Uncertainty Modeling with SymPy Stats

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

Scientific computing is becoming more challenging. On the computational machinery side heterogeneity and increased parallelism are increasing the required effort to produce high performance codes. On the scientific side, computation is used for problems of increasing complexity by an increasingly broad and untrained audience. The scientific community is attempting to fill this widening need-to-ability gulf with various solutions. This paper discusses symbolic mathematical modeling.

Symbolic mathematical modeling provides an important interface layer between the *description of a problem* by domain scientists and *description of methods of solution* by computational scientists. This allows each community to develop asynchronously and facilitates code reuse.

In this paper we will discuss how a particular problem domain, uncertainty propagation, can be expressed symbolically. We do this by adding a random variable type to a popular mathematical modeling language, SymPy [Sym, Joy11]. This allows us to describe stochastic systems in a highly separable and minimally complex way.

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. Uncertainty propagation is the act of determining the effects of uncertain inputs on outputs. To address these concerns it is important that we characterize the uncertainty in our inputs and understand how this causes uncertainty in our results.

Motivating Example - Mathematical Modeling

We motivate this discussion with a familiar example from kinematics.

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 to see where it hits lands.

```
>>> while y > yf: # evolve time forward until y hits the ground
...     t += dt
...     y = y0 + v*sin(theta)*t
...            + g*t**2 / 2
>>> x = x0 + v*cos(theta)*t
```

Notice that in this solution the mathematical description of the problem $y = y_0 + v\sin(\theta)t + \frac{gt^2}{2}$ lies within the while loop. The problem and method are woven together. This makes it difficult both to reason about the problem and to easily swap out new methods of solution.

If the artilleryman also has a computer algebra system he may choose to model this problem and solve it separately.

```
>>> 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))
```

In this case the *solve* operation is nicely separated. SymPy defaults to an analytic solver but this can be easily swapped out if analytic solutions do not exist. For example we can easily drop in a numerical binary search method if we prefer.

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

```
>>> 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.

* Corresponding author: mrocklin@cs.uchicago.edu
‡ University of Chicago, Computer Science

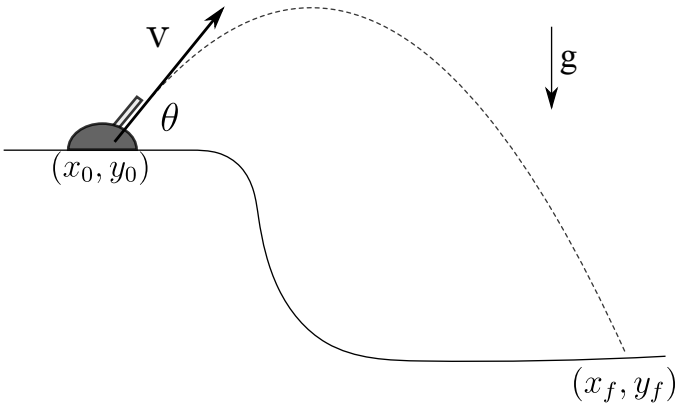


Fig. 1: *The trajectory of a cannon shot*

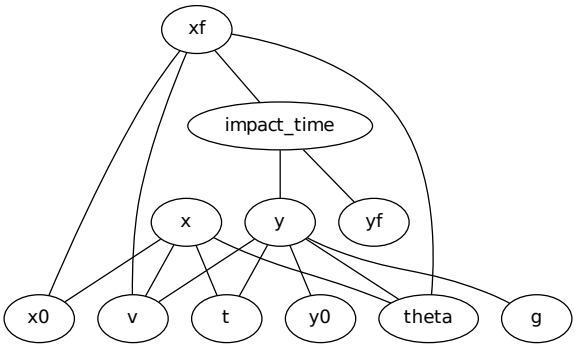


Fig. 2: *A graph of all the varibles in our system. Variables on top depend on variables connected below them. The leaves are inputs to our system.*

```
>>> impact_time
```

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

```
>>> xf
```

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

Rather than produce a numeric result, SymPy produces an abstract syntax tree. This form of result is easy to reason about for both humans and computers. This allows for the manipulations which provide the above expresssions and others. For example if the artilleryman later decides he needs derivatives he can very easily perform this operation on his graph.

Motivating Example - Uncertainty Modeling

To control the velocity of the cannon ball the artilleryman introduces a certain quantity of gunpowder to the cannon. He is unable to pour exactly the desired quantity of gunpowder however and so his estimate of the velocity will be 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.

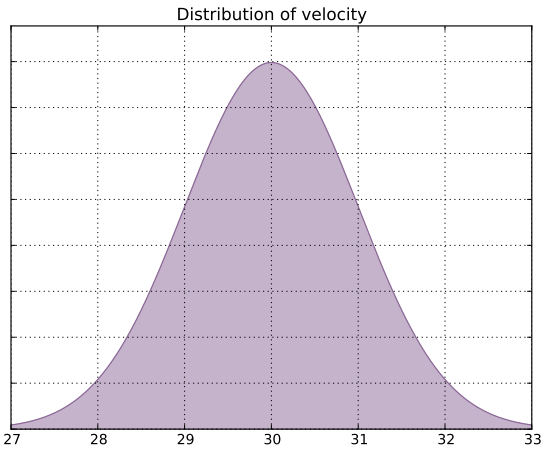


Fig. 3: *The distribution of possible velocity values*

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

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

`v` is now a random variable. We can query it with the following operators

```
P           -- # Probability
E           -- # Expectation
variance    -- # Variance
density     -- # Probability density function
sample      -- # A random sample
```

These 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 intermediate integral expression which is solved with SymPy's integration routines.

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

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

Every expression in our graph that depends on `v` is now a random expression

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

```
>>> a,b = symbols('a,b')
>>> density(x)(a) * density(y)(b)
```

$$\frac{e^{-\frac{a^2}{t^2}}e^{-\frac{\left(b+5t^2\right)^2}{t^2}}e^{30\frac{\sqrt{2}a}{t}}e^{30\frac{\sqrt{2}\left(b+5t^2\right)}{t}}}{\pi t^2 e^{900}}$$

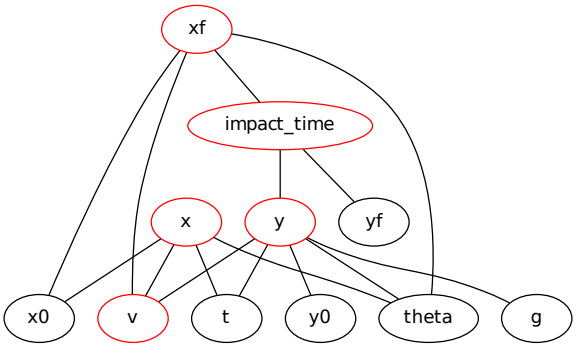
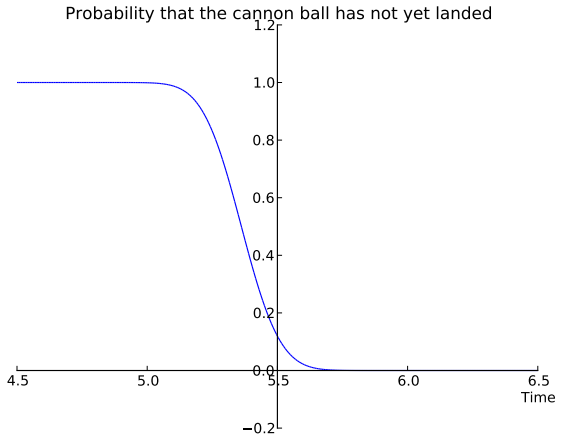


Fig. 4: *A graph of all the varibles in our system. Red variables are stochastic. Every variable that depends on the uncertain input, v, is red due to its dependence.*



Or we can plot the probability that the ball is still in the air at time t

```
>>> plot( P(y>yf), (t, 4.5, 6.5))
```

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 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.

This is an unfortunate though very common result. Mathematical models are usually far too complex to yield simple analytic solutions. I.e. this unresolved result is the common case. Fortunately computing integral expressions is a problem of very broad interest with many mature techniques. SymPy stats has successfully transformed a specialized and novel problem (uncertainty propagation) into a general and well studied one (computing integrals) to which we can apply general techniques.

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)*

Sampling

One method to approximate difficult integrals is through sampling.

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

```
>>> E(impact_time, numsamples=10000)
5.36178452172906
```

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.

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
```

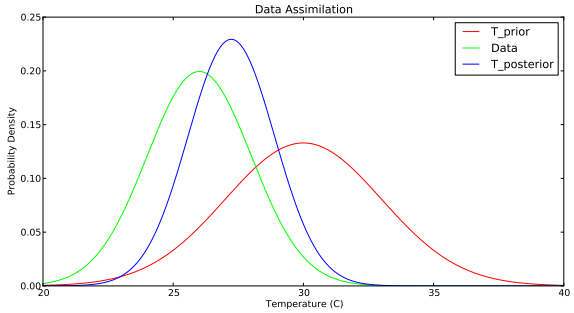


Fig. 5: *The prior, data, and posterior distributions of the temperature.*

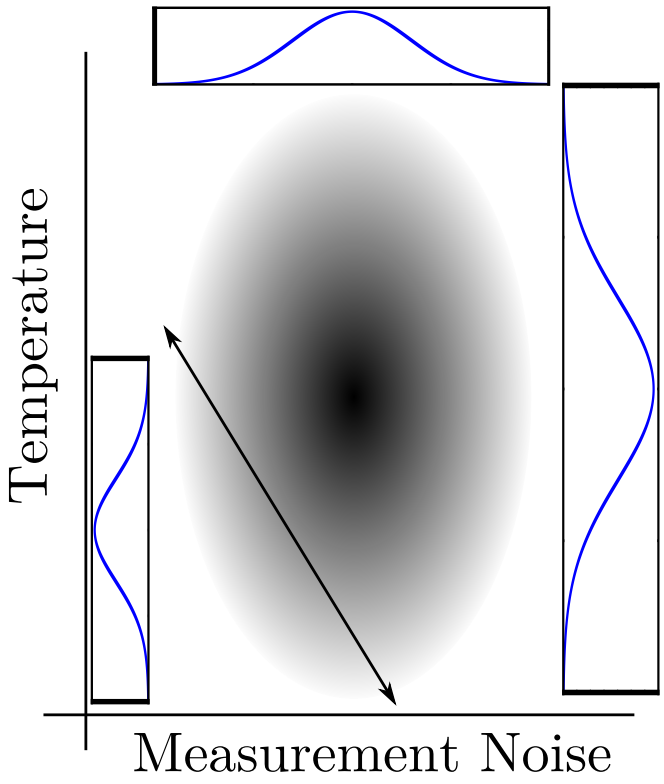


Fig. 6: *The joint prior distribution of the temperature and measurement noise. The constraint* `T + noise == 26` *(diagonal line) and the resultant posterior distribution of temperature on the left.*

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 6. 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 managed using

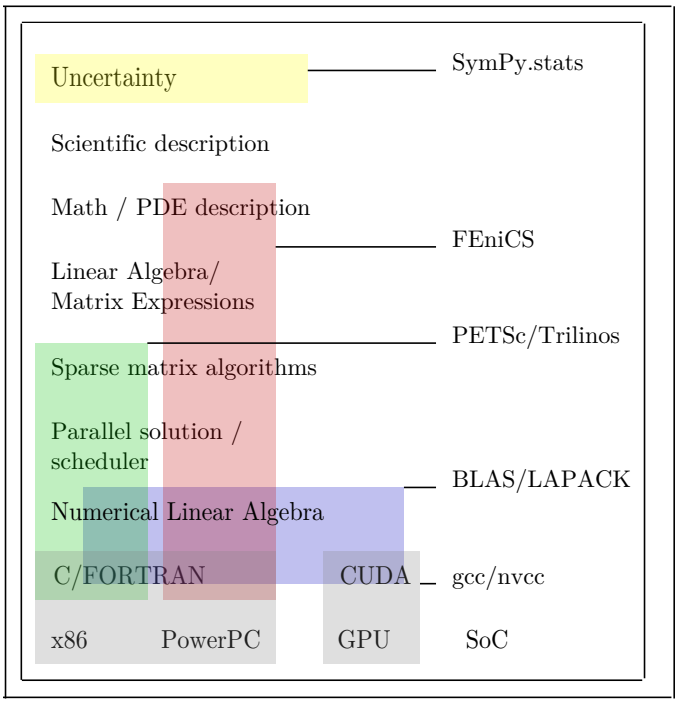


Fig. 7: *The scientific computing software stack. Various projects are displayed showing the range that they abstract. We pose that scientific computing needs more horizontal and thin layers in this image.*

core SymPy functionality.

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.

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.

Low-level Languages like C provide a standard interface for a range of conventional CPUs effectively abstracting low-level architecture details away from the common programmer.

Libraries such as BLAS and LAPACK provide an interface between linear algebra and optimized low-level code. These libraries provide an interface layer for a broad range of architecture (i.e. CPU-BLAS or GPU-cuBLAS both exist).

High quality implementations of vertical slices of the stack are available through higher level libraries such as PETSc and Trilinos or through code generation solutions such as FENICS. These projects provide end to end solutions but do not provide intermediate interface layers. They also struggle to generalize well to novel hardware.

Symbolic mathematical modeling attempts to serve as a thin horizontal interface layer near the top of this stack, a relatiely empty space at present.

SymPy stats is designed to be as vertically thin as possible. For example it transforms continuous 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.

Multivariate Normals produce Matrix Expressions

Other sympy.stats implementations generate similarly structured outputs. For example multivariate normal random variables found in `sympy.stats.mvnrv` generate matrix expressions. In the following example we describe a standard data assimilation task and view the resulting matrix expression.

```
mu = MatrixSymbol('mu', n, 1) # n by 1 mean vector
Sigma = MatrixSymbol('Sigma', n, n) # covariance matrix
X = MVNormal('X', mu, Sigma)

H = MatrixSymbol('H', k, n) # An observation operator
data = MatrixSymbol('data', k, 1)

R = MatrixSymbol('R', k, k) # covariance matrix for noise
noise = MVNormal('eta', ZeroMatrix(k, 1), R)

# Conditional density of X given  HX+noise==data
density(X , Eq(H*X+noise, data)  )
```

$$\mu = \begin{bmatrix}\mathbb{I} & \mathbf{0}\end{bmatrix}\left(\begin{bmatrix}\Sigma & \mathbf{0}\\ \mathbf{0} & R\end{bmatrix}\begin{bmatrix}H^T\\ \mathbb{I}\end{bmatrix}\left(\begin{bmatrix}H & \mathbb{I}\end{bmatrix}\begin{bmatrix}\Sigma & \mathbf{0}\\ \mathbf{0} & R\end{bmatrix}\begin{bmatrix}H^T\\ \mathbb{I}\end{bmatrix}\right)^{-1}\left(\begin{bmatrix}H & \mathbb{I}\end{bmatrix}\begin{bmatrix}\mu\\ \mathbf{0}\end{bmatrix} - data\right) + \begin{bmatrix}\mu\\ \mathbf{0}\end{bmatrix}\right)$$

$$\Sigma = \begin{bmatrix}\mathbb{I} & \mathbf{0}\end{bmatrix}\left(\mathbb{I} - \begin{bmatrix}\Sigma & \mathbf{0}\\ \mathbf{0} & R\end{bmatrix}\begin{bmatrix}H^T\\ \mathbb{I}\end{bmatrix}\left(\begin{bmatrix}H & \mathbb{I}\end{bmatrix}\begin{bmatrix}\Sigma & \mathbf{0}\\ \mathbf{0} & R\end{bmatrix}\begin{bmatrix}H^T\\ \mathbb{I}\end{bmatrix}\right)^{-1}\begin{bmatrix}H & \mathbb{I}\end{bmatrix}\right)\begin{bmatrix}\Sigma & \mathbf{0}\\ \mathbf{0} & R\end{bmatrix}\begin{bmatrix}\mathbb{I}\\ \mathbf{0}\end{bmatrix}$$

$$\mu = \mu + \Sigma H^T\left(R + H\Sigma H^T\right)^{-1}(H\mu - data)$$

$$\Sigma = \left(\mathbb{I} - \Sigma H^T\left(R + H\Sigma H^T\right)^{-1}H\right)\Sigma$$

Those familiar with data assimilation will recognize the Kalman Filter. This expression can now be passed as an input to other symbolic/numeric projects. Symbolic/numerical linear algebra is a vibrant and rapidly changing field. Because `sympy.stats` offers a clean interface layer it is able to easily engage with these developments. Matrix expressions form a clean interface layer in which uncertainty problems can be expressed and transferred to computational systems.

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 for heroic computations this approach falls short - maximal speedup often requires optimizing the whole problem at once.

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 symbolic code.

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

REFERENCES

[Sym] SymPy Development Team (2012). SymPy: Python library for symbolic mathematics URL http://www.sympy.org.

[Roc12] M. Rocklin, A. Terrel, *Symbolic Statistics with SymPy* Computing in Science & Engineering, June 2012

[Joy11] D. Joyner, O. Čertík, A. Meurer, B. Granger, *Open source computer algebra systems: SymPy* ACM Communications in Computer Algebra, Vol 45 December 2011