D_1, D_2)^T$ are random vectors. D is to be observed, and used to make inferences about B . Beliefs over B and D are as follows:

$$E(B) = E(D) = \begin{pmatrix} 4.16 \\ 6.25 \end{pmatrix}, \text{Var}(B) = \text{Var}(D) = \begin{pmatrix} 1.12 & 0.72 \\ 0.72 & 2.43 \end{pmatrix}, \text{Cov}(B, D) = \begin{pmatrix} 0.62 & 0.30 \\ 0.30 & 0.43 \end{pmatrix}.$$

BAYES-LIN defines objects for expectation vectors, variance and covariance matrices. We can set these up as follows.

```
(def b '("B1" "B2"))
(def d '("D1" "D2"))
(def eb
  (obs-vec b '(4.16 6.25)))
(def ed
  (obs-vec d '(4.16 6.25)))
(def vb
  (var-mat b #2a((1.12 0.72) (0.72 2.43))))
(def vd
  (var-mat d #2a((1.12 0.72) (0.72 2.43))))
(def cbd
  (cov-mat b d #2a((0.62 0.30) (0.30 0.43))))
```

We start by declaring the variables to LISP-STAT, and then create objects representing the belief specifications above. The global function `obs-vec` is used to create an object of type `OBS-VEC-PROTO`, and `var-mat` is used to create an object of type `VAR-MAT-PROTO`. All `BAYES-LIN` objects respond to the `:save` message, and so a good way to examine the current state of an object is to send it the `save` message. For example, typing `(send vb :save)` at the listener should give something like

```
(VAR-MAT (LIST "B1" "B2") #2A((1.12 0.72) (0.72 2.43)))
```

Notice that vector and matrix objects in `BAYES-LIN` *must* have a collection of variable labels bound to them. The motivation for this is discussed in [10]. In short, it ensures that operations on matrices and vectors always makes sense, and frees the user from having to make sure that bases match up properly. It also allows for convenient combination and sub-setting of objects. We can extract the labels, vectors and matrices as follows:

```
> (send eb :labels)
("B1" "B2")
> (send eb :obs)
(4.16 6.25)
> (send vb :labels)
("B1" "B2")
> (send vb :matrix)
#2A((1.12 0.72) (0.72 2.43))
```

If we put $A = (B, D)^T$, then we can set up expectation and variance matrices for A as follows.

```
(def a
  (append b d))
(def ea
  (send eb :append ed))
(def va
  (send vb :append vd :cov cbd))
```

Note that we can extract the expectation vector of B with `(send ea :e b)`, and the variance matrix of B with `(send va :var b)`. These return objects of type `OBS-VEC-PROTO` and `VAR-MAT-PROTO`, respectively, representing the expectation vector and variance matrix for B . $[B/D]$ users should also note that many objects, including those of type `OBS-VEC-PROTO` and `VAR-MAT-PROTO`, respond to the message `:bd-code`, which outputs $[B/D]$ code for construction of the object, allowing for convenient integration and comparison with the $[B/D]$ system.

Now we have set up an expectation vector and variance matrix for A , we can bind them together into a *belief structure* for A , representing all second order prior beliefs for A .

```
(def bs
  (belief-structure ea va))
```

The resulting object, of type `BELIEF-STRUCTURE-PROTO`, contains all prior beliefs relating to a collection of random quantities. In addition, it can also contains information relating to data and the adjustment of the belief structure by that data. Sending the object the `:info` message will return some brief information relating to the current state of the object. $[B/D]$ users may find it helpful to think of an object of type `BELIEF-STRUCTURE-PROTO` as being equivalent to a

[B/D] session. The key to understanding the fundamental difference between [B/D] and BAYES-LIN is that BAYES-LIN allows the construction of many such objects, which can communicate in order to carry out local computation. We can define some data for D and then introduce it into the belief structure as follows:

```
(def od
  (obs-vec d '(5.4 9.8)))
(send bs :observe od)
```

The `:observe` message carries out the adjustment of the structure by the observed data. Sending the adjusted object the `:e` or `:var` message will now return the adjusted expectation vector and variance matrix. [B/D] users can send the object the `:bd-code` message, and this will return a complete [B/D] program for setting up a similar belief structure, and carrying out an equivalent adjustment. This is very useful for checking the validity of results from BAYES-LIN.

Rather than adjusting by both observations simultaneously, sequential adjustment can be carried out. For example, the following commands should result in an object in the same state.

```
(def bs2
  (belief-structure ea va))
(send bs2 :observe
  (obs-vec '("D1") '(5.4)))
(send bs2 :observe
  (obs-vec '("D2") '(9.8)))
```

3 A simple DLM

3.1 A global analysis

Consider a simple univariate locally constant DLM,

$$X_t = \theta_t + v_t$$

$$\theta_t = \theta_{t-1} + \omega_t.$$

Here θ_t represents the slowly evolving underlying state of the process, and X_t denote the noisy observation. v_t and ω_t can be thought of as uncorrelated noise processes, with $E(v_t) = E(\omega_t) = 0$, $\text{Var}(v_t) = V$ and $\text{Var}(\omega_t) = W$. In this example, we will take $V = W = 1$, $E(\theta_1) = 1$ and $\text{Var}(\theta_1) = 1$. We can specify this as a belief structure as follows.

```
(def theta
  '("T1" "T2" "T3"))
(def x
  '("X1" "X2" "X3"))
(def all
  (append theta x))

(def ex-all
  (obs-vec all (repeat 1 6)))
(def var-all
```

```

(var-mat all #2a((1 1 1 1 1 1)
(1 2 2 1 2 2)
(1 2 3 1 2 3)
(1 1 1 2 1 1)
(1 2 2 1 3 2)
(1 2 3 1 2 4))))
(def bs
  (belief-structure ex-all var-all))

```

Entering `(send bs :info)` will give some information about the belief structure. Now, based on the observations $x_1 = 3, x_2 = -1$, we can introduce data sequentially

```

(send bs :observe
  (obs-vec '("X1") '(3)))
(send bs :observe
  (obs-vec '("X2") '(-1)))

```

and see the effect of this with `(send bs :info)`. This has used exactly the same global adjustment procedure as before. Obviously a global analysis does not make proper use of the conditional independencies we know are present in the model, and hence will not scale up well to larger models.

3.2 Local computation

We can now see how to do this example using local computation. First some more objects are required. Before any computation takes place, the model must be specified. This is most easily done through the specification of a DAG model. To do this we need an extra couple of objects. First we need a DAG node object, and then we need a DAG object to contain them. DAG nodes are constructed from the `DAG-NODE-PROTO` object, using the global function `dag-node`. DAG models are derived from the `DA-GRAPH-PROTO` using the global function `da-graph`. DAG nodes without any parents are defined in a similar way to belief structures. *eg.* the call `(dag-node NAMESTR nil EX nil VAR)` returns an object whose name string is `NAMESTR`, expectation vector is the `OBS-VEC`, `EX`, and variance matrix is the `VAR-MAT`, `VAR`. For a node with parents, the conditional distribution for the node needs to be specified. If the random variables represented by the parents of node Y are all put together into a single vector, X , then specifications need to be made for v, A and V , where

$$E_X(Y) = v + AX$$

$$\text{Var}_X(Y) = V.$$

Then a command like `(dag-node "Y" '("X1" "X2" "X3") NU A V)` can be used to build the node. The function `da-graph` accepts a list of DAG nodes and returns a DAG object. Alternatively, an empty DAG may be created, and nodes may be added one at a time using the `:add` method. This will be made clearer in the context of the example. A diagrammatic representation of the DAG for the model is given in Figure 1. This DAG may be constructed in `BAYES-LIN` using the following command.

```

(def dag (da-graph (list
  (dag-node "gT1" nil

```

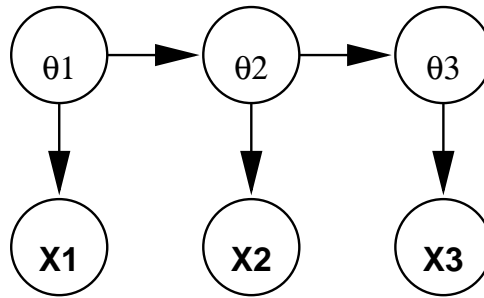


Figure 1: DAG for the DLM

```

(obs-vec '("T1") '(1))
nil
(var-mat '("T1") #2a((1)))
)
(dag-node "gT2" '("gT1")
(obs-vec '("T2") '(0))
(cov-mat '("T2") '("T1") #2a((1)))
(var-mat '("T2") #2a((1)))
)
(dag-node "gT3" '("gT2")
(obs-vec '("T3") '(0))
(cov-mat '("T3") '("T2") #2a((1)))
(var-mat '("T3") #2a((1)))
)
(dag-node "gX1" '("gT1")
(obs-vec '("X1") '(0))
(cov-mat '("X1") '("T1") #2a((1)))
(var-mat '("X1") #2a((1)))
)
(dag-node "gX2" '("gT2")
(obs-vec '("X2") '(0))
(cov-mat '("X2") '("T2") #2a((1)))
(var-mat '("X2") #2a((1)))
)
(dag-node "gX3" '("gT3")
(obs-vec '("X3") '(0))
(cov-mat '("X3") '("T3") #2a((1)))
(var-mat '("X3") #2a((1)))
)
)))

```

This defines the DAG object, completely specifying the model. In general however, propagation can't be carried out directly on the DAG*. So we need to create a clique (or junction) tree for propagation. The command

*However, this particular DAG is actually a causal tree, where propagation can be carried out directly, rather easily. Note that BAYES-LIN currently does not provide any facility for direct propagation on causal trees or polytrees, since junction tree propagation can always be used instead, and doesn't carry much additional overhead.

```
(send dag :make-clique-tree)
```

Constructs a moral graph, and then uses MCS to construct a triangulation and associated clique-tree. This clique tree is initialised, according to the scheme used in [7]. This scheme ensures global correctness of prior belief, but not local consistency. In BAYES-LIN local consistency is achieved with the `:propagate` message, which may be send to the appropriate DAG object (which passes it on to it's associated clique tree).

```
(send dag :propagate)
(send dag :info)
```

After propagation, information can be obtained with a `:info` message. This message should not be used on a tree which isn't locally consistent. Data can be introduced with the `:observe` message. Data items must be contained in a given DAG node. For example, the following commands introduce the data for our problem.

```
(send dag :observe "gX1" (obs-vec '("X1") '(3)))
(send dag :observe "gX2" (obs-vec '("X2") '(-1)))
```

Again, entering data results in a structure which is globally correct, but not locally consistent, so propagation must be carried out before attempting to obtain information from the graph.

```
(send dag :propagate)
(send dag :info)
```

4 Other examples

The above examples are very simple, so the BAYES-LIN package comes with some less trivial (but undocumented) examples which illustrate how the system can be used in practice. The file `ex-dlm2.lsp`, extends the previous example to an arbitrary number of time points, and shows how the adjusted information may be extracted and plotted. The code for this example is included below for ease of reference.

```
;; ex-dlm2.lsp
;; a bigger DLM example

;; variance components
(def v #2a((4)))
(def w #2a((0.25)))
;; data
(def x
 '(1 3 7 4 9 2 6 5 2 8 10 9 12 15 13 17 14 16 18 17 19 18 19 17 20 19 18 19)
)
;; prior for the initial values
(def t0 '(0))
(def v0 #2a((1000)))

;; now construct the DAG
(def dag (da-graph))
;; initial state
```

```

(send dag :add
  (dag-node "gT0" nil
    (obs-vec '("T0") t0)
    nil
    (var-mat '("T0") v0)
  )
)
;; set up rest of the model
(dolist (i (iseq 1 (length x)))
  (let* (
    (t-name (format nil "T~a" i))
    (x-name (format nil "X~a" i))
    (gt-name (format nil "gT~a" i))
    (gx-name (format nil "gX~a" i))
    (prev-t-name (format nil "T~a" (- i 1)))
    (prev-gt-name (format nil "gT~a" (- i 1)))
  )
    ;; add next theta
    (send dag :add
      (dag-node gt-name
        (list prev-gt-name)
        (obs-vec (list t-name) '(0))
        (cov-mat (list t-name) (list prev-t-name) #2a((1)))
        (var-mat (list t-name) w)
      )
    )
    ;; add next x
    (send dag :add
      (dag-node gx-name
        (list gt-name)
        (obs-vec (list x-name) '(0))
        (cov-mat (list x-name) (list t-name) #2a((1)))
        (var-mat (list x-name) v)
      )
    )
  )
)

;; once dag is set up, make the clique tree
(send dag :make-clique-tree)

;; optionally, we can make the tree locally consistent to
;; inspect it a priori, though we could just go ahead and
;; introduce data straight away if we wanted to
(send dag :propagate)
(send dag :info)

;; now introduce data
(dolist (i (iseq 1 (length x)))
  (let* ((x-name (format nil "X~a" i))
    (gx-name (format nil "gX~a" i))
  )
  )
)

```

```

    (send dag :observe gx-name
      (obs-vec (list x-name) (list (select x (- i 1))))
    )
  )
)

;; make the tree locally consistent, then inspect it
(send dag :propagate)
(send dag :info)

;; create a vector of smoothed values
(def smoothed-x (repeat 0 (length x)))
(dotimes (i (length x))
  (setf (select smoothed-x i)
    (first (send
      (send dag :e (format nil "gT~a" (+ i 1)))
      :obs)))
  )

;; plot the data and the smoothed values
(def plot
  (plot-points (iseq 1 (length x)) x
    :color 'green)
  )
(send plot :add-lines
  (iseq 1 (length x)) smoothed-x
  :color 'magenta)

;; now create a vector of adjusted standard deviations
(def sd-vec (repeat 0 (length x)))
(dotimes (i (length x))
  (setf (select sd-vec i)
    (first (send
      (send dag :sd-vec (format nil "gT~a" (+ i 1)))
      :obs)))
  )

;; now add sd-lines to the plot
(send plot :add-lines (iseq 1 (length x))
  (+ smoothed-x (* 2 sd-vec))
  :type 'dashed :color 'red)
(send plot :add-lines (iseq 1 (length x))
  (- smoothed-x (* 2 sd-vec))
  :type 'dashed :color 'red)

;; now add predictive standard deviations
(def vs (select v 0 0))
(send plot :add-lines (iseq 1 (length x))
  (+ smoothed-x (* 2 (sqrt (+ (^ sd-vec 2) vs))))
  :type 'dashed :color 'blue)

```

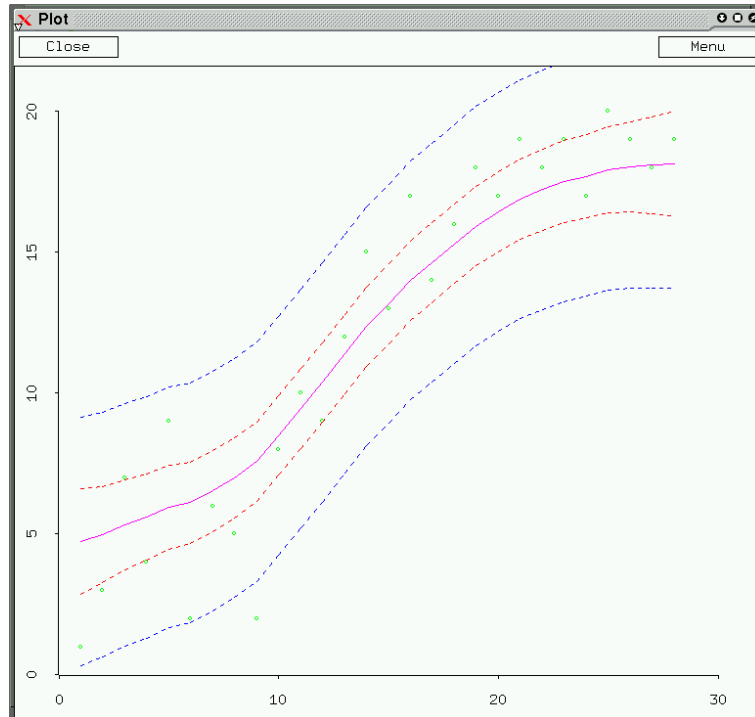


Figure 2: Plot produced by `ex-dlm2.lisp`.

```
(send plot :add-lines (iseq 1 (length x))
  (- smoothed-x (* 2 (sqrt (+ (^ sd-vec 2) vs))))
  :type 'dashed :color 'blue)
```

Running this example results in a plot similar to Figure 2.

There are also some example files for *directed* Markov random fields[†]. `ex-mrf.lisp` builds a small MRF and introduces a few observations. `ex-mrf2.lisp` is really just a bigger example of this. `ex-mrf3.lisp` builds a hidden field, with a noisy observational layer, and uses the observed layer to make inferences for the hidden layer. Some of these examples may require some customisation of your LISP-STAT executable — see the source files for details.

5 Reference

5.1 Global functions

```
belief-structure (obs-vec var-mat &optional (canonical nil))
blin-add (&rest args)
blin-mult (&rest args)
clique-tree (&optional (nodes nil))
cov-mat (llabels rlabels mat)
dag-node (name parents nu A X &optional (can nil))
da-graph (&optional (nodes nil))
latex-matrix (mat &optional (s t))
```

[†]Directed MRFs have rather strange asymmetry properties which mean they are probably not terribly useful for model real spatial phenomena.

```

matrix-trace (mat)
moral-graph (&optional (nodes nil))
moral-node (name neighbours labels &optional (ctn nil) (loc nil))
moral-plot (moral-graph &optional (rad 50))
obs-vec (labels obs)
tree-node (name neighbours bs)
tree-sep (name neighbours bs)
var-mat (labels mat)

```

Documentation for global functions can be obtained interactively using the on-line help system. *eg.* (help 'belief-structure) will give the documentation for the belief-structure function.

5.2 Object prototypes

```

belief-structure-proto '(h ph K pK obs) nil nil
blin-graph-proto '(nodes) nil nil
clique-tree-proto nil nil blin-graph-proto
cov-mat-proto '(llabels rlabels matrix) nil nil
dag-node-proto '(parents nu A L ctn ctns) nil
da-graph-proto '(mg ct) nil (list blin-graph-proto)
graph-node-proto '(name neighbours) nil nil
moral-graph-proto '(clique-tree) nil blin-graph-proto
moral-node-proto '(labels ctn location label) nil
moral-plot-proto '(moral-graph radius) nil graph-window-proto
obs-vec-proto '(labels obs) nil nil
tree-node-proto nil nil
tree-sep-proto nil nil
var-mat-proto '(labels matrix) nil nil

```

Documentation for object prototypes can be obtained interactively using the on-line help system. *eg.* (send belief-structure-proto :help) will give a brief description and a list of associated messages. Then, for example, (send belief-structure-proto :help :bd-code) will give the documentation for the :bd-code method.

References

- [1] M. Farrow and M. Goldstein. Bayes linear methods for grouped multivariate repeated measurement studies with application to crossover trials. *Biometrika*, 80(1):39–59, 1993.
- [2] M. Goldstein. Revising previsions: a geometric interpretation. *J. R. Statist. Soc.*, B:43:105–130, 1981.
- [3] M. Goldstein. Revising exchangeable beliefs: subjectivist foundations for the inductive argument. In P. Freeman and A.F.M. Smith, editors, *Aspects of Uncertainty: A Tribute to D. V. Lindley*. Wiley, 1994.
- [4] M. Goldstein. Bayes linear methods I - Adjusting beliefs: concepts and properties. Technical Report 1995/1, Department of Mathematical Sciences, University of Durham, 1995.
- [5] M. Goldstein. Prior inferences for posterior judgements. In M. L. D. Chiara et al., editors, *Structures and norms in science*. Pordrecht Kluwer, 1997.
- [6] M. Goldstein and D. A. Wooff. Bayes linear computation: concepts, implementation and programming environment. *Statistics and Computing*, 5:327–341, 1995.
- [7] S. L. Lauritzen. Propagation of probabilities, means, and variances in mixed graphical association models. *J. Amer. Statist. Assoc.*, 87,420:1098–1108, 1992.
- [8] J. Pearl. *Probabilistic reasoning in intelligent systems*. Morgan Kaufmann, 1988.
- [9] L. Tierney. *LISP-STAT: An object oriented environment for statistical computing and dynamic graphics*. Wiley, 1990.
- [10] D. J. Wilkinson. An object-oriented approach to local computation in Bayes linear belief networks. In R. Payne and P. Green, editors, *Proceedings in Computational Statistics 1998*, pages 491–496, Heidelberg, 1998. Physica-Verlag.
- [11] D. A. Wooff. Bayes linear methods II - An example with an introduction to [B/D]. Technical Report 1995/2, Department of Mathematical Sciences, University of Durham, 1995.