How does the fractal generator work?

How to make the generator pattern

The generator pattern that will later be applied is stored as an array of vectors called `moves`. Each vector represents the movement one must make to get from the current point to the next point in the pattern.
So how do we get all those vectors?

First we need to take a look at the *generator_canvas*. On this canvas you see a bunch of lines forming a grid and two points on the same horizontal line. Those two points are two points in a certain coordinate-system with the left point being $(0,0)$ and the right point being $(\tilde{x}, 0)$, where $\tilde{x}$ is the number entered in the input-field below the *generator_canvas*. So all the lines simply mark steps of 1 unit in the x- and y-direction.
In the code $\tilde{x}$ has the name `reference_point_x`.

The two reference points are always vertically centred on the canvas and the left one is always one quarter of the whole width of the *generator_canvas* to the right (starting from the left side of the canvas) and the right point is three quarters to the right.
This is archived with a vector called `ORIGIN_OFFSET` which gives the number of pixels one has to move to get from the upper left corner of the canvas to the left reference point:

```
const ORIGIN_OFFSET = new Vector(GEN_WIDTH / 4, GEN_HEIGHT / 2);
```

Furthermore there is a variable that holds the number of pixels on the canvas that translate to one unit:

```
let GRID_SCALE = GEN_WIDTH / (2 * reference_point_x);
```

With those two values we can write the following functions converting between pixels and coordinates:

```
function coords_to_px(coord_vec, orig_offset, scale) {
    let ret = new Vector(coord_vec.x, coord_vec.y);
    ret.scale(scale) // scale up the units to get the needed pixels
    ret.y *= -1; // flip the y-axis
    ret.add_vec(orig_offset); // shift by the offset
    return ret;
}
function px_to_coords(px_vec, orig_offset, scale) {
    let shifted = diff_vec(px_vec, orig_offset);
    // we need to multiply the y-value by -1 because the y-axis in the canvas-
      system points downwards
    shifted.y *= -1;
    // by dividing each component by scale we get the coordinates in units
    shifted.scale(1/scale);
    return shifted;
}
```

Since we can easily convert between pixels and coordinates now, calculating the next move vector is as simple as waiting for the user to click on the canvas, retrieving the pixels they clicked on, converting them to coordinates and compute the vector that points from the currently last point in

the pattern to the new point. The vector we receive from this will be added to the `moves`-array. We can do all of this with the following function:

```
function add_move(pixels) {
    const coords = px_to_coords(pixels, ORIGIN_OFFSET, GRID_SCALE);
    const snapped = snap_to_grid(coords);
    const last_point = add_moves();
    const next_move = diff_vec(snapped, last_point);
    // since next_move will only have whole numbers its length should be at
      least 1
    // if thats not the case there won't be movement
    if (next_move.norm() < 0.1) {
        return;
    }
    moves.push(next_move);

    update_start_button();
}
```

The input `pixels` is simply a vector containing the pixel-coordinates of the point in the canvas the user clicked on.
`snap_to_grid` is a simple function that takes a vector with two non-whole-numbered coordinates and returns the closed vector with only whole-numbered coordinates. `add_moves` simply adds all the current move-vectors to get the current end point of the pattern. This code of the two functions is the following:

```
function snap_to_grid(coords) {
    return new Vector(Math.round(coords.x), Math.round(coords.y));
}
function add_moves() {
    let ret = new Vector(0, 0);
    moves.forEach(element => {
        ret.add_vec(element);
    });
    return ret;
}
```

`update_start_button` is a function that checks whether our current pattern reaches the right reference point. Only then the user should be allowed to start drawing the actual fractal[1].

That was the main logic behind getting the generator pattern. Everything else that can be found in *generator.js* is just for drawing the pattern which is just a matter of stepwise going through the moves and adding them together, then converting the coordinates to pixels and drawing a line and some points at those positions.

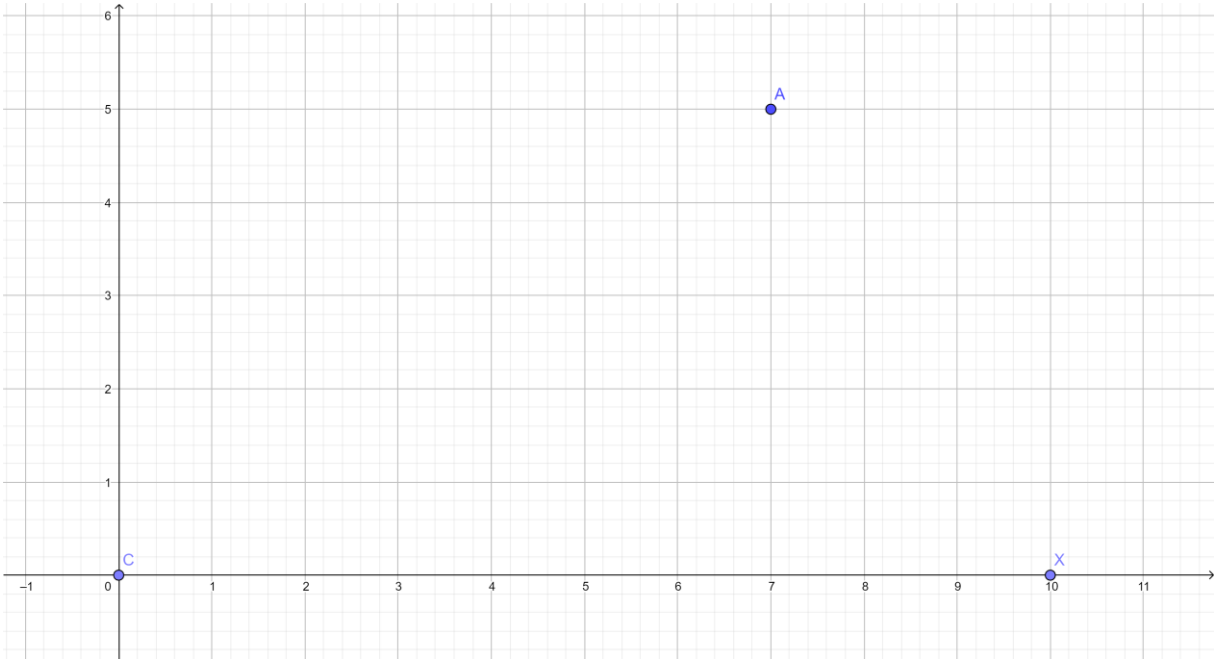
[1] I know that we don't draw actual fractals here since a real mathematical fractal would be the limit of the process if we were able to draw infinitely many iterations. To simplify things I will still refer to the figures that are drawn as *fractals*

How to draw the fractal

For drawing the fractal with a given pattern (i.e., the `moves`-array) the general procedure is the following: We start with two initial points. For each iteration we go through the list of all points and replace the line between two neighbouring points with the given pattern which will effectively add more points between them in the `points`-array. Because the number of points will grow exponentially with each iteration, we won't be able to do many iterations depending on the number of points we add with our generator.

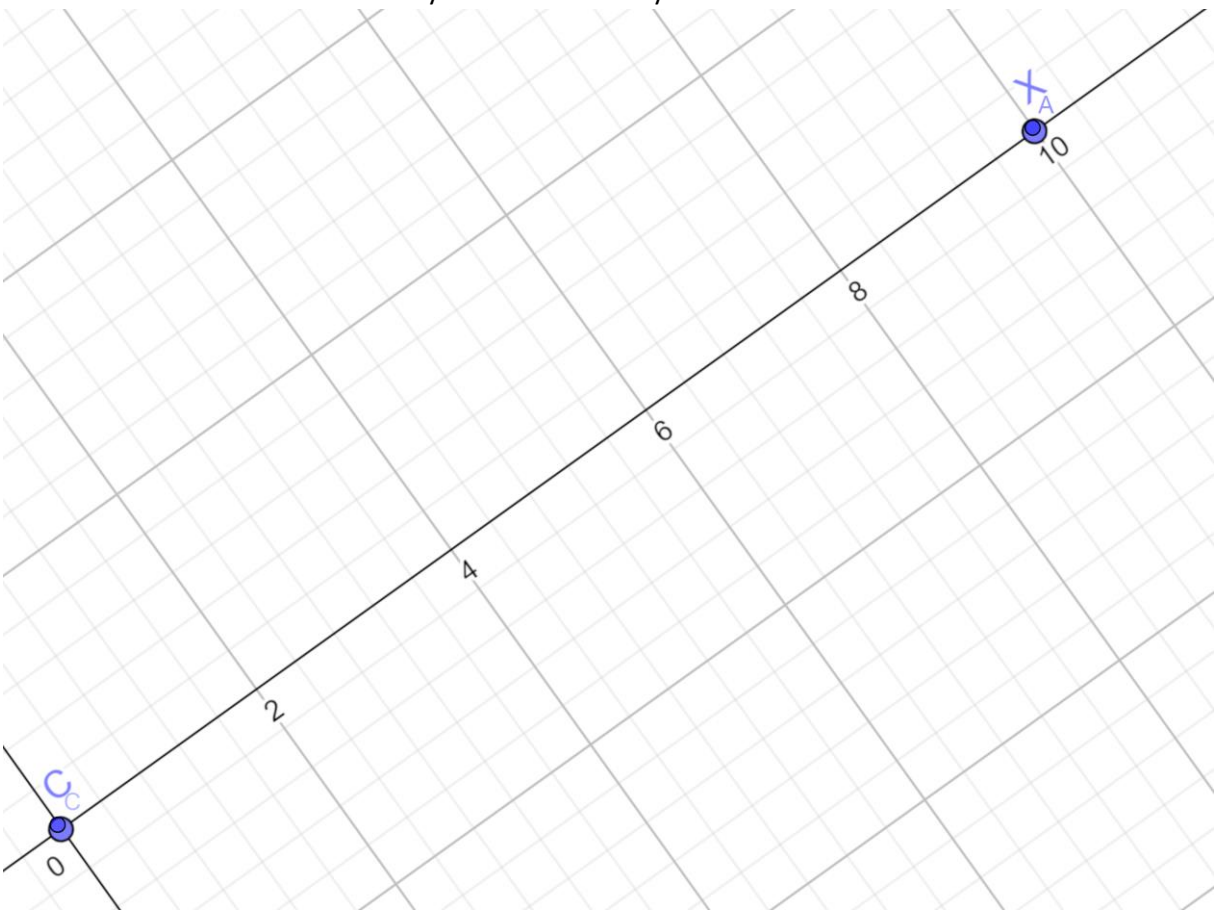
Now, given two points C and A on whose we want to apply the generator pattern, how do we proceed? The idea is to think of a new coordinate system that has C as its origin. We know need to find a linear transformation that scales and rotates this coordinate system in a way that the point $(\tilde{x}, 0)$ will land on A. When we now apply this transformation on each vector in the `moves`-Array, we can simply add those transformed *move*-vectors together to get the coordinates of the points we want to add.

So we start with the following:



X is just the point $(\tilde{x}, 0)$. So right know the pattern would work between C and X. That's why we

want to transform this coordinate system in such a way that X lands on A:



First we take care of the scaling (because it's a little simpler). We want that $\|X'\| = \|A\|$. So we simply have to divide by $\|X\|$ to normalize X and then multiply by $\|A\|$. So the scaling matrix S will be

$$S = \|A\| \cdot \frac{1}{\|X\|} \cdot \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

In the code we see this happening in the following lines:

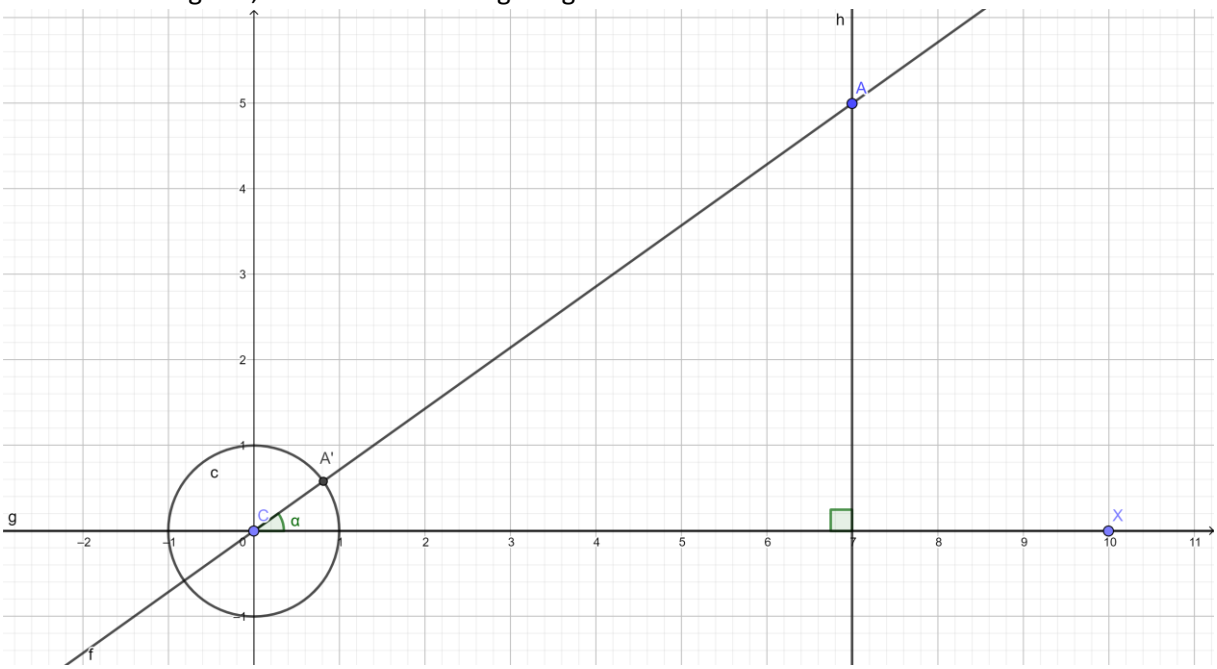
```
const A = diff_vec(b, a); // A is vector a->b
let S = new Matrix([1, 0, 0, 1]);
S.scale(A.norm()/working_reference_point);
```
[2]

Now to the rotation bit. The difficult part is just figuring out the angle α by which we will rotate because then we will simply insert that value in the rotation-matrix

$$R = \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix}$$

[2] Note that the variable is called `working_reference_point`. This is because the code for drawing the fractal makes copies of the variable `reference_point` and the `moves`-array to enable the user to draw the next pattern while a fractal is drawn. That's why the variables have the prefix `working_` in this part.

To see how we get α, look at the following image:



You can see that $A' = \frac{1}{\|A\|} \cdot A$. But also $A' = \begin{pmatrix}\cos(\alpha)\\ \sin(\alpha)\end{pmatrix}$. Therefore, we know that

$$\frac{1}{\|A\|} \cdot A.x = \cos(\alpha)$$

$$\Leftrightarrow \operatorname{acos}\left(\frac{1}{\|A\|} \cdot A.x\right) = \alpha$$

But we should be more precise since this is only true when A' is above the x-Axis. If it's below the x-Axis the actual angle should be $-\alpha$. Therefore, the whole code for getting α is:

```
let alpha = Math.acos(A.x / A.norm());
if (Math.asin(A.y / A.norm()) < 0) {
    alpha *= -1;
}
```

With those two snippets the combined function to get the final transformation is:

```
function get_trafo_matrix(a, b) {
    const A = diff_vec(b, a); // A is vector a->b
    let S = new Matrix([1, 0, 0, 1]);
    S.scale(A.norm()/working_reference_point);
    let alpha = Math.acos(A.x / A.norm());
    if (Math.asin(A.y / A.norm()) < 0) {
        alpha *= -1;
    }
    const R = new Matrix([
        Math.cos(alpha), -Math.sin(alpha),
        Math.sin(alpha), Math.cos(alpha)
    ]);
    return mat_mul(R, S);
}
```

We multiply R and S to get the combined transformation which first scales everything and then rotates.

So now that we know how to compute the transformation, applying the generator pattern is as simple as going through the current list of points, calculating the needed transformation for each pair of neighbouring points, then going through each move in `moves`, transforming this move and adding it to the current point. The result will be inserted between the current point and the second point of the neighbouring pair. In the code we do this by calling the function `apply_generator_on_pair` which will return an array of points that will take the place of the two points we're applying the generator to. We will then concatenate all those arrays of new points to one large array of points which will then be the new `points`-array.

The code looks like this:

```
function apply_generator() {
    let new_points = [];
    for (let i = 0; i < points.length - 1; i++) {
        const a = points[i];
        const b = points[i+1];
        const last_pair = (i == points.length - 2);
        const generated_points = apply_generator_on_pair(a, b, last_pair);
        new_points = new_points.concat(generated_points);
    }
    points = new_points.slice();
}
function apply_generator_on_pair(a, b, return_with_b = false) {
    let ret = [];
    let curr = new Vector(a.x, a.y);
    const F = get_trafo_matrix(a, b);
    working_moves.forEach(direction => {
        // we flip the direction because of the canvas coordinate system
        const flipped_direction = new Vector(direction.x, -direction.y);
        ret.push(new Vector(curr.x, curr.y));
        let movement = mat_vec_mul(F, flipped_direction);
        curr.add_vec(movement);
    });
    if (return_with_b) {
        ret.push(new Vector(b.x, b.y));
    }
    return ret;
}
```

Note that the vectors in `points` will store their position in pixel-coordinates. This is no issue because the transformation F will transform the pattern-vectors to work in those coordinates. The flag `return_with_b` is important because usually the return shouldn't include the point b since it will be in the list of new points for the next pair (because there it is the starting point). So including it at the end as well will have the result that b will occur twice in a row in `points`. But at the very end we want the last point to occur in the return-list. That's why we have this flag.

Since the vectors in `points` are always in pixel-coordinates we wouldn't have to do anything when drawing the points. The only thing that could happen is that the resulting fractal will be too big for the canvas. That's why you will find code for computing another scaling-matrix and an offset to apply on the points before drawing them. But this code should be self-explanatory.