Sampling from a distribution
ICIC Astrostats Workshop, September 2014

Andrew Jaffe

September 11, 2014

```
In [1]: # You should run this line the first time to get the LaTeX output!
        # from IPython.external import mathjax; mathjax.install_mathjax()
```

Some setup

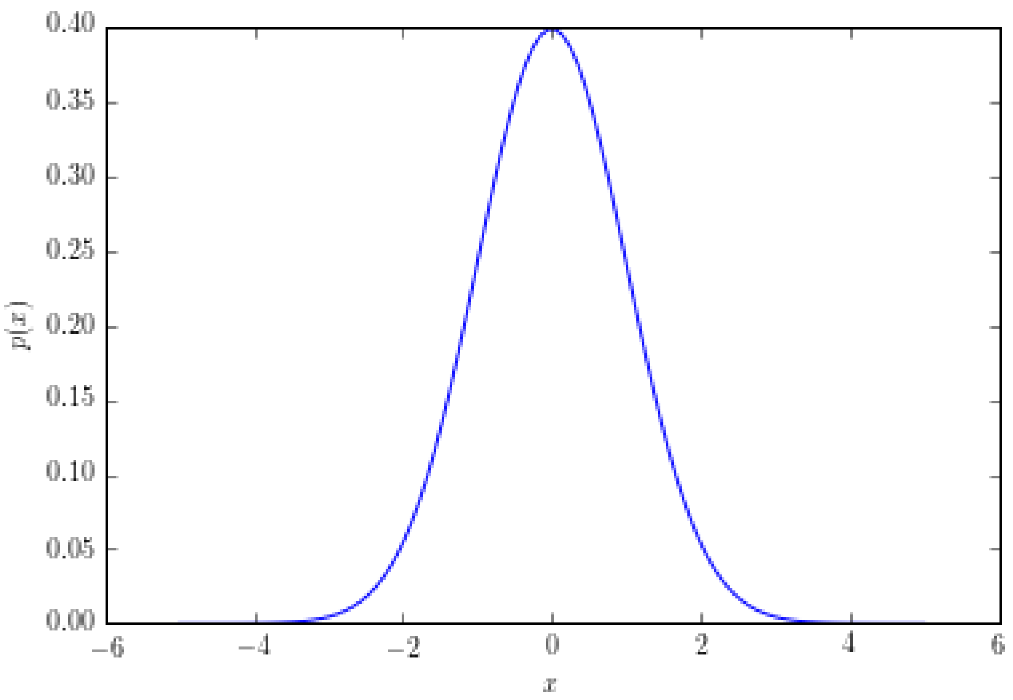
```
In [2]: import math
        import numpy as np
        import scipy as sp
        import matplotlib.pyplot as plt
        %matplotlib inline
```

0.0.1 Given some distribution $P(\theta|D)$, how do we use or understand its properties?

```
In [3]: 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)
```

We can plot it:

```
In [4]: x = np.linspace(-5,5,100)
        plt.plot(x, p_gaussian(x))
        plt.xlabel("$x$")
        plt.ylabel("$p(x)$")

Out[4]: <matplotlib.text.Text at 0x108807e10>
```



0.1 Moments

If we have an analytic (or just computable) form for the distribution, we can work out the moments by integration:

$$\langle x^n \rangle = \int x^n p(x)\, dx$$

```
In [5]: import scipy.integrate as si
        norm = si.quad(p_gaussian,-10,10)[0]
        mean =  si.quad(lambda x: x*p_gaussian(x), -10, 10)[0]
        var = si.quad(lambda x: x*x*p_gaussian(x), -10, 10)[0] - mean**2
        print "normalization: %f" % norm
        print "mean: %f " % mean
        print "variance: %f " % var
```

```
normalization: 1.000000
mean: 0.000000
variance: 1.000000
```

0.2 Random numbers

Sometimes, though, we want to generate [random] numbers as if they came from a given distribution. (As we will see later, we can do this in cases even in cases where the distribution itself is hard to compute.)

In some cases, this is easy. Here are random numbers from the uniform distribution over $x \in (0, 1)$

```
In [6]: np.random.rand(10)
```

```
Out[6]: array([ 0.41412928,  0.36326331,  0.0364867 ,  0.55610849,  0.99671092,
                0.66151529,  0.25614466,  0.57098066,  0.67090961,  0.12927589])
```

(Of course these are really deterministic *pseudo-random* numbers.)
Your language may be able to generate numbers from some other distributions.

```
In [7]: print np.random.__doc__
```

```
========================
Random Number Generation
========================

==================== =========================================================
Utility functions
==============================================================================
random_sample        Uniformly distributed floats over ``[0, 1)``.
random               Alias for `random_sample`.
bytes                Uniformly distributed random bytes.
random_integers      Uniformly distributed integers in a given range.
permutation          Randomly permute a sequence / generate a random sequence.
shuffle              Randomly permute a sequence in place.
seed                 Seed the random number generator.
==================== =========================================================

==================== =========================================================
Compatibility functions
==============================================================================
rand                 Uniformly distributed values.
randn                Normally distributed values.
ranf                 Uniformly distributed floating point numbers.
randint              Uniformly distributed integers in a given range.
==================== =========================================================

==================== =========================================================
Univariate distributions
==============================================================================
beta                 Beta distribution over ``[0, 1]``.
binomial             Binomial distribution.
chisquare            :math:`\chi^2` distribution.
exponential          Exponential distribution.
f                    F (Fisher-Snedecor) distribution.
gamma                Gamma distribution.
geometric            Geometric distribution.
gumbel               Gumbel distribution.
hypergeometric       Hypergeometric distribution.
laplace              Laplace distribution.
logistic             Logistic distribution.
lognormal            Log-normal distribution.
logseries            Logarithmic series distribution.
negative_binomial    Negative binomial distribution.
noncentral_chisquare Non-central chi-square distribution.
noncentral_f         Non-central F distribution.
normal               Normal / Gaussian distribution.
pareto               Pareto distribution.
poisson              Poisson distribution.
power                Power distribution.
rayleigh             Rayleigh distribution.
triangular           Triangular distribution.
```

```
uniform              Uniform distribution.
vonmises             Von Mises circular distribution.
wald                 Wald (inverse Gaussian) distribution.
weibull              Weibull distribution.
zipf                 Zipf's distribution over ranked data.
==================== =========================================================

==================== =========================================================
Multivariate distributions
==============================================================================
dirichlet            Multivariate generalization of Beta distribution.
multinomial          Multivariate generalization of the binomial distribution.
multivariate_normal  Multivariate generalization of the normal distribution.
==================== =========================================================

==================== =========================================================
Standard distributions
==============================================================================
standard_cauchy      Standard Cauchy-Lorentz distribution.
standard_exponential Standard exponential distribution.
standard_gamma       Standard Gamma distribution.
standard_normal      Standard normal distribution.
standard_t           Standard Student's t-distribution.
==================== =========================================================

==================== =========================================================
Internal functions
==============================================================================
get_state            Get tuple representing internal state of generator.
set_state            Set state of generator.
==================== =========================================================
```

0.2.1 We can generate random numbers from these distributions

```
In [8]: poissons = np.random.poisson(1, size=1200)
```

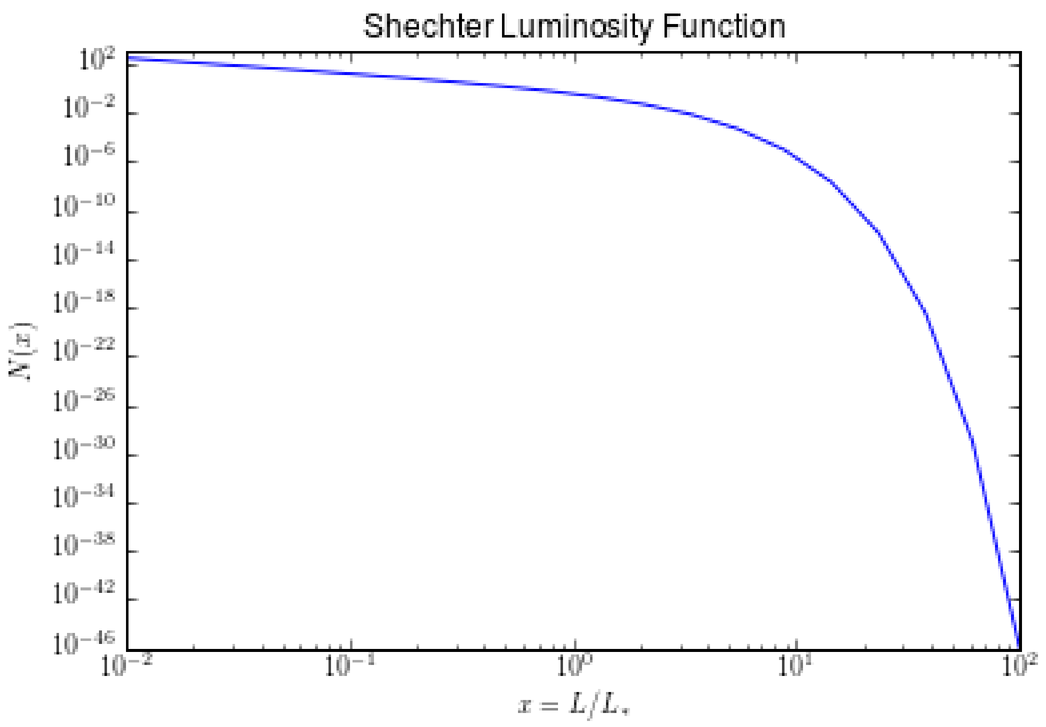
```
In [9]: np.mean(poissons), np.var(poissons)
```

```
Out[9]: (0.96499999999999997, 0.91710833333333333)
```

You can do a lot more with samples: they are essentially *simulations of random processes.*
Yesterday, we saw an example in the flux-density distribution
For example, if you have a luminosity function $P(L)$, samples from that distribution will be a simulation of a galaxy population

```
In [10]: def schechter(x, phi_star=1.0, a=-1.25):
             """ the luminosity function n(x) with x = L/Lstar """
             return phi_star * x**a * np.exp(-x)
```

```
In [11]: logxarr = np.logspace(-2,2,20)
         plt.loglog(logxarr, schechter(logxarr))
         plt.xlabel("$x=L/L_*$")
         plt.ylabel("$N(x)$")
         plt.title("Shechter Luminosity Function");
```



0.3 What to do with Samples

Samples x_i, $(i = 1, \ldots, N)$, from a distribution $p(x)$, satisfy

$$\lim_{N\to\infty} \frac{1}{N} \sum_i f(x^{(i)}) = \int f(x)p(x)\, dx \equiv \langle f(x) \rangle$$

so we can use the samples to calculate estimates for the mean $\langle x \rangle$ and other moments of the distribution.

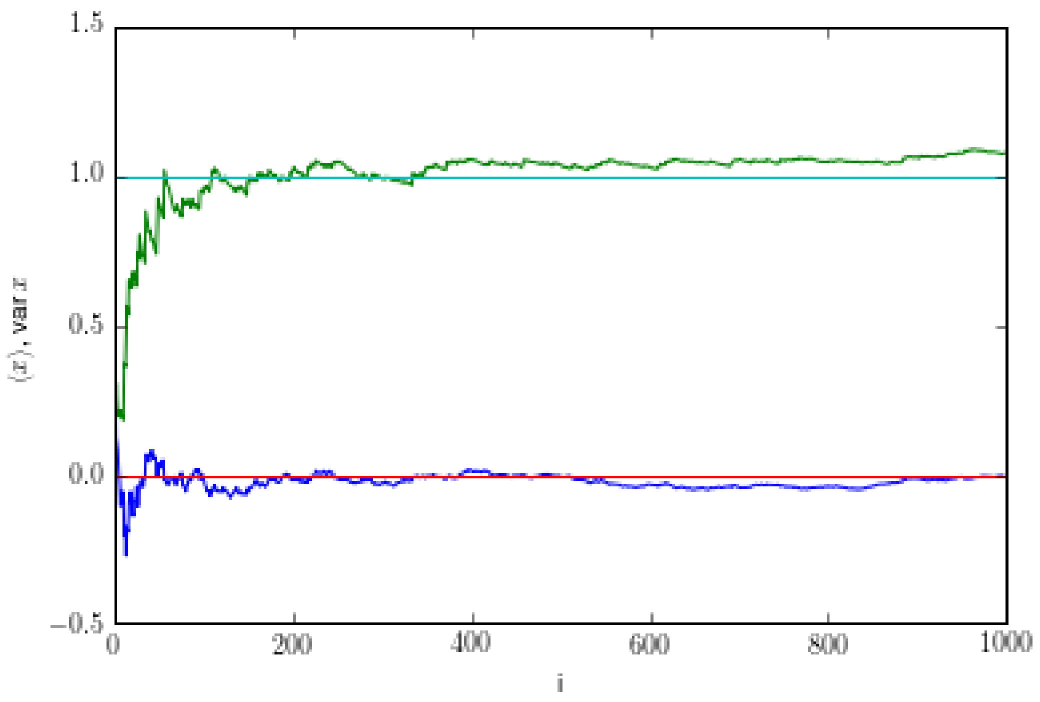
Let's check with a Gaussian:

```
In [12]: nsamp = 1000
         gaussian_samples = np.random.randn(nsamp)   ### 1000 random numbers from a standard normal dis
         avg = np.mean(gaussian_samples)
         var = np.var(gaussian_samples)
         print "%f ± %f → 0 ± 1" % (avg, var)
```

```
-0.009977 ± 1.080914 → 0 ± 1
```

Let's see how these are built up

```
In [13]: avgs = [np.mean(gaussian_samples[:n]) for n in range(1,nsamp)]
         vars = [np.var(gaussian_samples[:n]) for n in range(1,nsamp)]
         plt.plot(avgs)
         plt.plot(vars)
         plt.plot([0,999],[0,0])
         plt.plot([0,999],[1,1])
         plt.ylim(-0.5,1.5)
         plt.xlabel("i")
         plt.ylabel("$\langle{x}\\rangle$, var $x$");
```

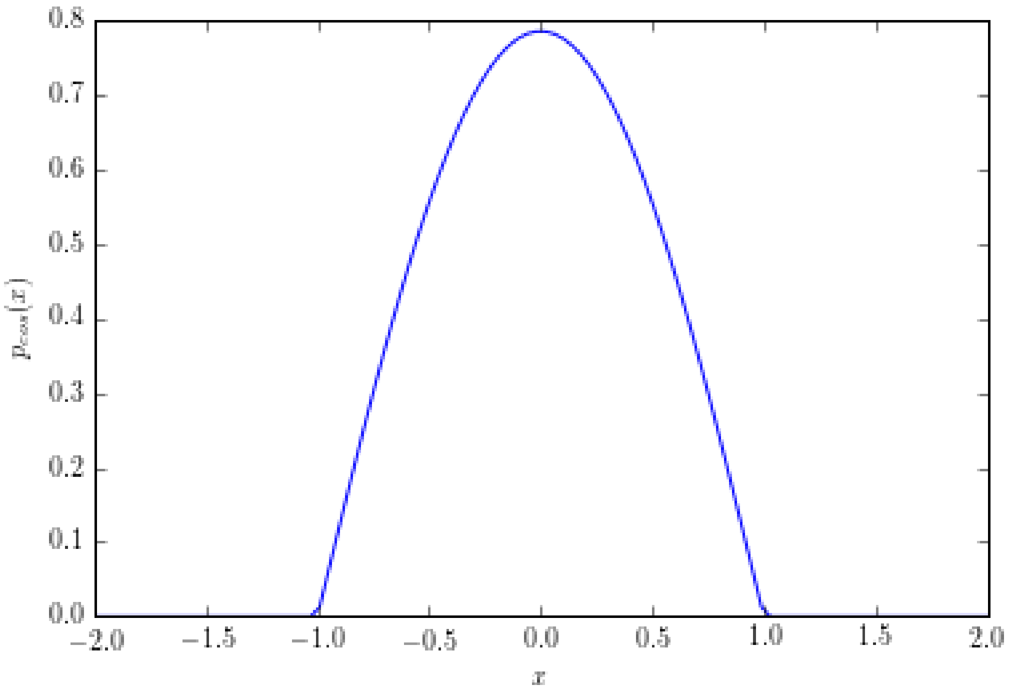


0.3.1 Other distributions

But you may need to sample from an essentially arbitrary $p(x)$.

```
In [14]: def p_cos_1(x):
             return 0 if (x<-1.0 or x>1.0) else np.cos(x*math.pi/2.0)*math.pi/4.0

         p_cos = np.vectorize(p_cos_1, otypes=[np.float])
         x = np.linspace(-2,2,100)
         plt.plot(x, p_cos(x))
         plt.xlabel("$x$")
         plt.ylabel("$p_{cos}(x)$")

         norm = si.quad(p_cos,-10,10)[0]
         mean =  si.quad(lambda x: x*p_cos(x), -2, 2)[0]
         var = si.quad(lambda x: x*x*p_cos(x), -2, 2)[0] - mean**2
         print "normalization: %f" % norm
         print "mean: %f " % mean
         print "variance: %f (= %f = 1-8/π^2)" % (var, 1-8/math.pi**2)
```

```
normalization: 1.000000
mean: 0.000000
variance: 0.189431 (= 0.189431 = 1-8/π^2)
```

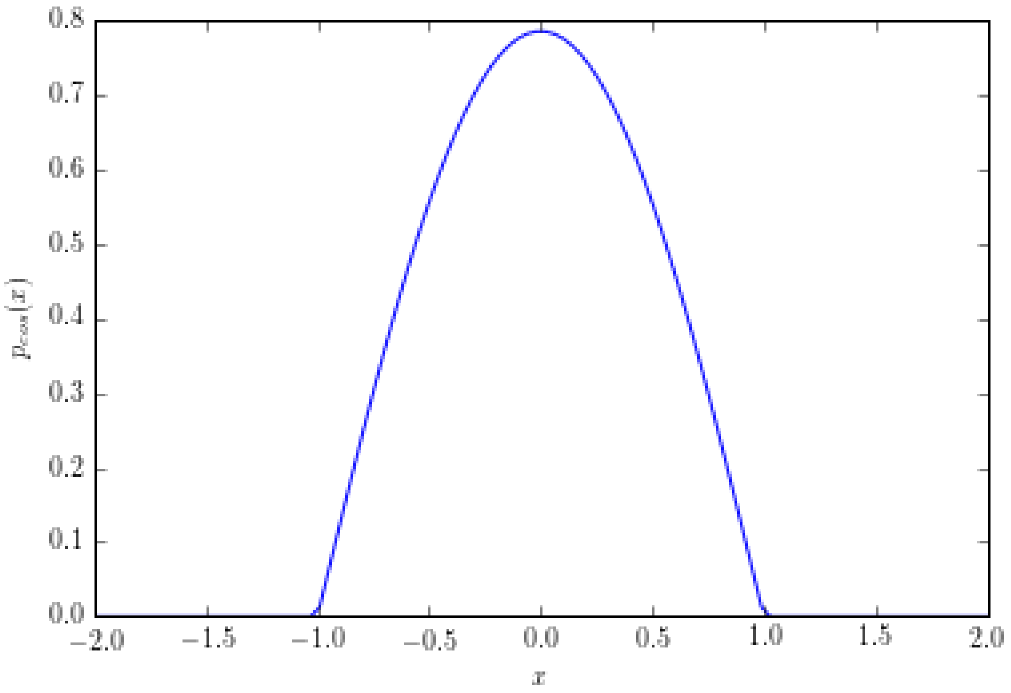


0.3.2 Rejection Sampling

One good tool is called *rejection sampling*.

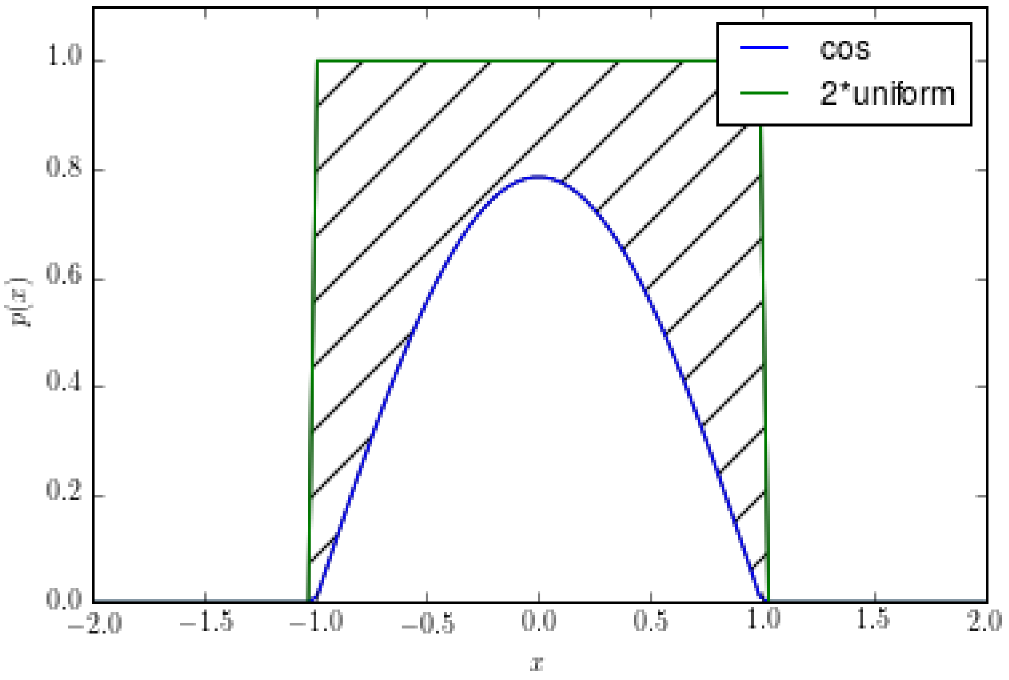
Consider the histogram of samples that you will generate: you want to fill in the area underneath the curve of $p(x)$.

```
In [15]: x = np.linspace(-2,2,100)
         plt.plot(x, p_cos(x))
         plt.xlabel("$x$")
         plt.ylabel("$p_{cos}(x)$")

Out[15]: <matplotlib.text.Text at 0x1092f2a50>
```



We don't know how to sample from this $p_{\cos}(x)$. But we do know how to sample from (among others) the uniform distribution, $u(x)$.

```
In [16]: def u_1(x):
             return 0 if (x<-1.0 or x>1.0) else 0.5
         u = np.vectorize(u_1, otypes=[np.float])

         plt.plot(x, p_cos(x), label="cos")
         plt.plot(x, 2*u(x), label="2*uniform")    ### note scale factor of 2
         plt.fill_between(x, 2*u(x), p_cos(x), hatch="/", facecolor='w')
         plt.xlabel("$x$")
         plt.ylabel("$p(x)$")
         plt.ylim(0,1.1)
         plt.legend();
```



What we want to do is *reject* some fraction of the samples from $u(x)$ — the ones in the shaded region — so that we get the right numbers for $p_{\cos}(x)$.

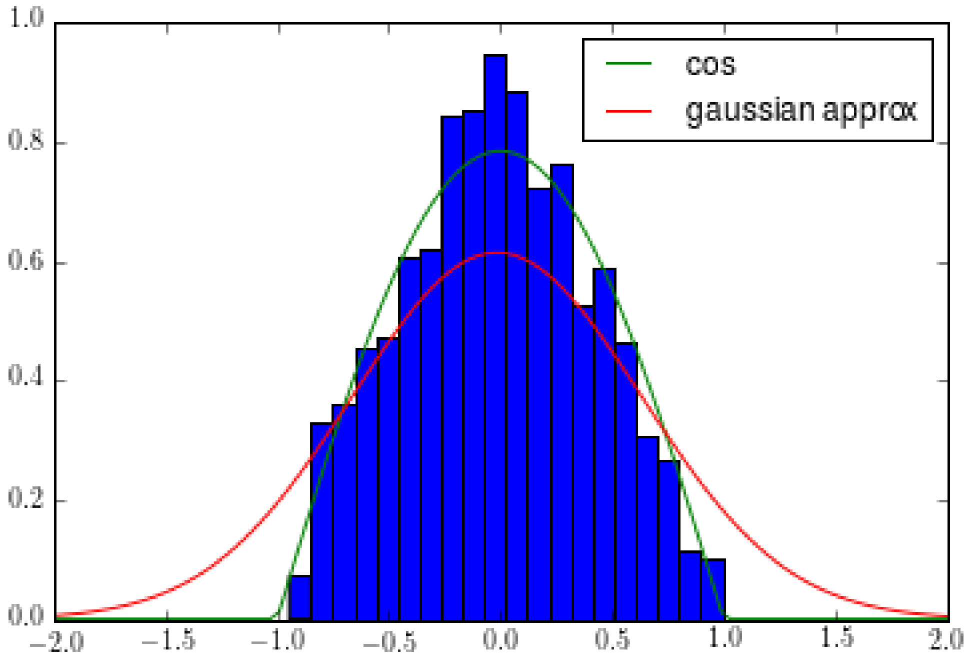
At a particular value of x, we need to reject exactly the fraction of samples corresponding to the ratio of $p_{\cos}(x)$ to $u(x)$.

```
In [17]: def rejection_sample(p=p_cos, 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
```

```
In [18]: rsamp = np.array([rejection_sample() for _ in range(1000)])

         sample_mean = np.mean(rsamp)
         sample_var = np.var(rsamp)
         print "sample mean: %f ~ %f" % (sample_mean, 0)
         print "sample var:  %f ~ %f" % (sample_var, 1-8/math.pi**2)
```

```
plt.hist(rsamp, normed=True, bins=20)
x = np.linspace(-2,2,100)
plt.plot(x, p_cos(x), label="cos")
plt.plot(x, p_gaussian(x, mu=sample_mean, sigma=np.sqrt(sample_var)), label="gaussian approx")
plt.legend();
```

```
sample mean: -0.014500 ~ 0.000000
sample var:  0.177231 ~ 0.189431
```



Note: * We don't require our other distribution to be uniform * But it does have to be one we can easily sample from * And we do need to know the maximum value of the desired $p(x)$ so that we have

$$[M \times u(x)] > p(x)$$

everywhere. * The Gaussian approximation is OK, but gets the tails badly wrong (cf. the central limit theorem).

0.4 Other tools for generating samples.

0.4.1 Changing varibles

If we can sample from $p(x)$, we can generate samples from a variety of distributions related to $p(x)$.

Consider the distribution $q(y)$ such that $y = y(x)$ and x is drawn from $p(x)$. These distributions satisfy

$$p(x)\ dx = q(y)\ dy\ .$$

From this, we can deduce that

$$q(y) = p(x) \left| \frac{dy}{dx} \right|^{-1}$$

where we need the inverse function $x = x(y)$ on the right-hand side, and must also express the (Jacobian) derivative in terms of y. (This can be generalized using a Jacobian determinant to multivariate distributions and non-invertible functions).

Exercise: If we have a distribution $p(x)$ for $x = \log(y)$ from which we can draw samples x_i, how can we use this to generate samples of y itself?

0.5 Multivariate distributions

Almost nothing that we have done so far depends on the fact that our "random variable" x is a single (scalar) parameter — it could just as easily be a vector of different parameters, $\vec{x}$. For example: * the mass of the sun, $M_\odot$ — a single parameter * the parameters of the ΛCDM universe, $\{H_0, n_s, \Omega_m, \Omega_\Lambda, \Omega_b\}$ * the individual values of the CMB power spectrum, C_ℓ, $\ell = \{2, 3, 4, \ldots\}$

How do we characterise these multivariate distributions?

0.5.1 Moments

We can just as easily write down moments of a multivariate distribution, $p(\vec{x})$:

$$\langle x_i x_j \cdots \rangle = \int d^n x \; (x_i x_j \cdots) p(\vec{x})$$

Just as a univariate Gaussian distribution is completely described by its mean, μ, and variance, σ^2, a multivariate Gaussian distribution is described by its vector of means $\vec{\mu} = \langle \vec{x} \rangle$ and the covariance matrix

$$C_{ij} = \langle (x_i - \mu_i)(x_j - \mu_j) \rangle$$

We can often use a gaussian with the same mean and variance as those of the samples as an approximation to the distribution.

0.5.2 Multivariate Samples and marginalization

Samples from a multivariate distribution can be very useful. In particular, marginalizing over one or the other is equivalent to just *ignoring the samples of that variable.* I.E., If we have a list of samples from $p(x, y)$:

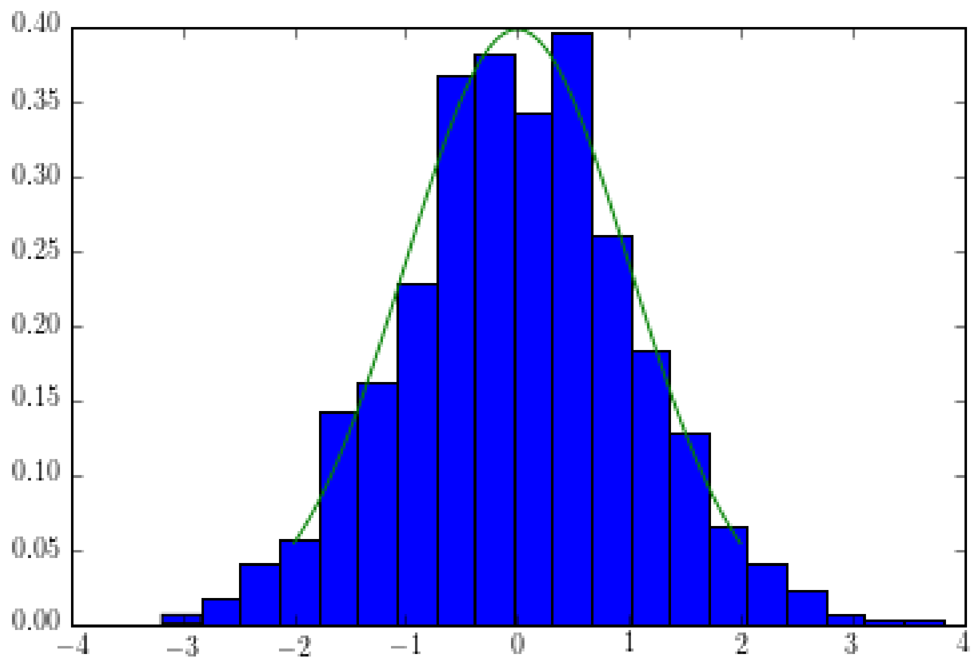
```
x_1 y_1
x_2 y_2
...
x_n y_n
```

In this case, * x_i are samples from $p(x) = \int dy \; p(x, y)$, and * y_i are samples from $p(y) = \int dx \; p(x, y)$

0.6 Plotting and Summarizing

Especially in one dimension, your plotting package may be able to do things for you:

```
In [19]: plt.hist(gaussian_samples, normed=True, bins=20)
         plt.plot(x, p_gaussian(x));
```



But there are some useful general tools for visualizing samples.

Consider a much simpler problem, when we have both the samples, x_i, themselves, as well as the actual value of the distribution at those points, $p(x_i)$. (It's much harder without that information – the general problem of characterising the distribution of samples in many dimensions is very hard!)

We generally want to characterise high-probability regions of $p(x)$, i.e., those regions that have the largest values of $p(x)$ and enclose some fraction α of the total probability:

$$\alpha = \int_{p>q(\alpha)} p(x)\, dx$$

where the value q depends on the chose level α. For $\alpha = 1$, this is just $q = 0$ and gives the whole range of x; for $\alpha = 0$ this is just any q greater than the maximum value of p. Typically, we try to find those regions that enclose the equivalent of n-σ for a Gaussian distribution.

This seems like a complicated definition, but it's easy to approximate from N samples $x^{(i)}$: * Sort the samples in order of decreasing probability $p\left(x^{(i)}\right)$. * Work your way down the list until you have $\alpha \times N$ samples: all of those samples come from the level-α region, and the last value gives an approximation $q(\alpha) \approx p\left(x^{(i)}\right)$

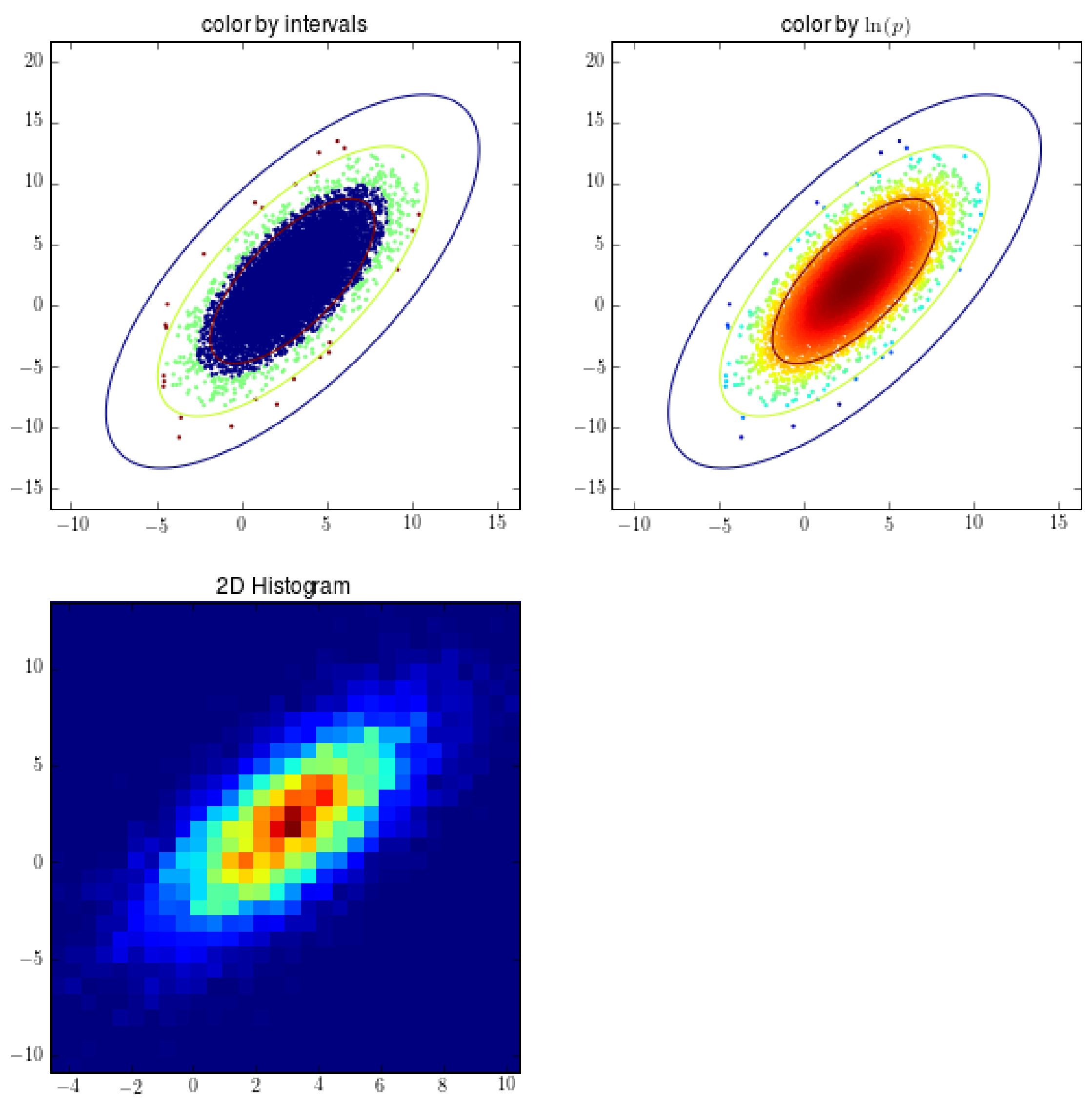
```
In [20]: sorted_probabilities = np.sort(p_gaussian(gaussian_samples))
         n = len(gaussian_samples)
         alphas = [0.683, 0.954, 0.9973]
         q_levels = [sorted_probabilities[np.round((1-a)*n)] for a in alphas]
         print q_levels
```

```
[0.23669849853114167, 0.046010471571392327, 0.0025621679947923979]
```

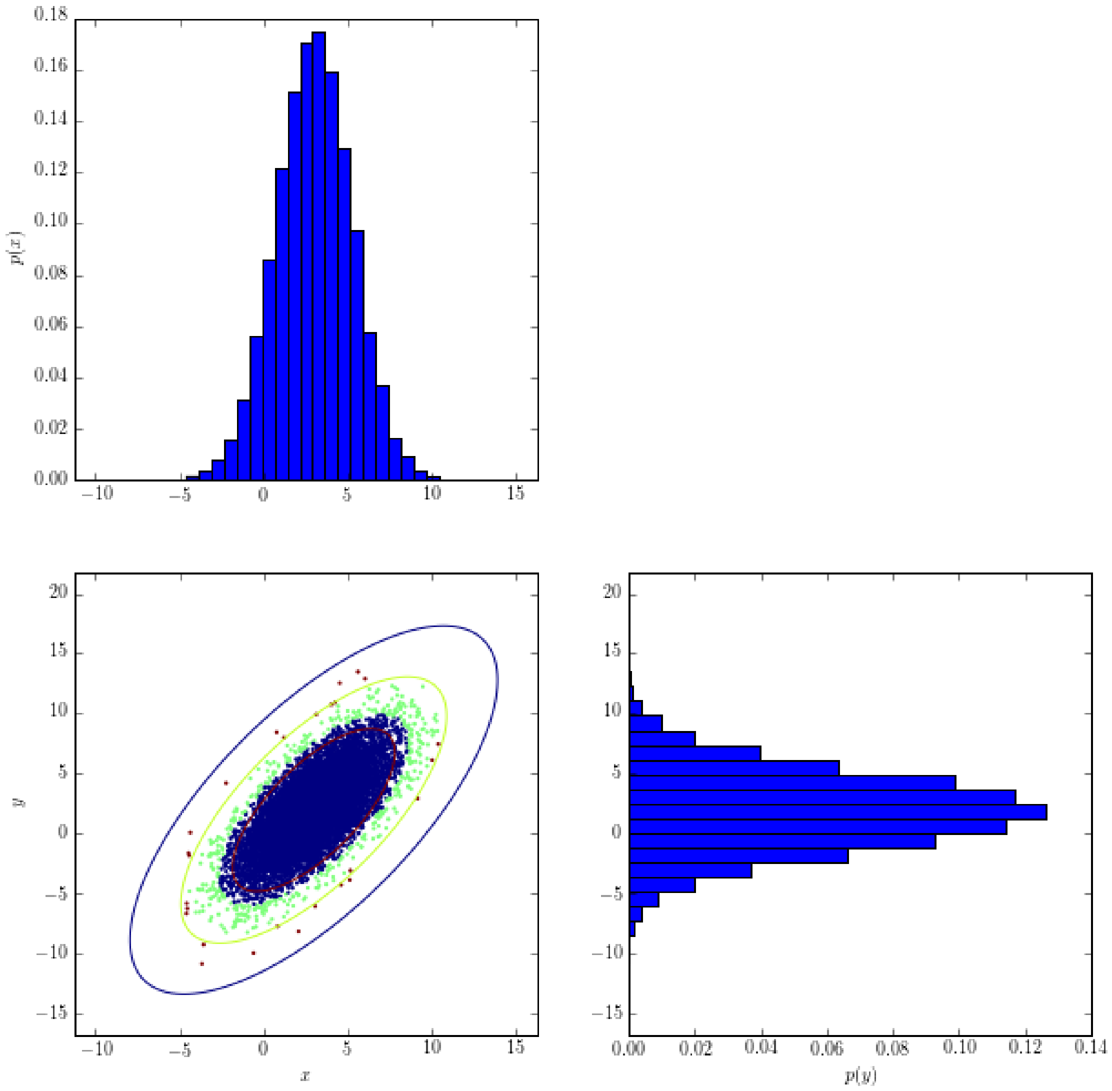
```
In [21]: covar = np.array([[5.0,  5.0],
                           [5.0, 10.0]])
         mean = (3,2)
         nsamples = np.random.multivariate_normal(mean=mean, cov=covar, size=10000)
         psamples = np.array([p_multigaussian(xi, mean, covar) for xi in nsamples])
```

```
        sorted_probabilities = np.sort(psamples)
        n = len(nsamples)
        alphas = [0.683, 0.954, 0.9973]    ### these are not the right levels for 1-, 2-, 3-sigma for 
        q_levels = [sorted_probabilities[np.round((1-a)*n)] for a in alphas]

        colors = np.zeros_like(psamples)
        for col, lev in enumerate(q_levels):
            colors[psamples<lev] = col

        samples_mean = np.average(nsamples.transpose(), axis=1)
        samples_covar = np.cov(nsamples.transpose())
        print "mean: ", samples_mean
        print "covar: ", samples_covar
        xarr = np.linspace(-10,15,100)
        yarr = np.linspace(-15,20,100)
        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.fla
        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(psamples), 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:  [ 2.99109787  1.96168081]
covar:  [[ 5.06887825  5.02110853]
 [ 5.02110853  9.97389658]]
```



```
In [22]: 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)
```



0.7 Importance Sampling

The basic idea of importace sampling is simple. Consider our fundamental "theorem of sampling":

$$\lim_{N\to\infty} \frac{1}{N} \sum_i f(x^{(i)}) = \int f(x)p(x)\, dx \equiv \langle f(x) \rangle_p$$

(where we add a subscript to $\langle \cdots \rangle$ to indicate that the expectation is taken with respect to the distribution $p(x)$.)

We can multiply and divide by some function $q(x)$ inside the integral:

$$\int \frac{f(x)}{q(x)} q(x) p(x)\, dx \equiv \left\langle \frac{f(x)p(x)}{q(x)} \right\rangle_q$$

but this is still equal to our original $\langle f(x) \rangle_p$.

Hence, if we have samples from $q(x)$, we can estimate averages of $f(x)$ under $p(x)$:

$$\lim_{N\to\infty} \frac{1}{N} \sum_i \frac{f(x^{(i)})p(x^{(i)})}{q(x^{(i)})} = \langle f(x) \rangle_p$$

We can think of this as just a re-weighting of our samples by $w_i = p(x^{(i)})/q(x^{(i)})$.

This is particularly useful for re-analyzing MCMC chains. If our chain was created with some prior $\pi_1(x)$, we can "substitute in" a new prior $\pi_2(x)$ by reweighting by π_2/π_1. For example, we may have created some chains from Planck with a uniform prior on the tensor-to-scalar ratio, r, but want to re-analyze our results with the BICEP2 results $b(r)$, so we just reweight by $b(r)/u(r) = b(r)$.

Note that the reweighted samples are not themselves individual random samples from the new distribution. So, when we are using the graphical techniques from before to find confidence intervals, we don't work our way down from the highest probability one at a time – instead, we now add up the w_i until we get the right fraction of the total weight.