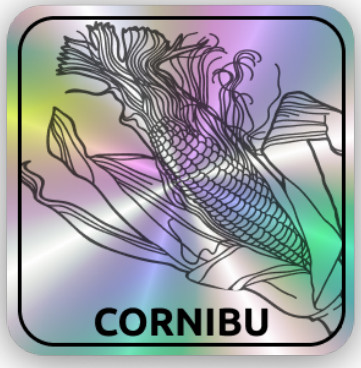
CORNIBU: Python Documentation of the canopy architecture structure model

Mario Serouart, Raul Lopez Lozano

ABSTRACT

Normally, all the functions are well described in the raw Python script. However, if you still have issues/questions, please open an issue in the Github repo.
Here is a short overall description supported with figures to better apprehend and understand how it works step-by-step.

If you used this model for any reason, or considered it as useful for any work, please cite : **LINK PAPER**



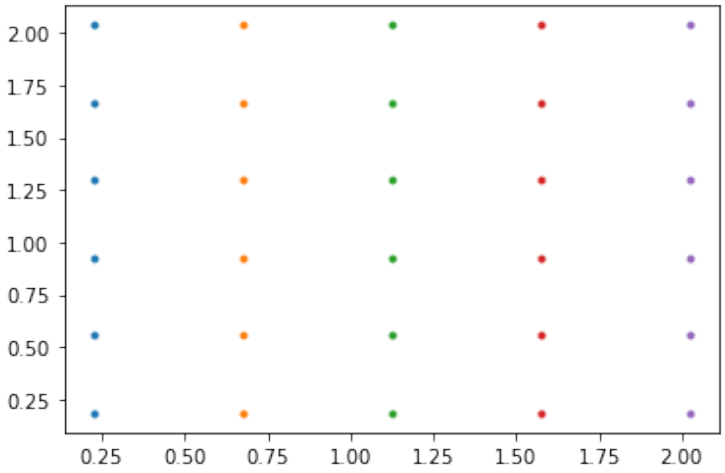
Code explanation

Sowing pattern

This snippet define the maize stem placement in the virtual canopy, i.e. the sowing pattern.

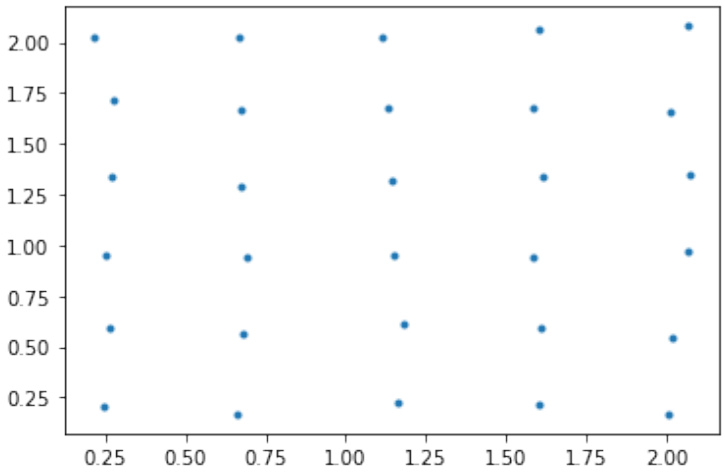
The number of rows can be determined earlier in code. For our specific study, 5 rows were chosen. The number of plants -and extensively the final mock-up size- will be determined by density and inter-row distance.

```
scene_side = inter_rang*5
dist_plt = 5/(scene_side*density)
Li = np.arange(dist_plt/2, scene_side, dist_plt)
y = np.tile(Li,5)
Lix = np.arange(inter_rang/2, scene_side, inter_rang)
x = np.tile(Lix,len(y/5))
x_unique = np.unique(x)
y_unique = np.unique(y)
xx, yy = np.meshgrid(x_unique, y_unique)
xyz = np.stack((xx, yy, np.zeros_like(xx)), axis=-1).reshape(-1, 3)
```



To avoid a too perfect mathematical simulated crop model, we add some noise around each plant/stem position. Thus, up to 10 cm are randomly added in both x- and y-axis.

```
xydelta = 0.1
newy = (np.random.rand(len(xyz), 1)*xydelta*2*dist_plt) - (dist_plt*xydelta/2)
np.asarray(xyz)[:, 1] += newy[:, 0]
xyz[:, 0] += newy[:, 0]
```



Individual plants considerations
Length, Width and Ratio values of each leaf

Leafsize function describes leaf shape (length and width), according to a ratio dependency between two factors.
Inputs : **St** (Maximum leaf area per plant - m²), **it** (maximum number of leaves - int)

According to : España, Marisa Frederic, Baret Aries, Franck Chelle, Michael Andrieu, Bruno Prévot, Laurent. (1999). Modeling maize canopy 3D architecture. Ecological Modelling 122. 25-43. 10.1016/S0304-3800(99)00070-8.

The leaf lamina is spread out onto a flat plane surface, the central rib being the symmetry axis. The relationship between the width of the leaf (w) at a given distance to the insertion point (l) describes the shape of the leaf. Leaf length and width were normalized respectively to the maximum length (lt) and width (wt) of the leaf. The relationship between leaf width and length can be described by a parabolic function.

$$w^*(l^*) = \alpha l^{*2} + \beta l^* + \gamma$$

where $l^* = l/lt$ and $w^* = w/wt$ are the normalized length and width. The boundary conditions on the point (l^*w) corresponding to the maximum width (for $l^* = l^*w$, $dw^*/dl^* = 0$ and $w^* = 1$) and at the tip of the leaf ($w^*(1) = 0$) reduce the number of parameters to only one such as a for example. It thus allows to compute the dimensionless area defined as:

$$S^*(\alpha) = \int_{l^*}^{l} w^* dl^*$$

The leaf area can be estimated from S^* and the maximum dimensions of the leaf, wt and lt, according to:

$$S = S^* l_t w_t$$

Observations show that an average value of $a = 3.0$ provides a good description of the leaf shape. This corresponds to a value of 0.72 for S^* that is in good agreement with experimental observations of *McKee (1964)* and *Keating and Wafula (1992)*. We will now focus on leaf size described by the total leaf dimensions (*wt* and *lt*).

Leaf size The maximum leaf dimensions obviously vary from plant to plant and depend on the leaf order(i). The leaf width is described by a polynomial function of the leaf order i:

$$w_t(i) = awi^3 + bwi^2 + cwi + dw$$

Observations show that the width of the first leaf can be assumed to be constant *wt(1)= 0.015m*. We impose the function to be minimum for the first leaf. It thus leads to:

$$w_t(i) = aw(-2i^3 + 3i^2 - 1) + bw(-i^2 + 2i - 1) + wt(1)$$

Parameters *aw* and *bw* are then expressed as a function of the order of the widest leaf (*i(wmax)*) and the width of the widest leaf (*wmax*). This leads to:

$$aw = \frac{w_{max} - wt(l)}{i(w_{max})^3 - 3i(w_{max}) + 2 - 3/2(i(w_{max})^2 - 1)(i(w_{max}) - 1)}$$

$$bw = \frac{-3aw(i(w_{max})^2 - 1)}{2(i(w_{max}) - 1)}$$

This reparameterized model provides a better interpretation of the parameters which are now *i(wmax)* and *wmax*. These two parameters are then related to the maximum leaf number (*it*) and vigor(*St*) through a simple linear regression:

$$i(w_{max}) = 1.59 + 0.65it$$

$$w_{max} = 0.066 - 0.0005it + 0.085St$$

The model evaluated on the test data set (Fig.1b) provides a good estimation of leaf width. Leaf length is approximated by a linear function of leaf order i up to the first seven leaves. For the remaining leaves, a parabolic function is used:

$$\begin{cases} 1 < i7 & lt = \alpha l + \beta l \\ 7 < i < it & lt = a_l i^2 + b_l i + c_l \end{cases}$$

Similarly to the leaf width and height of insertion, the length of the first leaf is not very variable and is set to its average value: *lt(1) = 0.04m*. If we impose the continuity between the two models for leaf seven, then parameters *al* and *bl* can be derived from *lt(1)* and parameters *al*, *bl* and *cl*. For leaves above leaf seven, the model is again re-parameterized using the length of the longest leaf (*lmax*) and the order of the longest leaf *i(lmax)* which is always above leaf seven. This gives:

$$\begin{gathered} cl = l_{max} + a_l i(l_{max})^2 \\ bl = -2a_l i(l_{max}) \\ \beta_l = l_t(1) - \alpha_l \\ \alpha l = \frac{a_l 7^2 + b_l 7 + c_l - l_t(1)}{7 - 1} \end{gathered}$$

We note that all the parameters of Eq. resume to *lt(1)*, *lmax*, *i(lmax)* and *al*. Both parameters *lmax* and *i(lmax)* were estimated from the cumulated leaf area (*St*) and the total number of leaves (*it*):

$$i(l_{max}) = 5.81 + 0.31 i_t$$

$$l_{max} = 0.99 - 0.04 i_t + 0.94 S_t$$

The last parameter *al* derived from the constraint on the cumulated plant leaf area where the only unknown is the parameter *al* in *lt(i)*:

$$S_t = S^* \sum_{i=1}^{i=i_t} w_t(i) l_t(i)$$

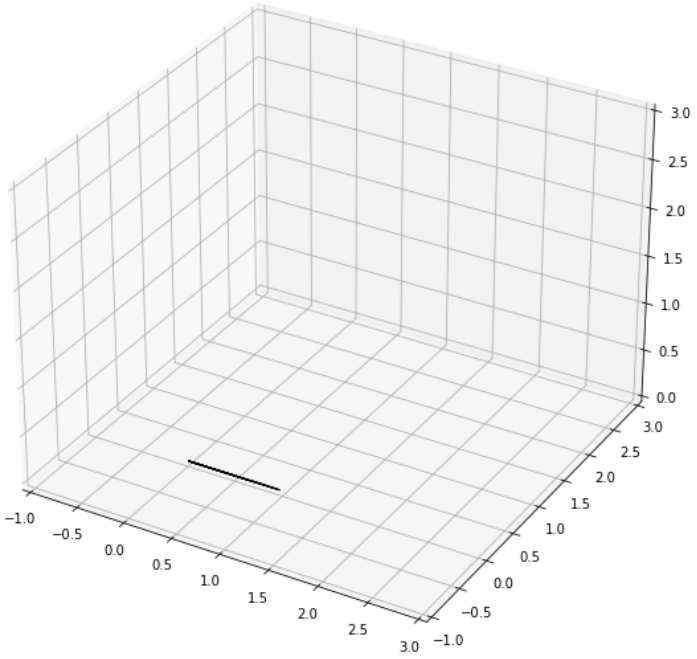
Evaluation on test dataset shows that the model is in good agreement with the observations.

```
def leafsize(St, it):

    matt = []
    for al in range(-1,2,1) :
        wl = 0.015
        ll = 0.04

        wtt = []
        i = [i for i in np.arange(1, it+1, 1)]
        for ii in i :
            iwmax = 1.59+0.65*it
```

```
            wmax = 0.066 - 0.0005*it + 0.085*St
            aw = (wmax - w1)/((iwmax**3 - 3*iwmax + 2) - (3/2)*(iwmax**2-1)*(iwmax-1))
            bw = (-3*aw*(iwmax**2 - 1))/(2*(iwmax - 1))
            wt = aw*(ii**3-3*ii+2) + bw*(ii**2-2*ii+1) + w1
            wtt.append(wt)
        wt = wtt

        lmax = 0.99 - 0.04*it + 0.94*St
        ilmax = 5.81 + 0.31*it
        il17 = [i for i in np.arange(0, 7, 1)]
        ilrest = [i for i in np.arange(8, it + 1, 1)]

        alphal = (al*7**2 - 2*al*ilmax*7 + lmax + ilmax**2*al - l1)/(7 - 1)

        lt17 = (np.asarray(alphal)*il17) + l1
        ltrest = al*np.power(ilrest,2) - 2*np.asarray(al)*np.asarray(ilrest)*ilmax + lmax + np.power(ilmax,2)*al
        lt = np.concatenate((lt17, ltrest))

        ajuste=(0.72*np.sum(wt*lt)) - St
        mat = matt.append([al, ajuste])

    mat = np.asarray(matt)
    coef = np.polyfit(mat[:, 0], mat[:, 1], 1)

    al=-coef[1]/coef[0]

    w1=0.015
    l1=0.04
    i = [i for i in np.arange(1, it+1, 1)]
    wtt = []
    for ii in i :
        iwmax = 1.59+0.65*it
        wmax = 0.066 - 0.00065*it + 0.085*St
        aw = (wmax - w1)/((iwmax**3 - 3*iwmax + 2) - (3/2)*(iwmax**2-1)*(iwmax-1))
        bw = (-3*aw*(iwmax**2 - 1))/(2*(iwmax - 1))
        wt = aw*(ii**3-3*ii+2) + bw*(ii**2-2*ii+1) + w1
        wtt.append(wt)
    wt = wtt

    lmax=0.99-0.04*it+St
    ilmax=5.81+0.27*it
    il17 = [i for i in np.arange(0, 7, 1)]
    ilrest = [i for i in np.arange(8, it + 1, 1)]
    alphal=(al*7**2-2*al*ilmax*7+lmax+ilmax**2*al-l1)/(7-1)
    lt17 = (np.asarray(alphal)*il17) / 1.5 + l1
    ltrest = al*np.power(ilrest,2) - 2*np.asarray(al)*np.asarray(ilrest)*ilmax + lmax + np.power(ilmax,2)*al
    lt = np.concatenate((lt17, ltrest))
    ratio = wt/lt

    return ratio, lt
```

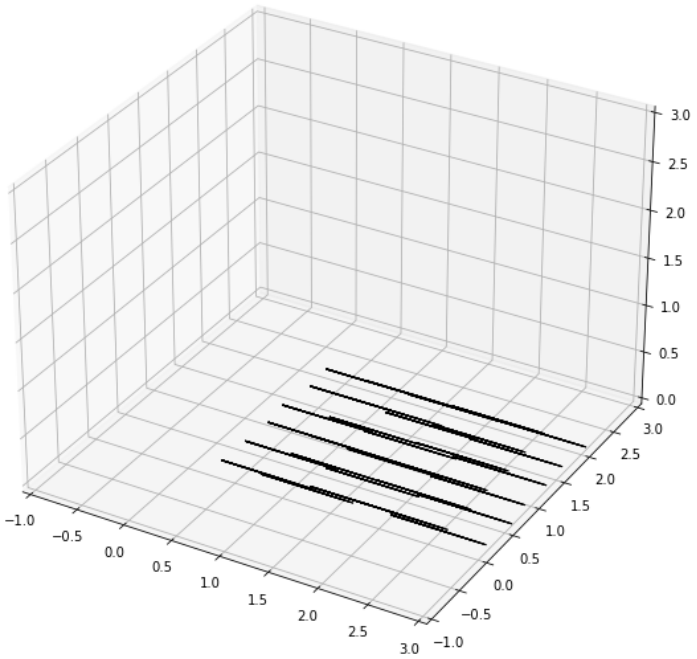
Here, we only used the length information -not associated with width or surface from previous explanation-. We did it to keep simple geometric relationships to perform future orientations, rotations, etc... That is why we have only 'stick-like' shapes, to only consider 1D -rather than 2/3D).

We also considered only length as width -and therefore surface- will be later defined in another way, to better represent the leaf lamina shape -which not follows the España et al. model-.



Then me multiply these 'mean' plants and place them according to sowing pattern already described earlier.
Here we have well positioned plants and *N* stacked leaves, each one described as a 1D-axis stick of length previously defined.

```
    for i in N :
      r_stemm.append(0.02 + (r_stem_N_max - 0.02)/ (N_max-1)* (i-1))
ABC_stem = []
for i in N :
    ABC_stem.append( [[r_stemm[i-1] ,0 ,0, 0 ,r_stemm[i-1] ,0, 0, 0, H_max] , [0, r_stemm[i-1], 0 ,-r_stemm[i-1], 0 ,0, 0 ,0, H_max
```



Leaf Orientation

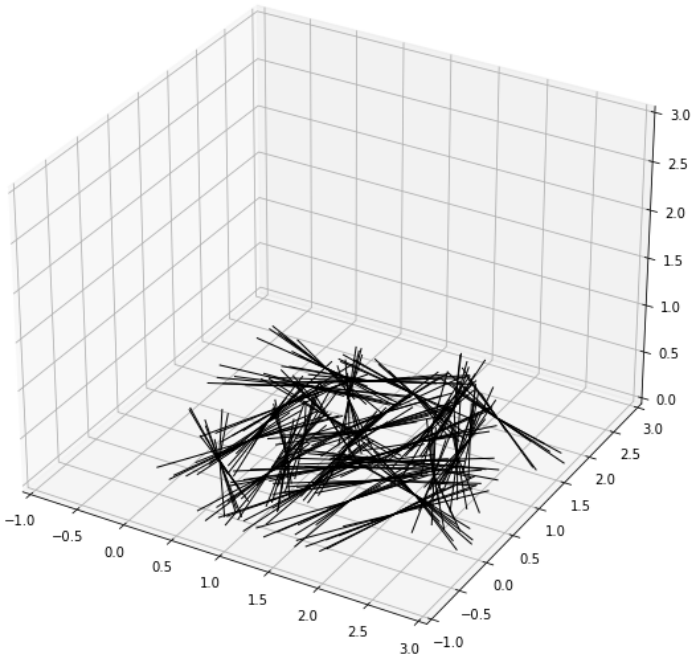
We will now give given orientations to each of these 1D-axis sticks.
To do so, we will first give each plant a main direction, according to a probability density function. Then, for each leaf, Gaussian dispersion to this main direction will be then added as noise.

As mentioned, for each *p* (plant) in the sowing pattern described earlier, we input a main direction according to a probability from a density continuous function (*dist.rvs* of *scipy package*). This given distribution and its *args* were, in our case, determined in Mario Serouart et al. *Analyzing Changes in Maize Leaves Orientation due to GxExM Using an Automatic Method from RGB Images. Plant Phenomics. 5;2023:0046. DOI:10.34133/plantphenomics.0046.* If you do not have a specific or precise idea of your distribution, you can then easily adapt this part to follow a Von Mises behavior. It will closely approach what we have done.

Once the direction value obtained, we input some dispersion around the latter thanks to **leaf_azimuth**. Note that, as our directions where fitted within a range of [0;90°], we added a random function of probability *0.5* to mirror the final azimuth angle ([0;90°] + Phyllotaxy [0;180°] will give back [0;360°] mock-ups).

Then we use **rot_3D_triangle** to reach the stick to the given angle (for each leaf, of each plant). It simply use trigonometric relationships to rotate an element -our stick- following a given angle in a 3D space -here to keep it simple, remember, we only rotate through x,y (soil plane)-. A robust literature : https://en.wikipedia.org/wiki/Rotation_matrix.

```
for p in range(len(xyz)) :

    teta_leaf = SS*delta_teta_leaf + teta_biggest + (np.random.randn(N_max,1)*rand_teta - rand_teta/2)

    ABC_leafind = copy.deepcopy(ABC_leaf)

    if p == 0 :
        teta_mat = teta_leaf[0]
    else :
        teta_mat = np.column_stack((teta_mat,teta_leaf[0]))


    dist = getattr(scipy.stats, dist_func)
    try :
        mOR = dist.rvs(loc=loc, scale=scale, *arg, size = 1)
    except :
        arg = [arg]
        mOR = dist.rvs(loc=loc, scale=scale, *arg, size = 1)

    mOR = mOR[0]

    while mOR < 0 or mOR > 90 :

        mOR = dist.rvs(loc=loc, scale=scale, *arg, size = 1)
        mOR = mOR[0]

    az_max = (leaf_azimuth(size = N_max, phyllotactic_angle = phyllotactic_angle, phyllotactic_deviation = phyllotactic_deviation, plant


    N = [i for i in np.arange(1, N_max+1, 1)]
    for n_leaf in N  :
        Post_Azi.append(math.degrees(az_max[n_leaf-1]))
        ABC_leafind[n_leaf-1,:] = rot_3D_triangle(ABC_leafind[n_leaf-1,:], [0, 0, 0])
        ABC_leafind[n_leaf-1,:] = rot_3D_triangle(ABC_leafind[n_leaf-1,:], [0, 0, math.degrees(az_max[n_leaf-1])])
        ABC_leafind[n_leaf-1,:] = ABC_leafind[n_leaf-1,:] + [0, 0, H[n_leaf-1], 0, 0, H[n_leaf-1], 0, 0, H[n_leaf-1]]
```

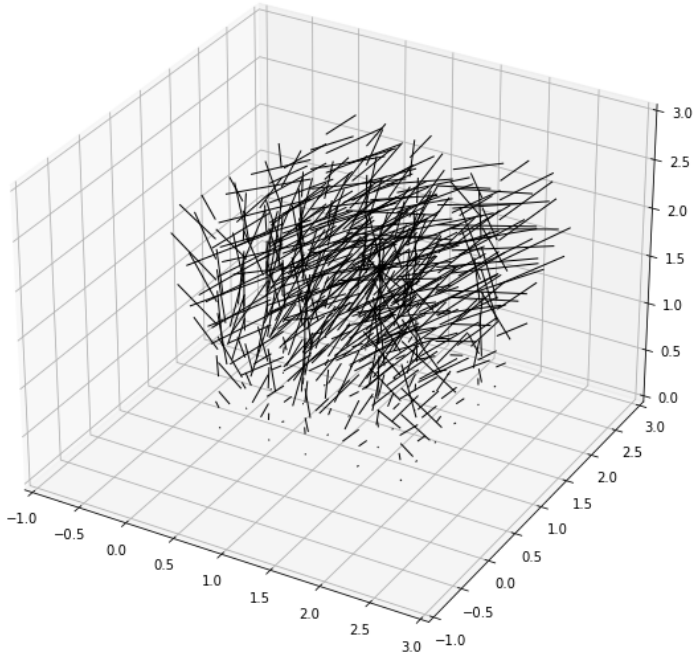


Height insertion

The insertion height (*h(i)*) of each leaf through the stem is modelled, height zero being the soil surface. It obviously depends on leaf order *i*. Our observations show that a power model provides a good approximation of leaf height and that the height of the first leaf, *h(1)*, is not varying strongly. We thus set *h(1)*, the first leaf in bottom of canopy, to the average observed value: *h(1)=0.015m*. The model of leaf insertion height can thus be written as:

```
H = (H_max - 0.015) / np.power(N_max, a-1) * np.power(N, a-1) + 0.015
```

Adding a z-axis value (height) to previous sticks will obviously not change their positions.



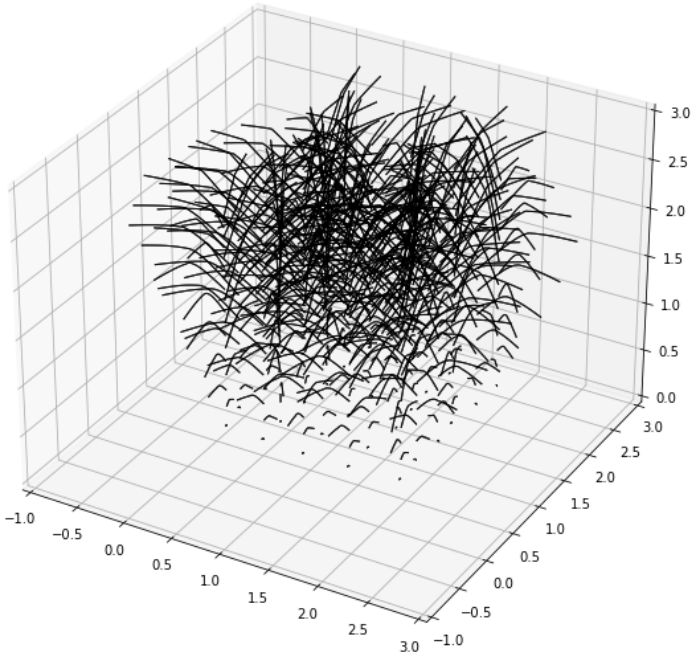
Inclination

From here, there is no other way, we have to work in a 3D space. However nothing prevents us from isolating elements in 2D space, just like x,y plane for azimuth directions. That is what we are going to do here. We consider the z-axis as the y-axis, so the leaf curvature linked to the leaf angle forms a parabola on x,z or y,z (again depending on rotations).

leaf_shape_rank will determine each 4 parameters that define shape of leaf curvature. Briefly, it generates x and y coordinates for a leaf profile shape of a single plant -including leaves from base to top-. This function sets the *l*, inflexion curve value, based on parameters related to the leaf position in the plant and its characteristics. It is calibrated on first leaf and interpolated linearly from *incli_base* first leaf to *incli_top* last leaf range.

Note that *vec_d* and *vec_g* denotes side of the future lamina -*g* and *d* for each side of midrib leaf-.

```
for n_leaf in range(len(ABC_couvert))  :

    Stacked_facets_corr = []
    Leaf_OI = np.mod(n_leaf, N_max)
    x_max, y_max = leaf_shape_rank(rank = Leaf_OI, incli_top = incli_top, incli_base = incli_base, l = l, delta = delta, nb_segment = in

    x_vec_g = (x_max/100 * math.cos(np.radians(Post_Azi[n_leaf])))+ ABC_couvert[n_leaf][0]
    y_vec_g = (x_max/100 * math.sin(np.radians(Post_Azi[n_leaf]))) + ABC_couvert[n_leaf][1]

    x_vec_d = (x_max/100 * math.cos(np.radians(Post_Azi[n_leaf]))) + ABC_couvert[n_leaf][6]
    y_vec_d = (x_max/100 * math.sin(np.radians(Post_Azi[n_leaf]))) + ABC_couvert[n_leaf][7]

    z_vec_g = ABC_couvert[n_leaf][2] + y_max/100
    z_vec_g_corr = ABC_couvert[n_leaf][2] + y_max/100
```



Surface fitted

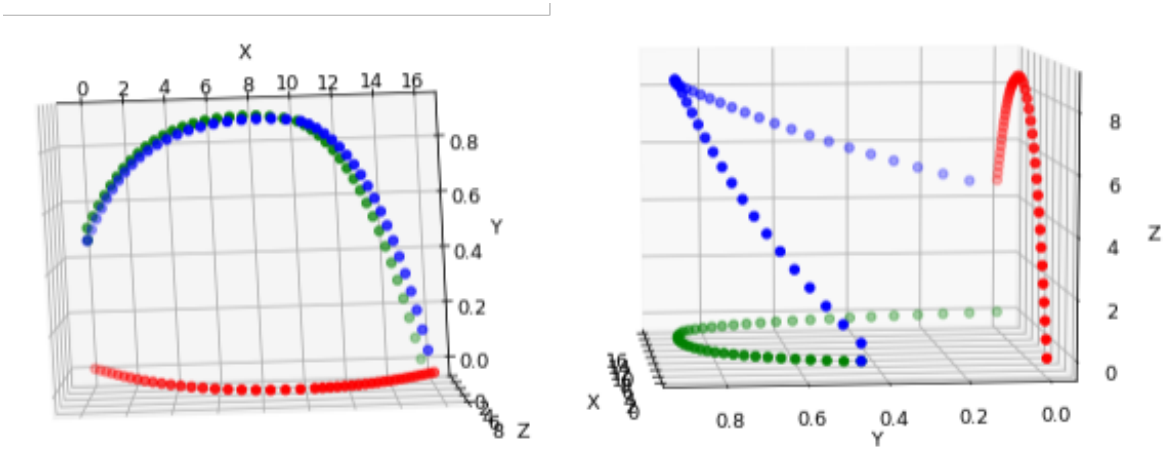
From this point, adding leaves width derived from the initial 4D Maize model would give us unrealistic isosceles triangles leaves, where the leaf base is equal to max width. See: *Raúl López-Lozano, Frédéric Baret, Michaël Chelle, Nadia Rochdi, Marisa España, Sensitivity of gap fraction to maize architectural characteristics based on 4D model simulations, Agricultural and Forest Meteorology, Volume 143, Issues 3–4, 2007, Pages 217-229, ISSN 0168-1923, https://doi.org/10.1016/j.agrformet.2006.12.005.*

We then need to transform modelled leaf surface into a realistic leaf shape.

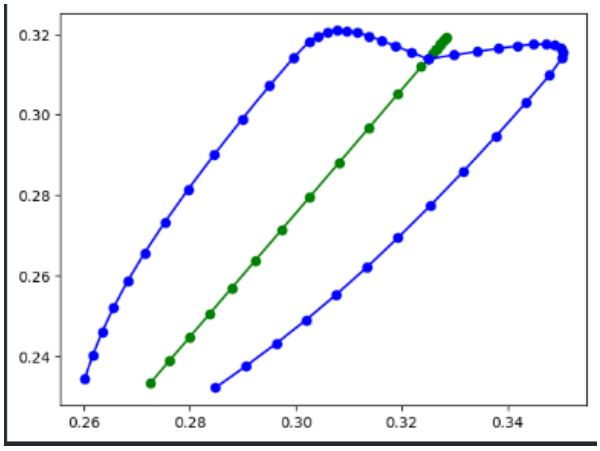
Thanks to *leaf_morpho_rel* we are able to do so. This function is derived from the excellent course available here : *https://github.com/openalea-training/hmba312_training*. Again, we fixed *x* as the length of leaf and *y*, the *y-value* it must takes to be realistic, while keeping the previous estimated value of *surface* from *España et al.* model defined earlier in the document. The shape is normalized to be adapted to each of leaf rank (*s* and *r* are multiplied to length and width, respectively).

Tips: Care must be taken at this stage, as area is a 2D concept. Simple rotation generates errors in the shape.

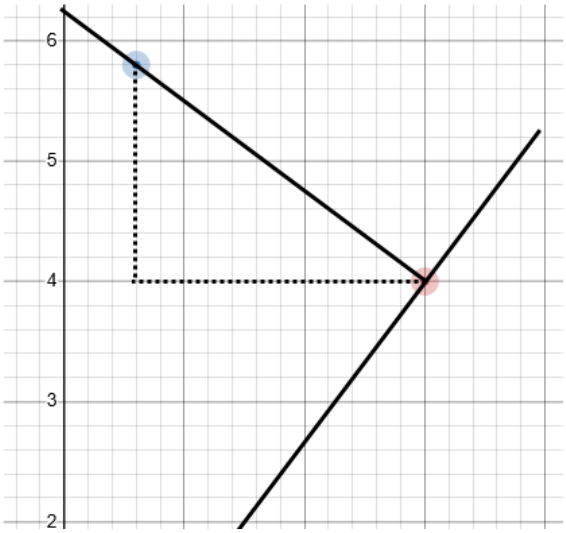
In the following figure, the red line corresponds to the previous **Inclination** fitted leaf profile, for a given leaf as example. The green line corresponds to the predicted leaf width at each length point -according to the midrib-. Then, if we simply add z-values of red points to the green line, we get the blue line. In this specific case, area under the green curve (on x,y plane) is different from the area under the blue curve (always on x,y plane).



Additionally, when rotated, the shape is distorted.



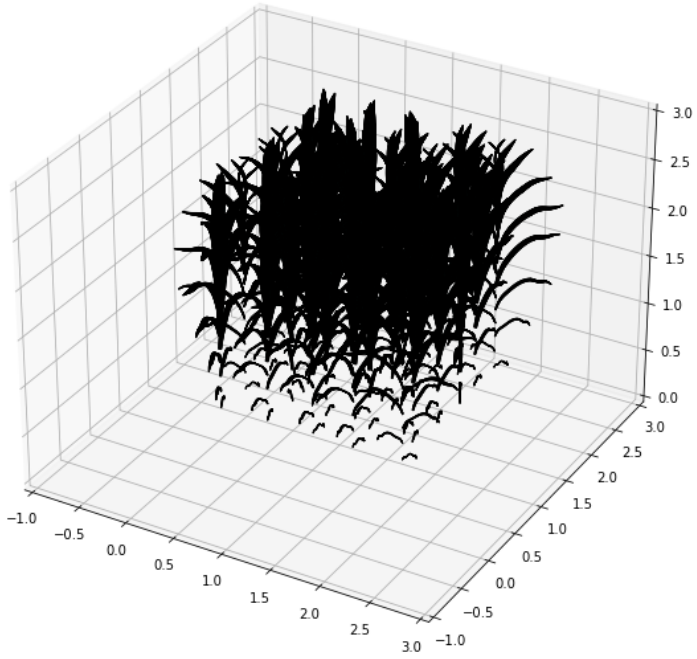
So, we must use *np.hypot* to compute the hypotenuse -our estimated width for a given length- of a given right-angled triangle's sides, in our case between 2 points as:



Note, here presented only half leaf lamina shape, *S_max* wad divided by 2 and leaf shape mirror-inverted, i.e. only half lamina is modelled then replicated backwards.

```
leaf_area = area_shape[Leaf_OI] / 2
w0=0.5+0.01*Leaf_OI
lm=0.5+(-.02)*Leaf_OI
wl_min=0.08+Leaf_OI*0.001

s,r=leaf_morpho_rel(nb_segment=len(x_max)-1, w0=w0, lm=lm)

l_min= longueur[Leaf_OI]
w_min=l_min*wl_min
s_min,r_min = s*l_min, r*w_min

integral = np.trapz(r_min,s_min)
w_min = w_min * (leaf_area / integral)
s_min,r_min = s*l_min, (r*w_min)


px_p = np.empty(len(x_vec_g))
py_p = np.empty(len(y_vec_g))
px_n = np.empty(len(x_vec_g))
py_n = np.empty(len(y_vec_g))
for idx in range(len(x_vec_g)-1):
    x0, y0, xa, ya = x_vec_g[idx], y_vec_g[idx], x_vec_g[idx+1], y_vec_g[idx+1]
    dx_p, dy_p = xa-x0, ya-y0
    norm_p = np.hypot(dx_p, dy_p) * 1/r_min[idx]
    px_p[idx] = x0-dy_p/norm_p
    py_p[idx] = y0+dx_p/norm_p

    dx_n, dy_n = xa-x0, ya-y0
    norm_n = np.hypot(dx_n, dy_n) * 1/-r_min[idx]
    px_n[idx] = x0-dy_n/norm_n
    py_n[idx] = y0+dx_n/norm_n

x_vec_g_corr = np.insert(px_p, 0, px_p[0])
y_vec_g_corr = np.insert(py_p, 0, py_p[0])
x_vec_d_corr = np.insert(px_n, 0, px_n[0])
y_vec_d_corr = np.insert(py_n, 0, py_n[0])
```

```
pp = np.arange(len(x_max)-1)
Stacked_facets_corr.append([x_vec_g_corr[pp], x_vec_g_corr[pp+1], x_vec_d_corr[pp], y_vec_g_corr[pp], y_vec_g_corr[pp+1], y_vec_d_co
Stacked_facets_corr.append([x_vec_g_corr[pp+1], x_vec_d_corr[pp+1], x_vec_d_corr[pp], y_vec_g_corr[pp+1], y_vec_d_corr[pp+1], y_vec_
```



Facets creation

As we are interested in facets light characterisation -facets are primitive shapes easier to handle-, we create triangles and linking up all points together every 3 vertices, taking into account each side of lamina midrib/nerve.

```
pp = np.arange(len(x_max)-1)
Stacked_facets.append([x_vec_g[pp], x_vec_g[pp+1], x_vec_d[pp], y_vec_g[pp], y_vec_g[pp+1], y_vec_d[pp], z_vec_g[pp], z_vec_g[pp+1],
Stacked_facets.append([x_vec_g[pp+1], x_vec_d[pp+1], x_vec_d[pp], y_vec_g[pp+1], y_vec_d[pp+1], y_vec_d[pp], z_vec_g[pp+1], z_vec_g|
```

Stems

Stems are added at the very end, as it was easier with mesh format which is explained in the next section. Each stem is defined as 4 long triangles (as a cone).

```
for tt in DEF_couvert :
    Stacked_facets.append( [tt[0], tt[3], tt[6], tt[1], tt[4], tt[7], tt[2], tt[5], tt[8]] )
```

MESH.txt to MESH.can format facets cloud

Following part is entirely according to *CanestraDoc.pdf* documentation -available in the Github repo- and obviously the excellent work on which relies Caribu raycasting model: *Chelle, Michael & Andrieu, Bruno. (1998). The nested radiosity model for the distribution of light within plant canopies. Ecological Modelling. 111. 75-91. 10.1016/S0304-3800(98)00100-8.*

The required *.can* file contains the geometric descriptions of a vegetation canopy. Exactly as our matrix **Stacked_facets** saved in *.txt* file, but which is in the wrong format and will need to be tranformed a bit later. The *.can* format allows to associate information of optics, facets, leaves, stems and plants through IDs allocation. Making easier interpretation in final output file that permit quantifying intercepted light by facets, leaves, stems and plants. This ASCII file contains then one line by facets.

This is what it looks like :

For a given facet : *p 2 100001001001 9 3 0.35 0.06 0.35 0.35 0.06 0.35 0.35 0.06 0.35*

p : polygon
2 : The number of args that will be declared before number of polygon vertices
100001001001: Optical species (1) + Plant no. (00001) + Organ no. (001) + Facet no. (001)
9 : Number of elements in triplet (3) * xyz coordinates (3)
3 : Number of polygon vertices
0.35 0.06 ... 0.35 : Vertices coordinates

In our case *Optical species* goes from 1 (Leaf) to 2 (Stem) and finally 3 (Soil). The optical properties will be defined in the next section. The goal is now just to transform *MESH.txt* to *MESH.can*, the correct working format. This is followed by loops of matrix translations and IDs additions in the exact order in which *Stacked_facets* was created.

The *OOO.txt* file is built as :
x1 x2 x3 y1 y2 y3 z1 z2 z3
Which are the 3 vertices of each facet/triangle

The *OOOC.txt* file is built as :
x1 y1 z1
x2 y2 z2
x3 y3 z3
Now, each vertices is a new line, with the 3-axis coordinate of that point in space

The final *OOO_CAN.can* file is built as :
IDs .. x1 y1 z1 x2 y2 z2 x3 y3 z3
According to the previous part, with IDs allocation

```
with open('/home/capte-gpu-2/Downloads/OOO.txt', 'r') as f_input, open('/home/capte-gpu-2/Downloads/OOOC.txt', 'w') as f_output:
        for line in f_input:
            xyz = line.split()
            for triple in zip(xyz[0:3], xyz[3:6], xyz[6:9]):
                f_output.write(' '.join(triple) + '\n')

    data = read_data("/home/capte-gpu-2/Downloads/OOOC.txt")

    a = []
    composite_list = [data[x:x + 3] for x in range(0, len(data), 3)]                                                    # .can (Chelle) struct

    shutil.copy("/home/capte-gpu-2/Downloads/OOOC.txt", "/home/capte-gpu-2/Downloads/OOOC.can") #
    text_file = open("/home/capte-gpu-2/Downloads/OOO_CAN.can", "w")

    Leaf = 1
    Plante = 1
    Stem = 1
    Stem_Tri = 1

    tri_counts = np.concatenate(Leaf_Tri_Count)
    for i in range(len(composite_list)):
        tri_type = 1 if i*3 < diff_stem_leaves else 2
        if tri_type == 2:
            if Stem_Tri == 5:
                Stem_Tri = 1
                Stem += 1
            id_str = '2' + str(Stem).zfill(5) + '00000' + str(Stem_Tri)
            Stem_Tri += 1
        else:
            if i != 0:
                if tri_counts[i] == 1:
                    Leaf += 1
                    if Leaf == N_max + 1:
                        Leaf = 1
                        Plante += 1
            id_str = '1' + str(Plante).zfill(5) + str(Leaf).zfill(3) + str(int(tri_counts[i])).zfill(3)
        with open("/home/capte-gpu-2/Downloads/OOO_CAN.can", "a") as text_file:
            exestr = 'p' + ' ' + '2' + ' ' + id_str + ' ' + '9' + ' ' + '3'
            for j in range(3):
                exestr += ' ' + str(composite_list[i][j][0]) + ' ' + str(composite_list[i][j][1]) + ' ' + str(composite_list[i][j][2])
            text_file.write(exestr + '\n')

    # SOIL
    with open('/home/capte-gpu-2/Downloads/OOO_CAN.can', 'r') as file:
        lines = file.readlines()
        last_lines = lines[-2:]
        modified_lines = []
```

```
idx = 0
for line in last_lines:
    elements = line.strip().split()

    if idx == 0 :
            elements[2] = '300001000001'
    else :
            elements[2] = '300001000002'

    modified_line = ' '.join(elements) + '\n'
    modified_lines.append(modified_line)
    idx = idx + 1

lines[-2:] = modified_lines
with open('/home/capte-gpu-2/Downloads/OOO_CAN.can', 'w') as file:
    file.writelines(lines)
```

Caribu computation

To finally work, Caribu needs you to define *optical properties* of each *optical species* (Leaf, Stem, Soil, ...).

The *par.opt* file
According to Chelle et al., 1998

espece 1
e d 0.10 d 0.10 0.05 d 0.10 0.05
espece 2
e d 0.01 d 0.01 0.005 d 0.01 0.005
espece 3
e d 0.5 d 0.5 0 d 0.5 0

Which means :

e : New declared species (espèce in french)
d : diffuse
0.10 : Species reflectance (upper surface)
0.05 : Species transmittance (upper surface)
d : diffuse
0.10 : Species reflectance (lower surface)
0.05 : Species transmittance (lower surface)

Finally,
n X must be described according to the number of species you are implementing. Here 3, Leaf, Stem, Soil. # number of species
n 3

Note *nir.opt* exists too but, what differs is only the wavelengths values you compute in reflectance/transmittance ranges, according to the literature, in the NIR domain.

```
opts = ['/home/capte-gpu-2/anaconda3/envs/adel/lib/python3.8/site-packages/alinea.caribu-8.0.7-py3.8.egg/alinea/caribu/data/par.opt']
```

Direct light conditions

According the sun path you want, you will define the Day Of the Year (in *int* number from 1 to 365). The energy -PPFD or PAR- is to be defined. We chose to normalize it, our sun source will launch from 0 to 1, and then with real weather station we just consider *True PPFD*.

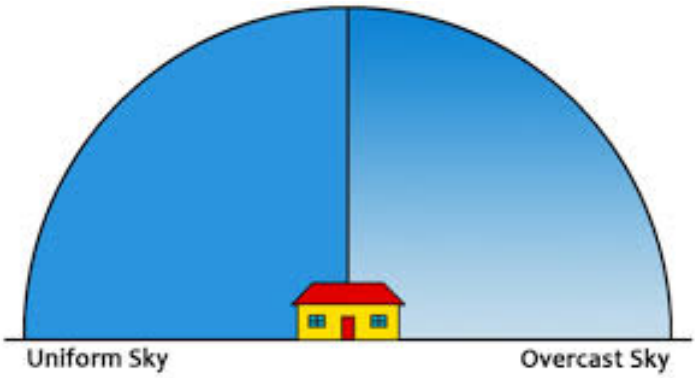
To have an accurate sun path, you will define Latitude from your experimental site. This snippet will then extract, for each hour, the sun position in the sky. Caribu will define sun as a flat panel with these given positions, from where it will 'cast rays' -See radiosity method from Caribu paper (SAIL volume based and Z-Buffer)-.

```
energy = 1
DOY = 175
latitude = latitude

for i in range(5,20) : #6, 20

    getsun = GetLightsSun.GetLightsSun(Gensun.Gensun()(energy, DOY, i, latitude)).split(' ')
    sun = tuple((float(getsun[0]), tuple((float(getsun[1]), float(getsun[2]), float(getsun[3])))))
```

Diffuse light conditions

The principle is the same. Except that to simulate a more opaque sky, in which the rays do not have privileged directions, we simulate a sky with "several suns". According to its position in the sky, a given sun will not have the same energy as another one. Indeed, it depends on their position in the half-sphere they occupy -as shown in figure below-.

Two systems exist: soc/uoc. We choose s-overcast, hence the position-dependent power.




```
sky_string = GetLight.GetLight(GenSky.GenSky()(1, overcast, 4, 5))      # (Energy, soc/uoc, number of azimuths, number of zeniths)

   sky = []
   for string in sky_string.split('\n'):
       if len(string) != 0:
           string_split = string.split(' ')
           t = tuple((float(string_split[0]), tuple((float(string_split[1]), float(string_split[2]), float(string_split[3])))))
           sky.append(t)
```

Computation

The final stage.

In running *cs* object, you may define *infinite* arg as *True* to replicate the scene and avoid border effects -you need a well-defined Bounding Box (BBox)-.

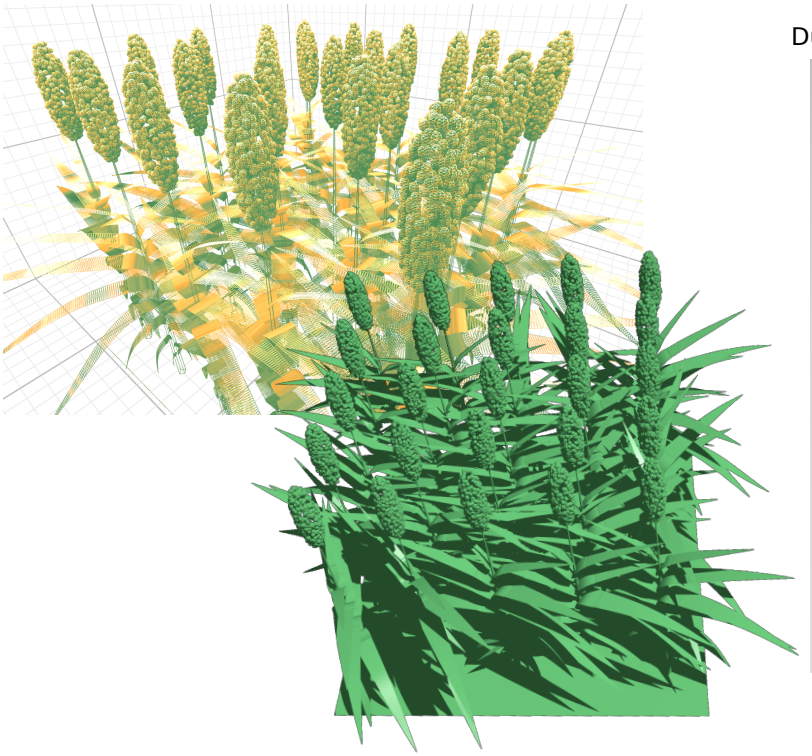
Then, you can choose *direct* arg as *True* to avoid computing direct illumination -first order = no transmitance, no reflectance, without rediffusion-. It will not take into account your *.opt* file with optical properties. However, you can define it as *False* to compute exact illumination (all orders, with rediffusions). It will takes more computational resources/time.

```
cs = CaribuScene(scene=can, light=sky or sun, opt=opts, scene_unit='m', pattern=(BBox[0], BBox[1], BBox[2], BBox[3]))
raw, agg = cs.run(infinite = True, direct = True)
```

Other features

Are provided in the Github repo, some other extended works using CORNIBU model.
It is some kind of dummies examples, but can therefore be useful to some of readers. Do not hesitate to use them if needed to directly translate this work to you own experiment.

Dummy Sorghum canopy derived from CORNIBU



Dummy Agrivoltaics system derived from CORNIBU