Symbolic Statistics with SymPy

Matthew Rocklin

Abstract—We add a random variable type to a mathematical modeling language. We demonstrate through examples how this is a highly separable way to introduce uncertainty and produce and query stochastic models. We motivate the use of symbolics and thin compilers in scientific computing.

Index Terms—Symbolics, mathematical modeling, uncertainty, SymPy

Introduction

Mathematical modeling is important.

Symbolic computer algebra systems are a nice way to simplify the modeling process. They allow us to clearly describe a problem at a high level without simultaneously specifying the method of solution. This allows us to develop general solutions and solve specific problems independently. This enables a community to act with far greater efficiency.

Uncertainty is important. Mathematical models are often flawed. The model itself may be overly simplified or the inputs may not be completely known. It is important to understand the extent to which the results of a model can be believed. To address these concerns it is important that we characterize the uncertainty in our inputs and understand how this causes uncertainty in our results.

In this paper we present one solution to this problem. We can add uncertainty to symbolic systems by adding a random variable type. This enables us to describe stochastic systems while adding only minimal complexity.

Motivating Example - Mathematical Modeling

Consider an artilleryman firing a cannon down into a valley. He knows the initial position (x_0, y_0) and orientation, θ, of the cannon as well as the muzzle velocity, v, and the altitude of the target, y_f.

```
# Inputs
>>> x0 = 0
>>> y0 = 0
>>> yf = -30 # target is 30 meters below
>>> g = -10 # gravitational constant
>>> v = 30 # m/s
>>> theta = pi/4
```

If this artilleryman has a computer nearby he may write some code to evolve forward the state of the cannonball. If he also has a computer algebra system he may choose to solve this system analytically.

Matthew Rocklin is with University of Chicago, Computer Science. E-mail: mrocklin@cs.uchicago.edu.

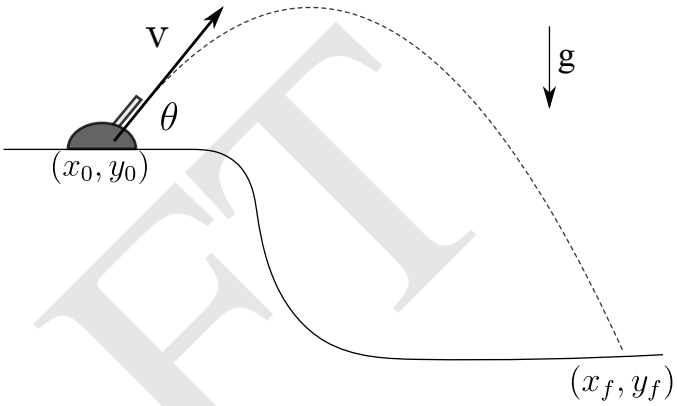

Fig. 1: *The trajectory of a cannon shot*

```
>>> t = Symbol('t') # SymPy variable for time
>>> x = x0 + v * cos(theta) * t
>>> y = y0 + v * sin(theta) * t + g*t**2
>>> impact_time = solve(y - yf, t)
>>> xf = x0 + v * cos(theta) * impact_time
>>> xf.evalf() # evaluate xf numerically
65.5842
# Plot x vs. y for t in (0, impact_time)
>>> plot(x, y, (t, 0, impact_time))
```

If he wishes to use the full power of SymPy he may choose to solve this problem generally. He can do this simply by changing the numeric inputs to be symbolic

```
>>> x0 = Symbol('x_0')
>>> y0 = Symbol('y_0')
>>> yf = Symbol('y_f')
>>> g = Symbol('g')
>>> v = Symbol('v')
>>> theta = Symbol('theta')
```

He can then run the same modeling code found in (missing code block label) to obtain full solutions for impact_time and the final x position.

```
>>> impact_time
```

$$\frac{-v\sin(\theta) + \sqrt{-4gy_0 + 4gy_f + v^2\sin^2(\theta)}}{2g}$$

```
>>> xf
```

$$x_0 + \frac{v\left(-v\sin(\theta) + \sqrt{-4gy_0 + 4gy_f + v^2\sin^2(\theta)}\right)\cos(\theta)}{2g}$$

Motivating Example - Uncertainty Modeling

To control the velocity of the cannon ball the artilleryman introduces a certain quantity of gunpowder to the cannon.

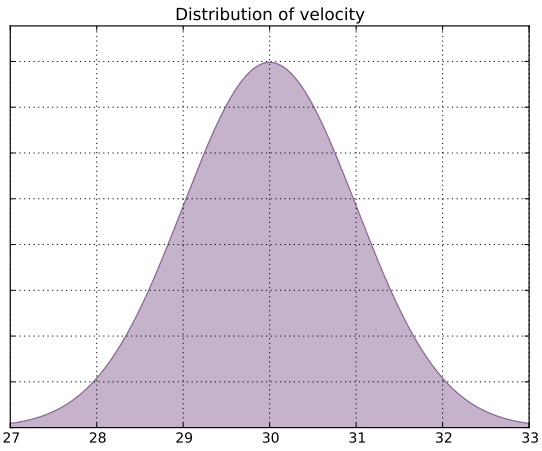

Fig. 2: *The distribution of possible velocity values*

While he takes care he is aware that his estimate of the velocity is uncertain.

He models this uncertain quantity as a *random variable* that can take on a range of values, each with a certain probability. In this case he believes that the velocity is normally distributed with mean 30 and standard deviation 1.

```
>>> from sympy.stats import *
>>> z = Symbol('z')
>>> v = Normal('v', 30, 1)
>>> pdf = density(v)
>>> plot(pdf(z), (z, 27, 33))
```

$$\frac{\sqrt{2}e^{-\frac{1}{2}(z-30)^2}}{2\sqrt{\pi}}$$

The artilleryman can now rerun the mathematical model (reference to code above) without modification. The expressions `x, y, impact_time, xf` are now stochastic expressions and we can use operators like `P, E, variance, density` to convert stochasitc expressions into computational ones.

For example we can ask the probability that the muzzle velocity is greater than 31.

```
>>> P(v > 31)
```

$$-\frac{1}{2}\operatorname{erf}\left(\frac{1}{2}\sqrt{2}\right)+\frac{1}{2}$$

This converts a random/stochastic expression `v > 31` into a deterministic computation. The expression `P(v > 31)` actually produces an integral expression.

```
>>> P(v > 31, evaluate=False)
```

$$\int_{31}^{\infty} \frac{\sqrt{2}e^{-\frac{1}{2}(z-30)^2}}{2\sqrt{\pi}}\,dz$$

We can ask similar questions about the other expressions. For example we can compute the probability density of the position of the ball as a function of time.

```
>>> density(x).expr
```

$$\frac{\sqrt{2}e^{-\frac{z^2}{t^2}}e^{30\frac{\sqrt{2}z}{t}}}{2\sqrt{\pi}e^{450}}$$

```
>>> density(y).expr
```

$$\frac{\sqrt{2}e^{-\frac{\left(z+10t^2\right)^2}{t^2}}e^{30\frac{\sqrt{2}\left(z+10t^2\right)}{t}}}{2\sqrt{\pi}e^{450}}$$

Note that to obtain these expressions the only novel work the modeler needed to do was to describe the uncertainty of the inputs. The modeling code (cite code) was not touched.

We can attempt to compute more complex quantities such as the expectation and variance of `impact_time` the total time of flight

```
>>> E(impact_time)
```

$$\int_{-\infty}^{\infty} \frac{\left(v+\sqrt{v^2+2400}\right)e^{-\frac{1}{2}(v-30)^2}}{40\sqrt{\pi}}\,dv$$

In this case the necessary integral proved too challenging for the SymPy integration algorithms and we are left with a correct though unresolved result.

Sampling

While this case is unfortunate it is also quite common. Many mathematical models are too complex for analytic solutions. There are many approaches to these problems, the most common of which is standard monte carlo sampling.

SymPy.stats contains a basic monte carlo backend which can be easily accessed with an additional keyword argument

```
>>> E(impact_time, numsamples=10000)
3.09058769095056
>>> variance(impact_time, numsamples=30000)
0.00145642451022709

>>> E(xf, numsamples=1000)
65.4488501921592
```

Implementation

A `RandomSymbol` class/type and the functions `P, E, density, sample` are the outward-facing core of sympy.stats and the `PSpace` class in the internal core representing the mathematical concept of a probability space.

A `RandomSymbol` object behaves in every way like a standard sympy `Symbol` object. Because of this one can replace standard sympy variable declarations like

```
x = Symbol('x')
```

with code like

```
x = Normal('x', 0, 1)
```

and continue to use standard SymPy without modification.

After final expressions are formed the user can query them using the functions `P, E, density, sample`. These functions inspect the expression tree, draw out the `RandomSymbols` and ask these random symbols to construct a probabaility space or `PSpace` object.

The `PSpace` object contains all of the logic to turn random expressions into computational ones. There are several types of probability spaces for discrete, continuous, and multivariate distributions. Each of these generate different computational expressions.

RV Type	Computational Type
Continuous	SymPy Integral
Discrete - Finite (dice)	Python iterators / generators
Discrete - Infinite (Poisson)	SymPy Summation
Multivariate Normal	SymPy Matrix Expression

TABLE 1: *Different types of random expressions reduce to different computational expressions (Note: Infinite discrete and multivariate normal are in development and not yet in the main SymPy distribution)*

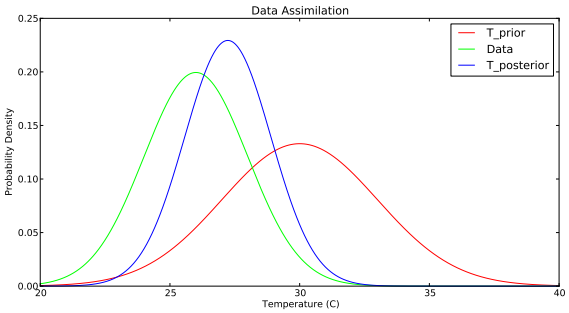

Fig. 3: *The prior, data, and posterior distributions of the temperature.*

Implementation - Bayesian Conditional Probability

SymPy.stats can also handle conditioned variables. In this section we describe how the continuous implementation of sympy.stats forms integrals using an example from data assimilation.

We measure the temperature and guess that it is about 30C with a standard deviation of 3C.

```
>>> from sympy.stats import *
>>> T = Normal('T', 30, 3) # Prior distribution
```

We then make an observation of the temperature with a thermometer. This thermometer states that it has an uncertainty of 1.5C

```
>>> noise = Normal('eta', 0, 1.5)
>>> observation = T + noise
```

With this thermometer we observe a temperature of 26C. We compute the posterior distribution that cleanly assimilates this new data into our prior understanding. And plot the three together.

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

We now describe how SymPy.stats obtained this result. The expression T_posterior contains two random variables, `T` and `noise` each of which can independently take on different values. We plot the joint distribution below in figure (reference figure). We represent the observation that `T + noise == 26` as a diagonal line over the domain for which this statement is true. We project the probability density on this line to the left to obtain the posterior density of the temperature.

These gemoetric operations correspond exactly to Bayesian probability. All of the operations such as restricting to the condition, projecting to the temperature axis, etc... are all managed using core SymPy functionality.

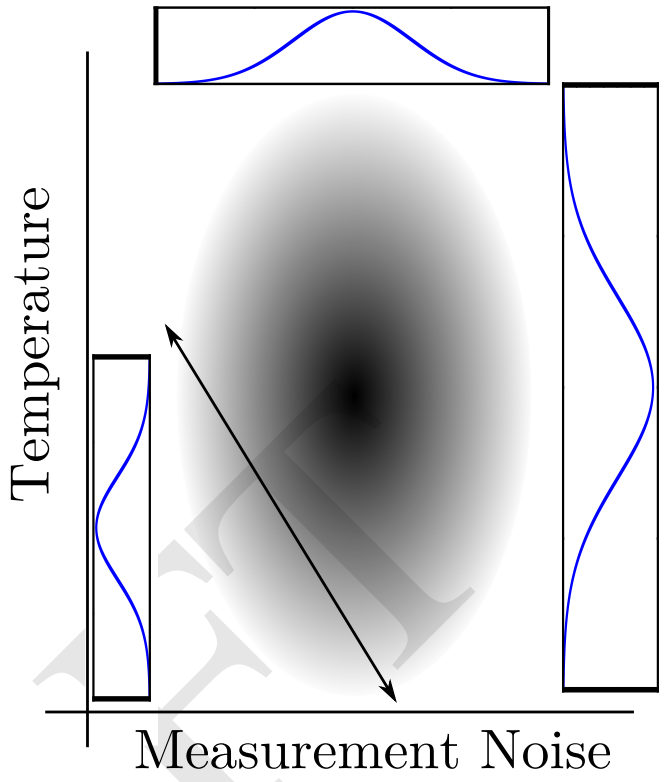

Fig. 4: *The joint prior distribution of the temperature and measurement noise. The constraint `T + noise == 26` (diagonal line) and the resultant posterior distribution of temperature on the left.*

Multi-Compilation

Scientific computing is a demanding field. Solutions frequently encompass concepts in a domain discipline (such as fluid dynamics), mathematics (such as PDEs), linear algebra, sparse matrix algorithms, parallelization/scheduling, and local low level code (C/FORTRAN/CUDA). Recently uncertainty layers are being added to this stack (Monte Carlo, polynomial chaos, etc....)

Often these solutions are implemented as single monolithic codes. This approach is challenging to accomplish, difficult to reason about after-the-fact and rarely allows for code reuse. As hardware becomes more demanding and scientific computing expands into new and less well trained fields this challenging approach fails to scale. This approach is not accessible to the average scientist.

Various solutions exist for this problem. Libraries such as BLAS and LAPACK provide very high quality solutions for the lowest level of the stack on various architectures (i.e. CPU-BLAS or GPU-cuBLAS). High quality implementations of the middle-to-bottom of the stack are available through higher level libraries such as PETSc and Trilinos or through code generation solutions such as FENICS.

continuous random variables transforms random expressions into integral expressions and then stops. It does not attempt to generate an end-to-end code. Because its backend interface layer (SymPy integrals) is simple and well defined it can be used in a plug-and-play manner with a variety of other back-end solutions.

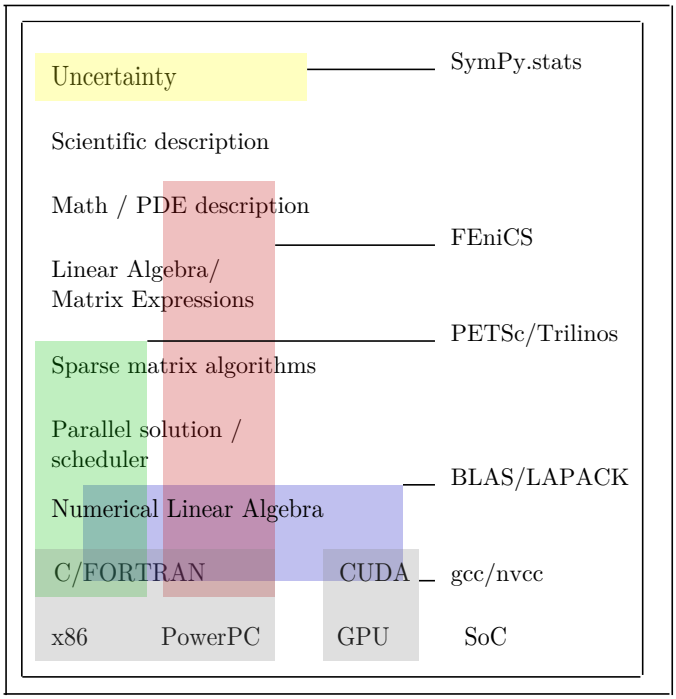

Fig. 5: *The scientific computing software stack. Various projects are displayed showing the range that they abstract. We pose that scientific computing needs more horizontal and thin layers in this image.*

In this case the following backends are easily accessible

- SymPy symbolic integration algorithms
- SymPy.stats Monte Carlo
- SciPy.integrate (uses QUADPACK library)
- Direct Fortran code generation

Additionally it is possible to create other methods to compute integrals. This is a general problem that interests and engages a far broader community than just those interested in uncertainty.

Other sympy.stats implementations such as multivariate-normal generate similarly structured outputs. In this case the matrix expressions generated by `sympy.stats.mvnrv` can be easily transformed to an input to other symbolic/numeric systems such as FLAME or to code generation solutions such as []. Symbolic/numerical linear algebra is a rapidly changing field. Because it offers a clean interface layer SymPy.stats is able to easily engage with these developments.

We generally support the idea of approaching the scientific computing conceptual stack (Physics/PDEs/Linear-algebra/MPI/C-FORTRAN-CUDA) with a sequence of simple and atomic compilers. The idea of using interface layers to break up a complex problem is not new but is oddly infrequent in scientific computing and thus warrants mention. It should be noted that maximal speedup often requires optimizing the whole problem at once and so for heroic computations this approach is not valid.

Conclusion

We have foremost demonstrated the use of `sympy.stats` a module that enhances `sympy` with a random variable type. We have shown how this module allows mathematical modellers to describe the undertainty of their inputs and compute the uncertainty of their outputs with simple and non-intrusive changes to their code.

Secondarily we have motivated the use of symbolics in computation and argued for a more separable computational stack within the scientific computing domain.