Repository of the code and slides:

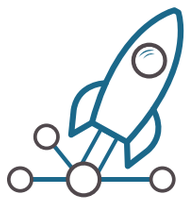
https://bit.ly/2JrHNLd

ALL THAT LIKELIHOOD

with PyMC3

Junpeng Lao

July 2018 @ Berlin

Powered by  PyMC3

PyMC3 theano

From the [PyMC3 documentation](#):

PyMC3 is a Python package for Bayesian statistical modeling and Probabilistic Machine Learning which focuses on **advanced Markov chain Monte Carlo** and variational fitting algorithms. Its flexibility and extensibility make it applicable to a large suite of problems.

Why PyMC3?

Easy to interact with different components of your model.

all categories ▸ | Latest | Top | Categories | + New Topic

Topic	Category	Users	Replies	Views	Activity
Setting pymc3 random seed at once	Questions		1	14	2h
☑ Using WAIC to compure Log Predictive Accuracy	Questions		3	6	2h
last visit					
Concepts of Parameter Estimation and Predictions, and Out of Sample Predicted Probability for Logistic Regression	Questions	H	5	36	4h
☑ ValueError: Bad initial energy: inf. The model might be misspecified error replicating Multilevel Regression and Poststratification with PyMC3 notebook	Questions	G	2	13	4h
Metropolis: ideal balance between slow-mixing and high rejection rate?	Questions		3	13	5h
☑ The pymc3 way: Linear regression and inferring given posterior/trace	Questions		4	26	18h
☑ Autocorrelation in a model	Questions	H	8	35	22h

https://discourse.pymc.io/

Some haunting questions:

- Why do we sample from the posterior? And how?
- Why does reparameterization “works”?

$$x \sim \text{Normal}(\mu, \sigma) \qquad \epsilon \sim \text{Normal}(0,1), x = \mu + \sigma\epsilon$$

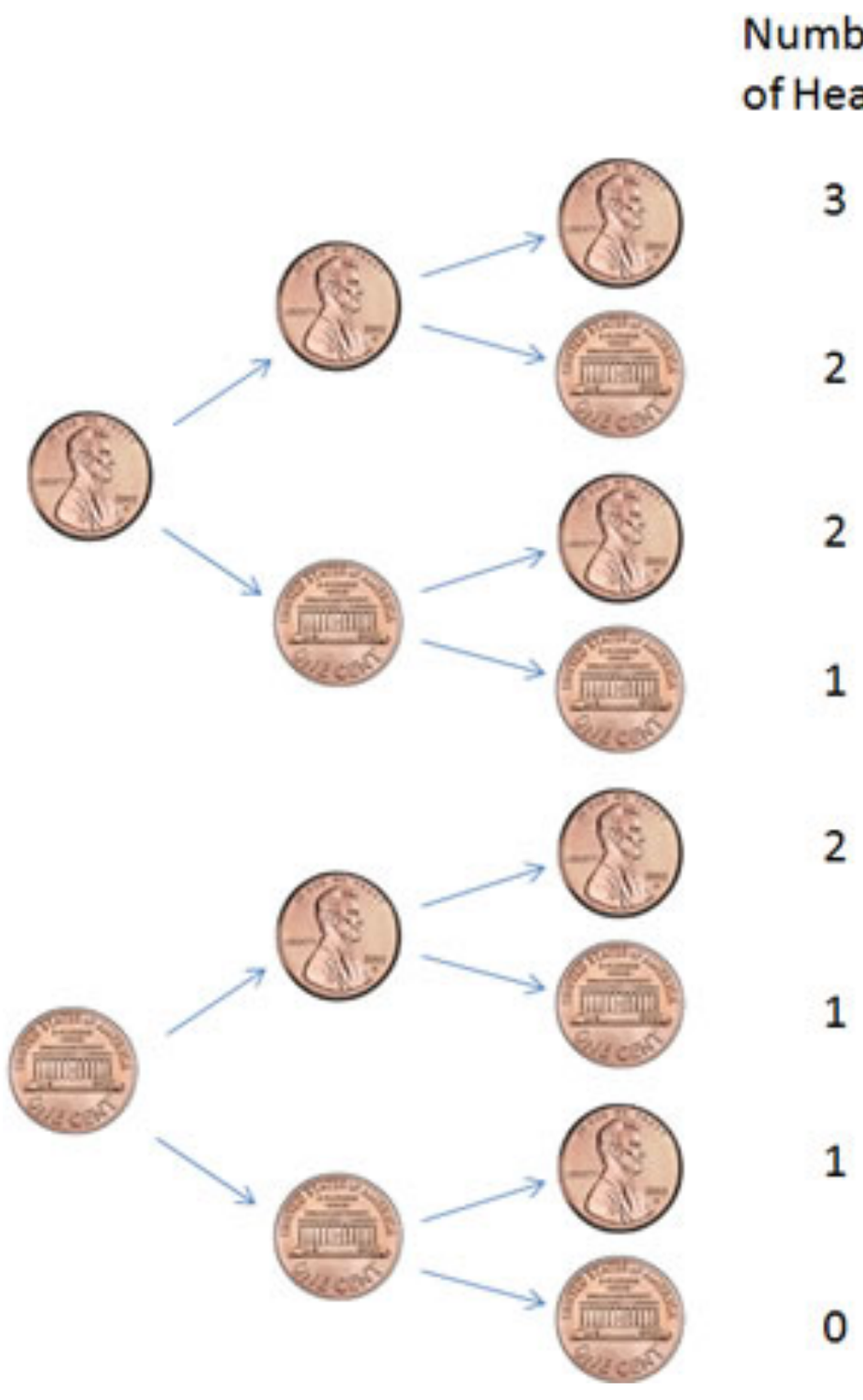
It all can be better understand…

- Probability density/mass function and likelihood function
- Computing Likelihood in PyMC3

Repository of the code and slides:

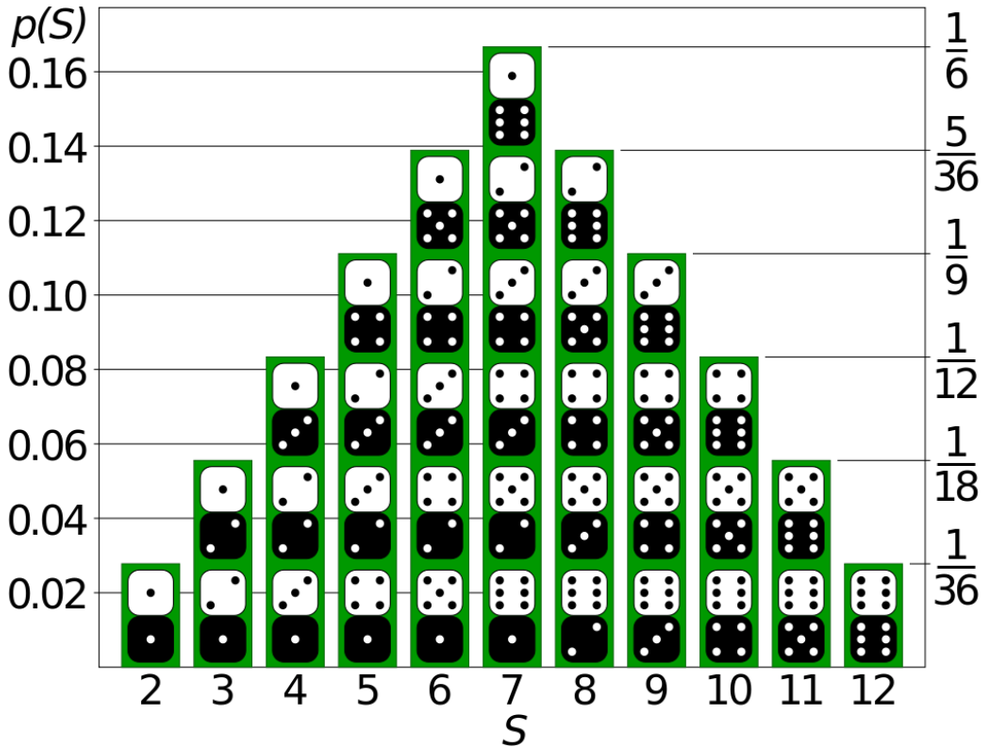
https://bit.ly/2JrHNLd

Random Variable

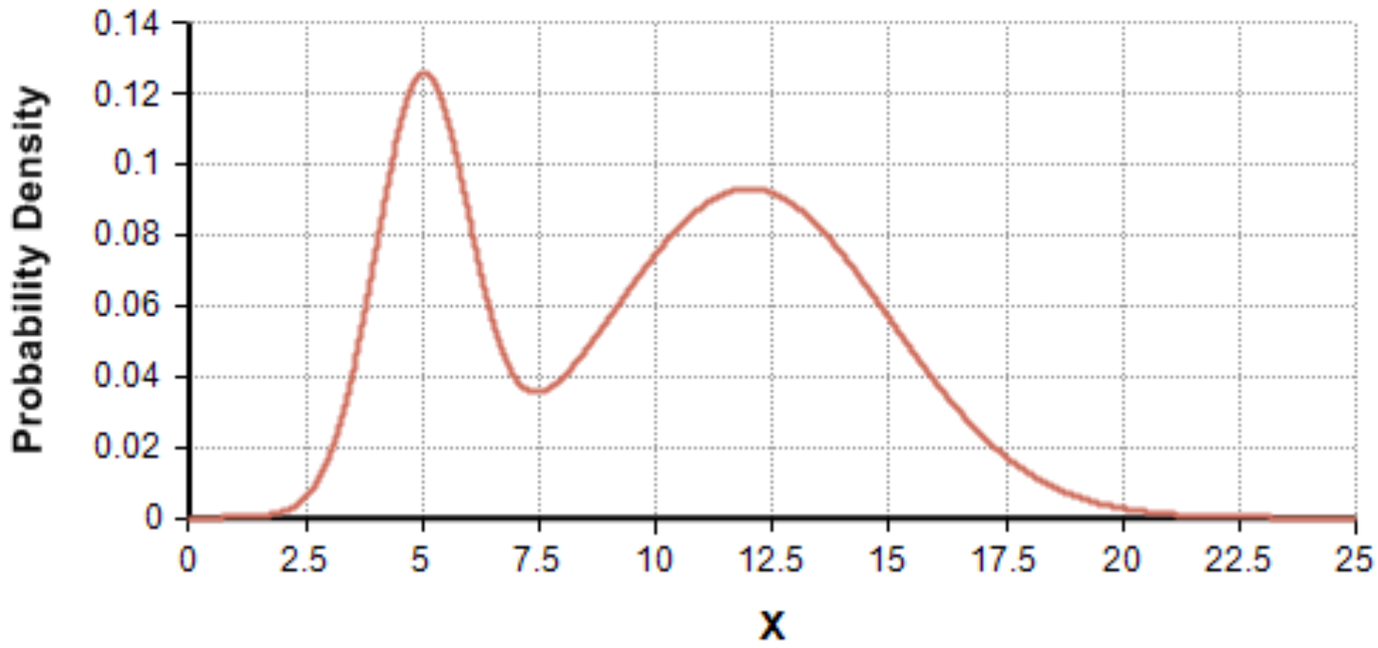


Probability distribution

Discrete case:



Continuous case



https://en.wikipedia.org/wiki/Probability_distribution

Probability theory

Measure-Theoretical Foundations of Probability Theory

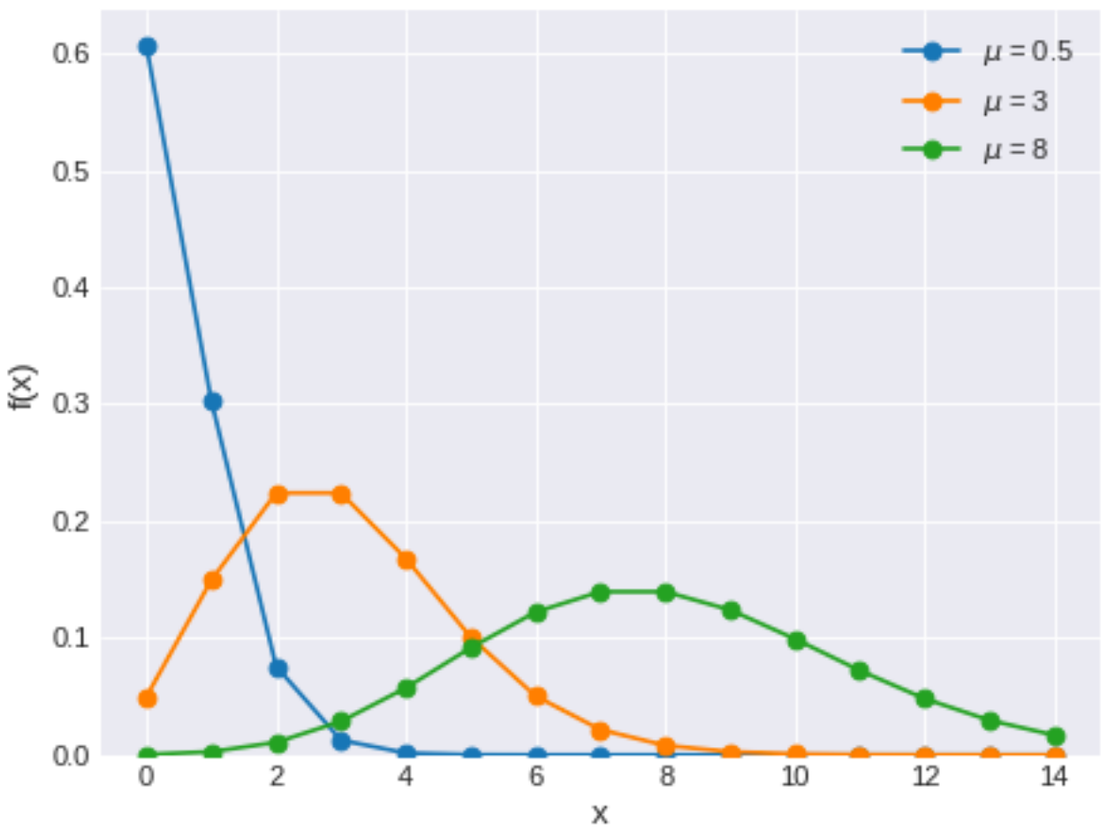
- Measure theory
  - σ-algebra
- Measurable mapping
  - pushforward measure
- Integral
  - expectation

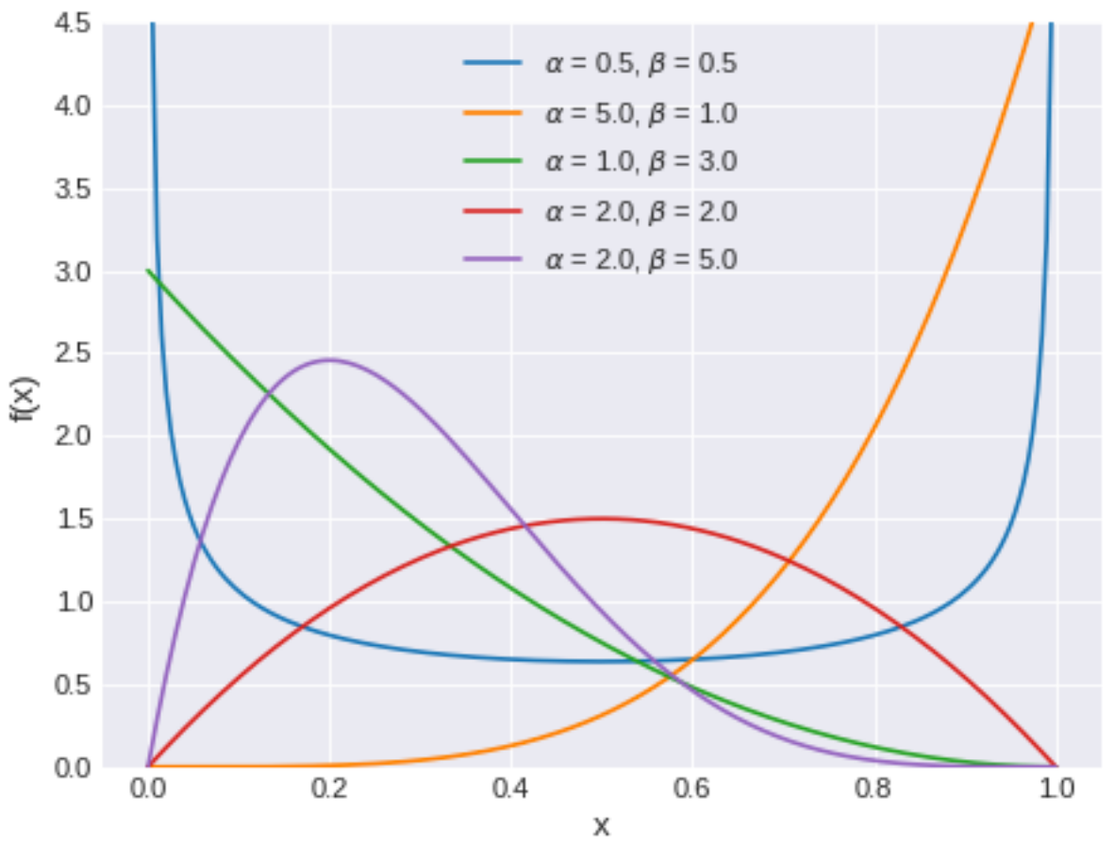
https://betanalpha.github.io/assets/case_studies/probability_theory.html

Probability distribution

$$f(x \mid \mu) = \frac{e^{-\mu}\mu^{x}}{x!}$$

$$f(x \mid \alpha, \beta) = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha, \beta)}$$





Random Variable

$$y \sim \pi(\theta)$$

"Alice flipped a coin 5 times and observed 3 heads"

- Setting 1: Flip total 5 times and records the output.

`Binomial(y|p,n)`

- Setting 2: Flip until observed 3 heads.

`Negative binomial(n|p,y)`

PMF and Likelihood

- Setting 1: Alice flip a coin 10 times and records the output.

`Binomial(y|p,n)`

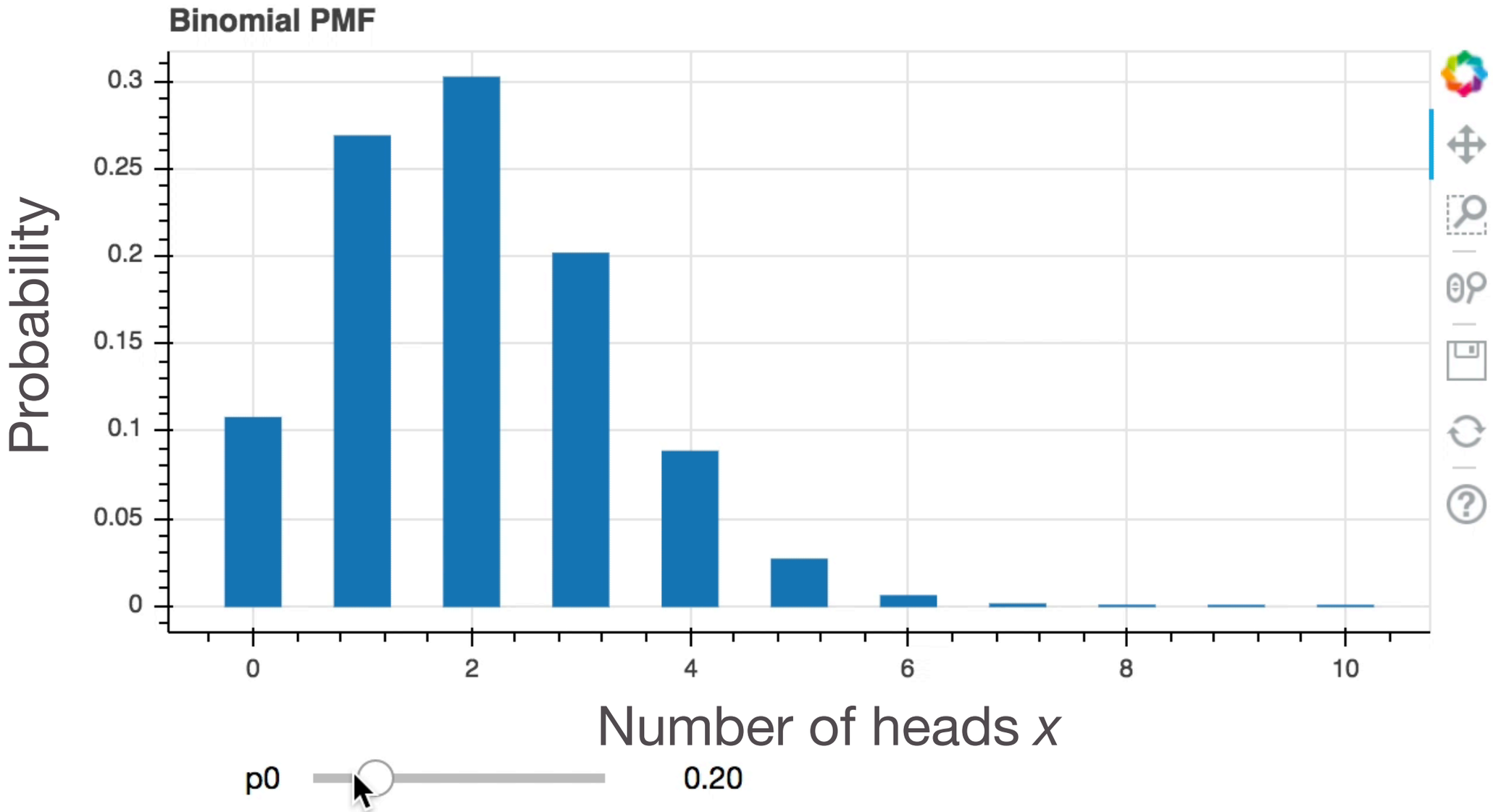
$$\pi(x \mid n, p) = \binom{n}{x} p^x (1-p)^{n-x}$$

$$f(x, n, p) = \binom{n}{x} p^x (1-p)^{n-x}$$

Likelihood_visual_demo.ipynb

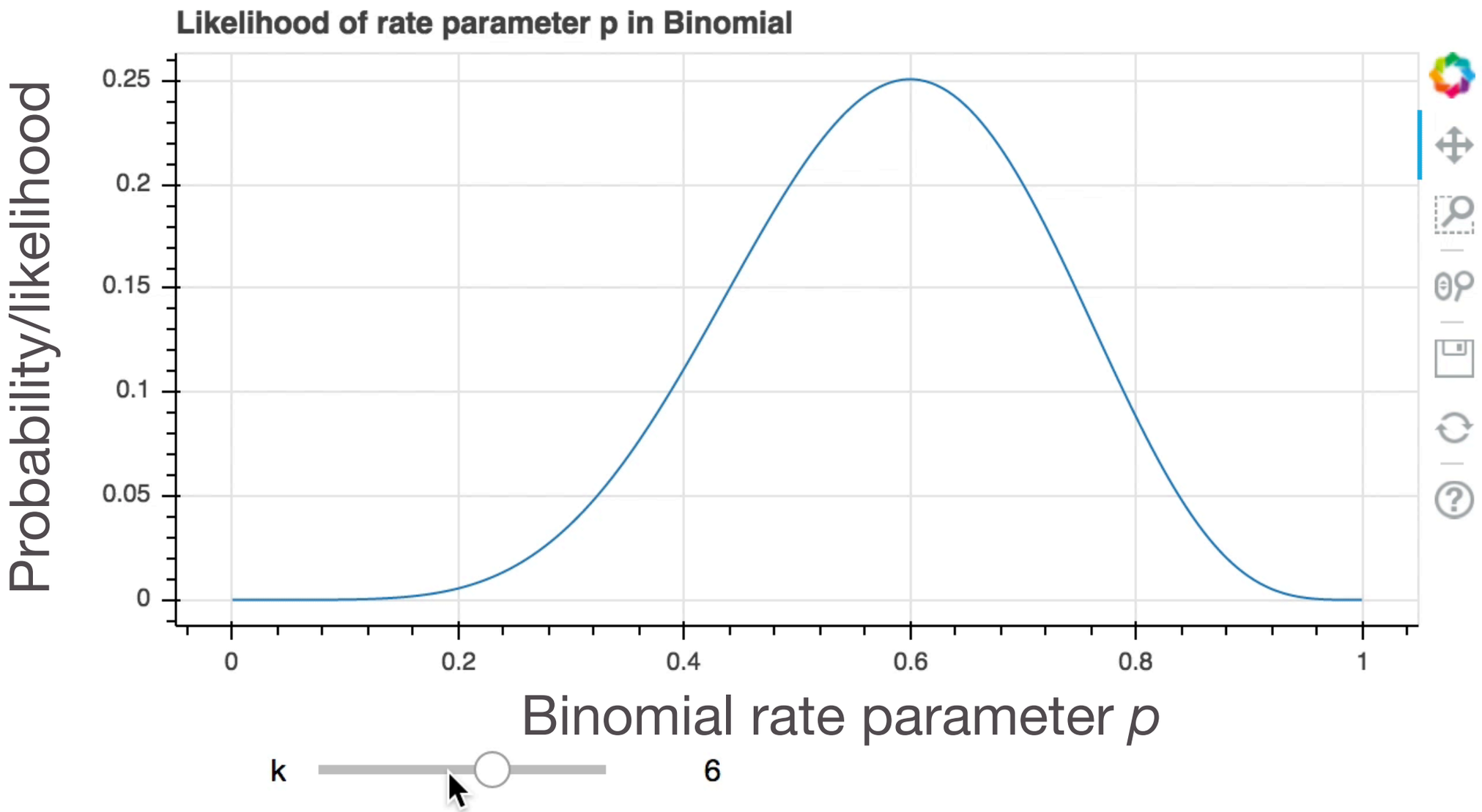
PMF and Likelihood

```
import scipy.stats as st
n0, p0 = 10, .2
x0 = np.arange(n0+1)
y0 = st.binom.pmf(x0, n0, p0)
```



PMF and Likelihood

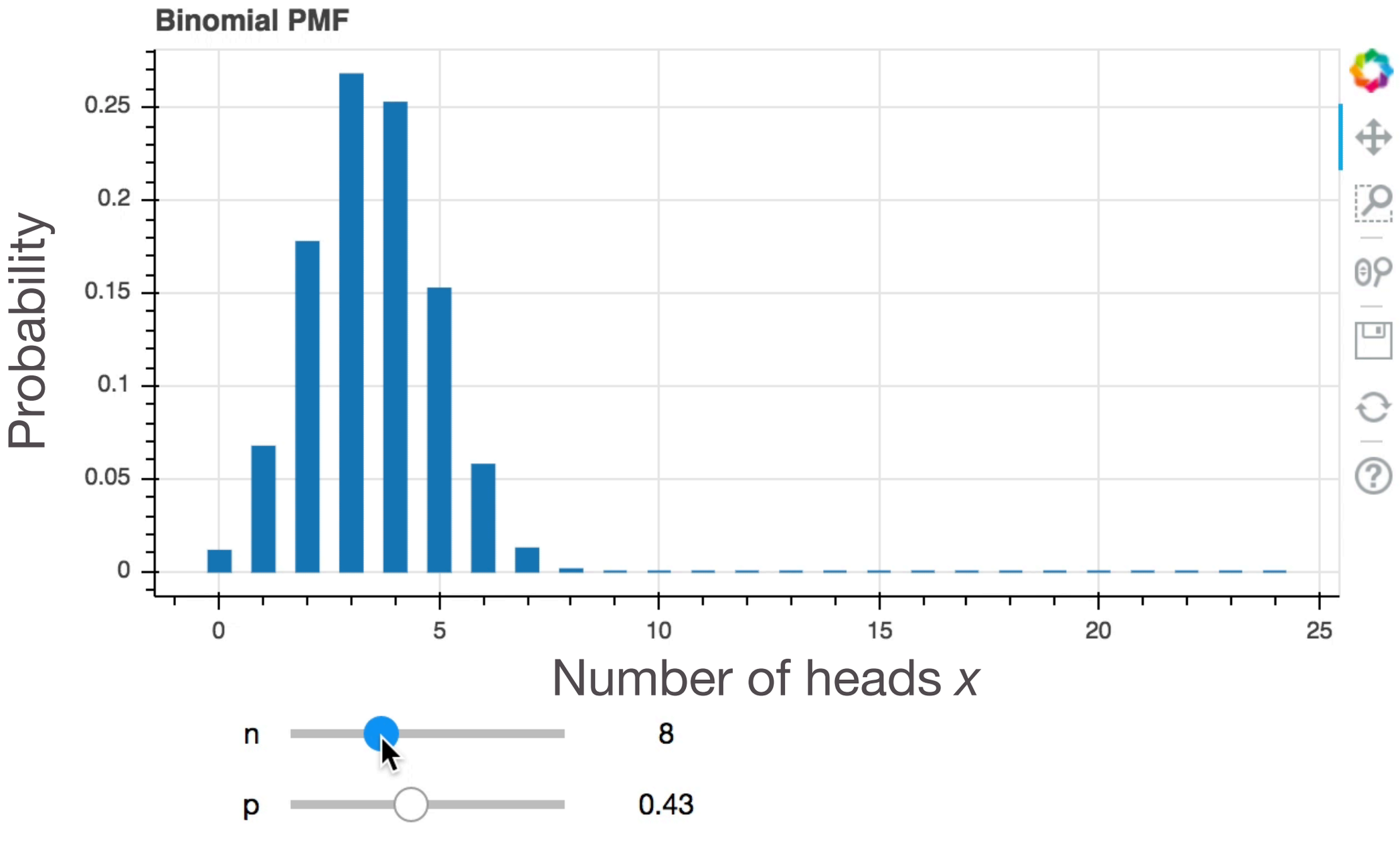
```
k = 6
xp = np.linspace(0., 1., 200)
ll = st.binom.pmf(k, n0, xp)
```



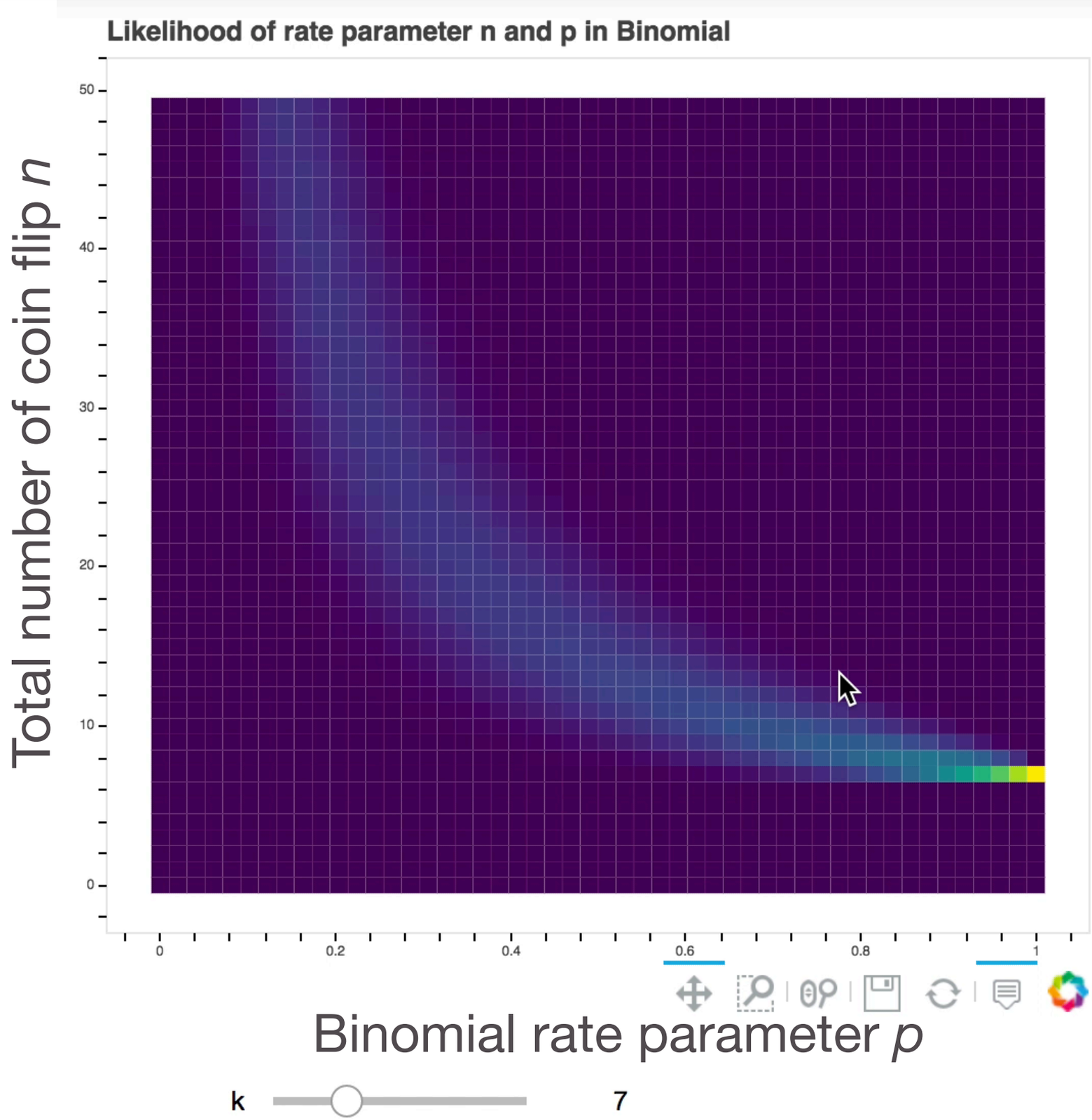

```
Observed k = 6
```

PMF and Likelihood

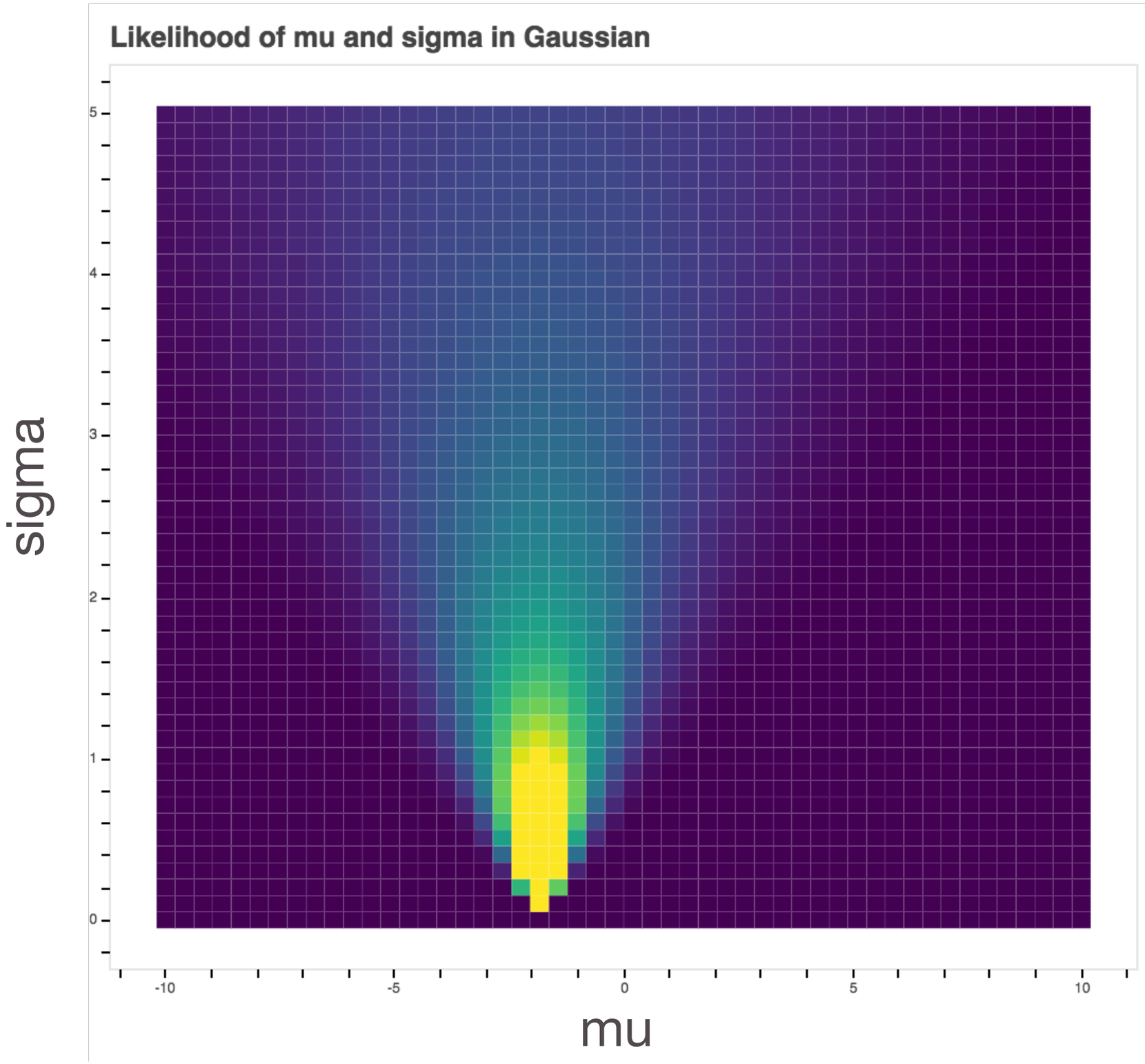
$$\pi(x \mid n, p) = \binom{n}{x} p^x (1-p)^{n-x}$$



PMF and Likelihood



Gaussian Likelihood



PMF and Likelihood

- Setting 1: Alice flip a coin 10 times and records the output.

`Binomial(y|p,n)`

$$\pi(x \mid n, p) = \binom{n}{x} p^x (1-p)^{n-x}$$

$$f(x, n, p) = \binom{n}{x} p^x (1-p)^{n-x}$$

```
st.binom.pmf(k, n, p)
```

Combining probabilities

Summary of probabilities

Event	Probability
A	$P(A) \in [0,1]$
not A	$P(A^\complement) = 1 - P(A)$
A or B	$P(A \cup B) = P(A) + P(B) - P(A \cap B)$ $P(A \cup B) = P(A) + P(B)$ if A and B are mutually exclusive
A and B	$P(A \cap B) = P(A\|B)P(B) = P(B\|A)P(A)$ $P(A \cap B) = P(A)P(B)$ if A and B are independent
A given B	$P(A \mid B) = \frac{P(A \cap B)}{P(B)} = \frac{P(B\|A)P(A)}{P(B)}$

https://en.wikipedia.org/wiki/Probability

Sum rule and product rule

Sum rule:

$$P(X) = \int_Y P(X, Y)$$

Product rule:

$$P(X, Y) = P(X \mid Y)P(Y)$$

Likelihood of observing:

X = [0, 0, 0, 0, 1, 1] with p = .7

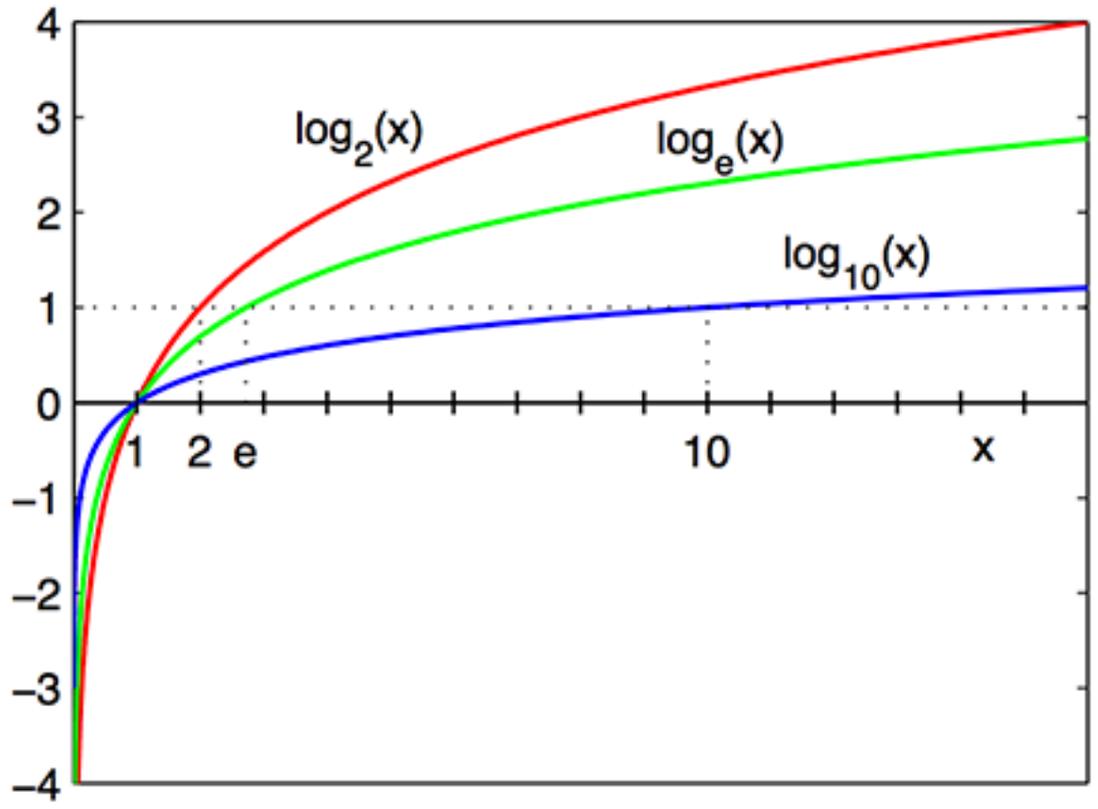
```
p = .7
y = np.asarray([0, 0, 0, 0, 1, 1])

prob = [p if y_ == 1 else (1-p) for y_ in y]
prob = np.cumprod(prob)
prob
```

```
array([0.3     , 0.09    , 0.027   , 0.0081  , 0.00567 , 0.003969])
```

```
prob2 = np.cumprod(st.bernoulli.pmf(y, p))
prob2
```

```
array([0.3     , 0.09    , 0.027   , 0.0081  , 0.00567 , 0.003969])
```

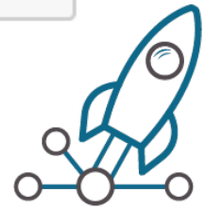
Log-Likelihood




```
np.cumsum(np.log(st.bernoulli.pmf(y, p)))
```

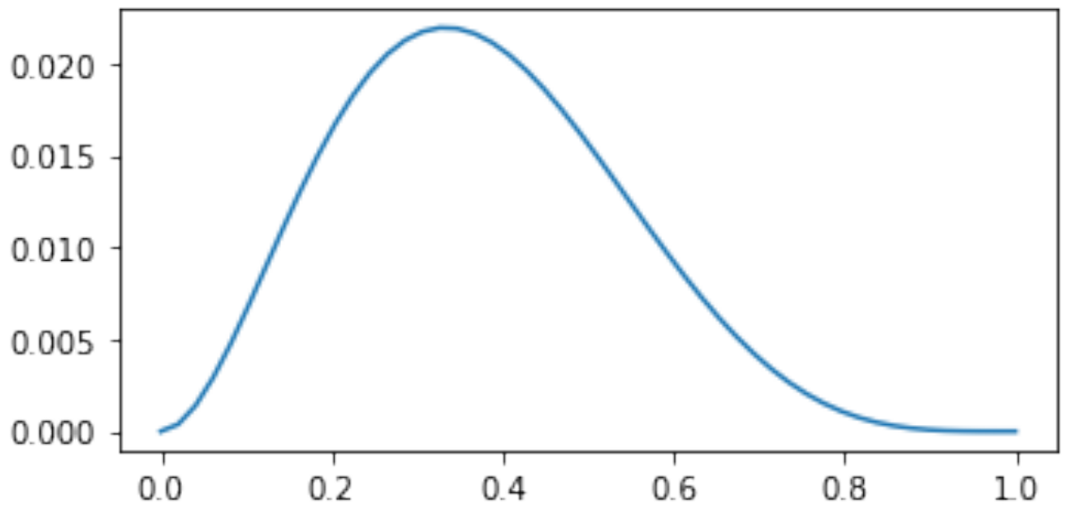
```
array([-1.2039728 , -2.40794561, -3.61191841, -4.81589122, -5.17256616,
       -5.52924111])
```

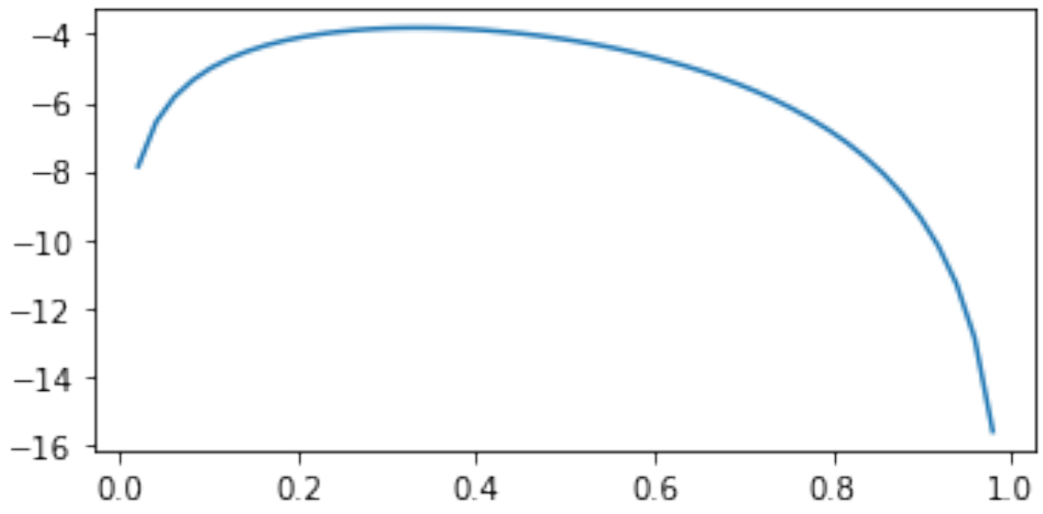
```
st.bernoulli.logpmf(y, p).cumsum()
```



Combining Likelihood

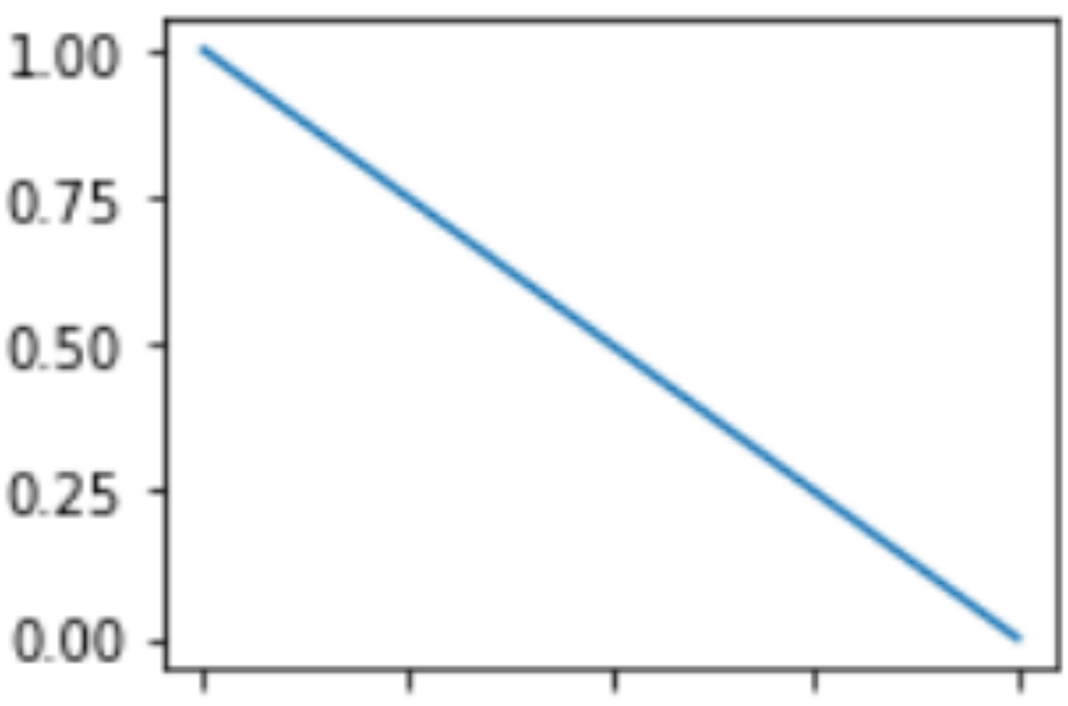
```
X = [0, 0, 0, 0, 1, 1]
lambda p: st.bernoulli.pmf(y, p).prod()
```



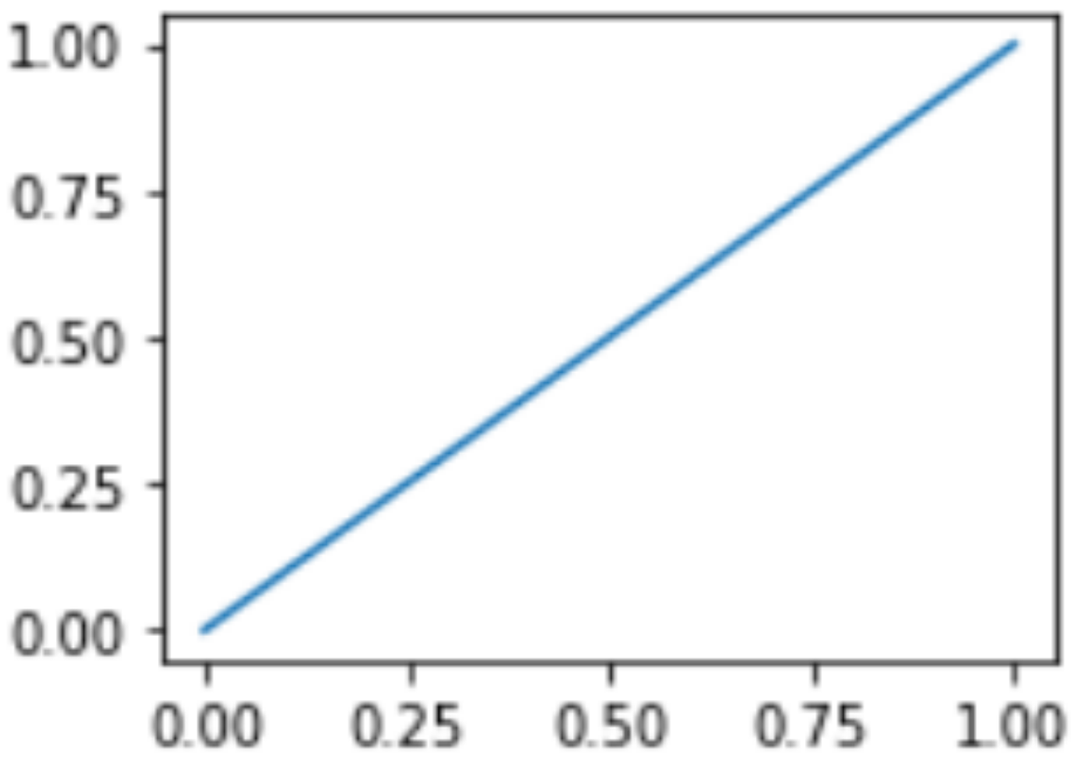


Likelihood of 1 coin flip

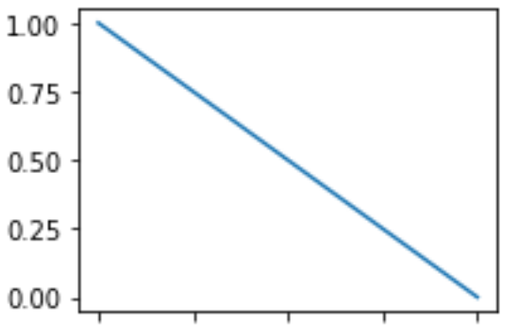
`x = [0]`

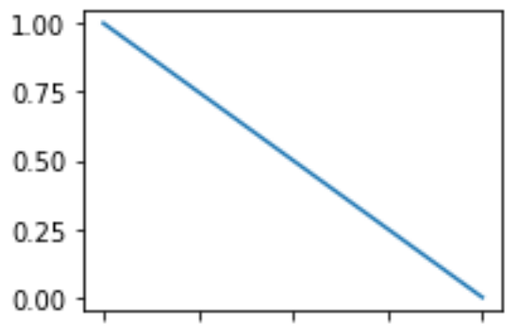


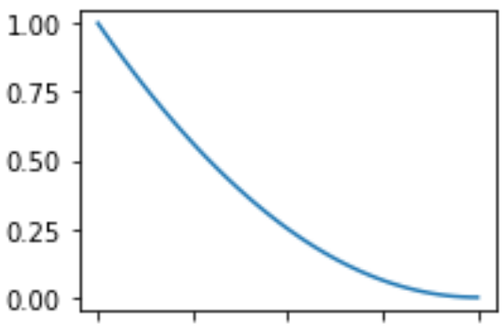
`x = [1]`



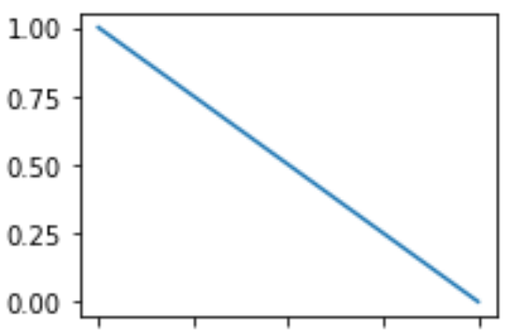
Update of Likelihood function

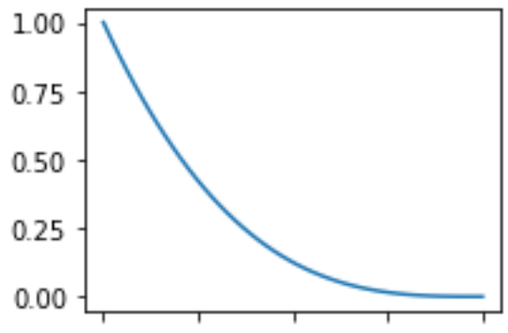


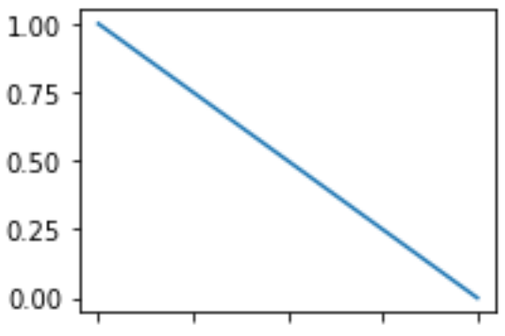


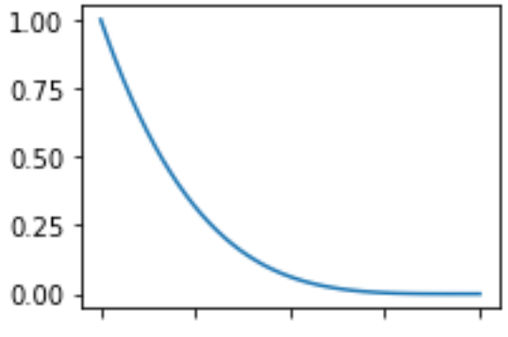



```
x = [0, 0, 0, 0, 1, 1]
```



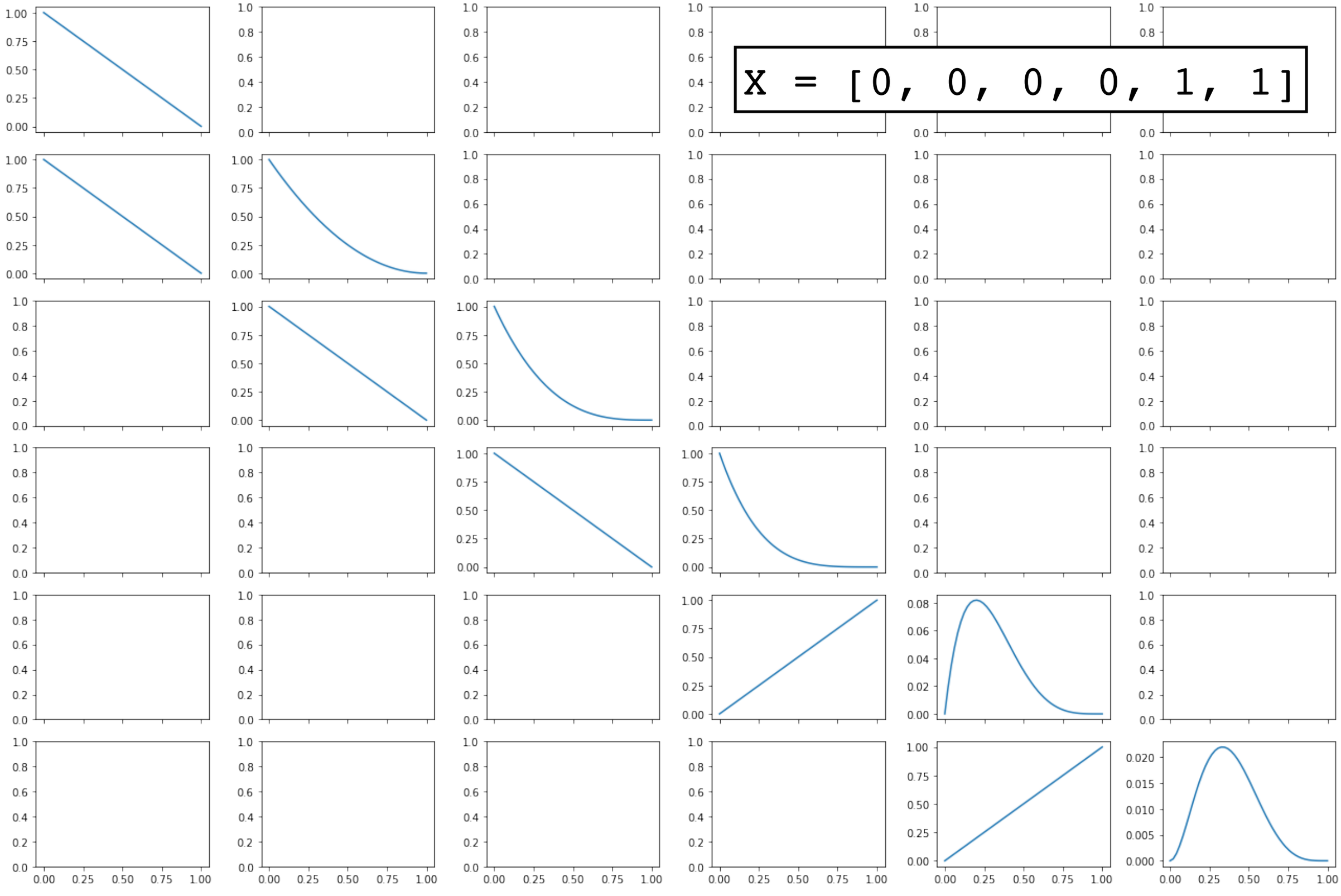






...

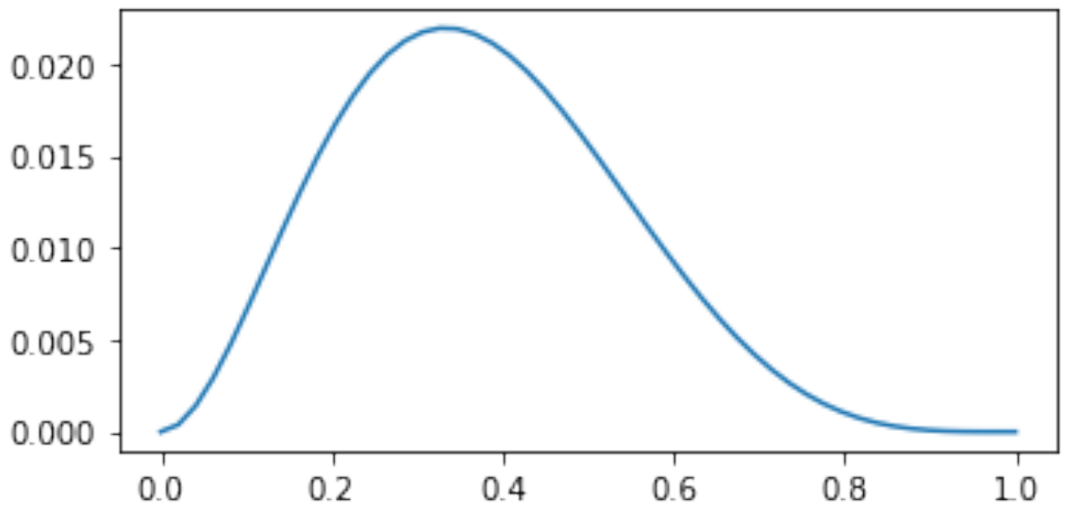
Update of Likelihood function

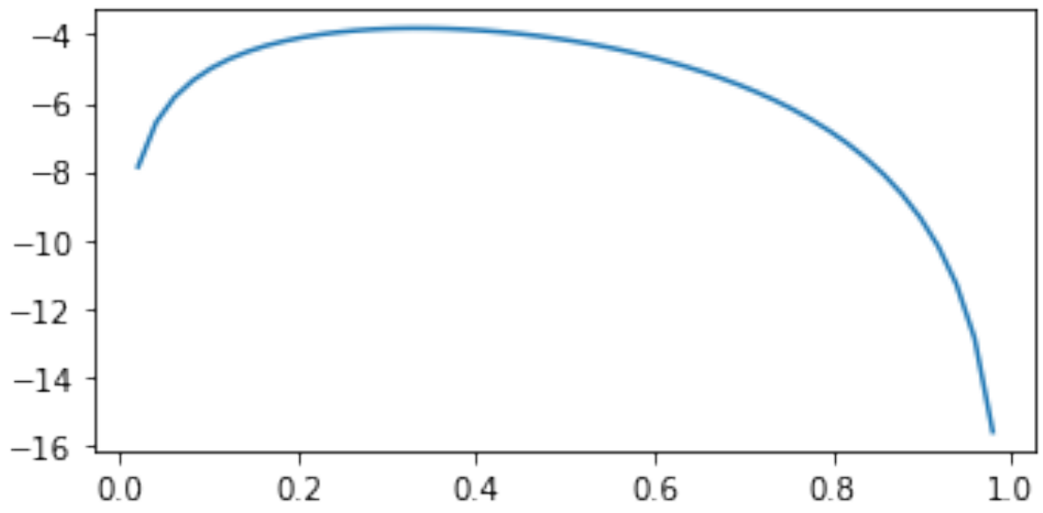


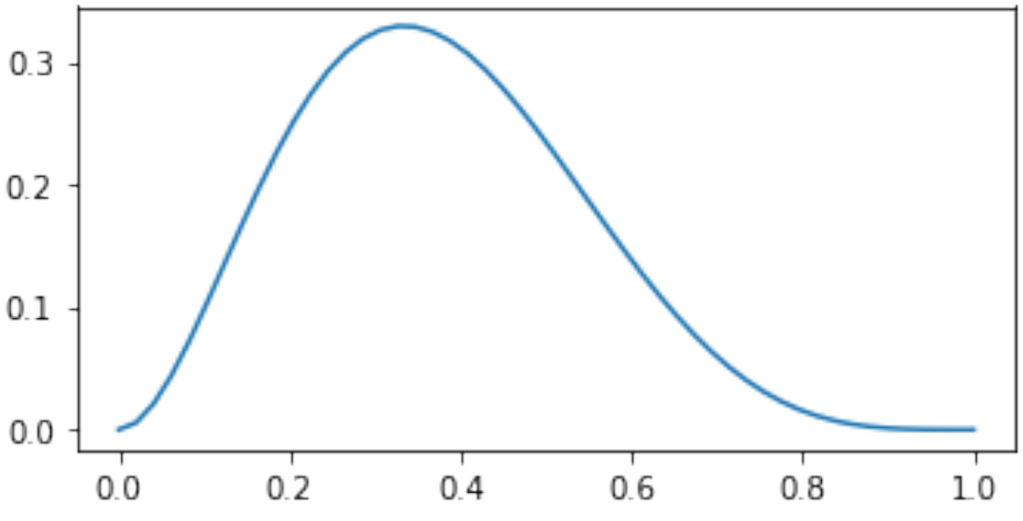
Likelihood function

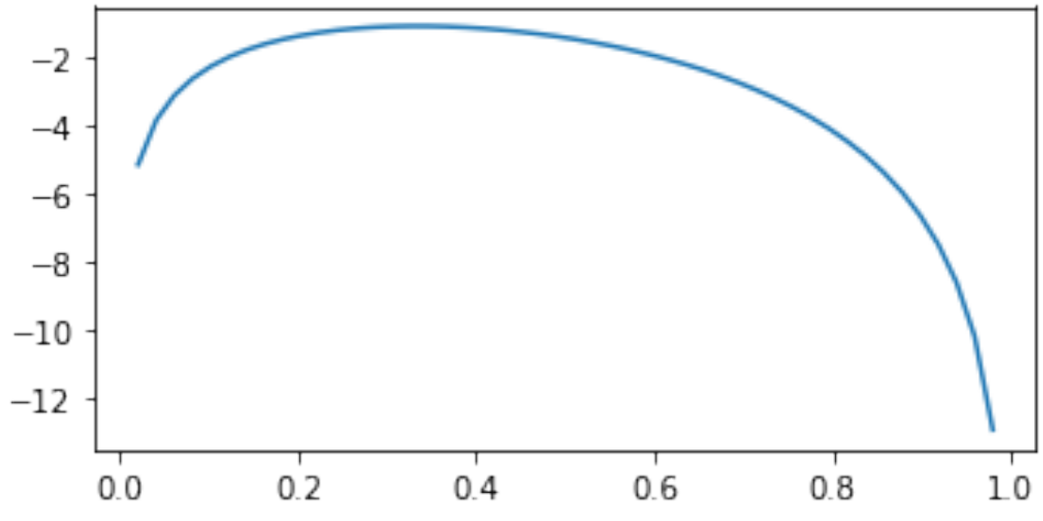
```
lambda p: st.bernoulli.pmf(y, p).prod()
```

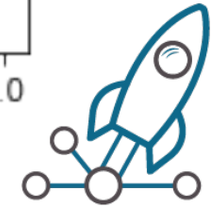
```
lambda p: st.binomial.pmf(y.sum(), len(y), p).prod()
```





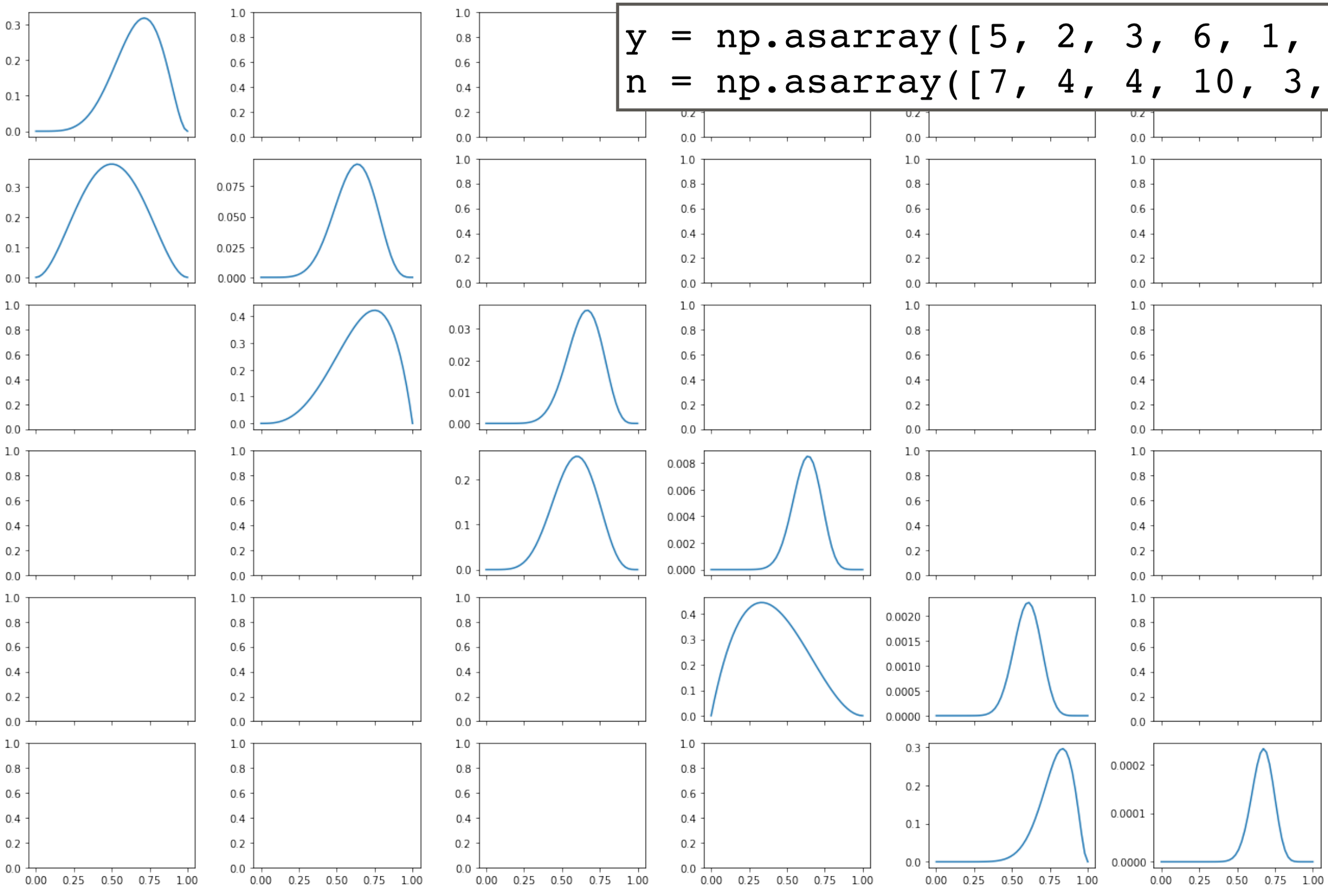




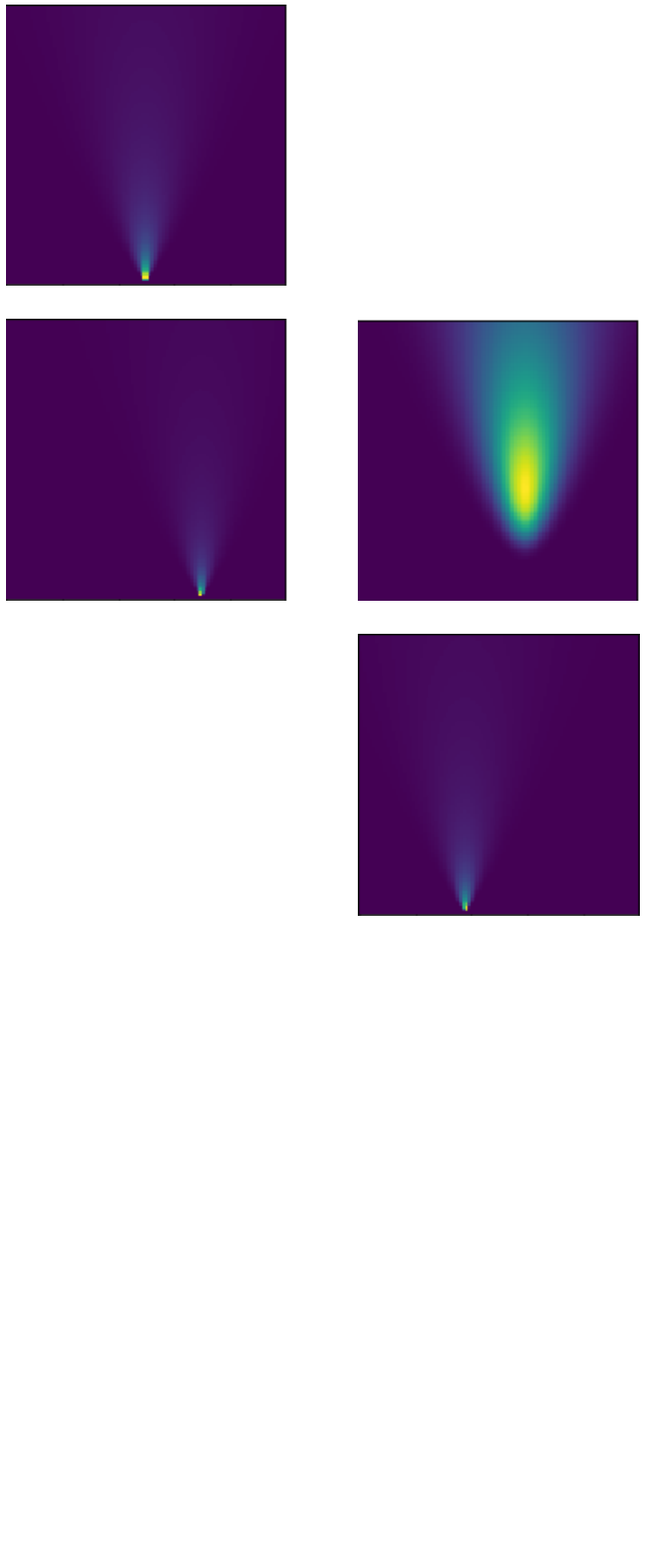


Update, or cumulating information

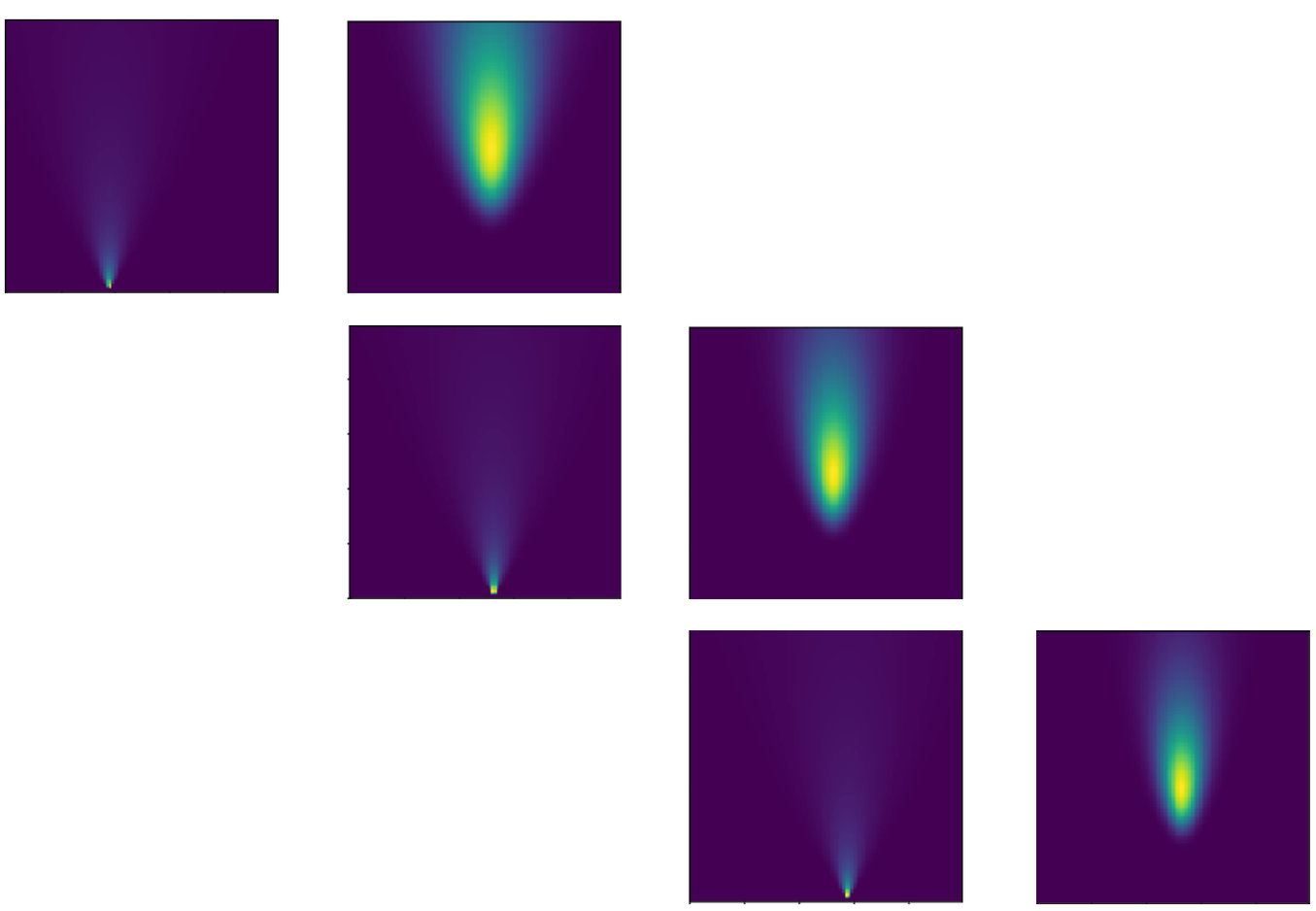
```
y = np.asarray([5, 2, 3, 6, 1, 10])
n = np.asarray([7, 4, 4, 10, 3, 12])
```



A Gaussian likelihood example



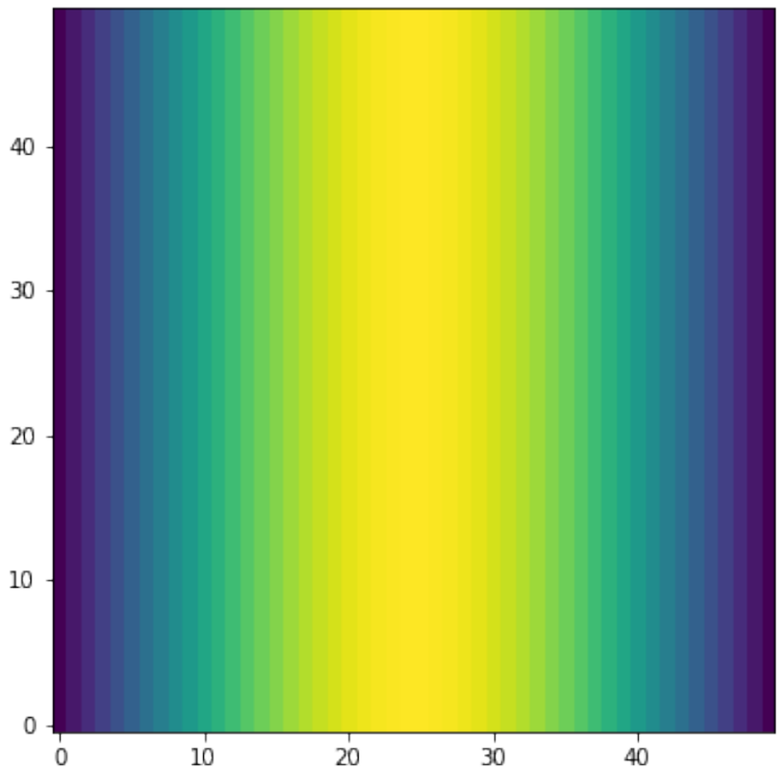
```
y = np.asarray([0., 2., -1.2, 0.3, .8])
mu = np.linspace(-5., 5., 100)
sd = np.linspace(0.001, 2.5, 100)
```

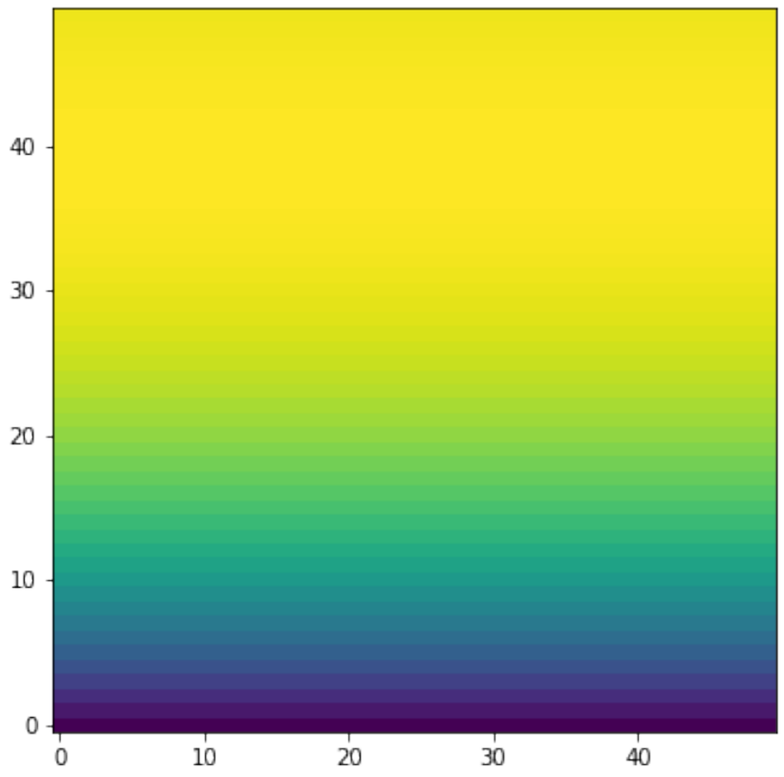


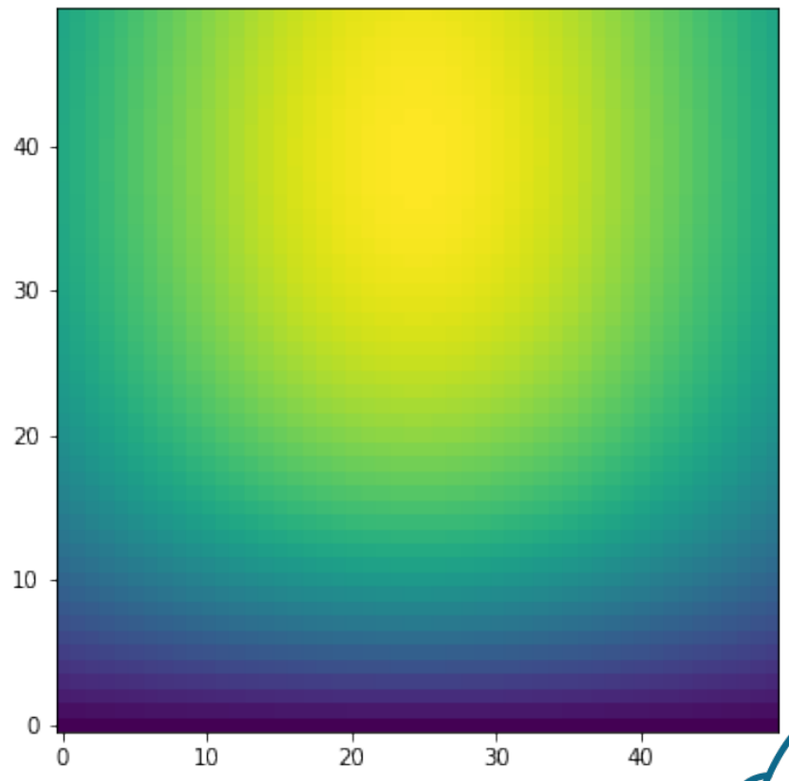
Priors

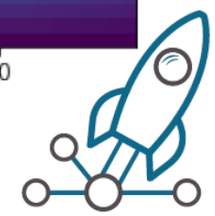
A and B	$P(A \cap B) = P(A\|B)P(B) = P(B\|A)P(A)$ $P(A \cap B) = P(A)P(B)$ if A and B are independent

```
mu_prior = st.norm.pdf(mu, 0, 5)
sd_prior = st.gamma.pdf(sd, 2., scale=1.0/.5)
```

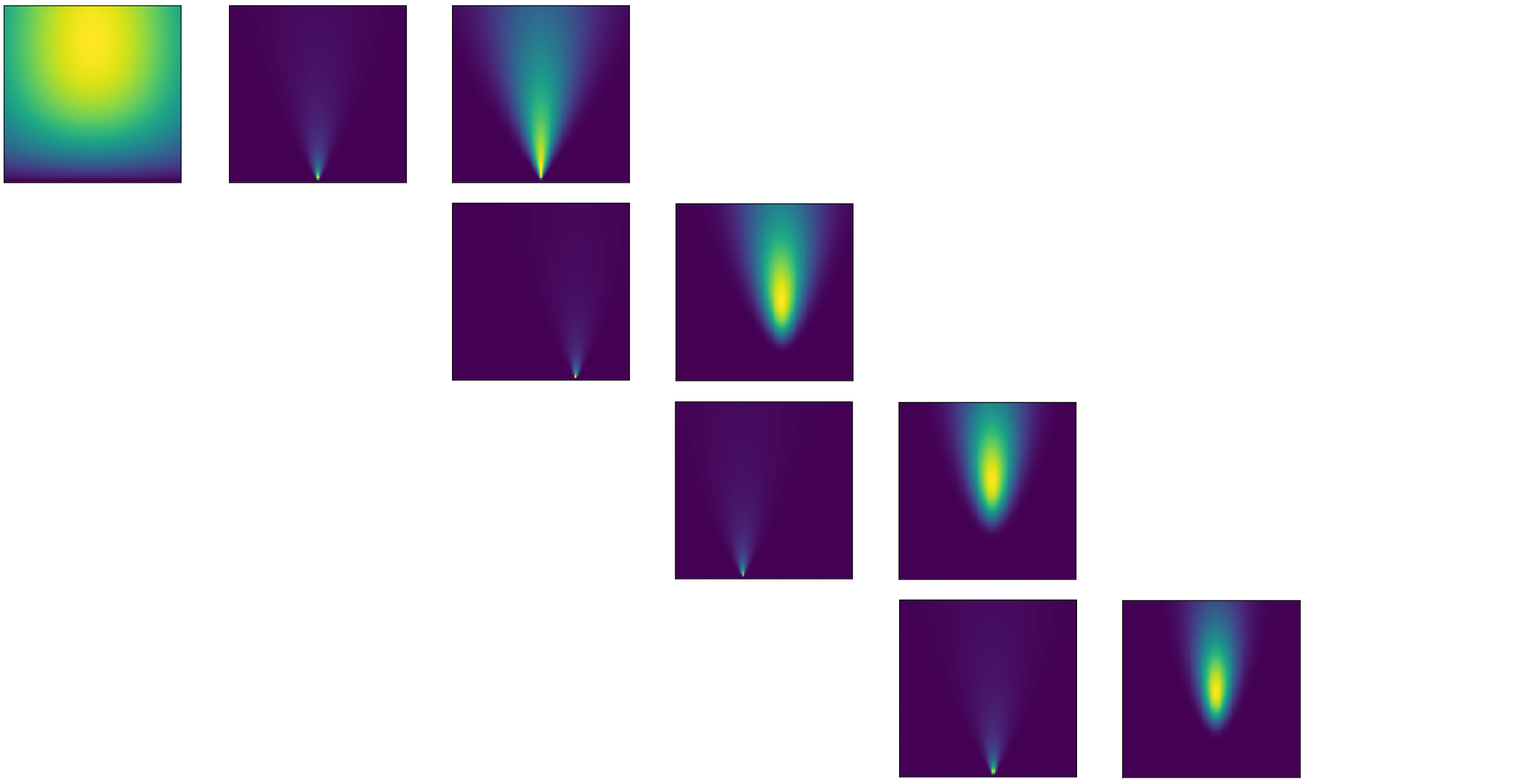






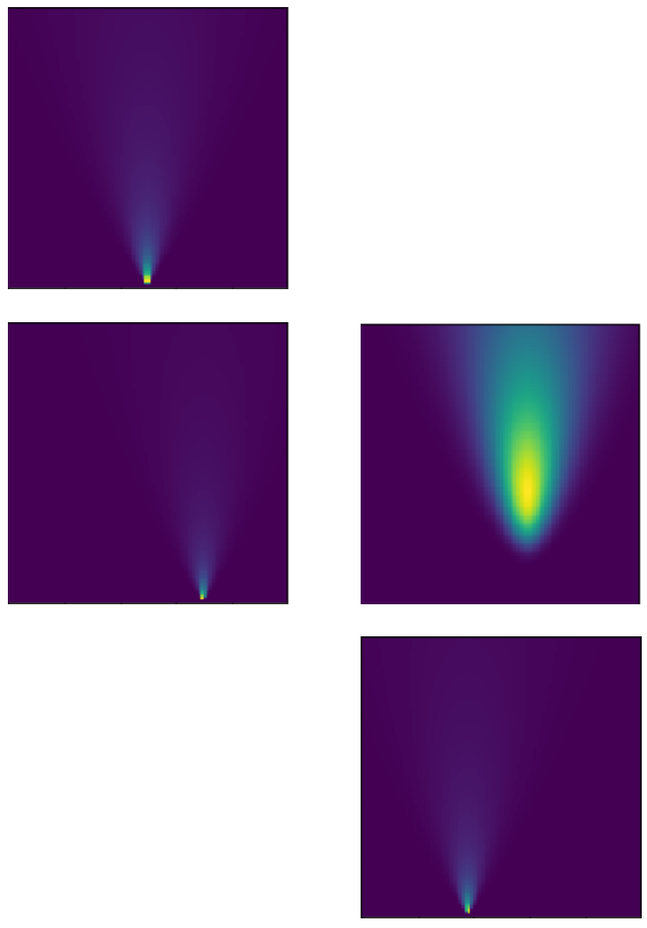


Update, or cumulating information

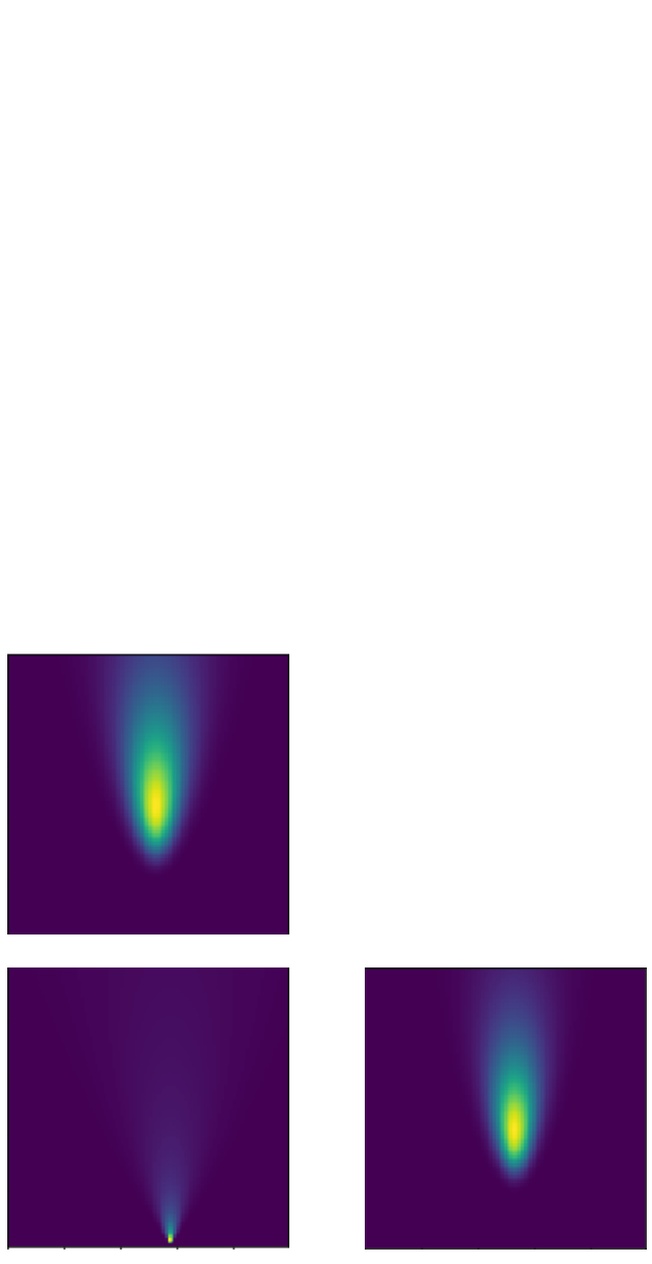


```
y = np.asarray([0., 2., -1.2, 0.3, .8])
mu = np.linspace(-5., 5., 100)
sd = np.linspace(0.001, 2.5, 100)
```

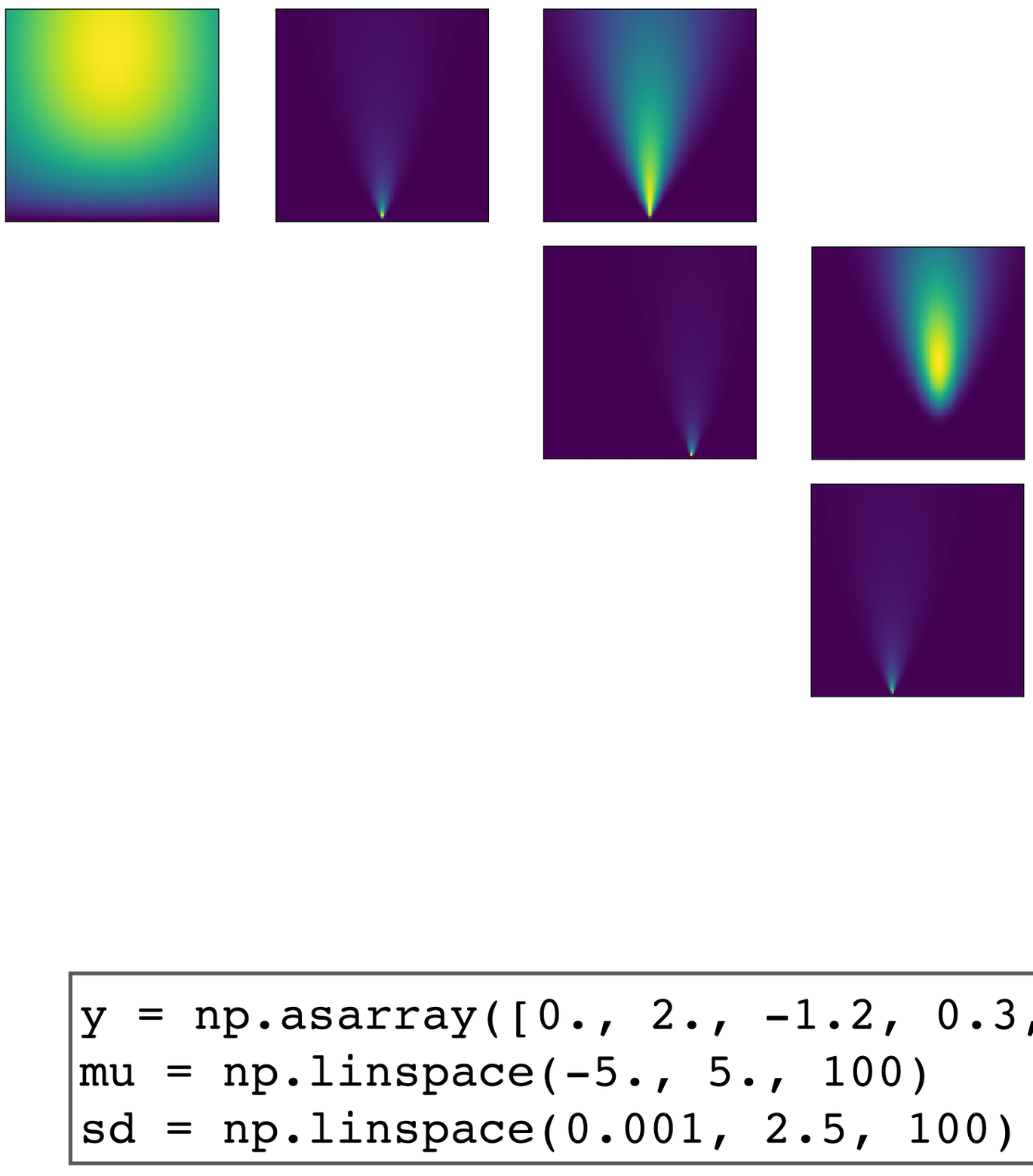
As comparison…



```
y = np.asarray([0., 2., -1.2, 0.3, .8])
mu = np.linspace(-5., 5., 100)
sd = np.linspace(0.001, 2.5, 100)
```



As comparison…



```
y = np.asarray([0., 2., -1.2, 0.3, .8])
mu = np.linspace(-5., 5., 100)
sd = np.linspace(0.001, 2.5, 100)
```

Coin flip in PyMC3

```
k = 7
k ~ Binomial(n=n, p=p)
n ~ DiscreteUniform(0, 25)
p ~ Uniform(0., 1.)
```

```
with pm.Model() as m:
    n = pm.DiscreteUniform('n', 0, 25)
    p = pm.Uniform('p', 0., 1., transform=None)
    y = pm.Binomial('k', n, p, observed=7)
```

Coin flip in PyMC3

```
with pm.Model() as m:
    n = pm.DiscreteUniform('n', 0, 25)
    p = pm.Uniform('p', 0., 1., transform=None)
    y = pm.Binomial('k', n, p, observed=7)
```

```
point = m.test_point
{'n': array(12), 'p': array(0.5)}
```

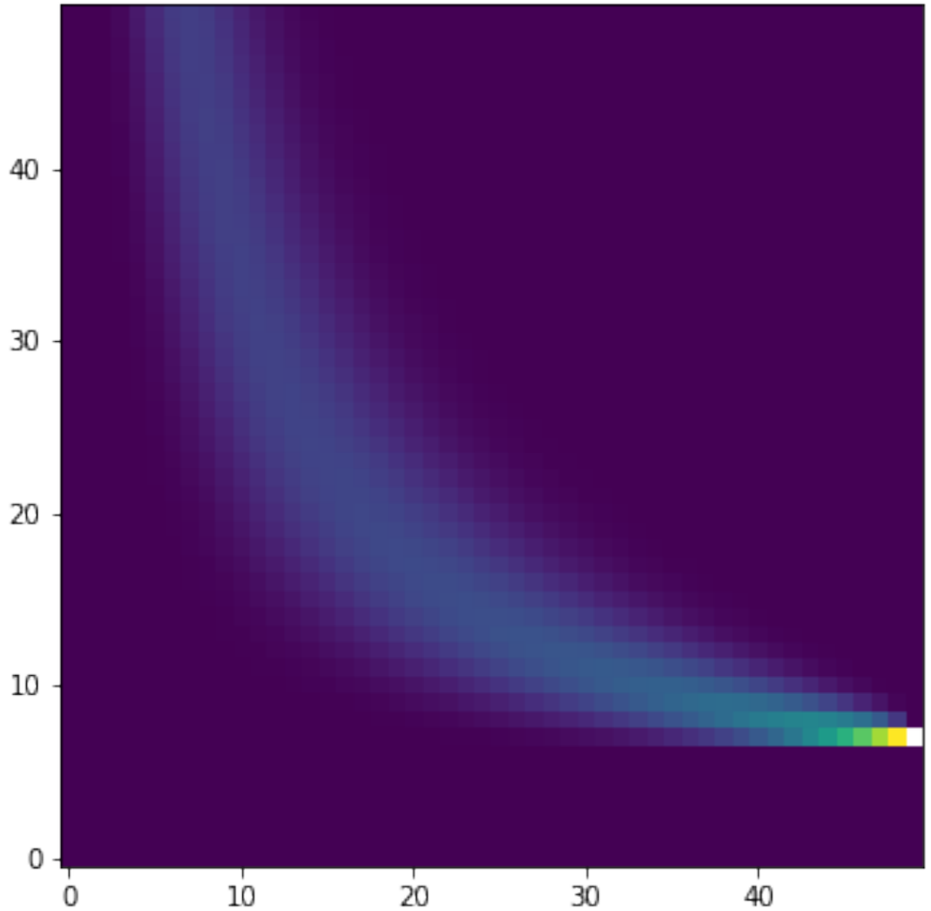
```
logp_m = m.logp
logp_y = y.logp
```

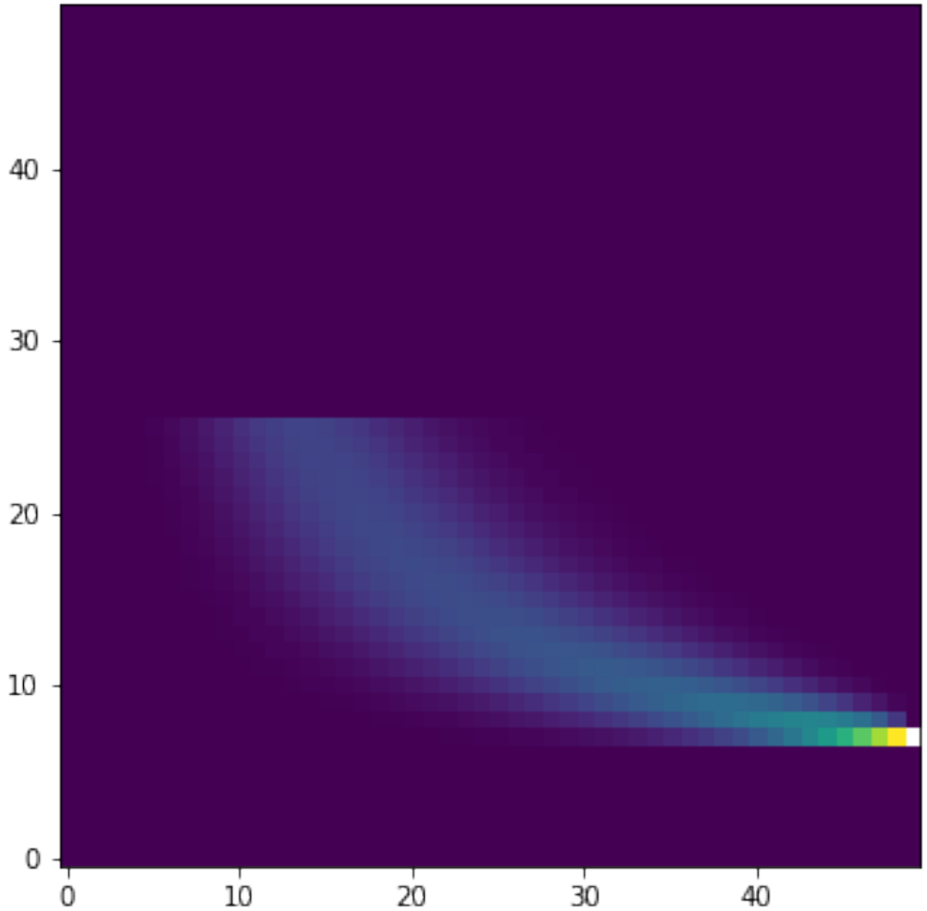
```
logp_m
```

```
<pymc3.model.LoosePointFunc at 0x125656048>
```

Coin flip in PyMC3

```
for i in range(len(lly)):
    point['n'] = nv_.flatten()[i]
    point['p'] = pv_.flatten()[i]
    lly[i] = np.exp(logp_y(point))
    llm[i] = np.exp(logp_m(point))
```





How PyMC3 use Theano

When we define a PyMC3 model, we implicitly build up a Theano function from the space of our parameters to their posterior probability density up to a constant factor.

```
m.logp
```

$$f: \mathbb{R} \times \mathbb{N} \mapsto \mathbb{R}$$

```
m.free_RVs
[n, p]
```

http://docs.pymc.io/theano.html

How PyMC3 use Theano

```
m.logp
```

$$f\colon \mathbb{R} \times \mathbb{N} \mapsto \mathbb{R}$$

```
m.free_RVs
[n, p]
```

```
n.distribution.logp
```

```
   <bound method DiscreteUniform.logp of
   <pymc3.distributions.discrete.DiscreteUniform
   object at 0x125ac6358>>
```

```
n.logpt
```

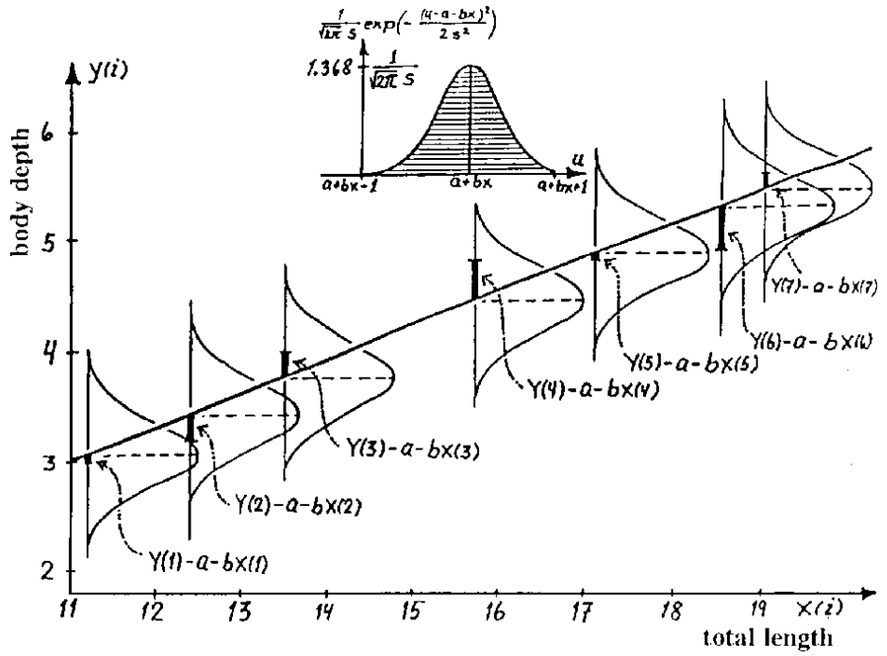
How PyMC3 use Theano

`m.logp`

$$f\colon \mathbb{R} \times \mathbb{N} \mapsto \mathbb{R}$$

```
m.logpt??
@property
def logpt(self):
    """Theano scalar of log-probability of the model"""
    with self:
        factors = [var.logpt for var in self.basic_RVs] +
self.potentials
        logp = tt.sum([tt.sum(factor) for factor in factors])
        if self.name:
            logp.name = '__logp_%s' % self.name
        else:
            logp.name = '__logp'
        return logp
```

General linear model

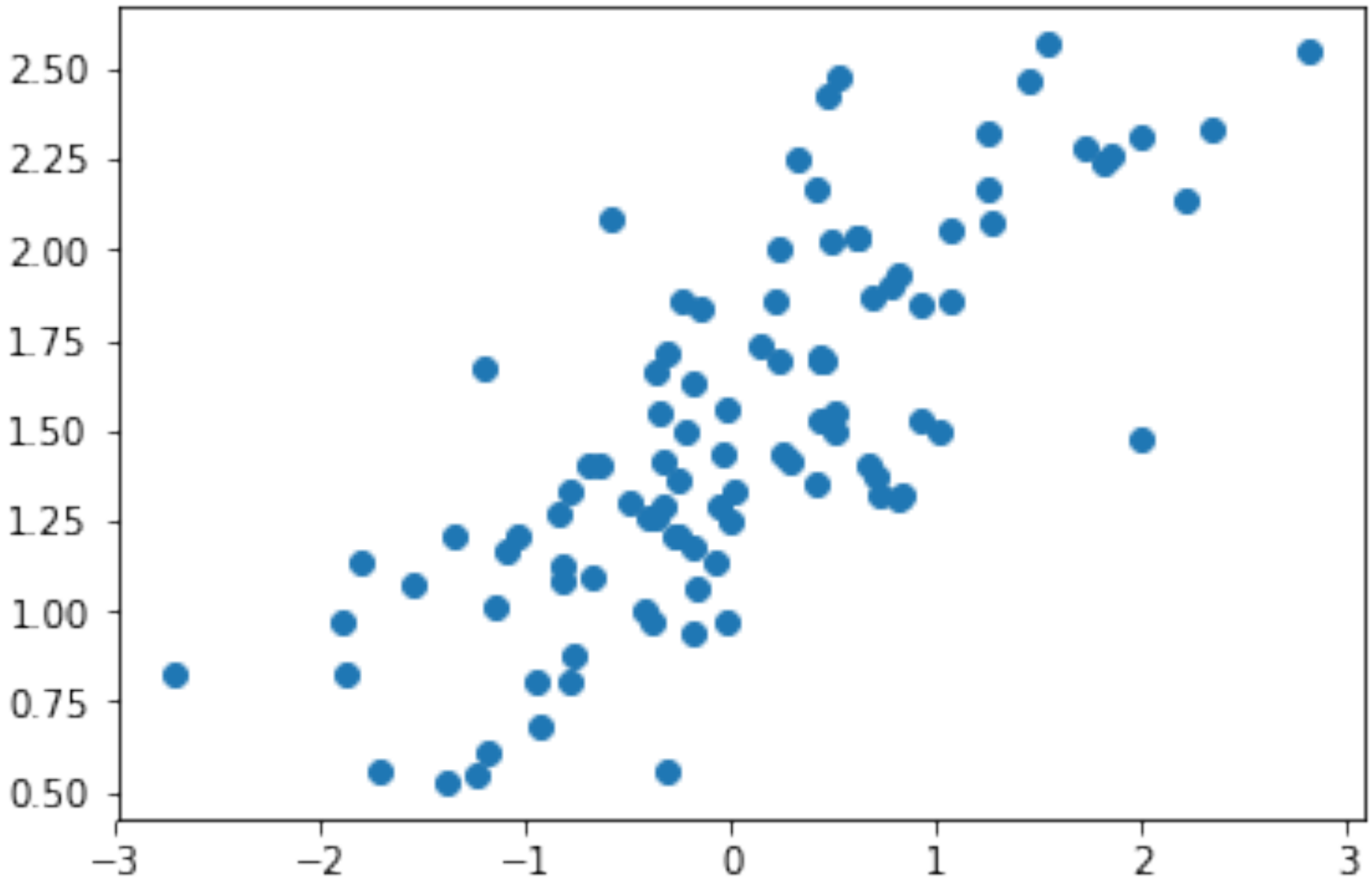


ANOVA

Source	d.f.	SS	MS	F
Treatment	$a-1$	SS_{treat}	$\frac{SS_{treat}}{a-1}$	$\frac{MS_{treat}}{MS_{error(a)}}$
Error (a)	$N-a$	$SS_{error(a)}$	$\frac{SS_{error(a)}}{N-a}$	
Time	$t-1$	SS_{time}	$\frac{SS_{time}}{t-1}$	$\frac{MS_{time}}{MS_{error(b)}}$
Treat x Time	$(a-1)(t-1)$	$SS_{treat\ x\ time}$	$\frac{SS_{treat\ x\ time}}{(a-1)(t-1)}$	$\frac{MS_{treat\ x\ time}}{MS_{error(b)}}$
Error (b)	$(N-a)(t-1)$	$SS_{error(b)}$	$\frac{SS_{error(b)}}{(N-a)(t-1)}$	
Total	$Nt-1$	SS_{total}		

$$\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} 1 & x_{11} & \dots & x_{1k} \\ 1 & x_{21} & \dots & x_{2k} \\ \vdots & \vdots & & \vdots \\ 1 & x_{n1} & \dots & x_{nk} \end{pmatrix} \begin{pmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_k \end{pmatrix} + \begin{pmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \vdots \\ \varepsilon_n \end{pmatrix}.$$

Linear regression in PyMC3



Regression with a twist.ipynb

Linear regression in PyMC3

```
with pm.Model() as m0:
    beta = pm.Normal('beta', 0., 10.)
    a = pm.Normal('a', 0., 10.)
    pm.Normal('y', X*beta+a, 1., observed=y)
```

```
with pm.Model() as m1:
    beta = pm.Flat('beta')
    a = pm.Flat('a')

    pm.Potential('logp_beta',
                 pm.Normal.dist(0., 10).logp(beta))
    pm.Potential('logp_a',
                 pm.Normal.dist(0., 10).logp(a))
    pm.Potential('logp_obs',
                 pm.Normal.dist(X*beta+a, 1.).logp(y))
```

Reparameterization: regression

```
with pm.Model() as m0:
    …
    pm.Normal('y', X*beta+a, sd, observed=y)
```

```
with pm.Model() as m1:
    …
    pm.Normal('eps', 0, sd, observed=y - X*beta - a)
```

Reparameterization: regression

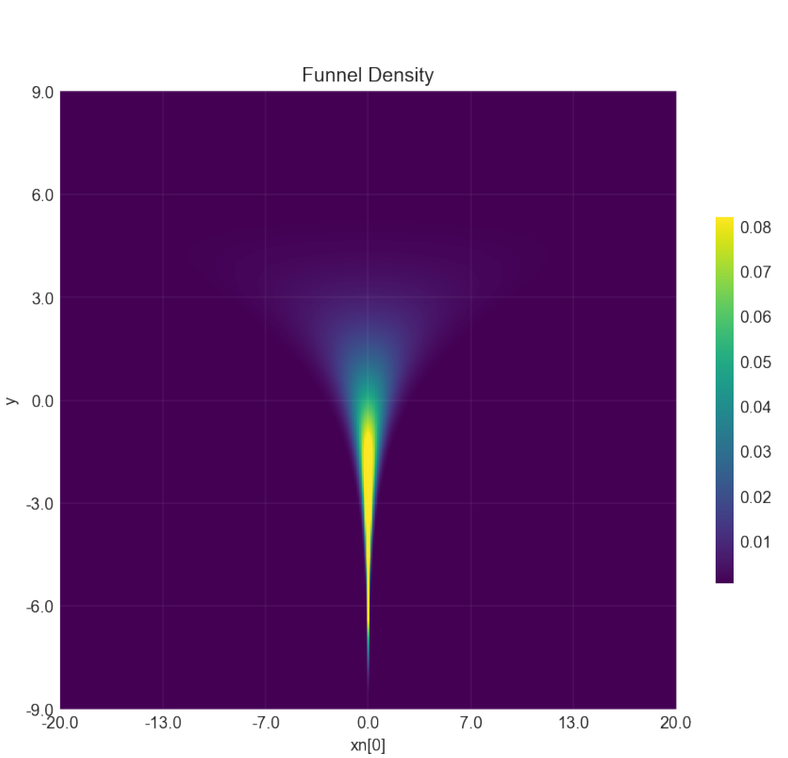
```
with pm.Model() as m0:
    …
    pm.Normal('y', X*beta+a, sd, observed=y)
```

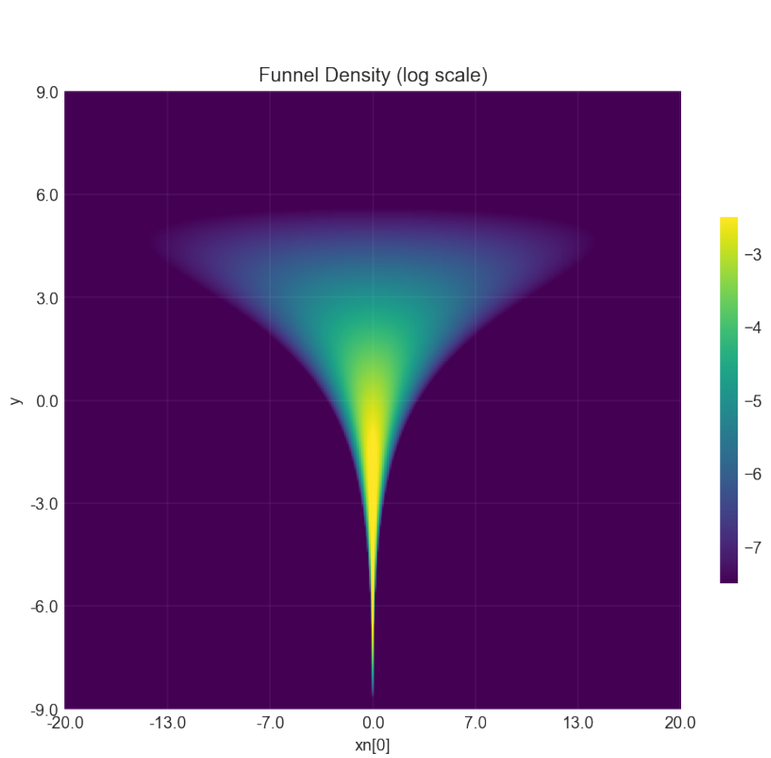
```
with pm.Model() as m1:
    …
    pm.Normal('eps', 0, sd, observed=y - X*beta - a)
```

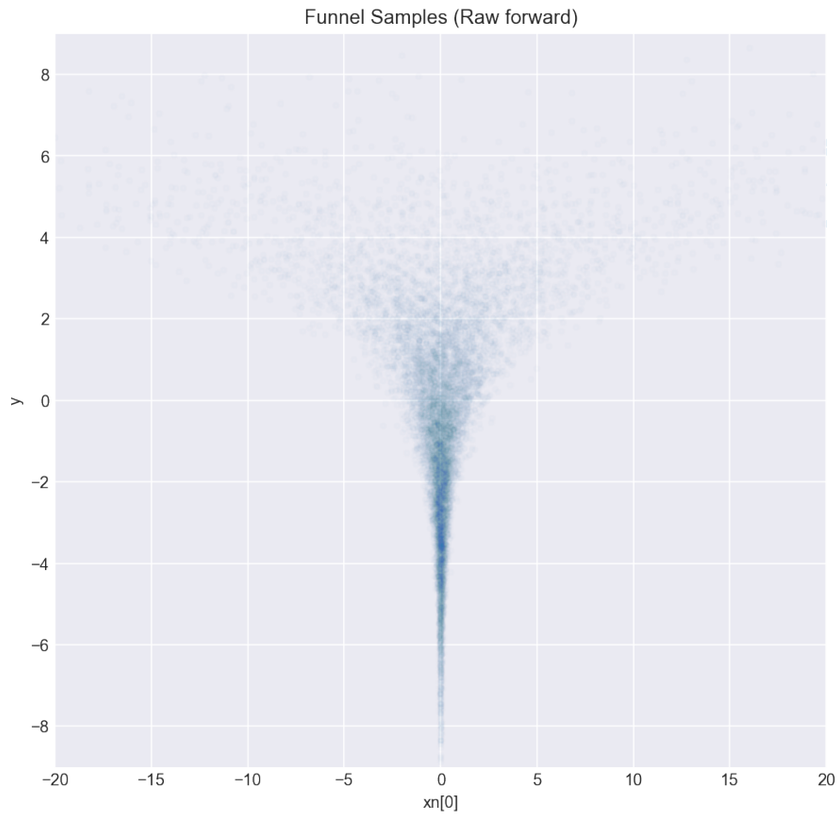
```
with pm.Model() as m2:
    …
    pm.Normal('eps', 0, 1,
              observed=(y - X*beta - a)/sd)
```

Reparameterization: Neal's Funnel

$$p(y, x_n) = \text{Normal}(y \mid 0,3) \times \prod_{n=1}^{9} \text{Normal}(x_n \mid 0, e^{\frac{y}{2}})$$







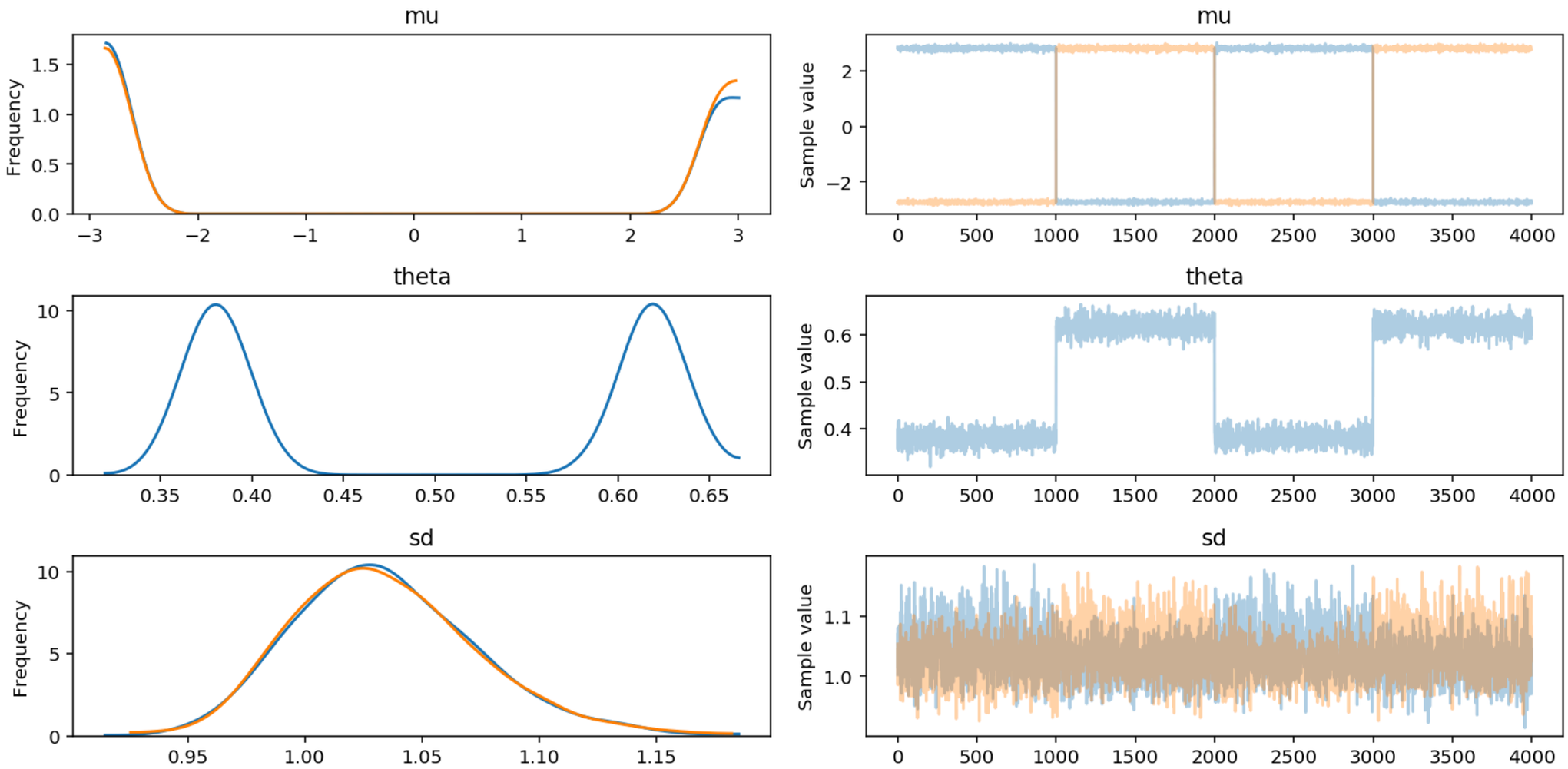
Neals_funnel.ipynb

Mixture model likelihood

Mixture models are difficult to fit…

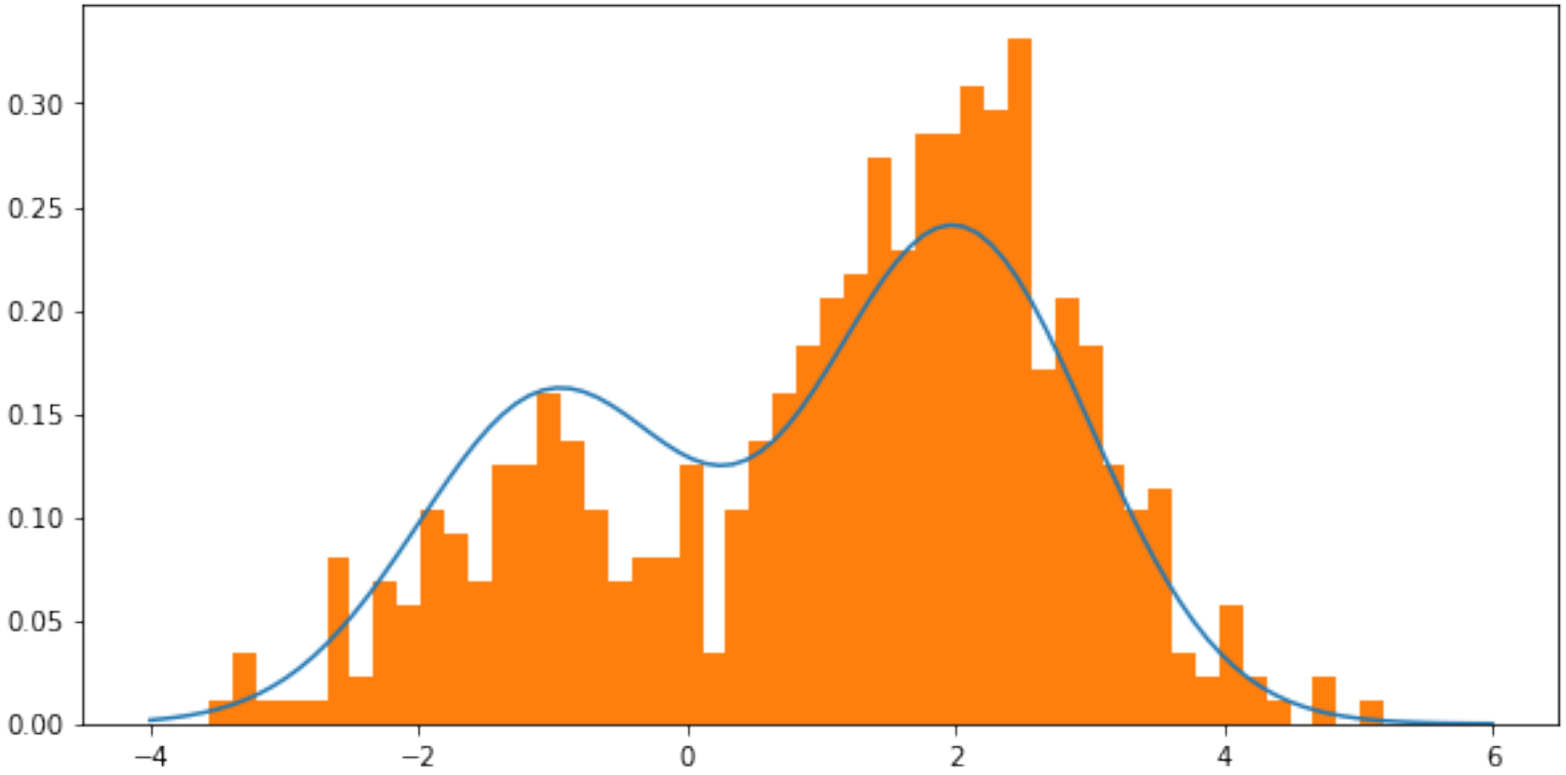
[Identifying Bayesian Mixture Models] by Michael Betancourt

[PyMC3 port]



Mixture model reparameterized

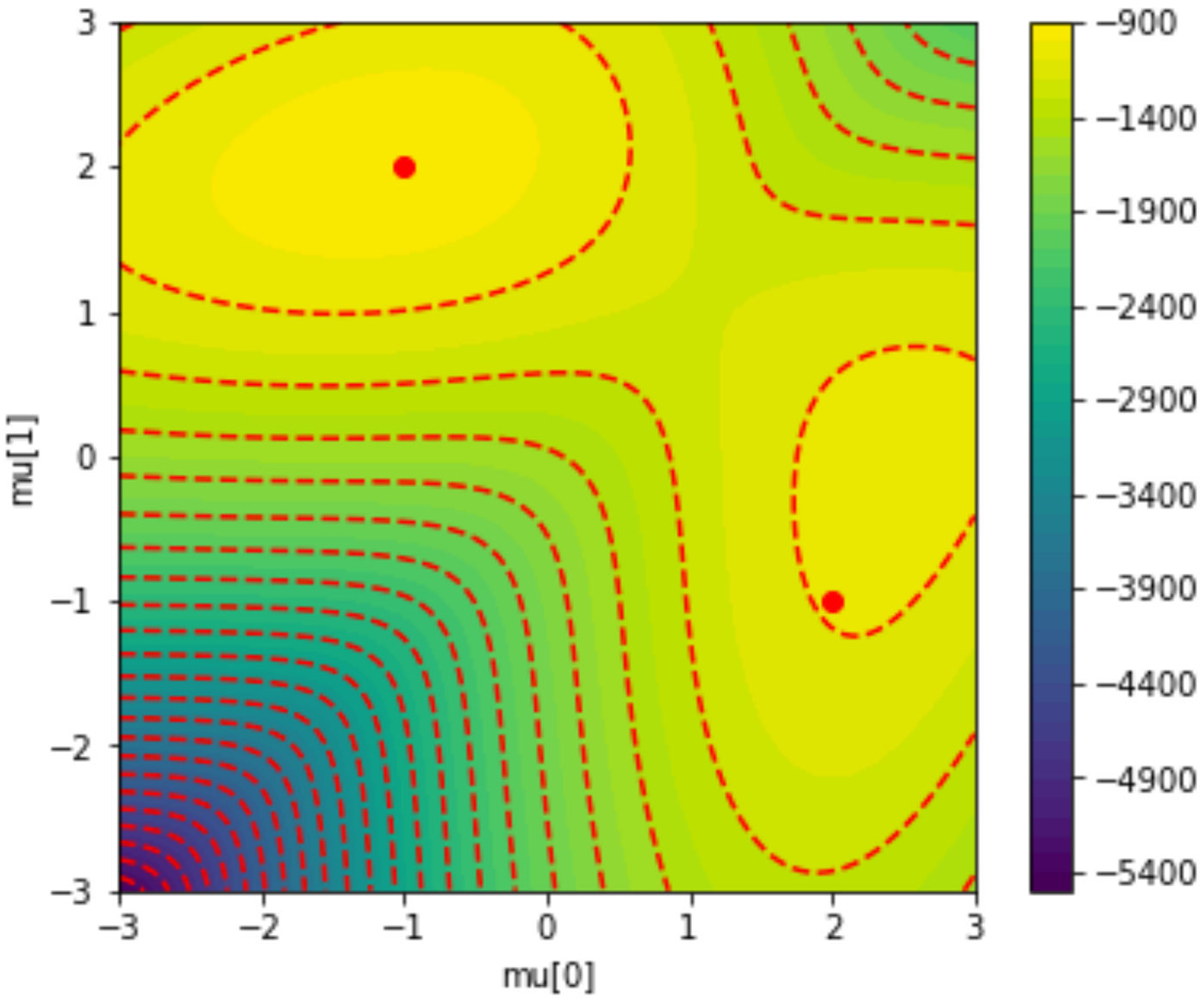
```
w0 = np.array([.3, .7])
mu0 = np.array([-1, 2])
sd0 = 1.
x = pm.NormalMixture.dist(w=w0, mu=mu0,
sd=sd0).random(size=500)
```



Normal_mixture_logp.ipynb

Mixture model reparameterized

```
with pm.Model() as m0:
    mu = pm.Normal('mu', 0., 5., shape=2)
    pm.NormalMixture('y',
                     w=w0,
                     mu=mu,
                     sd=theano.shared(1.),
                     observed=x)
```



Mixture model reparameterized

Non-exchangeable Priors

Enforcing an Ordering

```
import pymc3.distributions.transforms as tr
with pm.Model() as mixture_model:
    …
    mu = pm.Normal('mu', 0, 2, shape=2,
                   transform=tr.ordered,
                   testval=[0.1, 0.2])
    …
```

Mixture model reparameterized

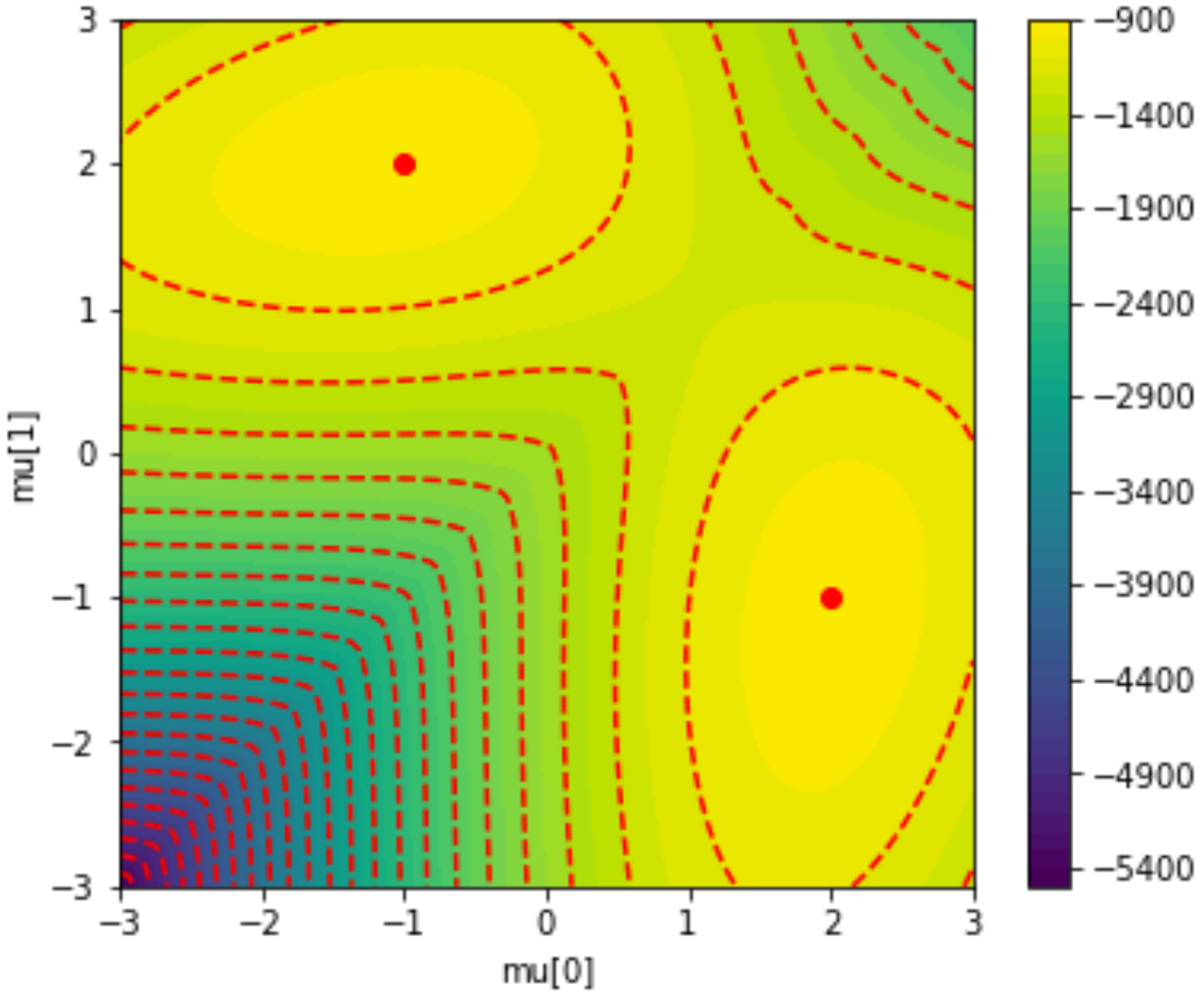
Enforcing an Ordering

```
with pm.Model() as mixture_model:
    …
    mu = pm.Normal('mu', 0, 2, shape=2,
                   transform=tr.ordered,
                   testval=[0.1, 0.2])
    …
```

```
with pm.Model() as mixture_model_:
    …
    mu_ = pm.Normal('mu_', 0, 2, shape=2)
    mu = tt.sort(mu_)
    …
```

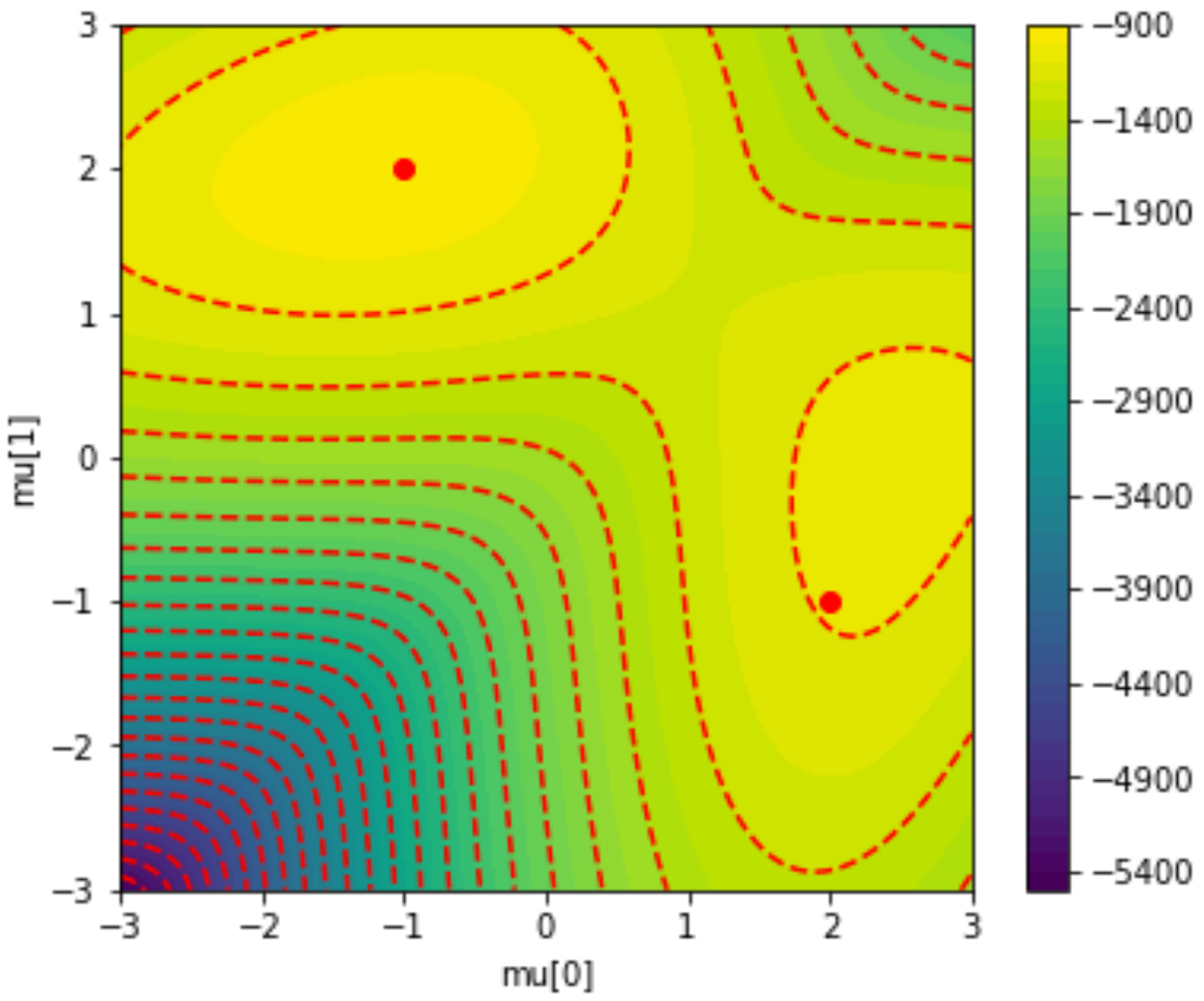
Mixture model reparameterized

```
with pm.Model() as m1:
    mu = pm.Normal('mu', 0., 5., shape=2)
    mu_ = tt.sort(mu)
    pm.NormalMixture('y',
                     w=w0,
                     mu=mu_,
                     sd=theano.shared(1.),
                     observed=x)
```

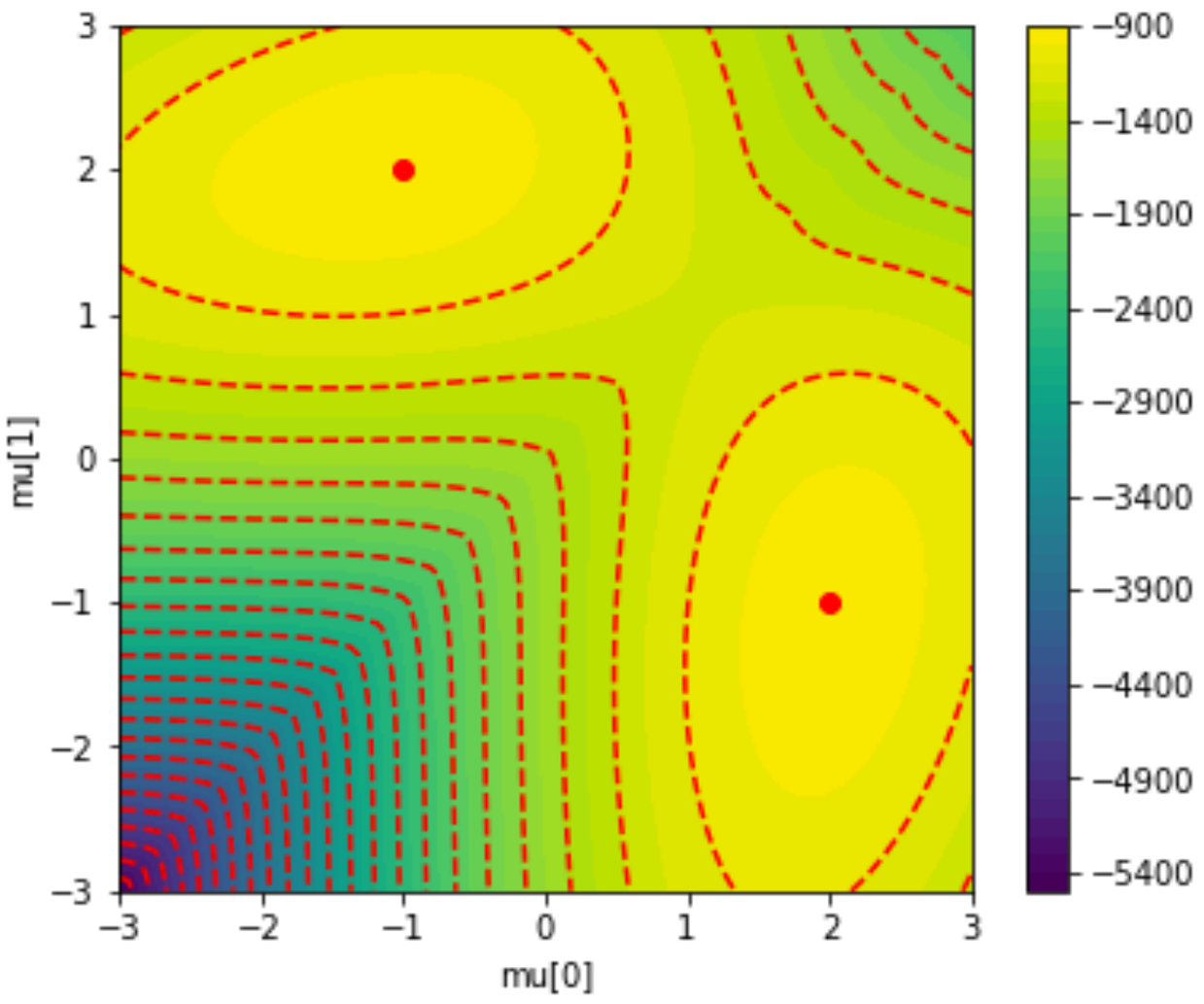


Mixture model reparameterized

Origin model



with tt.sort



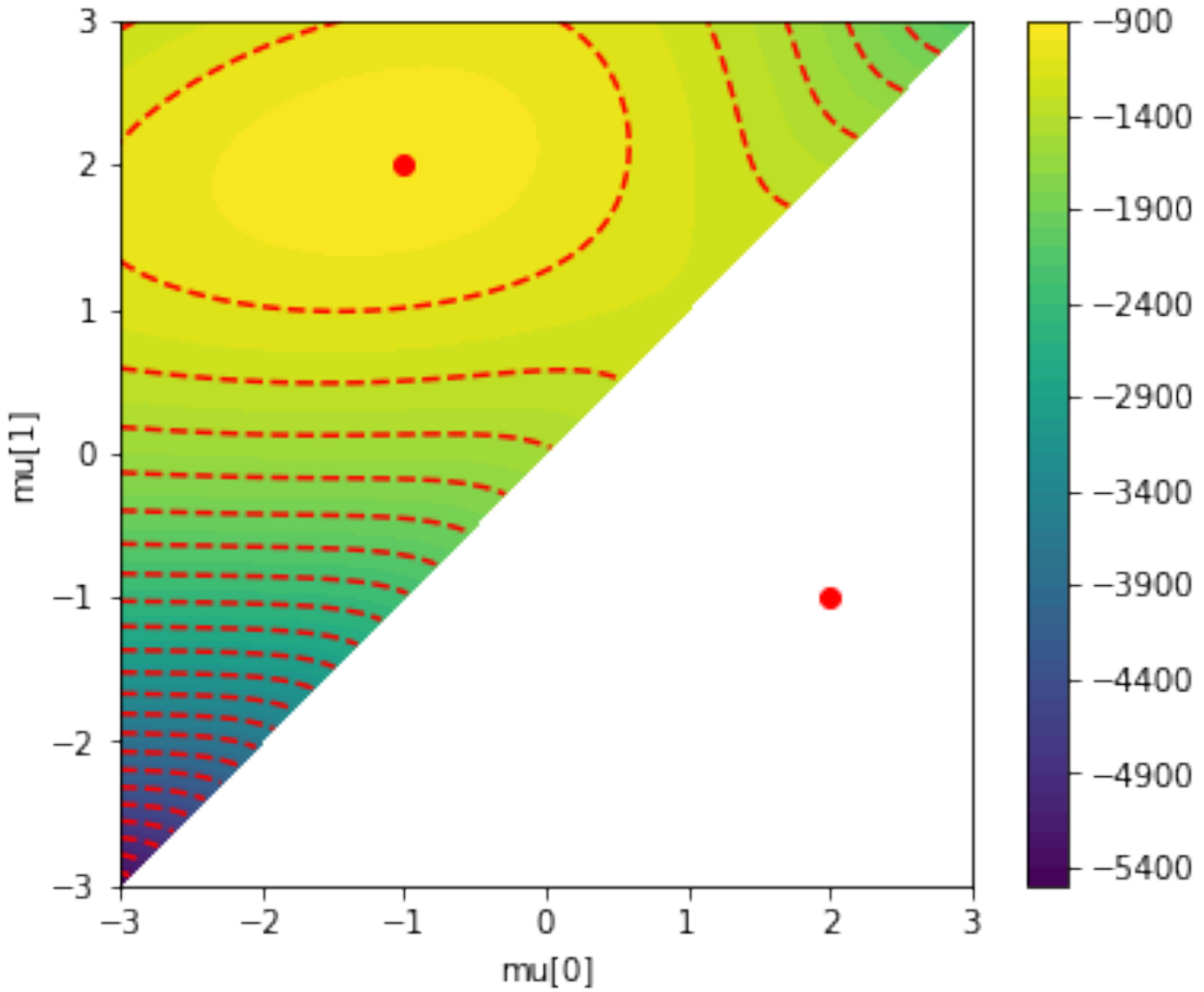
Mixture model reparameterized

```
with pm.Model() as m2:
    mu = pm.Normal('mu', 0., 5., shape=2)
    pm.NormalMixture('y',
                     w=w0,
                     mu=mu,
                     sd=theano.shared(1.),
                     observed=x)
    pm.Potential('order', tt.switch(mu[1]<mu[0], -np.inf, 0))
```

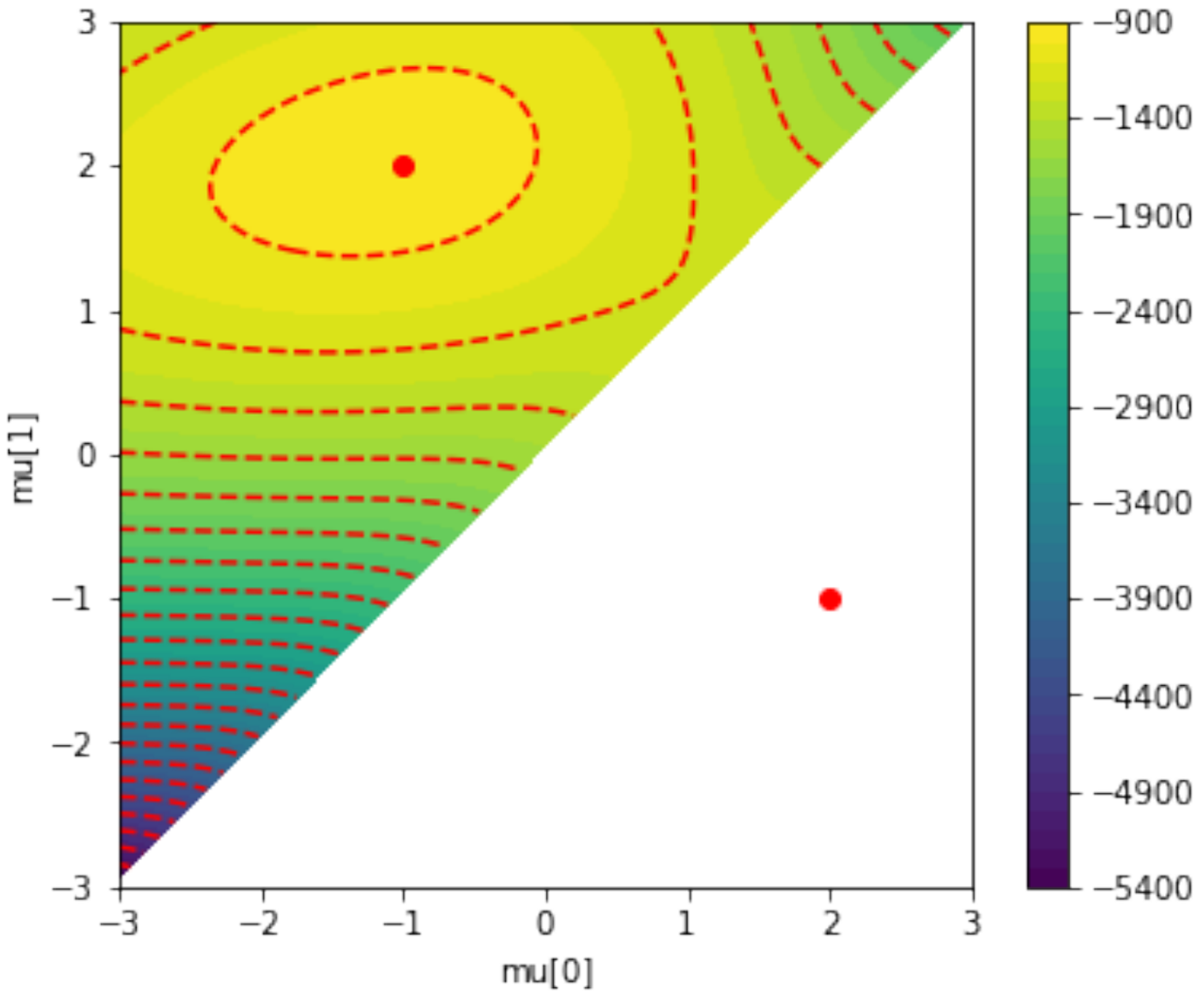
```
with pm.Model() as m3:
    mu = pm.Normal('mu', 0., 5.,
                   shape=2,
                   transform=tr.ordered,
                   testval=np.array([0., 1.]))
    pm.NormalMixture('y',
                     w=w0,
                     mu=mu,
                     sd=theano.shared(1.),
                     observed=x)
```

Mixture model reparameterized

with potential



with ordered transformation



Parameterization

There are many ways to write down the same model, but reparameterization always change the model logp:

- Make sure it is correct
- Slow model usually indicates problem
  - *The folk theorem of statistical computing: When you have computational problems, often there's a problem with your model*
- A good parameterisation should be easy to understand

Recap:

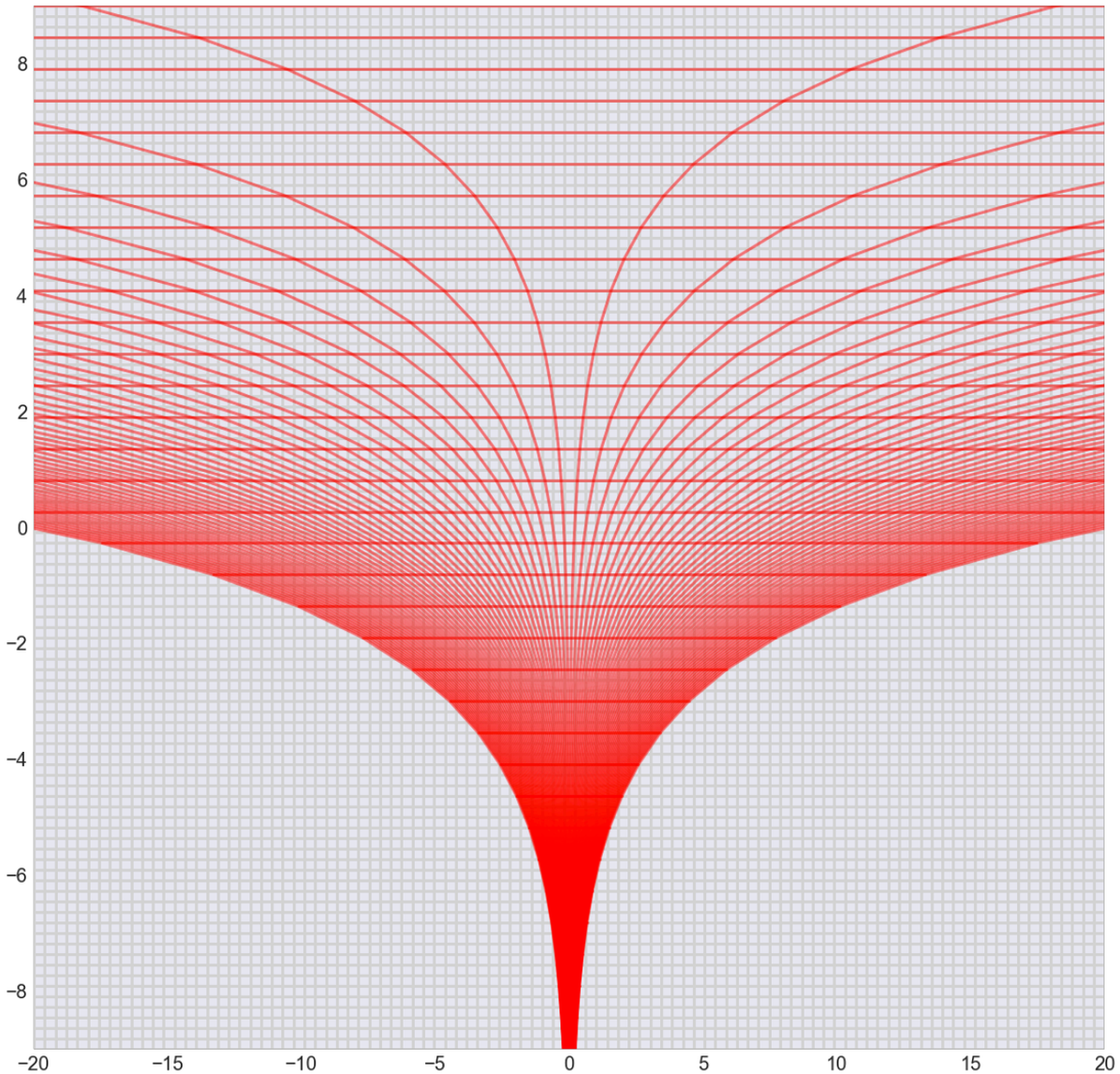
The distinction between probability and likelihood is probably unnecessary.

We “glue” random variables together in a conditional network to create a mapping that is the (log-) likelihood function.

- The dimension of the parameter space is the same as the number of unknowns.

When we are building model, we are setting up the space (concept of coordinate system)

Different coordinate systems

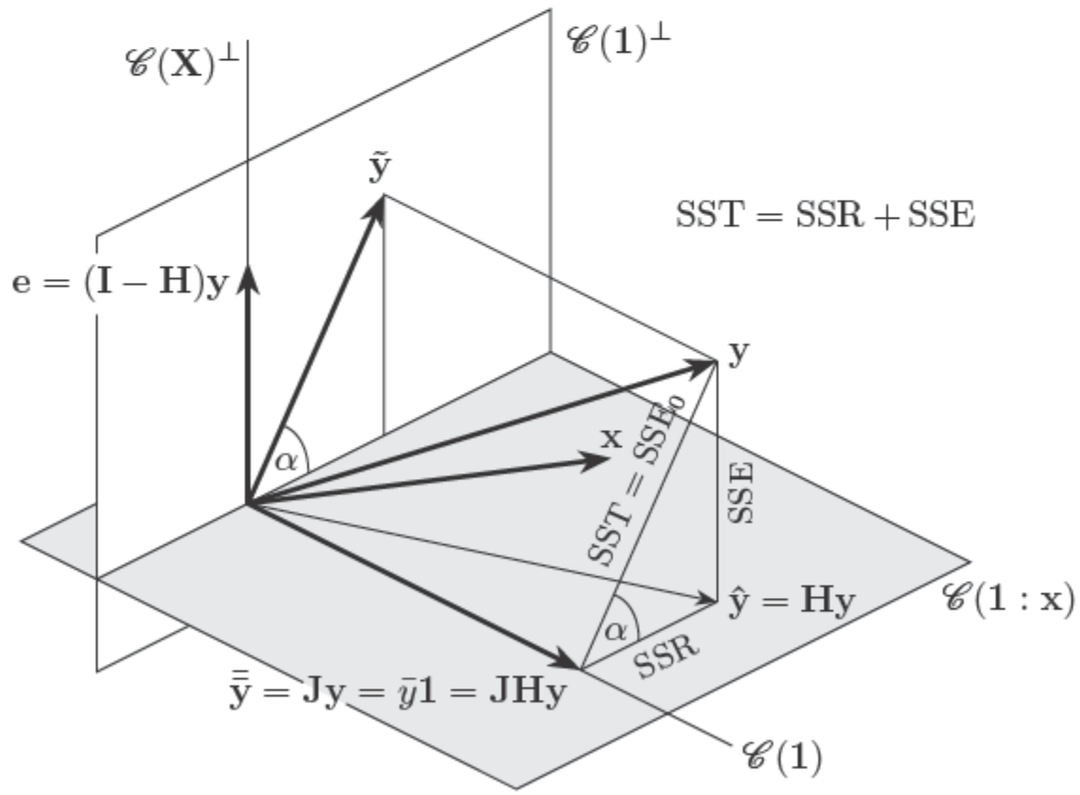


Neals_funnel.ipynb

L1 and L2 regularization

Lasso Regression (Least Absolute Shrinkage and Selection Operator)

$$\sum_{i=1}^{n}(y_i - \sum_{j=1}^{p} x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} | \beta_j |$$



Ridge regression

$$\sum_{i=1}^{n}(y_i - \sum_{j=1}^{p} x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} \beta_j^2$$

Recap:

The distinction between probability and likelihood is probably unnecessary.

We “glue” random variables together in a conditional network to create a mapping that is the (log-) likelihood function.

- The dimension of the parameter space is the same as the number of unknowns.

When we are building model, we are setting up the space (concept of coordinate system)

Plotting a slice of the likelihood “space” is very useful for debugging.

Use logp function to check model

Inferencing trigonometric time series model

Questions

narendramukherjee 2 6d

Following the conversation in How to model sinusoids in white gaussian noise, I have been trying to fit a sinusoidal model as well. I have tried a lot of different parametrizations with NUTS, some of which are as follows:

1. Try to fit $y(t) = Asin(\omega t) + Bcos(\omega t) + noise$ where A and B are drawn as Normal random variables.
2. Draw A and B as Normal random variables, and fit $y(t) = Csin(\omega t + \phi)$ where $C = \sqrt{A^2 + B^2}$ and $\phi = tan^{-1}\frac{B}{A}$.
   I tried some others which were worse than these two.

The conversation has revolved around the phase parameter mostly in this discussion, but I think the issue lies more with NUTS having problems in a nonlinear model. VERY suprisingly (to me at least!), Metropolis works fabulously in this model (I tried the parametrization number 1 above). There has been at least one previous report of NUTS being miserable in a nonlinear model when Metropolis worked well, and it seems that issue wasn't resolved back then:

Timeserie_model.ipynb

Some thoughts:

- Likelihood is an important concept in Bayesian Computation
  - Not always available or expensive to evaluate (ABC)
- Think in terms of the parameter space
  - Model fitting - what it is really?
- Designing model specific inference
  - Laplace approximation to take expectation of part of the variable
  - Expectation maximisation
- Connection to predictive distribution (elpd)

Thanks!