Code You Are Happy to Paste: An Algorithmic Dictionary of Exponential Families

Olivier Schwander

Département Signal et Systèmes Électroniques (SSE),
Laboratoire des Signaux et Systèmes (L2S),
CNRS-SUPELEC-PARIS SUD, France
olivier.schwander@supelec.fr

Abstract. We describe a library and a companion website designed to ease the usage of exponential families in various programming languages. Implementation of mathematical formulas in computer programs is often error-prone, difficult to debug and difficult to read afterwards. Moreover, this implementation is heavily dependent of the programming language used and often needs an important knowledge of the idioms of the language. In our system, formulas are described in a high-level language and mechanically exported to the chosen target language and a LaTeX export allows to quickly review correctness of formulas. Although our system is not limited by design to exponential families, we focus on this kind of formulas since they are of great interest for machine learning and statistical modeling applications. Besides, exponential families are a good usecase of our dictionary: among other usages, they may be used with generic algorithms for mixture models such as Bregman Soft Clustering, in which case lots of formulas from the canonical decomposition of the family need to be implemented. We thus illustrate our library by generating code which can be plugged into generic Expectation-Maximization schemes written in multiple languages.

1 Introduction

Except rare theoretical breakthroughs, machine learning research often needs to be validated with experiments and implementations in some programming language (common languages for this use are typically Matlab, Python, R, C, C++ or even Fortran). This implementation step goes through the translation of the mathematical formulas appearing in the new method into computer code. Although one may expect this translation to be straightforward, it usually needs some non trivial knowledge about the used language: the syntax for creating matrices and vectors and the syntax to access elements; the mathematical operators and common functions (like sqrt, exp, sin); the name of mathematical constants; the availability of special mathematical functions (like Γ, erf, etc); the various headers needed to enable access to the previous features (`#include`, `import`, etc); and finally the options to give to the compiler or the interpreter (to locate libraries and files).

T. Calders et al. (Eds.): ECML PKDD 2014, Part III, LNCS 8726, pp. 51–65, 2014.

Although the general structure of any implementation seems to be similar, a lot of small differences appear between languages. We can study these subtle differences by looking at the implementation of the probability density function of the Gaussian distribution in three different languages: Python (Fig. 1), Matlab (Fig. 2) and C (Fig. 3). Since these three versions come from the reference libraries for numerical computation, they are supposed to be idiomatic and respectful of the usages of each language. We notice the square is made with three different syntax: `**` in Python, `^` in Matlab and `u*u` (where `u` is a temporary variable holding the quantity to square) in C. We can also remark the various needs regarding the headers: various `import` for `sqrt`, `exp` and `pi` in Python, nothing in Matlab and one `include` in C for `sqrt`, `exp`, `M_PI` and `fabs` (the use of an absolute value around the standard deviation is rather surprising here, but this is beyond the scope of our study).

```
   7 import math
  24 from numpy import exp
  28 from numpy import pi
2112 _norm_pdf_C = math.sqrt(2*pi)
2116 def _norm_pdf(x):
2117     return exp(-x**2/2.0) / _norm_pdf_C
```

Fig. 1. Gaussian in Python (extract from the file `scipy/stats/distributions.py` from the library Scipy 0.13.3)

```
30 function pdf = stdnormal_pdf (x)
40   pdf = (2 * pi)^(- 1/2) * exp (- x .^ 2 / 2);
42 endfunction
```

Fig. 2. Gaussian in Matlab (extract from the file `scripts/statistics/distributions/normpdf.m` from Octave 3.8.1 since the Matlab sources are not available; some lines which check parameters are removed)

This language-specific knowledge is often not problematic at first glance since one tends to use a well-known language for the first experiments of a new method but may become a problem if some parts need to be rewritten in another language for performance reasons or to collaborate with other people using other languages. It also renders more difficult the path between a first research prototype and a real-scale application. Finally, and perhaps more importantly, the source code implementing the formula is often difficult to read: for the original programmer, bugs and mistakes are harder to find and for a newcomer wanting to study the implementation, the code is barely understandable and nearly impossible to use in another application without a lot of work.

```
 22 #include <math.h>
118 double
119 gsl_ran_gaussian_pdf (const double x, const double sigma)
120 {
121  double u = x / fabs (sigma);
122  double p = (1 / (sqrt (2 * M_PI) * fabs (sigma))) * exp (-u * u / 2);
123  return p;
124 }
```

Fig. 3. Gaussian in C (extract from the file `randist/gauss.c` from the Gnu Scientific Library (GSL) 1.16)

The library introduced in this paper allows to describe mathematical formulas in a programming language-agnostic way: the work of translating formulas into a computer-understandable implementation only needs to be made once, facilitating the choice of the most well-suited programming language. A first prototype may be exported to Python and then to C in order to work on a large real life dataset. Another researcher may generate Matlab code and then plug the formula into its own code base. And an engineer in a company may just take the C export and use it for an industrial application or the company may also design its own exporter backend to generate code suited to proprietary internal tools. Since the description language is not really more readable than a programming language, a LaTeX export is provided, allowing to easily proofread the formula. This library is aimed at any people who may want to use mathematical formulas inside computer code and can be used at hand with copy-pasting or in a more clean way by integrating it in a build process.

Beside the library itself, we also present *Code-Formula*, a web application demonstrating our library, which is designed both for educational purpose and to offer an encyclopedia of mathematical functions which can be picked-up when one needs an out-of-the-box implementation of a mathematical formula. This website is inspired by other online dictionaries of mathematical objects, like the Online Encyclopedia of Integer Sequences [8], the Digital Library of Mathematical Functions [5] or the Dynamic Dictionary of Mathematical Functions [2], but to the best of our knowledge, it is the first one focusing on the algorithmic side instead of mathematical properties.

Although the previous remarks can apply to a wide variety of formulas from mathematical science, computer science, physics or engineering fields, we chose to limit ourselves to a dictionary of exponential families for some reasons: first, it is better at first sight to limit the goals of the project to a reasonable set of objects; second, exponential families are widely used in a large variety of fields, including, but not limited to, machine learning; last, and perhaps most importantly, in the recent years, a lot of work has been devoted to the design of generic algorithms for mixtures of exponential families, where the precise family is a parameter of the algorithm, and a few implementations have been worked-on, in Java (`jMEF` [4]), in Python (`pyMEF` [7]), in C (`libmef` [6]) or even in R.

Each of these implementations has been confronted to the same kind of work: translating formulas into code. We hope that using our library and website, the implementation of such libraries may be done in a semi-automatic way.

This article in organized as follows: after this introduction detailing motivation and goals of the project, the architecture of the library is described. Then, a few examples of exponential families described using our library are given along the utilization of exported code to plug into a generic Expectation-Maximization method for mixture of exponential families. Finally, the website containing the encyclopedia itself is described.

2 Architecture

2.1 General view

The general architecture of the system is described in Fig. 4: mathematical formulas are described in a high-level frontend, then processed by the code of the library and finally passed to the backends which are in charge of generating programming code. Currently, the language used in the fronted is the same as for the library itself, that is OCaml (but very little knowledge of this language is needed to effectively write formulas). This choice has been made for facility reasons, avoiding the need of writing a parser for a domain specific language and because OCaml, although not well-known in the machine learning field, is well suited for this kind of task. Although a LaTeX frontend may look appealing, this is not feasible for two reasons: first a LaTeX parser is nearly impossible to write, even for the subset of the language expressing the mathematical formulas; second LaTeX formulas carry very few semantic, since the language is designed for display, not for computation.

The core part of the library provides a set of tools to manipulate the formulas, like changing the names of variables inside a formula but in the future other frontends may be added. So far, four backends are available: Python, Matlab, C and LaTeX, the later allowing to easily proofread formulas and to use them directly in publications or documentation.

The source code of this library, called `Formula`, can be browsed online on `http://hub.darcs.net/oschwand/formula` and downloaded on the webpage related to this article: `http://www.lix.polytechnique.fr/~schwander/ecml2014/`.

2.2 Frontend

The frontend is responsible for the translation between a human-understandable description of the formula into a data structure representing the formula which can be passed to export backends. If we stick to the example given in the introduction, that is, the probability density function of the Gaussian distribution

$$f(\sigma, \mu, x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{\sigma^2}\right) \tag{1}$$

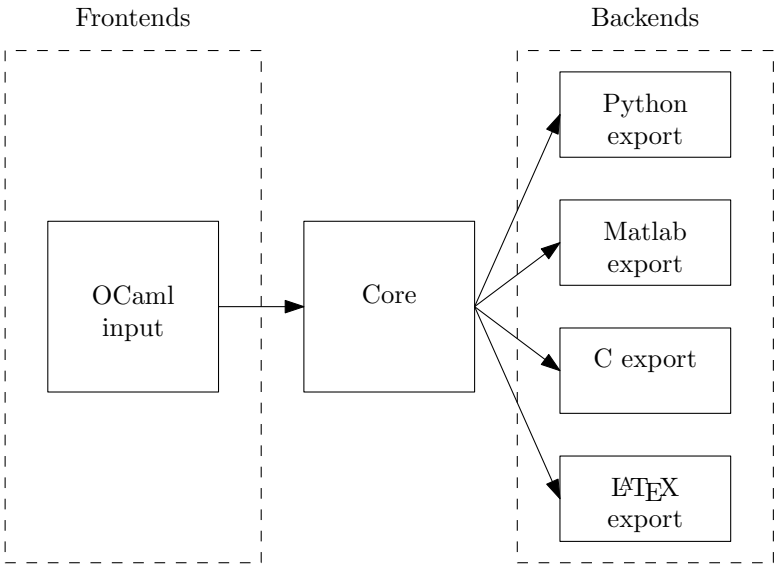

Fig. 4. General architecture of Code you are Happy to Paste

the steps will be the following: first describe the formula in OCaml (Fig. 5) and then compile the description which will be represented as an abstract syntax tree (Fig. 6; for brevity, it is simply a centered and normalized Gaussian). In addition to the formula itself, each description embeds its own documentation, with a description of the formula and with names and properties for the variables used inside.

In the description in Fig. 5, we build a function (line 2) called f, described as *Gaussian PDF* (line 3), taking three arguments (σ, μ and x, line 4) and returning a real number (line 5). After this header, we define three variables (lines 7, 8 and 9), each of them bearing a name and if necessary a documentation and a mathematical property (which is used only for documentation purposes). Finally, the formula itself is described, using a straightforward syntax similar to the one used in many languages (the `Syntax` keyword means the mathematical operators work on nodes of the syntax tree instead of numbers and the `!` are used to convert numbers into nodes).

2.3 Backends

Four backends are available so far, trying to cover various use-cases of scientific computing. In each case, the goal is to produce idiomatic code with as less differences as possible as with handmade code.

Latex The LaTeX output does not need more comments since most of the formulas in this document have been generated using our library. A particular attention has been paid to generate nice looking formula, especially by minimizing the number of parentheses.

```
let gaussian =
  Func.def "f"
    ~doc:"Gaussian PDF"
    ~args:["\\sigma"; "\\mu"; "x"]
    ~return:Real
    Syntax.(
      let x     = real "x" in
      let sigma = real ~doc:"standard deviation" ~prop:"positive"
                       "\\sigma" in
      let mu    = real ~doc:"mean" "\\mu" in
      !1 / (sqrt (!2 * pi * sigma ** !2)) *
        exp (- ((x - mu) ** !2 / sigma ** !2))
    )
```

Fig. 5. Description of the Gaussian distribution

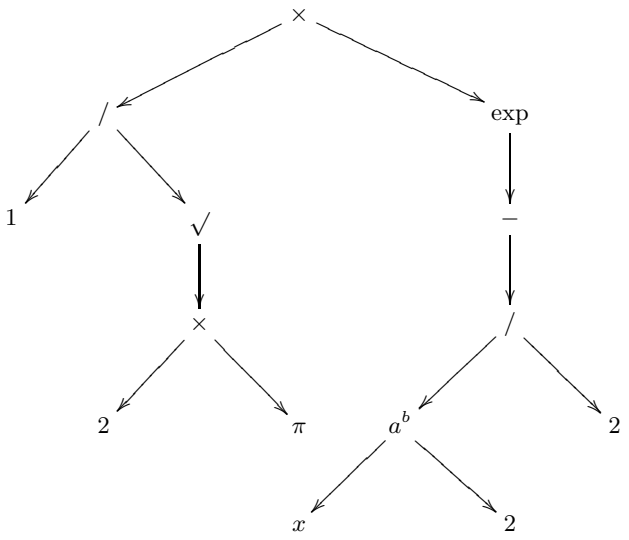

Fig. 6. Tree representing a centered and normalized Gaussian PDF ($\mu = 0, \sigma = 1$)

Python The Python backend outputs code relying on the library `numpy` which is the standard for scientific computing in Python. This library provides basic mathematical functions along with powerful vectors and matrices operations.

```
Python.def ~doc:true gaussian
```

```
def f(sigma, mu, x):
  """ Gaussian PDF

  x:  (real)
  \sigma: standard deviation (real, positive)
  \mu: mean (real)
  """

  return 1 / numpy.sqrt(2 * numpy.pi * sigma**2) * \
    numpy.exp(- (x - mu)**2 / sigma**2)
```

Matlab The Matlab backend generates code using only built-in functions of Matlab.

```
Matlab.def ~doc:true gaussian
```

```
% Gaussian PDF
%
% x:  (real)
% \sigma: standard deviation (real, positive)
% \mu: mean (real)

function f(sigma, mu, x)
  1 / sqrt(2 * pi * sigma^2) * exp(- (x - mu)^2 / sigma^2)
end
```

C The C backend is a little more subtle. First, the C language requires explicit typing indications in the code: thus we need to do the translation between real numbers (on the description side) into `double` (on the C side), and the same for integers and `int`.

```
C.def ~doc:true gaussian
```

```
/* Gaussian PDF

  x:  (real)
  \sigma: standard deviation (real, positive)
  \mu: mean (real)
*/
double f(double sigma, double mu, double x) {
  return 1 / sqrt(2 * M_PI * pow(sigma, 2)) * \
    exp(- pow(x - mu, 2) / pow(sigma, 2));
}
```

The Gaussian PDF is too simple to highlight the others subtleties of this backend, we thus add another example, estimating the mean of a set of values. For a straightforward mean function

$$\text{mean}(X) = \frac{1}{|X|}\left(\sum_{i=1}^{|X|} X_i\right) \tag{2}$$

described by

```
let mean =
  Func.def "mean"
    ~args:["X"]
    ~return:Real
    Syntax.(
      let x = var "X" (Vector Real) in
      !1 / (length x) * sum x
    )
```

we get the following C code:

```
double mean(gsl_vector* X) {
  double tmp1 = 0;
  for(unsigned int i=0; i<X->size; i++)
    tmp1 += gsl_vector_get(X, i);

  return 1 / (X->size) * tmp1;
}
```

First, we made the choice to rely on the Gnu Scientific Library (GSL) for all vectors and matrices operations: in addition to the data structures themselves we get also common mathematical operations on vectors and matrices, simplifying the generated code. Second, since GSL does not provide any function to sum the elements, we need to rewrite the formula to replace the $\sum$ operation by a temporary variable which is populated using a `for` loop.

3 Exponential Families

In order to present the content of our encyclopedia, we give a quick recall on exponential families before showing examples of formula descriptions and exported code.

3.1 Definition

Exponential families are an ubiquitous class of distributions and many widely used distributions belong to this class.

An exponential family is a set of distributions whose probability mass or probability density functions admit the following canonical decomposition:

$$p(x;\theta) = \exp(\langle t(x), \theta\rangle - F(\theta) + k(x)) \tag{3}$$

with

- $t(x)$ the sufficient statistic,
- θ the natural parameters,
- $\langle \cdot, \cdot \rangle$ the inner product,
- F the log-normalizer, which is strictly convex and differentiable,
- $k(x)$ the carrier measure.

Since this log-normalizer F is a strictly convex and differentiable function, it admits a dual representation, the convex conjugate F^*, by the Legendre-Fenchel transform:

$$F^\star(\eta) = \sup_\theta \{\langle \theta, \eta\rangle - F(\theta)\} \tag{4}$$

We get the maximum for $\theta = (\nabla F)^{-1}(\eta)$ and $F^\star$ can be computed with:

$$F^\star(\eta) = \langle \eta, (\nabla F)^{-1}(\eta)\rangle - F((\nabla F)^{-1}(\eta)) \tag{5}$$

Many generic information-geometric algorithms (like Bregman Hard Clustering or Bregman Soft Clustering [1]) rely on the knowledge of this decomposition and thus the implementation of these algorithms require to translate these formulas into computer code. Translating these formulas from a language-agnostic description allows to factorize the effort and is less error-prone than ad-hoc manual work.

3.2 Examples

We describe here the full canonical decomposition of two exponential families, the Gaussian distribution and the Laplace law. For brevity, we only give the description of each formula and the LaTeX export. The reader can find all the source code related to this article on the webpage `http://www.lix.polytechnique.fr/~schwander/ecml2014/`. The same content can also be retrieved in the website described in Section 4.

Gaussian distribution This is the opportunity to introduce new syntactic features: functions can take real vectors as arguments and return them using the type `Vector Real`. Inside the formula, elements of the vector can be accessed using the `@` operator (like `theta@0`).

```
let f =
  Func.def "F"
    ~doc:"Log-normalizer"
    ~args:["\\theta"]
    ~return:Real
    Syntax.(
      let theta = var ~doc:"natural parameter"
        ~prop:"dimension 2" "\\theta" (Vector Real)
      in
      - (!1 / !4 * ((theta@0) ** !2 / (theta@1))) +
        !1 / !2 * log(- (pi / (theta@1)))
    )
```

$$F(\theta) = -\frac{1}{4}\frac{\theta_0^2}{\theta_1} + \frac{1}{2}\log\left(-\frac{\pi}{\theta_1}\right) \tag{6}$$

```
let gradF =
  Func.def "\\nabla F"
    ~doc:"Gradient log normalizer"
    ~args:["\\theta"]
    ~return:(Vector Real)
    Syntax.(
      let theta = var ~doc:"natural parameter"
        ~prop:"dimension 2" "\\theta" (Vector Real)
      in
      vector [
        - (theta@0) / (!2 * (theta@1));
        - !1 / (!2 * (theta@1)) +
          (theta@0) ** !2 / (!4 * (theta@1) ** !2);
      ]
    )
```

$$\nabla F(\theta) = \left(\frac{-\theta_0}{2\theta_1}, \frac{-1}{2\theta_1} + \frac{\theta_0^2}{4\theta_1^2}\right) \tag{7}$$

```
let g =
  Func.def "F^\\star"
    ~doc:"Dual log-normalizer"
    ~args:["\\eta"]
    ~return:Real
    Syntax.(
      let eta = var ~doc:"expectation parameter"
        ~prop:"dimension 2" "\\eta" (Vector Real)
      in
```

```
    - (!1 / !2) * log((eta@0) ** !2 - (eta@1))
  )
```

$$F^\star(\eta) = -\frac{1}{2}\log\left(\eta_0^2 - \eta_1\right) \tag{8}$$

```
let t =
  Func.def "t"
    ~doc:"Sufficient statistic"
    ~args:["x"]
    ~return:(Vector Real)
    Syntax.(
      let x = real ~doc:"observation" "x" in
      vector [x; x ** !2]
    )
```

$$t(x) = \left(x, x^2\right) \tag{9}$$

Laplace distribution Since the Laplace distribution is of order 1 (with only one scalar parameter), the descriptions are much simpler since we do not need to deal with vectors.

```
let pdf =
  Func.def "f"
    ~doc:"Centered Laplace PDF"
    ~args:["\\sigma"; "x"]
    ~return:Mathset.Real
    Syntax.(
      let x     = real "x" in
      let sigma = real ~doc:"standard deviation" ~prop:"positive"
                       "\\sigma" in
      !1 / (!2 * sigma) *
        exp (- (abs x) / sigma)
    )
```

$$f(\sigma, x) = \frac{1}{2\sigma}\exp\left(\frac{-|x|}{\sigma}\right) \tag{10}$$

```
let f =
  Func.def "F"
    ~doc:"Centered Laplace log-normalizer"
    ~args:["\\theta"]
    ~return:Mathset.Real
```

```
    Syntax.(
      let theta = real ~doc:"natural parameter" "\\theta" in
      log (- !2 / theta)
    )
```

$$F(\theta) = \log \frac{-2}{\theta} \tag{11}$$

```
let grad_f =
  Func.def "\\nabla F"
    ~doc:"Centered Laplace gradient log-normalizer"
    ~args:["\\theta"]
    ~return:Mathset.Real
    Syntax.(
      let theta = real ~doc:"natural parameter" "\\theta" in
      - !1 / theta
    )
```

$$\nabla F(\theta) = \frac{-1}{\theta} \tag{12}$$

```
let g =
  Func.def "\\nabla F^\\star"
    ~doc:"Centered Laplace dual log-normalizer"
    ~args:["\\eta"]
    ~return:Mathset.Real
    Syntax.(
      let eta = real ~doc:"expectation parameter" "\\eta" in
      - log eta
    )
```

$$\nabla F^\star(\eta) = -\log \eta \tag{13}$$

```
let grad_g =
  Func.def "\\nabla F^\\star"
    ~doc:"Centered Laplace dual log-normalizer"
    ~args:["\\eta"]
    ~return:Mathset.Real
    Syntax.(
      let eta = real ~doc:"expectation parameter" "\\eta" in
      - !1 / eta
    )
```

$$t(x) = |x| \tag{14}$$

3.3 Mixture Models

In order to learn mixtures of exponential families, we use an Expectation-Maximization (EM) instance [3] called Bregman Soft Clustering [1], allowing to pass the family as an argument of the algorithm. As usual, this is an iterative algorithm where two steps are repeated until convergence of the log-likelihood of the mixture: expectation step and maximization step. See [4] for more details about the exponential family version of these two steps.

Expectation step

$$p(i|x_t, \eta) = \frac{\omega_i \exp\left(F^\star(\eta_i) + \langle t(x_t) - \eta_i, \nabla F^\star(\eta_i)\rangle\right)}{\sum_{j=1}^{k} \omega_j \exp\left(F^\star(\eta_j) + \langle t(x_t) - \eta_j, \nabla F^\star(\eta_j)\rangle\right)} \tag{15}$$

Maximization step

$$\omega_i = \frac{1}{N} \sum_{t=1}^{N} p(i|x_t, \eta) \tag{16}$$

$$\eta_i = \sum_{t=1}^{N} \frac{p(i|x_t, \eta)}{\sum_{t=1}^{N} p(i|x_t, \eta)} t(x_t) \tag{17}$$

Currently, theses two steps need to be implemented by hand in each target language since our description language is not expressive enough to manipulate functions inside the formula (we would need to pass $F^\star$, $\nabla F^\star$, t as arguments to the function, or let them as free variable). Nonetheless, as soon as these steps are implemented, with a `while` loop around to iterate, it can be plugged after the automatically generated formulas, forming a full EM iterative scheme.

4 Website

The *Code-Formula* website (accessible through `http://www.lix.polytechnique.fr/~schwander/codeformula`) is designed to spread knowledge about the exponential families. Following the ideas introduced by precursor online dictionaries of mathematical objects, we think the online format is way more suitable for this kind of content than static documents.

Each page on the site (see the screenshot Fig. 7) shows a card about an exponential family, with a list of formulas related to the family. Each formula is presented first with a rendered version of the the latex output followed by exports in the supported languages.

The goal is to become the reference about decomposition of exponential families, serving to diffuse knowledge, demonstrating our description library but also as a direct source for picking-up pre-made implementations of formulas of interest, for researchers and companies.

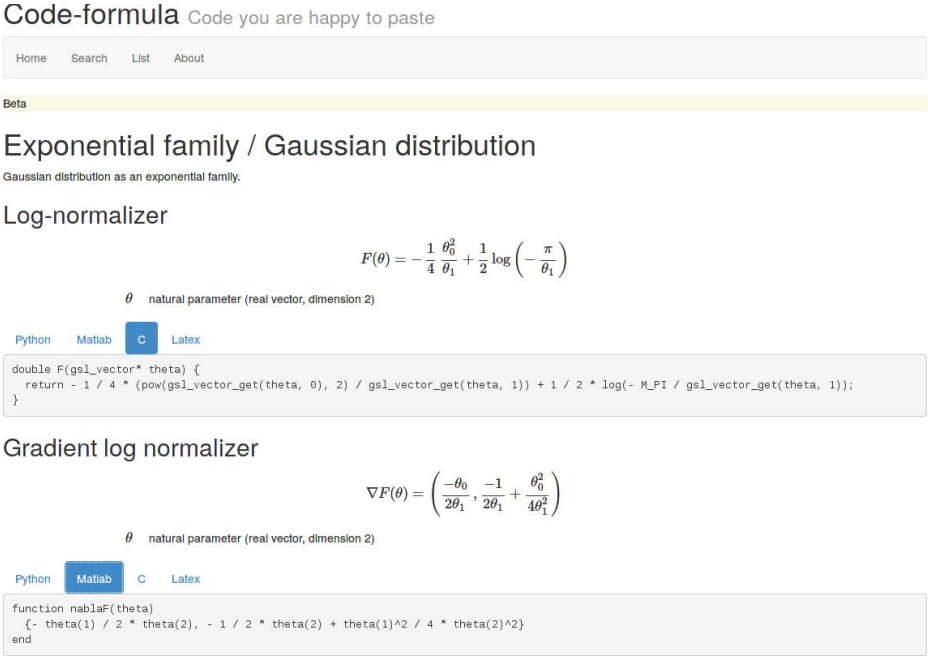

Fig. 7. Gaussian page on Code-Formula, the online encyclopedia of exponential families

5 Conclusion

We presented *Formula*, a library to describe mathematical formulas and to automatically generate code implementing these formulas, and *Code-Formula*, a website showing an online dictionary of exponential families. Both are aimed at reducing the time between chalk board work to real implementation of an algorithm. This is obviously useful for research purposes, easing the first implementation of a new method and also easing a re-implementation of a work by other researchers, but this may also be useful for students or for companies seeking to build a real-world implementation of a method.

There are a lot of perspectives which are under work: on the website side, enlarge the content (contributions are obviously welcomed); on the library side, it should be interesting to be able to generate the headers needed to execute the generated code along with necessary compilation flags; it may also be interesting to render the description language expressive enough to directly describe formulas using other functions (to be able to write the update rule of the EM algorithm for example). On the short term, new export backends are under work, like R and Julia.

Acknowledgments. The author would like to thank Frank Nielsen for the insightful discussions about dictionaries of information geometric objects, in par-

ticular for distances and distributions, and James Regis for providing hosting at the LIX laboratory.

References

1. Banerjee, A., Merugu, S., Dhillon, I.S., Ghosh, J.: Clustering with Bregman divergences. The Journal of Machine Learning Research 6, 1705–1749 (2005)
2. Benoit, A., Chyzak, F., Darrasse, A., Gerhold, S., Mezzarobba, M., Salvy, B.: The Dynamic Dictionary of Mathematical Functions (DDMF). In: Fukuda, K., van der Hoeven, J., Joswig, M., Takayama, N. (eds.) ICMS 2010. LNCS, vol. 6327, pp. 35–41. Springer, Heidelberg (2010)
3. Dempster, A.P., Laird, N.M., Rubin, D.B.: Maximum likelihood from incomplete data via the EM algorithm. Journal of the Royal Statistical Society. Series B (Methodological), 1–38 (1977)
4. Nielsen, F., Garcia, V.: Statistical exponential families: A digest with flash cards. CoRR 09114863 (2009)
5. Olver, F.W.J., Lozier, D.W., Boisvert, R.F., Clark, C.W. (eds.): NIST Digital Library of Mathematical Functions, `http://dlmf.nist.gov/`
6. Schwander, O., Nielsen, F.: Fast learning of gamma mixture models with k-MLE. In: Hancock, E., Pelillo, M. (eds.) SIMBAD 2013. LNCS, vol. 7953, pp. 235–249. Springer, Heidelberg (2013)
7. Schwander, O., Nielsen, F.: PyMEF – A framework for exponential families in Python. In: 2011 IEEE Statistical Signal Processing Workshop (SSP), pp. 669–672 (2011)
8. Neil Sloane. The On-Line Encyclopedia of Integer Sequences, `http://oeis.org/`