1. Gibbs Sampling with a Gaussian Mixture Model

May 2, 2017

Use conjugate priors in a Gaussian mixture model for clustering. Put a Dirichlet prior on the cluster probabilities, a multivariate Normal on the cluster means, and independent Gamma distributions on the variances of each feature.

1 a) Implement a Gibbs sampler to generate samples from the posterior distribution of cluster memberships.

First, I will use K-means to initialize the clusters intelligently.

```
In [1]: import numpy
        import math
        import random

        def Kmeans(data, K):
            """ Initializes the clusters intelligently using K-means. Copied from homework #4. '

            numpy.random.seed(1)
            # randomly generate K means
            means = []
            for k in range(K):
                meank = []
                # for each attribute, generate a mean between the min and max value of that attr
                for col in range(data.shape[1]):
                    meank.append(random.uniform(numpy.amin(data[:, col]), numpy.amax(data[:, col
                # add this mean to the list
                means.append(meank)
            means = numpy.array(means)

            # store whether the cluster assignments have been changed in a given iteration
            changed = True

            # while the cluster assignments are still changing, assign values to their nearest c
            while changed:
                # set changed to False for this round
                changed = False
```

```
            # calculate the Euclidian distance from each instance to each cluster mean
            dists = numpy.empty((data.shape[0], K))
            for k in range(K):
                dists[:, k] = numpy.apply_along_axis(lambda x: math.sqrt(numpy.sum((x-means[

            # find the nearest cluster center for each instance
            clusters = numpy.argmin(dists, axis = 1)

            # calculate the new mean of each cluster
            for k in range(K):
                clusterk = 1*(clusters == k)
                # if this cluster doesn't have any instances in it, try again
                if numpy.sum(clusterk) == 0:
                    return Kmeans(data, K)
                # loop through each attribute
                for col in range(data.shape[1]):
                    meanka = numpy.sum(data[:, col]*clusterk)/numpy.sum(clusterk)
                    # if the new mean of this cluster is different from the old mean, turn c
                    if meanka != means[k, col]:
                        changed = True
                    # add the new mean to the array
                    means[k, col] = meanka

        # once the cluster assignments stop changing, return the assignments and the means
        return clusters, means
```

Now I can use clusters generated by K-means to start Gibbs sampling.

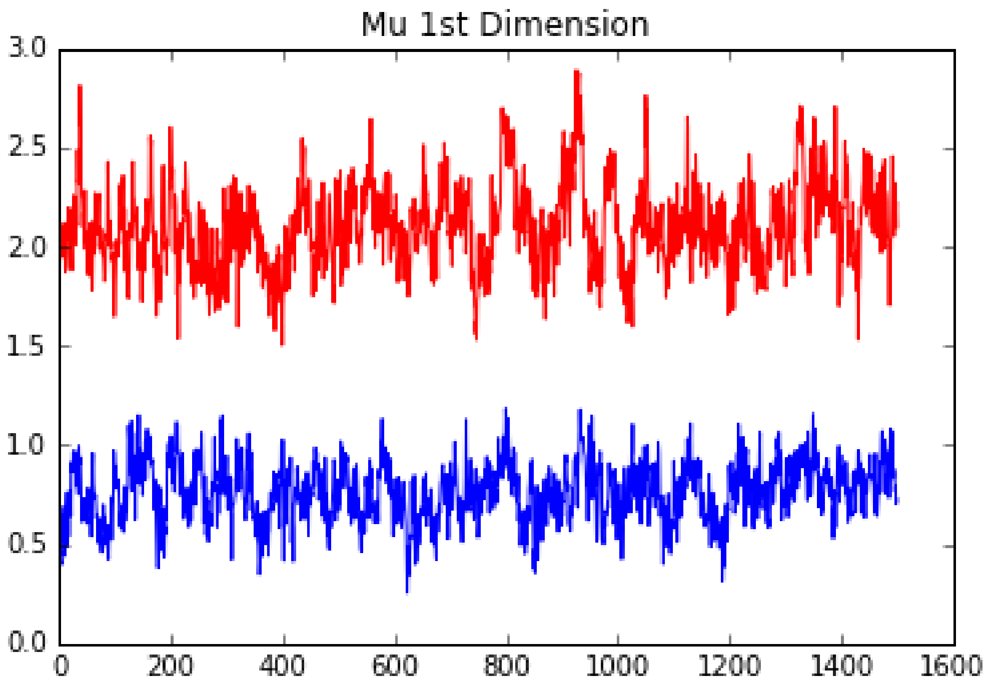
```
In [2]: import scipy.stats
        import matplotlib.pyplot as plt

        def GibbsSample(data, K, iters, prior_alpha, prior_mean, prior_variance, prior_a, prior_
            """ Iteratively samples cluster assignments and parameter values to approximate samp

            numpy.random.seed(1)

            # run K-means on the data
            clusters, means = Kmeans(data, K)

            # an empty array of cluster assignments to hold the results of each iteration
            Allclusters = numpy.empty((iters+1, data.shape[0]))
            # and put the initial clusters from K-means in it
            Allclusters[0, :] = clusters
            # an empty array of cluster proportions
            pis = numpy.empty((iters, K))
            # cluster means
            mus = numpy.empty((iters, K, data.shape[1]))
            # and cluster variances
```

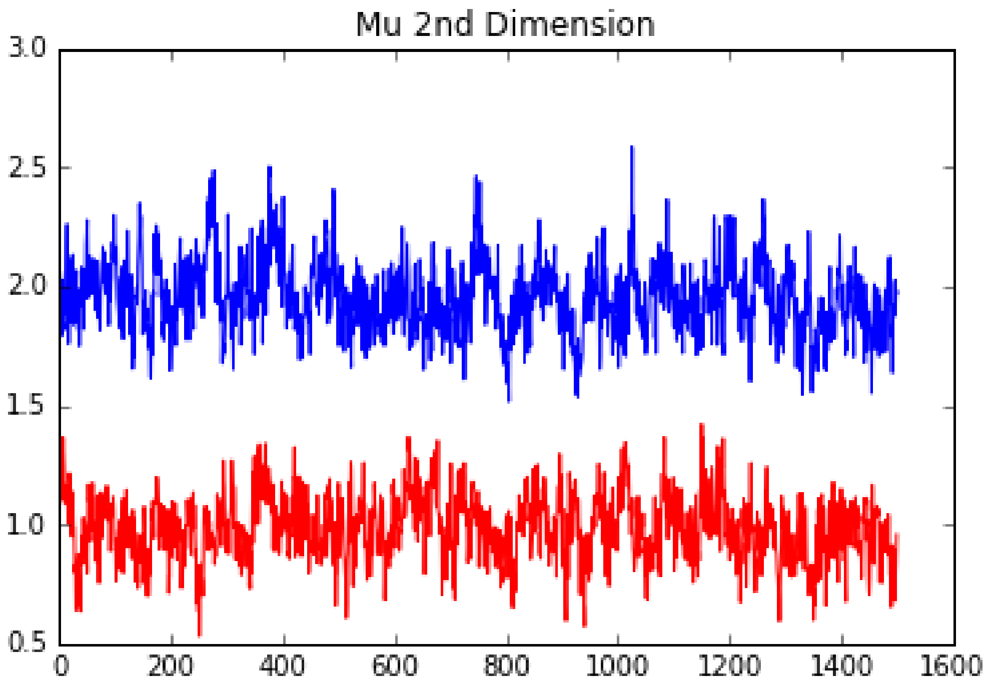
```
sigmas = numpy.empty((iters, K, data.shape[1]))
# also save the log likelihood
likelihood = numpy.zeros(iters)

# now iteratively sample for the designated number of iterations
for i in range(iters):
    # create a list of alpha parameters on the Dirichlet so those can be handled tog
    Dirichlet_alpha = numpy.empty(K)

    # update the other parameters for each cluster separately
    for k in range(K):
        # the number of instances in this cluster
        nk = numpy.sum(1*(Allclusters[i, :] == k))
        # if a cluster is empty or only has one instance in it, sample from the pric
        if nk <= 5:
            mus[i, k, :] = numpy.random.multivariate_normal(prior_mean*numpy.ones(da
                                                             prior_variance*numpy.ide
            for d in range(data.shape[1]):
                sigmas[i, k, d] = numpy.random.gamma(prior_a, 1/prior_b)
        # otherwise update the parameters and sample from the posterior
        else:
            # the variance among instances in this cluster
            hk = numpy.linalg.inv(numpy.diag(numpy.var(data[Allclusters[i, :] == k,

            # update the prior parameters to posteriors based on the current cluster
            # first the parameters on mu
            Normal_variance = numpy.linalg.inv(numpy.identity(hk.shape[0])*(prior_va
            Normal_mean = numpy.dot(prior_variance*prior_mean + nk*numpy.dot(hk,
                                    numpy.mean(data[Allclusters[i, :] == k, :], axis
            # now I can draw a value of mu
            mus[i, k, :] = numpy.random.multivariate_normal(Normal_mean, Normal_vari

            # update the parameters on the variance
            Gamma_a = prior_a + nk/2
            Gamma_b = prior_b + numpy.sum(numpy.apply_along_axis(lambda d: (d - mus[
            # sample from the posterior on sigma
            for d in range(data.shape[1]):
                sigmas[i, k, d] = numpy.random.gamma(Gamma_a, 1/Gamma_b[d])

        # update the parameters on pi
        Dirichlet_alpha[k] = prior_alpha + nk

    # sample from the posterior on pi for all of the clusters at once
    pis[i, :] = numpy.random.dirichlet(Dirichlet_alpha)

    # use the new parameter values to update the cluster assignments
    for n in range(data.shape[0]):
        qn = numpy.empty(K)
```

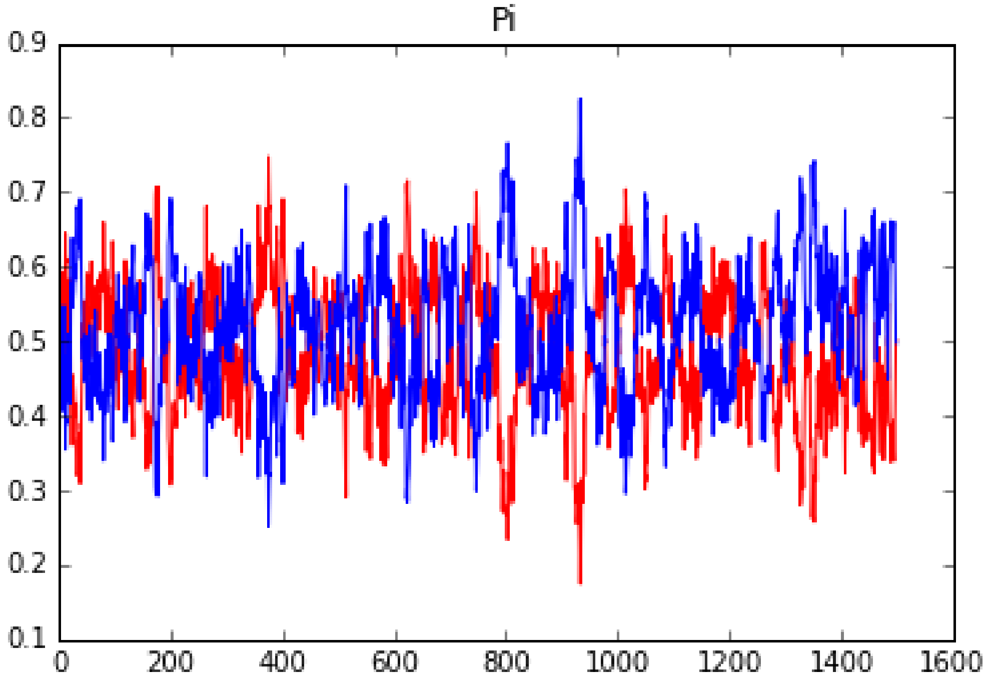
```
                for k in range(K):
                    qn[k] = pis[i, k]*scipy.stats.multivariate_normal.pdf(data[n, :], mean =
                qn = qn/numpy.sum(qn)
                z = int(numpy.random.choice(numpy.arange(K), p = qn))
                Allclusters[i+1, n] = z
                likelihood[i] += numpy.log(qn[z])

        return Allclusters, pis, mus, sigmas, likelihood
```

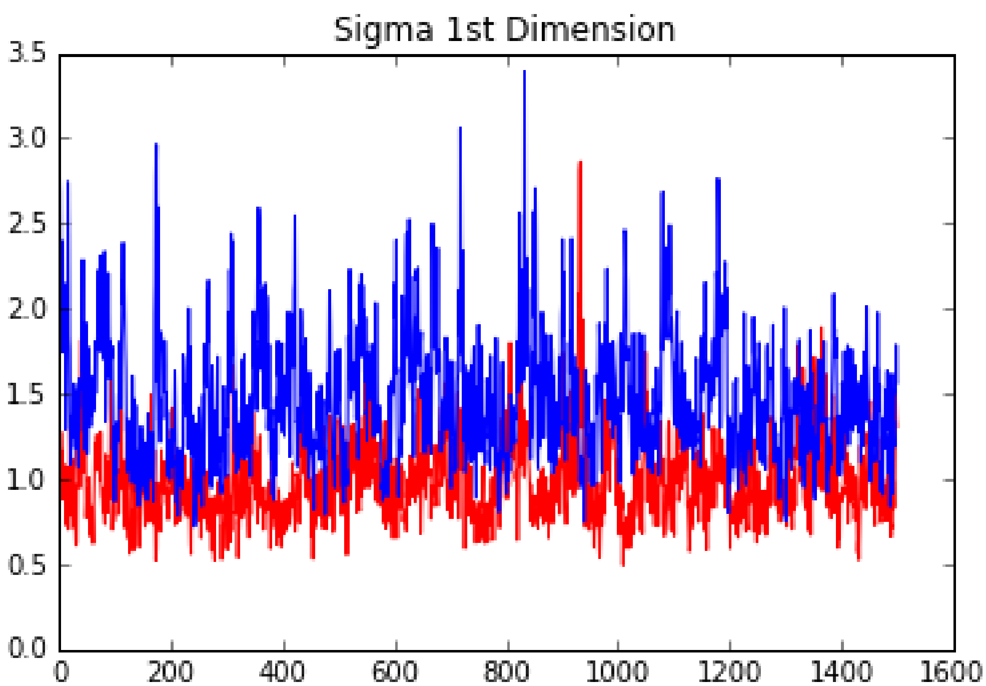
Now I'm going to load some of the data sets we used in homework 4 and cluster those.

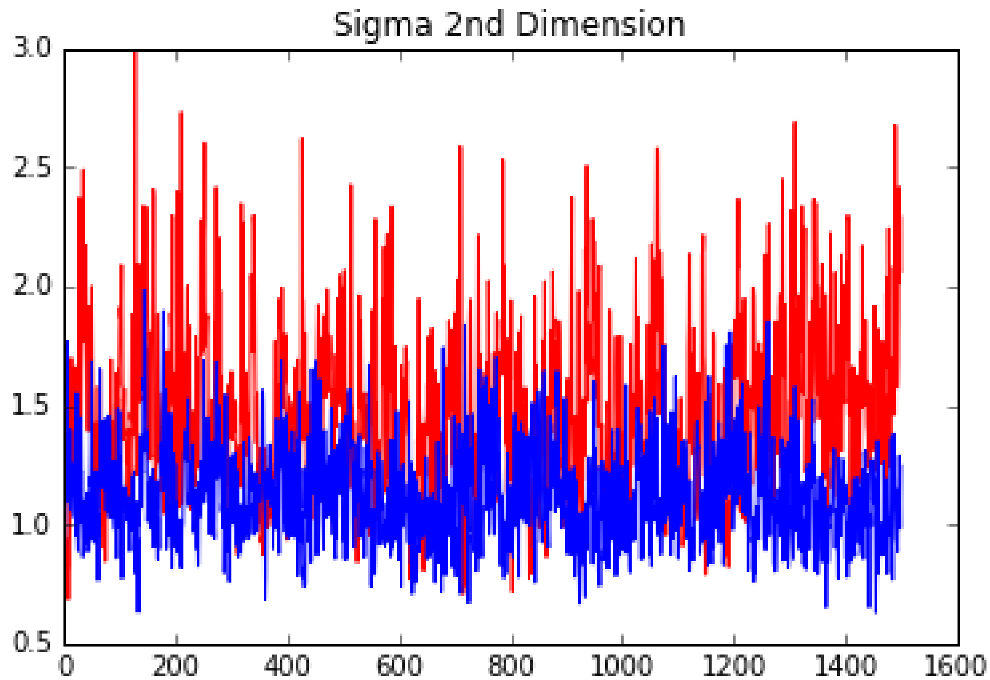
```
In [11]: import matplotlib.pyplot as plt

         S1data = numpy.loadtxt("S1train.csv", delimiter = ",", usecols = (1, 2))
         S1clusters, S1pi, S1mu, S1sigma, S1likelihood = GibbsSample(S1data, 2, 1500, 10, 1.5, .
         plt.plot(numpy.arange(1500), S1mu[:, 0, 0], "r", numpy.arange(1500), S1mu[:, 1, 0], "b"
         plt.title("Mu 1st Dimension")
         plt.show()
         plt.plot(numpy.arange(1500), S1mu[:, 0, 1], "r", numpy.arange(1500), S1mu[:, 1, 1], "b"
         plt.title("Mu 2nd Dimension")
         plt.show()
         plt.plot(numpy.arange(1500), S1pi[:, 0], "r", numpy.arange(1500), S1pi[:, 1], "b")
         plt.title("Pi")
         plt.show()
         plt.plot(numpy.arange(1500), S1sigma[:, 0, 0], "r", numpy.arange(1500), S1sigma[:, 1, 0
         plt.title("Sigma 1st Dimension")
         plt.show()
         plt.plot(numpy.arange(1500), S1sigma[:, 0, 1], "r", numpy.arange(1500), S1sigma[:, 1, 1
         plt.title("Sigma 2nd Dimension")
         plt.show()

         # plot the clusters
         plt.scatter(S1data[:, 0], S1data[:, 1], c = numpy.mean(S1clusters[-1000:, :], axis = 0)
         plt.title("Average Cluster Membership after Convergence")
         plt.colorbar()
         plt.show()
```

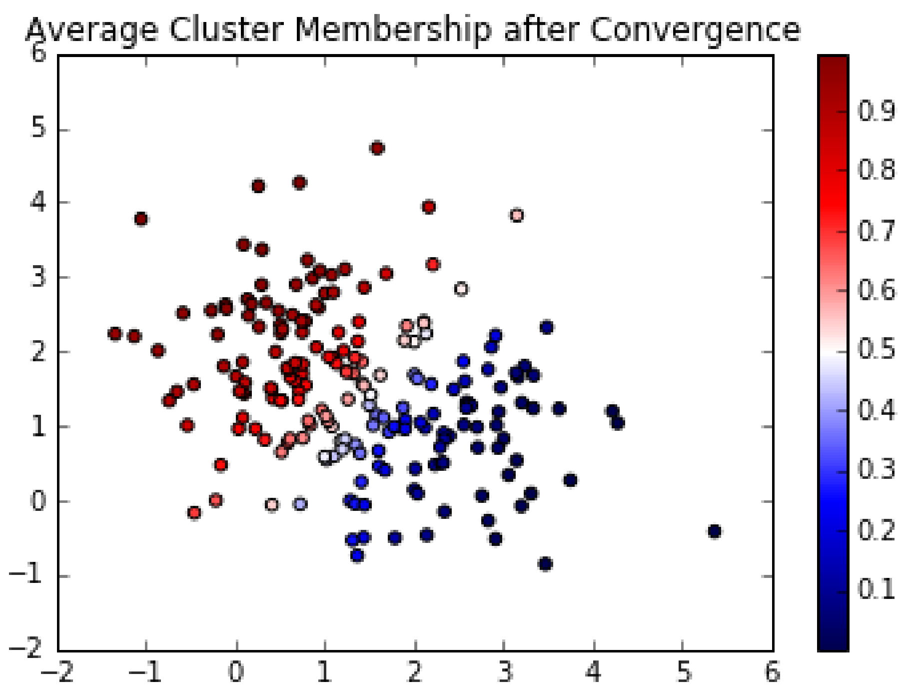

Mu 1st Dimension
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
200
400
600
800
1000
1200
1400
1600


Mu 2nd Dimension
3.0
2.5
2.0
1.5
1.0
0.5
0
200
400
600
800
1000
1200
1400
1600


Pi
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
200
400
600
800
1000
1200
1400
1600

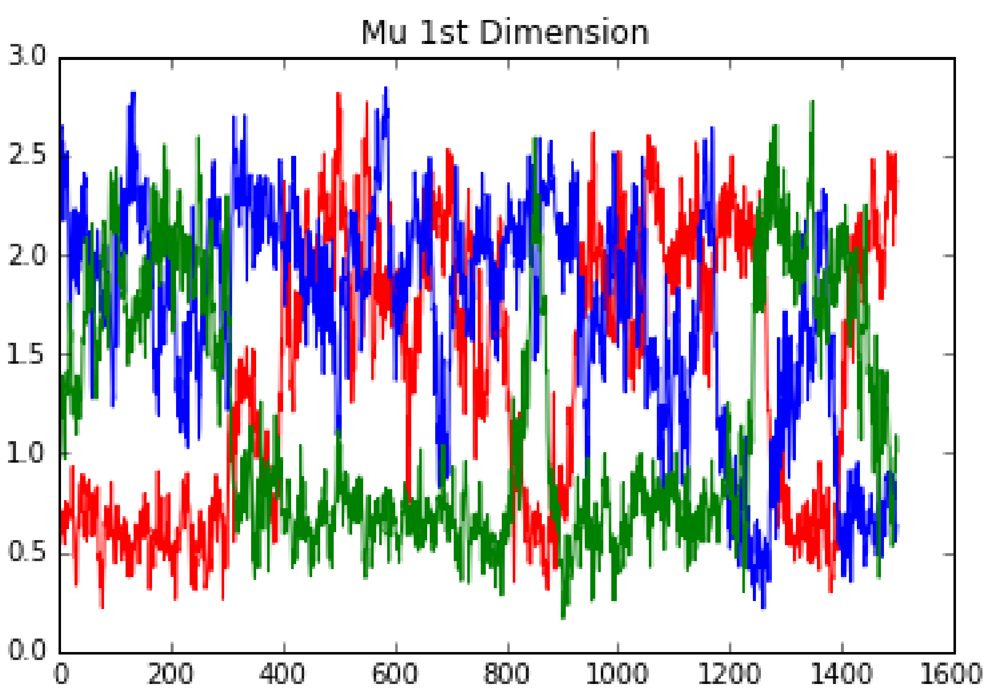

Sigma 1st Dimension
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
200
400
600
800
1000
1200
1400
1600

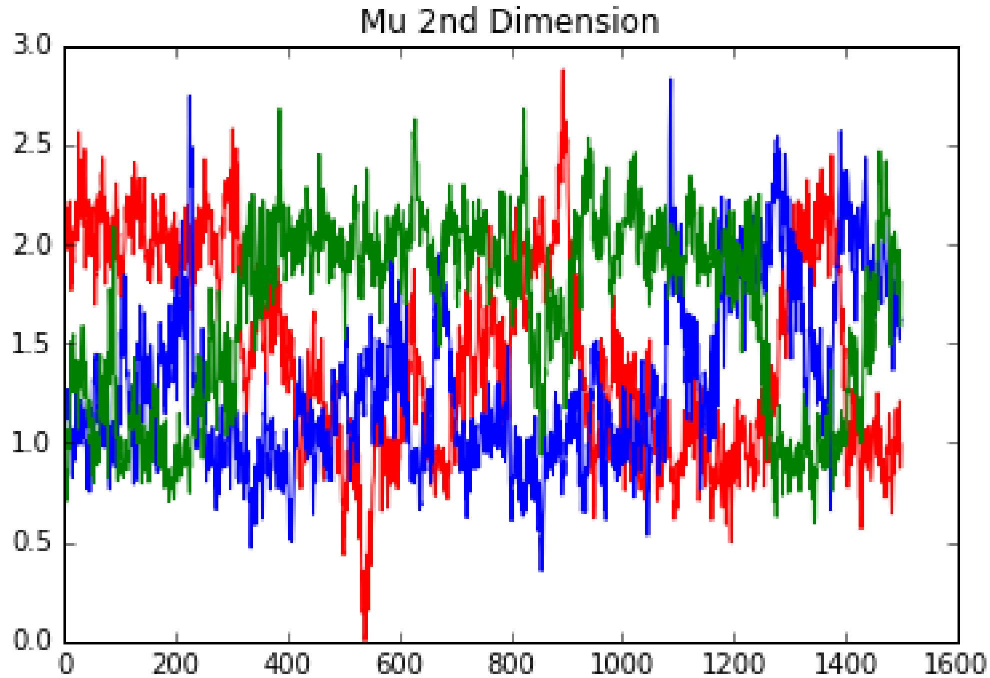


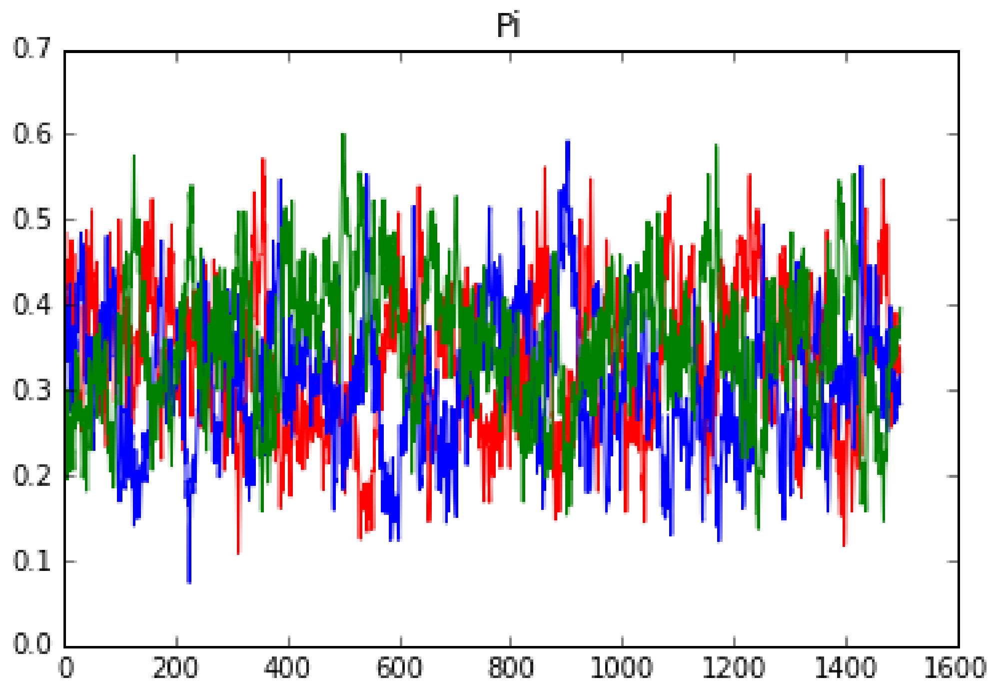


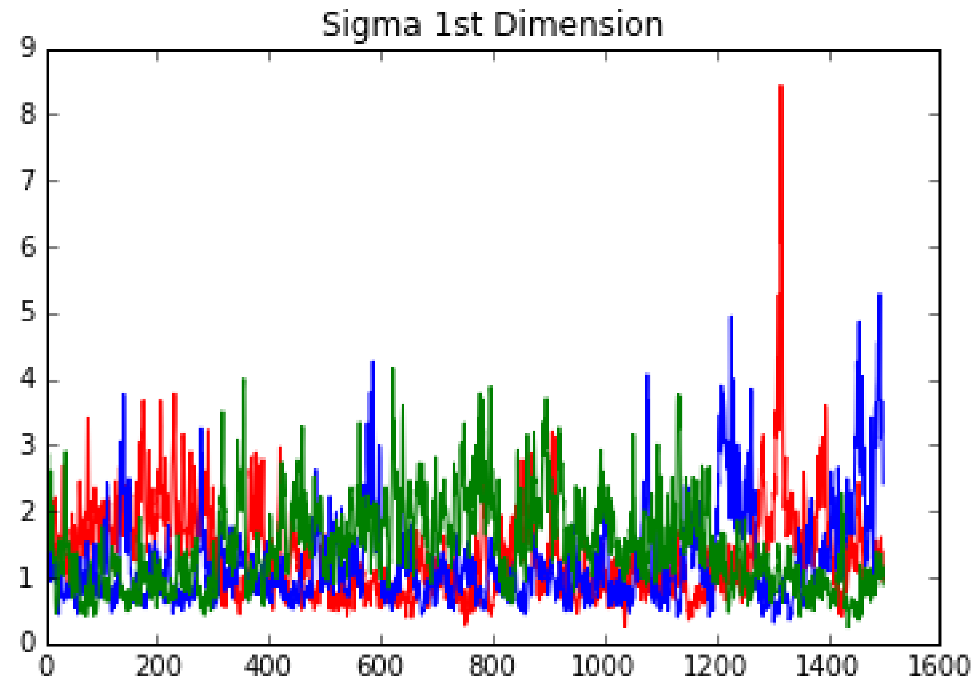

```
In [5]: # first synthetic data set with three clusters
        S1clusters3, S1pi3, S1mu3, S1sigma3, S1likelihood3 = GibbsSample(S1data, 3, 1500, 10, 1.
```

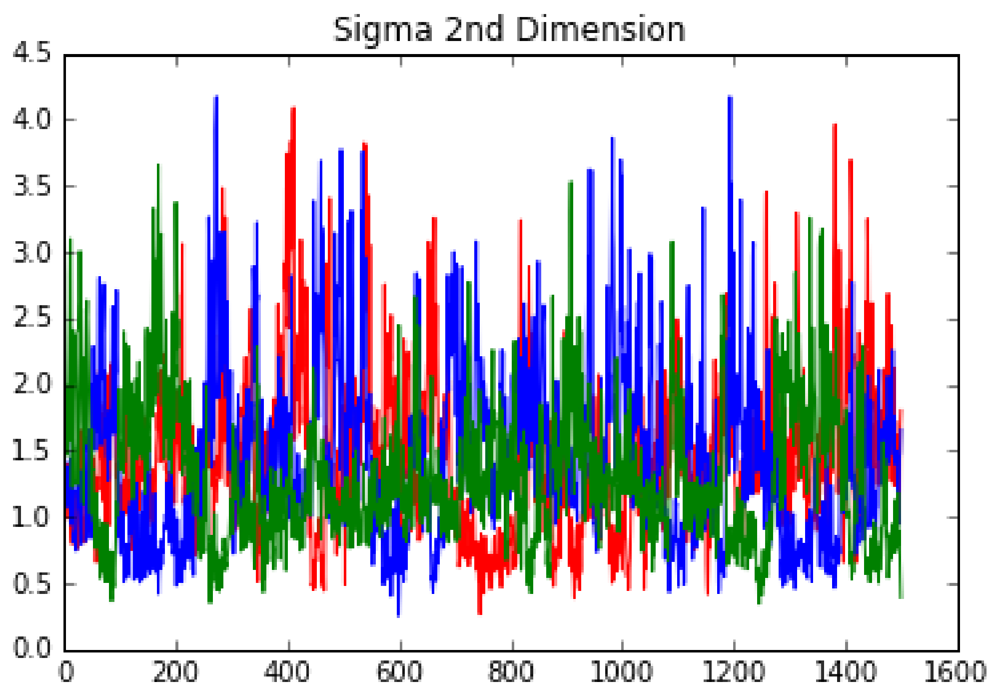
```
plt.plot(numpy.arange(1500), S1mu3[:, 0, 0], "r", numpy.arange(1500), S1mu3[:, 1, 0], "b
         numpy.arange(1500), S1mu3[:, 2, 0], "g")
plt.title("Mu 1st Dimension")
plt.show()
plt.plot(numpy.arange(1500), S1mu3[:, 0, 1], "r", numpy.arange(1500), S1mu3[:, 1, 1], "b
        numpy.arange(1500), S1mu3[:, 2, 1], "g")
plt.title("Mu 2nd Dimension")
plt.show()
plt.plot(numpy.arange(1500), S1pi3[:, 0], "r", numpy.arange(1500), S1pi3[:, 1], "b",
         numpy.arange(1500), S1pi3[:, 2], "g")
plt.title("Pi")
plt.show()
plt.plot(numpy.arange(1500), S1sigma3[:, 0, 0], "r", numpy.arange(1500), S1sigma3[:, 1,
         numpy.arange(1500), S1sigma3[:, 2, 0], "g")
plt.title("Sigma 1st Dimension")
plt.show()
plt.plot(numpy.arange(1500), S1sigma3[:, 0, 1], "r", numpy.arange(1500), S1sigma3[:, 1,
         numpy.arange(1500), S1sigma3[:, 2, 1], "g")
plt.title("Sigma 2nd Dimension")
plt.show()

# plot the clusters
plt.scatter(S1data[:, 0], S1data[:, 1], c = numpy.mean(S1clusters3[-1000:, :], axis = 0)
plt.title("Average Cluster Membership after Convergence")
plt.colorbar()
plt.show()
```

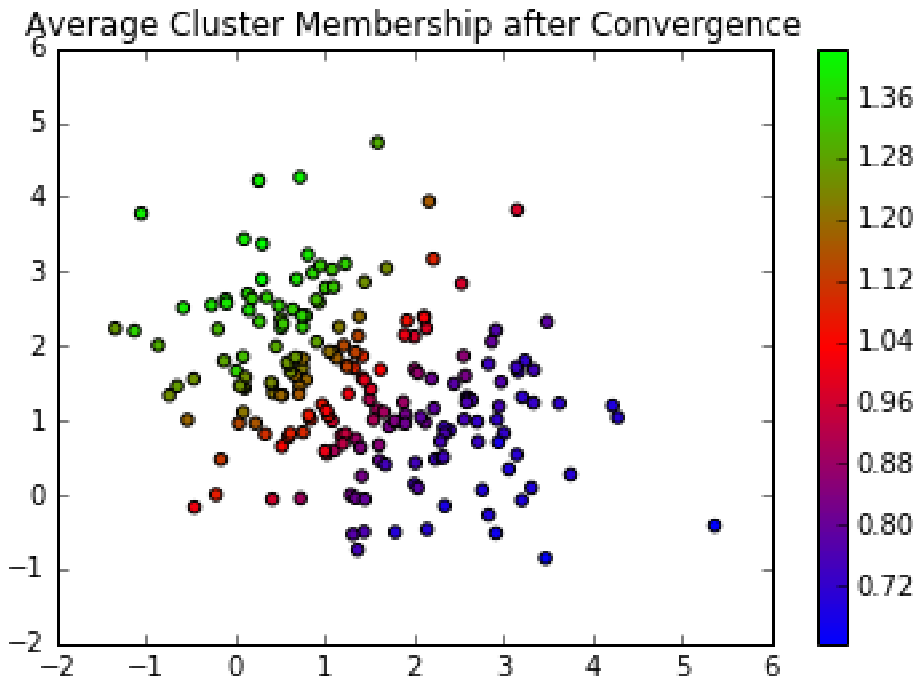



Mu 2nd Dimension
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
200
400
600
800
1000
1200
1400
1600


Pi
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0
200
400
600
800
1000
1200
1400
1600

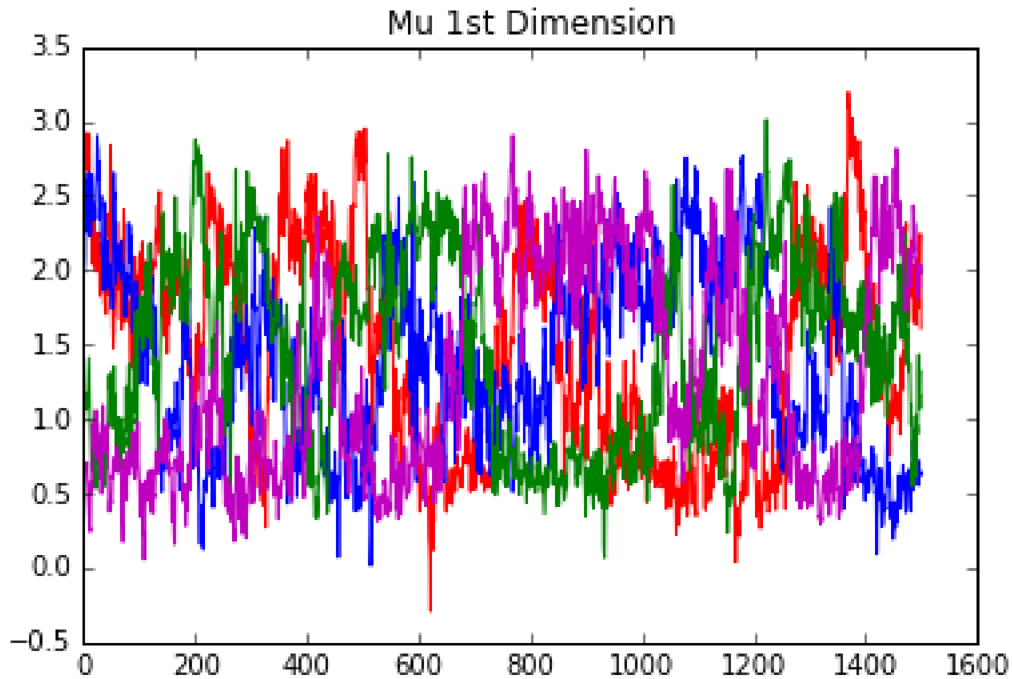

Sigma 1st Dimension
9
8
7
6
5
4
3
2
1
0
0
200
400
600
800
1000
1200
1400
1600

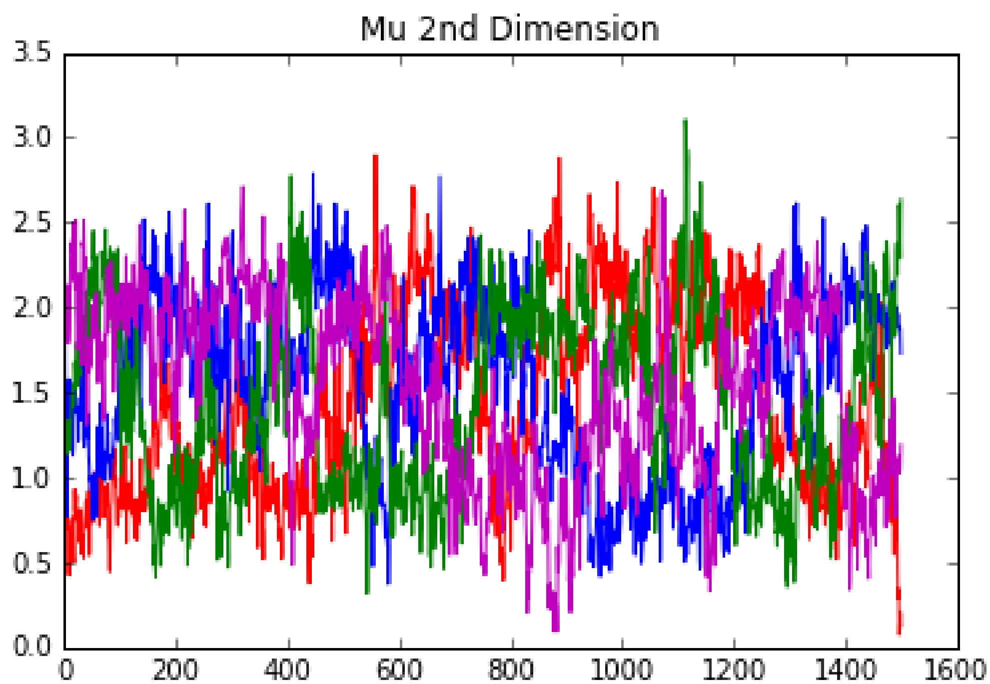

Sigma 2nd Dimension
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
200
400
600
800
1000
1200
1400
1600

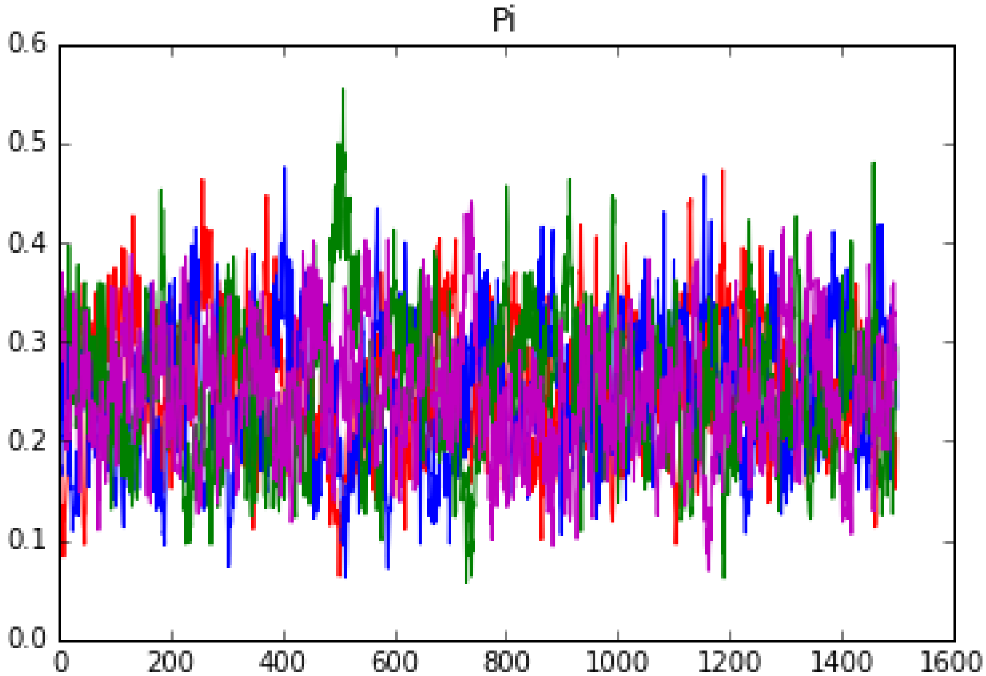


In [6]:
```
# first synthetic data set with four clusters
S1clusters4, S1pi4, S1mu4, S1sigma4, S1likelihood4 = GibbsSample(S1data, 4, 1500, 10, 1.
plt.plot(numpy.arange(1500), S1mu4[:, 0, 0], "r", numpy.arange(1500), S1mu4[:, 1, 0], "b
         numpy.arange(1500), S1mu4[:, 2, 0], "g", numpy.arange(1500), S1mu4[:, 3, 0], "m
plt.title("Mu 1st Dimension")
plt.show()
plt.plot(numpy.arange(1500), S1mu4[:, 0, 1], "r", numpy.arange(1500), S1mu4[:, 1, 1], "b
         numpy.arange(1500), S1mu4[:, 2, 1], "g", numpy.arange(1500), S1mu4[:, 3, 1], "m
plt.title("Mu 2nd Dimension")
plt.show()
plt.plot(numpy.arange(1500), S1pi4[:, 0], "r", numpy.arange(1500), S1pi4[:, 1], "b",
         numpy.arange(1500), S1pi4[:, 2], "g", numpy.arange(1500), S1pi4[:, 3], "m")
plt.title("Pi")
plt.show()
plt.plot(numpy.arange(1500), S1sigma4[:, 0, 0], "r", numpy.arange(1500), S1sigma4[:, 1,
         numpy.arange(1500), S1sigma4[:, 2, 0], "g", numpy.arange(1500), S1sigma4[:, 3,
plt.title("Sigma 1st Dimension")
plt.show()
plt.plot(numpy.arange(1500), S1sigma4[:, 0, 1], "r", numpy.arange(1500), S1sigma4[:, 1,
         numpy.arange(1500), S1sigma4[:, 2, 1], "g", numpy.arange(1500), S1sigma4[:, 3,
plt.title("Sigma 2nd Dimension")
plt.show()

# plot the clusters
```

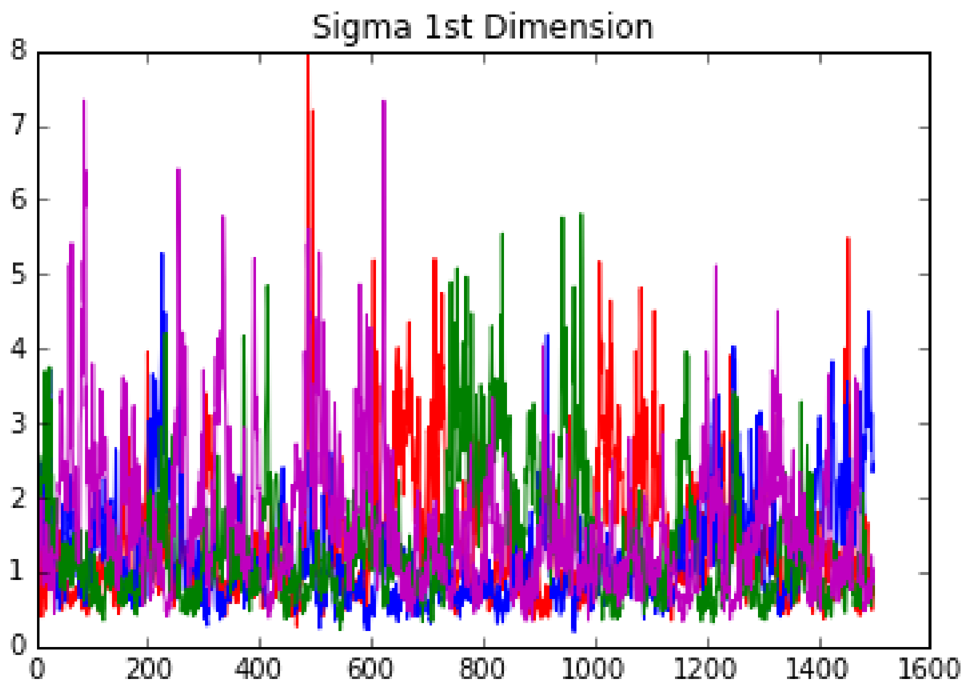
```
plt.scatter(S1data[:, 0], S1data[:, 1], c = numpy.mean(S1clusters4[-1000:, :], axis = 0)
plt.title("Average Cluster Membership after Convergence")
plt.colorbar()
plt.show()
```



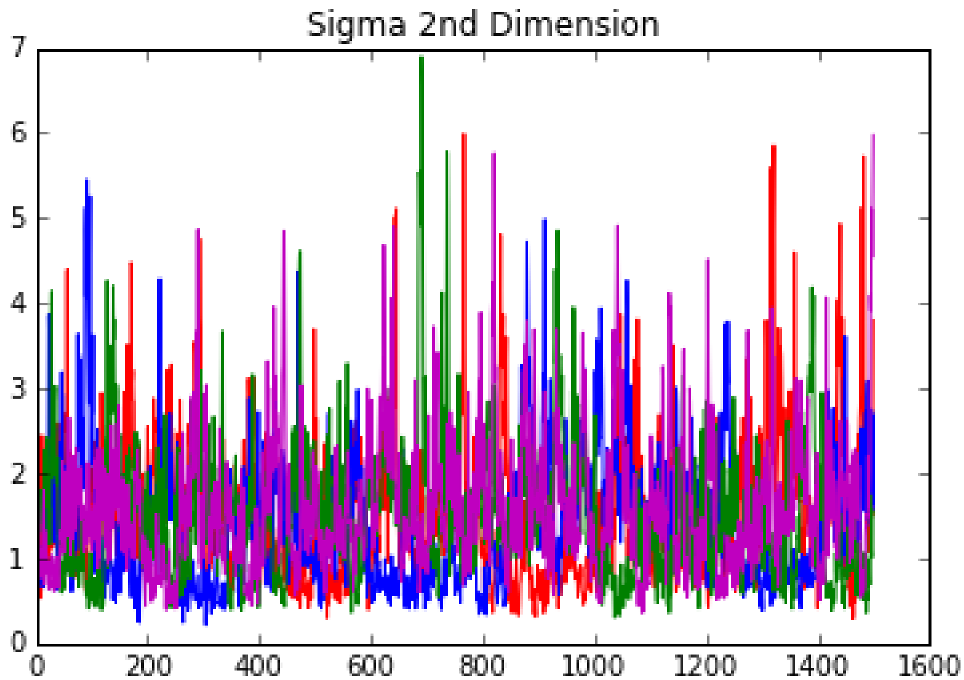


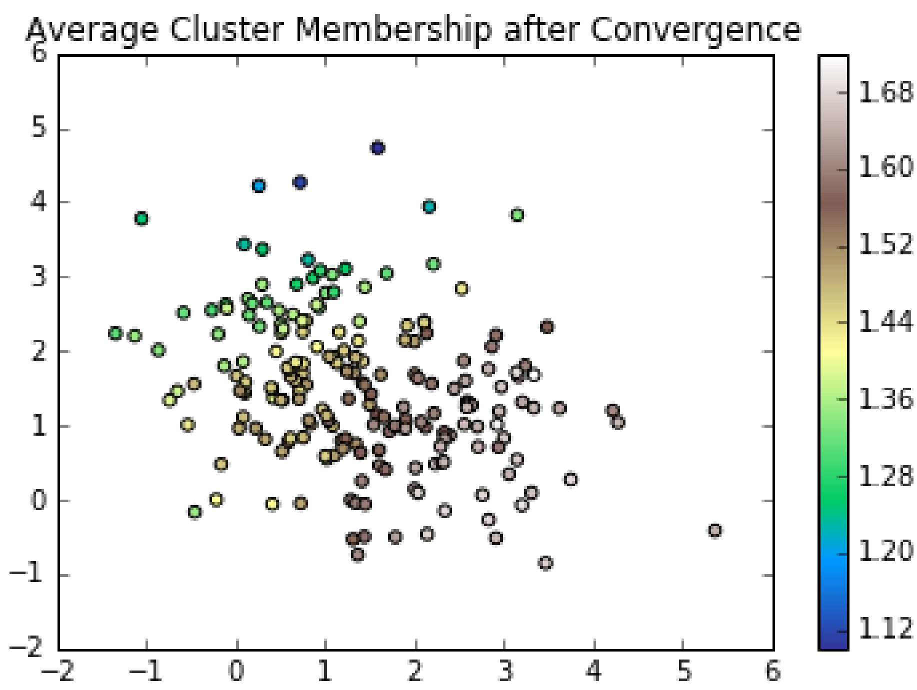


Pi
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0
200
400
600
800
1000
1200
1400
1600



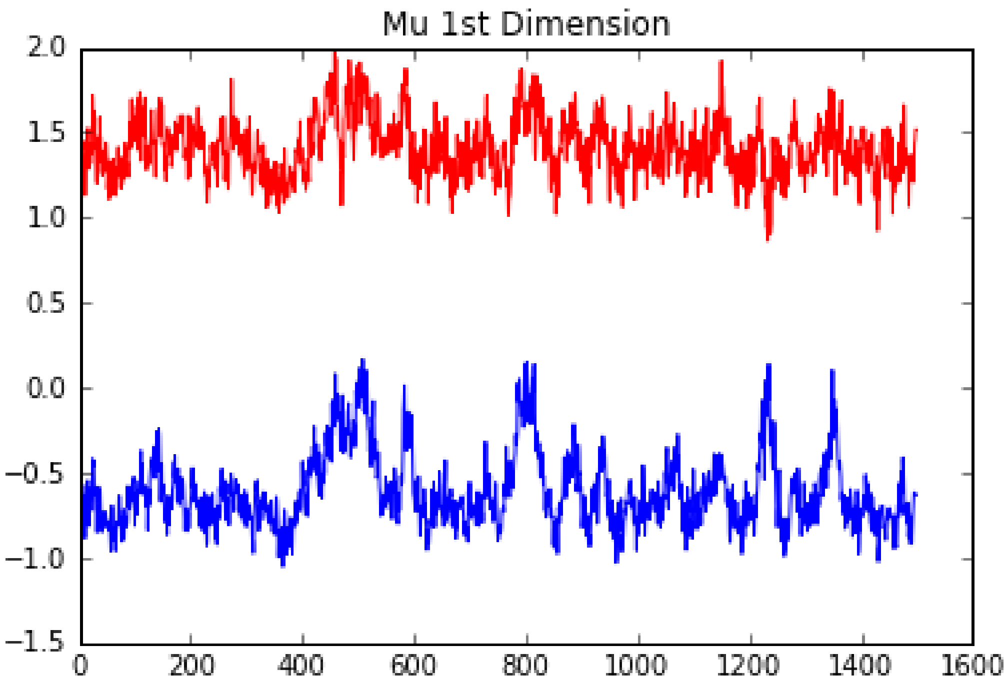
Sigma 1st Dimension
8
7
6
5
4
3
2
1
0
0
200
400
600
800
1000
1200
1400
1600

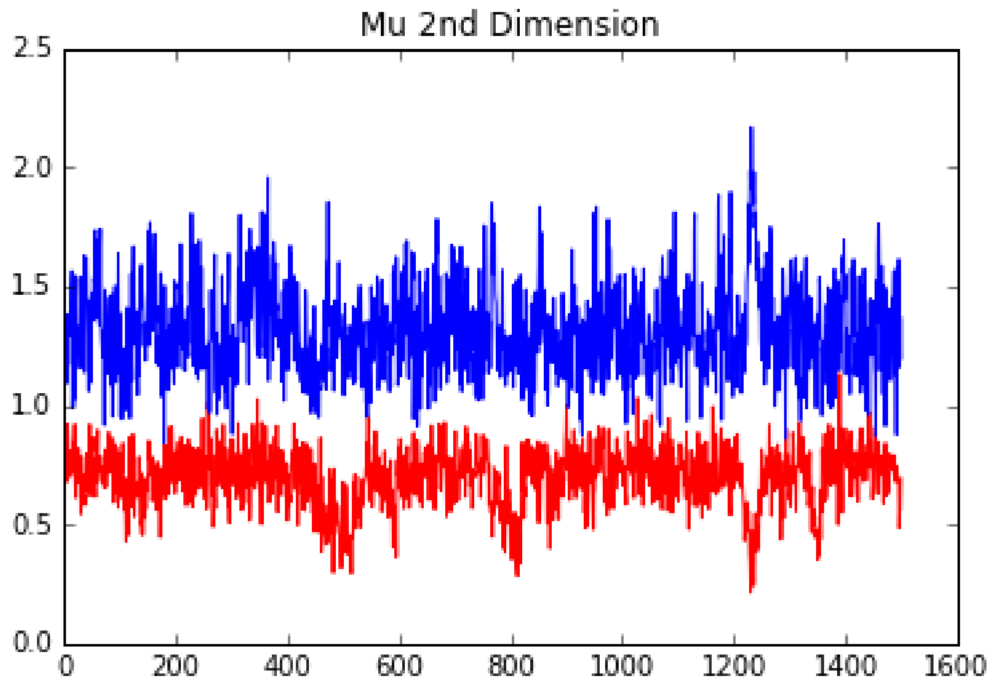


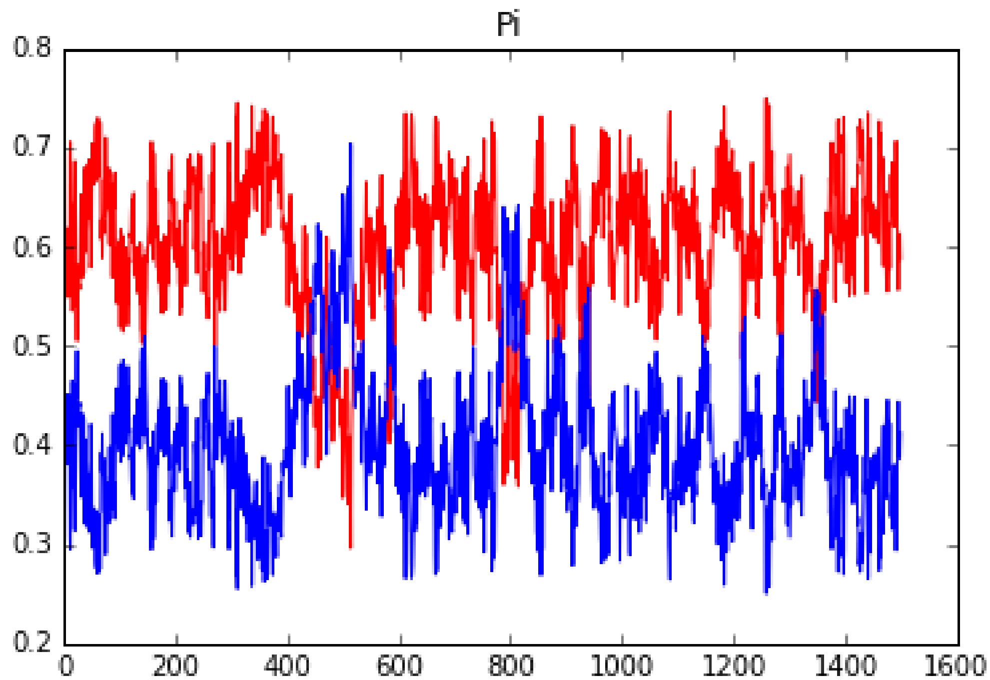


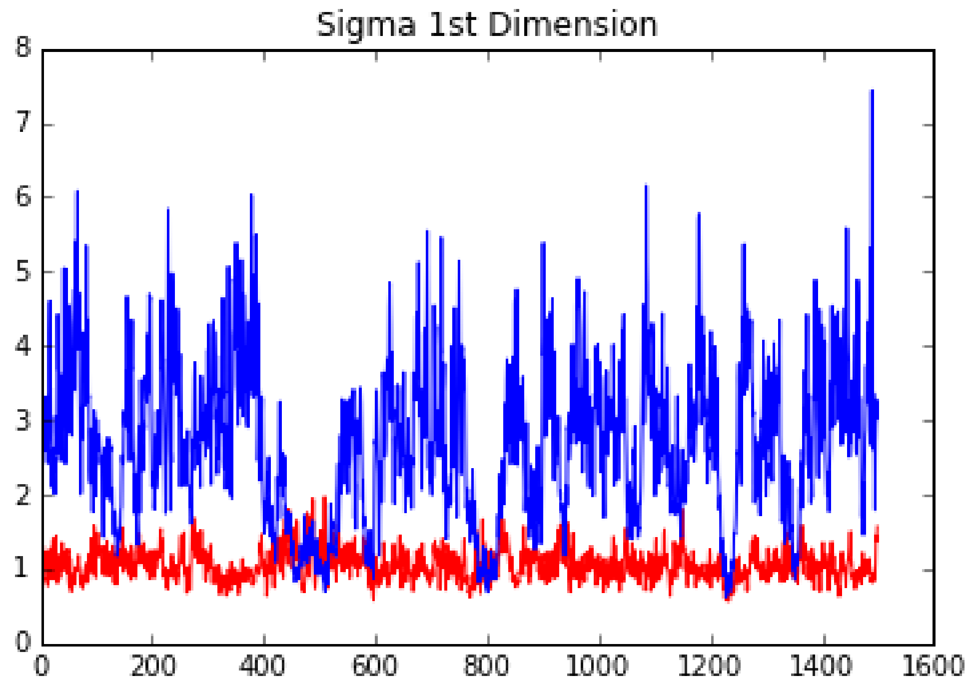

```
In [19]: # second synthetic data set
         S2data = numpy.loadtxt("S2train.csv", delimiter = ",", usecols = (1, 2))
```

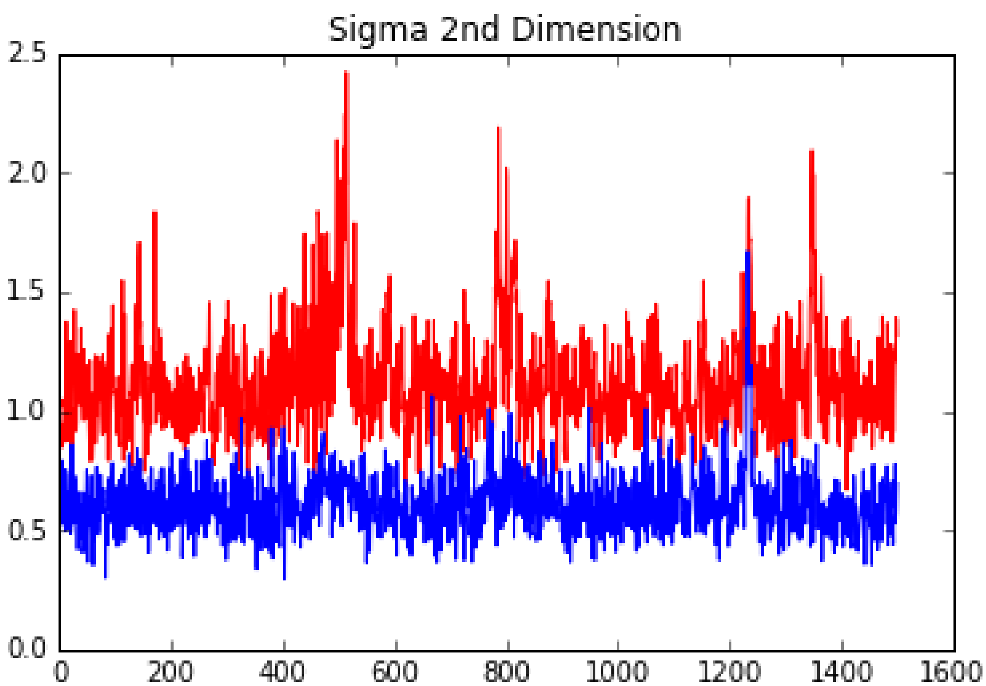
```
S2clusters, S2pi, S2mu, S2sigma, S2likelihood = GibbsSample(S2data, 2, 1500, 20, 1.5, .
plt.plot(numpy.arange(1500), S2mu[:, 0, 0], "r", numpy.arange(1500), S2mu[:, 1, 0], "b"
plt.title("Mu 1st Dimension")
plt.show()
plt.plot(numpy.arange(1500), S2mu[:, 0, 1], "r", numpy.arange(1500), S2mu[:, 1, 1], "b"
plt.title("Mu 2nd Dimension")
plt.show()
plt.plot(numpy.arange(1500), S2pi[:, 0], "r", numpy.arange(1500), S2pi[:, 1], "b")
plt.title("Pi")
plt.show()
plt.plot(numpy.arange(1500), S2sigma[:, 0, 0], "r", numpy.arange(1500), S2sigma[:, 1, 0
plt.title("Sigma 1st Dimension")
plt.show()
plt.plot(numpy.arange(1500), S2sigma[:, 0, 1], "r", numpy.arange(1500), S2sigma[:, 1, 1
plt.title("Sigma 2nd Dimension")
plt.show()

# plot the clusters
plt.scatter(S2data[:, 0], S2data[:, 1], c = numpy.mean(S2clusters[-1000:, :], axis = 0)
plt.title("Average Cluster Membership after Convergence")
plt.colorbar()
plt.show()
```

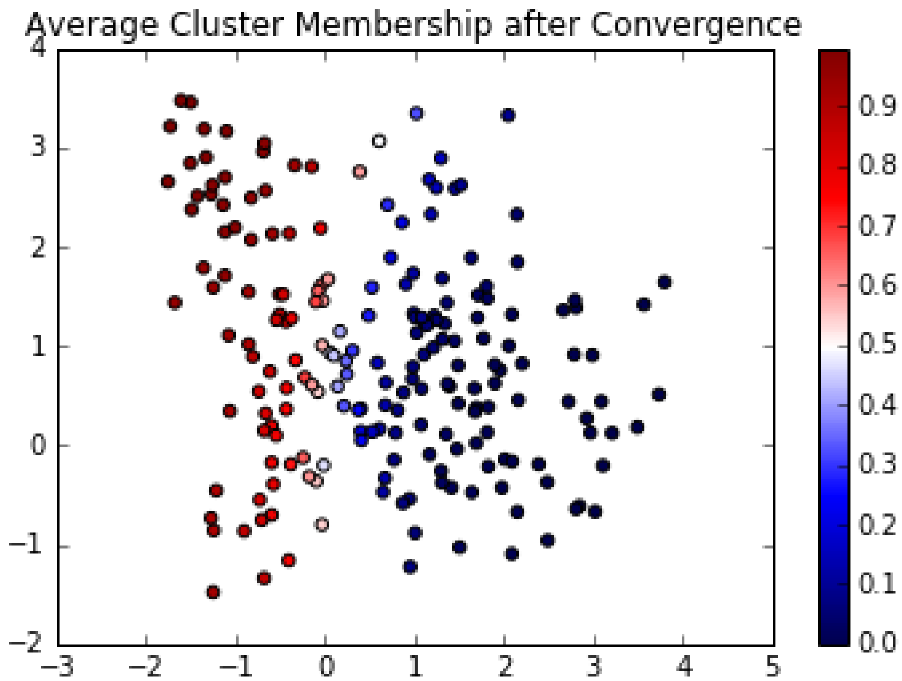



Mu 2nd Dimension
2.5
2.0
1.5
1.0
0.5
0.0
0
200
400
600
800
1000
1200
1400
1600

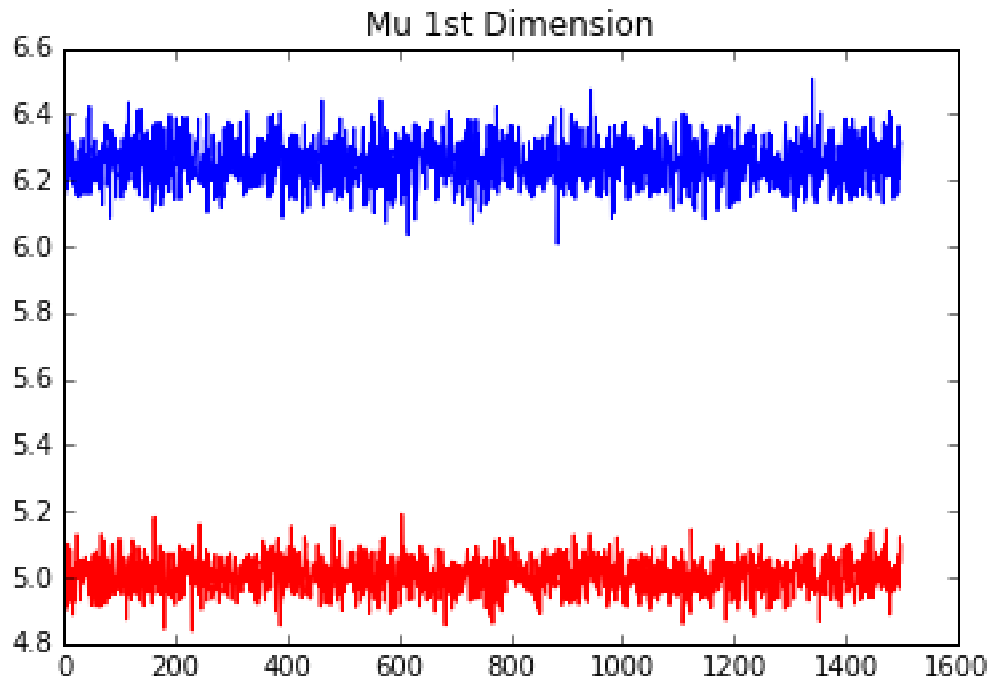

Pi
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0
200
400
600
800
1000
1200
1400
1600

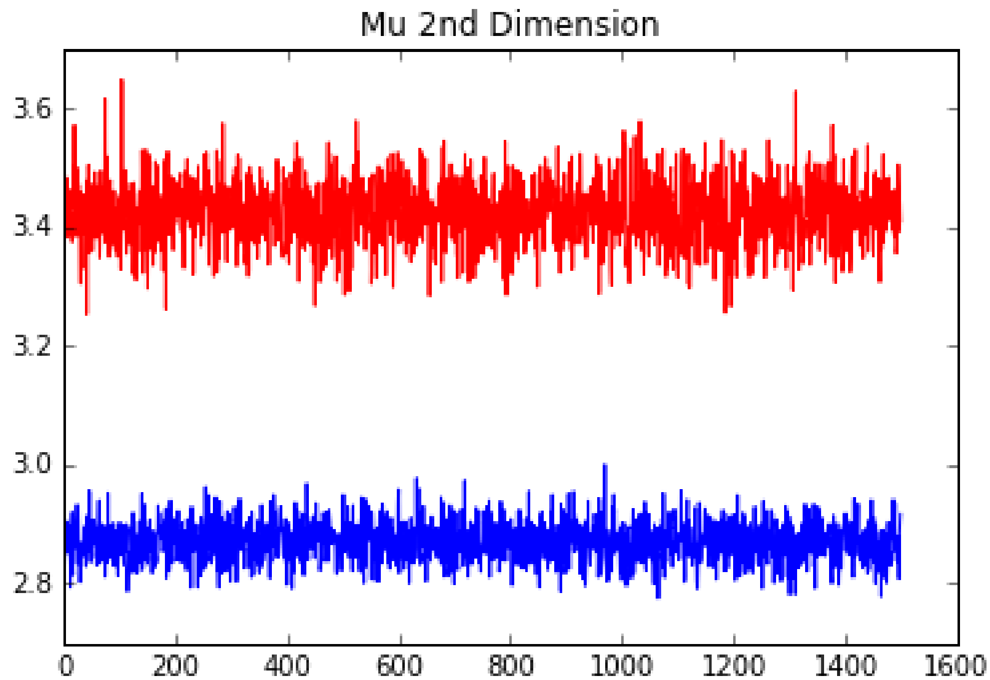

Sigma 1st Dimension
0
1
2
3
4
5
6
7
8
0
200
400
600
800
1000
1200
1400
1600

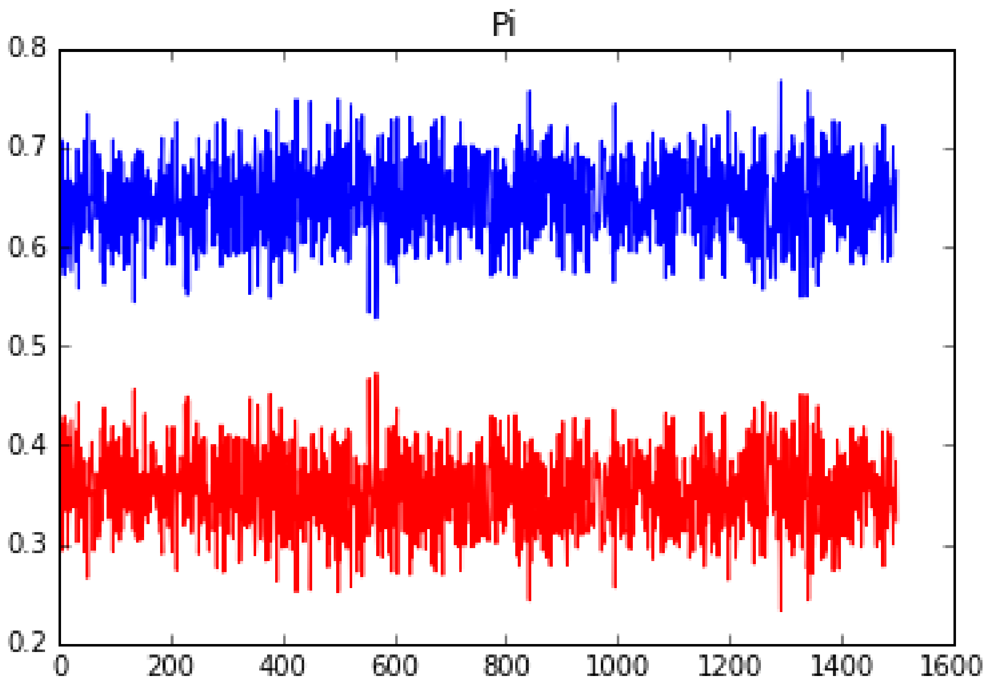

Sigma 2nd Dimension
0.0
0.5
1.0
1.5
2.0
2.5
0
200
400
600
800
1000
1200
1400
1600

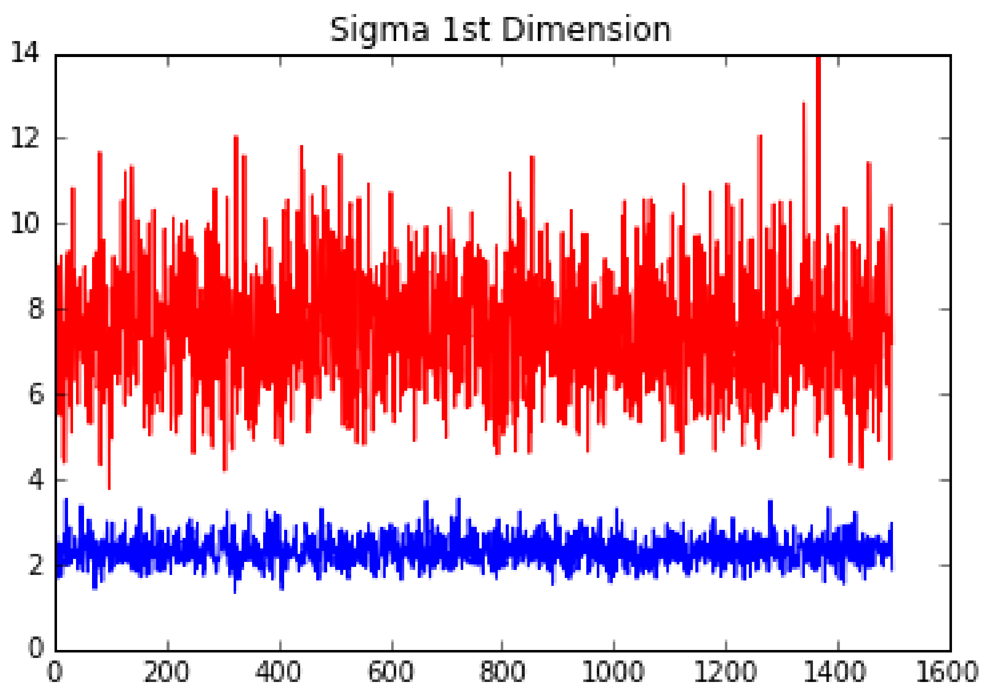


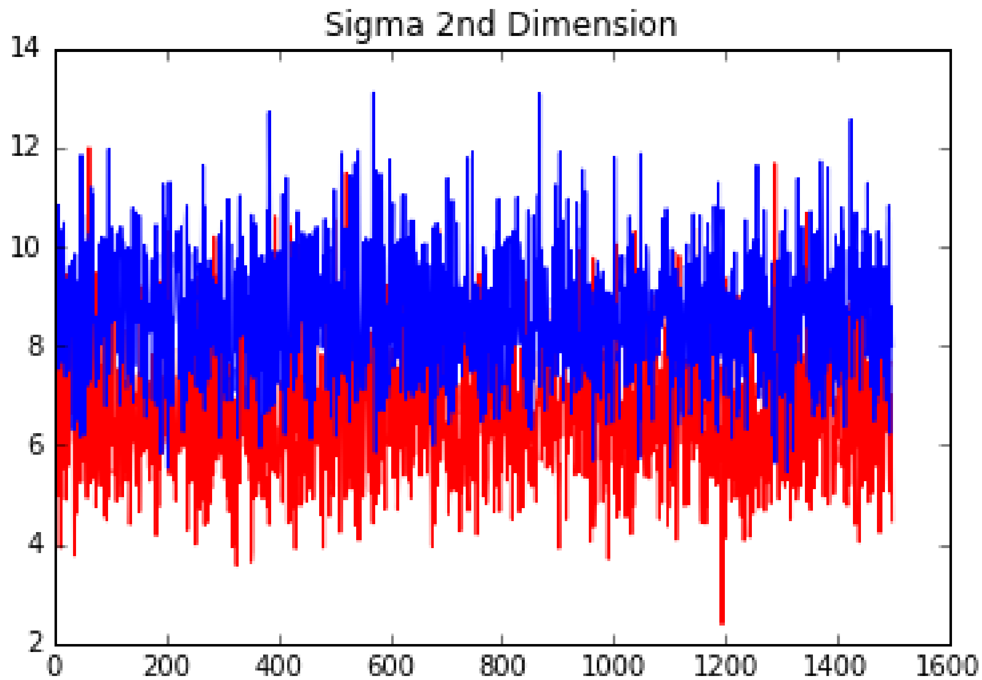

```
In [20]: # Iris data set with two clusters
         irisdata = numpy.loadtxt("iris_features.csv", delimiter = ",", skiprows = 1)
         irisclusters2, irispi2, irismu2, irissigma2, irislikelihood2 = GibbsSample(irisdata, 2,
         plt.plot(numpy.arange(1500), irismu2[:, 0, 0], "r", numpy.arange(1500), irismu2[:, 1, 0
         plt.title("Mu 1st Dimension")
         plt.show()
         plt.plot(numpy.arange(1500), irismu2[:, 0, 1], "r", numpy.arange(1500), irismu2[:, 1, 1
         plt.title("Mu 2nd Dimension")
         plt.show()
         plt.plot(numpy.arange(1500), irispi2[:, 0], "r", numpy.arange(1500), irispi2[:, 1], "b"
         plt.title("Pi")
         plt.show()
         plt.plot(numpy.arange(1500), irissigma2[:, 0, 0], "r", numpy.arange(1500), irissigma2[:
         plt.title("Sigma 1st Dimension")
         plt.show()
         plt.plot(numpy.arange(1500), irissigma2[:, 0, 1], "r", numpy.arange(1500), irissigma2[:
         plt.title("Sigma 2nd Dimension")
         plt.show()

         # plot the clusters
         plt.scatter(irisdata[:, 0], irisdata[:, 1], c = numpy.mean(irisclusters2[-1000:, :], ax
         plt.title("Average Cluster Membership after Convergence")
         plt.colorbar()
         plt.show()
```

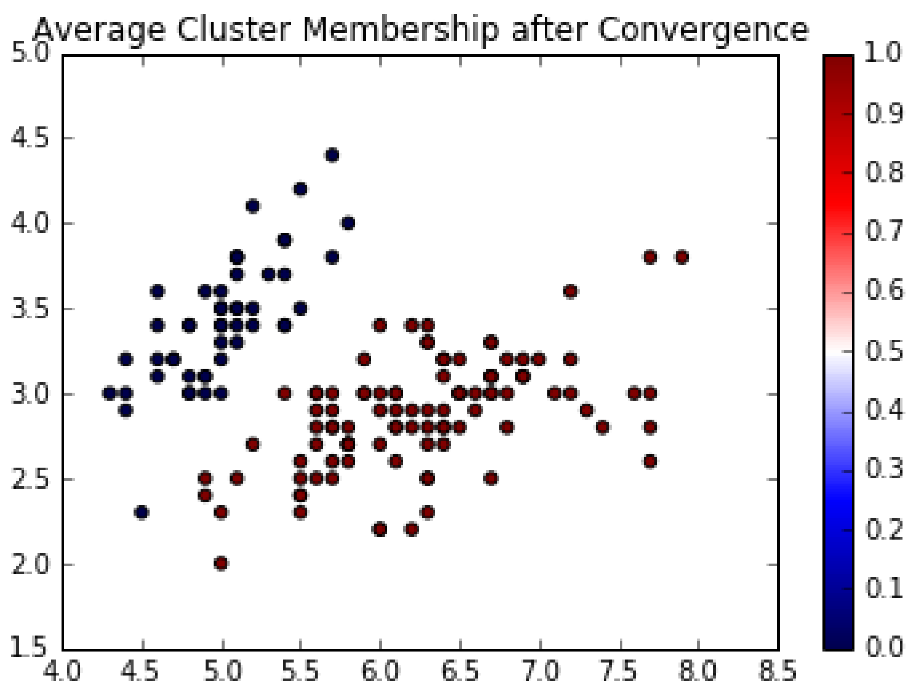

Mu 1st Dimension
6.6
6.4
6.2
6.0
5.8
5.6
5.4
5.2
5.0
4.8
0
200
400
600
800
1000
1200
1400
1600


Mu 2nd Dimension
3.6
3.4
3.2
3.0
2.8
0
200
400
600
800
1000
1200
1400
1600


Pi
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0
200
400
600
800
1000
1200
1400
1600

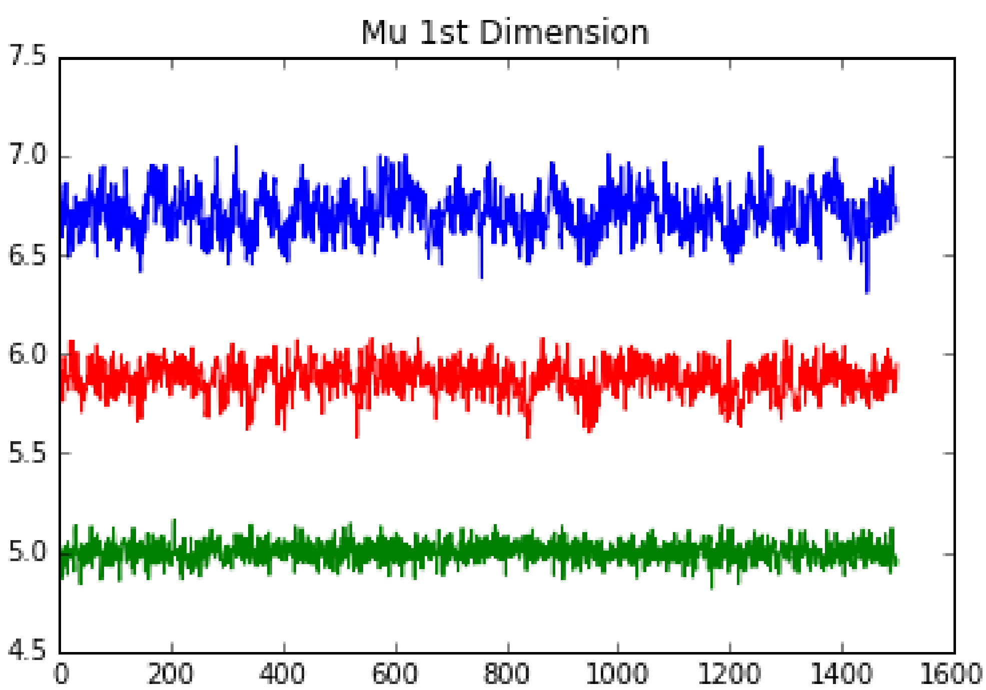

Sigma 1st Dimension
14
12
10
8
6
4
2
0
0
200
400
600
800
1000
1200
1400
1600

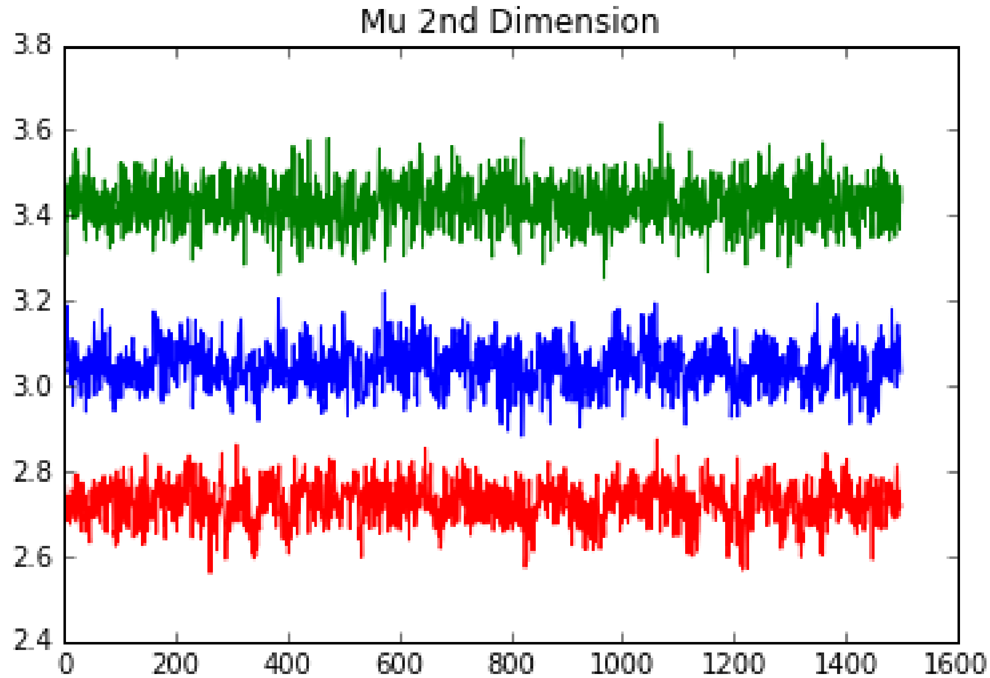


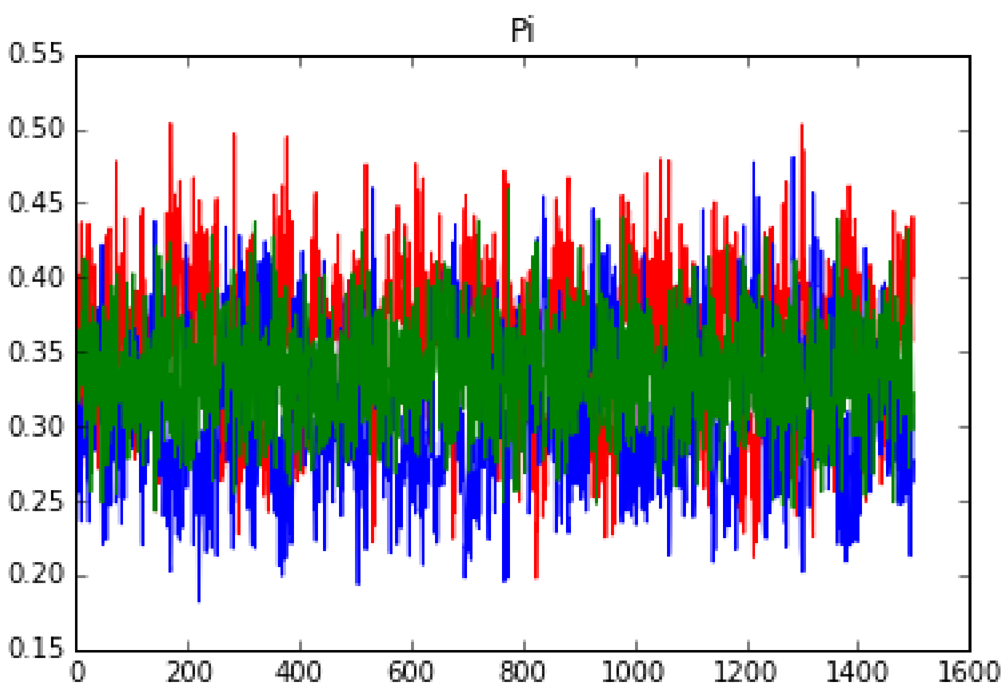


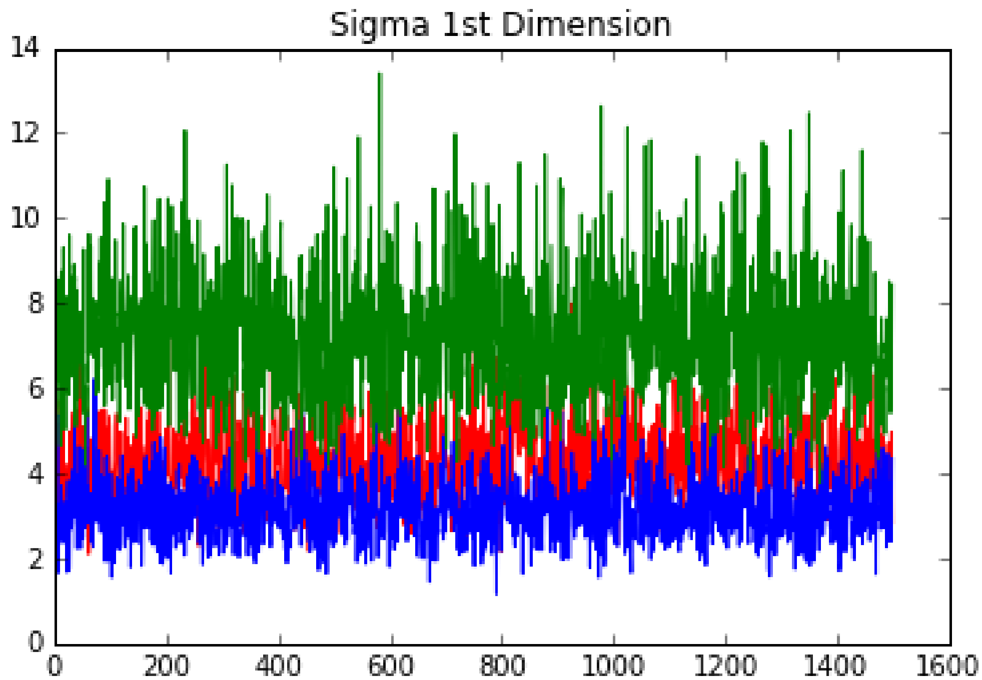
In [23]:
```
# Iris data set with three clusters
irisclusters3, irispi3, irismu3, irissigma3, irislikelihood3 = GibbsSample(irisdata, 3,
```

```
plt.plot(numpy.arange(1500), irismu3[:, 0, 0], "r", numpy.arange(1500), irismu3[:, 1, 0
        numpy.arange(1500), irismu3[:, 2, 0], "g")
plt.title("Mu 1st Dimension")
plt.show()
plt.plot(numpy.arange(1500), irismu3[:, 0, 1], "r", numpy.arange(1500), irismu3[:, 1, 1
        numpy.arange(1500), irismu3[:, 2, 1], "g")
plt.title("Mu 2nd Dimension")
plt.show()
plt.plot(numpy.arange(1500), irispi3[:, 0], "r", numpy.arange(1500), irispi3[:, 1], "b"
        numpy.arange(1500), irispi3[:, 2], "g")
plt.title("Pi")
plt.show()
plt.plot(numpy.arange(1500), irissigma3[:, 0, 0], "r", numpy.arange(1500), irissigma3[:
        numpy.arange(1500), irissigma3[:, 2, 0], "g")
plt.title("Sigma 1st Dimension")
plt.show()
plt.plot(numpy.arange(1500), irissigma3[:, 0, 1], "r", numpy.arange(1500), irissigma3[:
        numpy.arange(1500), irissigma3[:, 2, 1], "g")
plt.title("Sigma 2nd Dimension")
plt.show()

# plot the clusters
plt.scatter(irisdata[:, 0], irisdata[:, 1], c = numpy.mean(irisclusters3[-1000:, :], ax
plt.title("Average Cluster Membership after Convergence")
plt.colorbar()
plt.show()
```

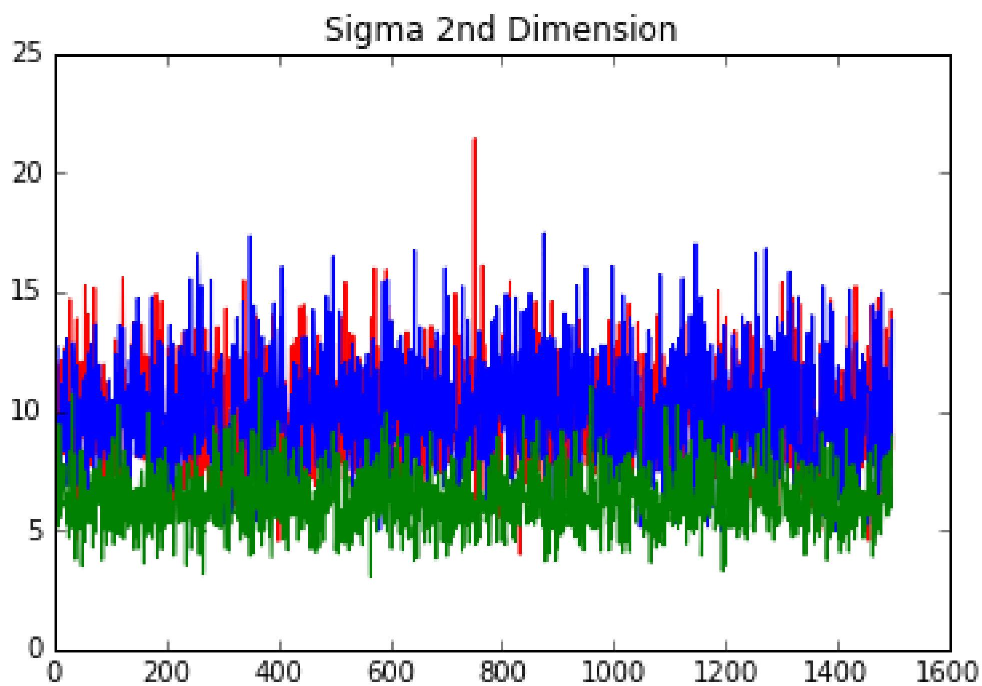



Mu 2nd Dimension
3.8
3.6
3.4
3.2
3.0
2.8
2.6
2.4
0
200
400
600
800
1000
1200
1400
1600

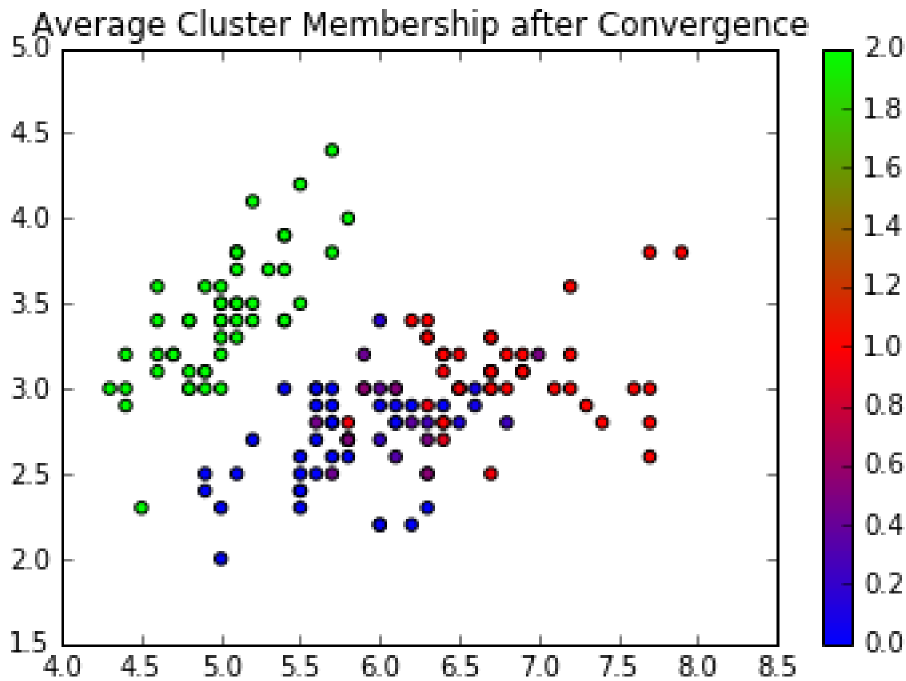

Pi
0.55
0.50
0.45
0.40
0.35
0.30
0.25
0.20
0.15
0
200
400
600
800
1000
1200
1400
1600



Sigma 1st Dimension
14
12
10
8
6
4
2
0
0
200
400
600
800
1000
1200
1400
1600

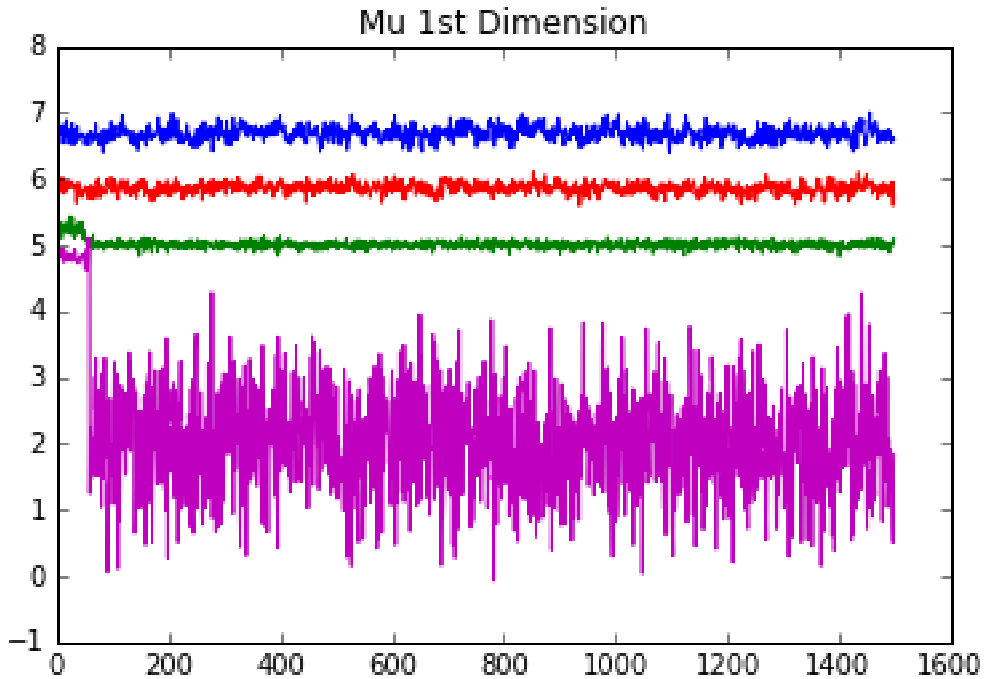


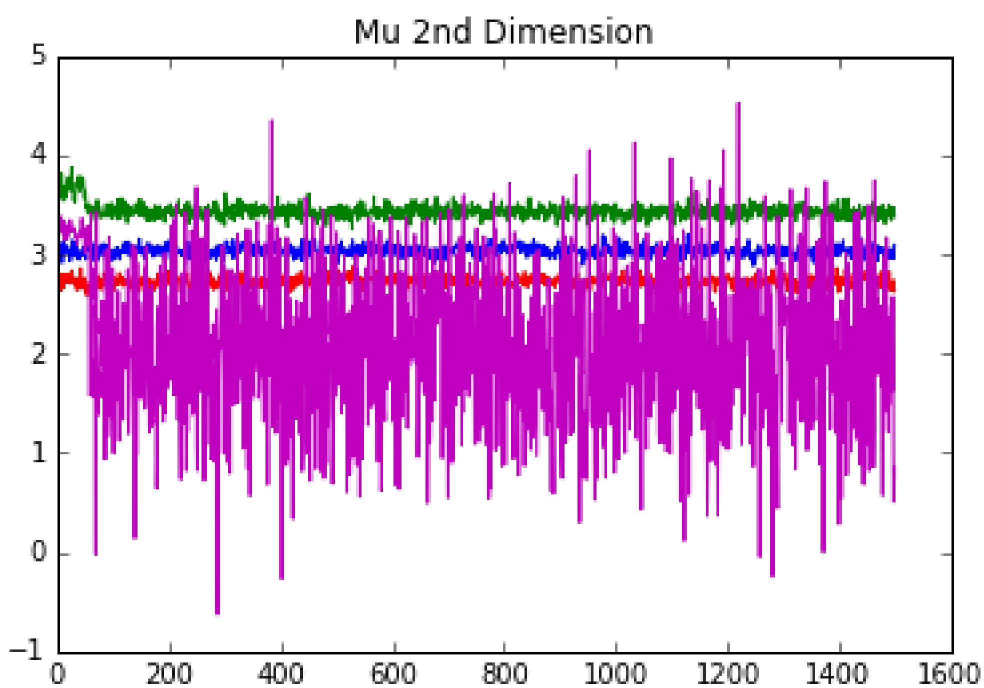
Sigma 2nd Dimension
25
20
15
10
5
0
0
200
400
600
800
1000
1200
1400
1600

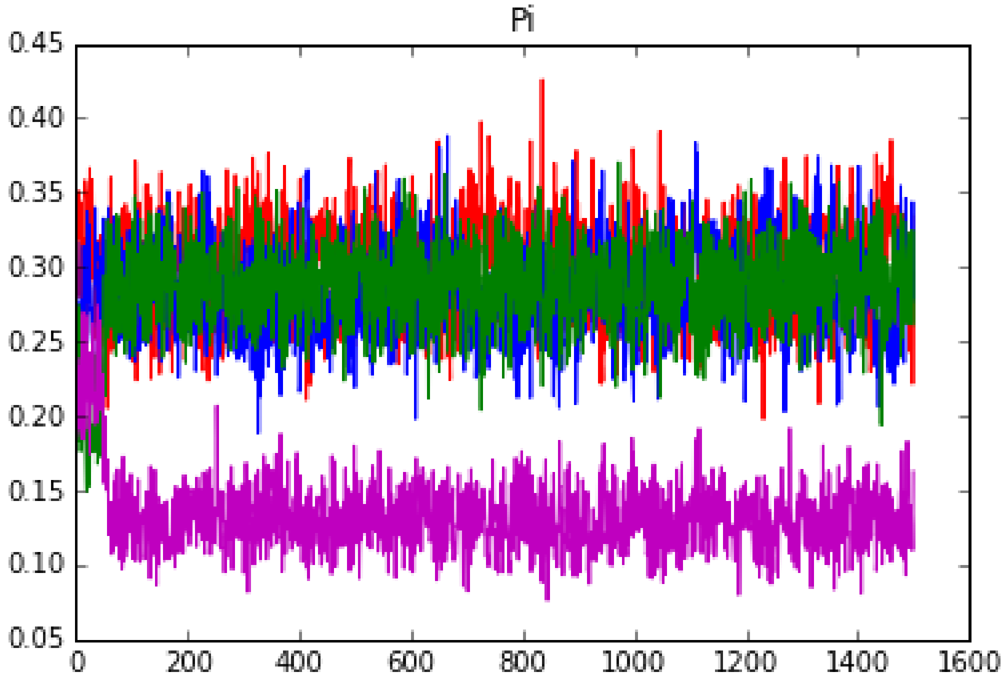


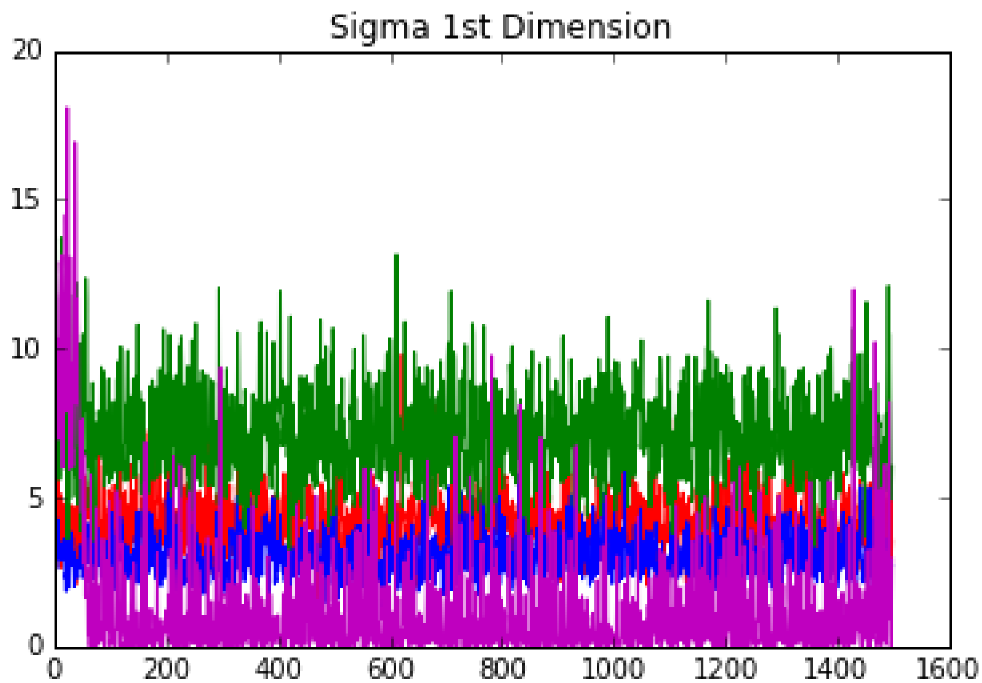

```
In [27]: # Iris data set with four clusters
         irisclusters4, irispi4, irismu4, irissigma4, irislikelihood4 = GibbsSample(irisdata, 4,
         plt.plot(numpy.arange(1500), irismu4[:, 0, 0], "r", numpy.arange(1500), irismu4[:, 1, 0
                  numpy.arange(1500), irismu4[:, 2, 0], "g", numpy.arange(1500), irismu4[:, 3, 0
         plt.title("Mu 1st Dimension")
         plt.show()
         plt.plot(numpy.arange(1500), irismu4[:, 0, 1], "r", numpy.arange(1500), irismu4[:, 1, 1
                  numpy.arange(1500), irismu4[:, 2, 1], "g", numpy.arange(1500), irismu4[:, 3, 1
         plt.title("Mu 2nd Dimension")
         plt.show()
         plt.plot(numpy.arange(1500), irispi4[:, 0], "r", numpy.arange(1500), irispi4[:, 1], "b"
                  numpy.arange(1500), irispi4[:, 2], "g", numpy.arange(1500), irispi4[:, 3], "m"
         plt.title("Pi")
         plt.show()
         plt.plot(numpy.arange(1500), irissigma4[:, 0, 0], "r", numpy.arange(1500), irissigma4[:
                  numpy.arange(1500), irissigma4[:, 2, 0], "g", numpy.arange(1500), irissigma4[:
         plt.title("Sigma 1st Dimension")
         plt.show()
         plt.plot(numpy.arange(1500), irissigma4[:, 0, 1], "r", numpy.arange(1500), irissigma4[:
                  numpy.arange(1500), irissigma4[:, 2, 1], "g", numpy.arange(1500), irissigma4[:
         plt.title("Sigma 2nd Dimension")
         plt.show()

         # plot the clusters
```

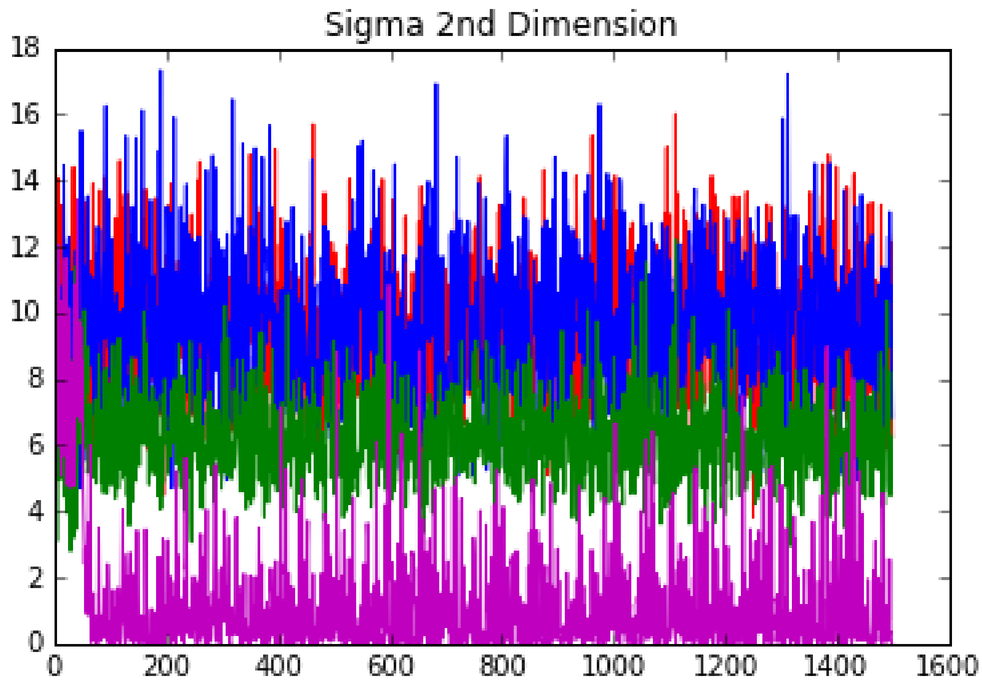
```
plt.scatter(irisdata[:, 0], irisdata[:, 1], c = numpy.mean(irisclusters2[-1000:, :], ax
plt.title("Average Cluster Membership after Convergence")
plt.colorbar()
plt.show()
```

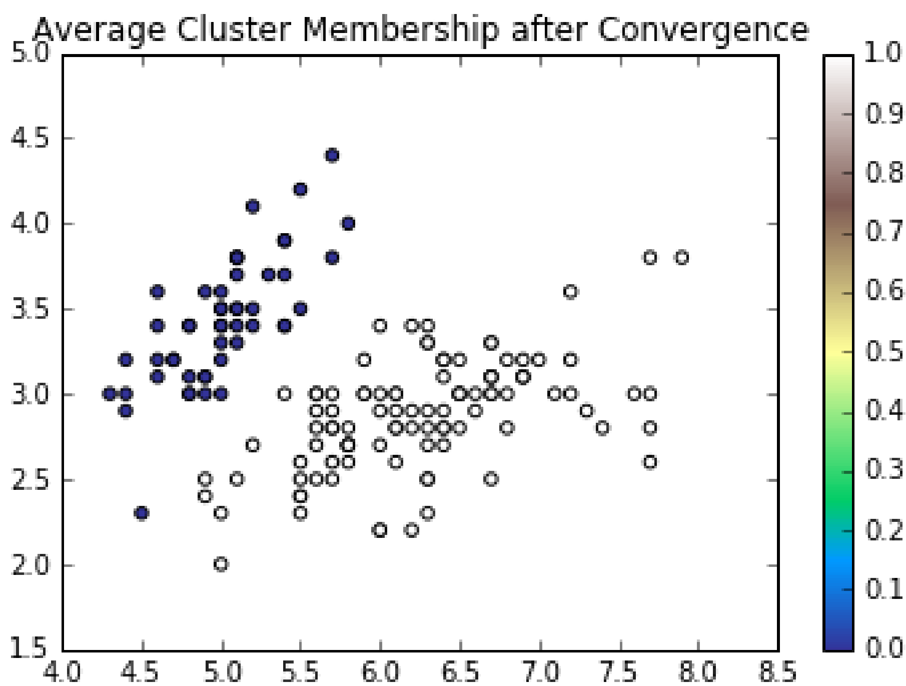





Pi
0.45
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0
200
400
600
800
1000
1200
1400
1600


Sigma 1st Dimension
20
15
10
5
0
0
200
400
600
800
1000
1200
1400
1600


Sigma 2nd Dimension
18
16
14
12
10
8
6
4
2
0
0
200
400
600
800
1000
1200
1400
1600

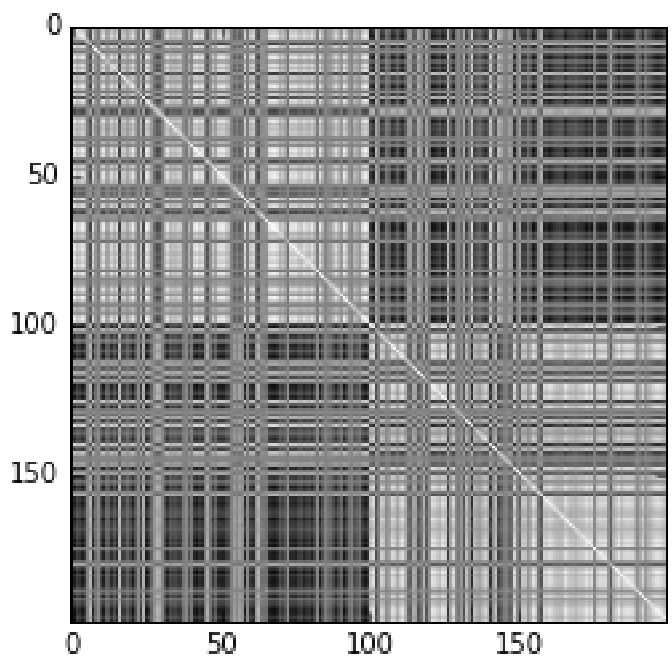

Average Cluster Membership after Convergence
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
4.0
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0
8.5
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0

2 b) Estimate the pairwise co-clustering matrix.

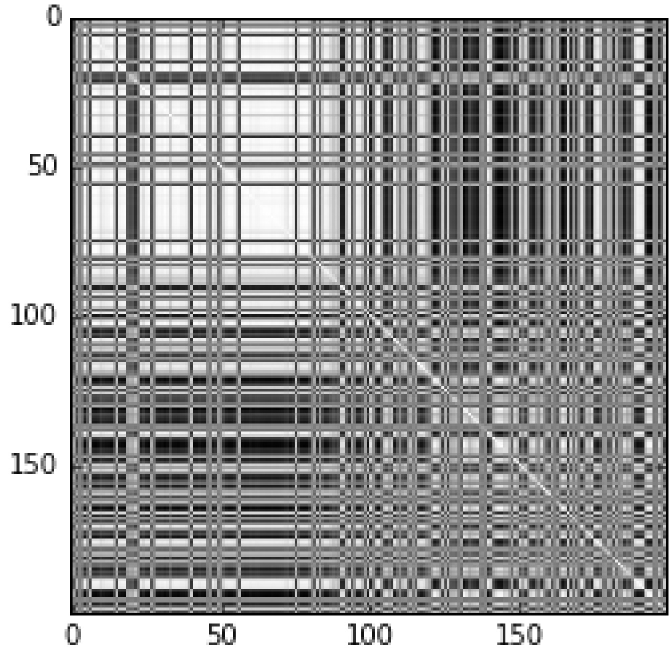
```
In [36]: def coclustering(clusters):
             """ Returns the pairwise co-clustering matrix given a list of clusters sampled for

             # create the empty matrix
             coclusters = numpy.empty((clusters.shape[1], clusters.shape[1]))

             # loop through each pair of points
             for i in range(clusters.shape[1]):
                 for j in range(clusters.shape[1]):
                     coclusters[i, j] = numpy.sum(1*(clusters[-1000:, i] == clusters[-1000:, j])

             return coclusters
```

Now I can display the coclustering matricies for each of the data sets clustered above.

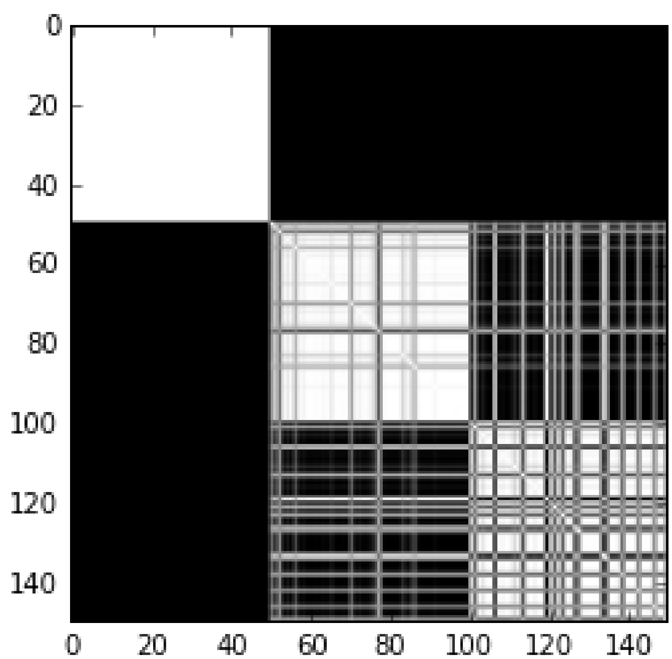
```
In [37]: # 1st synthetic data set
         coclusters = coclustering(S1clusters)
         plt.imshow(coclusters, cmap = "gray")
         plt.show()
```




```
In [38]: # 2nd synthetic data set
         coclusters = coclustering(S2clusters)
         plt.imshow(coclusters, cmap = "gray")
         plt.show()
```




```
In [39]: # iris data set
         coclusters = coclustering(irisclusters3)
         plt.imshow(coclusters, cmap = "gray")
         plt.show()
```



3 c) Estimate the log marginal likelihood of the training data.

```
In [16]: def logmarginalL(likelihood):
             """ Given a list of log likelihoods of each iteration, returns the overal log margi

             return (numpy.log(numpy.sum(numpy.exp(likelihood[-1000:] - numpy.mean(likelihood[-1
                     + numpy.mean(likelihood[-1000:]) + numpy.log(1/(1000+1)))
```

```
In [17]: # likelihood of the 1st synthetic data set
         print(logmarginalL(S1likelihood))
```

```
-45.6905648251
```

```
In [21]: # likelihood of the 2nd synthetic data set
         print(logmarginalL(S2likelihood))
```

```
-26.0345570935
```

```
In [22]: # likelihood of the iris data set with 2 clusters
         print(logmarginalL(irislikelihood2))
```

```
-0.00100444031679
```

```
In [25]: # likelihood of the iris data set with 3 clusters
         print(logmarginalL(irislikelihood3))
```

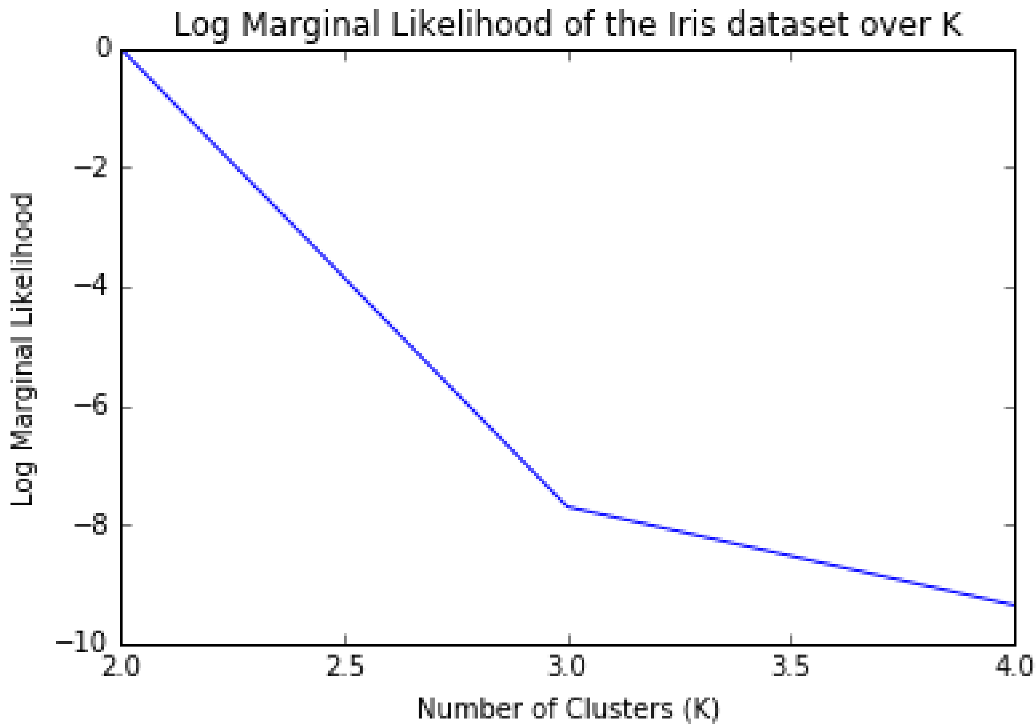
```
-7.70411275245
```

```
In [28]: # likelihood of the iris data set with 4 clusters
         print(logmarginalL(irislikelihood4))
```

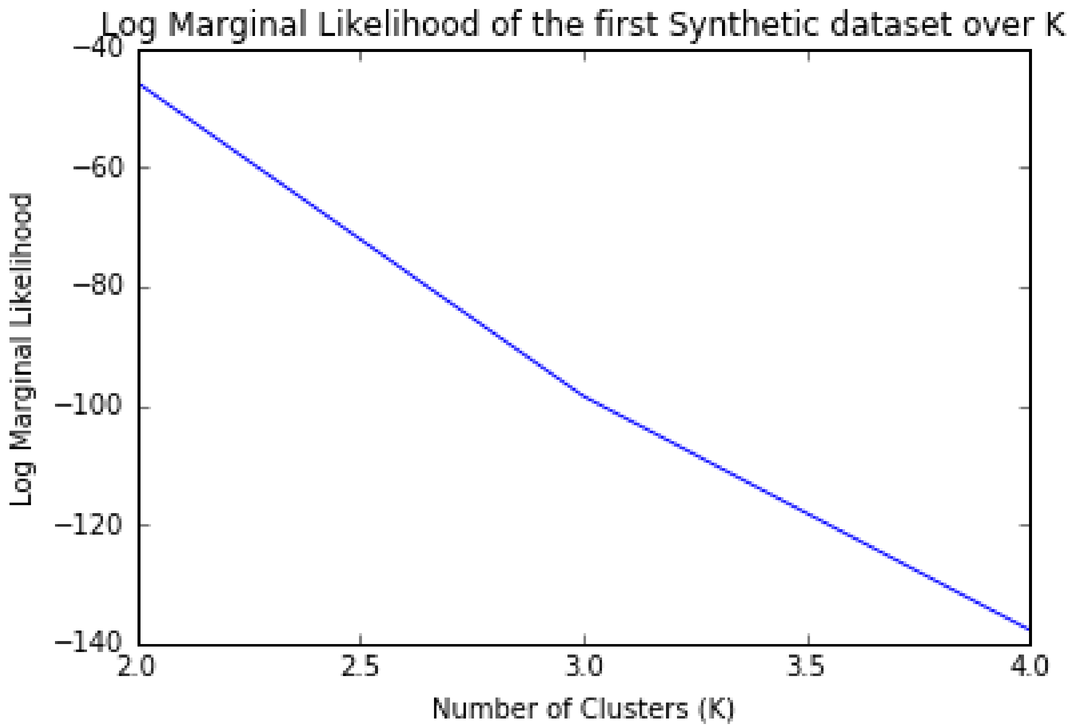
```
-9.35665644078
```

4 d) Plot and compare the results log-likelihood for different values of K for a few data sets. Does this metric tend to select a good value of K?

```
In [29]: plt.plot(range(2, 5), [logmarginalL(irislikelihood2), logmarginalL(irislikelihood3), lo
                 "b")
         plt.title("Log Marginal Likelihood of the Iris dataset over K")
         plt.xlabel("Number of Clusters (K)")
         plt.ylabel("Log Marginal Likelihood")
         plt.show()
```



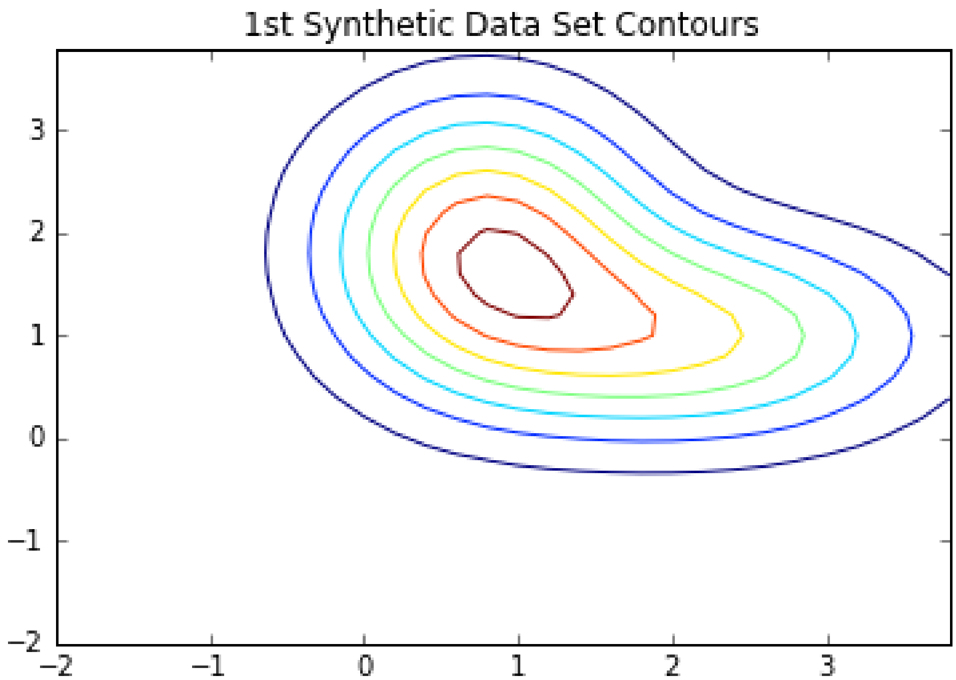

```
In [24]: plt.plot(range(2, 5), [logmarginalL(S1likelihood), logmarginalL(S1likelihood3), logmarg
                 "b")
         plt.title("Log Marginal Likelihood of the first Synthetic dataset over K")
         plt.xlabel("Number of Clusters (K)")
         plt.ylabel("Log Marginal Likelihood")
         plt.show()
```



It seems that log marginal likelihood prefers lower values of K. This may be because, as I have found, the more clusters there are, the less stable they are.

5 e) Plot contours of the marginal density function.

In [30]:
```
# for convenience, I'll use matplotlib's bivariate normal function
import matplotlib.mlab as mlab

# make the x and y coordinates of the grid
X, Y = numpy.meshgrid(numpy.arange(-2, 4, .2), numpy.arange(-2, 4, .2))
Z = numpy.zeros((30, 30))
# and the values at those points
for k in range(2):
    Z += numpy.mean(S1pi[-1000:, k])*mlab.bivariate_normal(
        X, Y, sigmax = numpy.mean(1/S1sigma[-1000:, k, 0]), sigmay = numpy.mean(1/S1sig
        mux = numpy.mean(S1mu[-1000:, k, 0]), muy = numpy.mean(S1mu[-1000:, k, 1]))

plt.contour(X, Y, Z)
plt.title("1st Synthetic Data Set Contours")
plt.show()
```



For this data set, I put a prior on the mean with mean 1.5 and variance .5 and a prior on the variance with both a and b equal to 1. This reflects the prior parameters on the mean fairly well, with the center of the contour around 1.5 in both dimensions. This is probably in part due to the fact that I decided on the prior mean of 1.5 based on a glace at the data. The variance appears to be closer to 1.5 or 2 than my prior guess of 1 suggests.

2. Bayes Net

May 2, 2017

1 a) Write down the factorization of the joint distribution that is implied by the graph.

$$p(x,d,e,t,l,b,a,s) = p(a)p(t|a)p(s)p(l|s)p(b|s)p(e|t,l,a,s)p(d|e,b,t,l,a,s)p(x|e,t,l,a,s)$$

2 b) Are s and a independent?

Yes. They are only connected through their children, in fact neither node has any parents at all. This makes their relationship a "head-to-head" connection, which information cannot flow through. They have no influence on each other.

3 c) Are s and a conditionally indpendent given x?

No. x is descended from both s and a. The relationship between s and a is a head-to-head connection, so information about their mutual descendent opens up the flow of information between them. Knowing the state of x implies that at least one of s or a must have caused it. If then information surfaces indicating that a is the culprit, that influences the probability of s by explaining it away, back to its original probability before the state of x had been discovered.

3. HMMs for Typo Correction

May 2, 2017

```
In [1]: import numpy

        # load the emission and transition probabilities
        transition = numpy.loadtxt("typing_transition_matrix.csv", delimiter = ",", skiprows = 1
        emission = numpy.loadtxt("typing_emission_matrix.csv", delimiter = ",", skiprows = 1)
```

1 a) Implement the forward-backward algorithm to obtain samples from the posterior distribution of the latent state sequence given the observed sequence.

```
In [2]: def filterForward(transition, emission, data):
            """ Calculates and returns the probability of each hidden state being in each locati
            sequence (translated into indices of the emission table) and probabilities of transi

            # empty forward message
            m = numpy.empty((len(data), transition.shape[0]))
            # and a variable to hold the sum of the log normalizing constants
            normc = 0
            # fill the first column
            m[0, :] = emission[:, data[0]]/transition.shape[0]
            # and normalize
            normc = numpy.log(numpy.sum(m[0, :]))
            m[0, :] = m[0, :]/numpy.sum(m[0, :])

            # loop through the message and fill each column
            for t in range(1, len(data)):
                # calculate the message
                m[t, :] = numpy.dot(numpy.transpose(transition), m[t-1, :])*emission[:, data[t]]
                # and normalize it
                normc += numpy.log(numpy.sum(m[t, :]))
                m[t, :] = m[t, :]/numpy.sum(m[t, :])

            # return the forward message and the cumulative log normalizing constant
            return m, normc
```

```
def backSample(transition, emission, data):
    """ Sample hidden states backwards. """

    # first calculate the probability of each hidden state being in each location by fil
    m, normc = filterForward(transition, emission, data)

    # create a list to hold the sampled states
    samples = numpy.empty(len(data), dtype = int)
    # initialize the last letter based on m
    samples[len(data)-1] = numpy.random.choice(27, p = m[len(data)-1])

    # loop thorugh the data backwards and sample hidden states
    for d in range(len(data)-2, -1, -1):
        # calculate the distribution for the next hidden state
        dist = m[d]*transition[:, samples[d+1]]
        # and normalize it
        dist = dist/numpy.sum(dist)
        # now sample an index from the distribution we just generated
        samples[d] = numpy.random.choice(27, p = dist)

    # return the samples
    return samples
```

2 b) Use the algorithm to sample a few thousand possible intended sequences given the observed sequence "kezrninh".

3 c) Check to see if the results are actual words and only print out the ones that are.

```
In [3]: import text_utils

        # transform the string into a sequence of indicies for convenience
        d = text_utils.encode_string_to_int_list("kezrninh")
        # save the dictionary
        engdict = text_utils.dict_from_file("brit-a-z.txt")

        # run the algorithm 1000 times, printing out each possible intended sequece that is a re
        for i in range(5000):
            r = backSample(transition, emission, d)
            result = text_utils.decode_int_list_to_string(r)
            # check to see if the word is in the dictionary
            if text_utils.check_validity(result, engdict):
                print(result)
```

```
lead inn
learning
meat inn
```

```
learning
mead inn
learning
lest inn
learning
jest inn
learning
learning
lead inn
learning
learning
```

4 d) Implement a Gibbs sampler that alternates between sampling hidden sequences conditioned on a best guess for transition and emission, and sampling transition and emission probabilities based on the current guess at the hidden sequence.

```
In [4]: def HMMGibbs(data, iters, alpha_transition, alpha_emission, K_emission):
            """ Uses Gibbs sampling to estimate the posterior on the transition and emission pro

            # array to store all of the generated sequences
            seqs = numpy.empty((iters, data.shape[0]), dtype = int)
            # with the input data as the first sequence
            seqs[0, :] = data
            # another to store the generated transition probabilities
            transitions = numpy.empty((iters, 27, 27))
            # and the generated emission probabilities
            emissions = numpy.empty((iters, 27, 27))

            # alternate updating parameters to sample from the posterior and sampling sequences
            for i in range(iters):
                # update each row individually
                for k in range(27):
                    # grab the indicies of the current best guess with value k
                    ks = numpy.where(seqs[i, :] == k)[0]

                    # calculate the number of times each transition occurs
                    nk_transition = numpy.zeros(27)
                    for kdex in ks:
                        if kdex < data.shape[0]-1:
                            nk_transition[seqs[i, kdex+1]] += 1
                    # update the posterior on the transitions
                    transition_posterior = nk_transition + alpha_transition/27
                    # sample transitions from the dirichlet
                    transitions[i, k, :] = numpy.random.dirichlet(transition_posterior)
```

```
            # calculate the number of times each emission occurs
            nk_emission = numpy.zeros(27)
            for kdex in ks:
                nk_emission[data[kdex]] += 1
            # update the posterior on the emissions
            emission_posterior = nk_emission + alpha_emission/27
            # add K to the value corresponding to the intended letter being generated
            emission_posterior[k] += K_emission
            # sample emissions from the dirichlet
            emissions[i, k, :] = numpy.random.dirichlet(emission_posterior)

        # sample a new sequence using the forward-backward algorithm unless this is the
        if i+1 < iters:
            seqs[i+1, :] = backSample(transitions[i, :, :], emissions[i, :, :], data)

    # return the results
    return seqs, transitions, emissions
```

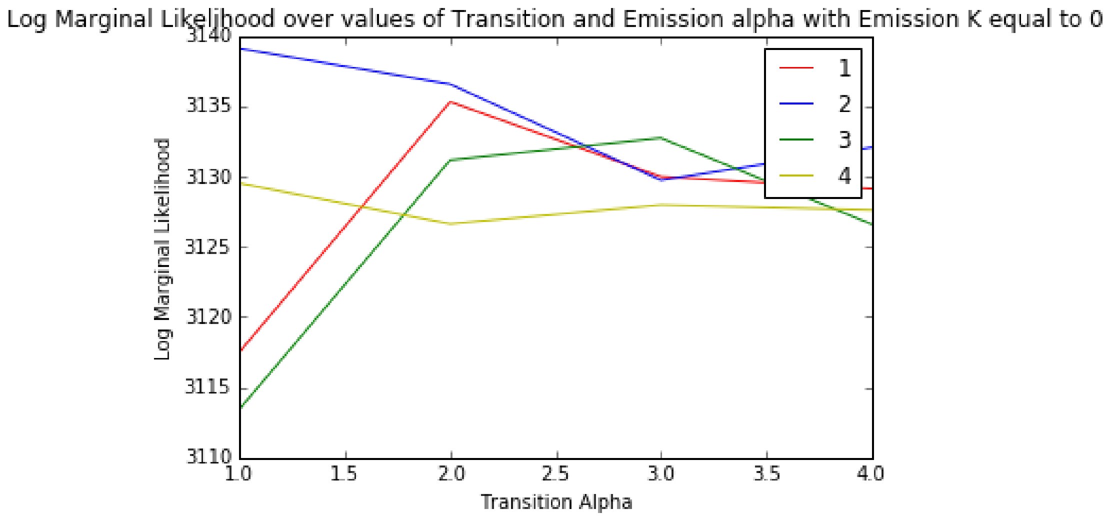
5 e) Generate three random sequences using the ground truth emission and transition probabilities and use them to train, determine the appropriate hyper-parameter values, and test the Gibbs sampler. Compare the best results to the actual probabilities.

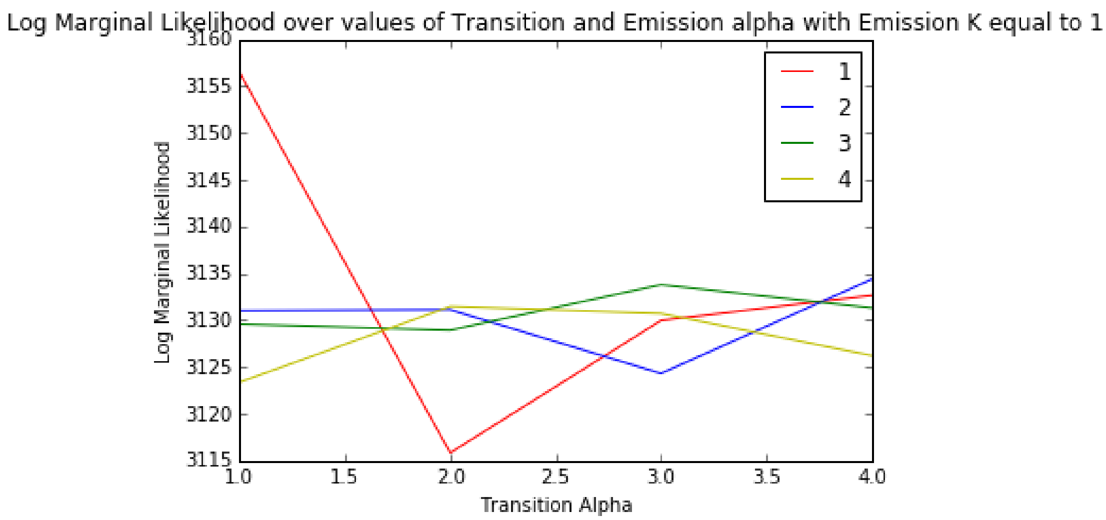
```
In [5]: def generateSeq(transition, emission, length):
            """ Generates a random sequence of the provided length using the provided emission a

            # create a list to hold the hidden states
            hiddens = numpy.empty(length, dtype = int)
            # and another to hold the resulting visible states
            visibles = numpy.empty(length, dtype = int)

            # generate the designated number of hidden and visible states
            for i in range(length):
                if i == 0:
                    # choose a random starting hidden state
                    hiddens[i] = numpy.random.choice(27)
                else:
                    # or use the previous hidden state to determine the next
                    hiddens[i] = numpy.random.choice(27, p = transition[hiddens[i-1], :]/numpy.s
                # and generate the corresponding visible state
                visibles[i] = numpy.random.choice(27, p = emission[hiddens[i], :]/numpy.sum(emis

            # return the completed sequences
            return hiddens, visibles

In [6]: def validation(trainset, validset, transition_alphas, emission_alphas, emission_Ks):
```

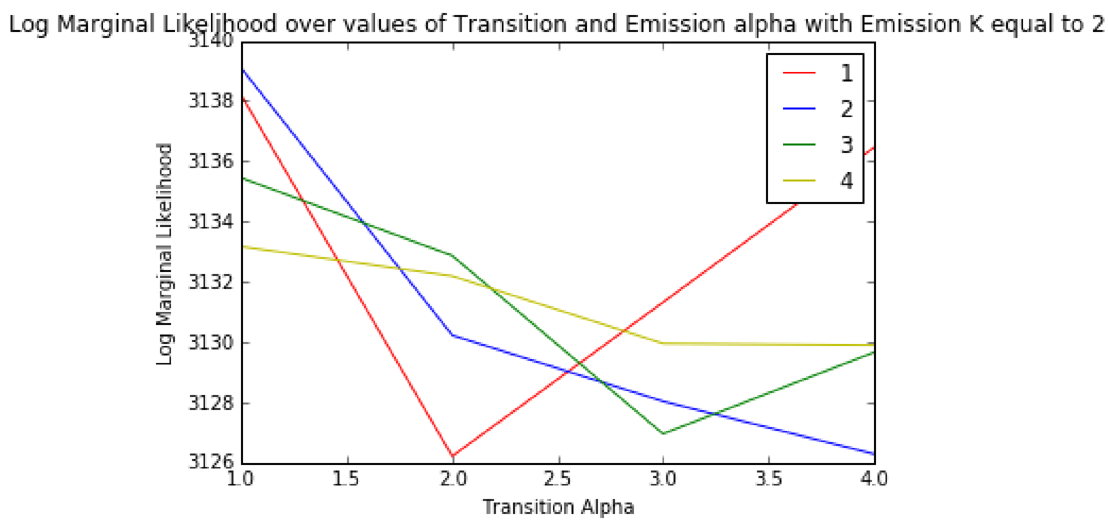
```
    """ Finds the marginal likelihood of the validation set for each combination of para

    # create an array to hold the marginal likelihood of each combination
    marginalL = numpy.empty((len(transition_alphas), len(emission_alphas), len(emission_

    for ta in range(len(transition_alphas)):
        for ea in range(len(emission_alphas)):
            for eK in range(len(emission_Ks)):
                # run Gibbs sampling for each combination of parameters
                seq, transition, emission = HMMGibbs(trainset, 5000, transition_alphas[t
                                                     emission_alphas[ea], emission_alpha
                # average all the post burn in transition and emission values
                transition = numpy.mean(transition[1000:, :, :], axis = 0)
                emission = numpy.mean(emission[1000:, :, :], axis = 0)
                # use the average parameter values to find the likelihood of the validat
                m, normc = filterForward(transition, emission, validset)
                # calculate the log likelihood of the data, save it and print it
                marginalL[ta, ea, eK] = numpy.sum(numpy.log(m[-1, :]))-normc

    # return the array of marginal likelihoods for analysis
    return marginalL
```

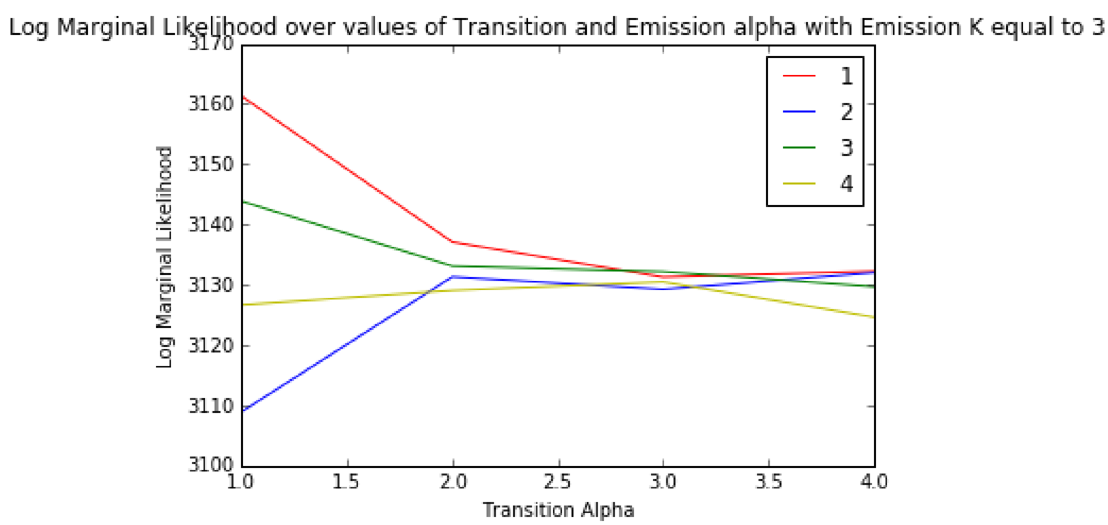
```
In [7]: # generate the training, validation, and test sets
        trainhid, trainvis = generateSeq(transition, emission, 1000)
        validhid, validvis = generateSeq(transition, emission, 1000)
        testhid, testvis = generateSeq(transition, emission, 1000)

        marginalL = validation(trainvis, validvis, range(1, 5), range(1, 5), range(1, 5))
```

```
In [9]: import matplotlib.pyplot as plt

        for i in range(4):
            plt.title("Log Marginal Likelihood over values of Transition and Emission alpha with
            plt.plot(range(1, 5), marginalL[:, 0, i], "r", range(1, 5), marginalL[:, 1, i], "b",
                 range(1, 5), marginalL[:, 2, i], "g", range(1, 5), marginalL[:, 3, i], "y")
            plt.legend(range(1, 5))
            plt.ylabel("Log Marginal Likelihood")
            plt.xlabel("Transition Alpha")
            plt.show()
```


Log Marginal Likelihood over values of Transition and Emission alpha with Emission K equal to 0
Log Marginal Likelihood
Transition Alpha
1
2
3
4


Log Marginal Likelihood over values of Transition and Emission alpha with Emission K equal to 1
Log Marginal Likelihood
Transition Alpha
1
2
3
4



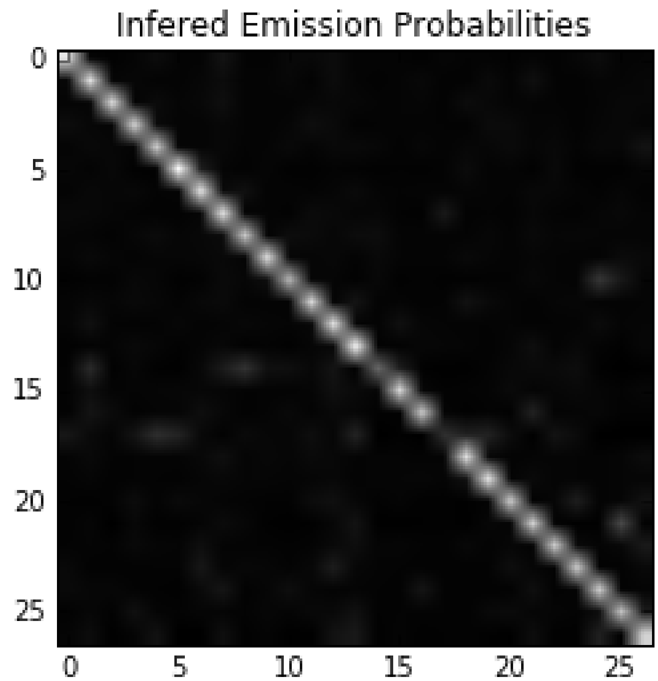


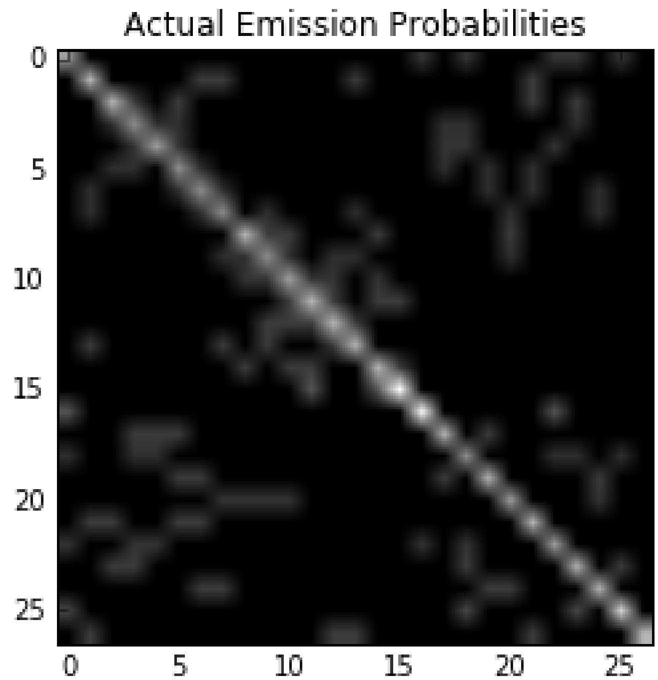

```
In [20]: # write the Log Marginal Likelihoods to file so I don't have to do this again
         for k in range(4):
             numpy.savetxt("3e_likelihood_eK" + str(k+1) + ".csv", marginalL[:, :, k], fmt = "%.
```

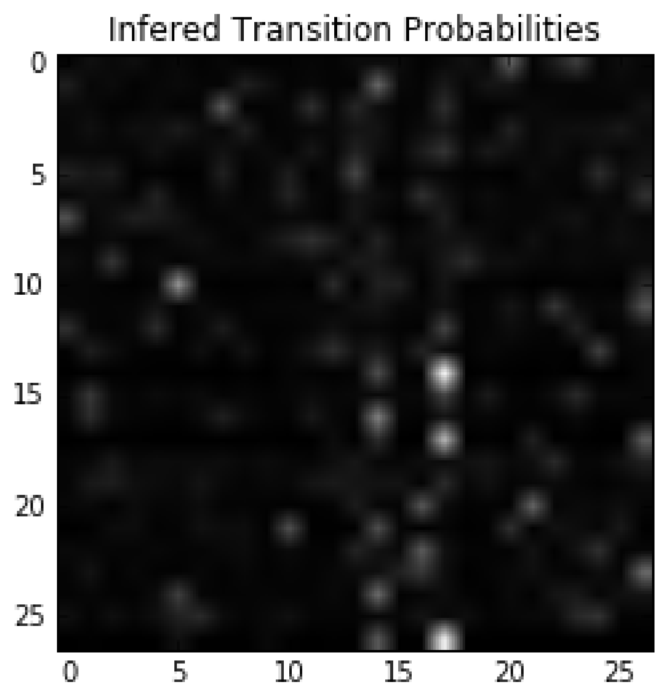
It looks like the optimal parameters are alpha of 1 for transition, an alpha of 2 for emission, and a K of 3. I can now use those to find the emission and transition probabilities for the test set.

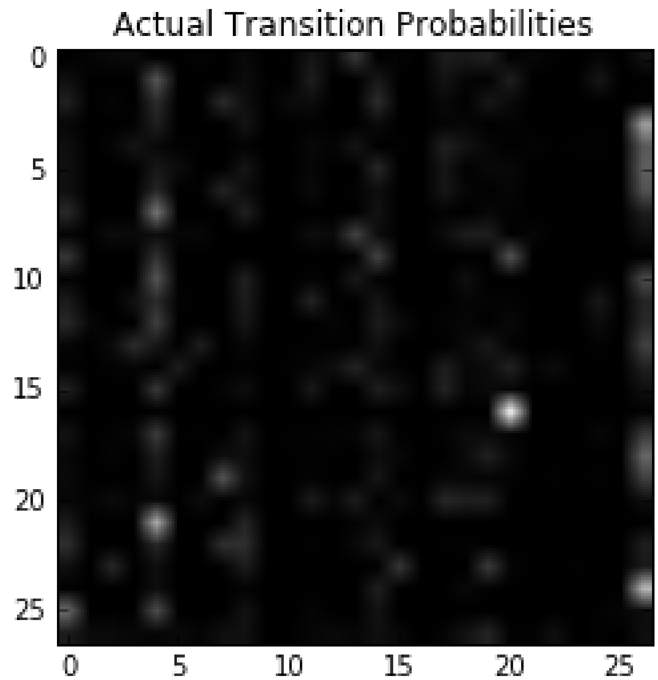
```
In [8]: import matplotlib.pyplot as plt

        seq, esttransition, estemission = HMMGibbs(testvis, 5000, 1, 2, 3)
```

```python
# plot the emission and transition probabilities to compare to the actual values.
plt.imshow(numpy.mean(estemission[2000:, :, :], axis = 0), cmap = "gray")
plt.title("Infered Emission Probabilities")
plt.show()
plt.imshow(emission, cmap = "gray")
plt.title("Actual Emission Probabilities")
plt.show()
plt.imshow(numpy.mean(esttransition[2000:, :, :], axis = 0), cmap = "gray")
plt.title("Infered Transition Probabilities")
plt.show()
plt.imshow(transition, cmap = "gray")
plt.title("Actual Transition Probabilities")
plt.show()
```




Actual Emission Probabilities
0
5
10
15
20
25
0
5
10
15
20
25


Infered Transition Probabilities
0
5
10
15
20
25
0
5
10
15
20
25



The images are rather blurry due to all the comparisons, but major trends are visible. There are some significant differences between the infered and actual transition and emission probabilities. Most clear among them is the fact that the infered probabilities tend to be lower than the actual ones. However, they exhibit a similar structure, with the diagonal of 1s in the emission probabilities and the bright vertical stripes in the transition probabilities. The stripes are in different places, but that may reflect the somewhat small sentences I used.