Sampling Solutions

September 11, 2014

Exercise 1

1. Consider the power-law distribution $p(S|\alpha) \propto S^{-\alpha}$ from yesterday's problems.
2. Generate samples, S_i from this distribution (for some fixed value of α), using rejection sampling or otherwise (but if your computer has a mechanism for directly generating power-law samples, please don't use that!). [In this case, do we need to know the normalization constant?]
   1. In this case, do we need to know the normalization constant?
   2. Do you need to make any additional assumptions?
3. Determine the mean and variance of the samples and check against an analytical calculation.
4. Plot the distribution and make a histogram of the samples.

```
In [1]: from __future__ import division
        import math
        import numpy as np
        import scipy as sp
        import matplotlib.pyplot as plt
        %matplotlib inline
```

In this case, we'll also need to pick a maximum flux $S_{\max}$ in order to choose a well-defined uniform distribution as input to our rejection sampler.

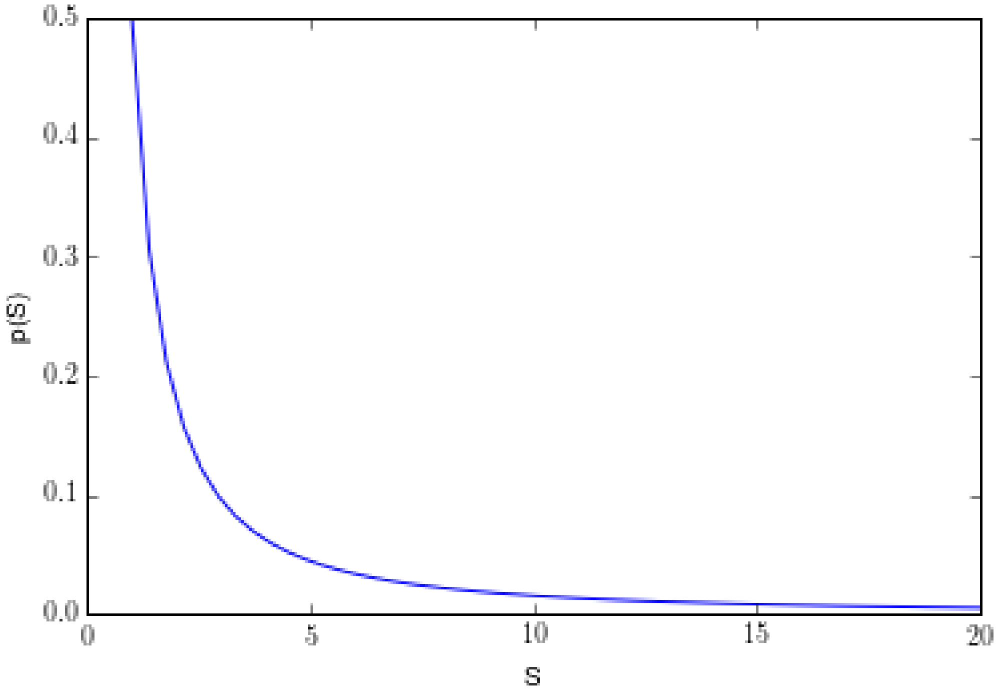
Notes:

- If we really want to draw samples over the whole region out to arbitrarily large values of S, we should use a different technique – see below.
- Power-law distributions can have a lot of probability in the tails. This means that the variance will diverge if $\alpha < 2$, and that an $S_{\max}$ cutoff may miss important samples.

0.0.1 Solution:

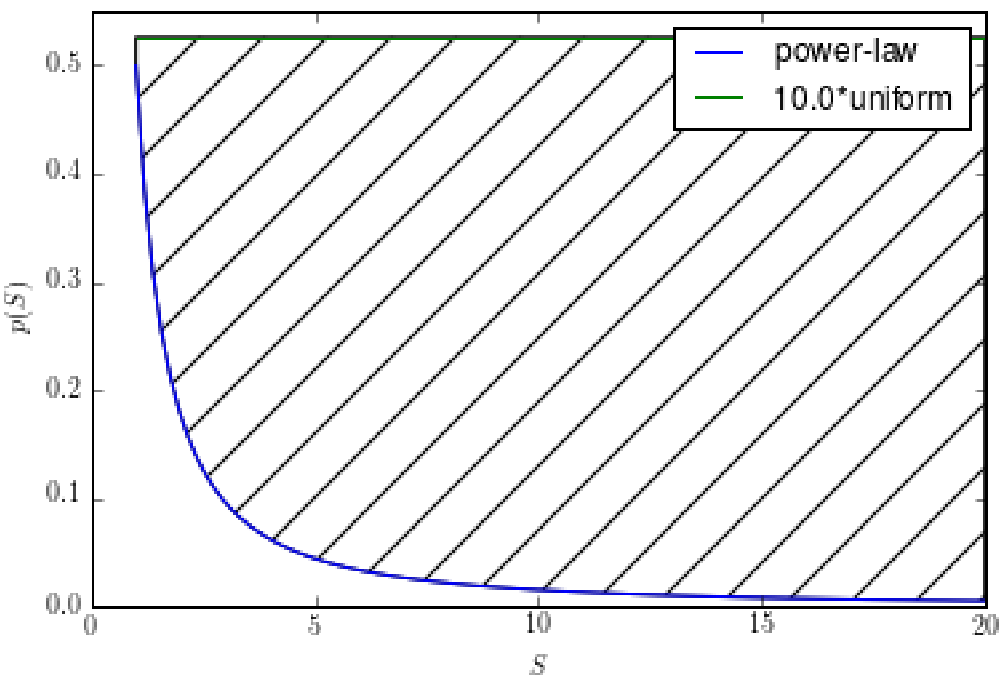
```
In [2]: def powerlaw(s, alpha=1.5, s0=1):
            return (1/s0)*(alpha-1)*(s/s0)**-alpha

        ### we know that it diverges a maximum as S->0; need to pick a minimum s0.
        ### Units don't matter, so pick S0=1
        s0 = 1

        sarr = np.linspace(s0, 20*s0, 50)
        plt.plot(sarr, powerlaw(sarr))
        plt.xlabel("S")
        plt.ylabel("p(S)")
```

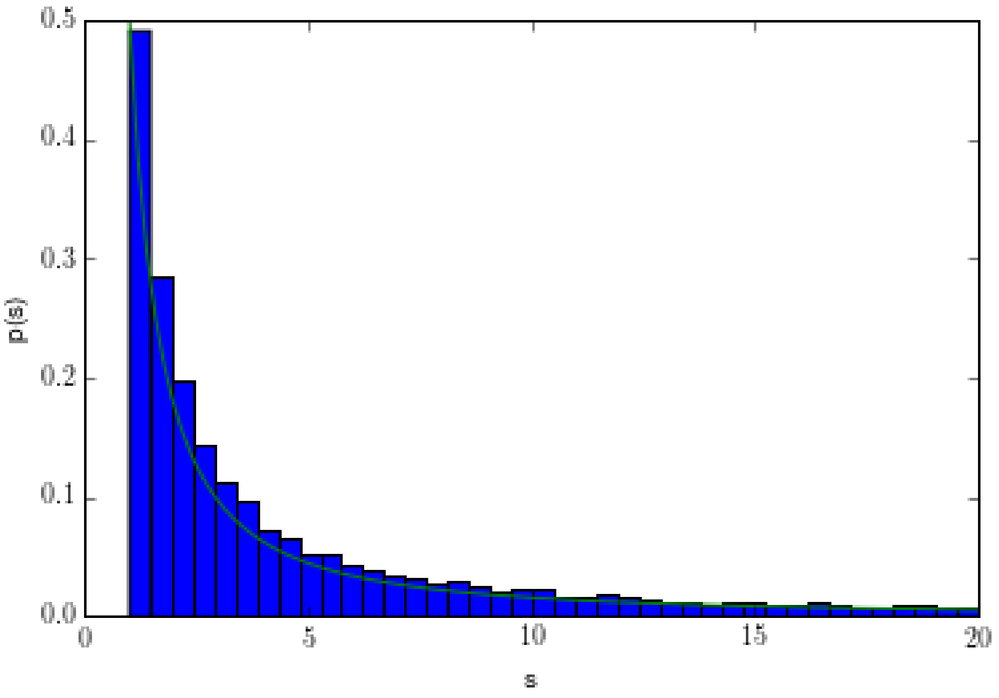
```
Out[2]: <matplotlib.text.Text at 0x107f06550>
```



```
In [3]: def u_1(x, xmin=1, xmax=20):
            return 0 if (x<xmin or x>xmax) else 1.0/(xmax-xmin)
        u = np.vectorize(u_1, otypes=[np.float], excluded={'xmin', 'xmax'})

        smax = 20
        pmax = powerlaw(s0)
        scale = pmax*smax

        sarr = np.linspace(s0,smax,100)
        plt.plot(sarr, powerlaw(sarr), label="power-law")
        plt.plot(sarr, scale*u(sarr,xmin=s0,xmax=smax), label="%3.1f*uniform"%scale)
        plt.fill_between(sarr, scale*u(sarr,xmin=s0,xmax=smax),powerlaw(sarr),
                         hatch="/", facecolor="w")
        plt.ylim(0,pmax*1.1)
        plt.xlabel("$S$")
        plt.ylabel("$p(S)$")
        plt.legend()
        plt.figure()

        def rejection_sample(p, xlim=(-1,1), pmax=0.9):
            """
                use rejection sampling to get samples from p(x), using uniform samples
            """

            delx = xlim[1]-xlim[0]                 #### range of x
            scale = delx*pmax
            keep = True
            while keep:   ### loop until you're meant to keep a sample
                #### generate a sample from u: np.random.random generates from U(0,1)
                u_sample = delx*np.random.random()+xlim[0]

                fraction_to_keep = p(u_sample)/(scale*u(u_sample))
                keep = np.random.random()>fraction_to_keep

            return u_sample


        ssamples = np.array([rejection_sample(powerlaw, xlim=(s0,smax),pmax=pmax)
                             for _ in range(10000)])
        plt.hist(ssamples, normed=True, bins=40)
        plt.plot(sarr, powerlaw(sarr))
        plt.ylabel("p(s)")
        plt.xlabel("s")
```

```
Out[3]: <matplotlib.text.Text at 0x1082018d0>
```





As noted above, this is probably not the best solution for this problem, as it requires specifiying a maximum flux, $S_{\max} < \inf$.

In this case, we can sample from the whole range by a change of variables. Consider the random variable $y = y(S)$ which we wish to have a uniform distribution $u(y)$ over $(0, 1)$, so $u(y) = 1$ over that range, $u(y) = 0$ otherwise:

$$p(S)\ dS = u(y)\ dy = dy$$

We can integrate this from $S = S_0$ corresponding to $y = 0$:

$$\int_{S_0}^{S} (\alpha - 1) \left(\frac{S'}{S_0}\right)^{-\alpha}\ \frac{dS'}{S_0} = \int_0^y\ dy'$$

or

$$1 - \left(\frac{S}{S_0}\right)^{1-\alpha} = y$$

so we can solve for $S = S(y)$:

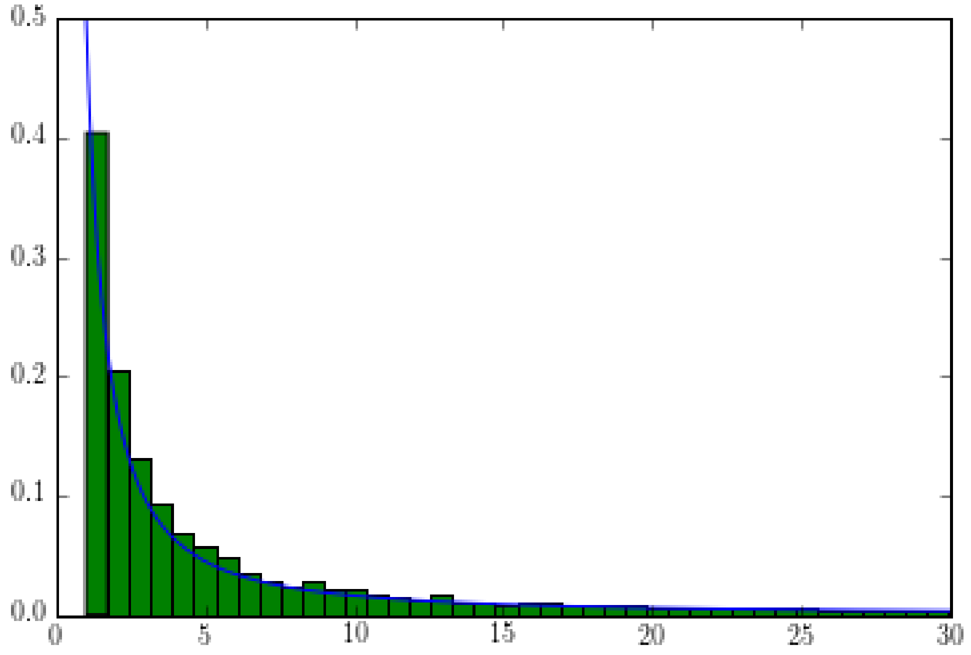
$$S = S_0 \left(1 - y\right)^{\frac{1}{1-\alpha}}$$

But this is just what we need: we can easily sample y from $u(y)$ and then just calculate $S = S(y)$ from this formula. (As an aside, there's a slight simplification — if y is from $u(0, 1)$, then so is $1 - y$, so we can just use $S = S_0 y^{\frac{1}{1-\alpha}}$.

Note that even in this case we'll need to pick and $S_{\max}$ for plotting purposes.

```
In [4]: alpha = 1.5
        S0 = 1
        expt = 1.0/(1.0-alpha)
        Smax=30
        nsamp = 10000
        ysamples = np.random.random(nsamp)
        ssamples = S0*ysamples**expt
        print "Max and min: %f, %f" % (ssamples.max(), ssamples.min())
        if Smax<ssamples.max():
            ssamples=ssamples[ssamples<Smax]
            print "new max: %f" % ssamples.max()
            print "throwing out fraction %3.3g of samples" % (1 - len(ssamples)/nsamp)
        else:
            Smax = ssamples.max()
        print "Mean and variance: %f, %f" % ( np.mean(ssamples), np.var(ssamples))
        sarr = np.linspace(1,Smax,1000)
        plt.plot(sarr, powerlaw(sarr, alpha=alpha))
        plt.hist(ssamples, normed=True, bins=40);
```

```
Max and min: 71059121.847504, 1.000337
new max: 29.926279
throwing out fraction 0.183 of samples
Mean and variance: 5.489023, 36.748019
```



We can also change variables in another way Consider the quantity $y = \ln(S/S_0)$, or $S = S_0 e^y$. This quanitity has the distribution

$$p(y|\alpha)\,dy = p(S|\alpha)\,dS = p(S = S_0 e^y|\alpha)\frac{dS}{dy}dy = S_0^{1-\alpha} e^{(1-\alpha)y} \propto e^{-(\alpha-1)y}$$

This is just an exponential distribution, which your package may be able to sample from directly.

Exercise 2

2. Consider the bivariate distribution that is uniform between -1 and 1 for the quantity $x - y$ and a (univariate) normal with mean 0 and variance 1 for the quantity $x + y$.
    1. How can we draw samples (pairs of random numbers) from this distribution using univariate uniform and normal random number generators?

2. Estimate the mean and covariance from the samples.
3. Plot the results in 2d, as well as the 1d marginals, with an appropriate color scheme.
4. Overlay the contours of the approximate gaussian with the estimated mean and covariance.

0.0.2 Solution

We can define $v = x - y$ with a $U(-1, 1)$ distribution and $w = x + y$ with an $N(0, 1)$ distribution, and draw samples for each of these:

```
In [5]: nsamp = 10000

        def p_gaussian(x, mu=0, sigma=1):
            """ a univariate gaussian distribution """
            return (2*math.pi*sigma)**(-0.5)*np.exp(-0.5*(x-mu)**2/sigma )

        def p_multigaussian(x, mu, covar):
            """ multivariate gaussian PDF """

            detC = np.linalg.det(2*math.pi*covar)
            delx = x-mu
            Cinv_mu = np.linalg.solve(covar, delx)
            chi2 = np.dot(delx, Cinv_mu)

            return detC**(-0.5)*np.exp(-chi2/2.0)


        vsamples=np.random.uniform(-1,1,size=nsamp)
        wsamples=np.random.normal(loc=0, scale=1, size=nsamp)
        probabilities = p_gaussian(wsamples, mu=0, sigma=1)*0.5    #### 0.5 is the value of the U(0,1) d
```

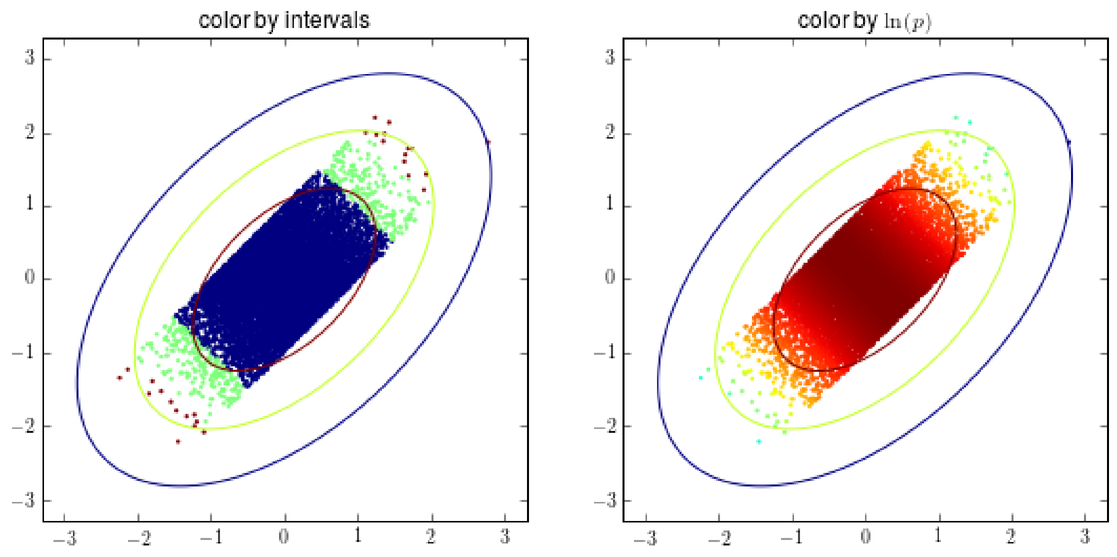
But these are simply related to x and y:

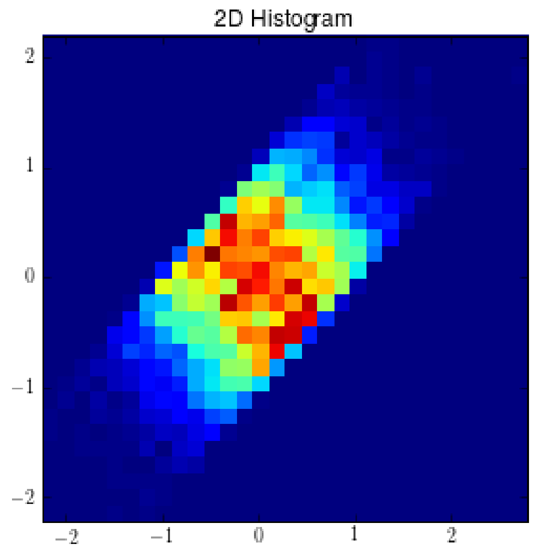
$$x = (v + w)/2 \qquad \text{and} \qquad y = (w - v)/2$$

so we can just convert. (Note that we don't care about the Jacobian of the transformation since it is just a constant, irrelevant for us here.)

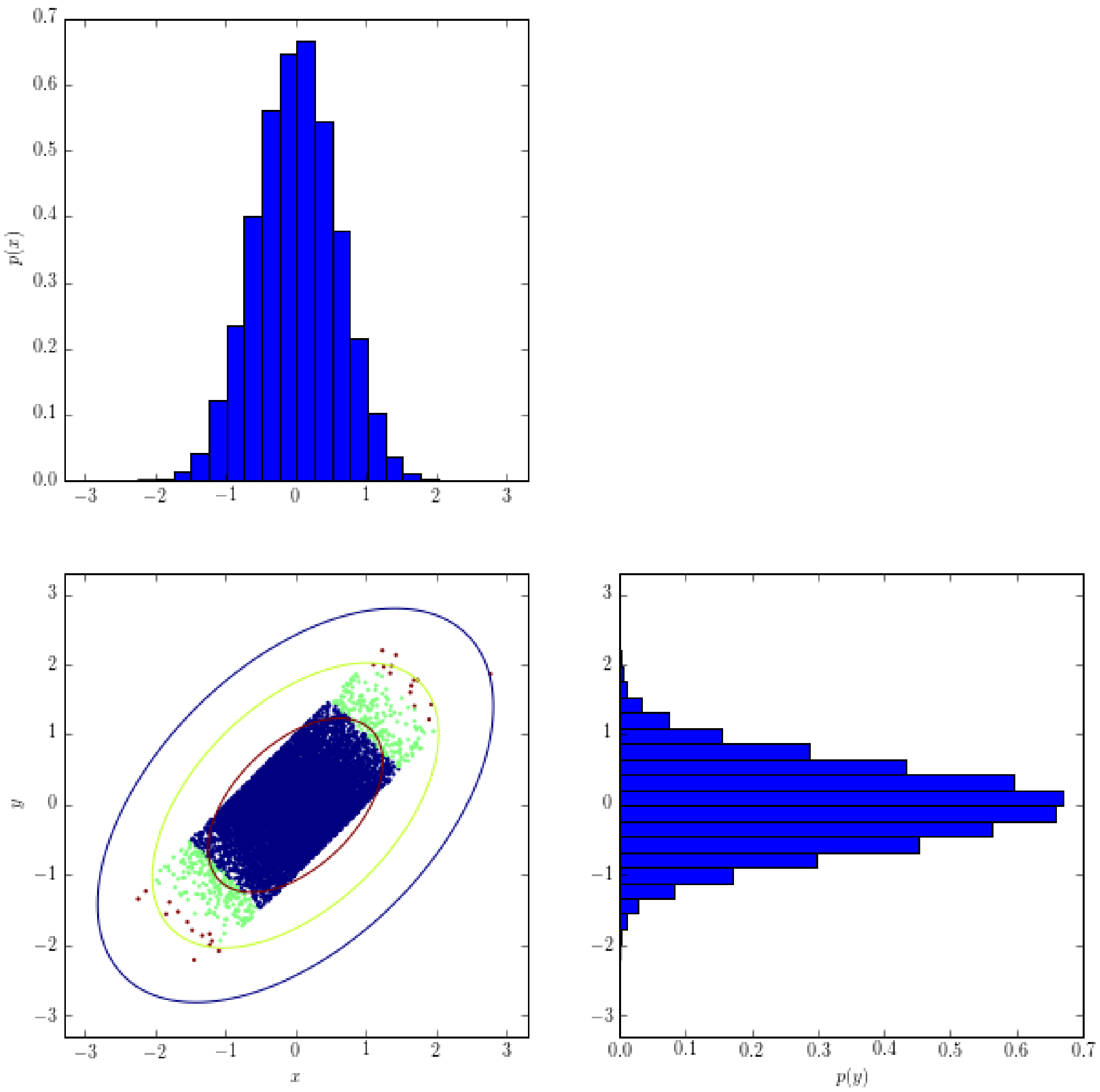
```
In [6]: xsamples = 0.5*(vsamples+wsamples)
        ysamples = 0.5*(wsamples-vsamples)

        nsamples = np.vstack((xsamples, ysamples))

        sorted_probabilities = np.sort(probabilities)

        n = len(probabilities)
        alphas = [0.683, 0.954, 0.9973]
        q_levels = [sorted_probabilities[np.round((1-a)*n)] for a in alphas]
        colors = np.zeros_like(probabilities)
        for col, lev in enumerate(q_levels):
            colors[probabilities<lev] = col


        samples_mean = np.average(nsamples, axis=1)
        samples_covar = np.cov(nsamples)
        print "mean: ", samples_mean
        print "covar: ", samples_covar

        xarr = np.linspace(-3,3,100)
        yarr = np.linspace(-3,3,100)
        xi,yi = np.meshgrid(xarr, yarr)
        zshape = xi.shape
        zarr = np.array([np.log(p_multigaussian(xy, samples_mean, samples_covar))
                          for xy in zip(xi.flat, yi.flat)])
        zarr = zarr-max(zarr)
        zarr.shape = zshape

        dchi2 = np.array([2.30, 6.17, 11.8])  ### corresponding to *1D* 1,2,3sigma


        with plt.rc_context(rc={'figure.figsize': (10.0, 10.0)}):
            plt.figure()
            plt.axes().set_aspect('equal');

            plt.subplot(2,2,3)
            plt.title("2D Histogram")
            plt.hist2d(nsamples[0,:], nsamples[1,:], bins=30)


            plt.subplot(2,2,2)
            plt.title("color by $\ln(p)$")
            plt.scatter(nsamples[0,:], nsamples[1,:], c=np.log(probabilities),
                        marker='.', lw=0 )
            plt.contour(xi, yi, zarr, levels=-0.5*(dchi2)*2)

            plt.subplot(2,2,1)
            plt.title("color by intervals")
            plt.scatter(nsamples[0,:], nsamples[1,:], c=colors, marker='.', lw=0 )
            plt.contour(xi, yi, zarr, levels=-0.5*(dchi2)*2)
```

```
mean:  [-0.00085376 -0.01079721]
covar:  [[ 0.3373383   0.16889489]
 [ 0.16889489  0.33475307]]
```





```
In [7]: with plt.rc_context(rc={'figure.figsize': (10.0, 10.0)}):
            plt.figure()
            plt.axes().set_aspect('equal');

            plt.subplot(2,2,3)
            plt.scatter(nsamples[0,:], nsamples[1,:], c=colors, marker='.', lw=0 )
            plt.contour(xi, yi, zarr, levels=-0.5*(dchi2)*2)
            plt.xlabel("$x$")
            plt.ylabel("$y$")

            ### save the 2-d x and y limits for use in the histograms
            xlim=plt.xlim()
            ylim=plt.ylim()

            plt.subplot(2,2,4)
            plt.hist(nsamples[1,:],normed=True, bins=20, orientation='horizontal')
            plt.xlabel("$p(y)$")
            plt.ylim(ylim)

            plt.subplot(2,2,1)
            plt.hist(nsamples[0,:],normed=True, bins=20)
            plt.ylabel("$p(x)$")
            plt.xlim(xlim)
```



Note that in this case, there is a slightly simpler way to do this, since the (v, w) plane is just a 45 degree rotation from (x, y), but the somewhat more general problem where (v, w) are related to (x, y) by an arbitrary linear transformation couldn't be solved this way.