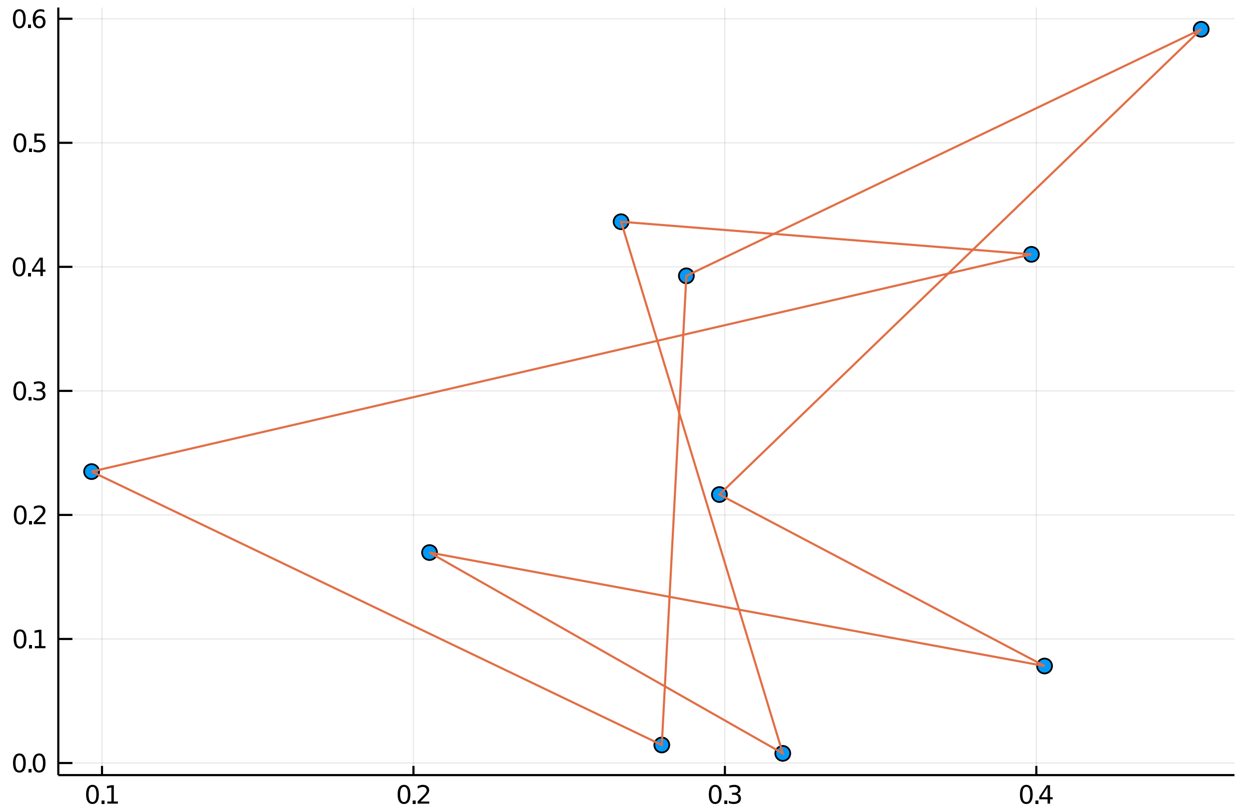
```julia
using LinearAlgebra, Plots
```

Let's make and plot an n-sided random polygon with 2-normalized vertices.

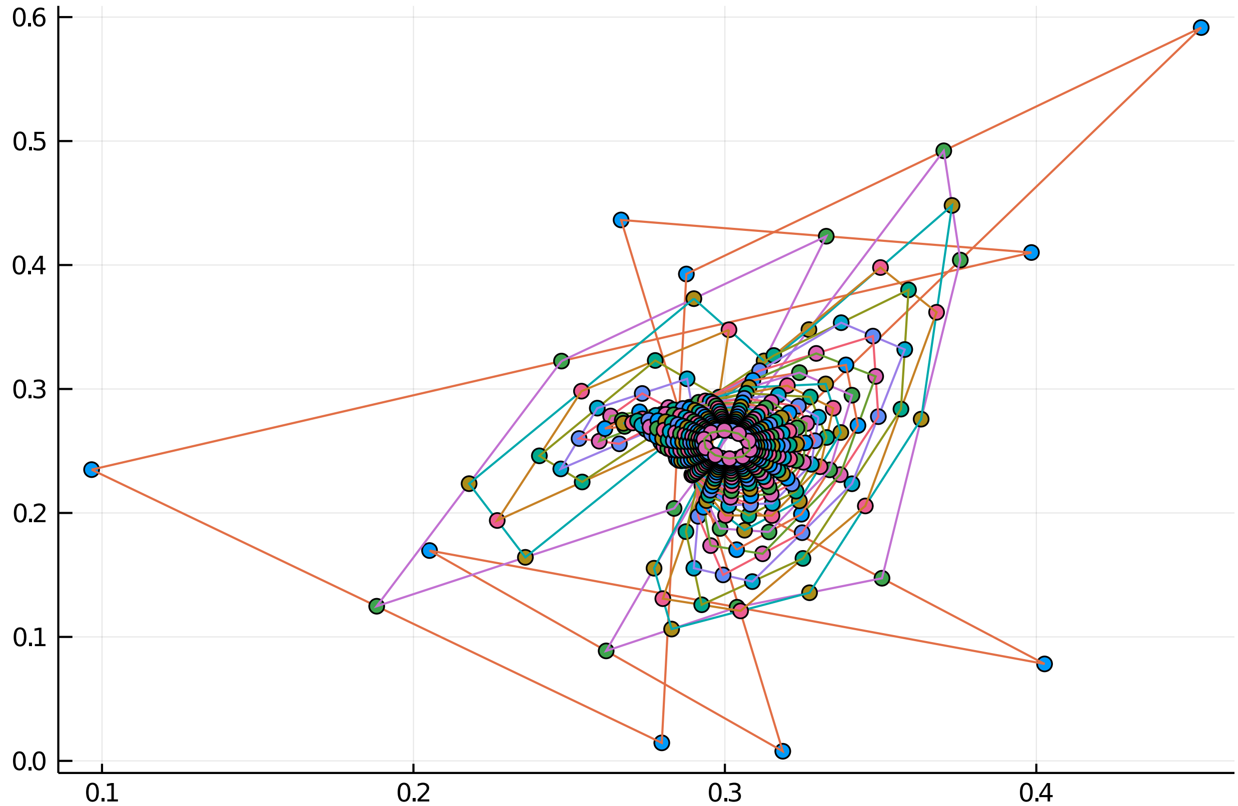
```
Float64[0.234947, 0.410083, 0.436373, 0.00777053, 0.169674, 0.0782119, 0.216452,
```

```julia
begin
    n = 10
    x = normalize!(rand(n))
    y = normalize!(rand(n))
end
```




```julia
begin
    scatter(x, y; legend=false)
    plot!([x; x[1]], [y; y[1]])
end
```

What happens when we consider the sequence of polygons whose vertices are the midpoints of previous polygon's vertices?




```
begin
    x1 = x[1]
    y1 = y[1]
    for i in 1:length(x)-1
        x[i] = (x[i]+x[i+1])/2
        y[i] = (y[i]+y[i+1])/2
    end
    x[end] = (x[end]+x1)/2
    y[end] = (y[end]+y1)/2
    scatter!(x, y; legend=false)
    plot!([x; x[1]], [y; y[1]])
end
```

Observation 1: they averaged polygons converge to a point, the centroid:

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i, \quad \text{and} \quad \bar{y} = \frac{1}{n}\sum_{i=1}^{n} x_i.$$

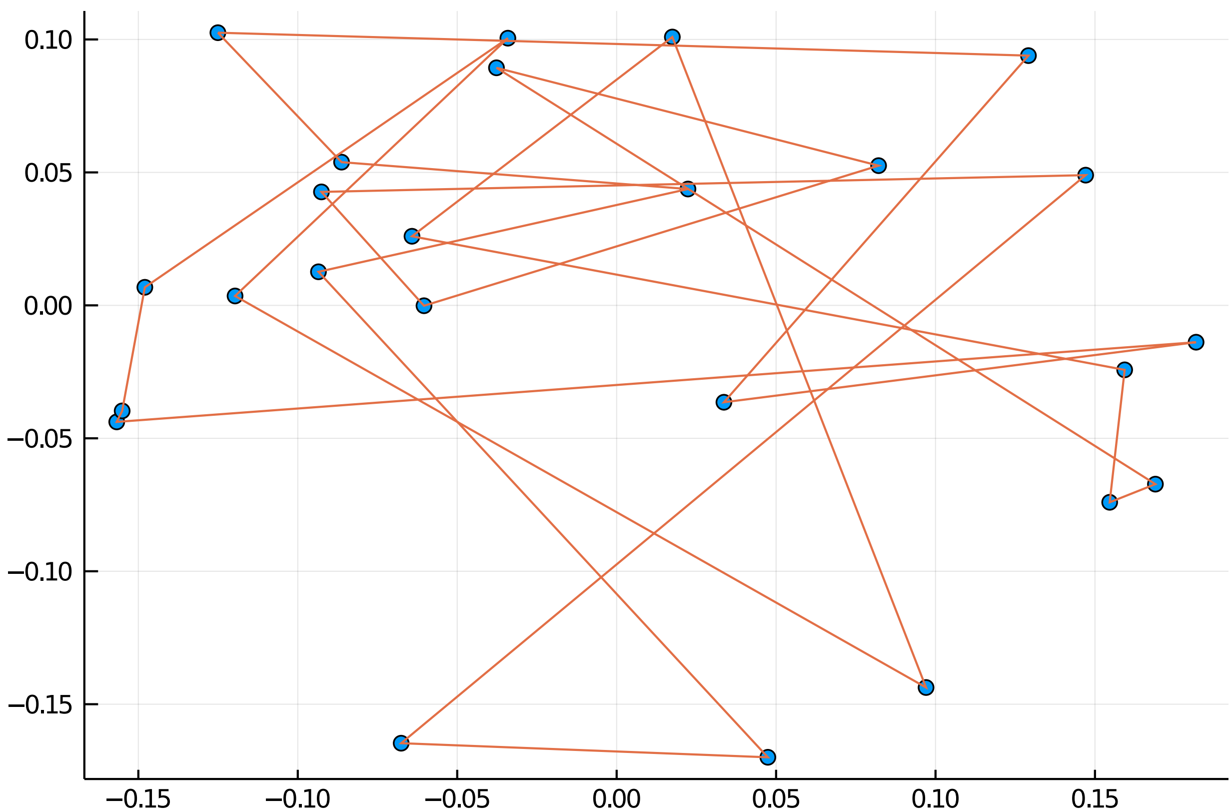
```
(0.30067, 0.255235)
```

```
sum(x)/n, sum(y)/n
```

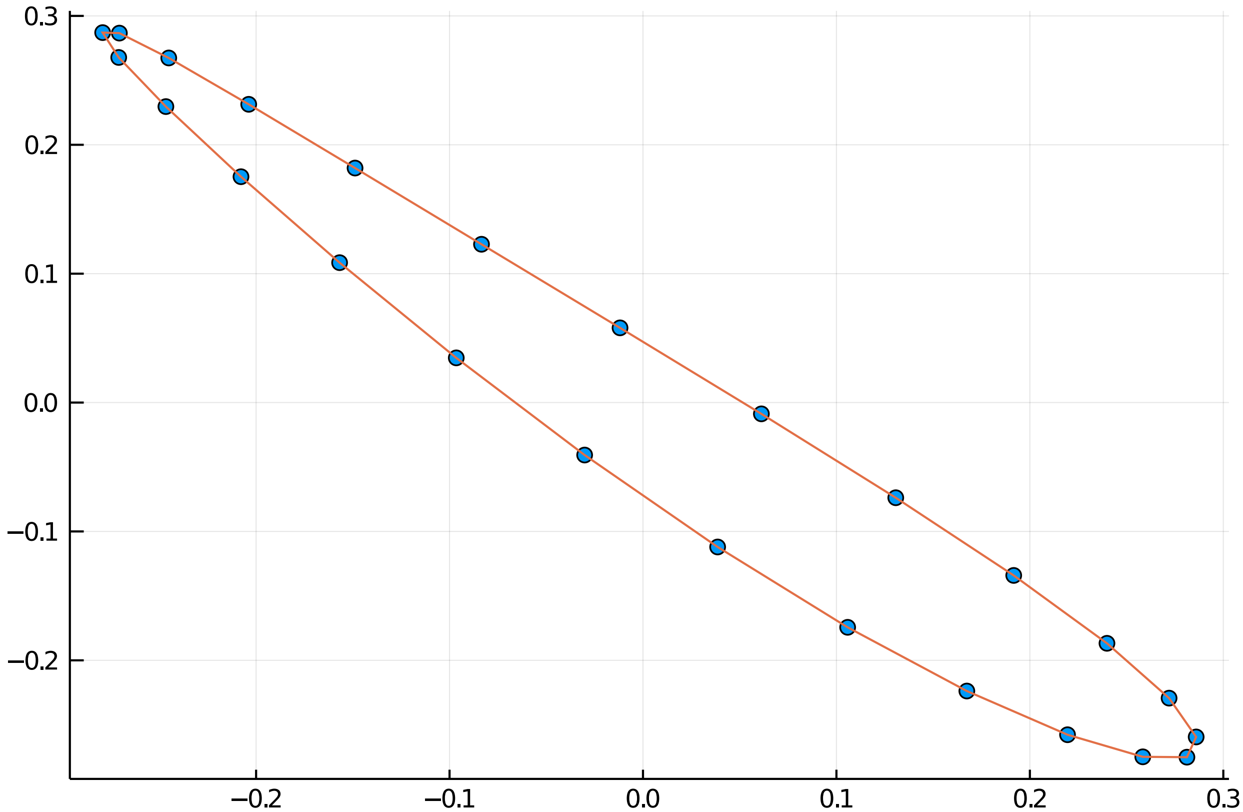
If we work with centroid-0 polygons and renormalize at every iteration, we will no longer converge to a point, but perhaps we can observe other phenomena.

```
Float64[0.102536, 0.0939218, -0.0364424, -0.013861, -0.0438241, -0.0396972, 0.00(
```

```julia
begin
    m = 25
    u = normalize!(rand(m))
    v = normalize!(rand(m))
    u .-= sum(u)/m
    v .-= sum(v)/m
end
```




```julia
begin
    scatter(u, v; legend=false)
    plot!([u; u[1]], [v; v[1]])
end
```




```
begin
    u1 = u[1]
    v1 = v[1]
    for i in 1:length(u)-1
        u[i] = (u[i]+u[i+1])/2
        v[i] = (v[i]+v[i+1])/2
    end
    u[end] = (u[end]+u1)/2
    v[end] = (v[end]+v1)/2
    normalize!(u)
    normalize!(v)
    scatter(u, v; legend=false)
    plot!([u; u[1]], [v; v[1]])
end
```

Observation 2: The vertices of the polygons converge to the boundary of an ellipse at a 45° angle.

Observation 3: After converging (to plotting accuracy), polygons of even iterates (and odd iterates) appear almost the same.

The averaging of vertices can be described in terms of a matrix-vector product:

$$\begin{bmatrix} x_1^{(k+1)} \\ x_2^{(k+1)} \\ \vdots \\ x_{n-1}^{(k+1)} \\ x_n^{(k+1)} \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 1 & 1 & & & \\ & 1 & 1 & & \\ & & \ddots & \ddots & \\ & & & \ddots & 1 \\ 1 & & & & 1 \end{bmatrix} \begin{bmatrix} x_1^{(k)} \\ x_2^{(k)} \\ \vdots \\ x_{n-1}^{(k)} \\ x_n^{(k)} \end{bmatrix},$$

with a similar equation holding for y.

This matrix can be described in terms of the $n \times n$ identity, I_n, and the circular shift:

$$S_n = \begin{bmatrix} 0 & 1 & & & \\ & 0 & 1 & & \\ & & \ddots & \ddots & \\ & & & \ddots & 1 \\ 1 & & & & 0 \end{bmatrix},$$

so that $x^{(k+1)} = \frac{1}{2}(I_n + S_n)x^{(k)}$.

In fact, not a single entry needs to be stored in order to apply this matrix to a vector. In Julia, the *array interface* consists of at least two functions: `size` and `getindex`.

```
begin
    struct HalfIdentityPlusCircularShift{T} <: AbstractMatrix{T}
        n::Int
    end
    Base.size(A::HalfIdentityPlusCircularShift{T}) where T = (A.n, A.n)
    function Base.getindex(A::HalfIdentityPlusCircularShift{T}, i, j) where T
        n = A.n
        if (i == j || i+1 == j) && 1 ≤ i ≤ n && 1 ≤ j ≤ n
            return T(0.5)
        elseif i == n && j == 1
            return T(0.5)
        else
            return T(0)
        end
    end
end
```

```
10×10 HalfIdentityPlusCircularShift{Float64}:
 0.5  0.5  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.5  0.5  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.5  0.5  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.5  0.5  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.5  0.5  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.5  0.5  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.5  0.5  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.5  0.5  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.5  0.5
 0.5  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.5
```

```
HalfIdentityPlusCircularShift{Float64}(n)
```

```
10×10 HalfIdentityPlusCircularShift{Rational{Int64}}:
 1//2  1//2  0//1  0//1  0//1  0//1  0//1  0//1  0//1  0//1
 0//1  1//2  1//2  0//1  0//1  0//1  0//1  0//1  0//1  0//1
 0//1  0//1  1//2  1//2  0//1  0//1  0//1  0//1  0//1  0//1
 0//1  0//1  0//1  1//2  1//2  0//1  0//1  0//1  0//1  0//1
 0//1  0//1  0//1  0//1  1//2  1//2  0//1  0//1  0//1  0//1
 0//1  0//1  0//1  0//1  0//1  1//2  1//2  0//1  0//1  0//1
 0//1  0//1  0//1  0//1  0//1  0//1  1//2  1//2  0//1  0//1
 0//1  0//1  0//1  0//1  0//1  0//1  0//1  1//2  1//2  0//1
 0//1  0//1  0//1  0//1  0//1  0//1  0//1  0//1  1//2  1//2
 1//2  0//1  0//1  0//1  0//1  0//1  0//1  0//1  0//1  1//2
```

```
HalfIdentityPlusCircularShift{Rational{Int}}(n)
```

By subtyping our struct as an `AbstractMatrix` and by implementing `size` and `getindex`, we get certain extras for free.

```
Float64[0.294435,  0.293545,  0.29537,  0.299205,  0.3036,  0.306889,  0.307811,  0.305
```

```
HalfIdentityPlusCircularShift{Float64}(n)*x
```

For more information on the random polygons, please see

1. A. N. Elmachtoub and C. F. Van Loan, **From Random Polygon to Ellipse: An Eigenanalysis**, *SIAM Rev.*, **52**:151–170, 2010.