Problem 1

a)

$s(\theta, \phi) = (\cos\theta\cos\phi, \sin\theta\cos\phi, \sin\phi)$

$\frac{\partial s}{\partial \theta} = (-\sin\theta\cos\phi, \cos\theta\cos\phi, 0)$

$\frac{\partial s}{\partial \phi} = (\cos\theta(-\sin\phi), \sin\theta(-\sin\phi), \cos\phi)$

b)

Matrix g is going to be defined by calculating the dot product between the following vectors:

$$g = \begin{bmatrix} \vec{v_i} \cdot \vec{v_i} & \vec{v_i} \cdot \vec{v_j} \\ \vec{v_j} \cdot \vec{v_i} & \vec{v_j} \cdot \vec{v_j} \end{bmatrix}$$

where $\vec{v_i}$ and $\vec{v_j}$ are the following:

$\vec{v_i} = \frac{\partial s}{\partial \theta} = (-\sin\theta\cos\phi, \cos\theta\cos\phi, 0)$

$\vec{v_j} = \frac{\partial s}{\partial \phi} = (\cos\theta(-\sin\phi), \sin\theta(-\sin\phi), \cos\phi)$

Therefore, we have to calculate the following:

$$\begin{aligned} \vec{v_i} \cdot \vec{v_i} &= (-sin\theta)cos\phi(-sin\theta)cos\phi + cos\theta cos\phi cos\theta cos\phi + 0.0 \\ &= sin^2\theta cos^2\phi + cos^2\theta cos^2\phi + 0 \\ &= cos^2\phi(sin^2\theta + cos^2\theta) \\ &= cos^2\phi.1 \\ &= cos^2\phi \end{aligned}$$

$$\begin{aligned} \vec{v_j} \cdot \vec{v_j} &= cos\theta(-sin\phi)cos\theta(-sin\phi) + sin\theta(-sin\phi)sin\theta(-sin\phi) + cos\phi cos\phi \\ &= cos^2\theta sin^2\phi + sin^2\theta sin^2\phi + cos^2\phi \\ &= sin^2\phi(sin^2\theta + cos^{\theta}) + cos^2\phi \\ &= sin^2\phi + cos^2\phi \\ &= 1 \end{aligned}$$

$$\begin{aligned} \vec{v_i} \cdot \vec{v_j} = \vec{v_j} \cdot \vec{v_i} &= (-sin\theta)cos\phi cos\theta(-sin\phi) + cos\theta cos\phi sin\theta(-sin\phi) + 0cos\phi \\ &= sin\theta cos\phi cos\theta sin\phi + -(sin\theta cos\phi cos\theta sin\phi) + 0 \\ &= 0 \end{aligned}$$

After calculating the entries of matrix g, we can now replace the values:

$$g = \begin{bmatrix} cos^2\phi & 0 \\ 0 & 1 \end{bmatrix}$$

c)

To get g^{-1} we first need to get the determinant of g:

$$
\begin{aligned}
det(g) &= \frac{1}{ad - bc} \\
&= \frac{1}{cos^{\phi}.1 - 0.0} \\
&= \frac{1}{cos^{\phi}}
\end{aligned}
$$

Now we can multiply the $det(g)$ by matrix g to obtaing g^{-1}:

$$
\begin{aligned}
g^{-1} &= det(g).\begin{bmatrix} 1 & -0 \\ -0 & cos^2\phi \end{bmatrix} \\
&= \frac{1}{cos^{\phi}}\begin{bmatrix} 1 & 0 \\ 0 & cos^2\phi \end{bmatrix} \\
&= \begin{bmatrix} \frac{1}{cos^2\phi} & 0 \\ 0 & 1 \end{bmatrix}
\end{aligned}
$$

d)

Before computing the formulas for the Christoffel symbols, we need to specify the following relationship:

$x^1 = \theta$

$x^2 = \phi$

For the unit sphere, the summation goes from 1 to 2 because we have two variables (θ and ϕ), therefore we have:

$$
\begin{aligned}
\Gamma_{ij}^{k} &= \frac{1}{2}\sum_{l=1}^{n} g^{kl}\left(\frac{\partial g_{jl}}{\partial x^i} + \frac{\partial g_{il}}{\partial x^j} - \frac{\partial g_{ij}}{\partial x^l}\right) \\
&= \frac{1}{2}\left[g^{k1}\left(\frac{\partial g_{j1}}{\partial x^i} + \frac{\partial g_{i1}}{\partial x^j} - \frac{\partial g_{ij}}{\partial x^1}\right) + g^{k2}\left(\frac{\partial g_{j2}}{\partial x^i} + \frac{\partial g_{i2}}{\partial x^j} - \frac{\partial g_{ij}}{\partial x^2}\right)\right]
\end{aligned}
$$

From all partial derivatives, we know that only 2 are not going to be 0, and those are:

$\frac{\partial g_{11}}{\partial x^2} = -sin(2\phi)$

$\frac{\partial g^{11}}{\partial x^2} = 2tan\phi sec^2\phi$

Now we are going to iterate over i, j and k to get all the Christoffel symbols and replace values using matrix g, g^{-1} and the partial derivatives calculated above:

$$
\begin{aligned}
\Gamma^1_{11} &= \frac{1}{2}\sum_{l=1}^{n} g^{1l}\left(\frac{\partial g_{1l}}{\partial x^1} + \frac{\partial g_{1l}}{\partial x^1} - \frac{\partial g_{11}}{\partial x^l}\right) \\
&= \frac{1}{2}\left[g^{11}\left(\frac{\partial g_{11}}{\partial x^1} + \frac{\partial g_{11}}{\partial x^1} - \frac{\partial g_{11}}{\partial x^1}\right) + g^{12}\left(\frac{\partial g_{12}}{\partial x^1} + \frac{\partial g_{12}}{\partial x^1} - \frac{\partial g_{11}}{\partial x^2}\right)\right] \\
&= \frac{1}{2}\left[\frac{1}{cos^2\phi}\left(0 + 0 - 0\right) + 0\right] \\
&= 0 \\
\Gamma^2_{11} &= \frac{1}{2}\sum_{l=1}^{n} g^{2l}\left(\frac{\partial g_{1l}}{\partial x^1} + \frac{\partial g_{1l}}{\partial x^1} - \frac{\partial g_{11}}{\partial x^l}\right) \\
&= \frac{1}{2}\left[g^{21}\left(\frac{\partial g_{11}}{\partial x^1} + \frac{\partial g_{11}}{\partial x^1} - \frac{\partial g_{11}}{\partial x^1}\right) + g^{22}\left(\frac{\partial g_{12}}{\partial x^1} + \frac{\partial g_{12}}{\partial x^1} - \frac{\partial g_{11}}{\partial x^2}\right)\right] \\
&= \frac{1}{2}\left[0 + 1\left(0 + 0 - -sin(2\phi)\right)\right] \\
&= \frac{sin(2\phi)}{2} \\
&= \frac{2sin\phi cos\phi}{2} \\
&= sin\phi cos\phi \\
\Gamma^1_{12} &= \frac{1}{2}\sum_{l=1}^{n} g^{1l}\left(\frac{\partial g_{2l}}{\partial x^1} + \frac{\partial g_{1l}}{\partial x^2} - \frac{\partial g_{12}}{\partial x^l}\right) \\
&= \frac{1}{2}\left[g^{11}\left(\frac{\partial g_{21}}{\partial x^1} + \frac{\partial g_{11}}{\partial x^2} - \frac{\partial g_{12}}{\partial x^1}\right) + g^{12}\left(\frac{\partial g_{22}}{\partial x^1} + \frac{\partial g_{12}}{\partial x^2} - \frac{\partial g_{12}}{\partial x^2}\right)\right] \\
&= \frac{1}{2}\left[\frac{1}{cos^2\phi}\left(0 + -sin(2\phi) - 0\right) + 0\right] \\
&= \frac{1}{2}\left[\frac{-sin(2\phi)}{cos^2\phi}\right] \\
&= \frac{-sin(2\phi)}{2cos^2\phi} \\
&= \frac{-(2sin\phi cos\phi)}{2cos\phi cos\phi} \\
&= \frac{-sin\phi}{cos\phi} \\
&= -tg\phi \\
\Gamma^2_{12} &= \frac{1}{2}\sum_{l=1}^{n} g^{2l}\left(\frac{\partial g_{2l}}{\partial x^1} + \frac{\partial g_{1l}}{\partial x^2} - \frac{\partial g_{12}}{\partial x^l}\right) \\
&= \frac{1}{2}\left[g^{21}\left(\frac{\partial g_{21}}{\partial x^1} + \frac{\partial g_{11}}{\partial x^2} - \frac{\partial g_{12}}{\partial x^1}\right) + g^{22}\left(\frac{\partial g_{22}}{\partial x^1} + \frac{\partial g_{12}}{\partial x^2} - \frac{\partial g_{12}}{\partial x^2}\right)\right] \\
&= \frac{1}{2}\left[0 + 1\left(0 + 0 - 0\right)\right] \\
&= 0
\end{aligned}
$$

$$
\begin{aligned}
\Gamma^1_{21} &= \frac{1}{2}\sum_{l=1}^{n} g^{1l}\left(\frac{\partial g_{1l}}{\partial x^2}+\frac{\partial g_{2l}}{\partial x^1}-\frac{\partial g_{21}}{\partial x^l}\right)\\
&= \frac{1}{2}\left[g^{11}\left(\frac{\partial g_{11}}{\partial x^2}+\frac{\partial g_{21}}{\partial x^1}-\frac{\partial g_{21}}{\partial x^1}\right)+g^{12}\left(\frac{\partial g_{12}}{\partial x^2}+\frac{\partial g_{22}}{\partial x^1}-\frac{\partial g_{21}}{\partial x^2}\right)\right]\\
&= \frac{1}{2}\left[\frac{1}{cos^2\phi}\left(-sin(2\phi)+0-0\right)+0\right]\\
&= \frac{1}{2}\left[\frac{1}{cos^2\phi}\left(0+-sin(2\phi)-0\right)+0\right]\\
&= \frac{1}{2}\left[\frac{-sin(2\phi)}{cos^2\phi}\right]\\
&= \frac{-sin(2\phi)}{2cos^2\phi}\\
&= \frac{-(2sin\phi cos\phi)}{2cos\phi cos\phi}\\
&= \frac{-sin\phi}{cos\phi}\\
&= -tg\phi
\end{aligned}
$$

$$
\begin{aligned}
\Gamma^2_{21} &= \frac{1}{2}\sum_{l=1}^{n} g^{2l}\left(\frac{\partial g_{1l}}{\partial x^2}+\frac{\partial g_{2l}}{\partial x^1}-\frac{\partial g_{21}}{\partial x^l}\right)\\
&= \frac{1}{2}\left[g^{21}\left(\frac{\partial g_{11}}{\partial x^2}+\frac{\partial g_{21}}{\partial x^1}-\frac{\partial g_{21}}{\partial x^1}\right)+g^{22}\left(\frac{\partial g_{12}}{\partial x^2}+\frac{\partial g_{22}}{\partial x^1}-\frac{\partial g_{21}}{\partial x^2}\right)\right]\\
&= \frac{1}{2}\left[0+1\left(0+0-0\right)\right]\\
&= 0
\end{aligned}
$$

$$
\begin{aligned}
\Gamma^1_{22} &= \frac{1}{2}\sum_{l=1}^{n} g^{1l}\left(\frac{\partial g_{2l}}{\partial x^2}+\frac{\partial g_{2l}}{\partial x^2}-\frac{\partial g_{22}}{\partial x^l}\right)\\
&= \frac{1}{2}\left[g^{11}\left(\frac{\partial g_{21}}{\partial x^2}+\frac{\partial g_{21}}{\partial x^2}-\frac{\partial g_{22}}{\partial x^1}\right)+g^{12}\left(\frac{\partial g_{22}}{\partial x^2}+\frac{\partial g_{22}}{\partial x^2}-\frac{\partial g_{22}}{\partial x^2}\right)\right]\\
&= \frac{1}{2}\left[\frac{1}{cos^2\phi}\left(0+0-0\right)+0\right]\\
&= 0
\end{aligned}
$$

$$
\begin{aligned}
\Gamma^2_{22} &= \frac{1}{2}\sum_{l=1}^{n} g^{2l}\left(\frac{\partial g_{2l}}{\partial x^2}+\frac{\partial g_{2l}}{\partial x^2}-\frac{\partial g_{22}}{\partial x^l}\right)\\
&= \frac{1}{2}\left[g^{21}\left(\frac{\partial g_{21}}{\partial x^2}+\frac{\partial g_{21}}{\partial x^2}-\frac{\partial g_{22}}{\partial x^1}\right)+g^{22}\left(\frac{\partial g_{22}}{\partial x^2}+\frac{\partial g_{22}}{\partial x^2}-\frac{\partial g_{22}}{\partial x^2}\right)\right]\\
&= \frac{1}{2}\left[0+1\left(0+0-0\right)\right]\\
&= 0
\end{aligned}
$$

e)

To show that the equator is a geodesic, we need to evaluate the second derivative of gamma at the equator ($\gamma(t) = s(t, 0)$) and see if it is equal to zero. That would be:

$$\begin{aligned}
\frac{d^2\gamma^2}{dt^2} &= -\sum_{i,j=1}^{n} \Gamma_{ij}^{k}(\gamma(t)) \frac{d\gamma^i}{dt} \frac{d\gamma^j}{dt} \\
&= -[0 + sin(\phi)cos(\phi) - tg(\phi) + 0 - tg(\phi) + 0 + 0 + 0] \frac{d\gamma^i}{dt} \frac{d\gamma^j}{dt} \\
&= -[sin(0)cos(0) - 2tg(0)] \frac{d\gamma^i}{dt} \frac{d\gamma^j}{dt} \\
&= -[0.1 - 2.0] \frac{d\gamma^i}{dt} \frac{d\gamma^j}{dt} \\
&= -[0] \frac{d\gamma^i}{dt} \frac{d\gamma^j}{dt}
\end{aligned}$$

Since the evaluation of the Christoffel symbols is zero and is multiplying the derivatives of γ^i, γ^j the whole equation will be 0. Therefore the condition that the second derivative of γ has to be equal to cero is satisfied, and we can confirm that the equator is a geodesic.

f)

I just showed that the equator, which is a great circle on S^2 with radius 1, is a geodesic. Now, we can generalize it and say that all great circles on S^2 with radius 1 are geodesics. To prove it, we first need to state the definition of isometry:

"An isometry is a diffeomorphism $\phi : M \to N$ of Riemannian manifolds that preserves the Riemannian metric. That is, if $\langle \cdot, \cdot \rangle_M$ and $\langle \cdot, \cdot \rangle_N$ are the metrics for M and N, respectively, then $\phi^* \langle \cdot, \cdot \rangle_M = \langle \cdot, \cdot \rangle_N$." - Terse Notes on Riemannian Geometry, Tom Fletcher, January 26, 2010

Based on the above definition, we can think of any great circle as an isometry of the equator, that would be the equator with a rotation applied. Furthermore, an isometry preserves the length of the curve, consequently we know that the equator an any other great circle are going to have the same length and since the equator is a smooth curve, the great circles are also going to be smooth.

Having said that, and since the equator is a geodesic, by the definition of isometry, any other great circle on S^2 with radius 1 will be a geodesic.

Problem 2

Note: In this section, there are 4 main sub-sections: 1) Implementations 2) Creation/Load of the Datasets 3) Methods evaluation (Can be skipped, since these are test that I made to see if everything was working well) 4) Experiments

In [ ]:

```
import numpy as np
import math
import matplotlib.pyplot as plt
import random

from scipy.spatial.transform import Rotation as R
```

1 - Implementations

Object class

Class that I am going to use to represent each object of the dataset. The implementations uses matrices of size 2xn, where n is the amount of points the object has in 2d. This class also includes methods to plot the object with matplotlib.

In [ ]:

```python
class Object:
    def __init__(self, points):
        points = np.array(points)
        if points.shape[0] > 2:
            self.matrix = np.transpose(points)
        else:
            self.matrix = points
        self.preshape = self.project_onto_preshape()

    def plot_preshape(self, color="dodgerblue", ax=None, label=""):
        x = self.preshape[0]
        y = self.preshape[1]
        if ax == None:
            # plt.scatter(x,y, color=color)
            plt.fill(x,y, edgecolor=color, fill=False, label=label)
        else:
            # ax.scatter(x,y, color=color)
            ax.fill(x,y, edgecolor=color, fill=False, label=label)

    def plot_matrix(self, color="dodgerblue", ax=None, label=""):
        x = self.matrix[0]
        y = self.matrix[1]
        if ax == None:
            # plt.scatter(x,y, color=color)
            plt.fill(x,y, edgecolor=color, fill=False, label=label)
        else:
            # ax.scatter(x,y, color=color)
            ax.fill(x,y, edgecolor=color, fill=False, label=label)

    def project_onto_preshape(self):
        # Remove translation
        rows_mean = np.mean(self.matrix, axis=1)
        first_row = self.matrix[0] - rows_mean[0]
        second_row = self.matrix[1] - rows_mean[1]
        matrix = np.array([first_row, second_row])

        # Remove scale
        frobenius_norm = np.linalg.norm(matrix, 'fro')
        matrix = matrix / frobenius_norm

        return matrix
```

Implementation 1: Project an object onto the preshape sphere

I implemented this as part of my "Object" class so I can execute it when an object of this class is instantiated. It basically does what was stated in the slides, and as indicated with the comments:

- **Remove translation**: subtract the row means from each row.
- **Remove scale**: divide by the Frobenius norm

Implementation 2: Orthogonal Procrustes Analysis (OPA)

I implemented a function for this following the algorithm presented in the slides. The issue that I faced here was that once I get the rotation matrix R, I need to tranpose it before multiplying it

with B to get the aligned version of B. I overcame this issue, because Dr. Fletcher mentioned in class that we may need to transpose the rotation matrix in order to work. I also tested this method in section **3 - Methods evaluation** with the test discussed in class.

In [ ]:
```
def procrustes_analysis(A, B):
    """
        A = target_preshape
        B = moving_preshape
    """
    BAt = np.matmul(B, np.transpose(A))
    U, D, Vt = np.linalg.svd(BAt)
    R = np.matmul(U, Vt)
    R = np.transpose(R) #!!!
    aligned_version = np.matmul(R, B)

    return aligned_version
```

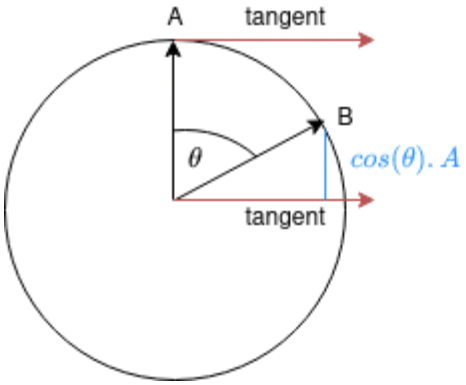
Implementation 3: Exponential map on Kendall shape

I implemented a function for this following the explanation given by Dr. Fletcher in class. I worked very hard trying to implement this method by myself, but it was really difficult to arrive to the equation to get B, before it was explained in class. A side note is that I added an error handling in case the norm of the tangent vector is 0, otherwise it was going to raise a 0 division error.

In [ ]:
```
def exponential_map(A, T):
    norm = np.linalg.norm(T)
    if norm == 0:
        return A
    normalized_T = T / norm
    B = math.cos(norm) * A + math.sin(norm) * normalized_T
    return B
```

Implementation 4: Log map on Kendall shape

I implemented a function for this based on the tips/clues given by Dr. Fletcher in class. It took me a while to figure out what was the right equation for getting the tangent vector, however it was really helpful that professor mentioned in which area, we should focus our trigonometric analysis. Side notes, a) I need to do a workaround np.dot, because for matrices it returned another matrix; b) I need to specify a max of 1 and a min of -1 for theta, because if it was slightly above or below one, I would get a valueError.



$$tangent = \theta * \frac{B - cos(\theta).A}{\|B - cos(\theta).A\|}$$

In [ ]:

```
def log_map(A, B):
    ### Calculate distance between A and B
    # Had to do this because np.dot for matrices return a matrix rather than
    dot_prod = np.sum(A * B)
    norm_A = np.linalg.norm(A)
    norm_B = np.linalg.norm(B)
    theta = dot_prod / (norm_A * norm_B)
    # Had to do this because if theta > 1 | theta < -1 (e.g. +/-1.00000000000
    if theta > 1:
        theta = 1
    if theta < -1:
        theta = -1
    distance = np.arccos(theta)

    ### Get tangent vector
    tangent = B - math.cos(distance) * A
    norm_tangent = np.linalg.norm(tangent)
    if norm_tangent == 0:
        tangent_vector = np.zeros(A.shape)
    else:
        tangent_vector = distance * (tangent / norm_tangent)

    return tangent_vector
```

Implementation 5: Estimate the Fréchet mean using Gradient descent

I implemented a function for this following the algorithm that is in the slides and applying OPA to align the objects before computing the log map as mentioned in class and office hours. My implementation was slightly different from the one in the slides, since I used a "for" loop that iterates over an specified amount of epochs instead of a "while". However I added a print statement inside the method, that will print at which epoch the norm of the gradient is equal to a value very close to zero. In this way you can specify if you want to iterate over the dataset more times to keep decreasing the gradient, although that it may be unnecessary because the gradient norm is already $1E^{-10}$ for my implementation.

In [ ]:

```
def gradient_descent(m_objects, epochs=5):
    N = len(m_objects)
    mu = [m_objects[0]]
    current_mu = mu[0]
    gradient0_reached = False
    for i in range(epochs):
        summation = np.zeros(m_objects[0].preshape.shape)
        for m_object in m_objects:
            # OPA
            aligned_object = Object(procrustes_analysis(current_mu.preshape,
            # Distance
            summation = summation + log_map(current_mu.preshape, aligned_obje
        gradient = summation / N
        new_mu = Object(exponential_map(current_mu.preshape, gradient))
        mu.append(new_mu)
        current_mu = new_mu

        if not gradient0_reached and np.linalg.norm(gradient) < 0.00000000001
            print(f"Gradient is equal to zero in epoch #{i}")
            gradient0_reached = True

    return mu[-1]
```

Implementation 6: Approximation of Principal Geodesic Analysis (PGA)

I implemented a function for this following the algorithm presented in "Statistics on Manifolds" - P. Thomas Fletcher, 2019. First you need to calculate the distance from all objects to the Fréchet mean, but before that, it is needed to align the objects using OPA. This time, instead of storing the distance in a 2 by k matrix, where k is the number of 2d points that the object has, I reshaped it so now it's a vector of 1 by 2k elements. After that, I have the distances from the Fréchet mean to all objects in the dataset, stored in S of shape (100 x n elements x n elements), where n is the number of features of the object. Now I need to sum all elements across the first axis and divide it by the length of the dataset minus 1, getting a square matrix of shape (n elements x n elements). Lastly I used np.linalg.svd to get the eigenvectors and eigenvalues. I used np.linalg.svd instead of np.linalg.eig, because svd returns the eigenvalues ordered.

In [ ]:

```
def PGA(f_mean, dataset):
    S = []
    for d in dataset:
        # OPA
        aligned_d = procrustes_analysis(f_mean.preshape, d.preshape)
        # Distance
        m = log_map(f_mean.preshape, d.preshape)
        m = m.reshape(1,-1)
        s = np.matmul(np.transpose(m), m)
        S.append(s)
    S = np.array(S)
    S = np.sum(S, axis=0) * (1 / (len(dataset) - 1))
    eigenvectors, eigenvalues, _ = np.linalg.svd(S)
    return eigenvectors, eigenvalues
```

Scree-plot

Function to plot the scree-plot in conjuction with the cumulative sum of variance.

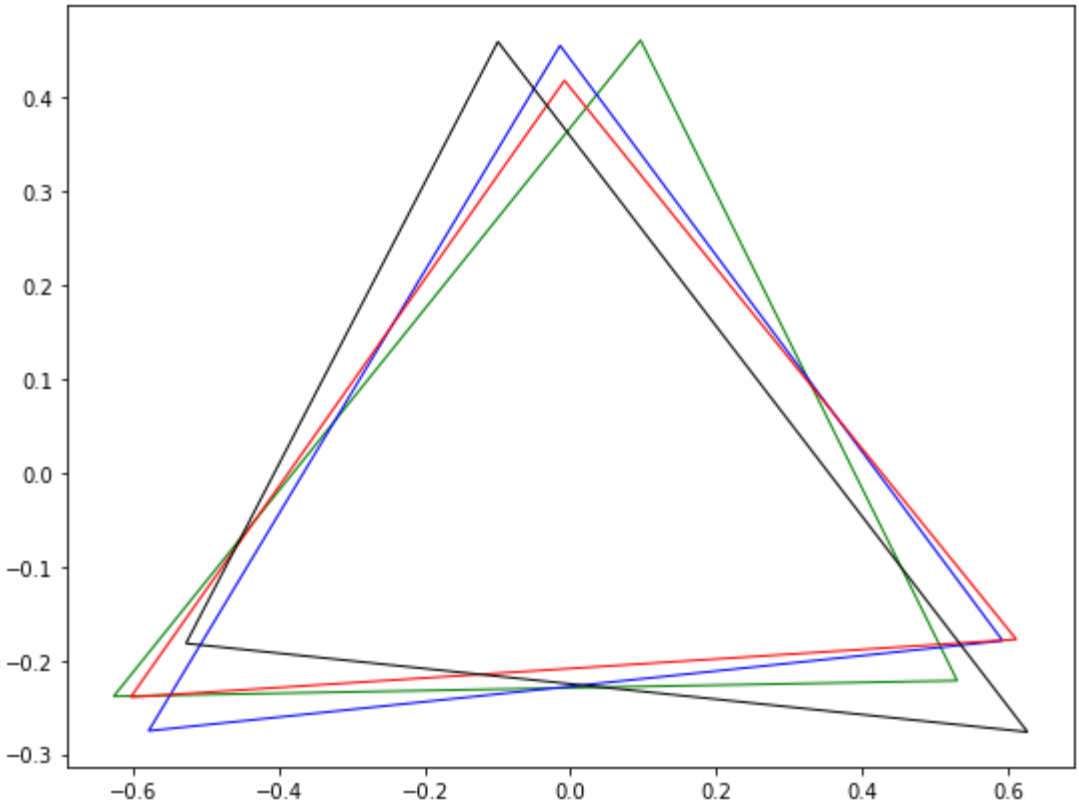
In [ ]:
```
def scree_plot(eigenvalues):
    plt.figure(figsize=(9,7))

    cumulative_val = 0
    total_val = np.sum(eigenvalues)
    c_x = []
    c_y = []
    for e, e_val in enumerate(eigenvalues):
        c_x.append(e+1)
        cumulative_val += e_val
        c_y.append(cumulative_val)
    c_x = np.array(c_x)
    c_y = np.array(c_y) * 100 / total_val
    plt.plot(c_x, c_y, 'ko-', linewidth=1, markersize=2.5, label="Cumulative

    x_vals=np.arange(len(eigenvalues)) + 1
    plt.plot(x_vals, eigenvalues/np.linalg.norm(eigenvalues) * 100, 'o-', lin
    plt.title('Scree Plot')
    plt.xlabel('Principal Component')
    plt.ylabel('Percentage of variance')
    plt.yticks(np.arange(0,101,20))

    plt.plot(c_x, [95] * len(eigenvalues), 'g--', linewidth=2, label="95$\%$
    plt.legend()
    plt.show()
```

2 - Creation/Load of the Datasets

Random Triangles

In [ ]:
```
triangle_dataset = []
for i in range(100):
    s = np.random.normal(1, 0.5, 1)[0]
    p0 = [-1,0]
    p1 = [1,0]
    p2 = [0,s]
    noise = np.random.normal(0, 0.1, (3,2))
    triangle = np.array([p0, p1, p2]) + noise
    triangle_dataset.append(Object(triangle))
```

In [ ]:
```
fig, ax = plt.subplots(1, figsize=(9,7))
colors = ["green", "blue", "red", "black"]
for i in range(4):
    triangle_dataset[i].plot_preshape(colors[i])
```



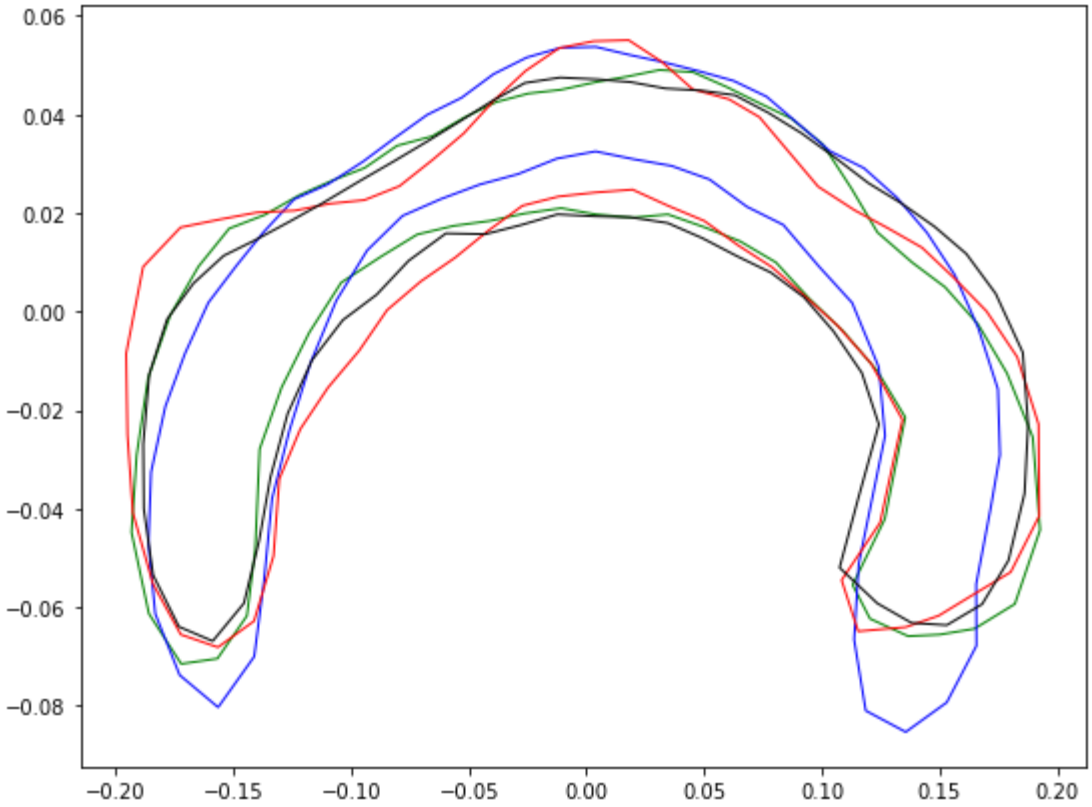
Corpus Callosum

In [ ]:

```
corpus_callosum_dataset = []
for shape_file in os.listdir("./cc-shapes/"):
    corpus_callosum = []
    f = open(f"./cc-shapes/{shape_file}", "r")
    for l in f:
        # Remove '\n'
        l = l.split('\n')[0]
        # Get x and y components
        l = l.split(' ')
        point = [float(l[0]), float(l[1])]
        corpus_callosum.append(point)
    corpus_callosum_dataset.append(Object(corpus_callosum))
```

In [ ]:

```
fig, ax = plt.subplots(1, figsize=(9,7))
colors = ["green", "blue", "red", "black"]
for i in range(4):
    corpus_callosum_dataset[i].plot_preshape(colors[i])
```

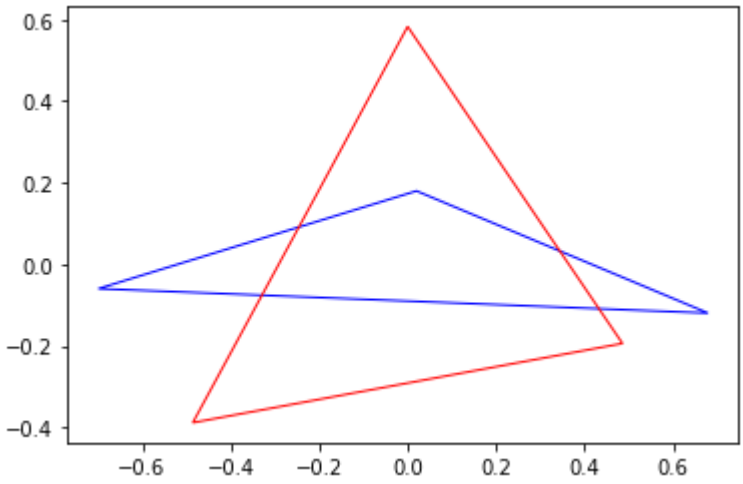


3 - Methods evaluation (Can be skipped)

Preshape

In [ ]:

```
points0 = [[-1,0.1], [1.3,0], [0.2,0.5]]
points1 = [[-0.5,0], [0.5,0.2], [0,1]]

triangle0 = Object(points0)
triangle1 = Object(points1)

triangle0.plot_preshape("blue")
triangle1.plot_preshape("red")
```



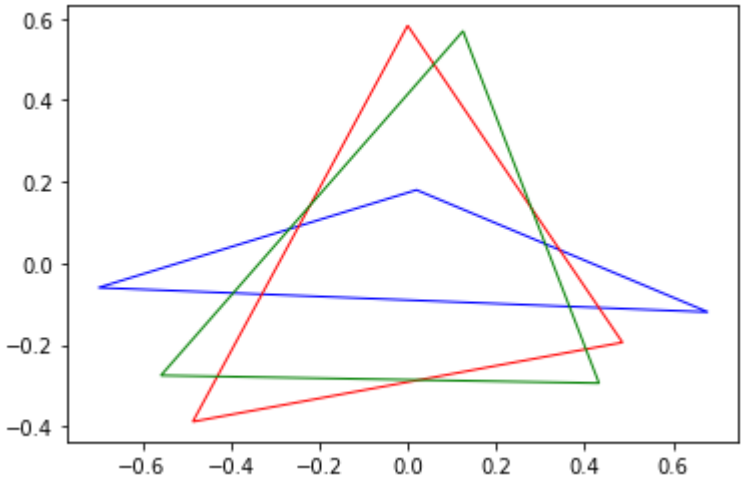
The mass center point of the triangles are aligned to (0,0), meaning that the preshape projection

is working properly.

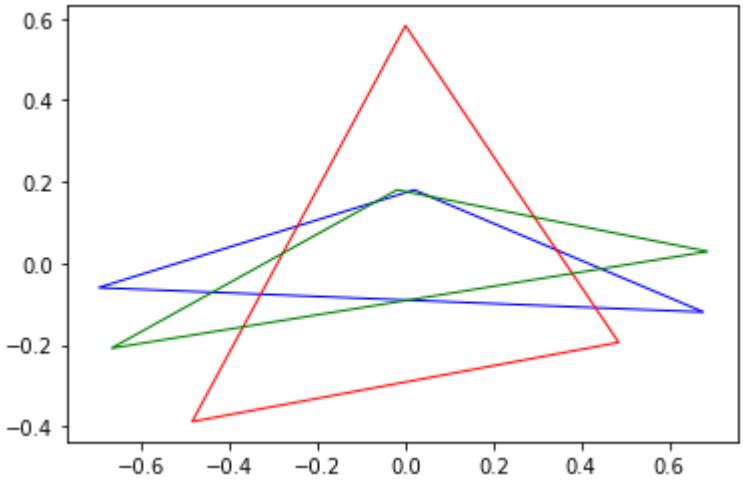
Procrustes analysis

In [ ]:
```
pa0 = Object(procrustes_analysis(triangle0.preshape, triangle1.preshape))
pa1 = Object(procrustes_analysis(triangle1.preshape, triangle0.preshape))
```

In [ ]:
```
triangle0.plot_preshape("blue")
triangle1.plot_preshape("red")
pa0.plot_preshape("green")
```

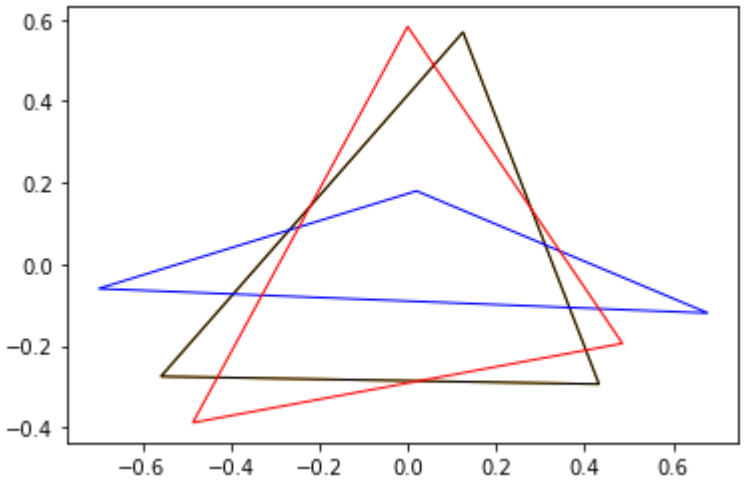


In [ ]:
```
triangle0.plot_preshape("blue")
triangle1.plot_preshape("red")
pa1.plot_preshape("green")
```



After applying OPA, the triangles seem to be aligned, meaning that procrustes_analysis is working properly. Nonetheless, another sanity check that can be done to debug if the OPA is working, is doing the following:

In [ ]:
```
pa0 = Object(procrustes_analysis(triangle0.preshape, triangle1.preshape))
pa1 = Object(procrustes_analysis(pa0.preshape, triangle1.preshape))
pa0.plot_preshape("orange")
pa1.plot_preshape("black")
triangle0.plot_preshape("blue")
triangle1.plot_preshape("red")
```



Essentially you apply OPA to two shapes (shape 1 and shape 2), and then you apply OPA to shape 1 and the already aligned shape 2. The new aligned shape should match which the previous one. That's why we don't see any orange triangle on the plot and we just see the black triangle (second OPA).

Log map

In [ ]:
```
tangent_vector = log_map(triangle0.preshape, triangle1.preshape)
tangent_vector
```

Out[ ]:
```
array([[ 0.09337812, -0.07593929, -0.01743883],
       [-0.36126821, -0.10215938,  0.46342758]])
```

In [ ]:
```
# Had to do this because np.dot for matrices return a matrix rather than a sc
np.sum(triangle0.preshape * tangent_vector).round(10)
```

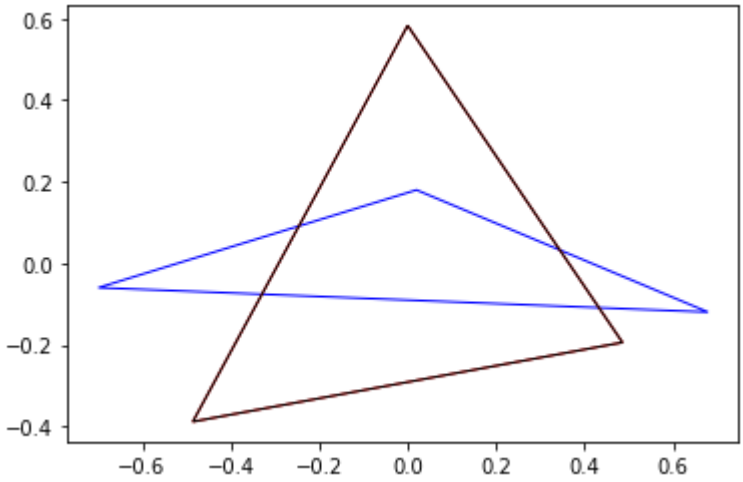
Out[ ]:
```
0.0
```

The dot product between a preshape and its tangent vector generated by log_map should be 0. The result of the dot product between the first triangle and the tangent vector is ~ 0, which means the the implementation of log_map is working properly.

Exponential map

In [ ]:

```
# Get tangent vector using log_map
tangent_vector = log_map(triangle0.preshape, triangle1.preshape)
# Get object using exponential_map
recovered_triangle1 = Object(exponential_map(triangle0.preshape, tangent_vect

triangle0.plot_preshape("blue")
triangle1.plot_preshape("red")
recovered_triangle1.plot_preshape("black")
```



It can be seen that the original triangle 1 (red) and the recovered triangle 1 (black) match in the above image, after applying:

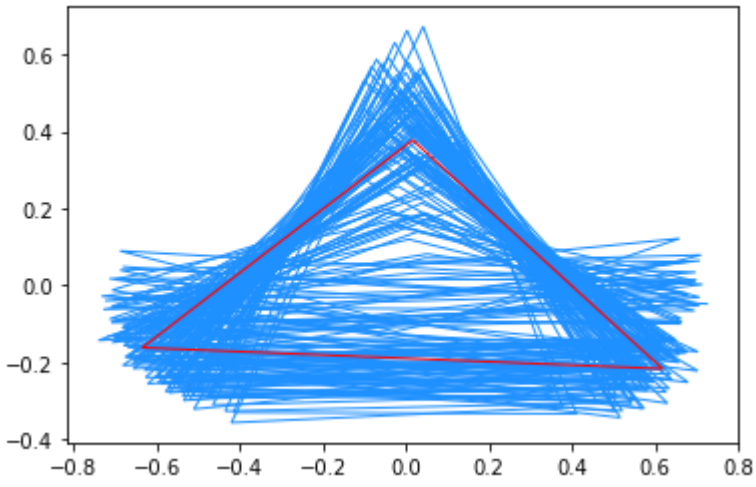
$exponentialMap(triangle_0, logMap(triangle_0, triangle_1))$

Gradient descent

In [ ]:

```
f_mean_t = gradient_descent(triangle_dataset, epochs=10)
for t in triangle_dataset:
    t.plot_preshape()
f_mean_t.plot_preshape("red")
```

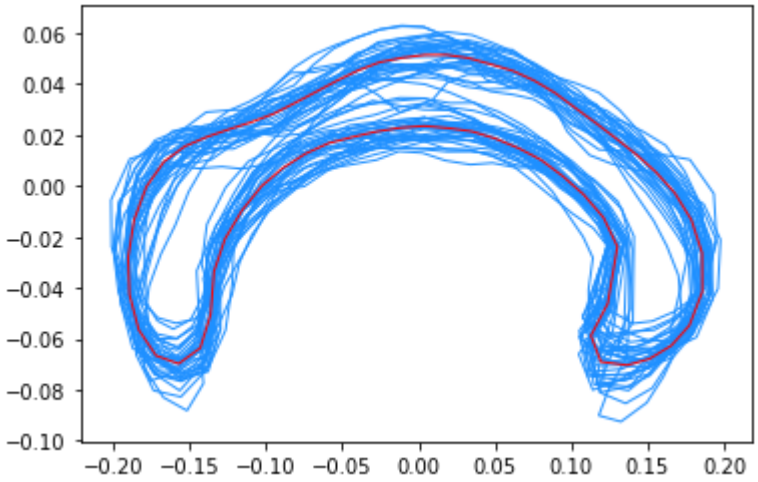
```
Gradient is equal to zero in epoch #9
```



In [ ]:

```
f_mean_cc = gradient_descent(corpus_callosum_dataset, epochs=10)
for t in corpus_callosum_dataset:
    t.plot_preshape()
f_mean_cc.plot_preshape("red")
```

```
Gradient is equal to zero in epoch #4
```



PGA

In [ ]:

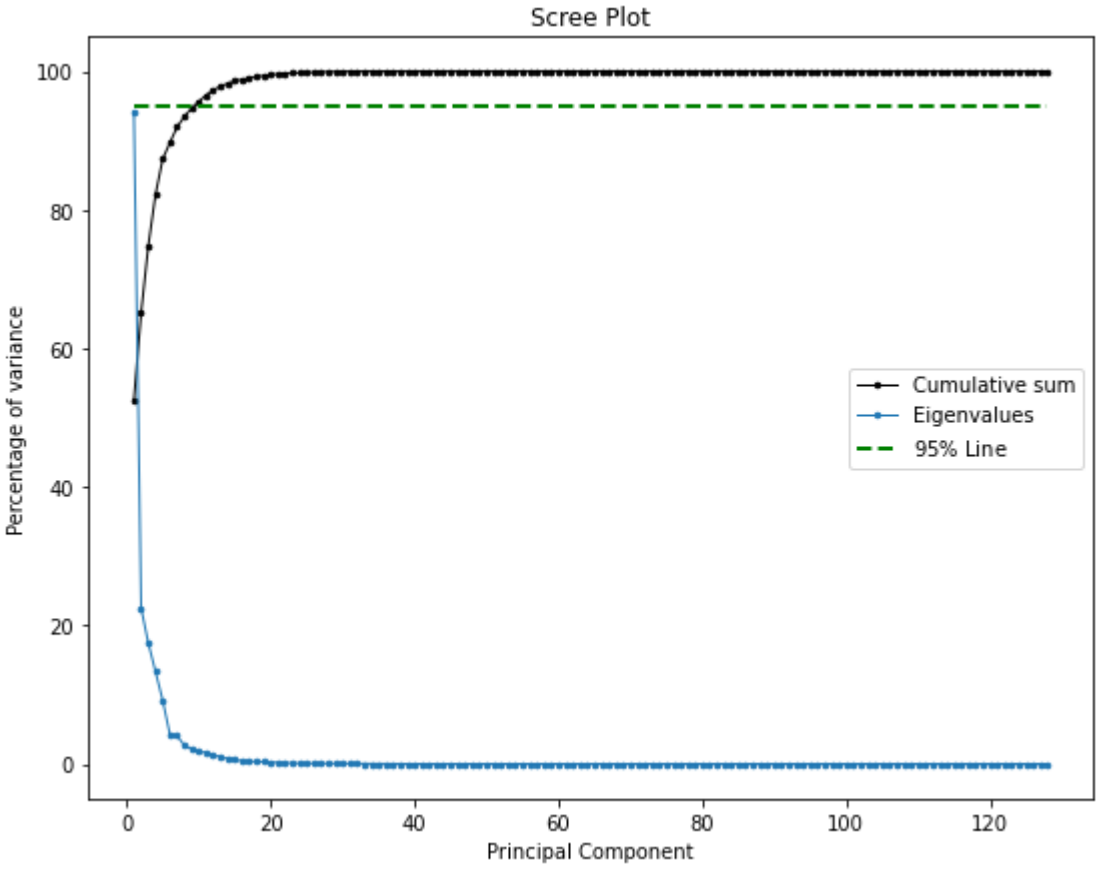
```
f_mean_t = gradient_descent(triangle_dataset, epochs=10)
eigenvectors, eigenvalues = PGA(f_mean_t, triangle_dataset)
scree_plot(eigenvalues)
```

```
Gradient is equal to zero in epoch #9
```

In [ ]:

```
f_mean_cc = gradient_descent(corpus_callosum_dataset, epochs=10)
eigenvectors, eigenvalues = PGA(f_mean_cc, corpus_callosum_dataset)
scree_plot(eigenvalues)
```

```
Gradient is equal to zero in epoch #4
```

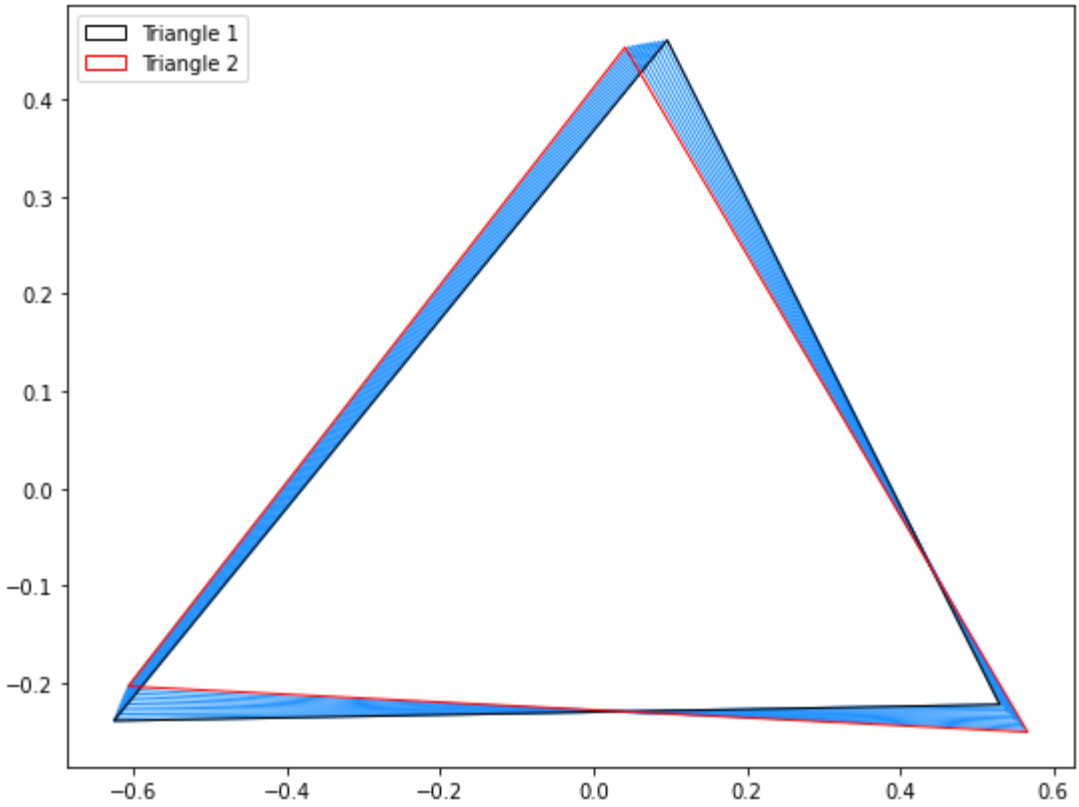


4 - Experiments - Triangles

Triangles - Geodesic between 2 objects

In [ ]:

```
triangle1 = triangle_dataset[0]
triangle2 = triangle_dataset[1]

aligned_triangle2 = Object(procrustes_analysis(triangle1.preshape, triangle2.
tangent = log_map(triangle1.preshape, aligned_triangle2.preshape)

fig, ax = plt.subplots(1, figsize=(9,7))

# Change this paramenter to plot 'r' images between 2 triangles
r = 10
for i in range(r):
    fig = Object(exponential_map(triangle1.preshape, i/r * tangent))
    fig.plot_preshape(ax=ax)
triangle1.plot_preshape("black", ax=ax, label="Triangle 1")
aligned_triangle2.plot_preshape("red", ax=ax, label="Triangle 2")
ax.legend(loc="upper left");
```



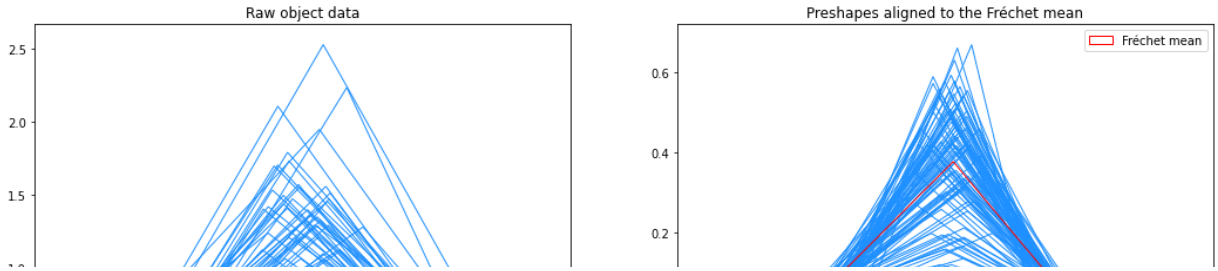
Triangles - Raw data & aligned preshapes

It can be clearly seen that some triangles in the raw data are significantly bigger than others, meaning that they have different scales. Further, the raw data is rotated in different ways, i.e. we can see that the top of the triangles points to different directions. On the other hand, the aligned preshapes have the same scale and they are pointing toward the same direction, which is the direction that the Fréchet mean has.

In [ ]:

```
f_mean_t = gradient_descent(triangle_dataset, epochs=10)
fig, ax = plt.subplots(1,2, figsize=(18,7))
ax[0].set_title("Raw object data")
ax[1].set_title("Preshapes aligned to the Fréchet mean")

for t in triangle_dataset:
    t.plot_matrix(ax=ax[0])
    aligned_t = Object(procrustes_analysis(f_mean_t.preshape, t.preshape))
    aligned_t.plot_preshape(ax=ax[1])

f_mean_t.plot_preshape("red", ax[1], label="Fréchet mean")
ax[1].legend();
```

```
Gradient is equal to zero in epoch #9
```



Triangles - Scree plot

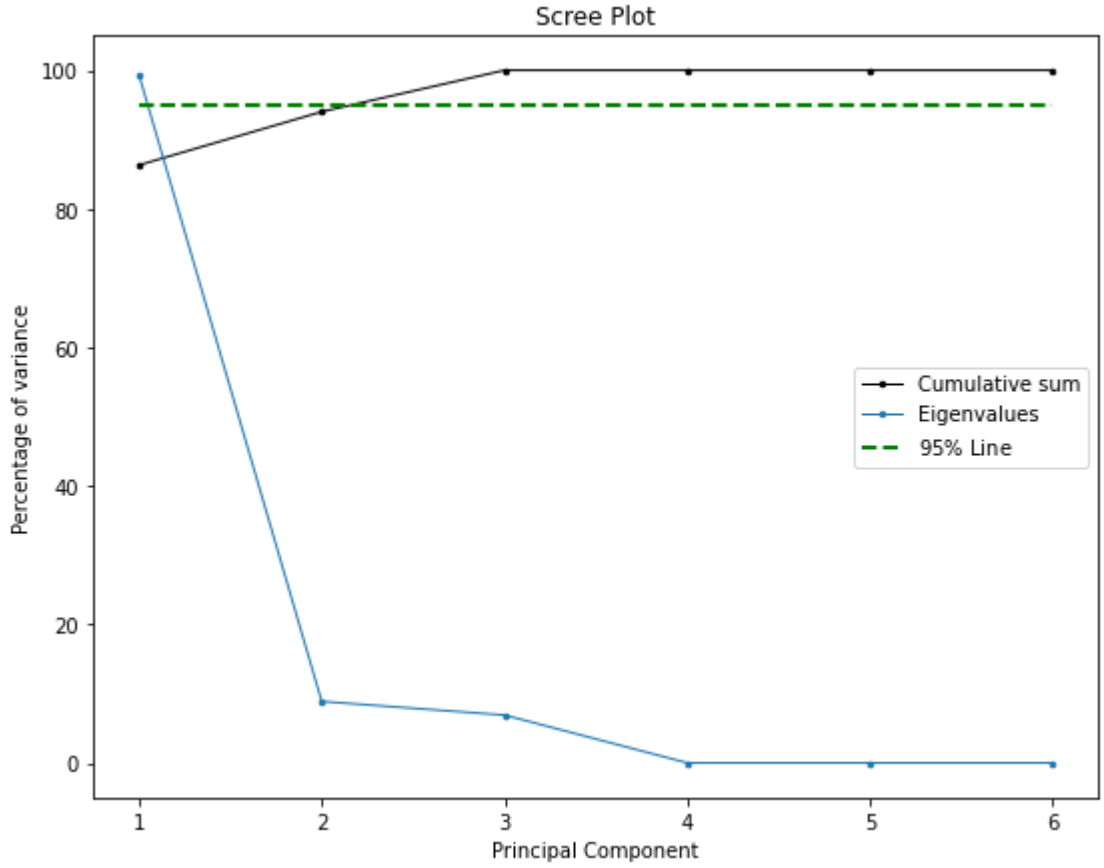
Seeing the plot below, I would choose 3 modes to represent the data because the cumulative sum of the variance is above 95% when it reaches the third mode. In the same way, we can see that the eigen values are not zero for the 3 first modes, but from the 4th they start to be 0.

Note: Since the triangles are generated with randomness, this analysis may not match with all the runs. I have seen a few times, that with just 2 modes the cumulative sum of variance is above 95%. In general, using 2 to 3 modes should be enough to describe the data, depending on how much variance you want to cover. Nonetheless, I attached an .html and a .pdf version of this notebook so the results are persisted.

In [ ]:

```
f_mean_t = gradient_descent(triangle_dataset, epochs=10)
eigenvectors, eigenvalues = PGA(f_mean_t, triangle_dataset)
scree_plot(eigenvalues)
```

```
Gradient is equal to zero in epoch #9
```



Triangles - Modes of variation

The 3 modes plotted below, represent different variations towards the direction of the corresponding eigenvector. Therefore we can see that the shapes plotted are different deformations of the Fréchet mean (black triangle). According to what I see, it seems that:

Mode 1: is representing movements in the height of the triangle. It can be seen that the black triangle which is the Fréchet mean is in the middle, and as you move towards negative standard deviations, the triangle height increases, while the opposite happens if you move towards the positive standard deviations.

Mode 2: is representing triangles rotations in some way. It can be seen that as you move towards the positive standard deviations, the triangle rotates to the left; and when you move towards the negative standard deviations, the triangle rotates to the right.

Mode 3: it is hard to describe what it is exactly happening in this mode. It seems to be representing a combination of rotation since the bottom line of the triangle is changing the slope; and a shift on the x-axis for the top of the triangle. The fact that this third mode does not show anything completely new/relevant could be explained since with two modes we are reaching almost 95% of the variance, meaning that most of the data could be represented with two modes and a third one may be unnesesary.

Note: the description presented may vary when executed again because of the randomness introduced when generating the triangles. Nonetheless, I attached an .html and a .pdf version of this notebook so the results are persisted.

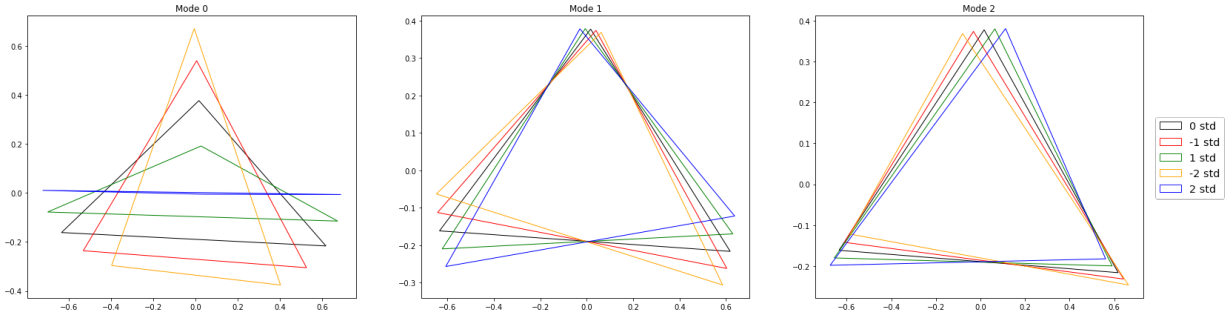
Overall, I expected something like this for the modes of variations, since these 3 modes should be capable of encoding the entire dataset, i.e. we should be able to generate almost all shapes in the dataset by moving on the manifold with the eigenvectors directions. And as described above, it seems that the two first modes are encoding specific aspects of the triangles, and the third one is kind of a combination of the previous two, thus allowing us to span almost all the triangles in our dataset.

Furthermore, if we take a look at how we are generating our 'synthetic' data, we are using a gaussian distribution to sample the y-position of the top point of our triangles, and that would explain why, the first mode represents the variation in the height of the triangle. In addition, we are adding gaussian noise to all of the x and y components of our triangles, and that would explain why the second and third mode represent the variations of the different points of the triangle.

In [ ]:

```
f_mean_t = gradient_descent(triangle_dataset, epochs=10)
eigenvectors, eigenvalues = PGA(f_mean_t, triangle_dataset)

fig, ax = plt.subplots(1,3, figsize=(27,7))

scale_colors = ["black", "red", "green", "orange", "blue"]
for i in range(3):
    for s, scale in enumerate([0,-1,1,-2,2]):
        current_eigenvector = eigenvectors[:,i].reshape((2,-1))
        scaled_eigenvector = scale * np.sqrt(eigenvalues[i]) * current_eigenv

        new_shape = Object(exponential_map(f_mean_t.preshape, scaled_eigenvec
        new_shape.plot_preshape(color=scale_colors[s], ax=ax[i], label=f"{sca

    ax[i].set_title(f"Mode {i}")

# Plot legend
handles, labels = ax[1].get_legend_handles_labels()
fig.legend(handles, labels, loc='center right', bbox_to_anchor=(0.96, 0.5), f
```

```
Gradient is equal to zero in epoch #9
```

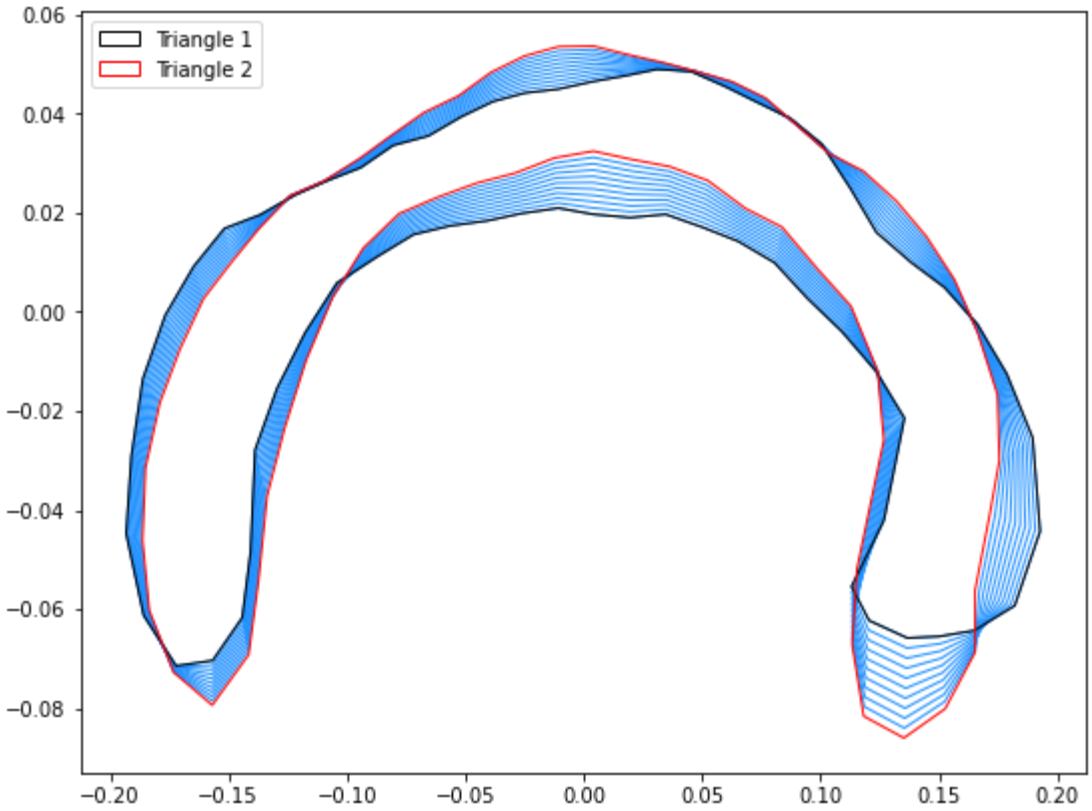


4 - Experiments - Corpus Callosum

Corpus callosum - Geodesic between two objects

In [ ]:

```
cc1 = corpus_callosum_dataset[0]
cc2 = corpus_callosum_dataset[1]

aligned_cc2 = Object(procrustes_analysis(cc1.preshape, cc2.preshape))
tangent = log_map(cc1.preshape, aligned_cc2.preshape)

fig, ax = plt.subplots(1, figsize=(9,7))

# Change this paramenter to plot 'r' images between 2 triangles
r = 10
for i in range(r):
    fig = Object(exponential_map(cc1.preshape, i/r * tangent))
    fig.plot_preshape(ax=ax)
cc1.plot_preshape("black", ax=ax, label="Triangle 1")
aligned_cc2.plot_preshape("red", ax=ax, label="Triangle 2")
ax.legend(loc="upper left");
```



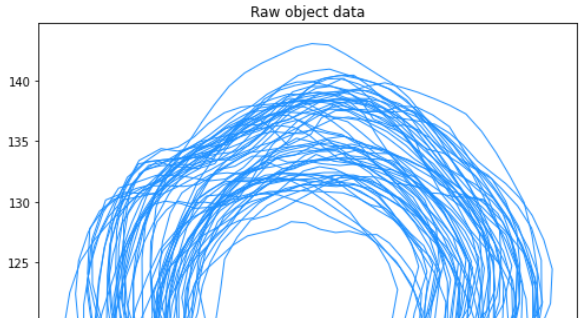
Corpus callosum - Raw data & aligned preshapes

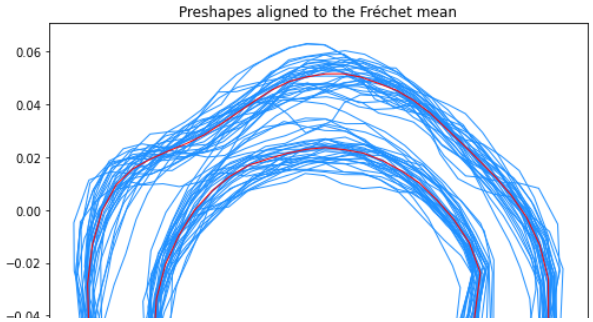
The analysis is somewhat analogous to the previous one. The raw data has different scales, and in this case the objects are translated in the space. If we take a closer look to the x and y axis of the Raw object data, we can see that x range goes from ~80 to ~170 and y range goes from ~110 to ~150. Lastly, the raw data shapes have different rotations since they were not aligned to anything. Instead the preshapes are aligned to the Fréchet mean, plus the scale and translation have been removed, therefore all the shapes look compact and less widespread.

In [ ]:

```
f_mean_cc = gradient_descent(corpus_callosum_dataset, epochs=10)
fig, ax = plt.subplots(1,2, figsize=(18,7))
ax[0].set_title("Raw object data")
ax[1].set_title("Preshapes aligned to the Fréchet mean")

for t in corpus_callosum_dataset:
    t.plot_matrix(ax=ax[0])
    aligned_t = Object(procrustes_analysis(f_mean_cc.preshape, t.preshape))
    aligned_t.plot_preshape(ax=ax[1])

f_mean_cc.plot_preshape("red", ax[1], label="Fréchet mean")
ax[1].legend();
```

```
Gradient is equal to zero in epoch #4
```





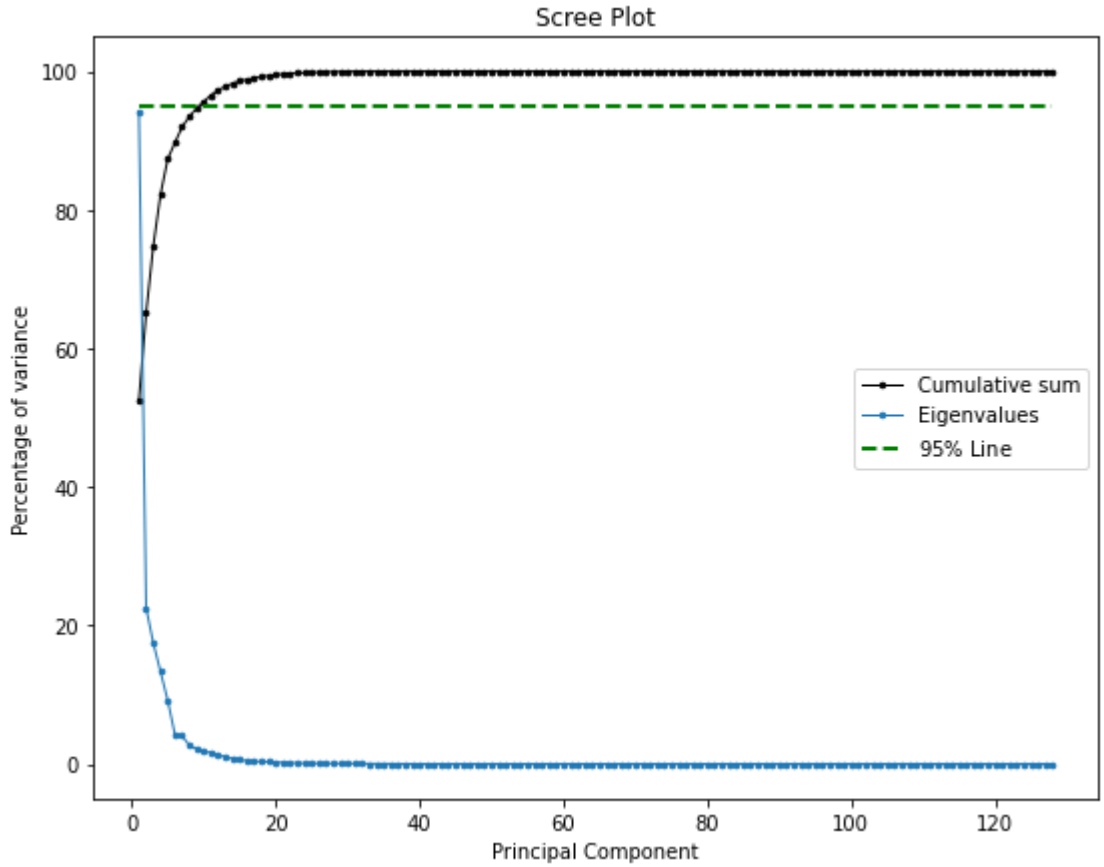
Corpus Callosum - Scree plot

Looking at the plot below, it can be seen that mode 9 is slightly below the 95% of variance, but it is above the 90%. So, I would choose the first 10 modes of variations to be able to represent at least 95% of the objects variance. However depending on how much % of variance you would like to have, you can choose to use less modes. To know how much % of variance is covered by using each mode, I added the black line which represent the cumulative % of variance.

In [ ]:

```
f_mean_cc = gradient_descent(corpus_callosum_dataset, epochs=10)
eigenvectors, eigenvalues = PGA(f_mean_cc, corpus_callosum_dataset)
scree_plot(eigenvalues)
```

```
Gradient is equal to zero in epoch #4
```



Corpus Callosum - Modes of variation

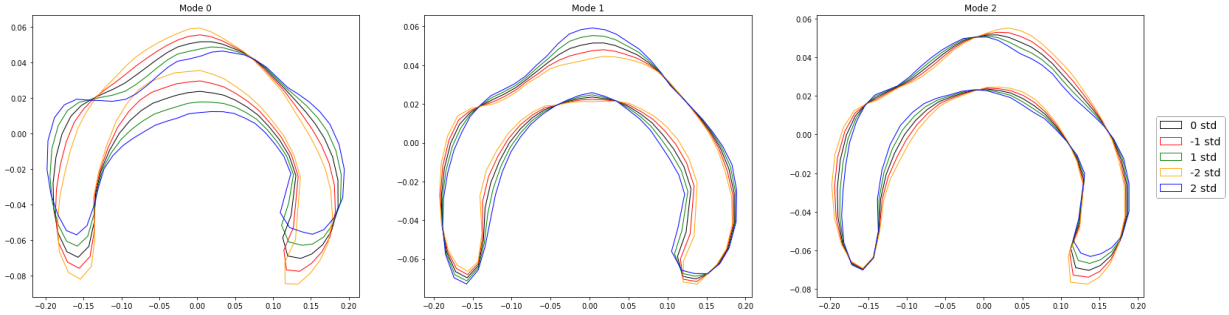
For the corpus callosum objects, it can be seen in the image below that the modes of variations

are enlarging or shrinking some segments of the shape. This makes sense since a corpus callosum object has 64 points, thus the shape is more complex, and the variations are not going to be as simple as they were for the triangle. This is also why in the above scree plot, we can see that we will need more than just 3 modes to represent the entire dataset.

In [ ]:

```
f_mean_cc = gradient_descent(corpus_callosum_dataset, epochs=10)
eigenvectors, eigenvalues = PGA(f_mean_cc, corpus_callosum_dataset)

fig, ax = plt.subplots(1,3, figsize=(27,7))

scale_colors = ["black", "red", "green", "orange", "blue"]
for i in range(3):
    for s, scale in enumerate([0,-1,1,-2,2]):
        current_eigenvector = eigenvectors[:,i].reshape((2,-1))
        scaled_eigenvector = scale * np.sqrt(eigenvalues[i]) * current_eigenv

        new_shape = Object(exponential_map(f_mean_cc.preshape, scaled_eigenve
        new_shape.plot_preshape(color=scale_colors[s], ax=ax[i], label=f"{sca

    ax[i].set_title(f"Mode {i}")

# Plot legend
handles, labels = ax[1].get_legend_handles_labels()
fig.legend(handles, labels, loc='center right', bbox_to_anchor=(0.96, 0.5), f
```

```
Gradient is equal to zero in epoch #4
```



Discussion

1) The maximal number of modes that are possible for triangle shapes is 6, which is the amount of features that a triangle has. We are representing a triangle with 3 points in a 2d plane, thus each point has 2 coordinates. Therefore, each triangle has 2 by 3 values to represent the 3 points in the 2d plane. So, in total, each triangle will be represented by 6 values, wich are also known as the 'features' of the object.

2) To generalize what was described above, the maximum amount of modes that are possible for a set of objects with n points will depend on the amount of points n and the dimension of the points. E.g. both of the datasets that we used for this assignment are shapes in a 2d plane, thus the dimension of the points is 2. For the corpus callosum, we have 2 by 64 values to reprsent the 64 points in the 2d plane, therefore the maximum amount of modes for the corpus callosum is 128.

In conclusion, the maximum amount of modes for a set of objects with n points is given by $n.dim$, where **n** is the amount of points, and **dim** is the dimension of the points.