Gaussian Mixture Modelling tutorial
by Eugene Vasiliev

This is an illustration of the Gaussian mixture modelling method in application to the line-of-sight velocities of stars in and around Milky Way globular clusters. We use the data collected by Baumgardt and colleagues at this webpage:
https://people.smp.uq.edu.au/HolgerBaumgardt/globular/appendix/appendix.html

These files contain a compilation of velocities from various sources, and do not clean up the sample in any particular way (i.e., list all stars ever observed spectroscopically within some distance from the cluster centre). Obviously there will be many contaminants -- the goal is to get rid of them (probabilistically) and estimate the mean velocity and its dispersion for cluster members. We pick up one random cluster from this dataset, say, NGC 288.

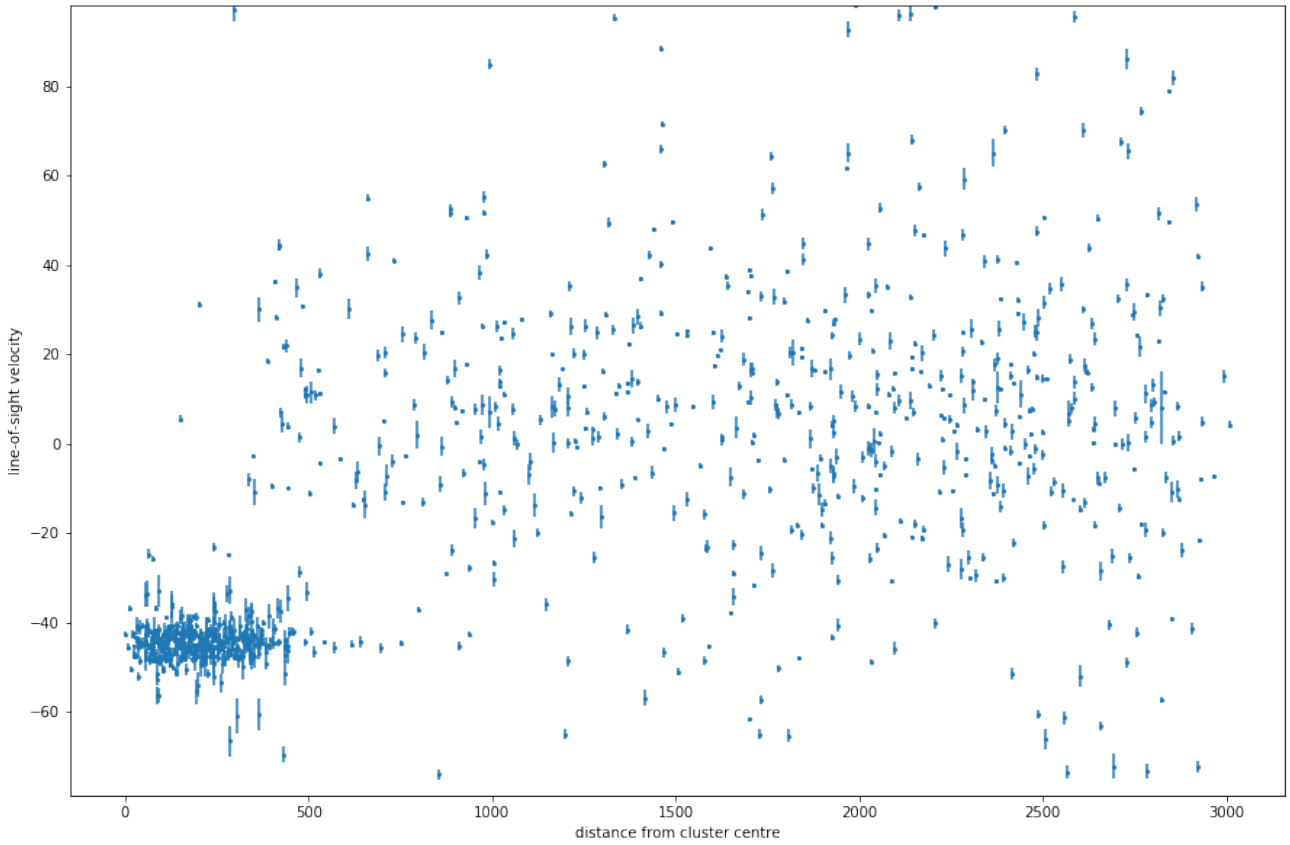
This example shows several implementations of the mixture modelling approach (restricted here for the 1d case and two components). First, we write the likelihood function for the mixture model (this function is the same for all variants), and use a conventional black-box minimization routine for a scalar function from scipy. Note that this routine implements several minimization methods, and not all of them may actually converge to a solution for this problem. Second, we write a quick-and-dirty homemade Expectation/Maximization algorithm, which is a numerically more stable way of solving this particular type of minimization problems (note however that it is still not guaranteed to find a global maximum of likelihood, and the result may depend on the starting point). Third and fourth, we try existing implementations of the Extreme Deconvolution approach from third-party libraries. Finally, all of the steps above only produce the best-fit parameters, but no uncertainties. In order to estimate the confidence intervals on these parameters, we run a MCMC simulation with the log-likelihood function from the first step, using the third-party library EMCEE.

In [1]:
```
import numpy, matplotlib.pyplot as plt
%matplotlib inline
```

In [2]:
```
# choose your own cluster!
# unfortunately, the format of these text files is not particularly friendly
# for loading into numpy, as the number of columns is not the same for all rows.
# we explicitly specify which columns we want to load (note that this assumes
# that the first _two_ columns contain the cluster name, like "ngc 288")
table = numpy.loadtxt('ngc288.txt', skiprows=2, usecols=(4,5,6))
dist = table[:,0]   # distance from cluster centre
vval = table[:,1]   # value of the line-of-sight velocity
verr = table[:,2]   # uncertainty (error estimate)
```

In [3]:
```
# a visual inspection shows that the dataset contains mostly cluster members
# tightly spaced around a particular velocity at small distances, and becomes
# progressively more dominated by field stars at larger distances.
# in this example, we are not using the distance information, but in principle
# it could aid the classification.
plt.figure(figsize=(15,10))
```

```
plt.errorbar(dist, vval, yerr=verr, markersize=2, fmt='o' )
plt.ylim(numpy.percentile(vval, [1,99]))   # narrow down the y range to cut outliers
plt.xlabel('distance from cluster centre')
plt.ylabel('line-of-sight velocity')
```



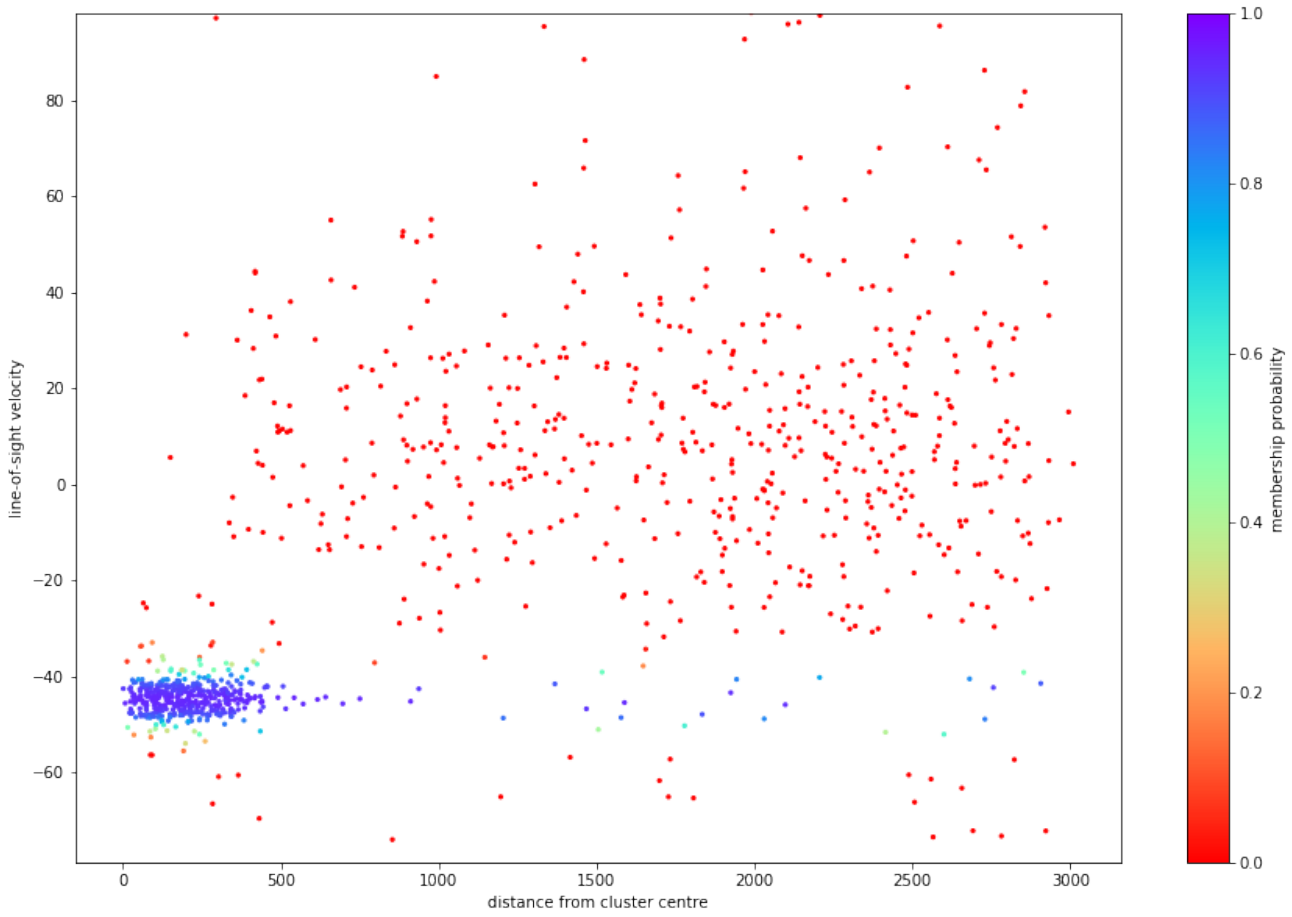
In [4]:
```
# the good old Gaussian function
def Gaussian(x, mean, sigma):
    return (2*numpy.pi)**-0.5 / sigma * numpy.exp(-0.5 * (x-mean)**2 / sigma**2)

# the model for the observed distribution of stars in velocity is a weighted sum
# of two gaussians, with 5 model parameters (mean and width for each component,
# and the fraction of the first component - this sequence will be the same in all methods).
# The widths of the two gaussians are broadened by measurement uncertainties
# (different for each star).
# The log-likelihood of the entire dataset is the sum of log-likelihoods for each datapoint.
# We also make sure that the input parameters are in the valid range and the first
# component is narrower, otherwise return a very large negative number
# (or even -infinity - not sure if this works with all minimization methods).
def loglikelihood(params):
    mean1, sigma1, mean2, sigma2, frac1 = params
    if frac1<=0 or frac1>=1 or sigma1<0 or sigma2<0 or sigma2<sigma1:
        return -1e100
    df1 = Gaussian(vval, mean1, (sigma1**2 + verr**2)**0.5)
    df2 = Gaussian(vval, mean2, (sigma2**2 + verr**2)**0.5)
    dfmix = frac1 * df1 + (1-frac1) * df2
    return numpy.sum(numpy.log(dfmix))
```

```
        # a complementary function that returns the membership probability of each star
        # (probability of belonging to the first component)
        def membership_probability(params):
            mean1, sigma1, mean2, sigma2, frac1 = params
            df1 = Gaussian(vval, mean1, (sigma1**2 + verr**2)**0.5)
            df2 = Gaussian(vval, mean2, (sigma2**2 + verr**2)**0.5)
            return frac1 * df1 / (frac1 * df1 + (1-frac1) * df2)
```

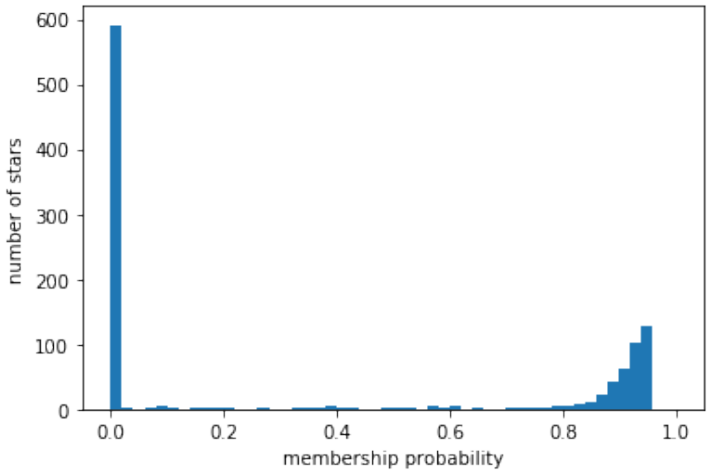
In [5]:
```
        # first method: standard black-box optimization routine -
        # we need to start from a reasonable initial guess though..
        # take the mean and dispersion of the innermost 25% of stars
        # as the narrow component's starting values, and the overall
        # mean and dispersion -- as the broad component's starting point.
        # One may need to choose a different method for 'minimize',
        # e.g., Nelder-Mead, Powell, etc.
        import scipy.optimize
        method     = 'Nelder-Mead'
        #method     = 'Powell'
        innerpart  = dist < numpy.percentile(dist, 25)
        initparams = [numpy.mean(vval[innerpart]), numpy.std(vval[innerpart]),
                      numpy.mean(vval), numpy.std(vval), 0.5]
        # since we call a minimizer not a maximizer, our function is minus log-likelihood
        bestfitparams = scipy.optimize.minimize(lambda params: -loglikelihood(params),
            initparams, method=method).x
        print(initparams, loglikelihood(initparams))
        print(bestfitparams, loglikelihood(bestfitparams))
```

```
[-44.891042471042475, 12.518448628169196, -14.544497098646037, 43.30497925892082, 0.5] -5275.30783648
[-44.72875327   2.30000771   2.76035943  46.10547634   0.36475791] -4887.41756800349
```

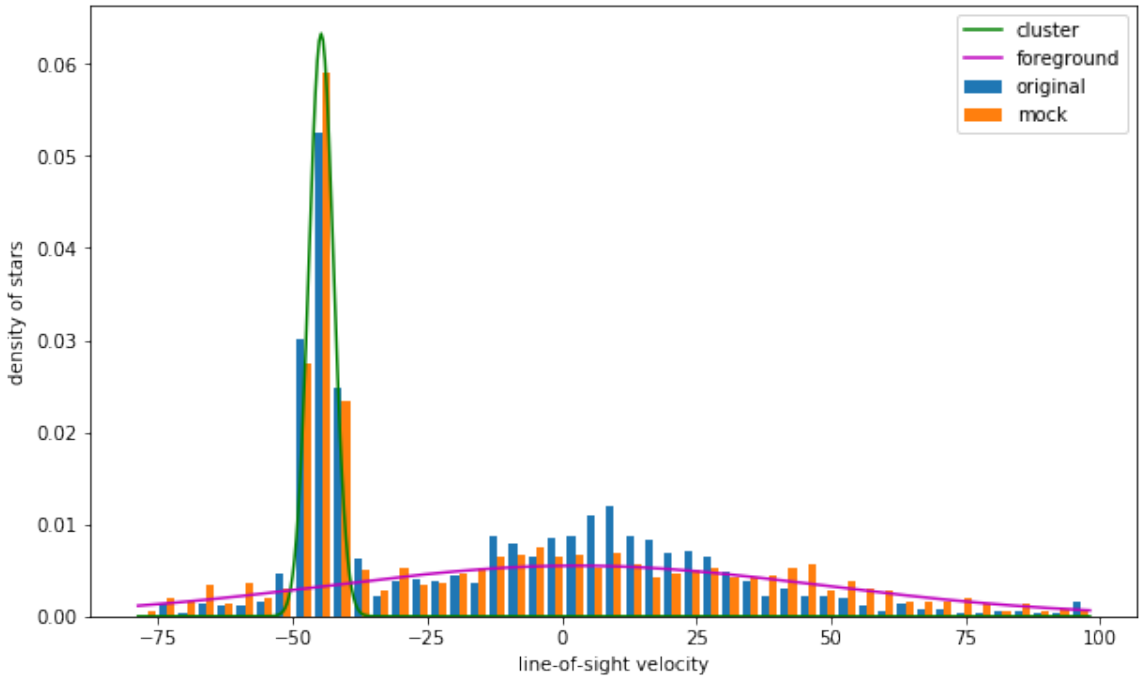
In [6]:
```
        # now plot the data again, coloured by membership probability
        plt.figure(figsize=(15,10))
        plt.colorbar(plt.scatter(dist, vval, s=5, marker='o',
            c=membership_probability(bestfitparams), cmap='rainbow_r', vmin=0, vmax=1),
            label='membership probability')
        plt.ylim(numpy.percentile(vval, [1,99]))
        plt.xlabel('distance from cluster centre')
        plt.ylabel('line-of-sight velocity')
```



In [7]:
```
# and also plot the distribution (histogram) of membership probabilities -
# ideally it should have two well separated peaks around 0 and 1,
# but in some cases the classification is less certain and there is a large
# fraction of stars with intermediate probabilities
plt.hist(membership_probability(bestfitparams), bins=numpy.linspace(0,1,51))
plt.xlabel('membership probability')
plt.ylabel('number of stars')
```



In [8]:
```
# it is always a good idea to check how well the model describes the data,
# by generating mock data from the model with the best-fit parameters
# and adding an observational error to each datapoint,
# then plotting the distribution of values of the original and the mock datasets
def sample_from_mixture_model(params, N):
    mean1, sigma1, mean2, sigma2, frac1 = params
    N1 = int(round(N*frac1))
    N2 = N-N1
    # sample the true values
    samples = numpy.hstack([
        numpy.random.normal(size=N1) * sigma1 + mean1,
        numpy.random.normal(size=N2) * sigma2 + mean2 ])
    numpy.random.shuffle(samples)  # shuffle their order
    # add errors drawn from Gaussians with each star's measurement uncertainty
    samples += numpy.random.normal(size=N) * verr
    return samples

def plot_model(params):
    mean1, sigma1, mean2, sigma2, frac1 = params
    grid = numpy.linspace(numpy.percentile(vval,1), numpy.percentile(vval,99), 500)
    plt.plot(grid, Gaussian(grid, mean1, sigma1) *    frac1 , 'g', label='cluster')
    plt.plot(grid, Gaussian(grid, mean2, sigma2) * (1-frac1), 'm', label='foreground')

plt.figure(figsize=(10,6))
plt.hist([vval, sample_from_mixture_model(bestfitparams, len(vval))],
    bins=numpy.linspace(numpy.percentile(vval,1), numpy.percentile(vval,99), 50),
    label=['original','mock'], density=True)
plot_model(bestfitparams)
plt.legend()
plt.xlabel('line-of-sight velocity')
plt.ylabel('density of stars')
```



In [9]:
```
# second method: homemade implementation of the expectation/maximization algorithm.
# the 'expectation' step is essentially the computation of membership probabilities
# for the given (not necessarily optimal) values of parameters,
# and the maximization step in the simple case of no measurement errors is just
# the weighted mean and dispersion estimate for each of the two components.
def maximization_simple(memberprobs, params=None):
    sum1  = numpy.sum(memberprobs)
    sum2  = len(memberprobs) - sum1
    mean1 = numpy.sum(vval *    memberprobs ) / sum1
    mean2 = numpy.sum(vval * (1-memberprobs)) / sum2
    sigma1=(numpy.sum((vval-mean1)**2 *    memberprobs ) / sum1)**0.5
    sigma2=(numpy.sum((vval-mean2)**2 * (1-memberprobs)) / sum2)**0.5
    frac1 = sum1 / len(memberprobs)
    return numpy.array([mean1, sigma1, mean2, sigma2, frac1])

# in the case of measurement errors, maximization (estimate of the mean and sigma
# of each gaussian component) is more complicated, and essentially performs the
# deconvolution of measurement errors, while still weighting each datapoint with
# its probability of membership in the given component.
# the expressions below are simplified from eq.16 in Bovy+(2011) for 1d and 2 comp.
# Note that this is an iterative procedure even in the case of one component,
# as it uses the mean/sigma from the previous step
def maximization_with_deconvolution(memberprobs, params):
    mean1, sigma1, mean2, sigma2, _ = params   # previous estimates of mean and sigma
    sum1  = numpy.sum(memberprobs)
    sum2  = len(memberprobs) - sum1
    Ti1   = sigma1**2 + verr**2
    Ti2   = sigma2**2 + verr**2
    bi1   = (vval * sigma1**2 + mean1 * verr**2) / Ti1
    bi2   = (vval * sigma2**2 + mean2 * verr**2) / Ti2
    Bi1   = verr**2 * sigma1**2 / (verr**2 + sigma1**2)
    Bi2   = verr**2 * sigma2**2 / (verr**2 + sigma2**2)
    mean1 = numpy.sum(bi1 *    memberprobs)  / sum1
    mean2 = numpy.sum(bi2 * (1-memberprobs)) / sum2
    sigma1=(numpy.sum(((bi1-mean1)**2 + Bi1) *    memberprobs ) / sum1)**0.5
    sigma2=(numpy.sum(((bi2-mean2)**2 + Bi2) * (1-memberprobs)) / sum2)**0.5
    frac1 = sum1 / len(memberprobs)
    return numpy.array([mean1, sigma1, mean2, sigma2, frac1])

# since we do have measurement errors, choose the second function
maximization = maximization_with_deconvolution

# finally, the following routine performs several iterations of
# expectation (membership estimate) and maximization (parameter estimate) steps,
# until the log-likelihood reaches the stationary point (the maximum)
def expectation_maximization(initparams, tol=1e-5):
    params = initparams.copy()
    prevlogL = loglikelihood(params)
    for it in range(100):
        memberprobs = membership_probability(params)
        params = maximization(memberprobs, params)
```

```
            logL = loglikelihood(params)
            #print(it, params, logL)
            if logL < prevlogL + tol:
                break  # converged
            prevlogL = logL
        return params

    emparams = expectation_maximization(initparams)
    print(emparams, loglikelihood(emparams))
```

```
[-44.72869964   2.30042233   2.7617      46.10554555   0.3647766 ] -4887.417572843808
```

In [10]:
```
# third method: use the Extreme Deconvolution implementation by Bovy et al.(2011);
# one could also use the implementation from AstroML (named XDGMM) or PyGMMis -
# they all should produce the same result, but have somewhat different syntax.
import extreme_deconvolution
dir(extreme_deconvolution)
# initialize the process from the same starting point -
# the convergence (or lack of it) is quite sensitive to a good initial point!
means = numpy.array([initparams[0], initparams[2]]).reshape(2,1)
disps = numpy.array([initparams[1]**2, initparams[3]**2]).reshape(2,1,1)
fracs = numpy.array([initparams[-1], 1-initparams[-1]]).reshape(2,1)
# run the process
extreme_deconvolution.extreme_deconvolution(
    vval.reshape(-1,1), verr.reshape(-1,1,1)**2, fracs, means, disps)
# format the results in the same way as for the first method
xdparams = numpy.array([means[0], disps[0]**0.5, means[1], disps[1]**0.5,
    fracs[0]]).reshape(-1)
print(xdparams, loglikelihood(xdparams))
```

```
[-44.72839782243014 2.3026749045365644 2.768883483583691 46.10540567598803
 0.3648744626527402] -4887.417773590178
```

In [11]:
```
# fourth method: another implementation of the Extreme Deconvolution approach
# in AstroML (perhaps easier to work with than the Bovy's original one).
# unfortunately, it is less tunable (there is no option to explicitly provide
# the starting point) and as a result, does not always converge to the true global maximum
import astroML.density_estimation
gmm = astroML.density_estimation.XDGMM(2)
gmm.fit(vval.reshape(-1,1), verr.reshape(-1,1,1)**2)
order = numpy.argsort(gmm.V[:,0,0])  # sort the components in order of increasing dispersion
amparams = numpy.array([gmm.mu[order[0],0], gmm.V[order[0],0,0]**0.5,
    gmm.mu[order[1],0], gmm.V[order[1],0,0]**0.5, gmm.alpha[order[0]]])
print(amparams, loglikelihood(amparams))
```

```
/usr/local/lib/python3.7/site-packages/sklearn/mixture/_base.py:269: ConvergenceWarning:
Initialization 1 did not converge. Try different init parameters, or increase max_iter,
tol or check for degenerate data.
  % (init + 1), ConvergenceWarning)
```

```
[-16.26201583  32.94243117  24.03954256 134.1463346    0.95730112] -5222.347284251748
```

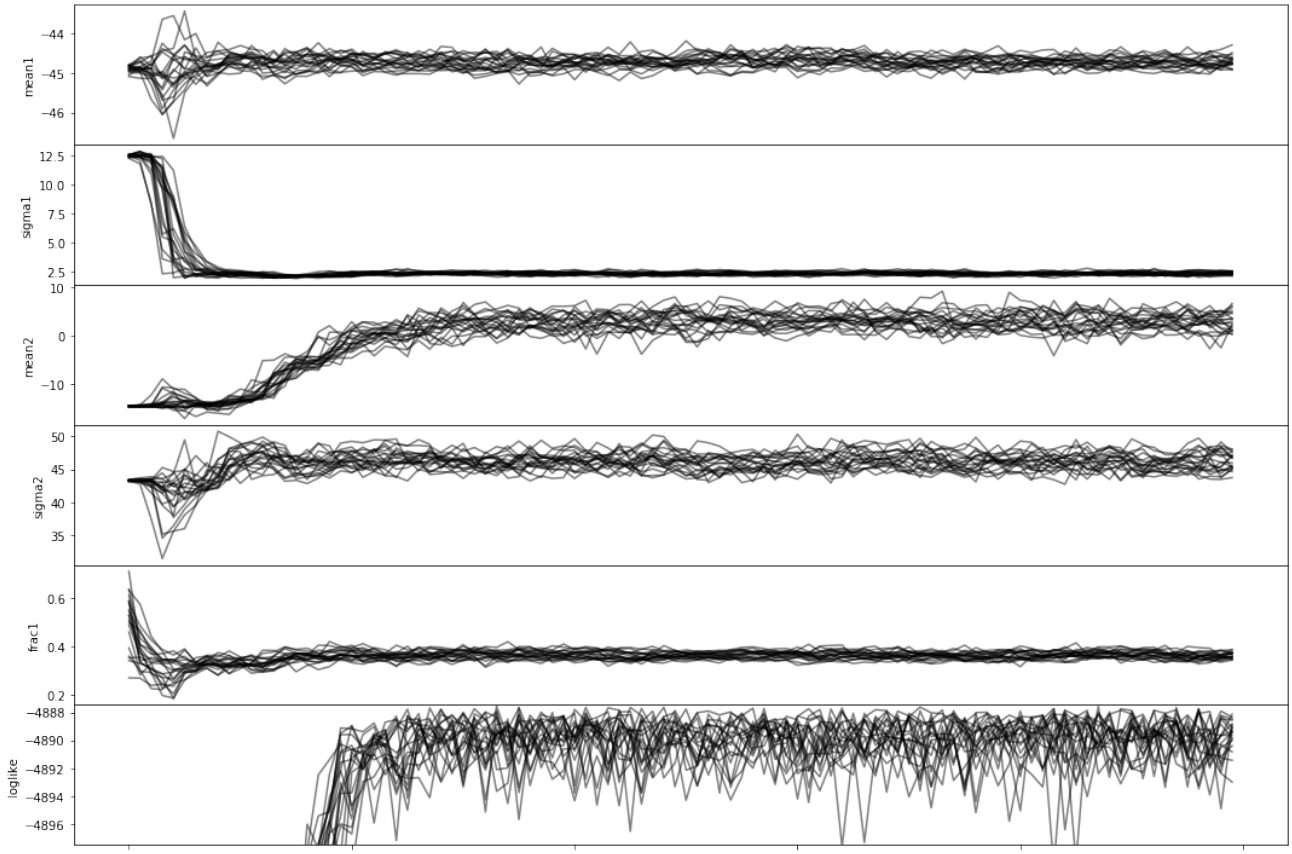
In [12]:
```
# however, we can hack into the XDGMM class, provide our own starting point,
# and then perform several iterations of the EM algorithm - with better results
gmm.mu = numpy.array([[initparams[0]], [initparams[2]]])
gmm.V  = numpy.array([[[initparams[1]**2]], [[initparams[3]**2]]])
gmm.alpha = numpy.array([initparams[4], 1-initparams[4]])
prevlogL = -numpy.inf
for i in range(100):
    gmm._EMstep(vval.reshape(-1,1), verr.reshape(-1,1,1)**2)
    logL = gmm.logL(vval.reshape(-1,1), verr.reshape(-1,1,1)**2)
    if logL < prevlogL + 1e-5:
        break
    prevlogL = logL
amparams = numpy.array([gmm.mu[0,0], gmm.V[0,0,0]**0.5,
    gmm.mu[1,0], gmm.V[1,0,0]**0.5, gmm.alpha[0]])
print(amparams, loglikelihood(amparams))
```

```
[-44.72869964   2.30042233   2.7617      46.10554555   0.3647766 ] -4887.417572843808
```

In [13]:
```
# finally, explore the parameter space more thoroughly by running
# an MCMC simulation and determining the confidence ranges of parameters,
# not just their best-fit values.
# for this example, we use the EMCEE code by D.Foreman-Mackey
import emcee, corner
# initialize a bunch of points around the starting guess
initwalkers = initparams + numpy.random.normal(size=(20, len(initparams))) * 0.1
# also make sure that they are all within bounds
initwalkers[:,(1,3)] = numpy.maximum(initwalkers[:,(1,3)], 0.1)  # dispersions
initwalkers[:,-1] = numpy.clip(initwalkers[:,-1], 0.1, 0.9)  # fraction of 1st comp
```

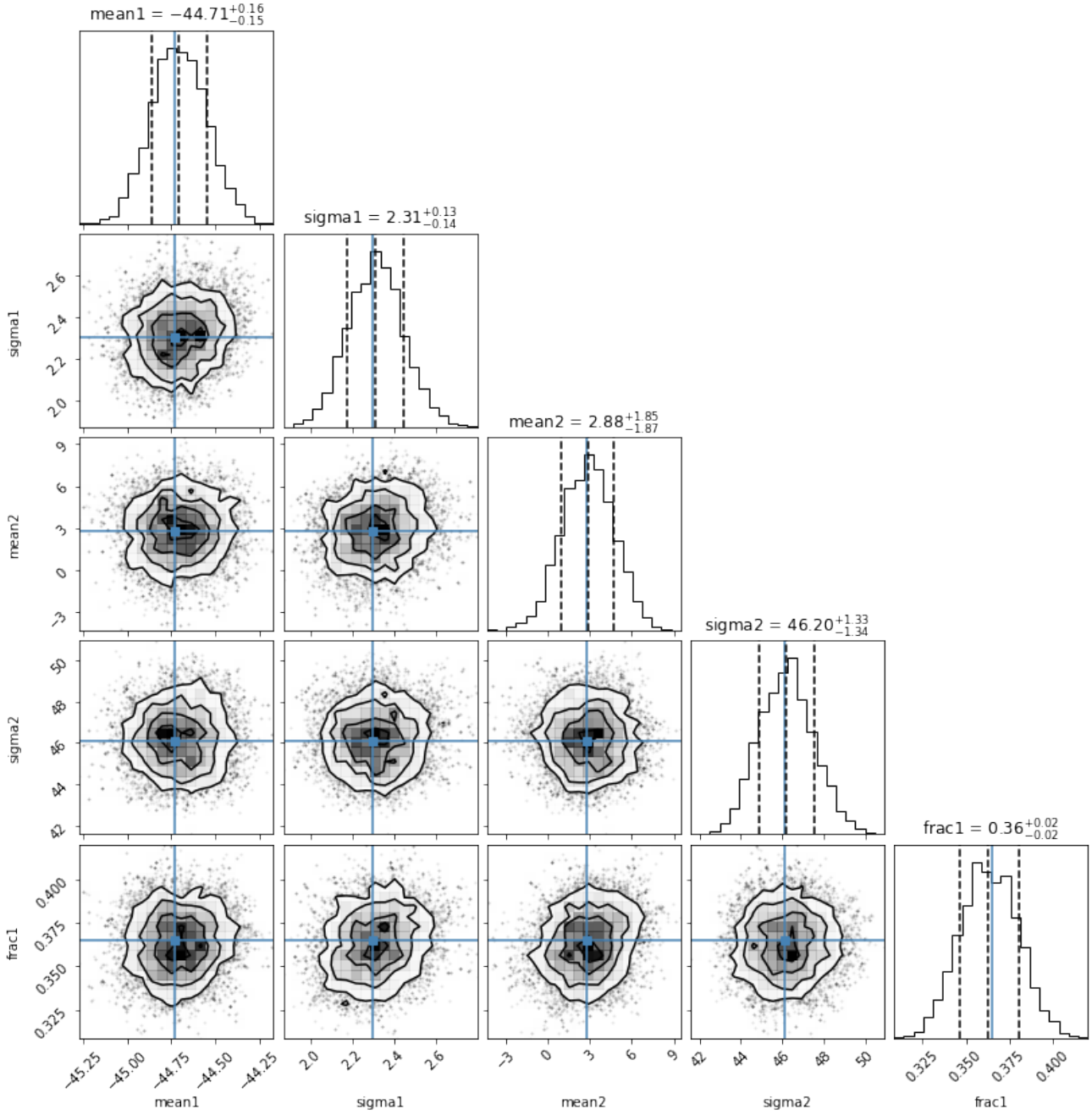
In [14]:
```
nsteps = 1000  # number of MCMC steps
sampler = emcee.EnsembleSampler(len(initwalkers), len(initparams), loglikelihood)
sampler.run_mcmc(initwalkers, nsteps)
chain = sampler.chain
loglike = sampler.lnprobability
```

In [15]:
```
# inspect the evolution of parameters as the simulation progresses:
# they may experience some weird oscillations before arriving into
# the region of high likelihood, where they stay indefinitely
paramnames = ['mean1', 'sigma1', 'mean2', 'sigma2', 'frac1']
axes = plt.subplots(len(initparams)+1, 1, sharex=True, figsize=(15,10))[1]
for i in range(len(initparams)):
    axes[i].plot(sampler.chain[:,::10,i].T, color='k', alpha=0.5)
    axes[i].set_xticklabels([])
    axes[i].set_ylabel(paramnames[i])
# last panel shows the evolution of the log-likelihood
axes[-1].plot(sampler.lnprobability[:,::10].T, color='k', alpha=0.5)
maxlp = numpy.amax(sampler.lnprobability)
axes[-1].set_ylim(maxlp-10, maxlp)
axes[-1].set_ylabel('loglike')
plt.tight_layout(h_pad=0)
plt.subplots_adjust(hspace=0,wspace=0)
```



In [16]:
```
# based on the above plot, we decide to discard the first 25% of the chain
# when it has not converged yet - this decision could be automated!
nsteps_discard = nsteps//4
chain = sampler.chain[:,nsteps_discard:].reshape(-1, len(initparams))
corner.corner(chain, quantiles=[0.159, 0.5, 0.841], labels=paramnames,
    show_titles=True, truths=bestfitparams)
for i in range(len(initparams)):
    low,med,upp = numpy.percentile(chain[:,i], [15.9, 50, 84.1])
    print("%s: median %g, 1-sigma range [%g : %g]" % (paramnames[i], med, low, upp))
```

```
mean1: median -44.7095, 1-sigma range [-44.8632 : -44.5486]
sigma1: median 2.31177, 1-sigma range [2.17587 : 2.44354]
mean2: median 2.87633, 1-sigma range [0.999152 : 4.74188]
sigma2: median 46.2017, 1-sigma range [44.8532 : 47.5343]
frac1: median 0.363132, 1-sigma range [0.346375 : 0.38021]
```


mean1 = −44.71$^{+0.16}_{-0.15}$
sigma1 = 2.31$^{+0.13}_{-0.14}$
mean2 = 2.88$^{+1.85}_{-1.87}$
sigma2 = 46.20$^{+1.33}_{-1.34}$
frac1 = 0.36$^{+0.02}_{-0.02}$
mean1
sigma1
mean2
sigma2
frac1