```
begin
    using Plots
    using LinearAlgebra
    using StaticArrays
    using StatsBase
end
```

```
function svectors(x::AbstractMatrix{T}, ::Val{N}) where {T,N}
    """ Converts matrices to vectors of SVectors"""
    size(x,1) == N || error("sizes mismatch")
    isbitstype(T) || error("use for bitstypes only")
    reinterpret(SVector{N,T}, vec(x))
end;
```

```
logrange(x1, x2, n) = (10^y for y in range(log10(x1), log10(x2), length=n));
```

Return to the origin

For a *finite* random walk of (large) length n, it is known that the expected number of returns to the origin T_n scales like follows:

$$\langle T_n \rangle \sim \begin{cases} \sqrt{n} & d = 1 \\ \log(n) & d = 2 \\ C_d & d \geq 3 \end{cases}$$

Notice that for $d \geq 3$, $\langle T_n \rangle$ does **not** grow with n, which must mean that the walker somehow "escapes" and never returns back to the origin. The probability of return to the origin is less than 1! For an infinite-length random walk, indeed the probability of returning to the origin ρ is seen to be

$$\rho \sim \begin{cases} 1 & d = 1 \\ 1 & d = 2 \\ < 1 & d \geq 3 \end{cases}$$

The *intuitive* explanation of this amazing fact is that, as the dimension d grows, there are "more directions available", and so more chances for the walker to "get lost" and never return to the origin. There is of course a formal proof as well, but today we will do a **computational verification** of these facts, which is no substitute for a formal proof but is often all we can do!

Generating Random Walks

Exercise 3.1

Write a function that generates a random walk of given length in d dimensions. Your random walker should move as follows:

- At each time-step, the walker moves only in one direction.
- At each time-step, the walker moves only by -1 or +1

```
function get_traj(length, dim)
    """Generate a RW in d dimensions

    Parameters
    ----------
    length: Int
        Length of the RW.
    dim: Int
        Dimension of the RW

    Returns
    -------
    traj : Matrix, (length, dim)
        The positions of the RW.

    Notes
    -----
    At each time-step, the walker moves in only one direction.
    At each time-step, the walker moves by -1 or +1
    """
    choices = hcat(Matrix(1.0I, dim, dim), Matrix(-1.0I, dim, dim))
    steps = vcat([@SVector(zeros(Float64, dim))], sample(svectors(choices, Val{dim}
()), length-1))
    traj = cumsum(steps)
     return if dim > 1
         reinterpret(reshape, Float64, traj) |> Transpose
     else
         reinterpret(reshape, Float64, traj)
     end
end;
```

```
get_traj(; length=100, dim=2) = get_traj(length, dim);
```

Verification

To make sure that your function works correctly, execute the following cell. Notice the use of `assert` statements: execution should fail if something goes wrong. If everything is fine, nothing should happen.

```
# basic checks for your RW generator
for dim in 1:5
    for length in [10, 100, 200, 500]
        traj = get_traj(length, dim)
        print(traj)
        # make sure traj has the right shape
        if dim > 1
            @assert size(traj) == (length, dim)
        else
            @assert size(traj) == (length,)
        end
        #@assert all(isequal((dim,)) ∘ size, traj)
        # make sure all steps are -1 or 1 in only one direction
        @assert all(isone, sum(!iszero, diff(traj, dims=1), dims=2))
    end
end
```

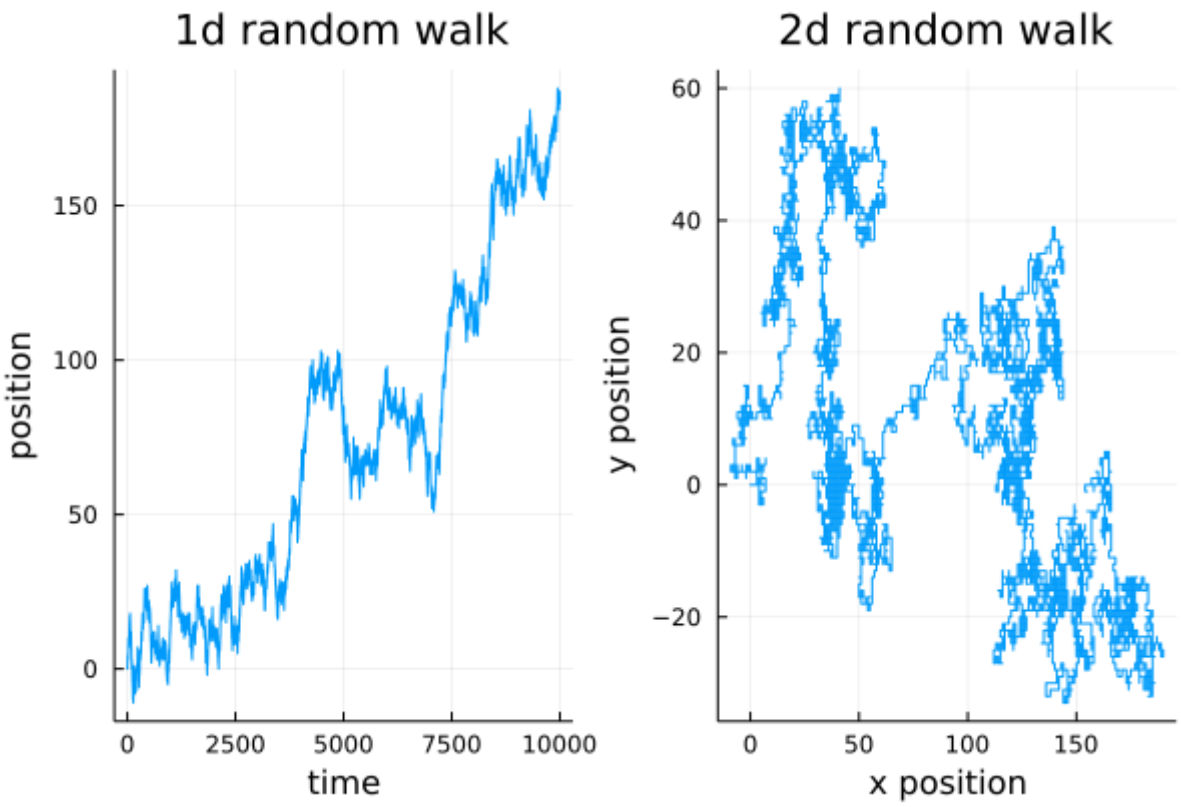
Exercise 3.2

Plot a random walk of length 10^4 for $d = 1$ (time in x-axis, position in y-axis) and $d = 2$ (x,y components in x,y-axis). Remember to use **axis labels**.

```
# it is better if you use one cell to generate the random walks, and a second cell to plot them
length = Int(1e4);
```

```
RW_1d = get_traj(length=length, dim=1);
```

```
RW_2d = get_traj(length=length, dim=2);
```




```
begin
    # We generate a figure with two subplots, called axis in Plots.jl.

    p1 = plot(RW_1d, xlabel="time", ylabel="position", title="1d random walk")
    p2 = plot(RW_2d[:,1], RW_2d[:,2], xlabel = "x position", ylabel = "y position",
title = "2d random walk")
    plot(p1, p2, layout=(1,2), legend=false)
end
```

Counting the number of returns to the origin

Since we are interested in how **the expected number of returns to the origin** scales with the RW length, we don't need to store the whole trajectory of each simulation (we will be performing many simulations!).

Exercise 3.3

Write a function that generates a RW of given length and dimension (calling `get_traj`), and returns the number of times it returned to the origin. To count the number of returns to the origin, you might need to use the following functions:

```
all()
zeros()
```

```
function get_num_returns(length, dim)
    # generate a RW of given length and dimension
    traj = get_traj(length, dim)
    # count how many times it goes through the origin
    count(iszero, eachrow(traj))
end;
```

Exercise 3.4

Write a function that computes the expected number of returns to the origin for a given length and dimension. Your function will call `get_num_returns()`, and should have an additional parameter that sets the sample size.

```
function get_average_num_returns(length, dim, num_trajs=200)
    #average_num_returns
    return mean(get_num_returns(length, dim) for _ in 1:num_trajs)
end;
```

```
get_average_num_returns(;length = 100, dim = 2, num_trajs=200) =
get_average_num_returns(length, dim, num_trajs);
```

Comparing with analytical results

We are now ready to compare our analytical results with numerical simulations! We want to plot the expected number of returns to the origin as a function of the RW length. To do this, it is useful to first define an array of RW lengths.

```
# define range of RW lengths
length_array = let length_min = 10, length_max = 100_000;
    # generate points logarithmically spaces
    # and convert them to integers
    [
        Int(round(x))
        for x in logrange(length_min, length_max, 20)
    ]
end;
```

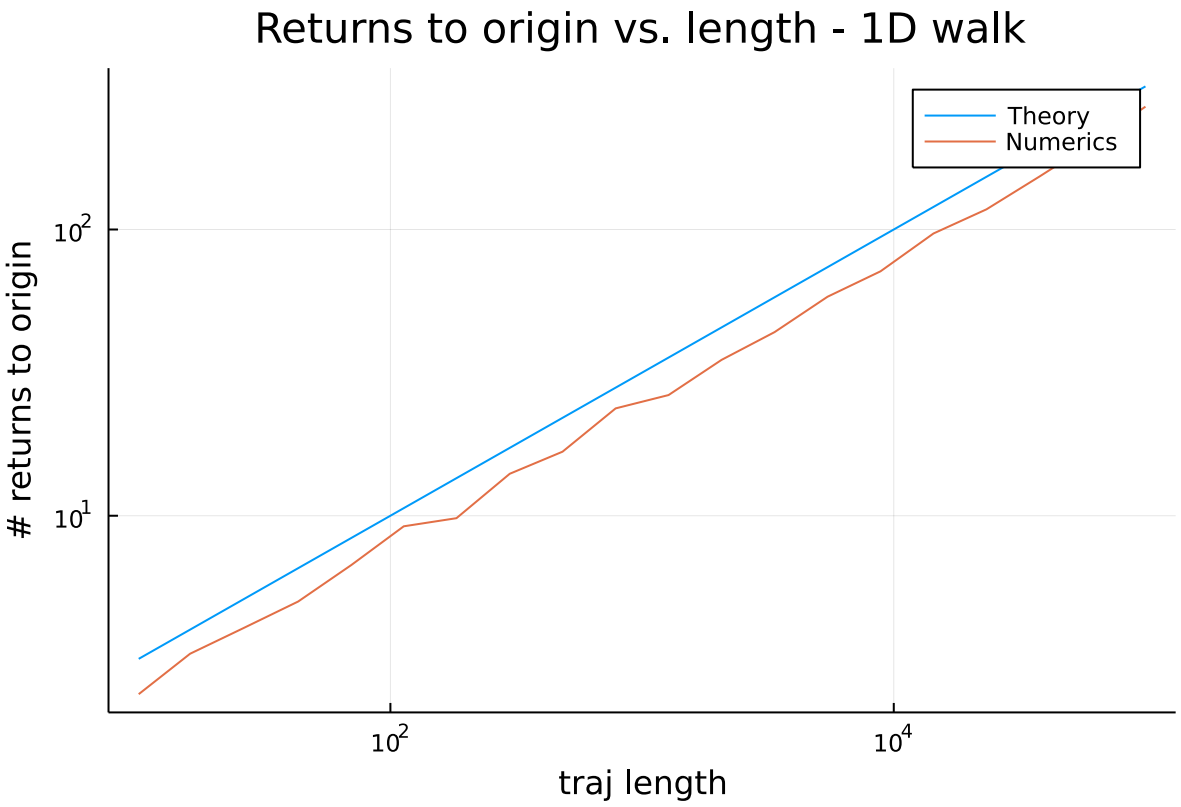
(tip: if your RW generating function is not very efficient, you might want to decrease `length_min`)

Executing the following cell will run all simulations for $d = 1$

```
num_returns_array_1D = let dim=1, num_trajs = 200
    [
        get_average_num_returns(length=length, dim=dim, num_trajs=num_trajs)
        for length in length_array
    ]
end;
```

Exercise 3.5

Plot the average number of returns to the origin of a 1D RW as a function of the RW length, together with the expected theoretical result. Do your results verify the $n^{1/2}$ scaling? **Tip** Use double-logarithmic scales in your plot. Remember to include label axis, and a legend!




```
begin
    # plot theoretical result
    plot(length_array, [sqrt.(length_array), num_returns_array_1D], label=["Theory" "Numerics"], legend=true)
    # add axis labels
    xaxis!("traj length", :log10)
    yaxis!("# returns to origin", :log10)
    # add a title (e.g. that says what dimension we used)
    title!("Returns to origin vs. length - 1D walk")
end
```

Exercise 3.6

Plot the average number of returns to the origin of a 2D RW as a function of the RW length. Do your results verify the $log(n)$ scaling? What are the best axis scales to use in this case?

```
num_returns_array_2D = let dim = 2, num_trajs = 200
    [
        get_average_num_returns(length=length, dim=dim, num_trajs=num_trajs)
        for length in length_array
    ]
end;
```


```
# plot theoretical result
let k = 0.3
    plot(length_array, [k*log.(length_array), num_returns_array_2D], label=["Theory" "Numerics"], legend=true)
    # add axis labels
    xaxis!("traj length", :log10)
    yaxis!("# returns to origin")
    # add a title (e.g. that says what dimension we used)
    title!("Returns to origin vs. length - 2D walk")
end
```

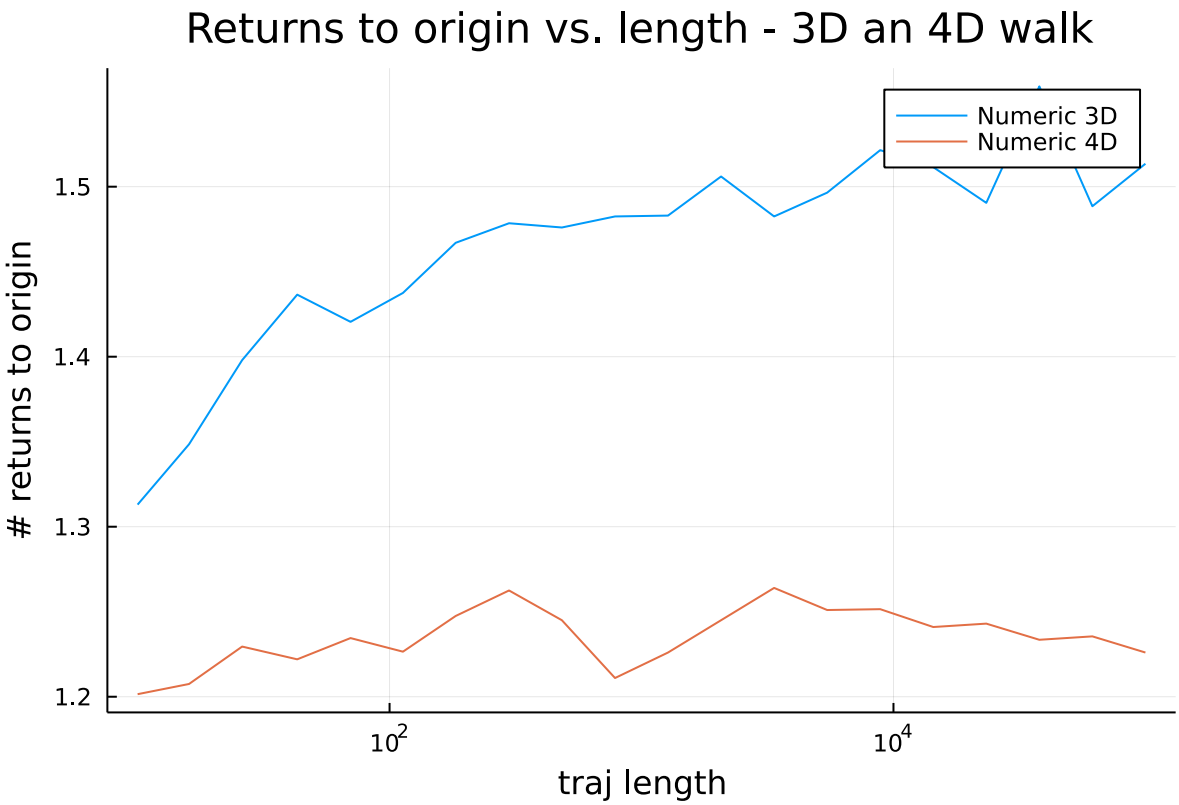
Notice that we had to put a multiplicative contant to the theoretical model to ease comparison with the numerical result

Exercise 3.7

Show numerically that, for $d = 3$ and $d = 4$, the expected number of returns to the origin is **constant**.

```
# do the simulations for d=3
num_returns_array_3D = let dim = 3, num_trajs = 2_000
    [
        get_average_num_returns(length, dim, num_trajs)
        for length in length_array
    ]
end;
```

```
# do the simulations for d=4
num_returns_array_4D = let dim = 4, num_trajs = 2_000
    [
        get_average_num_returns(length, dim, num_trajs)
        for length in length_array
    ]
end;
```




```
begin
    # plot theoretical result
    plot(length_array, [num_returns_array_3D, num_returns_array_4D], label=["Numeric 3D" "Numeric 4D"], legend=true)
    # add axis labels
    xaxis!("traj length", :log10)
    yaxis!("# returns to origin")
    # add a title (e.g. that says what dimension we used)
    title!("Returns to origin vs. length - 3D an 4D walk")
end
```

Self-Avoiding Walks

Self-avoiding walks (SAW) are simply random walks in a regular lattice with the additional constraint that no point can be visited more than once. That is, SAWs cannot intersect themselves. The most well-known application of SAW is to model linear polymers, where obviously two monomers cannot occupy the same space (excluded volume effect).

You can read more about self-avoiding walks in this nice introduction by Gordon Slade:

Self-Avoiding Walks, by Gordon Slade

Simulating Self-Avoiding Walks

Generating a SAW is not trivial. If you try to generate a SAW stochastically, that is, one step at a time, you will miserably fail: your walker might get into traps (configurations with no allowed movements), and if it does you will have to discard your simulation. It turns out you will have to discard your simulation *really* often, so that for large lengths, you will basically never find a valid path. In addition, the paths you will find for short lengths will not come up with the right probabilities. Bear in mind that we want to **uniformly sample** the set of SAW of given length n, SAW(n). That is, we want that all paths from SAW(n) are generated with the same probability.

The solution is to use a Monte Carlo algorithm that, given one element $\alpha \in \mathrm{SAW}(n)$, generates a new one $\beta \in \mathrm{SAW}(n)$ with some probability $P_{\alpha\beta}$. If in addition our algorithm satisfies **detailed balance** and is **ergodic**, then we known that it will converge to the equilibrium distribution (the uniform distribution in our case).

The pivot algorithm

We will implement the pivot algorithm, which is simple, effective, and satisfies detailed balance and ergodicity. You can read about the details of the pivot algorithm here:

The Pivot Algorithm: A Highly Efficient Monte Carlo Method for the Self-Avoiding Walk

(tip: if you're at home, **do not** use tools such as sci-hub to download the paper).

Given a self-avoiding walk of length n, the pivot algorithm generates the next walk $\beta \in \mathrm{SAW}(n)$ as follows:

1. **Choose a point of α at random**, splitting the path in two bits: the head (from the origin to the chosen point) and the tail (from the chosen point to the end of the path). Notice that both the head and the tail are SAWs.
2. **Apply a transformation to the tail**, leaving the head intact. The transformation must be an orthoganl transformation that leaves the regular lattice intact (so, either a reflection or a 90^o, 180^o or 270^o rotation). For simplicity, we will use only **rotations** (read the paper to see why this is ok).
3. **Check if the new path is self-avoiding**. If so, return it. Otherwise, return the original path.

Iterating these steps one obtains a **Markov** chain of SAWs: $\alpha_1 \to \alpha_2 \to \cdots \to \alpha_M$. Notice that α_i are not uncorrelated, but because the algorithm satisfies detailed balance and is ergodic, we know that it approaches the equilibrium distribution. This means that we can use our Markov chain to compute **expected values** as long as it is long enough.

Implementing the pivot step in 2D

To implement the **pivot algorithm** in 2D, we will write one function that does steps 1 and 2, and another function that does step 3. We will also need a function to generate standard 2D random walks.

Exercise 3.8

Write a function `get_traj` that generates a 2D random walk of given length.

```
function get_traj(length)
    """Generate a RW in 2 dimensions

    Parameters
    ----------
    length: Int
        Length of the RW.

    Returns
    -------
    traj : Matrix, (length, 2)
        The positions of the RW.

    Notes
    -----
    At each time-step, the walker moves in only one direction.
    At each time-step, the walker moves by -1 or +1
    """
    choices = hcat(Matrix(1.0I, 2, 2), Matrix(-1.0I, 2, 2))
    steps = vcat([@SVector(zeros(Float64, 2))], sample(svectors(choices, Val{2}()), length-1))
    traj = cumsum(steps)
    reinterpret(reshape, Float64, traj) |> Transpose
end;
```

```
function rotation_matrix(direction, point=[0,0])
    """Return a 2D matrix that rotates 90°, 180° or 270°"""
    @assert direction in (0, 1, 2)
    @assert size(point) == (2,)
    (x,y) = point
    (sinθ, cosθ) = if direction == 0
        (1, 0)
    elseif direction == 1
        (0, -1)
    elseif direction == 2
        (-1, 0)
    end
    [ cosθ -sinθ (-x*cosθ + y*sinθ + x)
      sinθ  cosθ (-x*sinθ - y*cosθ + y)
       0     0             1          ]
end;
```

```
function pivot_traj(traj)
    @assert size(traj, 1) > 1 "Trajectory must be at least 2 steps long"
    pivot = rand(1:size(traj, 1)-1)
    (head, tail) = (traj[begin:pivot,:], traj[pivot+1:end,:])
    tail = (hcat(tail, ones(size(tail, 1))) * transpose(rotation_matrix(rand(0:2), head[end,:])))[:, 1:2]
    return vcat(head, tail)
end;
```

Exercise 3.10

Write a function that counts the number of self-intersections of a RW. Notice that SAWs have 0 self intersections, so that will solve step 3 of the pivot algorithm, but will also be useful to generate the initial condition. One way of approaching this exercise is to count how many *different* points the path visits.

```
function count_self_intersections(traj)
    """Count the number of self-intersections of a RW"""
    num_self_intersections = 0
    for (i, point) in enumerate(eachrow(traj))
        if any(isequal(point), eachrow(traj[begin:i-1, :])) # point in preceding
trajectory
            num_self_intersections += 1
        end
    end
    num_self_intersections
end;
```

Exercise 3.11

Verify that your `count_self_intersecitons` function works properly by using short trajectories for which you know the answer.

```
traj = [0. 0.
        0. 1.
        1. 1. #
        1. 2. #
        2. 2. #
        1. 2. # 1st intersection
        1. 1. # 2nd intersection
        2. 1.
        2. 2. # 3rd intersection
        3. 2. #
        3. 1.
        3. 2. # 4th intersection
        3. 3. #
        2. 3. #
        1. 3. #
        1. 2. # 5th intersection
        1. 3. # 6th intersection
        2. 3. # 7th intersection
        3. 3. # 8th intersection
        3. 2. # 9th intersection
        4. 2.];
```

We can count 9 intersections in the trajectory above

```
9
```

```
count_self_intersections(traj)
```

The verification is thus complete.

Generating the initial condition

You might have noticed that the pivot algorithm requires an element of SAW(n) as starting condition, to then generate a Markov chain easily. But how do you get this first element? We will use the following strategy:

1. Generate a standard 2D random walk, and count the number of self intersections.
2. Apply the pivot transformation to get a new 2D random walk, and count the number of self-intersections-
3. If the number of self-intersections has decreased or not changed, keep the new path. Else, keep the old one.
4. Go to 2, till the number of self-intersections is 0.

Exercise 3.12

Write a function `get_first_SAW` that generates a SAW of given length

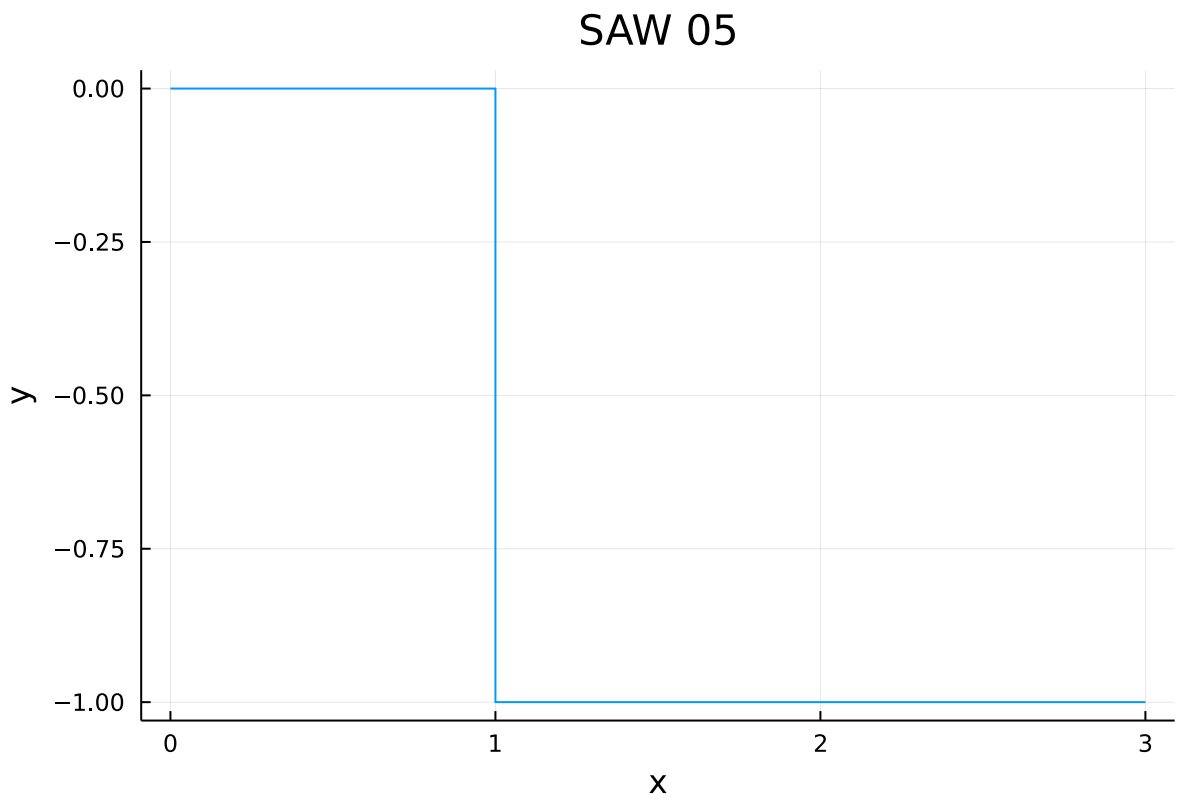
```
function get_first_SAW(length, max_tries=1_000_000)
    traj = get_traj(length)
    traj_inter = count_self_intersections(traj)
    for _ in 1:max_tries
        if traj_inter == 0
            return traj
        end
        candidate = pivot_traj(traj)
        candidate_inter = count_self_intersections(candidate)
        if candidate_inter ≤ traj_inter
            traj = candidate
            # if candidate_inter < traj_inter: # This is just to have an indicator of progress
            #    print(candidate_inter)
            traj_inter = candidate_inter
        end
    end
    error("No SAW found after $max_tries tries")
end;
```

Exercise 3.13

Generate and plot some 2D SAWs of different lengths. Be carefull, raise the length slowly! You can measure how long a cell takes executing watching the time indicator at the bottom-right of a cell.

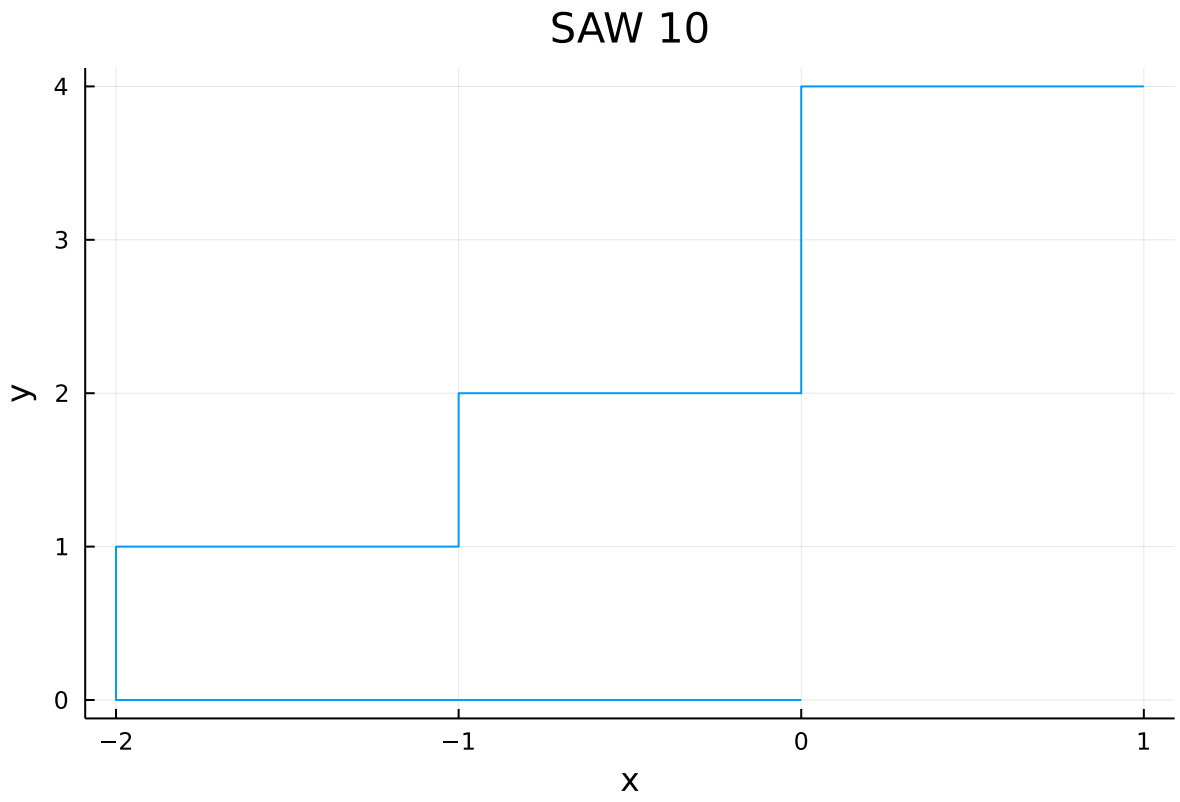
CAUTION!!! Some of the generated RW may have configurations such that it is unlikely that they will be pivoted into a SAW within `max_tries` attempts. In such cases trying again may lead to success, in particular with shorter walks.

```
SAW05 = get_first_SAW(5);
```



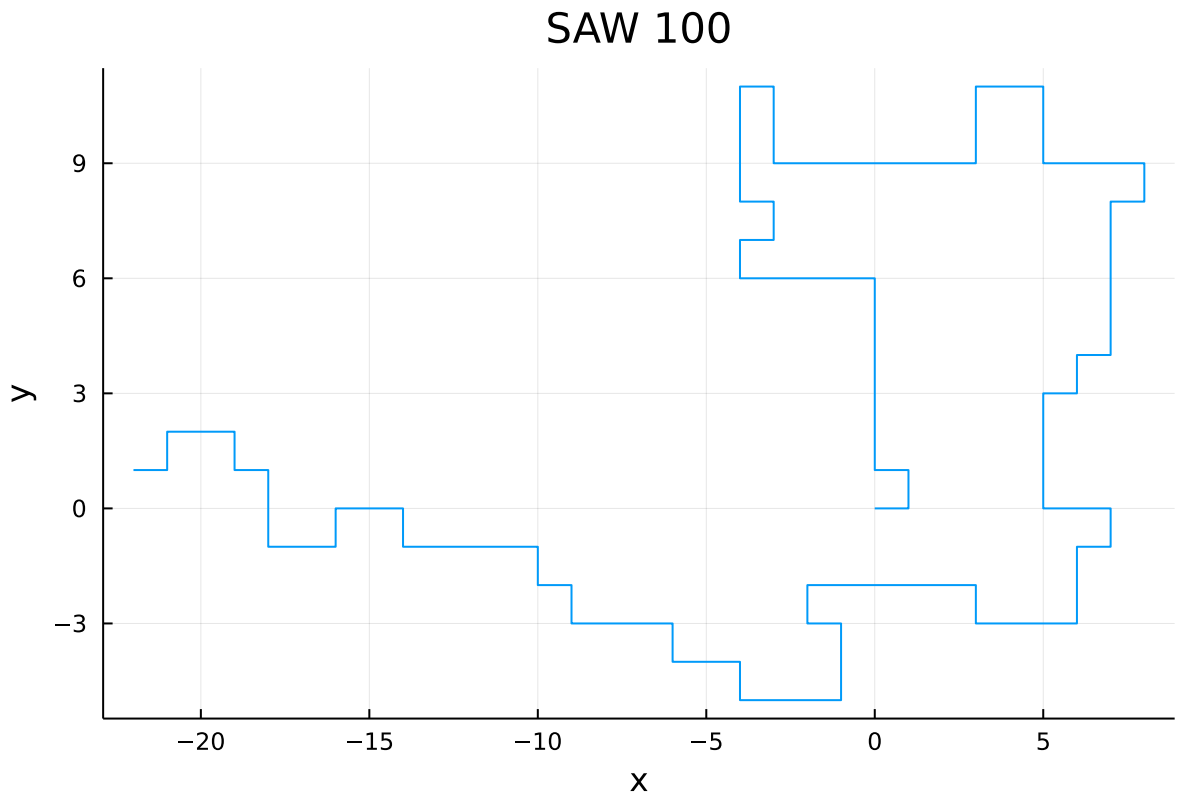

```
plot(SAW05[:, 1], SAW05[:,2], xlabel = "x", ylabel = "y", title = "SAW 05",
legend=false)
```

```
SAW10 = get_first_SAW(10);
```



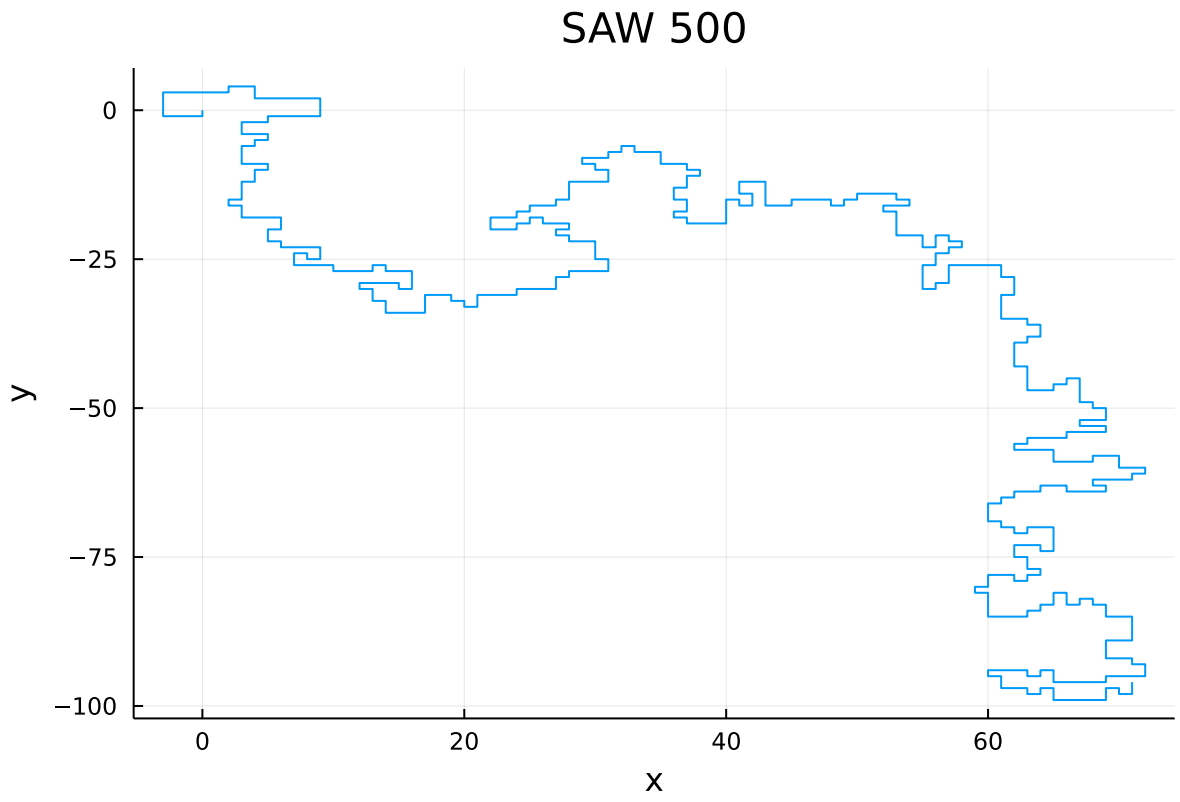

```
plot(SAW10[:, 1], SAW10[:,2], xlabel = "x", ylabel = "y", title = "SAW 10",
legend=false)
```

```
SAW100 = get_first_SAW(100);
```



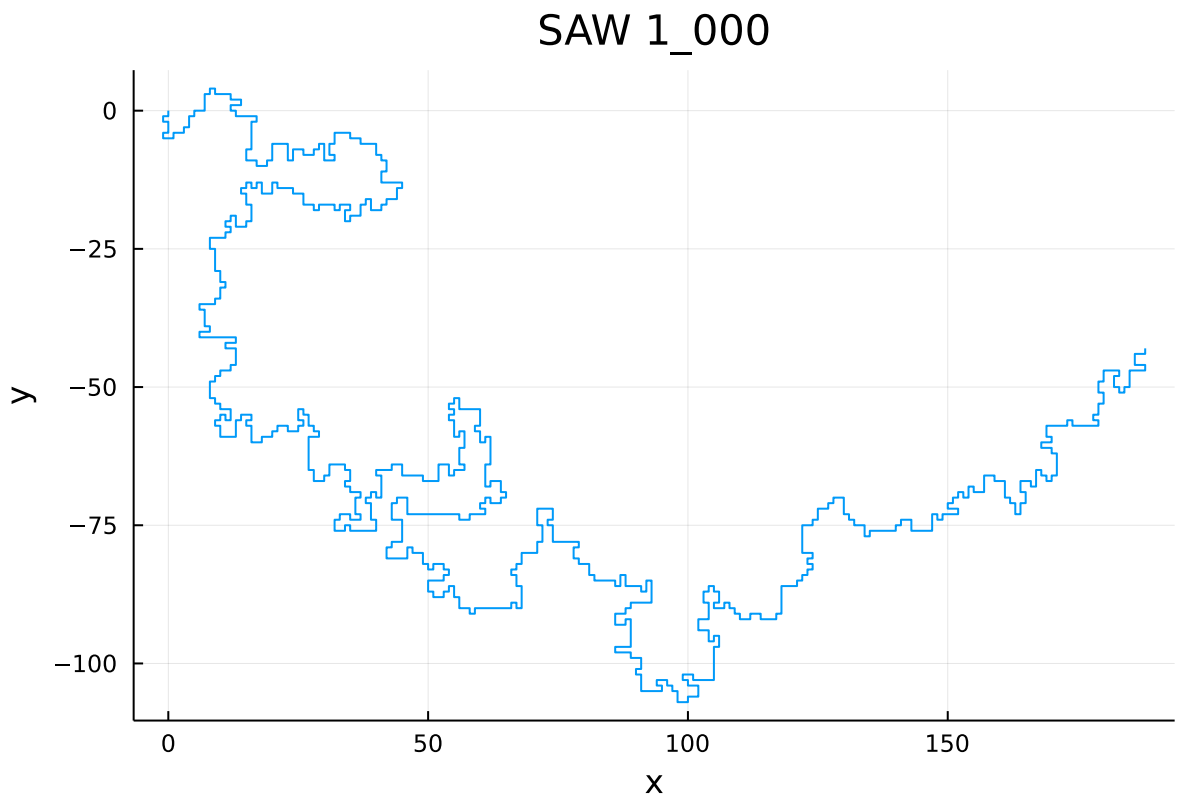

```
plot(SAW100[:, 1], SAW100[:,2], xlabel = "x", ylabel = "y", title = "SAW 100",
legend=false)
```

```
SAW500 = get_first_SAW(500);
```




```
plot(SAW500[:, 1], SAW500[:,2], xlabel = "x", ylabel = "y", title = "SAW 500",
legend=false)
```

```
SAW1_000 = get_first_SAW(1_000);
```




```
plot(SAW1_000[:, 1], SAW1_000[:,2], xlabel = "x", ylabel = "y", title = "SAW 1_000", legend=false)
```

Exercise 3.14

Write a function `get_next_SAW` that, given a SAW, generates another SAW using the pivot algorithm. Your function should check that the input RW is really a SAW. Remember the steps:

1. Apply the pivot transformation
2. Check if the new path is self-avoiding. **If so, return it. Otherwise, return the original path.**

```
function get_next_SAW(traj)
    # make sure input traj is SAW
    @assert count_self_intersections(traj) == 0

    # pivot step
    proposed_traj = pivot_traj(traj)

    # count intersections
    num_intersections = count_self_intersections(proposed_traj)

    # if it's a SAW
    if iszero(num_intersections)
        return proposed_traj
    end
    return traj
end;
```

Mean Squared Displacement in SAW

A quantity of interest in RWs is the mean squared displacement, which is simply the (squared) distance between the endpoints of the walk. Usually, one writes

$$\left\langle X(n)^2 \right\rangle \sim n^{2\nu}$$

As you know, for a standard RW of n steps, the mean-squared displacement scales like n, so $\nu = 1/2$. However, the exponent for SAW is **different**! Althought it has not been formally proven (still), it is believed that the exponent for SAW is $\nu = 3/4$. That is, for a self-avoiding random walk, the mean squared displacement scales as $n^{3/2}$.

Exercise 3.15

Explain why it makes sense that the mean-squared displacement exponent of SAW is **greater** than that of standard RW.

Explanation

The thesis is sensible since we expect the self-avoidance constraint to act as a repulsor from the points already visited, thus increasing displacement proportionally to the number of already visited points.

A more quantitative approach might be to consider the tightest (i.e. least displaced) and loosest (i.e. most displaced) walks and see what they tell us about the lower and upper limits of the scale relation between displacement and walk length.

Let us assume that exists a SAW that completely fills all n lattice points enclosed within a certain distance R from the origin: for sufficiently high n the number of lattice points in $n \propto R^2$, therefore $R \propto n^{1/2}$. Since we also know that, for an isotropic distribution in a finite space of radius R the mean-squared displacement is $\left\langle X(n)^2 \right\rangle \propto R^2$, then:

$$\left\langle X(n)^2 \right\rangle \propto n$$

which implies $\nu > \frac{1}{2}$, since space-filling walks are a lower bound case for displacement.

Then we consider the most displaced SAW: a straight walk. It is easy to calculate that in this case

$$\left\langle X(n)^2 \right\rangle \propto n^2$$

which implies that $\nu < 1$ since straight walks are an upper bound case for displacement.

Therefore we have demonstrated that

$$\nu \in (1/2, 1)$$

Exercise 3.16

Verify numerically the scaling of the mean-squared displacement of SAW. Notice that you don't need to store all the SAWs, just the endpoints. You could follow this scheme:

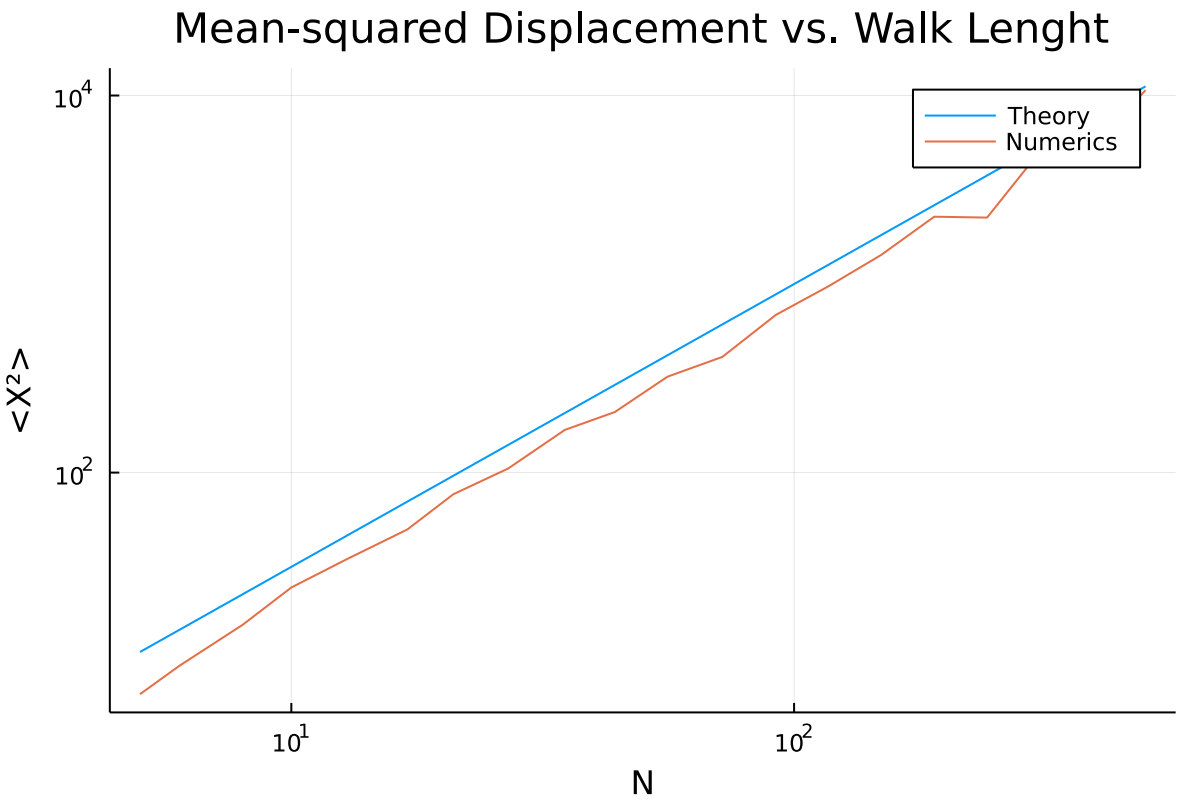
1. Generate a first SAW with your `get_first_SAW()` function
2. Generate the next SAW using your `get_next_SAW()` function, and store the endpoint.
3. Iterate step 2 for as many steps as required
4. Compute the average mean-squared displacement of the stored endpoints

Then repeating steps 1-4 for different lengths, and plot the results in double-logarithmic axis. Compare your results with the theoretical exponent. Do they agree?

```
function get_average_displacement(N, steps=1000)
    SAW = nothing
    sq_displacements = map(1:steps) do _
        SAW = if isnothing(SAW)
            get_first_SAW(N)
        else
            get_next_SAW(SAW)
        end
        norm(SAW[end, :])^2
    end
    return mean(sq_displacements)
end;
```

```
walk_length_array = let length_min = 5, length_max = 500
    [
        Int(round(x))
        for x in logrange(length_min, length_max, 20)
    ]
end;
```

```
average_displacement_array = [get_average_displacement(length) for length in
walk_length_array];
```




```
begin
    plot(walk_length_array, [walk_length_array .^(3//2),
average_displacement_array], label=["Theory" "Numerics"], legend=true)
    xaxis!("N", :log10)
    yaxis!("<X²>", :log10)
    title!("Mean-squared Displacement vs. Walk Lenght")
end
```

As can be seen in the plot above, the theoretically predicted power law is consistent with observed behavior.