A unit-disk embedding of a graph is a way to represent the vertices and edges of a graph in the Euclidean plane such that each vertex is mapped to a distinct point and each edge is represented by a curve that connects the endpoints of the edge. In a unit-disk embedding, each vertex of the graph is represented by a disk of unit radius and the disks corresponding to adjacent vertices intersect if and only if the corresponding vertices are adjacent in the graph. The goal of a unit-disk embedding is to find a geometric representation of the graph that preserves its connectivity properties and is aesthetically pleasing. Unit-disk embeddings have applications in wireless sensor networks, where the nodes are represented by disks and the communication range between nodes is limited to the radius of their disks.

```
using LinearAlgebra
```

```
using Plots
```

```
using ForwardDiff
```

N = 10

```
N = 10
```

E =
[(1, 2), (1, 3), (2, 3), (2, 4), (2, 5), (2, 6), (3, 5), (3, 6), (3, 7), (4, 5), (4, 8

```
E = [(1, 2), (1, 3),
    (2, 3), (2, 4), (2, 5), (2, 6),
    (3, 5), (3, 6), (3, 7),
    (4, 5), (4, 8),
    (5, 6), (5, 8), (5, 9),
    (6, 7), (6, 8), (6, 9),
    (7,9), (8, 9), (8, 10), (9, 10)]
```

distance (generic function with 1 method)

```
function distance(x, i, j)
    return sqrt((x[2 * i - 1] - x[2 * j - 1])^2 + (x[2 * i] - x[2 * j])^2)
end
```

is_unit_disk (generic function with 1 method)

```
function is_unit_disk(x::AbstractArray)
    E = [(1, 2), (1, 3), (2, 3), (2, 4), (2, 5), (2, 6), (3, 5), (3, 6), (3, 7), (4, 5), (4, 8), (5, 6), (5, 8), (5, 9), (6, 7), (6, 8), (6, 9), (7,9), (8, 9), (8, 10), (9, 10)]

    Loss = 0

    for i in 1:10
        for j in i + 1:10
            if (i, j) in E
                d_ij = distance(x, i, j)
                if d_ij >= 1
                    Loss += 1
                end
            else
                d_ij = distance(x, i, j)
                if d_ij <= 1
                    Loss += 1
                end
            end
        end
    end

    if Loss == 0
        println("The result is a unit disk.")
    else
        println("The result is not a unit disk.")
    end
    return Loss
end
```

Here we define the loss function based on distance, given as

$$L = L_{Edge} + L_{NonEdge}$$
$$L_{Edge} = \sum_{i,j} 2 * |x - 0.95|^{0.2} * \mathrm{sign}(x - 0.95)$$
$$L_{NonEdge} = -\sum_{i,j} |x - 1.05|^{0.2} * \mathrm{sign}(x - 1.05)$$

Here we set the cutoff slightly larger or smaller than 1 to fasten the convergence. Such a loss function will have a continium `ForwardDiff` so that gradient based optimalizer can be applied.

Loss_distance_12 (generic function with 1 method)

```julia
function Loss_distance_12(x::AbstractArray)
    E = [(1, 2), (1, 3), (2, 3), (2, 4), (2, 5), (2, 6), (3, 5), (3, 6), (3, 7), (4, 5), (4, 8), (5, 6), (5, 8), (5, 9), (6, 7), (6, 8), (6, 9), (7,9), (8, 9), (8, 10), (9, 10)]

    Loss_1 = 0
    Loss_2 = 0
    cut_1 = 0.95
    cut_2 = 1.05

    for i in 1:10
        for j in i + 1:10
            if (i, j) in E
                d_ij = distance(x, i, j)

                if d_ij >= cut_1
                    Loss_1 += 2 * (abs(d_ij - cut_1))^(.2) * sign(d_ij - cut_1)
                end
            else
                d_ij = distance(x, i, j)
                if d_ij <= cut_2
                    Loss_2 += - (abs(d_ij - cut_2))^(.2) * sign(d_ij - cut_2)
                end
            end
        end
    end

    return Loss_1, Loss_2
end
```

Loss_distance (generic function with 1 method)

```julia
function Loss_distance(x::AbstractArray)
    Loss_1, Loss_2 = Loss_distance_12(x::AbstractArray)
    Loss = Loss_1 + Loss_2
    return Loss
end
```

[0.468621, 0.0499091, 0.319865, -0.73502, 0.0456524, -0.596625, -0.981266, -1.38465,

```julia
ForwardDiff.gradient(Loss_distance, rand(20))
```

```
gradient_descent (generic function with 1 method)
```

```julia
function gradient_descent(f, x; niters::Int, learning_rate::Real)
    history = [x]
    for i=1:niters
        g = ForwardDiff.gradient(f, x)
        x -= learning_rate * g
        push!(history, x)
    end
    return x, f(x)
end
```

```
0
```

```julia
begin
    x_0 = 3 * rand(20)
    best_x, best_loss = gradient_descent(Loss_distance, x_0; niters = 500000,
    learning_rate = 0.0001)
    result = is_unit_disk(best_x)
end
```

```
The result is a unit disk.
```

As shown by the cell above, there are no *illegal* disks.

A plotted result via plot.py is given by **unit_disk_py**.