DiffusionMaps

June 17, 2018

1 Diffusion maps for single-cell data analysis

By means of this notebook, you can solve all the programming tasks of chapter 4. You can download all the needed files (data1.mat and guo.xlsx) from the homepage.

```
In [1]: %matplotlib inline
        import numpy as np
        from scipy.io import loadmat
        from scipy.spatial import distance_matrix
        from pandas import read_excel
        import matplotlib.pyplot as plt
        from mpl_toolkits.mplot3d import Axes3D
        from sklearn.decomposition import PCA
        from sklearn.manifold import TSNE, Isomap
        from sklearn.cluster import KMeans
        import time
        from IPython.display import Latex
```

1.1 Introduction

Task 1: Implement the diffusion maps algorithm. It is recommended to solve this task by defining a class for diffusion maps and implement a fit_transform function, which returns the embedding of a given data set. This standardizes the code when comparing diffusion maps with other dimensionality reduction methods.

```
In [2]: class DiffMap():
            """
            Class Diffusion Maps

            Parameters
            ----------
            n_components: int, optional, default = 2
                number of dimensions in which the data will be embedded
            sigma: optional, default = 10
                bandwidth of the Gaussian kernel
            alpha: optional, default = 1
                the density rescaling parameter
```

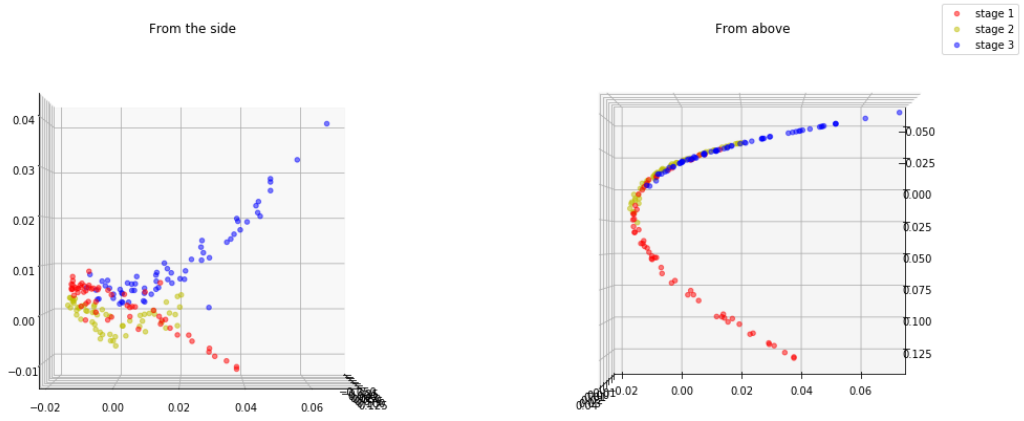
```
        localDensities: optional, default = True
            whether local densities are used, i.e. the transition matrix has a non-zero diag
        """

        def __init__(self, n_components = 2, sigma = 10., alpha = 1., localDensities = True)
            assert(n_components > 0 and sigma > 0 and alpha >= 0 and alpha <= 1)
            self.ndim = n_components
            self.sigma = sigma # bandwidth for the Gaussian kernel
            self.alpha = alpha
            self.localDensities = localDensities
            self.P = None

        def _gauss_matrix(self, x, y):
            """
            Computes the standard Gaussian kernel; similar in operation to distance_matrix f

            Parameters
            ----------
            x: array [m, n_comp]
               m vectors in n_comp dims
            y: array [k, n_comp]
               m vectors in n_comp dims

            Returns
            -------
            dists: array [m,k]
                   array of kernel output, where the (i,j) element is the kernel value betwe
            """
            # if either x or y are a single coordinate, force them to be a 2-dim array
            if len(x.shape) == 1:
                x = np.array([x])

            if len(y.shape) == 1:
                y = np.array([y])

            # start computing gaussian kernel
            D = distance_matrix(x,y)**2 # every element of D is the squared euclidean distan
                                        # distance_matrix returns array type
            return np.exp(-D/(2.*self.sigma**2))

        def getEigens(self):
            # Returns ordered eigenvalues & eigenvectors of the transition matrix
            P = self.getTransitionMatrix()
            evals, evecs = np.linalg.eig(P)
            sorted_indices = np.argsort(-evals) # sort the eigenvalues from largest to small
            return evals[sorted_indices], evecs[:, sorted_indices] # return the sorted eigen

        def getTransitionMatrix(self):
```

```
        # Returns the transition matrix
        if len(self.P) == 0 and self.P == None: # Model wasn't fit yet
            raise RuntimeWarning("The model has to be fit first.")
        return self.P

    def fit(self, X):
        # Fit X into an embedded space.
        # Computes the transition matrix given by the data X
        K = self._gauss_matrix(X, X) # get the similarity matrix
        Q_minusalpha = np.diag((K.dot(np.ones(X.shape[0])))**(-self.alpha))
        K_alpha = Q_minusalpha.dot(K.dot(Q_minusalpha))
        if not self.localDensities: K_alpha -= np.diag(np.diag(K_alpha)) # set the diago
        D_minusalpha = np.diag((K_alpha.dot(np.ones(X.shape[0])))**(-1))
        self.P = D_minusalpha.dot(K_alpha)
        return self

    def fit_transform(self, X):
        #Fit X into an embedded space and return that transformed output.
        self.fit(X)
        evals, evecs = self.getEigens()
        return np.multiply(evals[1:self.ndim+1], evecs[:,1:self.ndim+1])
```

Task 2: Perform a diffusion map analysis on the Buettner data set.

```
In [3]: def load_buettner_data():
            #load buettner data
            file = loadmat('data//data1.mat')
            data = file.get('in_X')
            data = np.array(data)

            #group assignments
            labels = file.get('true_labs')
            labels = labels[:,0] -1

            #group names
            stage_names = ['1', '2', '3']

            return data, stage_names, labels

In [4]: data, stage_names, labels = load_buettner_data()
        # data.shape = (182,8989)

In [5]: diff_coords = DiffMap(sigma = 20, n_components = 3, localDensities = False).fit_transfor

In [6]: # plot data in 3d space
        group = []
        for l in set(labels):
            group.append(diff_coords[labels == l, :])
```

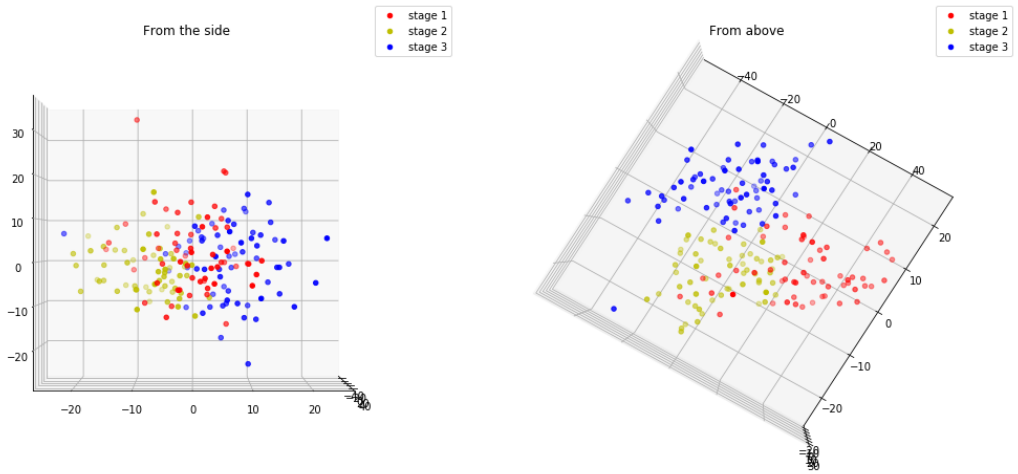
```
fig = plt.figure(figsize = (15, 7))
ax = fig.add_subplot(121, projection = '3d')
for l, col in zip(set(labels), ('r','y','b')):
    #fig = plt.figure()
    #ax = fig.add_subplot(111, projection='3d')
    curgroup = group[l]
    xs = curgroup[:, 0]
    ys = curgroup[:, 1]
    zs = curgroup[:, 2]
    ax.scatter(xs, ys, zs, alpha = 0.5, label = 'stage {0}'.format(stage_names[l]), c =
ax.set_title("From the side")
ax.elev=0
ax.azim=0

ax = fig.add_subplot(122, projection = '3d')
for l, col in zip(set(labels), ('r','y','b')):
    #fig = plt.figure()
    #ax = fig.add_subplot(111, projection='3d')
    curgroup = group[l]
    xs = curgroup[:, 0]
    ys = curgroup[:, 1]
    zs = curgroup[:, 2]
    ax.scatter(xs, ys, zs, alpha = 0.5, label = 'stage {0}'.format(stage_names[l]), c =
ax.set_title("From above")
ax.elev=90
ax.azim=0

ax.legend()
plt.tight_layout()
```



Task 3: Perform a PCA analysis of the Buettner data set.

In [7]:
```
# run PCA on the data
q = 3
pca_proj = PCA(n_components=q).fit_transform(data)
```

In [8]:
```
# plot data in 3d space
group = []
for l in set(labels):
    group.append(pca_proj[labels == l, :])

fig = plt.figure(figsize=(15,7))
ax = fig.add_subplot(121, projection='3d')
for l, col in zip(set(labels), ('r','y','b')):
    curgroup = group[l]
    xs = curgroup[:, 0]
    ys = curgroup[:, 1]
    zs = curgroup[:, 2]
    ax.scatter(xs, ys, zs, label = 'stage {0}'.format(stage_names[l]), c = col)
ax.elev = 0
ax.azim = 0
ax.set_title("From the side")
ax.legend()
ax = fig.add_subplot(122, projection='3d')
for l, col in zip(set(labels), ('r','y','b')):
    curgroup = group[l]
    xs = curgroup[:, 0]
    ys = curgroup[:, 1]
    zs = curgroup[:, 2]
    ax.scatter(xs, ys, zs, label = 'stage {0}'.format(stage_names[l]), c = col)
ax.elev = 90
ax.set_title("From above")
ax.legend()
plt.tight_layout()
```



1.1.1 Observations for 4.2 and 4.3

While both PCA and DiffusionMap give results that appear to be separable, there is clearly more structure in the DiffusionMap result, which is suggestive of some specific higher-dimensional structure in the original data (e.g. the DiffusionMap result may be further refined to a 2D plane with a polynomial structure, or perhaps it has a helical structure in 3D, etc), while the PCA result does not suggest anything of the kind.

1.2 Single-cell data analysis

In the following, we will apply diffusion maps to the Guo data. In the file, you will find some necessary information:

1. the input data, which is a matrix with a certain number of cells as row number and a certain number of genes as column number,
2. the names of the measured genes and
3. an assignment of each cell to an embryonic stage. These assignments have to be converted into numerical labels to use them for the scatter plots.

1.2.1 Pre-processing

Task 4: Pre-process the Guo data. Take a look at the file guo.xlsx. The naming annotation in the first column refers to the embryonic stage, embryo number, and individual cell number, thus 64C 2.7 refers to the 7th cell harvested from the 2nd embryo collected from the 64-cell stage. In the first row, you will find the names of the measured genes.

```
In [9]: def load_guo_data(preprocess=1):
            #load guo data
            data_frame = read_excel('data//guo.xlsx', sheet_name = 'Sheet1')

            #data
            adata = data_frame.values
            data = adata[:,1:]
            embryonic_stages = adata[:,0]

            #genes
            genes_tmp = data_frame.axes[1][1:]
            genes_names = [genes_tmp[k] for k in range(genes_tmp.size)]

            # data cleanup
            data_undetectable = np.mat(data > 28)
            bad_rows = np.array(data_undetectable*np.mat(np.ones((data_undetectable.shape[1],1))
                # where bad means the row has at least one entry > 28
            bad_rows = np.reshape(bad_rows, (bad_rows.shape[0],)) # reshape into a 1-dimensional
            bad_rows_and_1C = np.concatenate((np.ones((9,)), bad_rows[9:]), 0) # make sure the f
```

```
    if preprocess == 1:
        # normalise data via reference genes. in `data`, actb is row 0, gapdh is row 15.
        # rows with values > 28
        mean_actb = np.mean(data[bad_rows == 0, 0])
        mean_gapdh = np.mean(data[bad_rows == 0, 15])
        data = data[bad_rows_and_1C == 0, :] - 0.5*(mean_actb + mean_gapdh)

        # new threshold is the smallest integer >= the maximum value of the normalised d
        new_threshold = np.ceil(np.max(data[data < 28]))
        data[data == 28] = new_threshold # assign new threshold

        # round all entries to 3 decimal places
        data = np.around(data.astype(np.double), decimals = 3)

        # cut out the respective embryonic stages
        embryonic_stages = embryonic_stages[bad_rows_and_1C == 0]

        # stage_names and creating labels
        stage_names = ['2C', '4C', '8C', '16C', '32C', '64C']

    elif preprocess == 0:
        print('No preprocessing')
        # cut out only undetectable data and 1-cell stage
        data = data[bad_rows ==0, :]
        # cut out respective embryonic stages
        embryonic_stages = embryonic_stages[bad_rows == 0]
        stage_names = ['1C', '2C', '4C', '8C', '16C', '32C', '64C'] # since we don't cut

    # generate respective labels based on modified embryonic_data
    labels = np.array([l for ename in embryonic_stages for l, sname in enumerate(stage_n

    return data, stage_names, labels
```

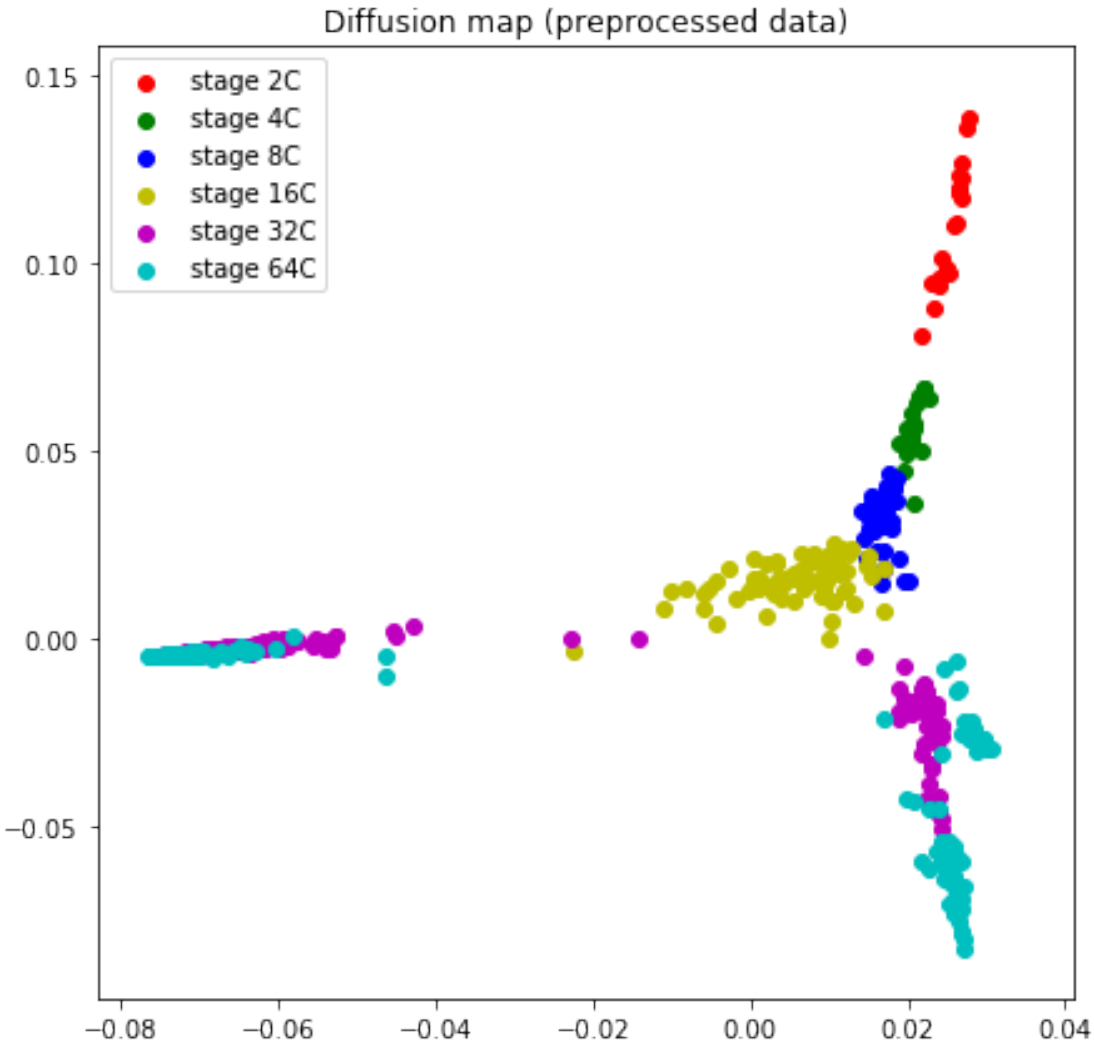
```
In [10]: data, stage_names, labels = load_guo_data()
```

Task 5: Perform a diffusion map analysis of the pre-processed Guo data.

```
In [11]: diff_coords = DiffMap(sigma = 10, n_components = 2).fit_transform(data)
```

```
In [12]: # plot data
         group = []
         for l in set(labels):
             group.append(diff_coords[labels == l, :])

         # combined plot
         fig = plt.figure(figsize = (7, 7))
         ax = fig.add_subplot(111)
         for l, col in zip(set(labels), ('r','g','b','y','m','c')):
```

```
        curgroup = group[l]
        xs = curgroup[:, 0]
        ys = curgroup[:, 1]
        ax.scatter(xs, ys, label = 'stage {0}'.format(stage_names[l]), c = col)

    ax.set_title('Diffusion map (preprocessed data)');
    ax.legend();
```



1.2.2 Interpreting the diffusion map data

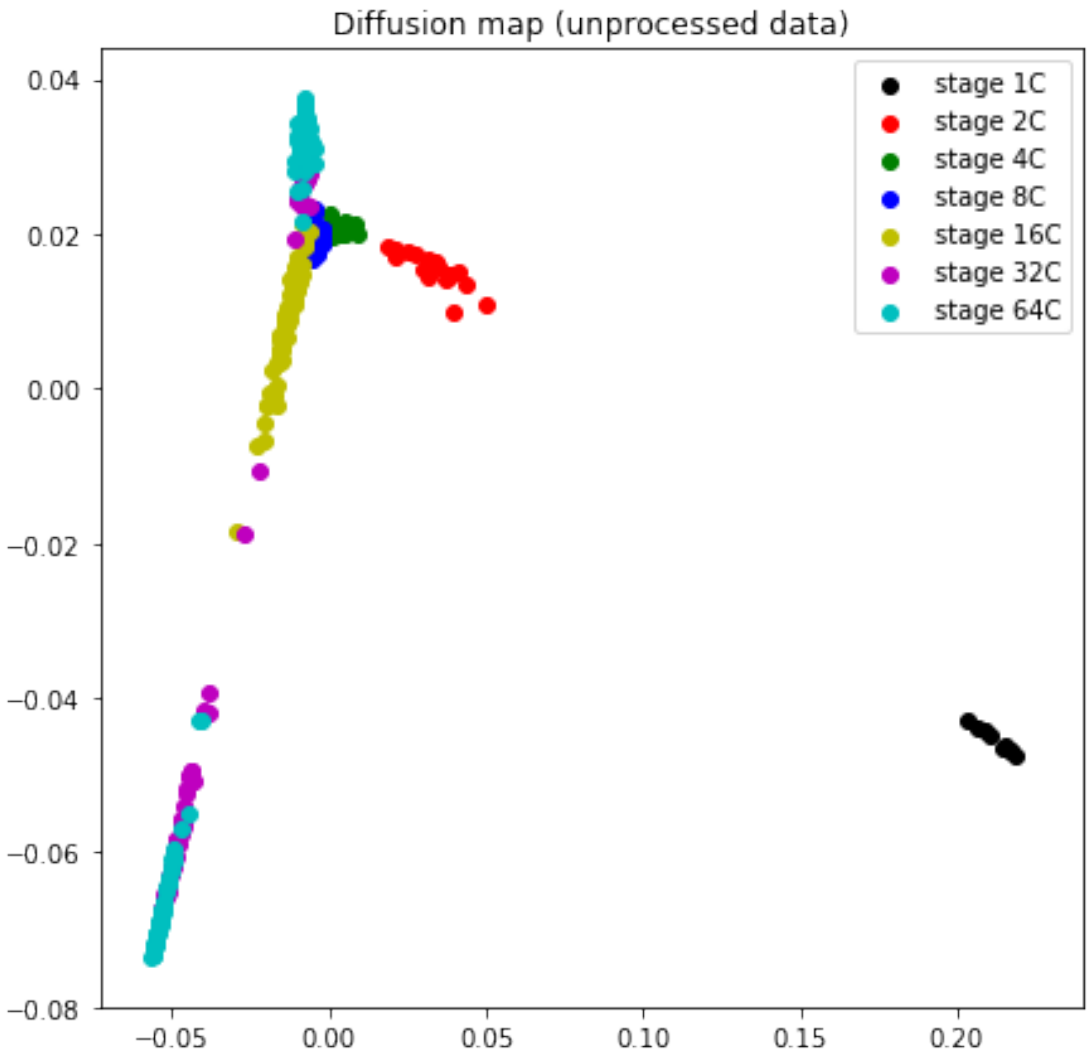
We can see here that there is a somewhat clear "branching" picture created by the diffusion map: we have the 2-cell stage in the top right corner, which then "flows" down to the next closest data clutser, the 4-cell stage, then to the 8- and 16- cell stages. At around this time we see that there is a branching action, where the 32- and 64-cell stages are split off in two different directions.

Task 6: Comparison with the un-pre-processed data.

```
In [13]: data, stage_names, labels = load_guo_data(preprocess=0)
```

```
No preprocessing
```

```
In [14]: diff_coords = DiffMap(sigma = 10, n_components = 2).fit_transform(data)

         # plot data
         group = []
         for l in set(labels):
             group.append(diff_coords[labels == l, :])

         # combined plot
         fig = plt.figure(figsize = (7,7))
         ax = fig.add_subplot(111)
         for l, col in zip(set(labels), ('k','r','g','b','y','m','c')):
             curgroup = group[l]
             xs = curgroup[:, 0]
             ys = curgroup[:, 1]
             ax.scatter(xs, ys, label = 'stage {0}'.format(stage_names[l]), c = col)

         ax.set_title('Diffusion map (unprocessed data)');
         ax.legend();
```



1.2.3 Data comparison between 4.5 and 4.6

The branching action is still visible in the diffusion map for 4.6, but is less clear than in the diffusion map for 4.5, while the 1-cell data cluster seems to have no relation to the rest of the data.

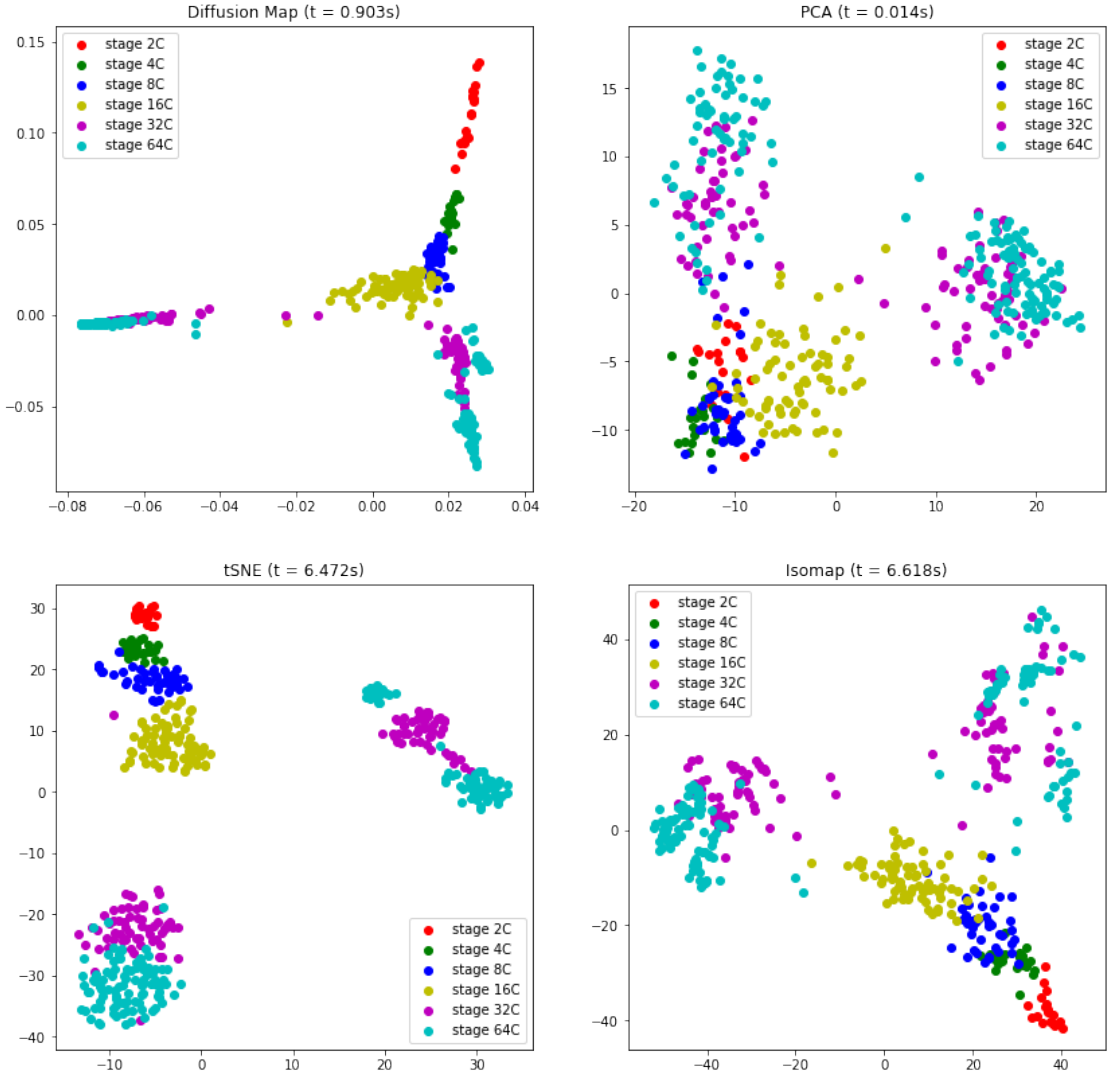
1.2.4 Comparison with other dimensionality reduction methods

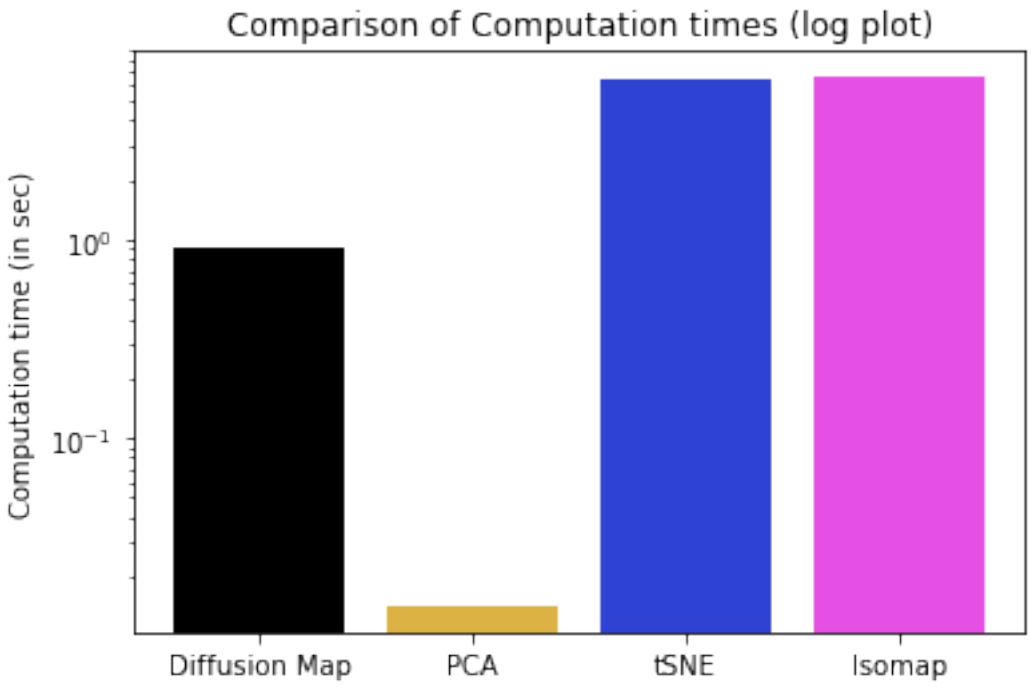
Task 7: Compare diffusion maps with two other methods.

```
In [15]: # load preprocessed data
         data, stage_names, labels = load_guo_data()

         # Methods we compare
         methods = ["Diffusion Map", "PCA", "tSNE", "Isomap"]
         # Dimension of the embedding
```

```
dim = 2

transformed, times = [], []
start = time.monotonic()
transformed.append(DiffMap(n_components = dim, sigma = 10).fit_transform(data))
times.append(time.monotonic() - start)
start = time.monotonic()
transformed.append(PCA(n_components = dim).fit_transform(data))
times.append(time.monotonic() - start)
start = time.monotonic()
transformed.append(TSNE(n_components = dim).fit_transform(data))
times.append(time.monotonic() - start)
transformed.append(Isomap(n_neighbors = 30, n_components = dim).fit_transform(data))
times.append(time.monotonic() - start)
```

In [16]:
```
# Plot the results (2 plots per row)
fig, ax = plt.subplots(2, int(np.ceil(len(transformed) / 2)), figsize = (14, 14))
for i, t in enumerate(transformed):
    group = []
    for l in set(labels):
        group.append(t[labels == l, :])
    for l, col in zip(set(labels), ('r','g','b','y','m','c')):
        curgroup = group[l]
        xs = curgroup[:, 0]
        ys = curgroup[:, 1]
        ax[i // 2, i % 2].scatter(xs, ys, label = 'stage {0}'.format(stage_names[l]), c
    ax[i // 2, i % 2].set_title(methods[i] + " (t = " + repr(round(times[i], 3)) + "s)"
    ax[i // 2, i % 2].legend()
plt.show()

fig, ax = plt.subplots()
ax.bar(range(len(times)), times, color = ['#000000', '#dbb243', '#2e42d3', '#e54fe3'])
ax.set_xticks(range(len(methods)))
ax.set_xticklabels(methods)
ax.set_yscale('log')
ax.set_title("Comparison of Computation times (log plot)")
plt.ylabel("Computation time (in sec)")
plt.show()
```


Diffusion Map (t = 0.903s)
PCA (t = 0.014s)
tSNE (t = 6.472s)
Isomap (t = 6.618s)
stage 2C
stage 4C
stage 8C
stage 16C
stage 32C
stage 64C



1.2.5 Observation

While all dimensionality reduction methods give results that evince some sort of "branching" structure in the data, in the PCA result it is the least obvious (and without the other results it would have likely been missed). The other three methods give relatively clear representation of this "branching" structure, though we would argue that it is most obvious in the tSNE result, then the Isomap result, and then DiffusionMap.

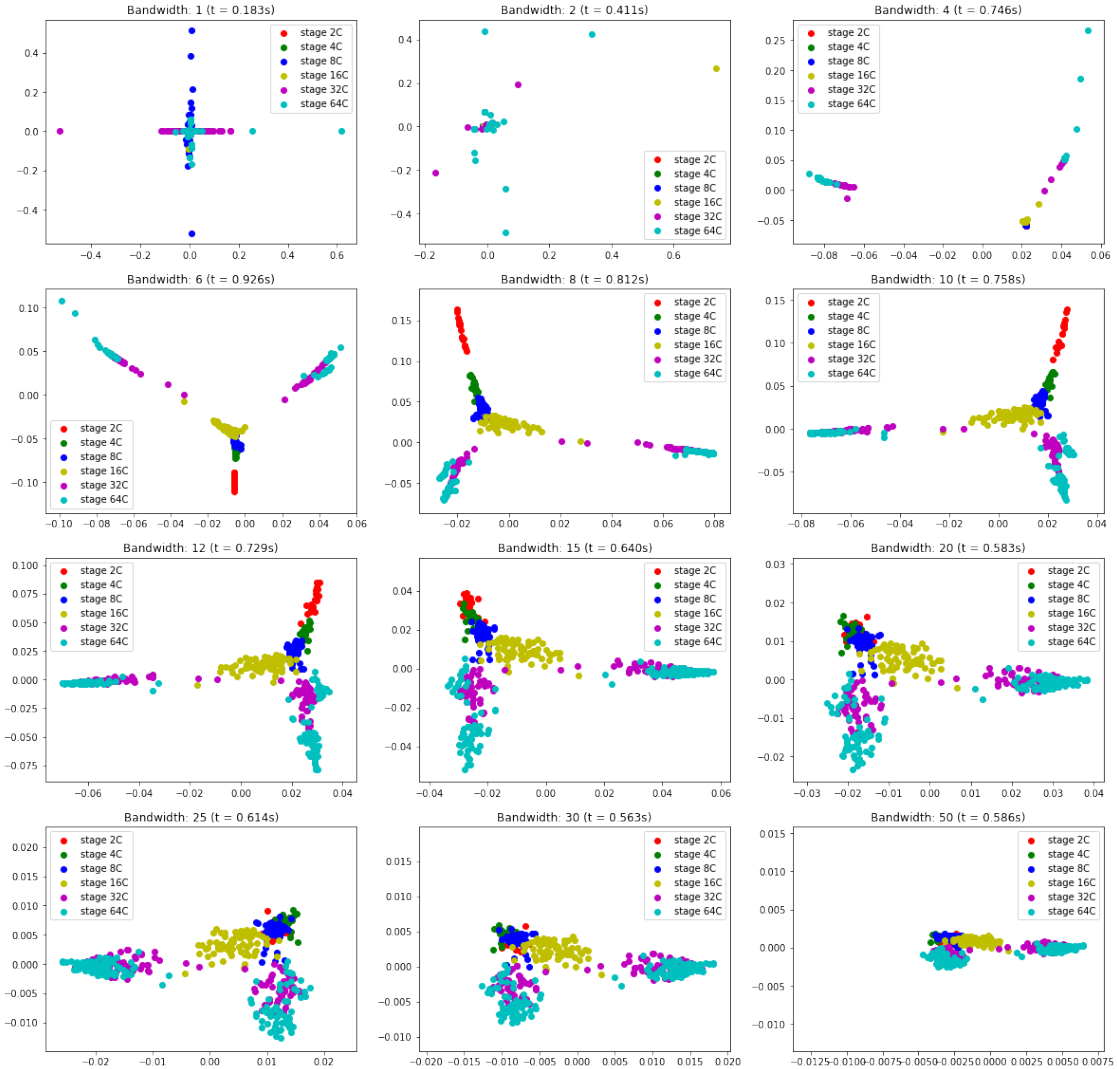
In light of the computation times, however, DiffusionMap appears to give a good tradeoff between a good visualisation of higher-dimensional data and quick results (at least for this data set).

1.2.6 Parameter selection

Task 8: Bandwidth comparison.

```
In [17]: # load preprocessed data
         data, stage_names, labels = load_guo_data()

         # Bandwidths we'd like to plot
         bandwidths = np.array([1, 2, 4, 6, 8, 10, 12, 15, 20, 25, 30, 50])

         transformed, times = [], []
         for b in bandwidths:
             start = time.monotonic()
             transformed.append(DiffMap(n_components = 2, sigma = b).fit_transform(data))
             times.append(time.monotonic() - start)
```

In [18]:
```
# Plot the corresponding embeddings (3 plots per row)
rows = int(np.ceil(len(transformed) / 3))
columns = int(np.ceil(len(transformed) / rows))
fig, ax = plt.subplots(rows, columns, figsize = (20, 20))
for i, t in enumerate(transformed):
    group = []
    for l in set(labels):
        group.append(t[labels == l, :])
    for l, col in zip(set(labels), ('r','g','b','y','m','c')):
        curgroup = group[l]
        xs = curgroup[:, 0]
        ys = curgroup[:, 1]
        ax[i // columns, i % columns].scatter(xs, ys, label = 'stage {0}'.format(stage_
    ax[i // columns, i % columns].set_title("Bandwidth: {0} (t = {1:0.3f}s)".format(ban
    ax[i // columns, i % columns].legend();
```



1.2.7 Observation

We observe that the greater the bandwidth, the more packed the datapoints appear to be in the diffusion space, i.e. the diffusion distances between all datapoints decrease as the bandwidth increases.

Let $n := \#\mathcal{X}$. The Gaussian kernel is

$$K(\mathbf{x}, \mathbf{y}) = \exp\left(-\frac{\|\mathbf{x} - \mathbf{y}\|^2}{2\sigma^2}\right)$$

where σ is the bandwidth. It tends to 1 from the left, monotone, as $\sigma \to +\infty$. The density-normalised Gaussian kernel is

$$K^{(\alpha)}(\mathbf{x}, \mathbf{y}) = \frac{K(\mathbf{x}, \mathbf{y})}{q^{(\alpha)}(\mathbf{x}) q^{(\alpha)}(\mathbf{y})}$$

which tends to $1/n^2$ when we choose $\alpha = 1$ (which we do throughout). We also see that $D^{(\alpha)}(\mathbf{x}) = \sum_{\mathbf{z} \in \mathcal{X}} K^{(\alpha)}(\mathbf{z}, \mathbf{x})$ tends to $1/n$, and hence $P(\mathbf{x}, \mathbf{y}) = K^{(\alpha)}(\mathbf{x}, \mathbf{y}) / D^{(\alpha)}(\mathbf{x}) \to 1/n$. Now, the mean of a Markov matrix is simply the reciprocal of the number data points, so remains constant across different σ. On the other hand, we have just seen that the entries of P tend toward $1/n$, so that the variance of the entries of P goes to 0. Finally, since the stationary distribution $\pi(\mathbf{z})$ may be explicitly computed by

$$\pi(\mathbf{z}) = \frac{D^{(\alpha)}(\mathbf{z})}{\sum_{w \in \mathcal{X}} D^{(\alpha)}(\mathbf{w})}$$

we see that $1/\pi(z) \to 1/n$. Since the diffusion distance $(t = 1)$ is

$$D^2(\mathbf{x}, \mathbf{y}) = \sum_{z \in \mathcal{X}} (P(\mathbf{x}, \mathbf{z}) - P(\mathbf{y}, \mathbf{z}))^2 \frac{1}{\pi(\mathbf{z})},$$

we conclude that the diffusion distances (on average) tend to zero with increasing σ.

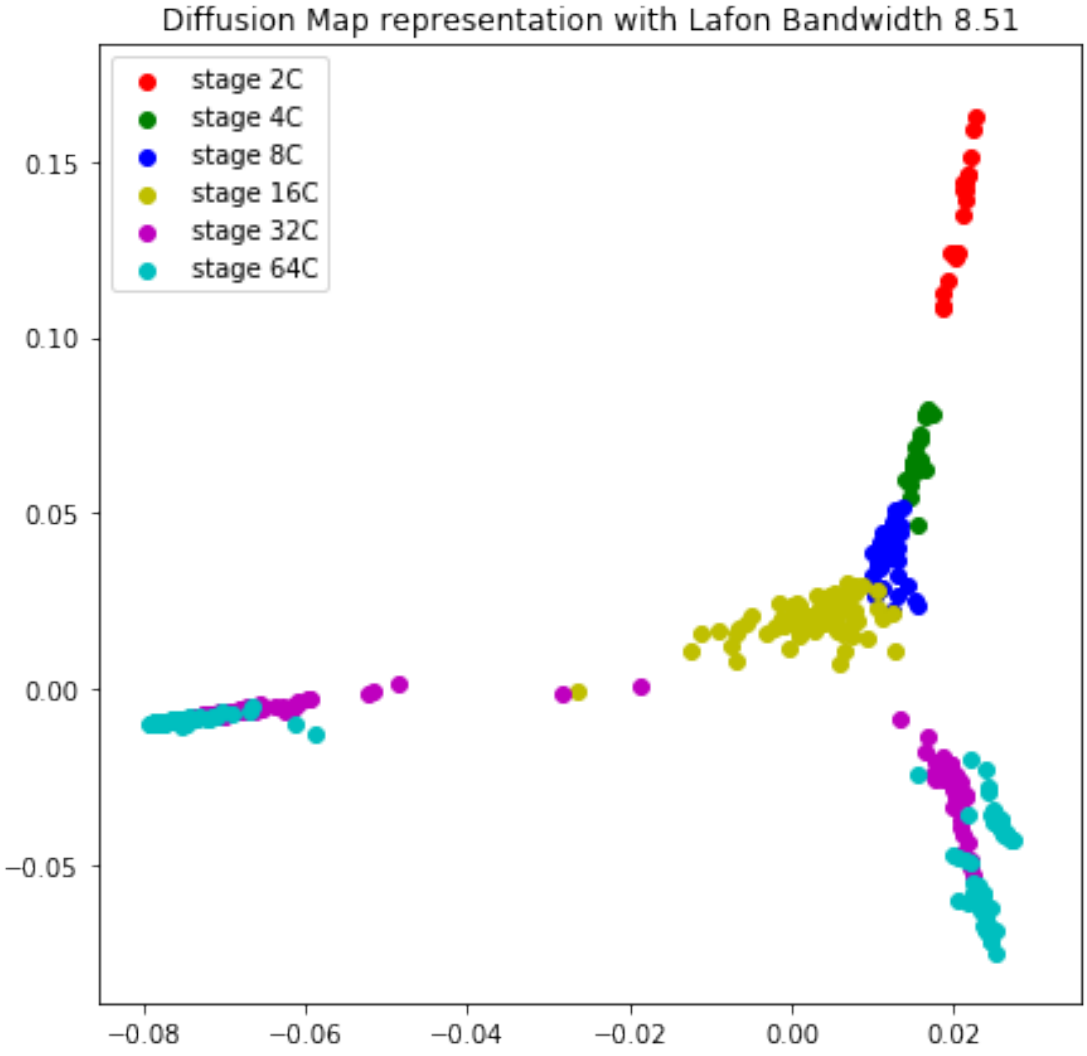
Looking at the plots one can further observe that since for a too low bandwidth the gaussian kernel K(x,y) will be small for all x and y, transitions to other points are unlikely, i.e. the diffusion distance is high. For a too high bandwidth transitions between all points will be likely, i.e. the clusters are well connected and the diffusion distance does not reflect the clusters anymore. Then one can see that there is a range in between those two extrema, where transitions between the clusters are likely but jumps to distant clusters are unlikely, i.e. the local geometry of the manifold is well represented.

Task 9: Implement the rule for σ and plot the embedding with the σ chosen by this rule.

```
In [19]: # load preprocessed data
         data, stage_names, labels = load_guo_data()

         # Calculate Lafon bandwidth
         D = distance_matrix(data, data)**2
         lafon_bandwidth = np.sqrt(1/(2.*D.shape[0])*np.sum(np.amin(D + (np.max(D) + 1)*np.eye(D

         print("Lafon bandwidth: " + repr(lafon_bandwidth))

         transformed = DiffMap(n_components = 2, sigma = lafon_bandwidth).fit_transform(data)
```

```
Lafon bandwidth: 8.508762822655022
```

```
In [20]: # Plot the embedded data corresponding to the Lafon bandwidth
         fig, ax = plt.subplots(figsize = (7, 7))
         group = []
         for l in set(labels):
             group.append(transformed[labels == l, :])
         for l, col in zip(set(labels), ('r','g','b','y','m','c')):
             curgroup = group[l]
             xs = curgroup[:, 0]
             ys = curgroup[:, 1]
             ax.scatter(xs, ys, label = 'stage {0}'.format(stage_names[l]), c = col)
         ax.set_title("Diffusion Map representation with Lafon Bandwidth {0:0.2f}".format(lafon_
         ax.legend();
```



1.2.8 Cell group detection

Now, we want to apply spectral clustering to detect cell groups in the single-cell data.

Task 10: Implement the spectral clustering algorithm using k-means with Λ as input.

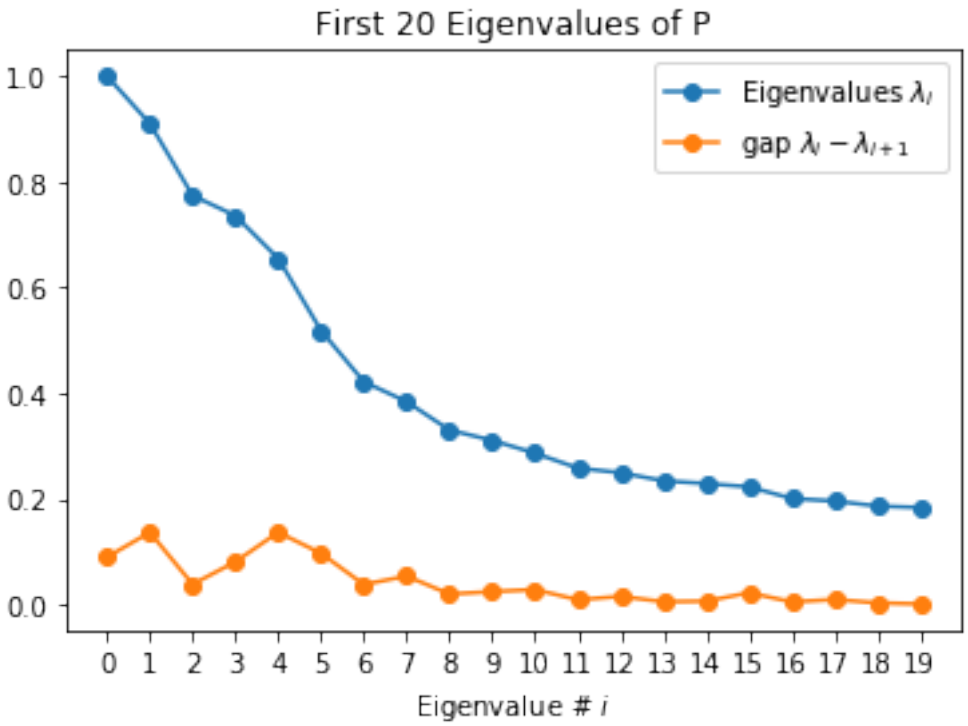
```
In [21]: class SpectralClustering(DiffMap): # Probably not the neatest solution, but we like to
             """
             Class Spectral Clustering

             Parameters
             ----------
             n_clusters: int, optional, default = 8
                 number of clusters that the data will be asigned to
             sigma: optional, default = 10
                 bandwidth of the Gaussian kernel (used for the similarity matrix)
             alpha: optional, default = 1
                 the density rescaling parameter (used for the similarity matrix)
             """
             def __init__(self, n_clusters = 8, sigma = 10, alpha = 1):
                 assert(n_clusters > 0)
                 DiffMap.__init__(self, 1, sigma, alpha)
                 self.n_clusters = n_clusters

             def fit_predict(self, X):
                 # Fits the model and asigns the data X to the clusters
                 self.fit(X)
                 evecs = self.getEigens()[1]
                 return KMeans(n_clusters = self.n_clusters).fit_predict(evecs[:, 0:self.n_clust
```

Task 11: Plot the first 20 eigenvalues of transition matrix P for the Guo data and identify Λ.

```
In [22]: # load preprocessed data
         data, stage_names, labels = load_guo_data()

         evals = DiffMap().fit(data).getEigens()[0]
         Lambda = np.argmax(evals[0:20] - evals[1:21]) + 1 # Lambda is the largest spectral gap

         # Plot the eigenvalues and corresponding spectral gaps
         fig, ax = plt.subplots()
         ax.plot(evals[0:20], marker = "o", label = 'Eigenvalues $\lambda_i$')
         ax.plot(evals[0:20] - evals[1:21], marker = "o", label = 'gap $\lambda_{i} - \lambda_{i
         ax.set_title("First 20 Eigenvalues of P")
         ax.set_xlabel("Eigenvalue # $i$")
         ax.set_xticks(np.arange(20))
         ax.legend()

         Latex("$\Lambda = {1}$; Largest spectral gap is $\lambda_{0} - \lambda_{1}$".format(Lam
```

Out[22]:

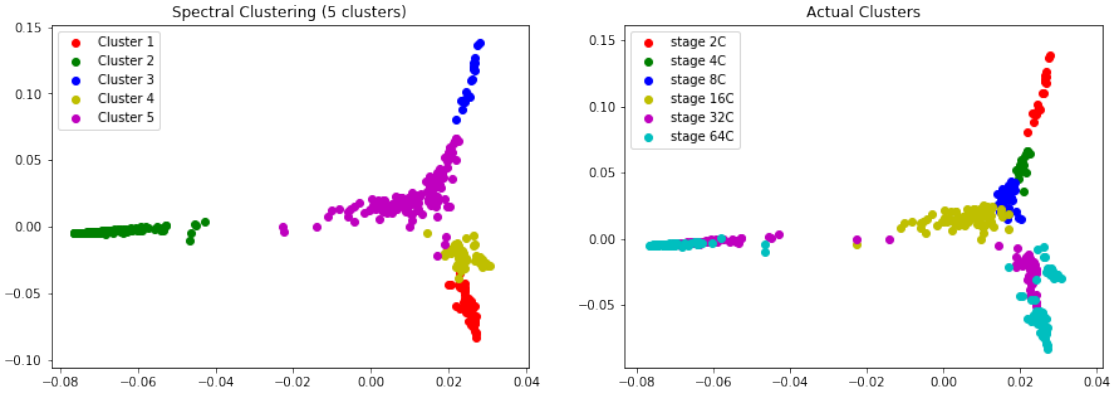
$\Lambda = 5$; Largest spectral gap is $\lambda_4 - \lambda_5$



Task 12: Perform the spectral clustering algorithm for the Guo data.

```
In [23]: n_clusters = Lambda # maximal spectral gap (determined above)

         diff_coords = DiffMap().fit_transform(data)

         clustering = SpectralClustering(n_clusters = n_clusters).fit_predict(data)

In [24]: fig, ax = plt.subplots(1, 2, figsize = (15,5))

         # Plot the spectral clustering
         group = []
         for l in range(n_clusters):
             group.append(diff_coords[clustering == l, :])
         for l, col in zip(range(n_clusters), ('r','g','b','y','m','c')):
             curgroup = group[l]
             xs = curgroup[:, 0]
             ys = curgroup[:, 1]
             ax[0].scatter(xs, ys, label = "Cluster " + repr(l+1), c = col)
         ax[0].set_title("Spectral Clustering (" + repr(n_clusters) + " clusters)")
```

```
ax[0].legend()

# Plot the actual clusters as comparison
group = []
for l in set(labels):
    group.append(diff_coords[labels == l, :])
for l, col in zip(set(labels), ('r','g','b','y','m','c')):
    curgroup = group[l]
    xs = curgroup[:, 0]
    ys = curgroup[:, 1]
    ax[1].scatter(xs, ys, label = 'stage {0}'.format(stage_names[l]), c = col)
ax[1].set_title('Actual Clusters')
ax[1].legend();
```



1.2.9 Observations

We first of all observe that the number of clusters determined by the maximal spectral gap is good, but not perfect (one off), probably since stage 32C and stage 64C have a lot of overlapping.

As per the spectral clustering algorithm, it returns the result of performing k-means on the eigenvectors $\{\phi_i\}_{i=0}^{\Lambda-1}$ (the first $\Lambda = 5$ eigenvectors). This is to say that k-means works in $\mathbb{R}^5$, which explains why the result does not appear as if the k-means algorithm worked on the data presented in the diagram.

The separation by the spectral algorithm of two clusters in the bottom right corner, in particular, likely points to the fact that in $\mathbb{R}^5$ there is greater distance between those clusters than appears in the diagram (in $\mathbb{R}^2$).