目前完成了迭代过程中最主要的两个运算函数，下面分别说明一下函数的运算过程（函数主体在 optimize.py 中）

优化 localframe 的函数：

```
def solve_localframe(restpose, pose_list, model_list, weight, edge_vpair, face_edge_index):
    lf_num = len(edge_vpair)
    n = lf_num * 3 * Config.BLENDSHAPENUM
    m = lf_num * 3 * Config.BLENDSHAPENUM
    A = np.zeros(m, n)  # Ax=b
    B = np.zeros(n, 1)
```

该函数求解所有 blendshape 的所有 localframe，在该函数中，权重值为已知量，未知量为各 blendshape 的各条边对应的三维矢量，因此未知量个数为：

Blendshape 数 * 模型边数（lf_num） * 3

该函数采用论文中描述的方式，将二次能量函数转化为线性系统（AX=B），这里采用对各未知量求偏导数的方式，因此共有 Blendshape 数 * 模型边数 * 3 个方程。

A 为（未知量个数 * 未知量个数）的矩阵。

该函数的参数说明：

Restpose 为方程式中的 idle pose，restpose[0][1]表示该模型第一个三角面片的 localframe 的第二个分量（面片数*3*3）。

Pose_list 为各 pose 的 localframe 数据，pose_list[0][1][1][0]表示第 1 个 pose 的第二个面片的第二个分量的 x 方向数值（pose 数*面片数*3*3）

$$E_{\text{reg}} = \sum_{i=1}^{n} w_i \left\| \mathbf{M}_i^B - \mathbf{M}_i^{A*} \right\|_F^2$$

Modellist 为能量函数中 reg 分量部分里面的模板 blendshape，其结构组织与 pose_list 类似。（模板数*面片数*3*3）
Weight 为各 blendshape 的权重，weight[0][1]代表了对应第一个 pose 的第二个 blendshape 的权重（pose 数*blendshape 数，取值范围 0~1）

edge_vpair, face_edge_index 为全局变量，分别为边与顶点的对应关系以及面片与边的对应关系，用于梳理 localframe 数据。

```
for fi in range(0, len(restpose)):
    for i1 in range(0, Config.BLENDSHAPENUM):
        for t in range(0, 2):
            w = (1 + np.linalg.norm(model_list[i1][fi])) / (Config.Ka + np.linalg.norm(model_list[i1][fi]))
            w = w ** Config.Theta

            A[(lf_num * i1 + face_edge_index[fi][t]) * 3][(lf_num * i1 + face_edge_index[fi][t]) * 3] += 2 * w
            B[(lf_num * i1 + face_edge_index[fi][t]) * 3] += 2 * model_list[i1][fi][t][0] * w

            A[(lf_num * i1 + face_edge_index[fi][t]) * 3 + 1][(lf_num * i1 + face_edge_index[fi][t]) * 3 + 1] += 2 * w
            B[(lf_num * i1 + face_edge_index[fi][t]) * 3 + 1] += 2 * model_list[i1][fi][t][1] * w

            A[(lf_num * i1 + face_edge_index[fi][t]) * 3 + 2][(lf_num * i1 + face_edge_index[fi][t]) * 3 + 2] += 2 * w
            B[(lf_num * i1 + face_edge_index[fi][t]) * 3 + 2] += 2 * model_list[i1][fi][t][2] * w
```

以上为 reg 部分，该部分对所有 blendshape 循环一次，每个 localframe 中的所有线性分量（前两个分量）参与计算。

```
(lf_num * i1 + face_edge_index[fi][t]) * 3
```

这一角标表示第 i1 个 blendshape 的第 fi 个面片的第 t 个分量，face_edge_index 代表了第 fi 个面片中的第 t 个分量对应的是边集合中的第几条边，也就是查询所对应的未知数的过程。最终*3 来对应 xyz 方向的三个坐标。

在 A 中对应项增量 2*w 是因为将对应的能量项展开求偏导后，对应的二次项系数，即 2*w，会添加到对应的未知量的系数上。对 B 的增量同理，对应的一次项系数，即 2*w*对应模板分量，会增加到方程右侧。

此过程重复三次，以得出 xyz 方向上的结果。

```
for i2 in range(i1,Config.BLENDSHAPENUM):
    for j in range(0, len(pose_list)):
        for t in range(0, 2):
            A[(lf_num * i1 + face_edge_index[fi][t]) * 3][(lf_num * i2 + face_edge_index[fi][t]) * 3] += 2 * weight[j][i2] * weight[j][i1]
            B[(lf_num * i1 + face_edge_index[fi][t]) * 3] += (restpose[fi][t][0] - pose_list[j][fi][t][0]) * weight[j][i1] * 2

            A[(lf_num * i1 + face_edge_index[fi][t]) * 3 + 1][(lf_num * i2 + face_edge_index[fi][t]) * 3 + 1] += 2 * weight[j][i2] * weight[j][i1]
            B[(lf_num * i1 + face_edge_index[fi][t]) * 3 + 1] += (restpose[fi][t][1] - pose_list[j][fi][t][1]) * weight[j][i1] * 2

            A[(lf_num * i1 + face_edge_index[fi][t]) * 3 + 2][(lf_num * i2 + face_edge_index[fi][t]) * 3 + 2] += 2 * weight[j][i2] * weight[j][i1]
            B[(lf_num * i1 + face_edge_index[fi][t]) * 3 + 2] += (restpose[fi][t][2] - pose_list[j][fi][t][2]) * weight[j][i1] * 2
```

之后是 fit 项的能量函数：

$$E_{\text{fit}} = \|\mathbf{M}_j^S - (\mathbf{M}_0^B + \sum_{i=1}^{n} \alpha_{ij}\mathbf{M}_i^B)\|_F^2$$

```
A[(lf_num * i1 + face_edge_index[fi][t]) * 3]\
[(lf_num * i2 + face_edge_index[fi][t]) * 3]
```

这一项表示的是在对第 i1 个 blendshape 中的第 fi 个面片的第 t 个分量求偏导后，第 i2 个 blendshape 上的对应未知量的系数。其系数为第 j 个 pose 对应的 i1,i2 号 blendshape 的权重乘积*2。求解这一系数的过程要经过对全部 blendshape 的两层循环，来计算各 blendshape 加权和的平方项。

```
for j in range(0, len(pose_list)):
    for t in range(0, 2):
        B[(lf_num * i1 + face_edge_index[fi][t]) * 3] += (restpose[fi][t][0] - pose_list[j][fi][t][0]) * weight[j][i1] * 2
        B[(lf_num * i1 + face_edge_index[fi][t]) * 3 + 1] += (restpose[fi][t][1] - pose_list[j][fi][t][1]) * weight[j][i1] * 2
        B[(lf_num * i1 + face_edge_index[fi][t]) * 3 + 2] += (restpose[fi][t][1] - pose_list[j][fi][t][1]) * weight[j][i1] * 2
    for i2 in range(i1,Config.BLENDSHAPENUM):
```

方程右端的常数项，即二次能量方程中的一次项，只需要对 blendshape 进行一次循环即可。

```
def E_fit():
    sum = 0
    target = [0, 0, 0]
    c = [0, 0, 0]
    for j in range(0, len(pose_list)):
        for fi in range(0, len(restpose)):
            for t in range(0, 2):
                c = restpose[fi][t] - pose_list[j][fi][t]
                for i in range(0, Config.BLENDSHAPENUM):
                    target += output[i][face_edge_index[fi][t]] * weight[j][i]
    res = c - target
    sum += (res[0] ** 2 + res[1] ** 2 + res[2] ** 2)
    return sum

def E_reg():
    sum = 0
    for fi in range(0, len(restpose)):
        for i in range(0, Config.BLENDSHAPENUM):
            w = (1 + np.linalg.norm(model_list[i][fi])) / (Config.Ka + np.linalg.norm(model_list[i][fi]))
            w = w ** Config.Theta
            for t in range(0, 2):
                target = output[i][face_edge_index[fi][t]] - model_list[i][fi][t]
                res = target[0] ** 2 + target[1] ** 2 + target[2] ** 2
                sum = sum + w * res
    return sum
print("Energy: " + str(E_fit()+E_reg()))
```

计算能量函数，用于迭代过程中查看。

对权重进行优化的函数：

```
def solve_weight(pose_list,bs_model,blendshape_list,pre_weight):
    weightnum = Config.BLENDSHAPENUM * len(pose_list)
```

$$E_{\mathrm{B}} = \sum_{k=1}^{N} \|\mathbf{v}_k^{S_j} - (\mathbf{v}_k^{B_0} + \sum_{i=1}^{n} \alpha_{ij}\mathbf{v}_k^{B_i})\|_2^2 + \gamma \sum_{i=1}^{n} (\alpha_{ij} - \alpha_{ij}^*)^2$$

对该函数采用二次规划法进行优化，采用的库为 cxvopt。
未知数为各 pose 对各 blendshape 的权重，个数为 pose 数*blendshape 数。

$$\begin{aligned} \min_{x} \quad & \frac{1}{2}x^\top Px + q^\top x \\ \text{subject to} \quad & Gx \preceq h \\ & Ax = b \end{aligned}$$

二次规划法即将能量函数转化为如上图的标准形式，然后将各系数带入求解器进行求解。
Poselist 为各 pose 中**顶点坐标**的数据（此处与求解 localframe 的函数不同，该函数中的数据为 localframe）。Pose_list[0][1][2]表示第一个 pose 中第 2 个顶点的 z 坐标。
Bsmodel 为 idle pose 的顶点坐标数据。
Blendshape_list 为本次迭代中参与计算的 blendshape 的顶点坐标数据。
Blendshape_list[0][1][2]代表第一个 blendshape 中的第 2 个顶点的 z 坐标。

```
P = np.zeros(weightnum,weightnum)
Q = np.zeros(weightnum,1)
G = np.zeros(weightnum,weightnum) - np.eye(weightnum,weightnum)
H = np.zeros(weightnum,1)
A = np.ones(weightnum,1)
B = [1.0]
```

求解器的参数初始设置如上，P 为二次项系数矩阵，对称阵。Q 为一次项系数。G 为主对角线数值均为-1 的对角阵，与 H 共同决定了所有权重值的下限为 0。A 与 B 则共同决定了所有权重值的和为 1。

```
for j in range(0,len(pose_list)):
    for k in range(0,len(pose_list[0])):
        c = pose_list[j][k] - bs_model[k]
        for i1 in range(0,Config.BLENDSHAPENUM):
            Q[j * bsnum + i1] += 2 * c[0] * blendshape_list[i1][k][0]
            Q[j * bsnum + i1] += 2 * c[1] * blendshape_list[i1][k][1]
            Q[j * bsnum + i1] += 2 * c[2] * blendshape_list[i1][k][2]
```

求解一次项系数的过程，只需对 blendshape 循环一次。
J * bsnum + i1 表示第 j 个 pose 对第 i 个 blendshape 的权重值。
C 为常数项（pose 与 idle pose 的差值），权重值的系数为常数项*对应坐标值*2。在 xyz 方向循环一次。

```
for i2 in range(i1,Config.BLENDSHAPENUM):
    P[j * bsnum + i1][j * bsnum + i2] += blendshape_list[i1][k][0] * blendshape_list[i2][k][0] * 2
    P[j * bsnum + i1][j * bsnum + i2] += blendshape_list[i1][k][1] * blendshape_list[i2][k][1] * 2
    P[j * bsnum + i1][j * bsnum + i2] += blendshape_list[i1][k][2] * blendshape_list[i2][k][2] * 2
    if(i1 != i2):
        P[j * bsnum + i2][j * bsnum + i1] += blendshape_list[i1][k][0] * blendshape_list[i2][k][0] * 2
        P[j * bsnum + i2][j * bsnum + i1] += blendshape_list[i1][k][1] * blendshape_list[i2][k][1] * 2
        P[j * bsnum + i2][j * bsnum + i1] += blendshape_list[i1][k][2] * blendshape_list[i2][k][2] * 2
```

之后求解二次项系数。除对角线上的元素外，要对矩阵对称两侧各进行一次结算。二次项系数为对应坐标的乘积*2。

$$\gamma \sum_{i=1}^{n} (\alpha_{ij} - \alpha_{ij}^{*})^2$$

最后增加用户设置的初值项。初值的权重为参数 pre_weight。

```
for j in range(0,len(pose_list)):
    for i in range(0, Config.BLENDSHAPENUM):
        P[j * bsnum + i][j * bsnum + i] += 2 * Config.Nju
        Q[j * bsnum + i] += -2 * Config.Nju * pre_weight[j][i]

result = solvers.qp(P, Q, G, H, A, B)
```

计算对应的二次项和一次项，之后进行求解即可。

目前存在的问题整理：

the $\mathbf{M}_i^{\mathcal{D}}$, we can reconstruct the vertex positions of each blendshape using a least-squares optimization as described in [Sumner and Popović 2004]. To prevent undesirable drifting, we constrain all vertices that are stationary in a template blendshape to remain fixed in the corresponding target blendshapes as well.

1、原文中防止偏移这一块我还没有想好具体的实现方式，目前的思路是：
在求解 localframe 的过程中，未知量是各个边，那么如果一个 pose 上的某些顶点与 idle pose 相比完全不变（坐标差值极小），就记录这些顶点，然后到边的集合里查找连接了这些顶点的边，在计算中将这些边从未知量中排除（不列入计算）。而在得出结果后，这些边的取值直接取 idle pose 中的数值。

这样的话对于每一个 pose 而言实际上还要再去记录该 pose 参与优化运算的边的数量，感觉会比较繁琐。另外，如果有些固定的顶点没有与其他顶点连接，则如何体现出其固定的特

性呢？

2、在之前的讨论中，我们认为可以先通过从 idle pose 到现有的某个 pose 组的 blendshape 的形变来求取某组尚未求解的 pose 的 blendshape 初值。不过在做了一些实验后我觉得这一步还是需要相应的权重值矩阵，因为单纯计算从某个 pose 形变为某个 blendshape 的各面片形变，然后将其应用到当前 pose 上，效果可能不如直接应用另一组拓扑逻辑一致的 blendshape 作为初值好。

这一问题可以总结为目前 blendshape 初值这一块如何设置才能更好。

3、从 localframe 还原出网格顶点坐标这一步，文中表示采用[Sumner and Popovi´c 2004]中的最小二乘优化方法，这部分目前还有一些问题，之前我直接采用最小二乘法去优化偏移量，但 Deformation Transfer 这篇文章中采用的坐标还原方法是求解稀疏线性方程组（Vertex Formulation 这一节），数据结构也与本文的 localframe 不一致。

4、数据基数较大，每次运算各 pose 的 localframe 以及其他模型测试时都会花费较长时间，导致出 bug 时不太方便测试，我目前采用的方法是定义一些小的数据作为参数来测试函数的准确性。