多地市距离数据直观化

姓名	学号
李军贤	2022211259
赵欣冉	2022211264
李文豪	2022216030

建立无约束优化模型

建立如下模型：

$$\text{loss} = \Sigma_i \Sigma_j [(X_i - X_j)^T (X_i - X_j) - D_{ij}^2]^2$$

其中，X_n 是想要最小化的每个城市的坐标，D_{ij} 是每两个城市间的实际距离。

代码如下：

```python
# 定义目标函数，即要最小化的函数
def objective_function(city_coordinates, actual_distances):
    num_cities = city_coordinates.shape[0] // 2
    loss = 0
    for i in range(num_cities):
        for j in range(i + 1, num_cities):
            x_i, y_i = city_coordinates[2*i], city_coordinates[2*i+1]
            x_j, y_j = city_coordinates[2*j], city_coordinates[2*j+1]
            distance_ij = np.sqrt((x_i - x_j) ** 2 + (y_i - y_j) ** 2)
            loss += (distance_ij**2 - actual_distances[i, j]**2)**2

    return loss
```

不使用无约束优化模型，也可以使用多维缩放方法（MDS）实现该功能，代码如下：

```python
from sklearn.manifold import MDS

mds = MDS(n_components=2, dissimilarity="precomputed", random_state=42)
city_positions = mds.fit_transform(distance_matrix)
```

应用无约束优化算法对多维数据进行可视化

解决距离矩阵不对称的问题：

```python
def fix_asymmetry(matrix):
    # 找到不对称的位置
    non_symmetric_positions = np.where(matrix != matrix.T)
    # 修复不对称的元素
    for row, col in zip(*non_symmetric_positions):
        if abs(matrix[row, col]) > abs(matrix[col, row]):
            matrix[row, col] = matrix[row, col]
        else:
            matrix[row, col] = matrix[col, row]

    return matrix
```

无约束优化算法代码如下：

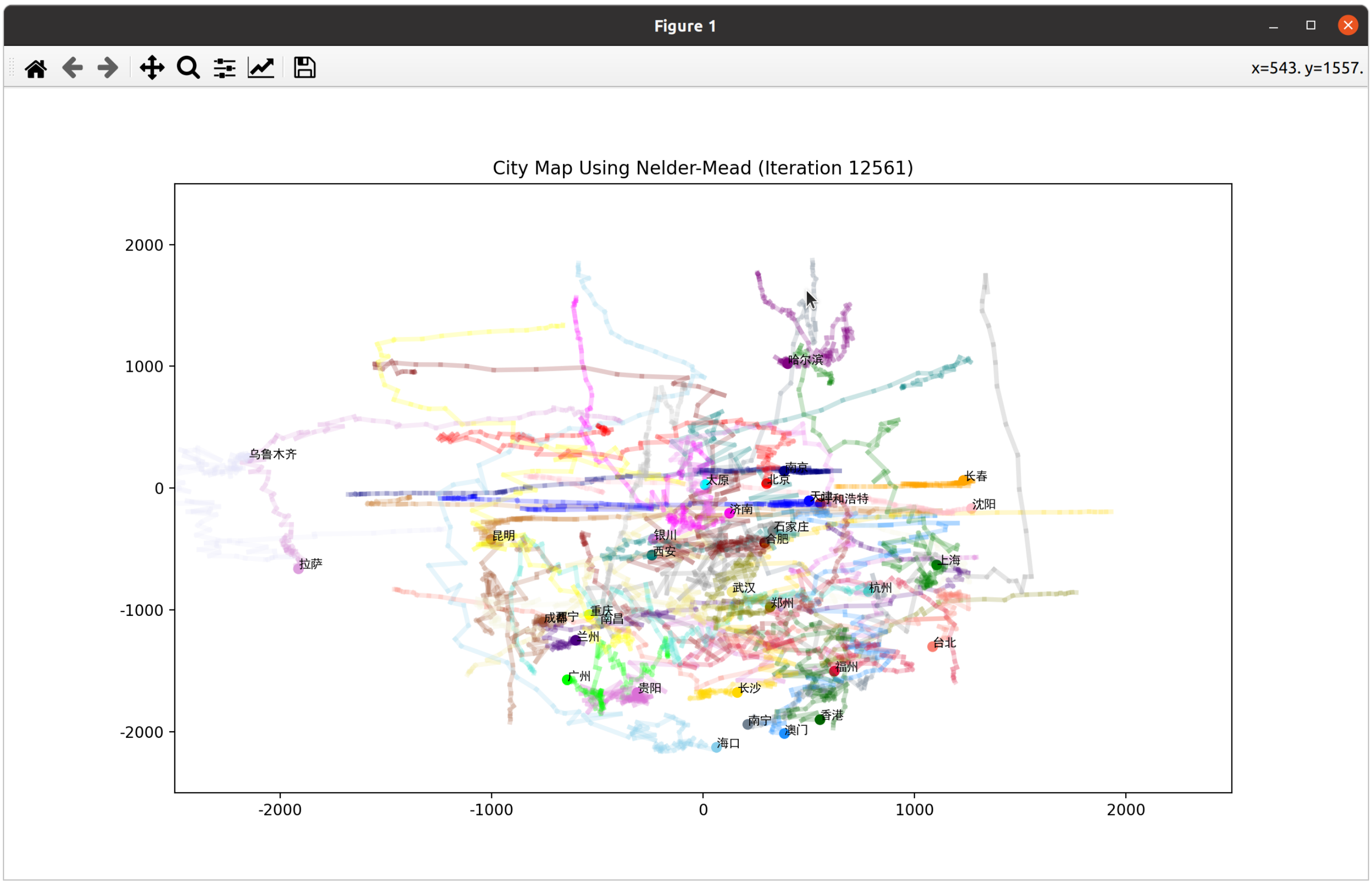
```python
# xk包含了当前迭代的结果
def callback_function(xk):
    results.append(xk)

def methods(method_name):
    return minimize(objective_function, city_coordinates, method=method_name, args=
(actual_distances), callback=callback_function)

num_cities = 34
city_coordinates = np.random.uniform(low=-2000, high=2000, size=(num_cities, 2))
# 城市的距离矩阵
actual_distances = distance_matrix
# 无约束优化算法
methods_list=["CG", "Nelder-Mead", "BFGS"]
# 存储每一次迭代的结果
results = []
# 最小化目标函数
result = methods(methods_list[0])
# result = methods(methods_list[1])
# result = methods(methods_list[2])
```

使用这些数据来构建这些城市的相对地理位置并直观展示

代码如下：

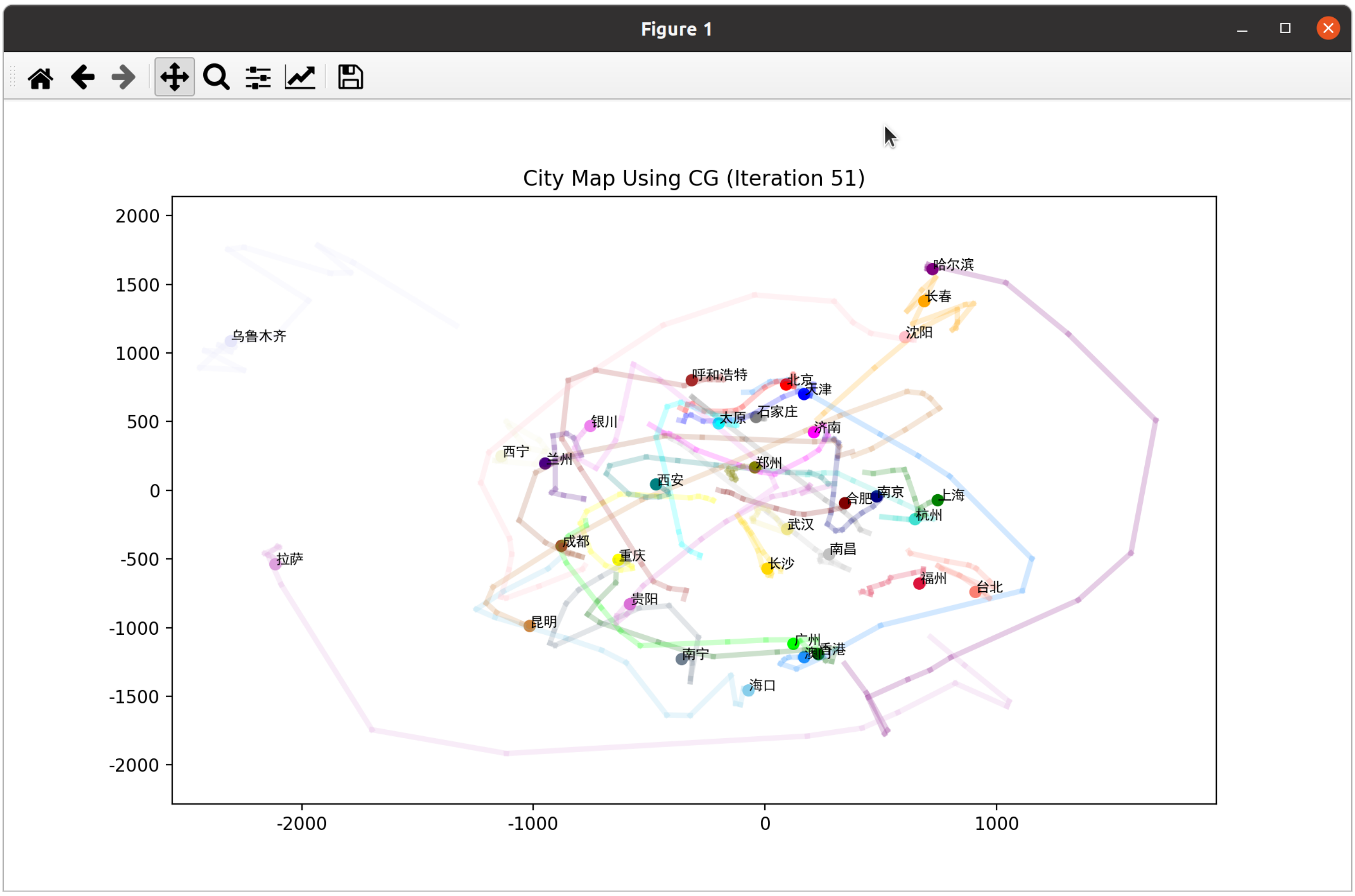
```python
def update(frame):
    # 设置字体和启用汉字支持
    font_properties = FontProperties(fname='SimHei.ttf')
    plt.rcParams['font.sans-serif'] = ['SimHei']
    plt.rcParams['axes.unicode_minus'] = False
    # 获取当前迭代的城市坐标
    current_coordinates = results[frame].reshape(-1, 2)
    # 添加城市名称前，清除之前的文本对象
    for text in ax.texts:
        text.remove()
    # 定义不同城市和轨迹的颜色
    city_colors = global_city_colors
    track_colors = city_colors
    # 绘制城市的轨迹
    for i in range(len(current_coordinates)):
        x, y = current_coordinates[i]
        city_color = city_colors[i]
        track_color = track_colors[i]
        # 绘制轨迹线段
        if frame > 0:
            previous_coordinates = results[frame - 1].reshape(-1, 2)
            x_values = [previous_coordinates[i, 0], x]
            y_values = [previous_coordinates[i, 1], y]
            ax.plot(x_values, y_values, color=track_color, linewidth=3, linestyle='-',
alpha=0.2)
    # 更新散点图的数据
    sc.set_offsets(np.c_[current_coordinates[:, 0], current_coordinates[:, 1]])
    # 添加城市名称
    for i, (x, y) in enumerate(current_coordinates):
        ax.text(x, y, city_names[i], fontsize=8, fontproperties=font_properties)
    # 更新标题，显示迭代次数
    ax.set_title(f'Iteration {frame+1}')


fig, ax = plt.subplots()
ax.set_xlim(-2500, 2500)
ax.set_ylim(-2500, 2500)

sc = ax.scatter(city_coordinates[:, 0], city_coordinates[:, 1],  color =
global_city_colors)

ani = FuncAnimation(fig, update, frames=len(results), repeat=False, interval=500)
plt.show()
```

Nelder-Mead法结果



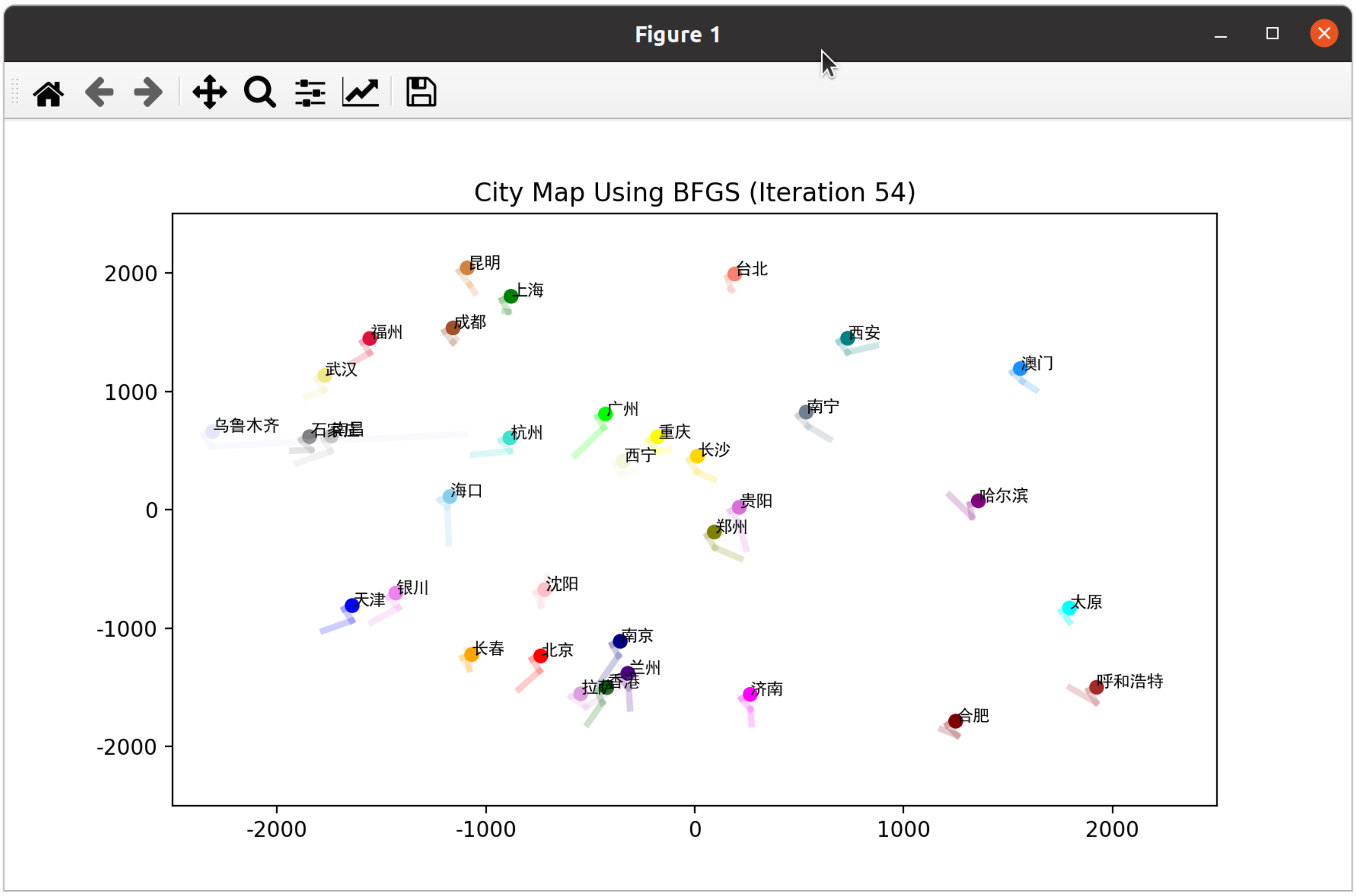
CG法结果

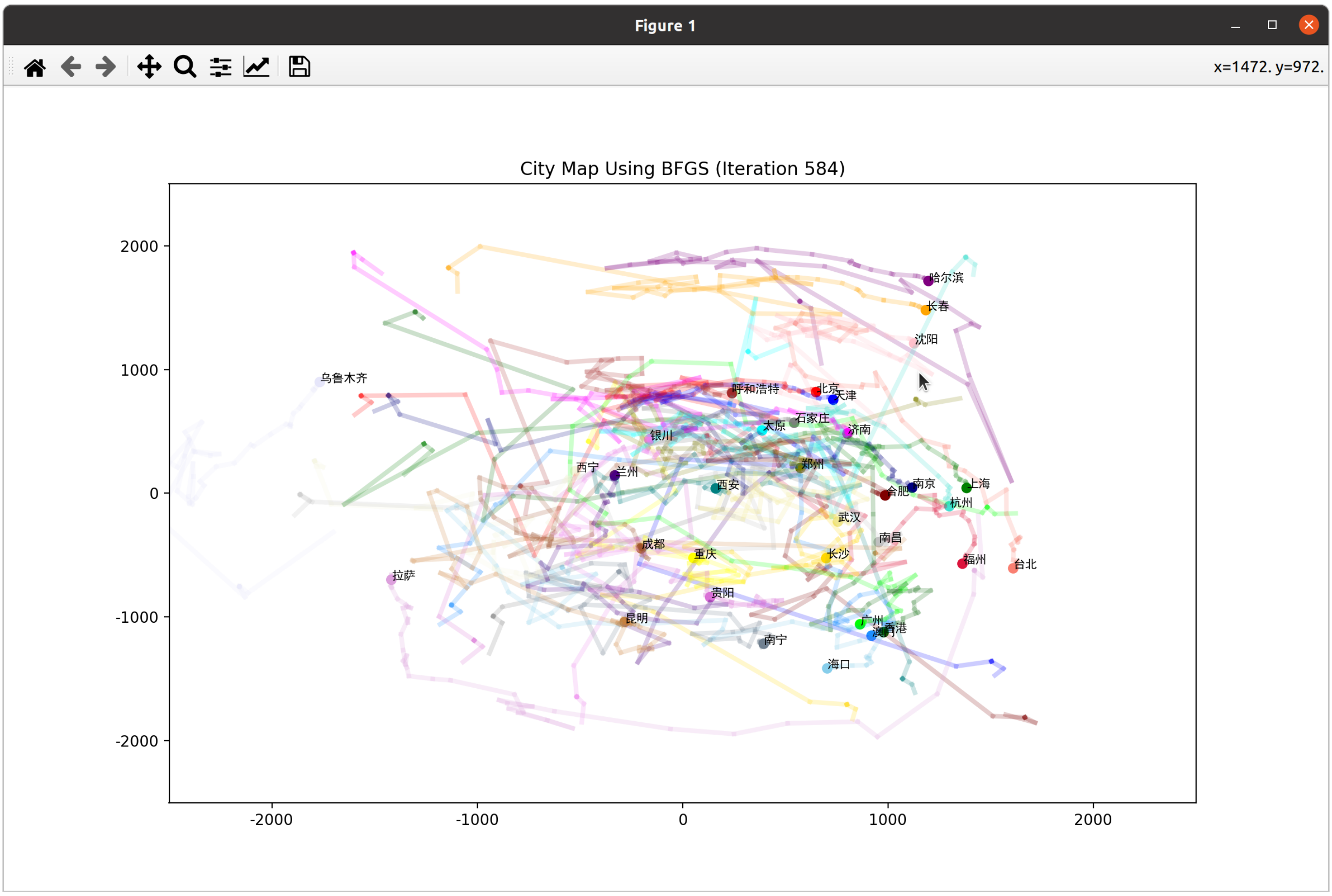
（程序输出的是一个动态图，请见文件中的 `2组CG法运行结果展示.mkv`）



BFGS法结果

第一幅图是陷入局部最优解的情况，第二幅图是得到全局最优解的情况。





保证地图的南北正确，代码如下：

```python
def trans(optimal):
    global results
    ha = 4 # 哈尔滨
    wu = 16 # 乌鲁木齐

    if(optimal[ha][0] < 0):
        optimal = [[-x,y] for x,y in optimal]
        results = [ [-elem if index % 2 == 0 else elem for index, elem in
enumerate(res) ] for res in results]

    if(optimal[ha][1] < 0):
        optimal = [[x,-y] for x,y in optimal]
        results = [ [-elem if index % 2 == 1 else elem for index, elem in
enumerate(res) ] for res in results]

    if(optimal[wu][0] > 0):
        optimal = [[y,x] for x,y in optimal]
        for index, my_list in  enumerate(results):
            for i in range(0, len(my_list)-1, 2):
                my_list[i], my_list[i+1] = my_list[i+1], my_list[i]
            results[index] = my_list

    results = np.array(results)
    print(results)
```

上述代码已开源：[visualization-of-city-distance]

比较几种无约束优化算法对此问题的效果

从我们的实验结果来看，Nelder-Mead方法需要大量的迭代次数（在我们的实验中迭代了一万多次，并且结果不全正确），收敛极其缓慢（在该问题下不收敛），这是因为Nelder-Mead法在求解高维问题时效率极低，其原因在于它涉及到构建和调整单纯形，这在高维空间中变得更加复杂。根据我们的建模，针对34个城市，每个城市设置了x和y两个变量，所以该问题是68维的，并不适合使用Nelder-Mead法。并且，最终结果中出现大部分城市的位置偏差严重的情况，可能受到初始点的选择和单纯形的形状的影响。 在某些情况下，它可能陷入局部最小值，而无法找到全局最小值。但是Nelder-Mead法也并非一无是处，该方法的实现相对简单，不需要计算目标函数的梯度信息，因此适合于一些问题，其中梯度信息难以获取或计算。当目标函数不可微或梯度难以计算时，Nelder-Mead方法是一个合适的选择。并且Nelder-Mead方法通常对于局部优化问题表现良好，特别是在目标函数的局部最小值附近。它可以被用作全局优化方法的一部分，以进一步提高局部搜索的精度。

反观Conjugate Gradient Method（CG）方法。该方法的效果十分不错。首先从收敛速度来看，CG方法只通过了少量的迭代次数便达到了收敛（我们的实验中只迭代了50-60次就可以收敛），可以看出该方法适用于求解高维问题。通过查阅相关资料我们也了解到CG法通常在大规模线性系统的求解中表现出色，因为它不需要存储完整的矩阵，而是使用矩阵-向量乘法，因此在处理大规模问题时可以节省内存。但是在我们的重复实验中，也出现了实验结果整体出现偏差（相对位置偏差很小）的情况，经过分析，这可能是由于CG法的性能受到参数的选择和问题的初始点的影响较为敏感导致的，这表明如果在处理对于病态矩阵（条件数较大）或非正定矩阵，它的性能可能较差。

最后来看Broyden-Fletcher-Goldfarb-Shanno（BFGS）。从收敛速度来看，BFGS法与CG法类似，收敛速度也比较快，但是（在我们的实验结果中）它比CG法要慢一个数量级。另外BFGS不需要计算目标函数的二阶导数（Hessian矩阵），较为适用于问题中难以获取或计算Hessian信息的情况。此外，BFGS的通用性较高，适用于各种非线性目标函数，包括连续可微和非光滑函数。但是，BFGS法需要维护和更新Hessian矩阵的逆，因此在高维问题中需要大量的内存，这可能限制了其应用于大规模问题。此外，与全局优化方法相比，BFGS方法更容易陷入局部最小值，因此对于非凸问题，可能需要多次运行以改善结果。