Angle generation

Author: Pedro O S

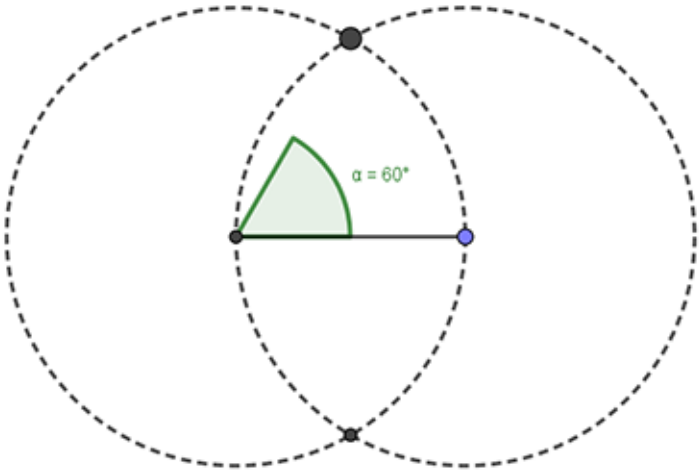
Note : this is a work in progress.

The use of the compass means drawing circles around points and intersecting.

The first construction (A) is

- Drawing a circle around a point on a line
- Drawing a second circle on the intersection of the first circle and the line
- Intersecting the two circles

The angle between the line and the intersection will be of 60°, or $\frac{1}{6}$ of a circle.



A circle is the most elementary continuous form; an infinite curve.
Whereas a polygon has a finite number of sides, a line has no sides; a circle must have sides, otherwise it would be a line.
Then, a circle has “infinite” sides.

But then, a larger circle must have more sides than a smaller circle, or otherwise they would be the same size.
Then, we conclude there are different infinities at hand. That is, in the continuous realm.

The circle, then, contains the passage to the uncountable, and it’s only one of the three elementary geometric forms (line, triangle, circle) from which basic theorems in geometry are derived.

These three elementary forms already contain the main problematique. As such, they can be seen as an expression of it in a particular form or language, the language of Geometry, not superior, inferior, older or newer than other theories. In Euclid we were already in front of this thousands of years ago. In the meantime, the theory got larger, but the root problematique remains the same.

Yet in spite of this property of density the rational numbers are not sufficient to represent every point on the number axis. Even the Greek mathematicians recognized that when a given line segment of unit length is chosen there are intervals whose lengths cannot be represented by rational numbers; these are the so-called segments incommensurable with the unit. — Richard Courant, 1937

The passage from a polygon to a circle represents the passage from discrete to continuous.

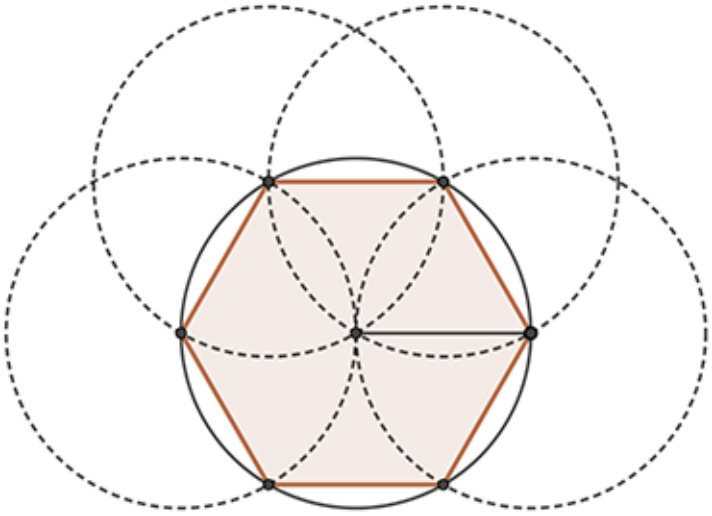
To emphasize that using a compass is working with circles, we'll draw full compass circles.

For example, since the angle is $\frac{1}{6}$ of a circle, we can trivially inscript a 6-sided regular polygon inside the circle (B).

For each intersection:

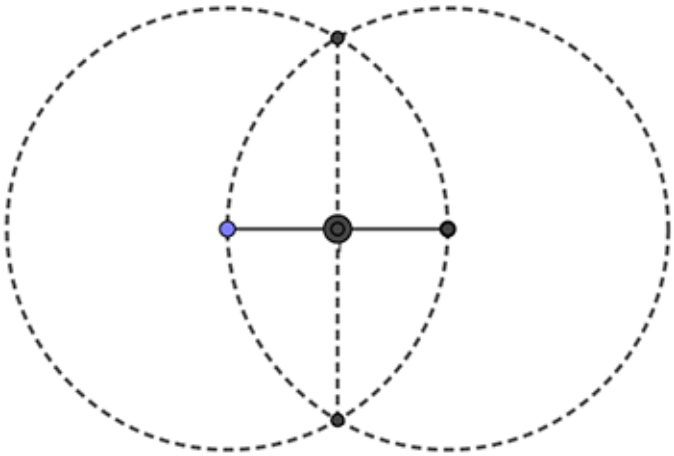
- Use the intersection as the center of a new circle
- Intersect the new circle again

Doing this four times is enough because each circle defines two intersection points. So with four circles the two extra points not in common between them are taken care of.



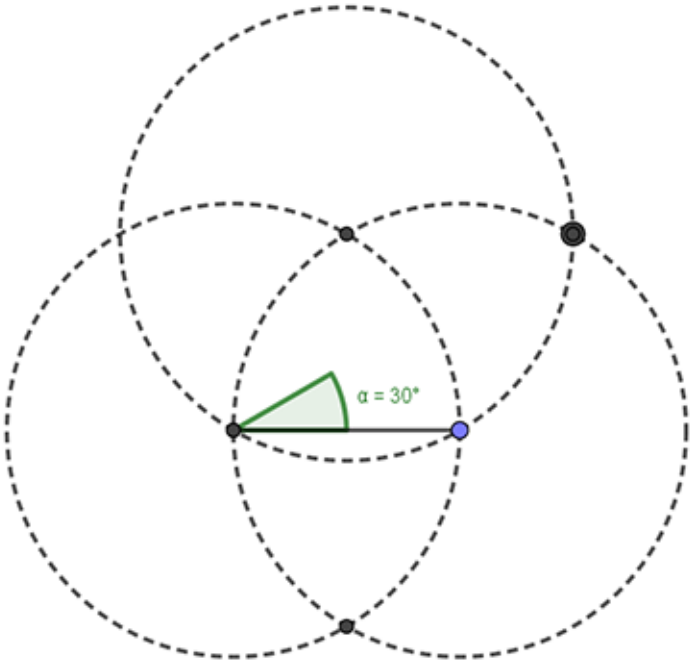
We have begun with a point and found a regular polygon with it as the center.
But, beginning with a segment, we can find its middle point (C).

- Draw the same two circles as the first construction
- Draw a segment between its two intersections
- Intersect the segment with the origin line



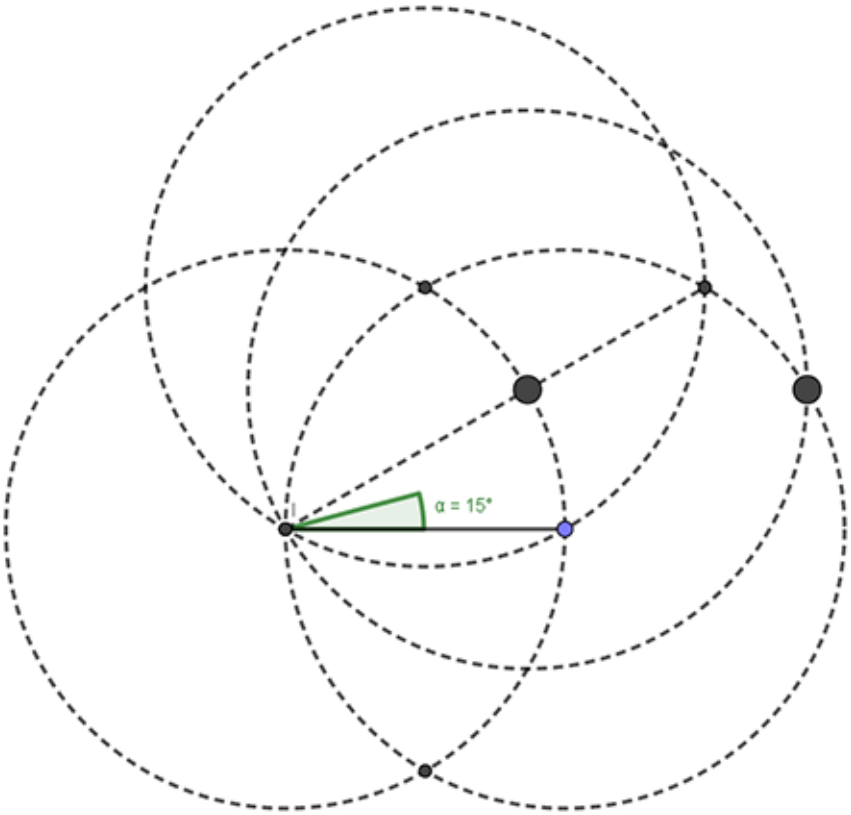
Two points in a circle forming a 60° angle can be used to find a 30° angle (D).

- Draw one further circle with center at the intersection
- Intersect the new circle with the previous one

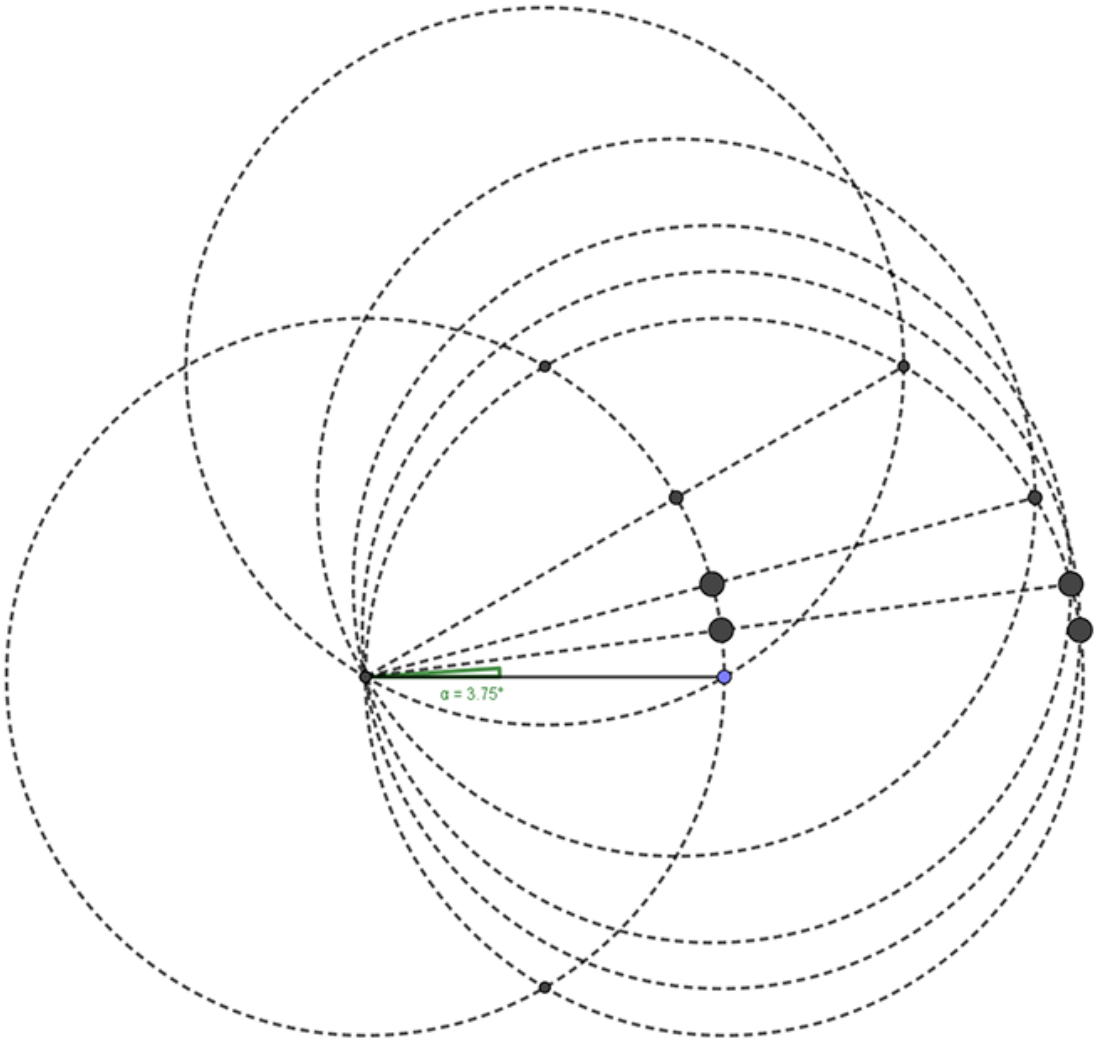


To get half of the 30° angle (E), intersecting between circles is not enough. It is necessary to intersect a circle and a segment:

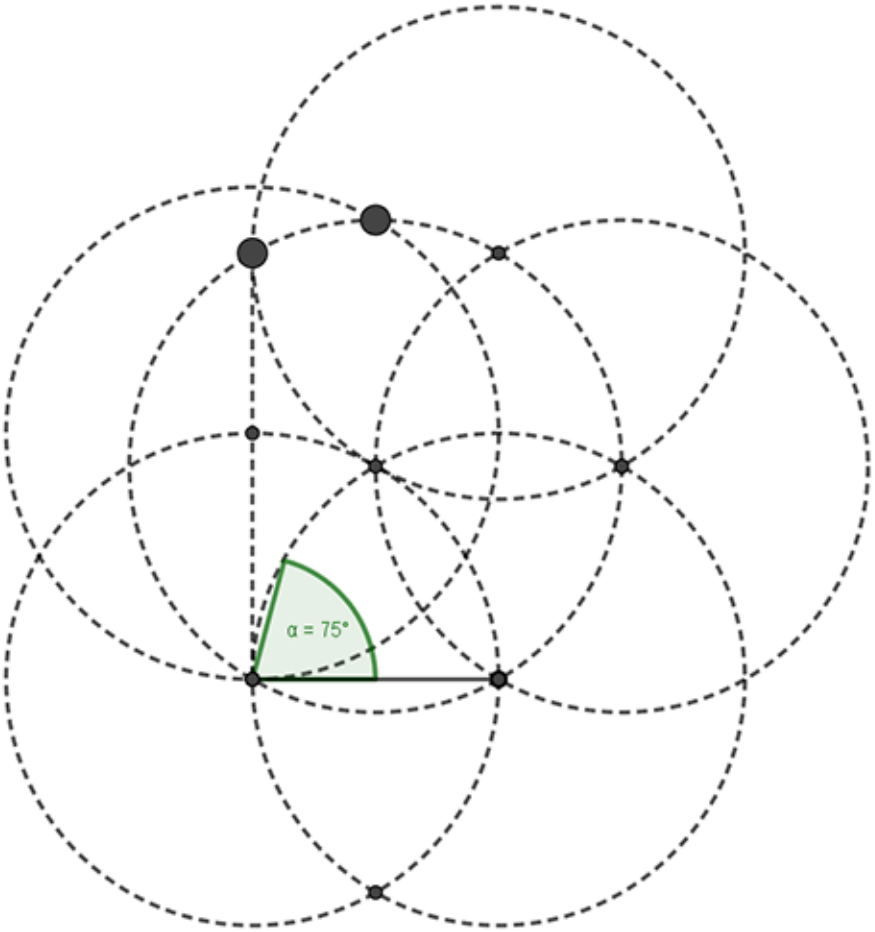
- Create a segment from the origin point to the 30° intersection point
- The intersection of the segment with the first circle defines a 30° point on the first circle
- Use it as a center for another circle of the same diameter
- The intersection of the new circle with the circle which defines the 30° angle is then at 15°



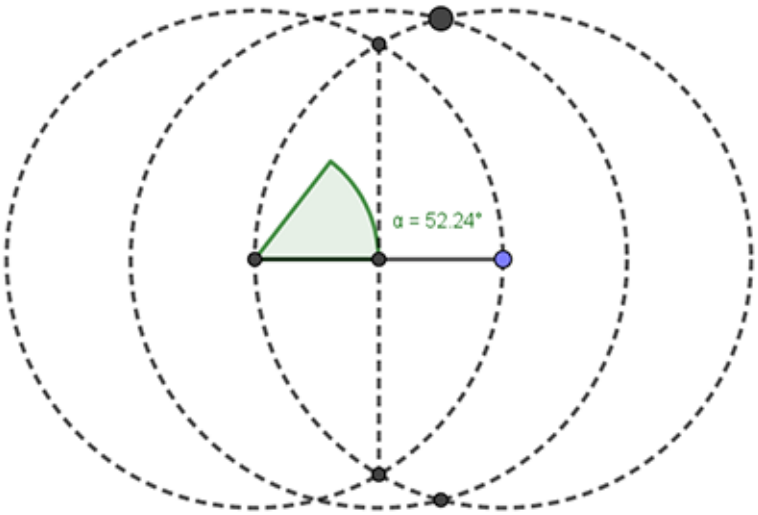
Let’s take it two steps further to generate a 3.75° angle (F).



This can be used to generate different angles by picking different “initial” circles. For example, in the next drawing we obtain the 75° angle (G) by using an intersection with a segment angled at 90°:



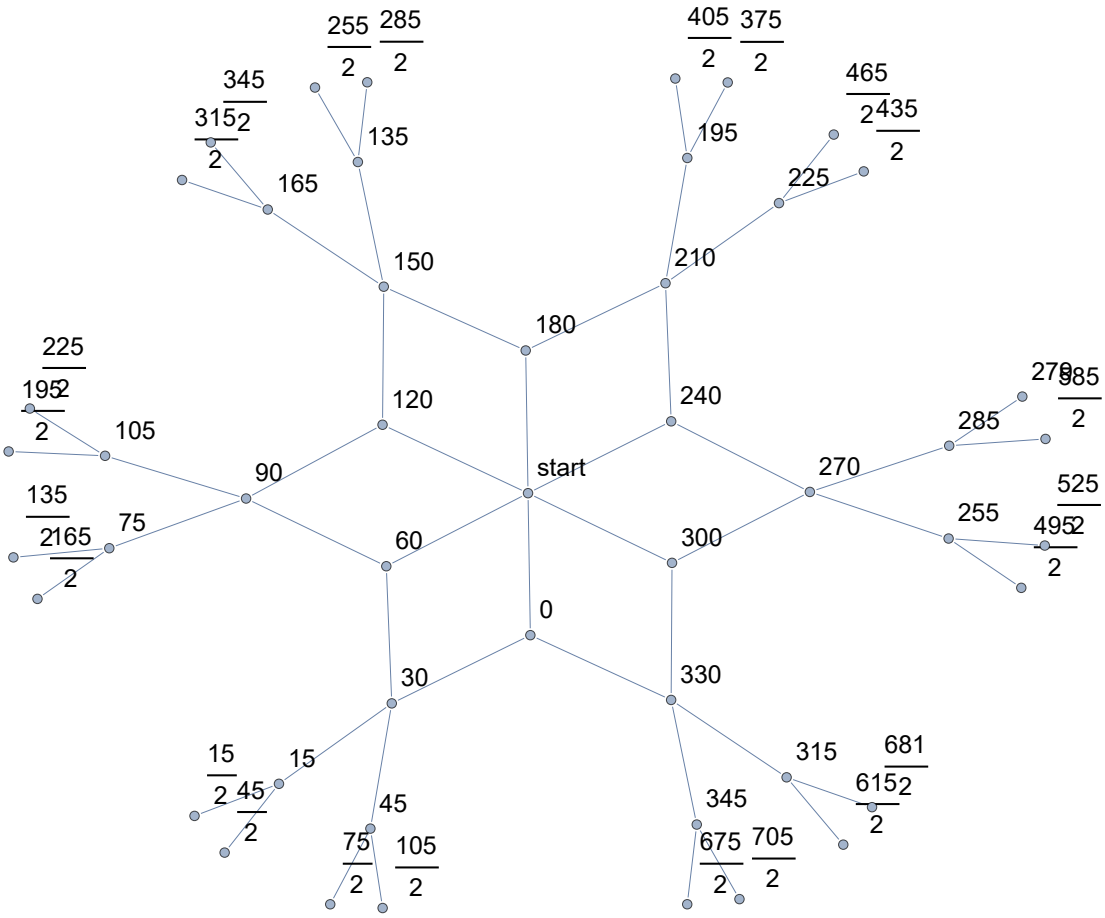
Curiously, the same process doesn't work for the 60° construction, to find a 30° angle (H):



Angle space

Let's examine the set of angles obtainable from subdividing, consecutively, integer multiples of 60°, up to 360 ° = 0 °, in half.

Out[ ]=



And so on.

There are at least two ways this can be computed up to how many levels we wish.

Level method

The first method is mathematically more natural, because it is only a recursive construction, and not a stateful algorithm.
We identify the pattern from the manual values calculated for the above graph and extract a general formula.

Let's denote by L the current level of the algorithm.

At $L = 0, x_0$

$= 60$

At $L = 1$, x_{1_1}, x_{1_2}

$$= x_{L-1_1} \pm \frac{x_0}{2^L}$$

$$= x_{0_1} \pm \frac{60}{2^1}$$

$$= 60 \pm \frac{60}{2}$$

$$= \{90, 30\}$$

At $L = 2$, $x_{2_1}, x_{2_2}, x_{2_3}, x_{2_4}$

$$= \left\{x_{L-1_1} \pm \frac{x_0}{2^L}, x_{L-1_2} \pm \frac{x_0}{2^L}\right\}$$

$$= \left\{x_{1_1} \pm \frac{60}{2^2}, x_{1_2} \pm \frac{60}{2^2}\right\}$$

$$= \left\{90 \pm \frac{60}{4}, 30 \pm \frac{60}{4}\right\}$$

$$= \{\{105, 75\}, \{45, 15\}\}$$

At $L = 3$,

$x_{3_1}, x_{3_2}, x_{3_3}, x_{3_4}, x_{3_5}, x_{3_6}, x_{3_7}, x_{3_8}$

$$= \left\{\left\{x_{L-1_1} \pm \frac{x_0}{2^L}, x_{L-1_2} \pm \frac{x_0}{2^L}\right\}, \left\{x_{L-1_3} \pm \frac{x_0}{2^L}, x_{L-1_4} \pm \frac{x_0}{2^L}\right\}\right\}$$

$$= \left\{\left\{x_{2_1} \pm \frac{60}{2^3}, x_{2_2} \pm \frac{60}{2^3}\right\}, \left\{x_{2_3} \pm \frac{60}{2^3}, x_{2_4} \pm \frac{60}{2^3}\right\}\right\}$$

$$= \left\{\left\{105 \pm \frac{60}{8}, 75 \pm \frac{60}{8}\right\}, \left\{45 \pm \frac{60}{8}, 15 \pm \frac{60}{8}\right\}\right\}$$

$$= \left\{\left\{\left\{105 + \frac{15}{2}, 105 - \frac{15}{2}\right\}, \left\{75 + \frac{15}{2}, 75 - \frac{15}{2}\right\}\right\},\right.$$
$$\left.\left\{\left\{45 + \frac{15}{2}, 45 - \frac{15}{2}\right\}, \left\{15 + \frac{15}{2}, 15 - \frac{15}{2}\right\}\right\}\right\}$$

We note:

- The first level is the initial condition for the recursion X_0
- Each level L has a result of (flattened) size 2^L
- Each level references the previous level's results: $X_{L-1_1} \cdots X_{L-1_{2^{L-1}}}$
  - This is the state of the computation that is kept, and since this state is not simply the number of the current iteration (it is the results of the last iteration), the algorithm can not be modeled as a simple sequence. Since the state is not kept, though, except for passing from one step to the other (it is not external state), the state can be modeled as an argument to a recursive function
- Each level references the initial value X_0

Function implementation

Let's implement each property of the function in turn.

Let's obtain the general correct form of the recursion.

In[•]:=
```
f1=.;f1=Function[{a,lvl,maxLevel},
    If[lvl≤maxLevel,
        Print[Labeled[a,"lvl="<>ToString@lvl,Left]];
        With[{v=f1[{a,a},lvl+1,maxLevel]},
            v
        ],]
];
```

In[•]:=
```
Block[{a}, f1[a, 0, 3]]
```

```
lvl=0 a
lvl=1 {a, a}
lvl=2 {{a, a}, {a, a}}
lvl=3 {{{a, a}, {a, a}}, {{a, a}, {a, a}}}
```

The function we implement does the following steps.

1. Project each element a of the current result into a $\{a+b, a-b\}$ pair

In[•]:=
```
Block[{a, b}, ({# + 1, # - 1}) & /@ {a, b}]
```

Out[•]=
```
{{1 + a, -1 + a}, {1 + b, -1 + b}}
```

In[•]:=
```
Block[{a, b}, ({# + 1, # - 1}) & /@ {{a, b}, {a, b}}]
```

Out[•]=
```
{{{1 + a, 1 + b}, {-1 + a, -1 + b}}, {{1 + a, 1 + b}, {-1 + a, -1 + b}}}
```

```
In[•]:= First@{{1, 1}}
Out[•]= {1, 1}

In[•]:= FlattenAt[{{1, 1}}, 1]
Out[•]= {1, 1}
```

```
In[•]:= f1b=.;f1b=Function[{res,lvl,maxLevel},
    If[lvl≤maxLevel,
        Block[{flt=If[ListQ@res,Flatten@res,res],
            (*Project each element of the current result into a ± pair*)
            newarg=Map[({#,#})&,If[ListQ@res,res,{res}]]},
            If[lvl==0,newarg=FlattenAt[newarg,1]];
            Print[Labeled[Column@{
                Labeled[res,"Result:",Left],
                Labeled[flt,"Flattened:",Left],
                Labeled[newarg,"New argument:",Left]
            },"lvl="<>ToString@lvl,Left]];
            With[{v=f1b[newarg,lvl+1,maxLevel]},
                (*Print@Labeled[v,"New result:",Left];*)
                v
            ]
        ]
    ,]
];
```

There is some treatment over the first input to the recursion, which, unlike the following inputs, is not a list. This is a natural reflection of the starting terminus of the recursive computation.

```
In[•]:= Block[{a}, f1b[a, 0, 3]]

      Result: a
lvl=0 Flattened: a
      New argument: {a, a}

      Result: {a, a}
lvl=1 Flattened: {a, a}
      New argument: {{a, a}, {a, a}}

      Result: {{a, a}, {a, a}}
lvl=2 Flattened: {a, a, a, a}
      New argument: {{{a, a}, {a, a}}, {{a, a}, {a, a}}}

      Result: {{{a, a}, {a, a}}, {{a, a}, {a, a}}}
lvl=3 Flattened: {a, a, a, a, a, a, a, a}
      New argument: {{{{a, a}, {a, a}}, {{a, a}, {a, a}}}, {{{a, a}, {a, a}}, {{a, a}, {a, a}}}}
```

A second method of computing the angle space is by generating a tree.

Tree method

By tree, we mean it is not a simple recursion with a function to transform the argument, but a recursion with a state variable which is not the function argument passed on each call.
In other words, an algorithm with external state is an imperative algorithm, and the function, an impure function. In this way, this algorithm is more “complicated” than the level method, but it is naturally expressed in the realm of a programming language.
Outside of the state aspect, the generation of the tree itself is very simple.

The basic operation can be defined as $x_1, x_2 = x_L \pm \frac{x_{L-1}}{2}$, where L denotes the level.

In[ ]:=
```
tree=.;tree=Function[{x,fun,level,maxLevel},
    Print@level;
    If[level<maxLevel,
        With[{v=tree[fun[x],fun,level+1,maxLevel]},
            Print@v;
            v
        ],
    ]
];
```

In[ ]:=
```
tree[1, # + 1 &, 1, 3]
```

```
1
2
3
Null
Null
```

In[ ]:=
```
Clear[f1];
f1=Function[{x},{x-x/2,x+x/2}];
```

In[•]:=
```
Style[Column@{
   f1[x],
   f1[f1[x]],
   f1[60],
   f1[f1[60]]
  }, Large]
```

Out[•]=

$$\left\{\frac{x}{2}, \frac{3x}{2}\right\}$$

$$\left\{\left\{\frac{x}{4}, \frac{3x}{4}\right\}, \left\{\frac{3x}{4}, \frac{9x}{4}\right\}\right\}$$

$$\{30, 90\}$$

$$\{\{15, 45\}, \{45, 135\}\}$$