Comparison of View Factor operational models

	Analytical formula	Double contour integration	Monte Carlo Ray Tracing	Infinitesimal Surface Approximation
Principle	Double integral on surfaces	Double integral on contours	Statistical method	Vector calculation
Resolution	Rarely possible	Ok for many simple shapes	Ok for any shape but not suitable for simple models	Ok for any shape
Precision	/	/	Ok by increasing complexity (increases with position of random rays)	Ok but not for close surfaces
Calculation time	/	worse	medium	good

1. Introduction : ISA - Infinitesimal Surface Approximation

There are many ways to derive the view factors between surfaces. In 2003, J. J. MacFarlane[1] introduced four methods in the VISARD system. Three of them were developed from the DCI method (Double contour integration) while the last one (equation 1.1), the infinitesimal surface method, has a simpler vector calculation.

Equation of ISA method:

$$F_{ji} = -\frac{1}{2A_j\pi}\frac{2A_jA_i}{|\vec{R}|^4}(\vec{n}_j \cdot \vec{R})(\vec{n}_i \cdot \vec{R}) \quad (1.1)$$

Then this equation can be reduced to:

$$F_{ji} = -\frac{2A_i}{2\pi|\vec{R}|^2}|\vec{n}_j| \cdot |\vec{n}_i| \cdot \cos\theta_i \cdot \cos\theta_j \quad (1.2)$$

Figure 1.1 shows the variables in equation where $\vec{R}$ is the vector links the centroid point of both surfaces, θ_i and θ_j are the vector angles between $\vec{R}$ and surfaces' normal vectors, $\vec{n}_i$ and $\vec{n}_j$. However, ISA method can only be applied on the surfaces who obey the five-times rule.

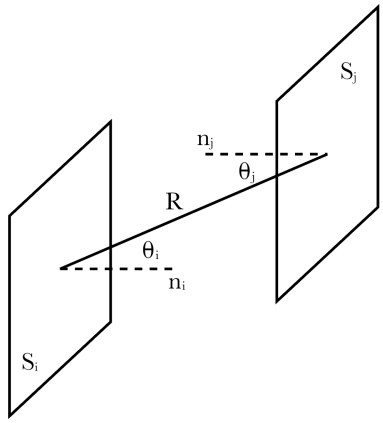

Figure 1.1: Infinitesimal surface approximation (ISA) simulation Chart

1.1 Five-times Rule

The five-times rule is the limitation of ISA method which shows in Figure 1.2. It means that to get a reliable result of view factor, the distance between two surfaces $|\vec{R}|$ should be bigger than five times the maximum projected width d_{max} of the source surface (emitter)[2]:

$$\frac{|\vec{R}|}{d_{max}} \leq 5 \quad (1.3)$$

Where the d_{max} is the diameter of the smallest external circle or the diagonal of the smallest external rectangle.

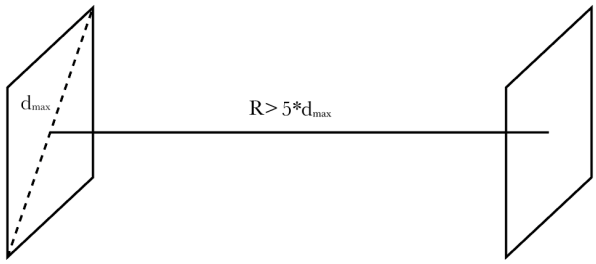

Figure 1.2: Five-times rule

The five-times rule ensures that our error rate in calculating the view factor is less than 2.5% (since our acceptable error range is less than 4%).

1.2 Methodology

In this section, I separate the functions who form the optimized ISA method. I use python as my computer language. Jupyter notebook is the software I use.

1.2.1 ISA method

Our input values are Geodataframe who include our study faces' properties like geometry, personal id, index, and label. We can see from the equation 1.2 that the normal vector, centroid, and area of the plane are the input values that we will use to calculate the view factor. It should be noted that the following program is based on the study of rectangular planes mainly and does not consider complex polygons.

1.2.1.1 Normal Vector

First, we need to select three points that are not co-linear according to the coordinates of the boundary points that make up the plane (Figure 1.3). Based on these three points in turn, two vectors $vec1$ and $vec2$ are calculated (equation 1.4, 1.5).

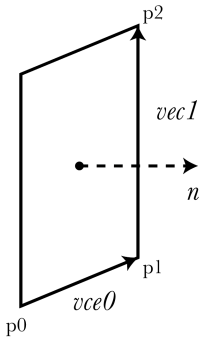

Figure 1.3: Normal vector

$$\overrightarrow{vec1} = \langle p1 - p0 \rangle = \langle x1 - x0, y1 - y0, z1 - z0 \rangle = \langle x_{v1}, y_{v1}, z_{v1} \rangle \quad (1.4)$$
$$\overrightarrow{vec2} = \langle p2 - p1 \rangle = \langle x2 - x1, y2 - y1, z2 - z1 \rangle = \langle x_{v2}, y_{v2}, z_{v2} \rangle \quad (1.5)$$

The surface normal vector is calculated as:

$$\vec{n} = \overrightarrow{vec1} \times \overrightarrow{vec2} = \langle (y_{v1}z_{v2} - z_{v1}y_{v2}), (z_{v1}x_{v2} - x_{v1}z_{v2}), (x_{v1}y_{v2} - y_{v1}x_{v2}) \rangle \quad (1.6)$$

Since the normal vector of the plane is the unit vector. The result should be normalized. In addition, the direction of normal vector depends on the order of three points because cross product meet right-hand rule. This is very important when you face the building surfaces because unifying the directional rules of the normal vectors of the building wall planes can simplify your operations later.

```
def module(vector):
    return np.sqrt(np.dot(np.array(vector),np.array(vector)))

def SurfaceNormal(self):
    coord = self.exterior.coords
    facet_p0=np.array(coord[0])
    facet_p1=np.array(coord[1])
    facet_p2=np.array(coord[2])
    vec1 = facet_p1 - facet_p0
    vec2 = facet_p2 - facet_p1
    a = vec1[1]*vec2[2]-vec1[2]*vec2[1]
    b = vec1[2]*vec2[0]-vec1[0]*vec2[2]
    c = vec1[0]*vec2[1]-vec1[1]*vec2[0]
    normal = np.array([a,b,c])
    mod = module(normal)
    n  = normal/mod
    return n
```

1.2.1.2 Centroid Point of Faces

In the models I have studied only rectangular planes are generally considered (including building facades or ground floors). Also, Python now has many methods for finding polygon centroids, so I will only consider the simplest case here (Figure 1.4).

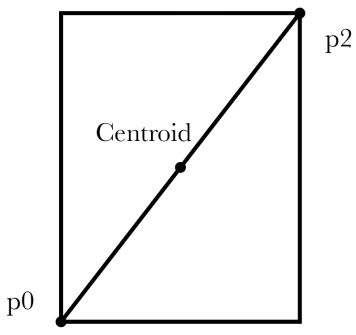

Figure 1.4: Centroid point of face

$$Centoid = \left(\frac{x_{p0}+x_{p2}}{2}, \frac{y_{p0}+y_{p2}}{2}, \frac{z_{p0}+z_{p2}}{2}\right) \quad (1.7)$$

The function centroid(self) returns the centroid point of a rectangular face as type of Point.

```
def centroid(self):
    coord = self.exterior.coords
    face_p0=np.array(coord[0])
    face_p2=np.array(coord[2])
    face_ce=(face_p0+face_p2)/2
    return Point(face_ce)
```

1.2.1.3 Surface Area

The area of a rectangle face is equal to length times width. In our case, it should be the module of $vec0$ times mudule of $vec1$ (Figure 1.4). The functions area3d(self) and distance3d(point1,point2) present the algorithm which shows the difference compared to the existing method of calculating 2d planes area in Python.

```
def distance3d(point1,point2):
    d = sqrt((point1.x-point2.x)**2 + (point1.y-point2.y)**2 +
(point1.z-point2.z)**2)
    return d

def area3d(self):
    coord = self.exterior.coords
    d1 = distance3d(Point(coord[0]),Point(coord[1]))
    d2 = distance3d(Point(coord[1]),Point(coord[2]))
    return d1*d2
```

1.2.1.4 View Factor ISA Calculator

According to the equation 1.1, we can build a function whose input values are two faces. During our calculations, we need to pay attention to the directionality of the program because the order of the input values may lead to different results.

```
def isa(face1,face2):#view factor from 1 to 2 (F12)
    n1=SurfaceNormal(face1.geometry)
    n2=SurfaceNormal(face2.geometry)
    a2=area3d(face2.geometry)
    vec_R=np.array([face1.CentroidPoint.x-
face2.CentroidPoint.x,face1.CentroidPoint.y-
face2.CentroidPoint.y,face1.CentroidPoint.z-face2.CentroidPoint.z])
    vf=-a2*np.dot(n1,vec_R)*np.dot(n2,vec_R)/(pi*module(vec_R)**4)
    return vf
```

1.2.2 Analytical Method

The view factor can be calculated by using analytical method under specific conditions: (1) parallel surfaces, (2) perpendicular surface. These equations can give us a realistic reference value when we are testing a new model.

1.2.2.1 Parallel Surfaces

For two finite parallel plates who share the same size, the view factor can be found analytically or numerically. The independent variable are the size and position of plates (Figure 1.4). The view factor for aligned parallel rectangles is calculated as follow:

$$F_{ij} = \frac{2}{\pi AB}\left[\ln\left(\frac{(1+A^2)(1+B^2)}{1+A^2+B^2}\right)^{0.5} + \left(A\sqrt{1+B^2}\right)\cdot\tan^{-1}\left(\frac{A}{\sqrt{1+B^2}}\right) + \left(B\sqrt{1+A^2}\right)\cdot\tan^{-1}\left(\frac{B}{\sqrt{1+A^2}}\right) - A\tan^{-1}A - B\tan^{-1}B\right] \quad (1.8)$$

where $A = \frac{W}{L}$, $B = \frac{H}{L}$.

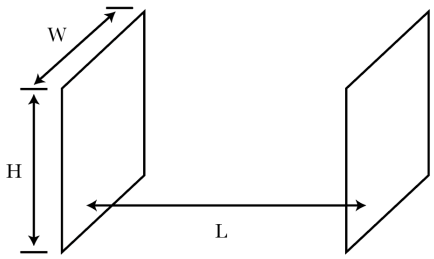

Figure 1.4: Finite Parallel Plates

The implementation in code is as follows:

```
def ParallelVF(a,b):#a=width/distance,b=height/distance
    f = 2/(pi*a*b)*(log(((1+a**2)*(1+b**2)/(1+a**2+b**2))**0.5)+
                    (a*(1+b**2)**0.5)*atan(a/(1+b**2)**0.5)+
                    (b*(1+a**2)**0.5)*atan(b/(1+a**2)**0.5)-
                    a*atan(a)-
                    b*atan(b))
    return f
```

1.2.2.2 Perpendicular Surfaces

For the perpendicular rectangles with a common edge (Figure 1.5), the view factor between them can also be calculated numerically. The view factor depends on their own aspect ratio. The equation is as follow:

$$F_{ij} = \frac{1}{\pi B}\left[B \cdot \tan^{-1}\left(\frac{1}{B}\right) + A \cdot \tan^{-1}\left(\frac{1}{A}\right) - \left(\sqrt{A^2+B^2}\right) \cdot \tan^{-1}\left(\frac{1}{\sqrt{(A^2+B^2)}}\right) + \frac{1}{4}\ln\left(\frac{(1+B^2)(1+A^2)}{1+B^2+A^2} \cdot \left[\frac{A^2(1+B^2+A^2)}{(1+A^2)(B^2+A^2)}\right]^{A^2} \cdot \left[\frac{B^2(1+B^2+A^2)}{(1+B^2)(B^2+A^2)}\right]^{B^2}\right)\right] \quad (1.9)$$

where $A = \frac{H}{W}$, $B = \frac{L}{W}$.

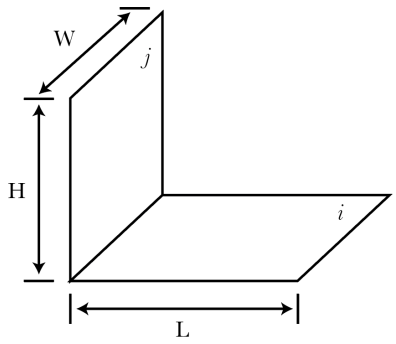

Figure 1.5: Finite rectangles with a common edge

The implementation in code is as follows:

```
def VerticalVF(a,b):#a = y/w , b = x/w
    f = 1/(pi*b)*(b*atan(1/b)+a*atan(1/a)-
((a**2+b**2)**0.5)*atan(1/((a**2+a**2)**0.5))+0.25*log((1+b**2)*(1+a**
2)/(1+b**2+a**2)*((b**2*(1+b**2+a**2)/((1+b**2)*(b**2+a**2)))**(b**2))
*(a**2*(1+a**2+b**2)/((1+a**2)*(a**2+b**2)))**(a**2)))
    return f
```

1.2.3 Subdivision

Since five-times rule cannot be always met when we are considering the view factor between surfaces in a city block or any occasion with high distribution density. Also, ISA method cannot be simply applied on the two attached vertical surfaces. In such case, we must subdivide our source surfaces (and our target surfaces in some cases). I divide these situations into two common ones: (1) parallel surfaces; (2) vertical surfaces. The reason why I considering these two cases is that we already have analytical formula of these two cases which could provide me a reference value of view factor.

1.2.3.1 Subdivide Rectangle

As a preliminary, we built a function that can split the rectangle into any number of meshes. As part of subsequent iterations, we expect this function to return a Geodataframe with geometry, centroid point, and max diameter. Input values are surface, row number and column number.

```
#grid surface3d into meshes of size n*m
def grid3d(self,n,m):#n rows, m columns
    coord = self.exterior.coords
    p0=np.array(coord[0])
    p1=np.array(coord[1])
    p2=np.array(coord[2])
    p3=np.array(coord[3])
    vec1 = (p1-p0)/m
    vec2 = (p2-p1)/n
    poly=[]
    cen_p=[]
    sur_d=[]
    for i in range(n):
        for j in range(m):
            p0_t=Point(p0+j*vec1+i*vec2)
            p1_t=Point(p0+(j+1)*vec1+i*vec2)
            p2_t=Point(p0+(j+1)*vec1+(i+1)*vec2)
            p3_t=Point(p0+j*vec1+(i+1)*vec2)
            face_t = Polygon([p0_t,p1_t,p2_t,p3_t,p0_t])
            poly.append(face_t)
            cen_p.append(centroid(face_t))
            sur_d.append(Surface_diameter(face_t))
    self_grid=GeoDataFrame({'geometry': poly},crs='epsg:2154')
    self_grid['CentroidPoint']=cen_p
    self_grid['diameter']=sur_d
    return self_grid
```

1.2.3.2 Parallel Surfaces

The view factor between parallel planes is very common in simulating the heat transfer between buildings. In this case, we first must determine if the diameter of source surface and the distance between their centroid points meet five-times rule. If it does, we can calculate view factor directly using function isa(face1,face2). Otherwise, we need to subdivide the source plane equally according to the maximum size obtained by the five-times rule (Figure 1.6). The source surface is split into a rectangular grid with n rows and m columns (both m and n are integers).

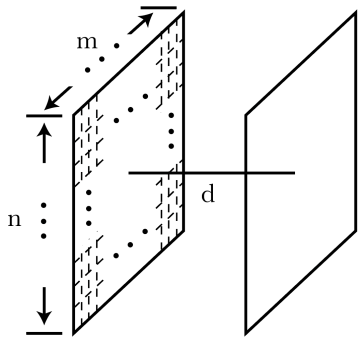

Figure 1.6: Subdivide source surface

To avoid errors due to incomplete division, the values of n and m are equal to the quotient plus 1. The code is implemented as follows:

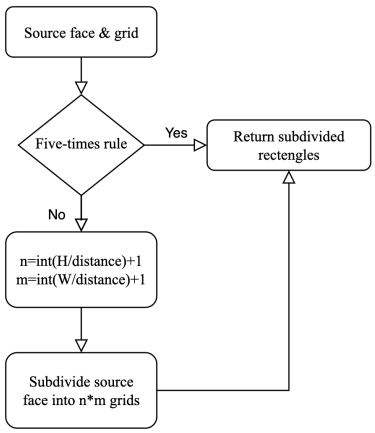

Figure 1.7: Flow chart of subdivide function (parallel surfaces)

```
def isFiveTimeRuleOk(f1,f2):
    return distance3d(f1.CentroidPoint,f2.CentroidPoint) >=
5*f1.diameter

def Subdivide2(f1,f2): #Subdivede f1
    if isFiveTimeRuleOk(f1,f2):
        return [f1]
    else:
        d = distance3d(f1.CentroidPoint,f2.CentroidPoint)/5
        coord1 = f1.geometry.exterior.coords
        w1 = distance3d(Point(coord1[0]),Point(coord1[1]))
        h1 = distance3d(Point(coord1[1]),Point(coord1[2]))

        n1 = int(h1/d)+1
        m1 = int(w1/d)+1

        return grid3d(f1.geometry,n1,m1)
```

1.2.3.3 Perpendicular Surfaces

Vertical planes generally occur between the building facade and the ground. In our study, because the shape of the ground is not stable (cut by buildings bottom) we artificially divide the ground into very small grids (Figure 1.8). So the view factor calculation between the vertical planes is equal to the combine of view factor from source plane to ground grid (equation 1.10).

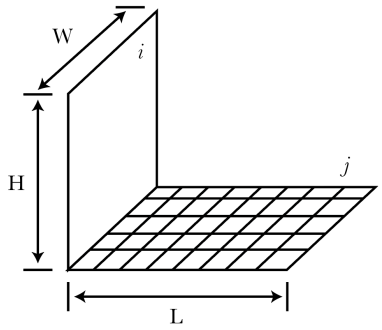

Figure 1.8: Perpendicular surfaces

$$F_{ij} = \underbrace{F_{ij_0} + F_{ij_1} + \cdots + F_{ij_{n-1}} + F_{ij_n}}_{n=number\ of\ grids} = \sum_n^x F_{ij_x} \quad (1.10)$$

In order to calculate the view factor between source surface and ground grid, we first have to check their centroid points distance. If it meets five-times rule, we can do the calculation. If it doesn't, we must subdivide our source face. This subdivide process needs to be iterated until all the subdivided grids satisfy the five-times rule (Figure 1.9).

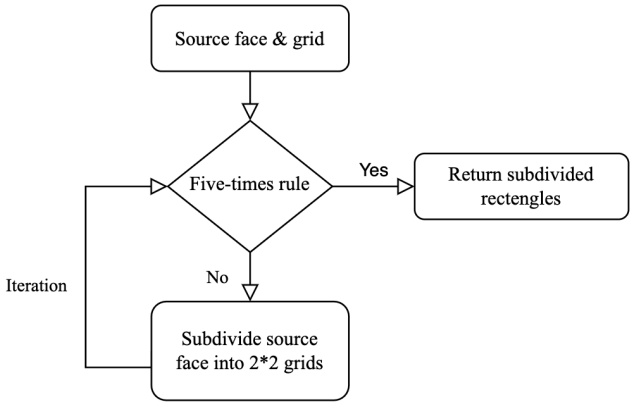

Figure 1.9: Flow chart of subdivide function (perpendicular surfaces)

```
def Subdivide(f1,f2): #f1 is the emitting surface, f2 is the target
surface
    if isFiveTimeRuleOk(f1,f2):
        return [f1]
    else:
        result = []
        f_grid = grid3d(f1.geometry,2,2)
        for i in range(len(f_grid)):
            if isFiveTimeRuleOk(f_grid.iloc[i],f2):
                result.append(f_grid.iloc[i])
            else:
                result += Subdivide(f_grid.iloc[i],f2)
        return result
```

Here are some examples to show more visually how this algorithm works (Figure 1.10). When the distance between source surface and ground grid is quite small or even share the same edge, the

closer the area to the grid, the greater the degree of subdivision. As the ground grid gradually moves away from the source surface, the size of the subdivided grids starts to get larger, and the number starts to get smaller.

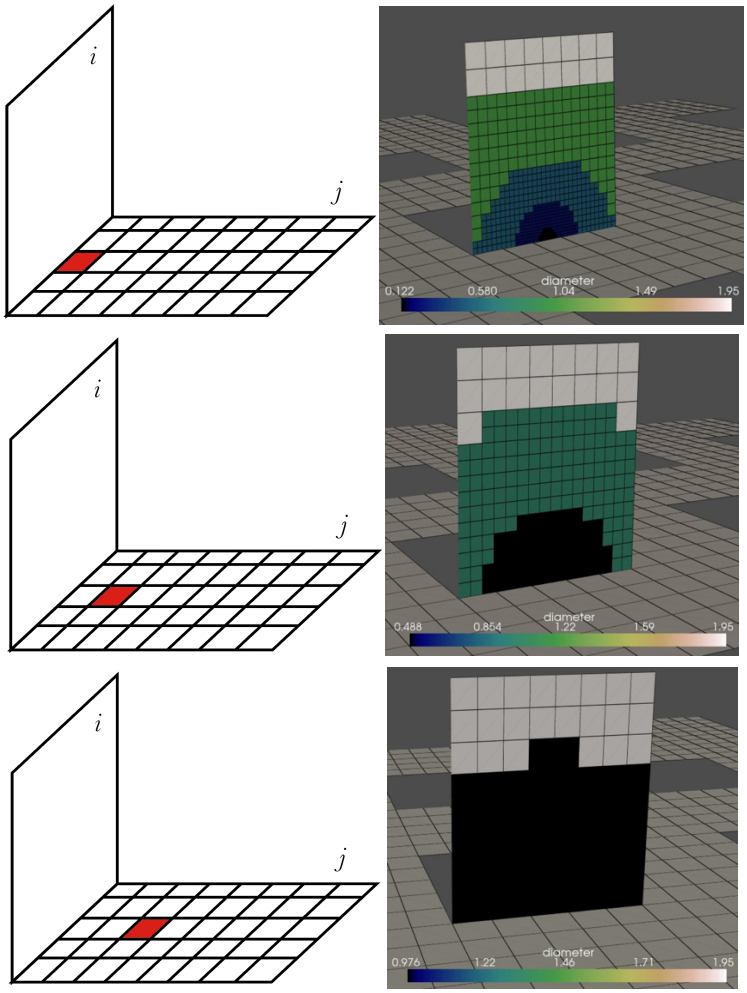

Figure 1.10: ground grid position (left, red grid is target grid), subdivide result of source face (right, color represent their diameter)

1.3 Results And Discussion

The subjects of this experimental test are finite parallel planes and finite planes with a common edge. The current implementation and improvement of the ISA model is to obtain a more efficient computing speed while satisfying the accuracy. I set two criteria parameters for each of these two aspects.

1.3.1 Criteria Parameters

Since analytical formula provide us reliable result of view factor under simple condition like I mentioned, the percentage error of ISA model is calculated by being compared with analytical formula:

$$error\% = \frac{|F_{analytical} - F_{ISA}|}{F_{analytical}} \times 100\% \quad (1.11)$$

In addition, the computation time is obtained from the program that comes with Python.

1.3.2 Parallel Surfaces

As mentioned earlier, applying the optimized ISA model to the source surface can improve the accuracy of the calculation but it may also increase the error if the size of the subdivided grid differs significantly from our target plane. Therefore, I have conducted experiments for two cases: (1) subdivide only the source surface; (2) subdivide both the source surface and the target surface.

The parallel rectangle surfaces with $H \times W$ dimensions are separated by distance L (Figure1.11). I set the rectangle to be a square ($W = H = D$). I take the ratio of the square side length to L as the independent variable and calculate the view factor of the ISA model.

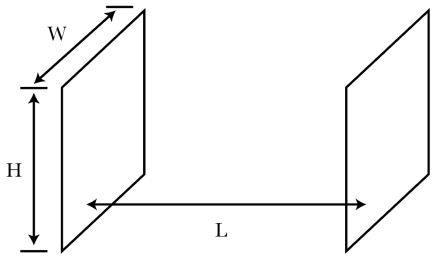

Figure 1.11: Parallel square surfaces

1.3.2.1 Subdivide source surface

In this case, I only apply subdivision function on source surface. The results are shown in Table 1.1 and Figure 1.12.

Table 1.1: View Factor for different dimensions

	D/L						
	1.4	1.2	1.0	0.8	0.6	0.4	0.2
Analytical	0.2980	0.2508	0.1998	0.1464	0.0934	0.0461	0.0124
ISA method	0.3817	0.3127	0.2407	0.1691	0.1030	0.0486	0.0126
Error%	28.066	24.680	20.471	15.562	10.314	5.3766	1.6288
Time (s)	0.0916	0.0651	0.0506	0.0386	0.0559	0.0219	0.0163

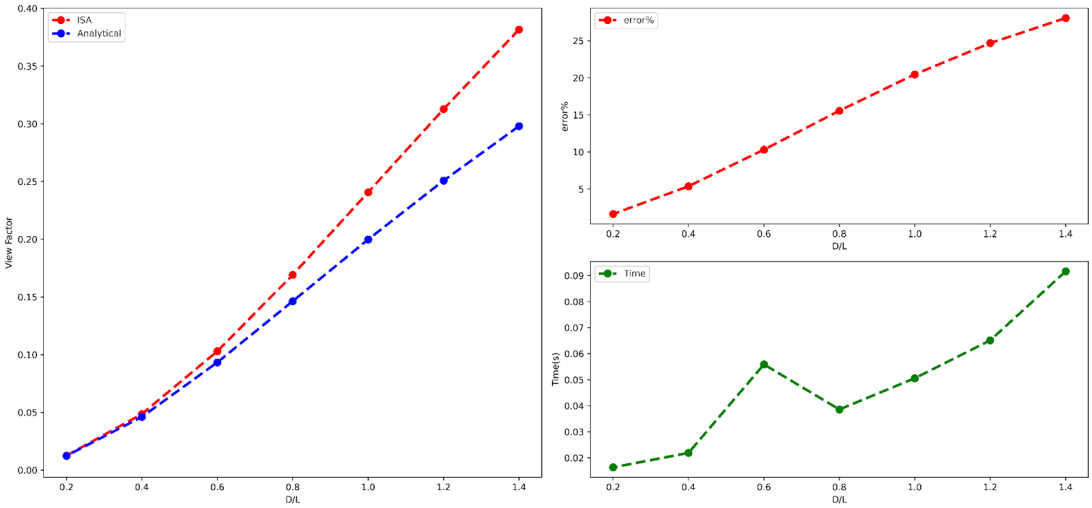

Figure 1.12: View Factor Data Analysis Charts: view factor results by ISA model and Analytical formula (left); percentage error of ISA model (right above); Calculation timing (right down)

```
L1 = [1.4,1.2,1,0.8,0.6,0.4,0.2]
error1 = []
time1 = []
vf1 = []
vf_a = []
for j in range(len(L1)):
    start = timeit.default_timer()
    f1,f2 = CreateFaces(L1[j],L1[j],'parallel')
    d = GeoDataFrame(Subdivide2(f1.iloc[0],f2.iloc[0]))
    vftest = 0
    for i in range(len(d)):
        face_0 = d.iloc[i]
        vf1_isa = isa(face_0,f2.iloc[0])
        vftest = vftest + vf1_isa/len(d)
    vf1.append(vftest)
    vf_a.append(ParallelVF(L1[j],L1[j]))
    stop = timeit.default_timer()
    error1.append((vftest-
ParallelVF(L1[j],L1[j]))/ParallelVF(L1[j],L1[j])*100)
    time1.append(stop-start)
```

My research subjects here are focusing on the parallel surfaces whose D/L bigger than 0.2 which also means they don't meet five-times rule. The result shows that, if I only subdivide my source surface, the error rate does not meet demand (lower than 4%) from D/L higher than 0.2. Since calculation times are few so they only take less than 0.1 seconds to finish.

1.3.2.2 Subdivide source surface and target surface

If we subdivide source surface and target surface, the performance of ISA method is as follow:

Table 1.2: View Factor for different dimensions

	D/L						
	1.4	1.2	1.0	0.8	0.6	0.4	0.2
Analytical	0.2980	0.2508	0.1998	0.1464	0.0934	0.0461	0.0124
ISA method	0.2999	0.2527	0.2015	0.1478	0.0943	0.0466	0.0125
Error%	0.6416	0.7369	0.8471	0.9612	1.042	0.9955	0.6399
Time (s)	2.6163	1.6033	0.9299	0.4914	0.1873	0.0735	0.0309

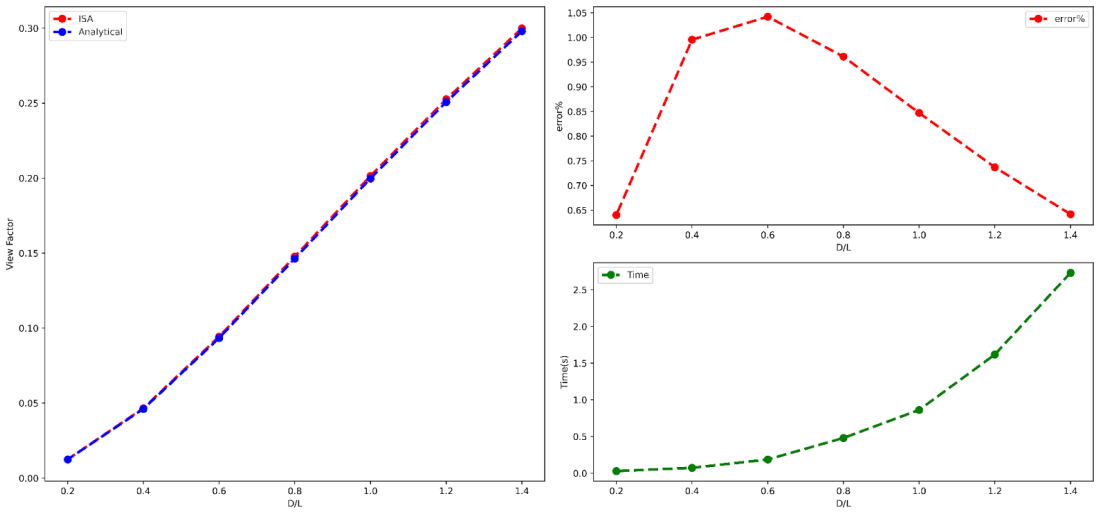

Figure 1.13: View Factor Data Analysis Charts: view factor results by ISA model and Analytical formula (left); percentage error of ISA model (right above); Calculation timing (right down)

```
L2 = [1.4,1.2,1,0.8,0.6,0.4,0.2]
error2 = []
time2 = []
vf2 = []
vf_a = []
for j in range(len(L2)):
    start = timeit.default_timer()
    f1,f2 = CreateFaces(L2[j],L2[j],'parallel')
    d = GeoDataFrame(Subdivide2(f1.iloc[0],f2.iloc[0]))
    c = GeoDataFrame(Subdivide2(f2.iloc[0],f1.iloc[0]))
    vftest = 0
    for i in range(len(d)):
        face_0 = d.iloc[i]
        vf2_isa = [isa(face_0,face_1) for face_1 in c.iloc]
        vftest = vftest + sum(vf2_isa)/len(d)
    vf2.append(vftest)
    vf_a.append(ParallelVF(L2[j],L2[j]))
    stop = timeit.default_timer()
    error2.append((vftest-
ParallelVF(L2[j],L2[j]))/ParallelVF(L2[j],L2[j])*100)
    time2.append(stop-start)
```

Obviously, if we subdivide both source plane and target plane, the results fit better with the actual values. Also, the error percentage this time surely meet our demand and most of times lower than 1%. As for the calculation timing, it will increase since we increase our target surfaces but still, its results are acceptable.

1.3.3 Perpendicular Surfaces

For the case of vertical planes, I subdivide the plane considered as ground first (Figure 1.14). Similarly, in my experiments, I set the source surface and the ground plane to be squares with common edge. The independent variable in this experiment is the number of ground planes that are subdivided ($n \times n$). The result is shown on Table 1.3 and Figure 1.15.

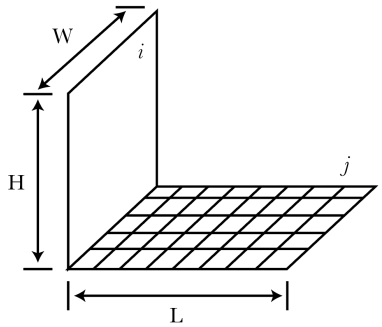

Figure 1.14: Perpendicular square surfaces share the same edge

Table 1.3:

	Grid number ($n \times n$)						
n	2	3	4	5	6	7	8
Analytical	0.20004	0.20004	0.20004	0.20004	0.20004	0.20004	0.20004
ISA method	0.1983	0.1994	0.1998	0.1999	0.20003	0.20008	0.20012
Error%	0.8813	0.3191	0.1342	0.0501	0.0053	0.0204	0.0390
Time (s)	2.9046	6.7047	12.4096	18.9903	28.4300	39.9939	52.2002

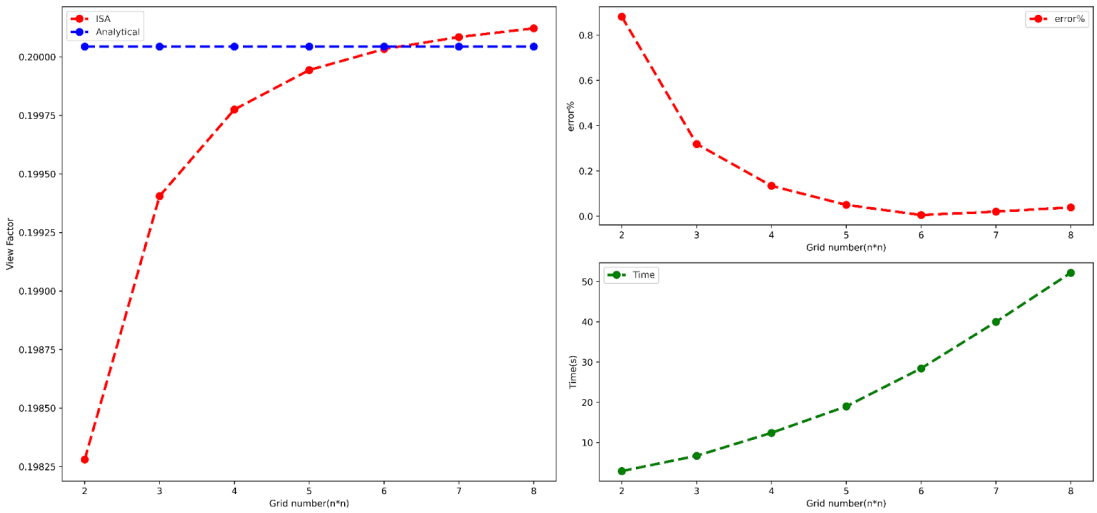

Figure 1.15: View Factor Data Analysis Charts: view factor results by ISA model and Analytical formula (left); percentage error of ISA model (right above); Calculation timing (right down)

```
f3,f4 = CreateFaces(1,1,'vertical')
L3 = [2,3,4,5,6,7,8]
viewfactor = []
error3=[]
time3=[]
vf_a_v=[]
for i in L3:
    start = timeit.default_timer()
    grid = grid3d(f4.geometry.squeeze(),i,i)
    vf = []
    for j in range(len(grid)):
        f4_grid = grid.iloc[j]
        list_sub = Subdivide(f3.iloc[0],f4_grid)
        testgeo = GeoDataFrame(list_sub)
        area = [area3d(item) for item in testgeo.geometry]
        testgeo['area']=area
        vf_isa = [isa(tg,f4_grid)*tg.area/100 for tg in testgeo.iloc]
        vf.append(sum(vf_isa))
    viewfactor.append(sum(vf))
    vf_a_v.append(VerticalVF(1,1))
    error3.append(abs(VerticalVF(1,1)-sum(vf))/VerticalVF(1,1)*100)
    stop = timeit.default_timer()
    time3.append(stop-start)
```

Since the ground plane has been subdivided in our study, the error rate shows that this approach is successful. Even if the ground plane is divided into 2×2 equal squares, the error rate is still less than 1%. Similarly, increasing the number of ground divisions significantly increases the computation time and stabilizes the value of the view factor.

1.4 Conclusion

The ISA model itself is not a perfect model, and we must optimize the original model for sufficient accuracy. We use the features of five-times rule to subdivide both source plane and target plane so that they can satisfy the five-times rule.

For the parallel plane, we tested the subdivision of only the source plane and the subdivision of both the source plane and the target plane. It is found that subdividing both planes at the same time is efficient and greatly increases the accuracy. Although parallel planes are often found in city blocks, there are more planes that are perpendicular to the ground but not parallel, so this optimization model may still be inadequate.

For the vertical plane, we tested the effect of subdivision the density of the ground on the view factor results. We found that increasing the density of the ground grid improves the accuracy of the model but also increases the computation time significantly. Such results suggest that meaninglessly increasing the density of the ground grid may not have a significant impact on the results.

In summary, the ISA model has obvious advantages over the Monte Carlo ray tracing model, with short computation time and stable data. However, the ISA model is not applicable to the view factor calculation for all shapes.

There is room for improvement in this experiment and in the code, e.g., the computation speed in other languages may be very different and I did not use Python multiprocessing in this experiment, otherwise the computational speed would be significantly improved.

Bibliography:

[1] MacFarlane, J.J.. (2003). VISRAD—A 3-D view factor code and design tool for high-energy density physics experiments. Journal of Quantitative Spectroscopy and Radiative Transfer. 81. 287-300. 10.1016/S0022-4073(03)00081-5.

[2] Ashdown, Ian. (1994). Radiosity - a programmer's perspective.