

```

# ----- Checkerboard -----
# -----
def con_isometry_checkerboard(self, l0, angle0):
    """
    keep 2 kinds of diagonal edge-lengths and their crossing angle
    X += [ld1, ld2, ud1, ud2]
    1. (v1-v3) = ld1*ud1, ud1**2=1
    2. (v2-v4) = ld2*ud2, ud2**2=1
    3. ld1 == init_ld1, ld2 == init_ld2
    4. ud1*ud2 == init_ud1*init_ud2
    """
    w = self.get_weight('isometry_checkerboard')
    V = self.mesh.V
    num = self.mesh.num_quadface
    N = self.N
    X = self.X
    numl = self._N7-8*num
    numud = self._N7-6*num
    arr = np.arange(num)

    c_ld1 = numl+arr
    c_ld2 = numl+num+arr
    vi = self.mesh.quadface
    v1, v2, v3, v4 = vi[:, :4], vi[:, 1:4], vi[:, 2:4], vi[:, 3:4]
    c_v1 = np.r_[v1, V+v1, 2*V+v1] # [x, y, z]
    c_v2 = np.r_[v2, V+v2, 2*V+v2] # [x, y, z]
    c_v3 = np.r_[v3, V+v3, 2*V+v3] # [x, y, z]
    c_v4 = np.r_[v4, V+v4, 2*V+v4] # [x, y, z]
    c_ud1 = np.r_[numud+arr, numud+num+arr, numud+2*num+arr]
    c_ud2 = c_ud1+3*num

    He1, re1 = self._edge(X, c_v1, c_v3, c_ld1, c_ud1, num, N)
    He2, re2 = self._edge(X, c_v2, c_v4, c_ld2, c_ud2, num, N)
    Hu1, ru1 = self._unit(X, c_ud1, num, N)
    Hu2, ru2 = self._unit(X, c_ud2, num, N)
    Hl1, rl1 = self._constl(c_ld1, l0[:num], num, N)
    Hl2, rl2 = self._constl(c_ld2, l0[num:], num, N)
    Ha, ra = self._constangle(X, c_ud1, c_ud2, angle0, num, N)

    H = sparse.vstack((He1, He2, Hu1, Hu2, Hl1, Hl2, Ha))
    r = np.r_[re1, re2, ru1, ru2, rl1, rl2, ra]
    self.add_iterative_constraint(H*w, r*w, 'isometry(checkboard)')

```

Constraint for isometry\_checkerboard way as in Caigui paper

1. basic: get each quad faces' vertex indices  
即 self.mesh.quadface 函数，见后面
2. 表示isometry条件  
way1: 使用提取给定初始网对角线长度&夹角 (该方法)。  
way2: 使用读取两个对应网格，将其vertices一起作为变量
3. 将下面约束条件表示成稀疏矩阵H 和列表 r
4. 求解非齐次稀疏矩阵线性解

变量X =[vertices, ld1, ld2, ud1, ud2]

Vertices: 所有格点3维坐标

Ld1,ld2: 分别是两组对角线长度

Ud1,ud2: 分别是两组对角线方向单位向量

Way1:

$$(v_0 - v_2)^2 = C_0, (v_1 - v_3)^2 = C_1, (v_0 - v_2) \cdot (v_1 - v_3) = C_3.$$

Way2:

$$c_{iso,0}(f) = (v_0 - v_2)^2 - (v'_0 - v'_2)^2 = 0,$$

$$c_{iso,1}(f) = (v_1 - v_3)^2 - (v'_1 - v'_3)^2 = 0,$$

$$c_{iso,2}(f) = (v_0 - v_2) \cdot (v_1 - v_3) - (v'_0 - v'_2) \cdot (v'_1 - v'_3) = 0.$$

```

def _edge(self, X, c_v1, c_v3, c_ld1, c_ud1, num, N):
    "(v1-v3) = ld1*ud1"
    ld1 = X[c_ld1]
    ud1 = X[c_ud1]
    a3 = np.ones(3*num)
    row1 = np.tile(np.arange(3*num), 4)
    col = np.r_[c_v1, c_v3, np.tile(c_ld1, 3), c_ud1]
    data = np.r_[a3, -a3, -ud1, -np.tile(ld1, 3)]
    r = -np.tile(ld1, 3)*ud1
    H = sparse.coo_matrix((data, (row1, col)), shape=(3*num, N))
    return H, r

def _unit(self, X, c_ud1, num, N):
    "ud1**2=1"
    arr = np.arange(num)
    row2 = np.tile(arr, 3)
    col = c_ud1
    data = 2*X[col]
    r = np.linalg.norm(X[col].reshape(-1, 3, order='F'), axis=1)**2 + np.ones(num)
    H = sparse.coo_matrix((data, (row2, col)), shape=(num, N))
    return H, r

def _constl(self, c_ld1, init_l1, num, N):
    "ld1 == const."
    row3 = np.arange(num, dtype=int)
    col = c_ld1
    data = np.ones(num, dtype=int)
    r = init_l1
    H = sparse.coo_matrix((data, (row3, col)), shape=(num, N))
    return H, r

```

### \_edge函数

表示：对角线向量==对角线长度\*单位向量  
被con\_isometry\_checkberboard函数调用2次  
返回稀疏矩阵，和列表 H, r

### \_unit函数

表示：对角线单位向量  
被con\_isometry\_checkberboard函数调用2次  
返回稀疏矩阵，和列表 H, r

### \_constl函数

表示：对角线长度==给定初始长度值  
被con\_isometry\_checkberboard函数调用2次  
返回稀疏矩阵，和列表 H, r

### \_constangle函数

表示：单位对角线向量夹角为给定cos(alpha)  
被con\_isometry\_checkberboard函数调用1次  
返回稀疏矩阵，和列表 H, r

```

def _constangle(self, X, c_ud1, c_ud2, angle0, num, N):
    "ud1*ud2 == const."
    row4 = np.tile(np.arange(num), 6)
    col = np.r_[c_ud1, c_ud2]
    data = np.r_[X[c_ud2], X[c_ud1]]
    r = np.einsum('ij, ij->i', X[c_ud1].reshape(-1, 3, order='F'), X[c_ud2].reshape(-1, 3, order='F')) + angle0
    H = sparse.coo_matrix((data, (row4, col)), shape=(num, N))
    return H, r

```

```
def quadfaces(self):
    "for quad diagonals"
    "quadface, num_quadface, quadface_order"
    f, v1, v2 = self.face_edge_vertices_iterators(order=True)
    f4, vi = [], []
    for i in range(self.F):
        ind = np.where(f==i)[0]
        if len(ind)==4:
            f4.extend([i,i,i,i])
            vi.extend(v1[ind])
            #vj.extend(v2[ind])
    self._num_quadface = len(f4) // 4
    #v1,v2,v3,v4 = vi[::4],vi[1::4],vi[2::4],vi[3::4]
    self._quadface = np.array(vi,dtype=int)
    self._quadface_order = np.unique(f4)
```

由halfedge半边数据结构表示出每个quadface的格点索引值  
即返回列表[v1,v2,v3,v4] = quadface

```
def face_edge_vertices_iterators(self, sort=False, order=False):
    H = self.halfedges
    f = H[:,1]
    vi = H[:,0]
    vj = H[H[:,2],0]
    if order:
        i = self.face_ordered_halfedges()
        f = f[i]
        vi = vi[i]
        vj = vj[i]
    else:
        i = np.where(H[:,1] >= 0)[0]
        f = f[i]
        vi = vi[i]
        vj = vj[i]
        if sort:
            i = np.argsort(f)
            vi = vi[i]
            vj = vj[i]
    return f, vi, vj
```

```
def face_ordered_halfedges(self):
    H = np.copy(self.halfedges)
    i = np.argsort(H[:,1])
    i = i[np.where(H[i,1] >= 0)]
    f = H[i,1]
    index = np.arange(i.shape[0])
    _, j = np.unique(f, True)
    f = np.delete(f, j)
    index = np.delete(index, j)
    while f.shape[0] > 0:
        _, j = np.unique(f, True)
        i[index[j]] = H[i[index[j]] - 1, 2]
        f = np.delete(f, j)
        index = np.delete(index, j)
    return i
```

# Killing field

$$\begin{aligned} E_k &= E_t + \lambda \cdot E_c + w_{fair} \cdot E_{fair} \\ &= X^T (T + \lambda \cdot C + w_{fair} \cdot K) X \\ &= X^T A X \end{aligned}$$

constraints on velocity field (infinitesimal isometry)

velocity

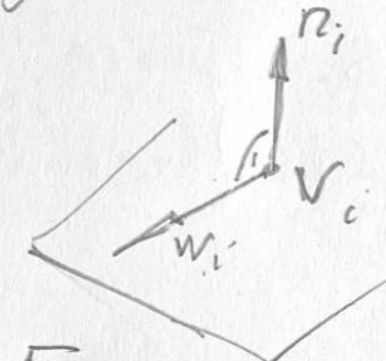
$$\begin{aligned} (1) \quad & (v_2 - v_0) \cdot (w_2 - w_0) = 0 \\ (2) \quad & (v_3 - v_1) \cdot (w_3 - w_1) = 0 \\ (3) \quad & (v_2 - v_0) \cdot (w_3 - w_1) + (w_2 - w_0) \cdot (v_3 - v_1) = 0 \end{aligned} \quad \left\{ \begin{array}{l} \text{Linear} \\ \text{homogeneous} \\ (v_i \text{ given}) \end{array} \right.$$

$$E_t = \sum (w_i \cdot n_i)^2 = 0$$

tangent

normal at  $v_i$

old constraints (1-3), squared  $E_c$



$$E_c = \sum_f ((v_2 - v_0) \cdot (w_2 - w_0))^2 + \sum_f ((v_3 - v_1) \cdot (w_3 - w_1))^2 + \sum_f ((v_2 - v_0) \cdot (w_3 - w_1) + (v_3 - v_1) \cdot (w_2 - w_0))^2$$

$$E_t = \sum (w_i \cdot n_i)^2.$$

$$E_{fair} = \sum_v (w_{i-1} + w_{i+1} - 2w_i)^2$$



```
def get_killing_eigen(self, killing=True, efair=0.01):
    """
    X = only [wi] , i=1...V
    Ek = Et+la*Ec = X' * (T+la*C) * X
    A = T+la*C + efair*K
    eigen of M:
        if num=1, close to iso.to surf.of revelotion
        if num=3, close to const. Gaussian curv. K
    A is influenced by normals & efair
    """
    lamda = self.get_weight('iso_velocity')
    refermesh = self.mesh
    Vi = self.mesh.vertices
    V = self.mesh.V
    Mnum = 3*V
    iv0, iv1, iv2, iv3 = self.mesh.quadface.T
    d1 = Vi[iv2] - Vi[iv0]
    d2 = Vi[iv3] - Vi[iv1]
    cw0 = np.r_[iv0, V+iv0, 2*V+iv0]
    cw1 = np.r_[iv1, V+iv1, 2*V+iv1]
    cw2 = np.r_[iv2, V+iv2, 2*V+iv2]
    cw3 = np.r_[iv3, V+iv3, 2*V+iv3]

    C = self.__iso_matrix(cw0, cw1, cw2, cw3, d1, d2, Mnum) * lamda
    A = C

    if efair:
        "v3+v1-2*v2-->0"
        K = self.__fairness_matrix(V, efair, Mnum)
        A += K

    if killing:
        normals = refermesh.vertex_normals()
        closest = refermesh.closest_vertices(Vi)
        normals = normals[closest, :]
        c_w = np.arange(Mnum)
        T = self.__matrix_1(c_w, normals, Mnum) # self.__killing_ma
        A += T
```

Global killing field

求解关于mesh所有 vertices 的特征方程:

需要表示3个对称矩阵: C, T, K

Solving eigen value problem of  $X^T A X = 0$

$$E_k := E_t + \lambda * E_c + 0.01 * E_{\{fair\}}$$

$$= X^T (T + \lambda * C + 0.01 * K) X$$

$$= X^T A X$$

- $E_t$ : tangent condition
- $E_c$ : i-velocity (1) (2) (3),  $\lambda=1$
- $E_{\{fair\}}$ : Fairness on wi

```
"column v[:,i] is the eigenvector corresponding to the eigenvalue w[i]"
vals, vecs = np.linalg.eigh(A.toarray())
vals = vals / (3*V) # per vertex
amin = np.argmin(np.abs(vals))
vmin = vecs[:, amin]

print('-'*20)
eig = list(np.abs(vals))
print('list top 5 smallest eigen values:\n')
for i in heapq.nsmallest(5, eig):
    print('*', i)
print('='*20)

return self.mesh.vertices, vin
```

```

def __fairness_matrix(self, Vnum, efair, Mnum, xnum=0):
    "(w1+w3-2*w)^2=0; (w2+w4-2*w)^2=0"
    v, v1, v2, v3, v4 = self.mesh.ver_regular_star.T
    m13 = self.__fair(v, v1, v3, Vnum, Mnum, xnum=xnum)
    m24 = self.__fair(v, v2, v4, Vnum, Mnum, xnum=xnum)
    return (m13+m24) * efair

def __fair(self, v, v1, v3, Vnum, Mnum, xnum=0, arcc=None, arrl=None, arrr=None):
    "(w1+w3-2*w)^2=0;"
    def __matrix(c_w, num, Mnum):
        if arcc is not None:
            data = np.array([])
            for i in range(len(arcc)):
                one = np.array([arrr[i], arrr[i], -2*arcc[i]])
                d = np.outer(one, one).flatten()
                data = np.r_[data, d]
        else:
            one = np.array([1, 1, -2])
            d = np.outer(one, one).flatten()
            data = np.tile(d, num)
        rw = (c_w.reshape(-1, 3, order='F')).flatten()
        row = rw.repeat(3)
        cw = c_w.reshape(-1, 3, order='F')
        col = np.hstack((cw, cw, cw)).flatten()
        m = sparse.coo_matrix((data, (row, col)), shape=(Mnum, Mnum))
        return m
    num = len(v)
    w13x = np.r_[v1, v3, v] + xnum
    w13y = np.r_[Vnum+v1, Vnum+v3, Vnum+v] + xnum
    w13z = np.r_[2*Vnum+v1, 2*Vnum+v3, 2*Vnum+v] + xnum
    m1 = __matrix(w13x, num, Mnum)
    m2 = __matrix(w13y, num, Mnum)
    m3 = __matrix(w13z, num, Mnum)
    return m1+m2+m3

```

表示fairness 对称矩阵

$$E_{fair} = \sum_v (w_{i-1} + w_{i+1} - 2w_i)^2$$

self.mesh.ver\_regular\_star:

表示每个regular格点处，上下左右相邻的4个格点列表

\_matrix1 和 \_matrix2 是基本的，公用的对称矩阵

表示tangent 对称矩阵

表示velocity 对称矩阵

```
def __iso_matrix(self, c_w0, c_w1, c_w2, c_w3, d1, d2, Mnum):
    " $((v2-v0)*w2-(v2-v0)*w0)^2=0$ "
    m0 = self.__matrix_1(c_w0, d1, Mnum)
    m2 = self.__matrix_1(c_w2, d1, Mnum)
    m02 = self.__matrix_2(c_w0, c_w2, d1, d1, Mnum)
    " $((v3-v1)*w3-(v3-v1)*w1)^2=0$ "
    m1 = self.__matrix_1(c_w1, d2, Mnum)
    m3 = self.__matrix_1(c_w3, d2, Mnum)
    m13 = self.__matrix_2(c_w1, c_w3, d2, d2, Mnum)
    " $(v2-v0)*w3 + (v3-v1)*w2 - (v2-v0)*w1 - (v3-v1)*w0$ "
    mm0 = self.__matrix_1(c_w0, d2, Mnum)
    mm1 = self.__matrix_1(c_w1, d1, Mnum)
    mm2 = self.__matrix_1(c_w2, d2, Mnum)
    mm3 = self.__matrix_1(c_w3, d1, Mnum)
    mm01 = self.__matrix_2(c_w0, c_w1, d2, d1, Mnum)
    mm02 = self.__matrix_2(c_w0, c_w2, d2, d2, Mnum)
    mm03 = self.__matrix_2(c_w0, c_w3, d2, d1, Mnum)
    mm12 = self.__matrix_2(c_w1, c_w2, d1, d2, Mnum)
    mm13 = self.__matrix_2(c_w1, c_w3, d1, d1, Mnum)
    mm23 = self.__matrix_2(c_w2, c_w3, d2, d1, Mnum)
    return m0+m2-m02+m1+m3-m13+mm0+mm1+mm2+mm3+mm01-mm02-mm03-mm12-mm13+mm23
```

```
def __matrix_1(self, c_w, normals, Mnum):
    """
     $(wi*ni)^2 = 0$ 
     $(a,b,c)^T * (a,b,c)$ 
    = [aa ab ac
        ab bb bc
        ac bc cc]
    """
    data = np.array([])
    for ni in normals:
        d = np.outer(ni, ni).flatten()
        data = np.r_[data, d]
    rw = (c_w.reshape(-1, 3, order='F')).flatten()
    row = rw.repeat(3)
    cw = c_w.reshape(-1, 3, order='F')
    col = np.hstack((cw, cw, cw)).flatten()
    m = sparse.coo_matrix((data, (row, col)), shape=(Mnum, Mnum))
    return m

def __matrix_2(self, c_wi, c_wj, di, dj, Mnum):
    """
     $2*ni*nj*wi*wj = 0$ 
     $(a,b,c)^T * (d,e,f)$ 
    = [ad ae af
        bd be bf
        cd ce cf]
    """
    data = np.array([])
    for k in range(len(di)):
        ni, nj = di[k], dj[k]
        d = np.outer(ni, nj).flatten()
        data = np.r_[data, d]
    rw = (c_wi.reshape(-1, 3, order='F')).flatten()
    row = rw.repeat(3)
    cw = c_wj.reshape(-1, 3, order='F')
    col = np.hstack((cw, cw, cw)).flatten()
    m = sparse.coo_matrix((data, (row, col)), shape=(Mnum, Mnum))
    return m.T+m
```

1. (12 points) Implement an algorithm which computes a principal mesh using the diagonal mesh approach (Jiang et al. AAG 2020). Also compute a torsion free support structure and provide some good illustrations of examples.

# Checkerboard Patterns with Black Rectangles

CHI-HAN PENG\*, KAUST  
 CAIGUI JIANG\*, KAUST  
 PETER WONKA, KAUST  
 HELMUT POTTMANN, KAUST

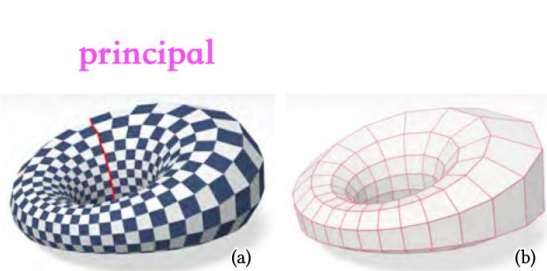


Fig. 6. A checkerboard pattern with black rectangles and white planar quads represents a discrete principal curvature parameterization. While the pattern (a) lacks some fairness (e.g., the red polyline), the diagonal meshes of the control mesh, one shown in (b), do not suffer from this problem and are also discrete principal curvature parameterizations.

(3) For more boundaries or higher genus, one needs to provide more flexibility. One cannot specify a target surface precisely, but optimization will lead towards a possible target

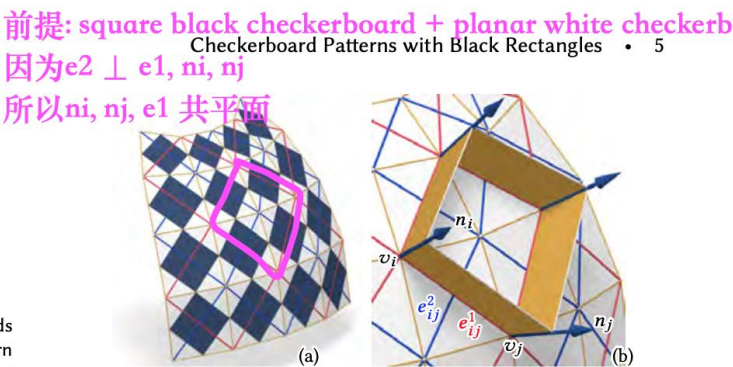


Fig. 7. A checkerboard pattern with black rectangles and planar white faces and both diagonal meshes of the control net, shown in blue and red, are discrete principal curvature parameterizations (a). The diagonal mesh pair allows us to define vertex normals such that connected vertices possess coplanar normals (b). This property facilitates the layout of support structures in architectural applications.

## 3.3 Planar white quads: discrete principal curvature parameterizations

For certain applications, e.g., architecture, it is very useful if not only the black faces, but all faces are planar. We simplify this requirement to only constrain the white quads to be planar and exempt white faces with more than four edges from this requirement. This is achieved if and only if the two diagonal meshes  $D_1, D_2$  in the control mesh are composed of planar quads (PQ meshes). A PQ mesh discretizes a so-called conjugate parameterization of a surface [Bobenko and Suris 2008], but here we also have orthogonality. The only orthogonal conjugate parameterizations are those where the iso-parameter curves are principal curvature lines. This means that a pattern  $P$  from black rectangles and planar white quads is a discrete principal curvature parameterization, or shortly, a principal mesh. Both the pattern  $P$  and the two diagonal meshes  $D_1, D_2$  of the control mesh are principal meshes (see Fig. 6).

Principal meshes have received a lot of interest, both within discrete differential geometry [Bobenko and Suris 2008] and applications such as architecture (see e.g. [Liu et al. 2006; Pottmann et al. 2007]). Here we have a new approach to principal meshes. We briefly outline some advantages and show that this is going beyond the currently used discretization.



**Diagonal Orthogonality.** For all the quads of  $C$ , we require their two diagonals to be orthogonal. This is the fundamental constraint for checkerboard patterns with black rectangles. We write the constraint as an energy term

$$E_{orth} = \sum_{k \in F} ((\mathbf{v}_{k1} - \mathbf{v}_{k3}) \cdot (\mathbf{v}_{k2} - \mathbf{v}_{k4}))^2, \quad (1)$$

**Planarity.** The black rectangles derived from the control mesh are automatically planar. The following constraint encodes the optional planarity for white quads. The planarity of white faces can be expressed by requiring the neighbouring vertices of each vertex  $\mathbf{v}_i$  to form a planar quad. We use the same planarity formulation of [Tang et al. 2014] and [Jiang et al. 2015],

$$E_{plan} = \sum_{i \in V} \sum_{(i,j) \in E} ((\mathbf{v}_i - \mathbf{v}_j) \cdot \mathbf{n}_i)^2 + \sum_i (\mathbf{n}_i \cdot \mathbf{n}_i - 1)^2, \quad (3)$$

where  $\mathbf{n}_i$  are the face normals of the white quads.

```
def con_principal_checkboard(self):
    """orthogonal + planarity (X +=[n])
    (v1-v3)*(v2-v4)=0
    n*(vi-vj)=0, n^2=1
    """
    w = self.get_weight('principal_checkboard')
    V = self.mesh.V
    num = self.mesh.num_quadface
    vi = self.mesh.quadface
    v1,v2,v3,v4 = vi[:,0:4],vi[:,1:4],vi[:,2:4],vi[:,3:4]
    c_v1 = np.r_[v1,V+v1,2*V+v1] # [x,y,z]
    c_v2 = np.r_[v2,V+v2,2*V+v2] # [x,y,z]
    c_v3 = np.r_[v3,V+v3,2*V+v3] # [x,y,z]
    c_v4 = np.r_[v4,V+v4,2*V+v4] # [x,y,z]
    Ho,ro = self._con_orthogonal_checkboard(self.X,c_v1,c_v2,c_v3,c_v4,num,self.N)
    Hp,rp = self._con_planarity_white_checkboard(self.X,V,self.N)
    H = sparse.vstack((Ho,Hp))
    r = np.r_[ro,rp]
    self.add_iterative_constraint(H*w, r*w, 'principal_checkboard')
```

```

def _con_orthogonal_checkboard(self,X,c_v1,c_v2,c_v3,c_v4,num,N):
    """for principal
    (v1-v3)*(v2-v4)=0
    """
    col = np.r_[c_v1,c_v2,c_v3,c_v4]
    row = np.tile(np.arange(num),12)
    d1 = X[c_v2]-X[c_v4]
    d2 = X[c_v1]-X[c_v3]
    d3 = X[c_v4]-X[c_v2]
    d4 = X[c_v3]-X[c_v1]
    data = np.r_[d1,d2,d3,d4]
    H = sparse.coo_matrix((data,(row,col)), shape=(num, N))
    r = np.einsum('ij,ij->i',d1.reshape(-1,3, order='F'),d2.reshape(-1,3, order='F'))
    return H,r

def _con_planarity_white_checkboard(self,X,V,N):
    """planar white quads
    n*(v1-v2)=n*(v2-v3)=n*(v3-v4)=0, n^2=1
    """
    num = self.mesh.num_regular
    _,v1,v2,v3,v4 = self.mesh.ver_regular_star.T
    c_v1 = self.mesh.columnnew(v1,0,V)
    c_v2 = self.mesh.columnnew(v2,0,V)
    c_v3 = self.mesh.columnnew(v3,0,V)
    c_v4 = self.mesh.columnnew(v4,0,V)
    c_n = self._N7_n-3*num+np.arange(3*num)
    H1,r1 = self._con_planarity(X,c_v1,c_v2,c_n,num,N)
    H2,r2 = self._con_planarity(X,c_v2,c_v3,c_n,num,N)
    H3,r3 = self._con_planarity(X,c_v3,c_v4,c_n,num,N)
    Hn,rn = self._unit(X,c_n,num,N)
    H = sparse.vstack((H1,H2,H3,Hn))
    r = np.r_[r1,r2,r3,rn]
    return H,r

```