Symbolic Statistics with SymPy

Matthew Rocklin

Department of Computer Science
University of Chicago

Andy R. Terrel

Texas Advanced Computing Center
University of Texas at Austin

Abstract

Symbolic methods are becoming more popular for describing complex physical models. By replacing symbols with random variables, one can naturally add statistical operations to these models. We present three examples of symbolic statistical modeling using new features from the popular SymPy project.

1 Introduction

Real world simulations require not only the physical models that currently dominate scientific computing but also the encapsulation of uncertainty, that is the use of statistics. Unfortunately, the difficulty of implementing the physical model often takes most of the development efforts and adding uncertainty to the models is yet another axis of requirements.

To address this challenge, we present the use of random variables at the symbolic level with several examples of their use. Just as many domain specific languages are able to use symbolic methods to represent problems and generate implementations, we are able to add statistics to a physical model by augmenting the language such that the variables involved represent the uncertainty.

Below we give three examples of increasing complexity that use the random variable capabilities in SymPy, a popular symbolics package for the Python programming language, see a summary by Joyner et al[2]. The first dice example gives an example to reacquaint the reader with some basic statistics. Second we have an example of modeling temperatures and assimilating the physical model with data measurements. Finally, we give a larger more complete example using a classic kinematics problem.

2 Example 1 - Dice

Formally random variables are functions on probability spaces. Practically, they represent ranges of values, each with a certain probability. In addition to numeric variables like $x = 3$, SymPy can also manipulate symbolic variables, $x = \text{Symbol}('x')$. SymPy now includes a random variable type which is used to represent uncertainty in variables. For example, one can declare and ask questions about a set of simple six–sided dice.

```
>>> from sympy.stats import *
>>> X = Die(6) # a six sided die
>>> Y = Die(6) # a six sided die
>>> P(X>3) # probability that X is greater than 3
1/2
>>> E(X+Y) # expectation of the sum of two dice
7
```

X represents the value of a single die roll. It can take on the values $1, 2, 3, 4, 5$ or 6, each with probability $\frac{1}{6}$. In general these variables act like normal symbolic variables and can participate in normal mathematical expressions. For example, if you want to modify a die roll by a handicap, you can add an integer to it.

```
>>> mod_X = X + 3
```

The difference between a random variable and a normal symbol can be seen when using the statistical operators described in Table 1. Here SymPy turns statistical expressions like `E(2*X + 4)` into a computed result. Additionally, one can compute statistics of the expression conditioned on additional knowledge as in Table 2.

Operators	SymPy function	example
probability	`P`	`P(X > 3)` $\rightarrow 1/2$
expectation	`E`	`E(2*X)` $\rightarrow 7$
variance	`var`	`var(X + 3)` $\rightarrow 35/12$
density	`Density`	`Density(2*X)` $\rightarrow \{2 : 1/6,\ 4 : 1/6,\ 6 : 1/6,\ 8 : 1/6,\ 10 : 1/6,\ 12 : 1/6\}$

Table 1: Operators over expressions using random variables

Explanation	SymPy function	example
Conditional random variable	`Given`	`X2 = Given(X, X < 3)`
Conditional density	`Density`	`Density(X, X > 3)` $\rightarrow \{4 : 1/3,\ 5 : 1/3,\ 6 : 1/3\}$
Conditional expectation	`E`	`E(X, X > 3)` $\rightarrow 5$

Table 2: Conditional operators over random expressions

While we use the `Die` function above as a syntactic sugar, one can represent any probability space by providing the density function, a map from value to probability summing to one. For example, we define a fair coin as follows:

```
>>> density_map = {'heads':.5, 'tails':.5}
>>> coin = FiniteRV(density_map)
```

By introducing a random variable type into the SymPy algebraic system, we have created a language for modeling and querying symbolic statistical systems. While these simple examples using dice are minimal, they form the basis for more sophisticated examples. These simple operations allow us to model statistical systems in a readable and intuitive manner.

3 Example 2: Data Assimilation

We just saw how to use random variables in simple examples involving discrete random variables, in particular six sided dice. These problems were solved internally by iteration as done by Erwig et al [1].

In this section, we consider continuous random variables. Instead of iterating through the continuum of possible values, we generate and solve integral expressions. While the backend mechanics have been completely changed, the method of setting up the statistical problem remains unchanged. This is an advantage of integrating uncertainty in the language, a theme upon which we will continue to elaborate throughout the article.

Consider the problem of data assimilation. On a hot summer day, one might guess the temperature outside to be about $30C$. The uncertainty of this guess can be encapsulated by an error of $\pm 3C$. In SymPy one can model model this estimate with a normal random variable:

```
>>> T = Normal(30, 3)
```

As in the last section, we may ask statistical questions such as, *W*hat is the probability that the temperature is greater than 33C?, $P(T > 33)$. Such questions produce integral expressions which are then solved using the SymPy integration engine to produce numeric results.

$$\begin{aligned} P(T > 33) &= \int_{33}^{\infty} \frac{\sqrt{2}e^{-\frac{1}{18}(x-30)^2}}{6\sqrt{\pi}}\, dx \\ &= -\frac{1}{2}\operatorname{erf}\left(\frac{1}{2}\sqrt{2}\right) + \frac{1}{2} \\ &= 0.15866 \end{aligned}$$

Unsatisfied with this single estimate of the temperature, we want to update the our understanding with measurements from a thermometer. However, the thermometer is difficult to read because the lines are spaced closely together. This means that our observations will include some noise, which we include in our model below:

```
>>> noise = Normal(0, 1.5)
>>> observation = T + noise
```

The thermometer measures a value of $26C$. We now have two pieces of information to model the temperature: the measured data ($26 \pm 1.5C$) and our *prior* understanding ($30 \pm 3C$). We plot them both in Fig. 1.

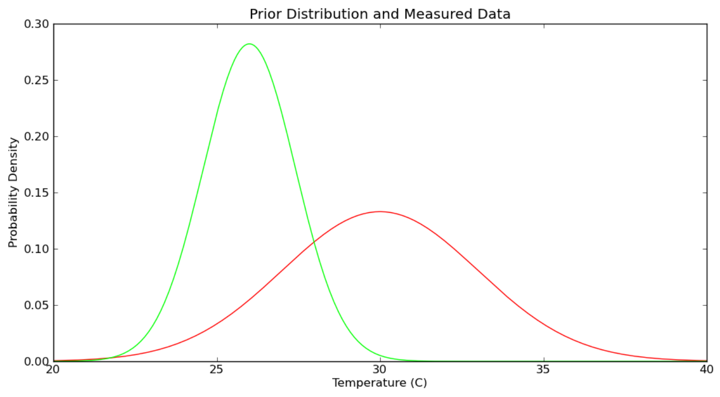

Figure 1: The probability density functions of our two pieces of data. Note that the observation is less uncertain and thus more centered about the mean.

After this measurement is made how should the final estimate of the temperature change? We have the original estimate, $30 \pm 3C$, and the new measurement $26 \pm 1.5C$. One could select the thermometer's reading because it is more precise but this entirely neglects the original information which still holds some value. It would be best to cleanly assimilate the new bit of data $(26C \pm 1.5C)$ into the prior understanding $(30C \pm 3C)$

This is the problem of Data Assimilation. We want to assimilate a new measurement (data) into our previous understanding (prior) to form a new and better-informed understanding (posterior). That is we want to compute $T' = T$ given an observation of $26C$. We compute this in SymPy as follows:

```
>>> T_posterior = Given(T, Eq(observation , 26))
```

The new distribution is plotted in Fig. 2. The exact functional form is proportional to $e^{-\frac{2}{9}(-x+26)^2}e^{-\frac{1}{18}(x-30)^2}$

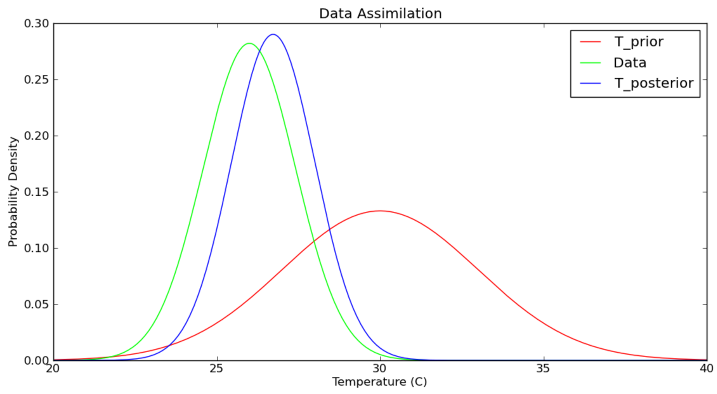

Figure 2: The density of the posterior represents our understanding after the observation has been assimilated into our prior understanding. Note that it is more centered/certain than either of the two.

This is a classic result in data assimilation. It is important to note that the rules for this particular example of data assimilation are not explicitly written into SymPy. Rather the fundamental rules of Bayesian statistics are coded into SymPy and these rules cause this and many other classical results to develop naturally.

4 Example 3: Kinematics

For the final example, we give a more complete simulation from a classical kinematics. This example is selected both because it is familiar to the reader and because it demonstrates what happens when the systems we model increase in complexity.

Consider shooting a cannon ball off of a cliff. We ask where will the ball land below? We start by modeling this mathematical problem in SymPy without statistics:

```
# Symbols for position, velocity, angle, time, and gravity
>>> x0, y0, yf, v, theta, t, g = symbols('x0 y0 yf v theta t g')
# x and y positions as a function of time
>>> x = x0 + v*cos(theta)*t
>>> y = y0 + v*sin(theta)*t + g/2*t**2
# Solve y = yf for time to obtain the duration of the flight
>>> t_impact = solve(y-yf, t)[1]
# Final x value of cannon ball on impact
>>> xf = x0 + v*cos(theta)*t_impact
```

This code yields the following symbolic solution:

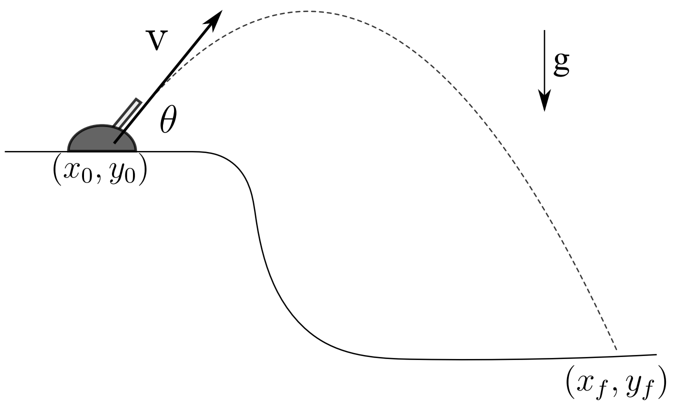

Figure 3: Variables involved in shooting a cannon ball from a cliff

$$x_f = x_0 - \frac{v\left(v\sin(\theta) + \sqrt{-2gy_0 + 2gy_f + v^2\sin^2(\theta)}\right)\cos(\theta)}{g} \tag{1}$$

To obtain numeric results for a particular set of values, we can replace the symbolic variables, $x_0, y_0, y_f, v, \theta, g$ with numeric ones:

```
# Substitute out symbolic variables in xf for numeric values
>>> xf_num = xf.subs({ g: -9.8, v: 10, theta: pi/4,
                       x0: 0, y0: 10, yf: 0})
```

$$x_f = \frac{250}{49} + \frac{50}{49}\sqrt{123} = 16.4189 \tag{2}$$

The expression in Eqn. 1 is less appealing to the naked eye since it is hard to visualize its meaning but it serves two purposes. Because it is a full symbolic description of the problem, it can provide insight as to the relation of the variables to the result. Additionally, it serves as a staging ground to produce many numeric solutions such as the one in Eqn. 2. Once the mathematical model has been described no further conceptual work is necessary on the part of the programmer to produce numeric results.

4.1 Adding Uncertainty

We now consider the statistical problem. Imagine first that we are uncertain about the initial height of the cannon. Rather than knowing the precise value $y_0 = 10$, we suspect it is distributed normally with standard deviation 1, $y_0 \sim \mathcal{N}(10, 1)$. Due to this uncertainty x_f no longer has a precise value. Instead of asking for an exact number, we may ask for the expectation value, variance, or even the full distribution.

Traditionally, this is done either through painstaking algebra (for simple cases) or by building external codes to sample using Monte Carlo methods. We now show how SymPy can be used to model this statistical problem and provide solutions.

Rather than defining y_0 as a symbolic or numeric variable, we declare it to be a random variable distributed normally with mean 10 and standard deviation 1.

```
>>> y0 = Normal(10,1)
```

This variable acts in all ways like a SymPy symbol and so we may still form the symbolic expression in Eqn. 1. Because this expression now contains random elements, it itself is random and is active under our statistical operators.

```
>>> E(xf)
```

This code follows standard theory to generate the following integral

$$\int_{-\infty}^{\infty} \frac{25}{49} \frac{\left(\sqrt{\frac{98}{5}x_1 + 50} + 5\sqrt{2}\right) e^{-\frac{1}{2}(x_1-10)^2}}{\sqrt{\pi}}\, dx_1$$

This integral is challenging and the solution includes special functions. Fortunately, algorithmic tools are capable of producing exact analytic solutions. The full solution is omitted here in the interests of space and clarity; numeric equivalents are presented below in their place (floating point results are requested using the `evalf()` method).

$$\text{E}(x_f)\text{.evalf()} = 16.4099...$$
$$\text{var}(x_f)\text{.evalf()} = .2044...$$

In principle, we could ask for the full analytically computed density of x_f although the complexity prohibits any meaningful insight.

4.2 Dealing with Complexity

Imagine now a more difficult problem where more of the variables in our model were uncertain

```
>>> v = Uniform(9,11.5) # Uniformly distributed between 9 and 11.5
>>> theta = Normal(pi/4, pi/20)
```

Our queries may also become more complex such as, *What is the probability that theta was greater than* $\pi/4$ *given that the ball landed past* $x_f > 16$*?* , $P(\theta > \frac{\pi}{4} | x_f > 16)$. Asking this question in SymPy is just as easy as the example above:

```
>>> P(theta>pi/4   ,  xf>16)
```

Solving the resultant integral however is substantially more challenging. For situations like these exact solutions are often infeasible and Monte Carlo codes are built around the problem. Because this problem statement is now cleanly separated from the solution code, it is trivial to connect a simple sampling engine onto SymPy stats allowing all such queries to be approximated using a simple syntax

```
>>> P(theta>pi/4,  xf>16, numsamples=10000)
0.3428
```

The path to solution presented here is typical in a lab environment. First a mathematical model is built, then initial numeric results are computed and finally errors, uncertainty and statistics are addressed. The work presented here shows how a computer algebra system that includes symbolic statistics can facilitate this process in a real-world setting.

5 Conclusion

In this short presentation, we have seen a number of concepts around using symbolic statistics that have been recently added to the SymPy project. At a low level is the random expression type as a tool for making meaningful statistical statements over complex expressions. At a high level we hope to have motivated the use of symbolics in everyday computation.

Just as the article in this section from March 2011 [3], the use of language features to include domain specifics has led to a natural way of thinking about a difficult subject. While not covered in this article, the SymPy implementation of random variables has allowed for a rich set of probabilistic spaces which are available to use in complex expressions. The authors invite users to experiment with these features and contribute back to the project with their own specific interests and expertise.

References

[1] Martin Erwig and Steve Kollmansberger. Probabilistic Functional Programming in Haskell. *Journal of Functional Programming*, 16(1):21–34, 2006.

[2] David Joyner, Ondřej Čertík, Aaron Meurer, and Brian Granger. Open source computer algebra systems: SymPy. *ACM Communications in Computer Algebra*, 45(5), December 2011.

[3] Andy R. Terrel. From equations to code: Automated Scientific Computing. *Computing in Science and Engineering*, 13, March/April 2011.

A Sidebars

A.1 Sidebar: SymPy project

What is SymPy?

SymPy is a Python library for symbolic mathematics. It aims to become a full-featured computer algebra system (CAS) while keeping the code as simple as possible in order to be comprehensible and easily extensible. SymPy is written entirely in Python and does not require any external libraries.

Where do I find it?

Website:	http://sympy.org
Mailing List:	http://groups.google.com/group/sympy
IRC:	irc://irc.freenode.net/sympy
Source:	http://github.com/sympy/sympy

Who develops SymPy?

SymPy has been developed since 2006 as an open source project under the bazaar model with over 150 contributors. Surprisingly, it is not developed by any particular university group or company but has become a competitive alternative to many CAS projects that are. Google Summer of Code has funded 25 student contributors over the summers since 2007.

Who uses SymPy?

Because SymPy is open source and pure Python, it is easy to integrate its functionality into other research projects. Several projects exist which make use of SymPy in different disciplines. Examples include Sage, PyDy, Ignition, several symbolic math GUI's, and many more.

How can you contribute?

While no project is perfect, SymPy could be a little faster, better documented and allow for easier embedding in other projects. New wanted features include new solvers, new symbolic integration algorithms, and better domain support. For those interested in helping see the development section of the website.

A.2 Sidebar: Random Variable Implementation

While the operations in the examples may seem magic, we wanted to give a peek into the implementation to show they are in fact quite straight forward. SymPy uses fairly general expression trees that are operated on by the various functions to produce new trees. To add a domain specific symbol, one needs to do only a few things:

1. Create a class inheriting from the `Symbol` class.
2. Put all canonicalization of a symbol into the `__new__` method, this insures that symbols can take advantage of SymPy's caching system. Arguments are stored in an `args` array for functions to inspect.
3. Add domain specific functions to a distinct module which manipulate SymPy expressions
4. Add special methods for printing, integration, and other common operations. These usually start with `_eval_`, and if not provided will be substituted with default representations.

For random variables, we first create the abstract `PSpace` class to hold information about underlying probability spaces. The `RandomSymbol` class inherits from the `Symbol` class and represents the implemented `PSpace` within SymPy expressions. The statistical operators `P, E, Density, ...` inspect the expression tree, extract the active probability spaces represented within, and construct a new non-statistical expression to compute the desired quantity.

In the case of continuous random variables the statistical operators turn a statistical expression (i.e. `P(T>33))` into a non-statistical integral expression (i.e. $\int_{33}^{\infty} \frac{\sqrt{2}e^{-\frac{1}{18}(x-30)^2}}{6\sqrt{\pi}}\, dx$). For discrete random expressions (i.e. dice) form iterators or summation expressions. Multivariate random variables form matrix expressions, etc.... In each case SymPy-Stats offloads all computational work onto other more specialized and powerful modules. This allows SymPy-Stats to leverage the power and continuous growth of the more mature core modules.

While the project required several improvements to SymPy core, it does show the power of the simple language approach SymPy uses. We include these implementation details in the hopes that the reader will be able to add her own domain specific symbols.