Generating Data Analysis Programs from Statistical Models

Position Paper

Bernd Fischer[†], Johann Schumann[†], and Tom Pressburger[‡]

[†]RIACS / NASA Ames, [‡]QSS / NASA Ames, Moffett Field, CA 94035 USA,
{fisch,schumann,ttp}@ptolemy.arc.nasa.gov

Abstract. Extracting information from data, often also called data analysis, is an important scientific task. Statistical approaches, which use methods from probability theory and numerical analysis, are well-founded but difficult to implement: the development of a statistical data analysis program for any given application is time-consuming and requires knowledge and experience in several areas.
In this paper, we describe AutoBayes, a high-level generator system for data analysis programs from statistical models. A statistical model specifies the properties for each problem variable (i.e., observation or parameter) and its dependencies in the form of a probability distribution. It is thus a fully declarative problem description, similar in spirit to a set of differential equations. From this model, AutoBayes generates optimized and fully commented C/C++ code which can be linked dynamically into the Matlab and Octave environments. Code is generated by schema-guided deductive synthesis. A schema consists of a code template and applicability constraints which are checked against the model during synthesis using theorem proving technology. AutoBayes augments schema-guided synthesis by symbolic-algebraic computation and can thus derive closed-form solutions for many problems. In this paper, we outline the AutoBayes system, its theoretical foundations in Bayesian probability theory, and its application by means of a detailed example.

1 Introduction

Data analysis denotes the transformation of data (i.e., pure numbers) into more abstract information. It is at the core of all experimental sciences—after all, experiments result only in new data, not in new information. Consequently, scientists of all disciplines spend much time writing and changing data analysis programs, ranging from trivial (e.g., linear regression) to truly complex (e.g., image analysis systems to detect new planets). A variety of methods is used for data analysis but all rigorous approaches are ultimately based on statistical methods [BH99]. Amongst these, Bayesian methods offer conceptual advantages in handling prior information and missing data and have thus become the methods of choice for many applications.

We believe that data analysis, especially data analysis based on Bayesian statistics, is a very promising application area for program generation. Probability theory provides an established, domain-specific notation which can form the basis of a specification language. Probability theory and numerical analysis provide a wide variety of solution methods and potentially applicable algorithms.

Manual development of a customized data analysis program for any given application problem is a time-consuming and error-prone task. It requires a rare combination of profound expertise in several areas—computational statistics, numerical analysis, software engineering, and of course the application domain itself. The algorithms found in standard libraries need to be customized, optimized, and appropriately packaged before they can be integrated; the model and its specific details usually influence many algorithmic design decisions. Most importantly, the development process for data analysis programs is typically highly iterative: the underlying model is usually changed many times before it is suitable for the application; often the need for these changes becomes apparent only after an initial solution has been implemented and tested on application data. However, since even small changes in the model can lead to entirely different solutions, e.g., requiring a different approximation algorithm, developers are often reluctant to change (and thus improve) the model and settle for sub-optimal solutions.

A program generator can help to solve these problems. It encapsulates a considerable part of the required expertise and thus allows the developers to program in models, thereby increasing their productivity. By automatically synthesizing code from these models, many programming errors are prevented and turn-around times are reduced. We are currently developing AutoBayes, a program generator for data analysis programs. AutoBayes starts from a very high-level description of the data analysis problem in the form of a statistical model and generates imperative programs (e.g., C/C++) through a process which we call *schema-based deductive synthesis*. A schema is a code template with associated semantic constraints which describe the template's applicability. Schemas can be considered as high-level simplifications which are justified by theorems in a formal logic in the domain of Bayesian networks. The schemas are applied recursively but AutoBayes augments this schema-based approach by symbolic-algebraic calculation and simplification to derive closed-form solutions whenever possible. This is a major advantage over other statistical data analysis systems which use slower and possibly less precise numerical approximations even in cases where closed-form solutions exist.

The back-end of AutoBayes is designed to support generation of code for different programming languages (e.g., C, C++, Matlab) and different target systems. Our current version generates C/C++ code which can be linked dynamically into the Octave [Mur97] or Matlab [MLB87] environments; other target systems can be plugged in easily.

This paper describes work in progress; design rationales and some preliminary results of the AutoBayes-project have been reported in [BFP99]. In Section 2, we give a short overview over Bayesian networks and their application for data

analysis. We then proceed with a detailed description of the system architecture, the process of synthesizing the algorithm, and the steps to produce actual code. Section 4 contains a worked example which illustrates the operation of AutoBayes on a small, yet non-trivial example. We compare our approach to related work in Section 5 before we discuss future work in in Section 6.

2 Bayesian Networks and Probabilistic Reasoning

Bayesian networks or *graphical models* are a common representation method in machine learning [Pea88,Bun94,Fre98,Jor99]; AutoBayes uses them to represent the data analysis problem internally. Figure 1 shows the network for the example used throughout this paper.

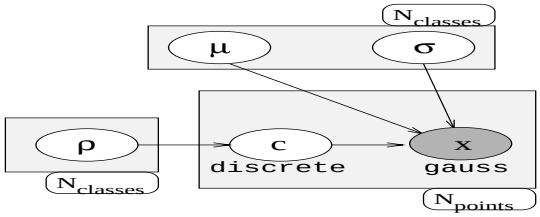

Fig. 1. Bayesian network for the mixture of Gaussians example.

A Bayesian network is a directed, acyclic graph that defines probabilistic dependencies between random variables. Its arcs can sometimes be interpreted as causal links but more precisely the absence of an arc between two vertices denotes the conditional independence of the two random variables, given the values of their parents. Hence, since the example network has no arc between the vertices μ and c, the *joint probability* $P(\mu, c)$ to observe certain values of μ and c at the same time is the same as the product of $P(\mu)$ and $P(c \mid \rho)$, the conditional probability of c given ρ. The network thus superimposes a structure on the global joint probability distribution which can be exploited to optimize probabilistic reasoning. Hence, the example defines the global joint probability $P(x, c, \rho, \sigma, \mu)$ in terms of simpler, possibly conditional probabilities:

$$P(x, c, \rho, \sigma, \mu) = P(\rho) \cdot P(c \,|\, \rho) \cdot P(\mu) \cdot P(\sigma) \cdot P(x \,|\, c, \mu, \sigma)$$

The central theorem of probabilistic reasoning is *Bayes rule*

$$P(h \,|\, d) = \frac{P(d \,|\, h) \cdot P(h)}{P(d)}$$

which expresses the probability $P(h \mid d)$ that hypothesis h holds under given data d in terms of the *likelihood* $P(d \,|\, h)$ and the *prior* $P(h)$; the probability of the data, $P(d)$, is usually only a normalizing constant.

AutoBayes uses an extended version of hybrid Bayesian networks, i.e., nodes can represent discrete as well as continous random variables. Shaded vertices denote known variables, i.e., input data. Distribution information for the variables is attached to the vertices, e.g., the input x is Gaussian distributed. Boxes enclosing a set of vertices denote vectors of independent, co-indexed random variables, e.g., μ and σ are both vectors of size $N_{classes}$ which always occur indexed in the same way. As a consequence, a box around a single vertex denotes the familiar concept of a vector of identically distributed and indexed random variables.

An Example: Mixture of Gaussians

We illustrate how AutoBayes works by means of a simple but realistic classification example. Figure 2 shows the raw input data, a vector of real values. We know that each data point falls into one of three classes; each class is Gaussian distributed with mean μ_i and standard deviation σ_i. The data analysis problem is to infer from that data the relative class frequencies (i.e., how many points belong to each class) and the unknown distribution parameters μ_i and σ_i for each class. Although this example is deliberately rather simple, it already demonstrates the potential of generating data analysis programs; it also illustrates some of the problems.

Figure 3 shows the statistical model in AutoBayes's input language. The model (called "Mixture of Gaussians" – line 1) assumes that each of the data points (there are `n_points` – line 5) belongs to one of `n_classes` classes; here `n_classes` has been set to three (line 3), but `n_points` is left unspecified. Lines 16 and 17 declare the input vector and distributions for the data points[1]. Each point `x(I)` is drawn from a Gaussian distribution `c(I)` with mean `mu(c(I))` and standard deviation `sigma(c(I))`. The unknown distribution parameters can be different for each class; hence, we declare these values as vectors (line 11). The unknown assignment of the points to the classes (i.e., distributions) is represented by the hidden (i.e., not observable) variable `c`; the class probabilities or relative frequencies are given by the also unknown vector `rho` (lines 9–14). Since each point belongs to exactly one class, the sum of the probabilities must be equal to one (line 10). Additional constraints (lines 4,6,7) express further basic assumptions. Finally, we specify the goal inference task (line 19), maximizing the probability $P(x|\rho_i, \mu_i, \sigma_i)$. Due to Bayes' rule, this calculates the most likely values of the parameters of interest, ρ_i, μ_i, and σ_i.

3 System Architecture

The system architecture of AutoBayes (cf. Figure 4) has been designed for high flexibility and modularity granting easy extensibility of the system for interactive refinement of specifications. Here, we describe the main components

[1] Vector indices start with 0 in a C/C++ style.

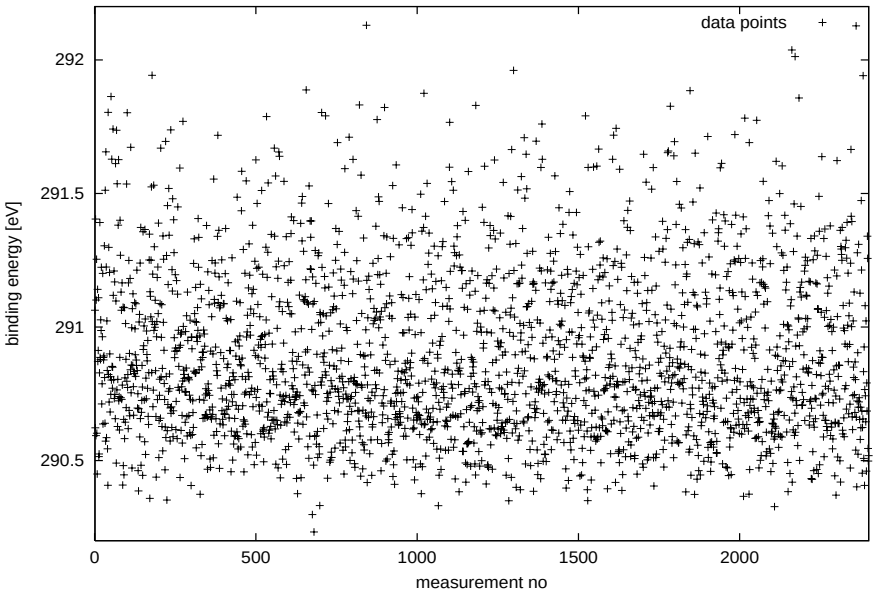

Fig. 2. Artificial input data for the mixture of Gaussians example: 2400 data points in the range $[290.2, 292.2]$. Each point belongs in one of three classes which are Gaussian distributed with $\mu_1 = 290.7, \sigma_1 = 0.15$, $\mu_2 = 291.13, \sigma_2 = 0.18$, and $\mu_3 = 291.55, \sigma_3 = 0.21$. The relative frequencies ρ for the points belonging to the classes are $61\%, 33\%$, and 6%, respectively.

of the system (synthesis kernel, intermediate language, back-end, and generation of artificial data). The entire system has been implemented in SWI-Prolog [SWI99]. For symbolic mathematical calculations, a small rewriting engine has been built on top of Prolog. A set of system utilities (e.g., pretty-printer, graph handling, set representations, I/O functions) facilitates the implementation of AutoBayes. Since AutoBayes requires a combination of symbolic mathematical calculation, rewriting, general purpose operations (e.g., input/output), and reasoning over graphs, Prolog is a reasonable choice as the underlying implementation language. The main reason not to choose a symbolic algebra system as for example Mathematica is its possible unsoundness. During symbolic calculation, simplifications are done by such systems without explicitly stating all assumptions. These unsound transformations can lead to incorrect results and hence incorrect programs. AutoBayes keeps track of all assumptions (e.g., an expression being non-zero) and either discharges them during synthesis or generates assertions to be checked during run-time.

```
 1  model mog as 'Mixture of Gaussians';
 2
 3  const int n_points as 'number of data points'
 4        with 0 < n_points;
 5  const int n_classes := 3 as 'number of classes'
 6        with 0 < n_classes
 7        with n_classes << n_points;
 8
 9  double rho(0..n_classes - 1) as 'class probabilites'
10        with 1 = sum(idx(I, 0, n_classes - 1), rho(I));
11  double mu(0..n_classes - 1), sigma(0..n_classes - 1);
12
13  int c(0..n_points) as 'class assignment vector';
14  c ~ discrete(vec(idx(I, 0, n_classes - 1), rho(I)));
15
16  data double x(0..n_points - 1) as 'data points (known)';
17  x(I) ~ gauss(mu(c(I)),sigma(c(I)));
18
19  max pr(x | {rho,mu,sigma}) wrt {rho, mu, sigma};
```

Fig. 3. AutoBayes-specification for the mixture of Gaussians example. Line numbers have been added for reference in the text. Keywords are underlined.

3.1 Synthesis Kernel

The synthesis kernel takes the model specification and builds an initial Bayesian network. Each variable declaration in the model corresponds directly to a network node. Each distribution declaration, e.g., $x \sim \text{gauss}(\Theta)$, induces edges from the distribution parameters Θ to the node corresponding to the random variable x; these edges reflect the dependency of the (random) values of x on the values of the parameters Θ. Building the network is relatively straightforward and requires no sophisticated dataflow analysis because the model is purely declarative. However, Θ needs to be flattened, i.e., nested random variables need to be lifted and fresh index variables need to be introduced in their place in order to represent the dependencies properly. Hence, the example declaration $x(i) \sim \text{gauss}(\mu(c(i)), \sigma(c(i)))$ induces not only the two obvious edges but three: $\mu(j) \longrightarrow x(i), \sigma(j) \longrightarrow x(i)$, and $c(i) \longrightarrow x(i)$ (cf. Figure 1). Note that x and c are still co-indexed but that each $x(i)$ now depends on all $\mu(\cdot)$ and $\sigma(\cdot)$, reflecting the unknown values of their original indices $c(i)$. A compact representation of the indexed nodes and their dependencies is achieved by using Prolog-variables to represent index variables.

Synthesis proceeds from this initial network and the original probabilistic inference task by exhaustive application of *schemas*. A schema can be understood as an "intelligent macro": it comprises a *pattern*, a *parameterized code template*, and a set of preconditions or *applicability constraints*. An example will be shown below. The pattern and code template are similar to the left- and right-hand side of a traditional macro definition; they comprise the syntactic part of the schema. Schema-guided synthesis, however, is not just macro expansion. Different schemas can match the same pattern, possibly in different ways. AutoBayes

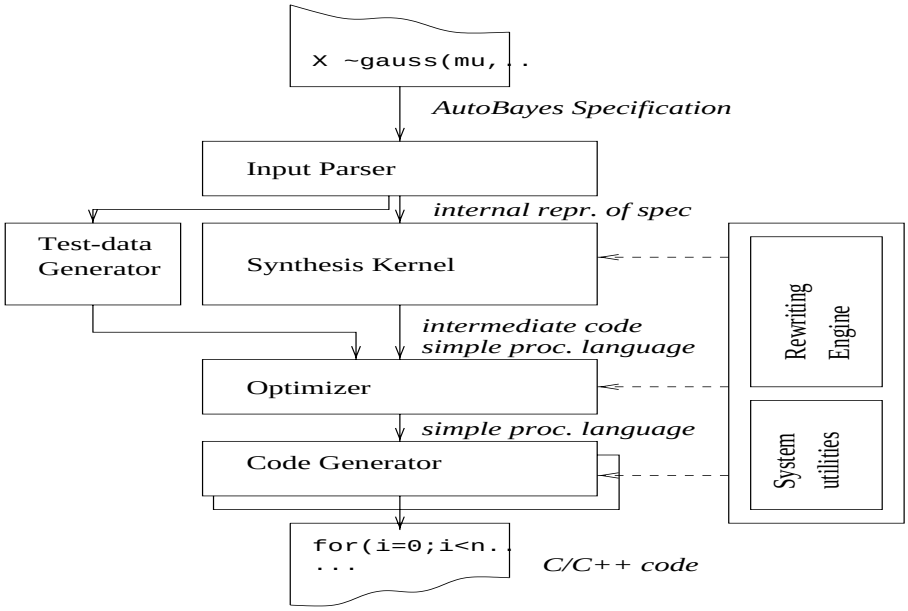

Fig. 4. System architecture for AutoBayes.

covers this search space by a depth-first, left-to-right search, with backtracking over possible multiple solutions.

The constraints refine the semantics of the schema: a schema can be understood as an axiom which asserts that the program (i.e., the appropriately instantiated template) solves the probabilistic inference task specified by the pattern *if* the constraints are satisfied; however, checking the constraints may instantiate the template parameters further. The search process mentioned above is thus a proof search; the proof is constructive in the sense that it actually generates a program (the *witness*) and does not just assert its existence.

Network decomposition schemas. AutoBayes uses four different kinds of schema. *Network decomposition* schemas are encodings of independence theorems for Bayesian networks. They describe how a probabilistic inference task over a given network can be decomposed equivalently into several simpler tasks over simpler networks and, hence, how a complex data analysis program can be composed from simpler components. The applicability constraints for these schemas can be checked by pure graph reasoning. Consider for example the following decomposition theorem.

Theorem 1 ([BFP99]) *Let U, V be sets of vertices in a Bayesian network with $U \cap V = \emptyset$. Then $V \cap \mathit{descendants}(U) = \emptyset$ and $\mathit{parents}(U) \subseteq V$ hold iff*

$$P(U \mid V) = P(U \mid \mathit{parents}(U)) = \prod_{u \in U} P(u \mid \mathit{parents}(u))$$

The theorem allows us to simplify the conditional probability $P(U \mid V)$ into $P(U \mid \mathit{parents}(U))$ (i.e., it allows us to ignore all assumptions not reflected in the network by incoming edges) and then further into a finite product of atomic probabilities (i.e., each variable depends only on the parameters of its associated distribution), provided that the applicability constraints hold over the network; here, $\mathit{descendants}(U)$ is defined as $\mathit{children}^*(U) - U$ with $\mathit{children}^*$ being the full graph reachability relation. This theorem then induces the following schema for maximizing the probability $P(U|V)$ with respect to a set of variables X.

$$
\begin{aligned}
&\text{schema(}\ \max\ P(U|V)\ \text{wrt}\ X,\ \mathit{Template}\ \text{)}\ \ \texttt{:-}\\
&\qquad U \cap V = \emptyset\\
&\quad \wedge\ U \cap \mathit{descendants}(V) = \emptyset\\
&\quad \wedge\ \mathit{parents}(U) \subseteq V\\
&\quad \rightarrow \mathit{Template} = \underline{\texttt{begin}}\\
&\qquad\qquad \langle \max\ P(\{u\}|\mathit{parents}(\{u\}))\ \text{wrt}\ X \mid u \in U \rangle\\
&\qquad\qquad \underline{\texttt{end}}
\end{aligned}
$$

The schemas are written as Prolog-rules. During the search for applicable schemas, pattern-matching with the rule head (first line) is tried. When the match succeeds, the variables (U, V, X) are bound, and the body of the rule (separated by the `:-`) is processed. Here, the body is a logical implication. The implication's antecedents directly encode the applicability constraints as AutoBayes's symbolic reasoning engine contains an operationalization of the graph predicates. The schema's code template consists of a sequence of simpler maximization tasks. Their ordering is irrelevant because the $u \in U$ are independent of each other; this is a consequence of the applicability constraints. A number of similar decomposition theorems have been developed in probability theory; AutoBayes currently includes three different schemas based on such theorems, with the one shown above being by far the simplest.

Formula decomposition schemas. *Formula decomposition* schemas are similar to the network decomposition schemas above but they work on complex formulae instead of a single probability. A typical schema in this class is index decomposition. It applies to an inference task for a formula which contains possibly multiple occurrences of probabilities involving vectors and "unrolls" this task into a loop over the simpler inference task for a single vector element. Most of the applicability constraints for index decomposition can still be checked by graph reasoning but some checks involving the formula structure require proper symbolic reasoning.

Statistical algorithm schemas. Proper *statistical algorithm* schemas are also graph-based but they are not simple consequences of the independence theorems. Instead, their correctness is proven independently, or they are just empirically validated during construction of the domain theory. These schemas involve larger modifications of the graph, e.g., introduction of new nodes with known values, and storing the results of intermediate calculations. These schemas thus enable the further application of the decomposition schemas; however, they are much more intricate and less theorem-like. They also have much larger code templates

associated and they can require substantial symbolic reasoning during instantiation. AutoBayes currently implements two such algorithms which are known in the literature as *expectation maximization* (or simply *EM algorithm* [DLR77]) and *k-Means*, respectively. Both schemas are applicable to general mixture models which underpin many classification tasks similar to the running example.

Numerical algorithm schemas. The graph-based reasoning continues until all conditional probabilities $P(U \mid V)$ have been converted into atomic form, i.e., V are all parameters of U's distribution. These can then be replaced by the appropriately instantiated probability density functions. AutoBayes's domain theory contains rewrite rules for the most common probability density functions, e.g., Gaussian and Poisson distributions. With this rewrite step the original probabilistic inference task becomes a pure optimization problem which can be solved either symbolically or numerically. AutoBayes first attempts to find closed-form symbolic solutions, which are much more time-efficient during run-time than the usually iterative numeric approximation algorithms. In order to solve the optimzation problem, AutoBayes symbolically differentiates the formula with respect to the optimization variables, sets the result to zero and tries to solve this system of simultaneous equations. Symbolic differentiation is implemented as a term rewrite system; however, the need to check for whether a term depends on the variable that the derivative is taken with respect to implies that some rules are conditional rewrite rules. Equation solving currently employs only a variant of Gaussian variable elimination: whenever a variable can be isolated modulo the symbolic model constants, the remaining equation is solved by a polynomial solver.

If no symbolic solution can be found, AutoBayes applies iterative numerical algorithm schemas, e.g., the Newton-Raphson method or the Nelder-Mead simplex algorithm. Such algorithms are also provided by general-purpose numeric libraries, e.g., [PF+92], but program generation can substantially improve this black-box style reuse, because it can instantiate actual parameters symbolically and evaluate the inlined expressions partially. This provides further optimization opportunities, often in the inner loops of the algorithms.

Control. During synthesis, these schemas are tried exhaustively in a left-to-right, depth-first manner. Whenever a dead end is encountered (i.e., no schema is applicable), AutoBayes backtracks. This search allows AutoBayes to generate program variants if more than one of the schemas are applicable and opens up possibilities to generate multiple solutions for the same problem, which then can be assessed using tests on the given data.

3.2 Generating Explanations

Certification procedures for safety-critical applications (e.g., in aircrafts or spacecrafts) often mandate manual code inspection. This inspection requires that the code is readable and well documented. Even for programs not subject to certification, understandability is a strong requirement as manual modifications are often neccessary, e.g., for performance tuning or system integration. However,

existing program generators often produce code that is hard to read and understand. In order to overcome this problem, AutoBayes generates explanations along with the programs which make the synthesis process more transparent and provide traceability from the generated program back to the model specifiation.

AutoBayes generates heavily commented code: approximately a third of the output is automatically generated comments (cf. Figure 8 for an example). This is achieved by embedding documentation templates into the code templates. Future versions of AutoBayes will not only generate fully documented code; we are aiming at producing a detailed design-document for the generated code. This document will also show the "synthesis decisions" made by AutoBayes (e.g., which algorithm schema has been used) and the reasons which led to them. Open proof obligations and model assumptions will be laid out clearly.

Reliability of generated code entails that the code is robust (e.g., robustness against erroneous inputs or sensor failures). Thus, all assumptions from the specification or made by AutoBayes which cannot be discharged during synthesis are brought to the user's attention and are listed in the documentation. Important assumptions which can be checked efficiently during run-time are converted into assertions which are inserted into the code (e.g., $N_{classes} < N_{points}$).

3.3 Intermediate Language

The synthesis kernel of AutoBayes generates code for an intermediate language before code for the actual target system is produced. This intermediate language is a simple procedural language with several domain-specific extensions (e.g., for convergence loops, vector normalization, and simultaneous vector assignment). Each statement of the intermediate language can be annotated. In the current version, annotations carry the generated explanations. In future versions annotations will also be used to guide optimization and to carry out automatic instrumentation of the generated code for evaluation and testing purposes.

Using an intermediate language offers major benefits because it allows to perform code optimization independently from the selected target language and target system without excessive overhead. For example, we are able to extract loop-independent expressions without having to apply data-flow analysis to the generated code, because the structure of the loops is known from the instantiated algorithm schema.

The intermediate code is close enough to allow for a simple translation into the target language (e.g., C, C++, Matlab). The additional domain-specific constructs facilitate target-specific transformations. For example, the language construct for calculating the sum of array elements (`sum`) can be converted into a usual `for`-loop (C, C++), an iterator construct (e.g., when using sparse matrices), or a direct call of a summation-operator (e.g., when generating interpreted Matlab code).

3.4 Backend and Code generation

The actual code-generator can be adapted easily to a specific target language and a given environment. With the help of rewrite rules all constructs of the intermediate language are transformed into constructs of the target language and printed using a simple pretty-printer. On this target-specific level, another set of optimization steps are performed (e.g., replacing of E^{-1} by $1/E$, or E^2 by $E \times E$ for an expression E). Standard optimizations (e.g., evaluation of constant expressions) are left for the subsequent compilation phase—there is no need to perform the same optimization steps as any modern compiler. The back-end also generates code for interfacing the algorithm with the target system, and to check for correct types of arguments.

Our current prototype produces C++-code for Octave [Mur97] and C-code for Matlab [MLB87]. Future work will include code-generators for design-tools for embedded systems, e.g., ControlShell [Con99] or MatrixX [Aut99].

3.5 Synthesis of Artificial Test Data

The given input specification contains enough information to generate artificial data with properties corresponding to the specified statistical model. AutoBayes is capable of generating code producing artificial data. For this task we use the same underlying machinery and back-end as described above. This feature of AutoBayes offers several benefits: using artificial data, the performance (e.g., speed or convergence) of the generated code can be evaluated and assessed; comparisons between different algorithm schemas can be made easily. Artificial data can also be used to test the generated code. For example, by using different sets of parameters, the behavior of the generated analysis algorithm can be tested for stability.

4 A Worked Example

In this section, we discuss synthesis and execution of the example described in Section 2. The specification in Figure 3 comprises the entire input to AutoBayes. After parsing this specification, AutoBayes generates the dependency graph (see Figure 1) and tries to break it down into independent parts. When trying to solve the optimization problem, the system fails to find a closed-form symbolic solution. Therefore, the EM algorithm schema is tried and instantiations are sought. This algorithm schema consists of an iterative loop which has to be executed until a convergence criterion is met. Within this loop, new estimates for ρ, σ, μ are calculated and compared to the old values. When the difference becomes small enough, the loop can be exited (cf. Figure 8).

For our example, AutoBayes generates a C++ file consisting of 477 lines (including comments and separation lines). This code is then compiled into a dynamically linkable function for Octave. Thus, when invoking the function "mog" (line 1 of the specification) from inside Octave, our compiled C++ code is invoked automatically. As shown in Figure 5, AutoBayes also synthesizes a usage

line and produces a short help-text. The entire synthesis process of AutoBayes, including compilation of the generated C++ code takes about 35s on a G3 powerbook under Linux.

The following results have been obtained using artificial data. Starting with a total of 2400 points falling into 3 classes (cf. Figure 2), the algorithm searches for the values of `mu, sigma`, and `rho` for each class. For the final result, this run required 1163 iteration steps, taking approximately 54 sec on a G3 notebook[2]. The convergence, i.e., the normalized change of the parameters to be optimized during each iteration cycle, is shown in Figure 6.[3] AutoBayes automatically instruments the generated code to produce these run-time figures for debugging and testing purposes.

```
octave:2> mog
usage: [vector mu,vector rho,vector sigma] = mog(vector x)

octave:3> help mog
mog is a builtin function

Mixture of Gaussians. Maximize the conditional probability
pr([c,x]|[rho,mu,sigma]) w.r.t. [rho, mu, sigma ], given
data x and n_classes=3.
...
octave:4> x = [ ... ];
octave:4> [mu,rho,sigma] = mog(x)
mu =

  291.12
  291.28
  290.69
...
```

Fig. 5. Octave sample session using code (function "mog") generated by AutoBayes.

Although this example has been run with artificial data, there are several real applications for this kind of model. For example, when molecules (or atoms) in a gas are excited with a specific energy (e.g., light from a laser), they can absorb this energy by excitation or by emission by one or more of their electrons, respectively. This basic mechanism generates spectral lines, e.g., in the light of stars. Single atoms usually have sharp, well-defined spectral lines but molecules which are more complex (e.g., CH_4 or NH_3) can have several peaks of binding

[2] This figure can change from run to run, since the algorithm starts with a random initial class assignment for each data point.

[3] This algorithm does not converge monotonically. It can reach some local minimum, from which it has to move away by increasing the error again. After some ups and downs the global minimum (i.e., an optimal estimate for the parameters) is reached and the loop ends. This behavior is typical for parameter estimation processes, e.g., as found in artificial Neural Networks [MR88].

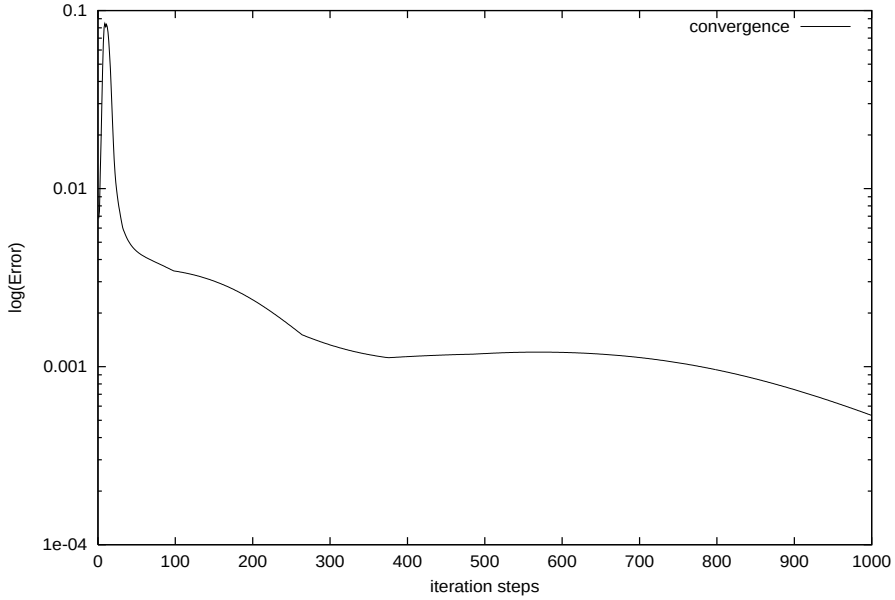

Fig. 6. Convergence behavior: differences between old and new parameters (log-scale) over iteration step. Only the first 1000 iteration cycles are shown.

energy, depending on their internal configuration. Thus, they can absorb (or emit) energy at different levels.

The example here (cf. Figure 7) is taken from [Ber79] and shows the spectrum of the energy of emitted photoelectrons that is directly related to the excess energy of the photon over the photoionization potential of the molecule CH_4 (for details see [Ber79], caption of Figure 67). In a simple statistical model, each of the peaks is assumed to be Gaussian distributed and the percentage of molecules being in a specific configuration is known. When we measure the binding energy for a large number of (unknown) molecules, we obtain a set of data, similar to that shown in Figure 2 above. If we suspect that the molecules might be CH_4 which has 3 distinct configurations, we can use the generated code to classify the data into these three classes and to obtain the parameters. The histogram of the data is shown in Figure 7, super-imposed with Gaussian curves using the parameter values as estimated by the program.

5 Related Work

Work related to AutoBayes appears mainly in two different fields. In the first field, statistics, there is a long tradition of composing programs from library components but there are only a few, recent attempts to achieve a similar degree of

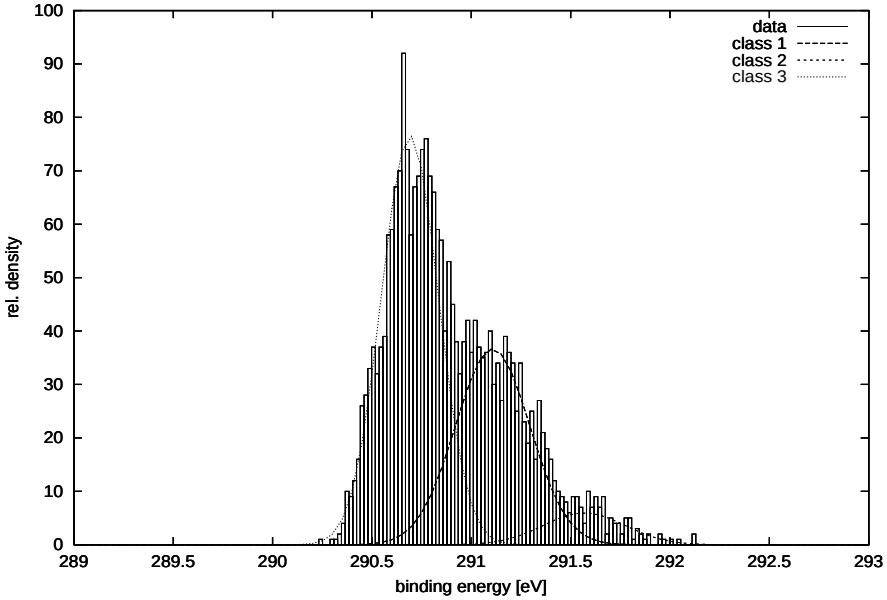

Fig. 7. Histogram (spectrum) of the artificial test data (Figure 2, number of bins = 100) and Gaussian distributions as obtained by running the generated code.

automation as AutoBayes does. The Bayes Net Toolbox [Mur00] is a Matlab-extension which allows users to program models; it provides several Bayesian inference algorithms which are attached to the nodes of the network. However, the Toolbox is a purely interpretive system and does not generate programs. The widely used Bugs-system [TSG92] also allows users to program in models but it uses yet another, entirely different execution model: instead of executing library code or generating customized programs, it interprets the statistical model using Gibbs sampling, a universal—but less efficient—Bayesian inference technique. Bugs, or more precisely, Gibbs sampling, could thus be integrated into AutoBayes as an algorithm schema.

The other field, deductive synthesis, is still an active research area. Some systems, however, have already been applied to real-world problems. The Amphion system [SW+94] has been used to assemble programs for celestial mechanics from a library of Fortran components, for example the simulation of a Saturn fly-by. Amphion is more component-oriented than AutoBayes, i.e., the generated programs are linear sequences of subroutine calls into the library. It uses a full-fledged theorem prover for first-order logic and extracts the program from the proof. [EM98] describes a system for the deductive synthesis of numerical simulation programs. This system also starts from a high-level specification of a mathematical model—in this case a system of differential equations—but is

again more component-oriented than AutoBayes and does not use symbolic-algebraic reasoning. Planware [BG+98] (which grew out of the Kids system [Smi90]) synthesizes schedulers for military logistics problems. It is built on the concept of an algorithm theory which can be considered as an explicit hierarchy of schemas.

[Big99] presents a short classification of generator techniques (albeit cast in terms of their reuse effects). AutoBayes falls most closely into the category of inference-based generators but also exhibits some aspects of pattern-directed and reorganizing generators, e.g., the typical staging of the schemas into multiple levels.

6 Conclusions

We have presented AutoBayes, a prototype system that automatically generates data analysis programs from specifications in the form of statistical models. AutoBayes is based on deductive, schema-guided synthesis. After constructing the initial Bayesian network from the model, a variety of different schemas are tried exhaustively. These schemas are guarded by applicability constraints and contain code-blocks which are instantiated. By way of an intermediate language, AutoBayes generates executable, optimized code for a target system. The current version of AutoBayes produces C/C++-code for dynamic linking into Octave and Matlab; future versions of AutoBayes will include code generation for sparse matrices and for design-tools for embedded systems (e.g., ControlShell).

We have applied AutoBayes successfully to a number of textbook problems where it was able to find closed-form solutions equivalent to those in the textbooks. The largest iterative solution generated so far comprises 477 lines of C++ code. Synthesis times (including compilation of the generated code) are generally well below one minute on standard hardware. We are currently testing AutoBayes in two major case studies concerning data analysis tasks for finding extra-solar planets either by measuring dips in the luminosity of stars [KB+00], or by measuring Doppler effects [MB97], respectively. Both projects required substantial effort to manually set up data analysis programs. Our goal for the near future is to demonstrate AutoBayes's capability to handle major subproblems (e.g., the CCD-sensor registration problem) arising in these projects.

AutoBayes has two unique features which result from using program generation (instead of compilation) and which make it more powerful and versatile for its application domain than other tools and statistical libraries. First, AutoBayes generates efficient procedural code from a high-level, declarative specification without any notion of data- or control-flow. Thus, it covers a relatively large semantic gap. Second, by combining schema-guided synthesis with symbolic calculation, AutoBayes is capable of finding closed-form solutions for many problems. Thus, the generated code for these kinds of problems is extremely efficient and accurate, because it does not rely on numeric approximations.

The explanation technique provides further benefits, especially in safety-critical areas. Code is not only documented for human understanding, but assumptions made in the specification and during synthesis are checked by assertions during run-time. This makes the generated code more robust with respect to erroneous inputs or sensor failures.

AutoBayes is still an experimental prototype and has to be extended in several ways before it can be released to users. In particular, further schemas have to be added and the expressiveness of the kernel with respect to the model descriptions has to be increased. However, since the schemas cannot be derived automatically from the underlying theorems, more machine support for this manual domain engineering process may become necessary, e.g., type-checking of the schemas [Bjø99]. Nevertheless, we are confident that the paradigm of schema-guided synthesis is an appropriate approach to program generation in this domain which will lead to a powerful yet easy-to-use tool.

Acknowledgements: Wray Buntine contributed much to the initial development of AutoBayes and the first version of the prototype. We would like to thank the anonymous reviewers for their helpful comments.

References

[Aut99] MatrixX: AutoCode Product Overview. ISI, 1999. `http://www.isi.com`.

[Ber79] J. Berkowitz. *Photoabsorption, photoionization, and photoelectron spectroscopy*. Academic Press, 1979.

[BFP99] W. L. Buntine, B. Fischer, and T. Pressburger. "Towards Automated Synthesis of Data Mining Programs". In S. Chaudhuri and D. Madigan, (eds.), *Proc. 5th Intl. Conf. Knowledge Discovery and Data Mining*, pp. 372–376, San Diego, CA, August 15–18 1999. ACM Press.

[BG+98] L. Blaine, L.-M. Gilham, J. Liu, D. R. Smith, and S. Westfold. "Planware – Domain-Specific Synthesis of High-Performance Schedulers". In D. F. Redmiles and B. Nuseibeh, (eds.), *Proc. 13th Intl. Conf. Automated Software Engineering*, pp. 270–280, Honolulu, Hawaii, October 1998. IEEE Comp. Soc. Press.

[BH99] M. Berthold and D. J. Hand, (eds.). *Intelligent Data Analysis—An introduction*. Springer, Berlin, 1999.

[Big99] T. J. Biggerstaff. "Reuse Technologies and Their Niches". In D. Garlan and J. Kramer, (eds.), *Proc. 21th Intl. Conf. Software Engineering*, pp. 613–614, Los Angeles, CA, May 1999. ACM Press. Extended abstract.

[Bjø99] N. S. Bjørner. "Type checking meta programs". In *Workshop on Logical Frameworks and Meta-languages*, Paris, France, 1999.

[Bun94] W. L. Buntine. "Operations for learning with graphical models". *J. AI Research*, **2**:159–225, 1994.

[Con99] ControlShell. RTI Real-Time Innovations, 1999. `http://www.rti.com`.

[DLR77] A. P. Dempster, N. M. Laird, and D. B. Rubin. "Maximum likelihood from incomplete data via the EM algorithm (with discussion)". *J. of the Royal Statistical Society series B*, **39**:1–38, 1977.

[EM98] T. Ellman and T. Murata. "Deductive Synthesis of Numerical Simulation Programs from Networks of Algebraic and Ordinary Differential Equations". *Automated Software Engineering*, **5**(3):291–319, 1998.

[Fre98] B. J. Frey. *Graphical Models for Machine Learning and Digital Communication.* MIT Press, Cambridge, MA, 1998.

[Jor99] M. I. Jordan, (ed.). *Learning in Graphical Models.* MIT Press, Cambridge, MA, 1999.

[KB+00] D. G. Koch, W. Borucki, E. Dunham, J. Jenkins, L. Webster, and F. Witteborn. "CCD Photometry Tests for a Mission to Detect Earth-size Planets in the Extended Solar Neighborhood". In *Proceedings SPIE Conference on UV, Optical, and IR Space Telescopes and Instruments*, 2000.

[MB97] G. W. Marcy and R. P. Butler. "Extrasolar Planets Detected by the Doppler Technique". In *Proceedings of Workshop on Brown Dwarfs and Extrasolar Planets*, 1997.

[MLB87] C. B. Moler, J. N. Little, and S. Bangert. *PC-Matlab Users Guide.* Cochituate Place, 24 Prime Park Way, Natick, MA, USA, 1987.

[MR88] J. L. McClelland and D. E. Rumelhart. *Explorations in Parallel Distributed Processing.* MIT Press, 1988.

[Mur97] M. Murphy. "Octave: A Free, High-Level Language for Mathematics". *Linux Journal*, **39**, July 1997.

[Mur00] K. Murphy. Bayes Net Toolbox 2.0 for Matlab 5, 2000. `http://www.cs.berkeley.edu/~murphyk/Bayes/bnt.html`.

[Pea88] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann Publishers, San Mateo, CA, USA, 1988.

[PF+92] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C.* Cambridge Univ. Press, Cambridge, UK, 2nd. edition, 1992.

[Smi90] D. R. Smith. "KIDS: A Semi-Automatic Program Development System". *IEEE Trans. Software Engineering*, **16**(9):1024–1043, September 1990.

[SW+94] M. Stickel, R. Waldinger, M. Lowry, T. Pressburger, and I. Underwood. "Deductive Composition of Astronomical Software from Subroutine Libraries". In A. Bundy, (ed.), *Proc. 12th Intl. Conf. Automated Deduction, Lect. Notes Artifical Intelligence* **814**, pp. 341–355, Nancy, June-July 1994. Springer.

[SWI99] SWI Prolog, 1999. `http://swi.psy.uva.nl/projects/SWI-Prolog/`.

[TSG92] A. Thomas, D. J. Spiegelhalter, and W. R. Gilks. "BUGS: A program to perform Bayesian inference using Gibbs sampling". In J. M. Bernardo, J. O. Berger, A. P. Dawid, and A. F. M. Smith, (eds.), *Bayesian Statistics 4*, pp. 837–842. Oxford Univ. Press, 1992.

```
// Mixture of Gaussians
proc(mog) {
const:  int n_classes := 3;                    // Number of classes
        int n_points := size(x, 1);            // Number of data points
input:  double x[0:n_points - 1];
output: double mu[0:n_classes-1],rho[0:n_classes-1],sigma[0:n_classes-1];
local: ...
{ ...
  // Initialization
  // Randomize the hidden variable c
  for( [idx(pv64, 0, n_points - 1)])
    c(pv64) := random_int(0, n_classes - 1);
  // Initialize the local distribution; the initialization is "sharp",
  // i.e., q1 is set to zero almost everywhere and to one at the index
  // positions determined by the initial values of the hidden variable.
  for( [idx(pv154, 0, n_points - 1), idx(pv155, 0, n_classes - 1)])
    q1(pv154, pv155) := 0;
  for( [idx(pv156, 0, n_points - 1)])
    q1(pv156, c(pv156)) := 1;

  // EM-loop
  while( converging([vector([idx(pv157, 0, n_classes-1)], rho(pv157)),
            vector([idx(pv158, 0, n_classes-1)], mu(pv158)),
            vector([idx(pv159, 0, n_classes-1)], sigma(pv159))]) )
    {
      // Decomposition I;
      // the problem to optimize the conditional probability
      // pr([c, x] | [rho, mu, sigma]) w.r.t. the variables rho, mu,
      // and sigma can under the given dependencies by Bayes rule be
      // decomposed into independent subproblems.
          ...
      // using the Lagrange-multiplier l1.
      l1 := sum([idx(pv68, 0, n_classes - 1)],
                sum([idx(pv66, 0, n_points - 1)], q1(pv66, pv68)));
      for( [idx(pv68, 0, n_classes - 1)])
        rho(pv68) := l1 ** -1 * sum([idx(pv66, 0, n_points - 1)],
                     q1(pv66, pv68));
      // The conditional probability pr([x] | [sigma, mu, c]) is
      // under the given dependencies by Bayes rule equivalent to
      // prod([idx(pv126, 0, n_points-1)],
      //      pr([x(pv126)] | [c(pv126), mu, sigma]))
      // The probability occuring here is atomic and can be
      // replaced by the respective probability density function.
      ...
      for( [idx(pv64, 0, n_points-1), idx(pv65, 0, n_classes-1)])
        q1(pv64, pv65) := select(norm([idx(pv163, 0, n_classes-1)],
                  exp(-1 / 2 * (-1 * mu(pv163) + x(pv64)) ** 2 *
                  sigma(pv163) ** -2) * rho(pv163) * 2 ** (-1 / 2) *
                  pi ** (-1 / 2) * sigma(pv163) ** -1), [pv65]);
    } } }
```

Fig. 8. Pseudo-code for the Mixture of Gaussians example (excerpts).