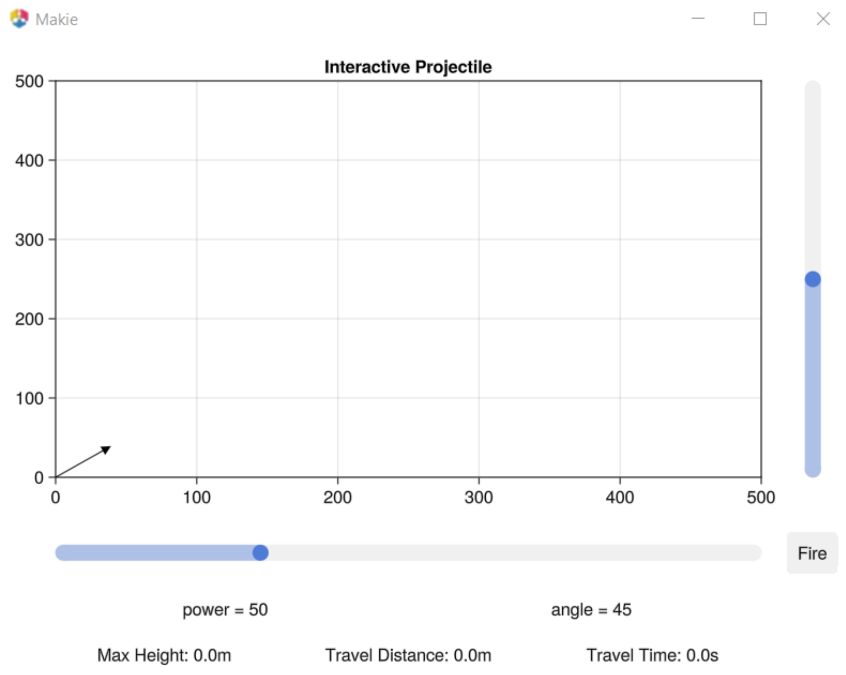
CSE 628
Term Project

Christopher K. Moore

12/11/2023

For the term project, I created an interactive 2D plot that simulates the trajectory of a projectile. The user can manipulate 2 sliders that set initial velocity and angle relative to the ground. After the sliders are set, the projectile's path is plotted onto the figure.

Figure 1. Initial view for manipulating sliders



power = 50

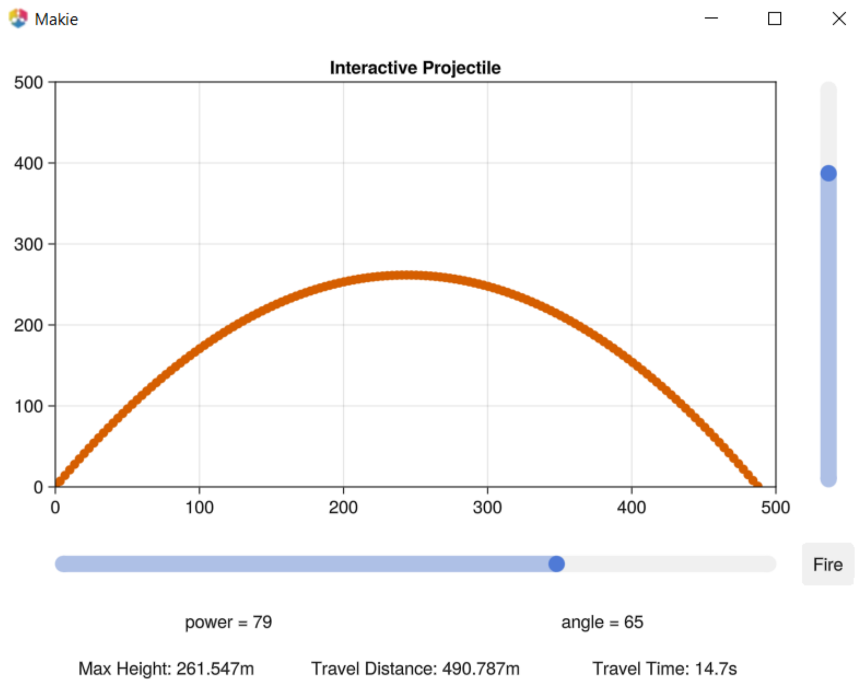
angle = 45

Max Height: 0.0m

Travel Distance: 0.0m

Travel Time: 0.0s

Figure 2. View after pressing the fire button



power = 79

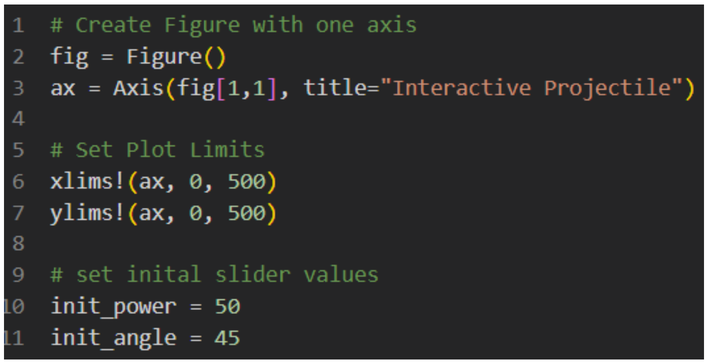
angle = 65

Max Height: 261.547m

Travel Distance: 490.787m

Travel Time: 14.7s

The first step is to create the figure that will house all the plot elements. The figure only needs one axis for the projectile.

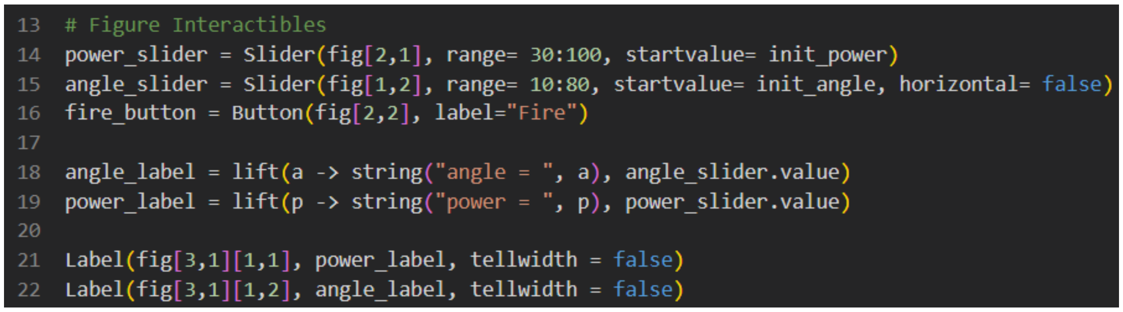


```julia
# Create Figure with one axis
fig = Figure()
ax = Axis(fig[1,1], title="Interactive Projectile")

# Set Plot Limits
xlims!(ax, 0, 500)
ylims!(ax, 0, 500)

# set inital slider values
init_power = 50
init_angle = 45
```

Next, the intractable elements are created. Two sliders, one for power and one for angle. The power slider will appear below the axis, and the angle slider will be next to the axis and be oriented vertically. Using observables, labels are created to display the current value of both sliders. A button is added to the corner of the figure to 'fire' the projectile with the given slider stats.

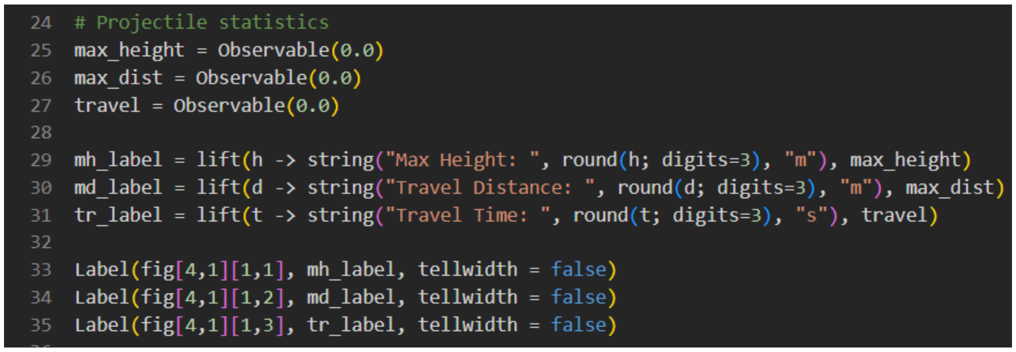


```julia
# Figure Interactibles
power_slider = Slider(fig[2,1], range= 30:100, startvalue= init_power)
angle_slider = Slider(fig[1,2], range= 10:80, startvalue= init_angle, horizontal= false)
fire_button = Button(fig[2,2], label="Fire")

angle_label = lift(a -> string("angle = ", a), angle_slider.value)
power_label = lift(p -> string("power = ", p), power_slider.value)

Label(fig[3,1][1,1], power_label, tellwidth = false)
Label(fig[3,1][1,2], angle_label, tellwidth = false)
```

Statistics about the trajectory of the projectile will need to be displayed to the user. Observables are created to store the maximum height the projectile reached, the horizontal distance the projectile traveled before hitting the *ground* (y = 0), and the time the projectile spent traveling through the air.

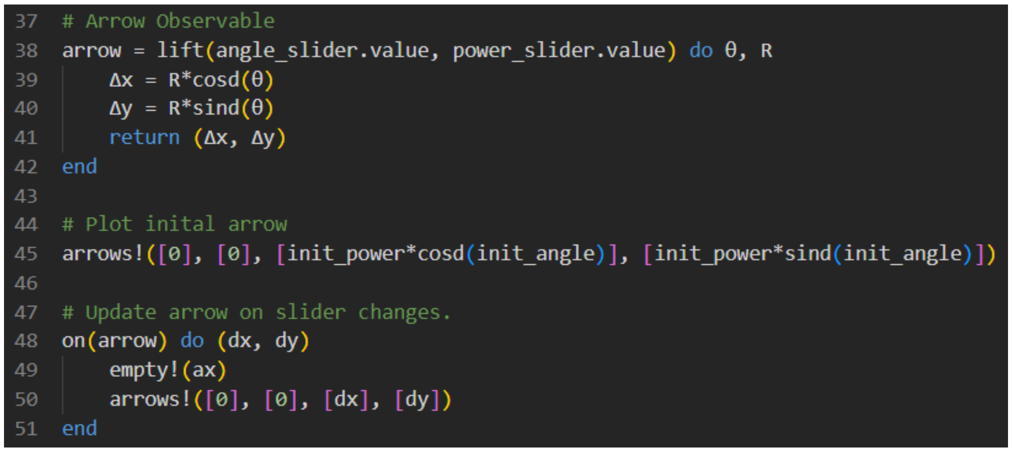


```julia
# Projectile statistics
max_height = Observable(0.0)
max_dist = Observable(0.0)
travel = Observable(0.0)

mh_label = lift(h -> string("Max Height: ", round(h; digits=3), "m"), max_height)
md_label = lift(d -> string("Travel Distance: ", round(d; digits=3), "m"), max_dist)
tr_label = lift(t -> string("Travel Time: ", round(t; digits=3), "s"), travel)

Label(fig[4,1][1,1], mh_label, tellwidth = false)
Label(fig[4,1][1,2], md_label, tellwidth = false)
Label(fig[4,1][1,3], tr_label, tellwidth = false)
```

To give a preview of the projectile's path to the user, an arrow will be plotted that represents the initial projectile vector. The arrow will have a magnitude equal to the power and an angle relative to the ground that is the same as the angle from the slider. A new observable, *arrow*, is created by lifting the slider variables and returning the x and y components of the vector. The Makie *arrows!()* function is used to plot the vector at the origin and is replotted when the sliders are changed.

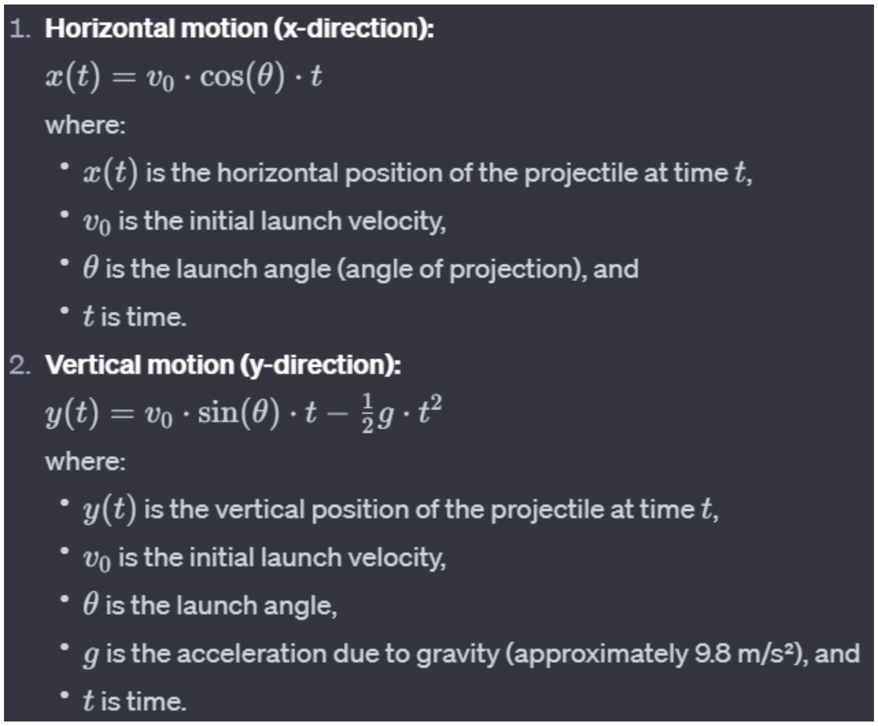


```julia
# Arrow Observable
arrow = lift(angle_slider.value, power_slider.value) do θ, R
    Δx = R*cosd(θ)
    Δy = R*sind(θ)
    return (Δx, Δy)
end

# Plot inital arrow
arrows!([0], [0], [init_power*cosd(init_angle)], [init_power*sind(init_angle)])

# Update arrow on slider changes.
on(arrow) do (dx, dy)
    empty!(ax)
    arrows!([0], [0], [dx], [dy])
end
```

To mimic a projectile's motion in 2 dimensions, we need to implement a transform of the kinematic equations that give x and y positions given an initial velocity, angle relative to the ground, and time. I asked chatGPT for the equations and it gave these results:



1. **Horizontal motion (x-direction):**

   $x(t) = v_0 \cdot \cos(\theta) \cdot t$

   where:

   - $x(t)$ is the horizontal position of the projectile at time t,
   - v_0 is the initial launch velocity,
   - θ is the launch angle (angle of projection), and
   - t is time.

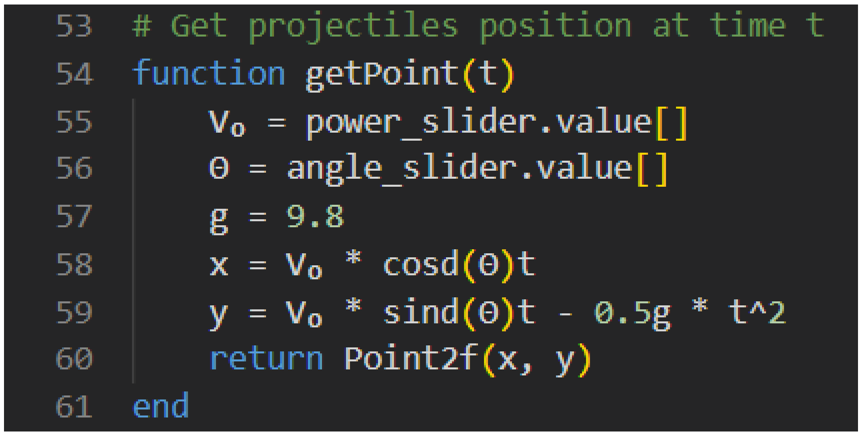
2. **Vertical motion (y-direction):**

   $y(t) = v_0 \cdot \sin(\theta) \cdot t - \frac{1}{2} g \cdot t^2$

   where:

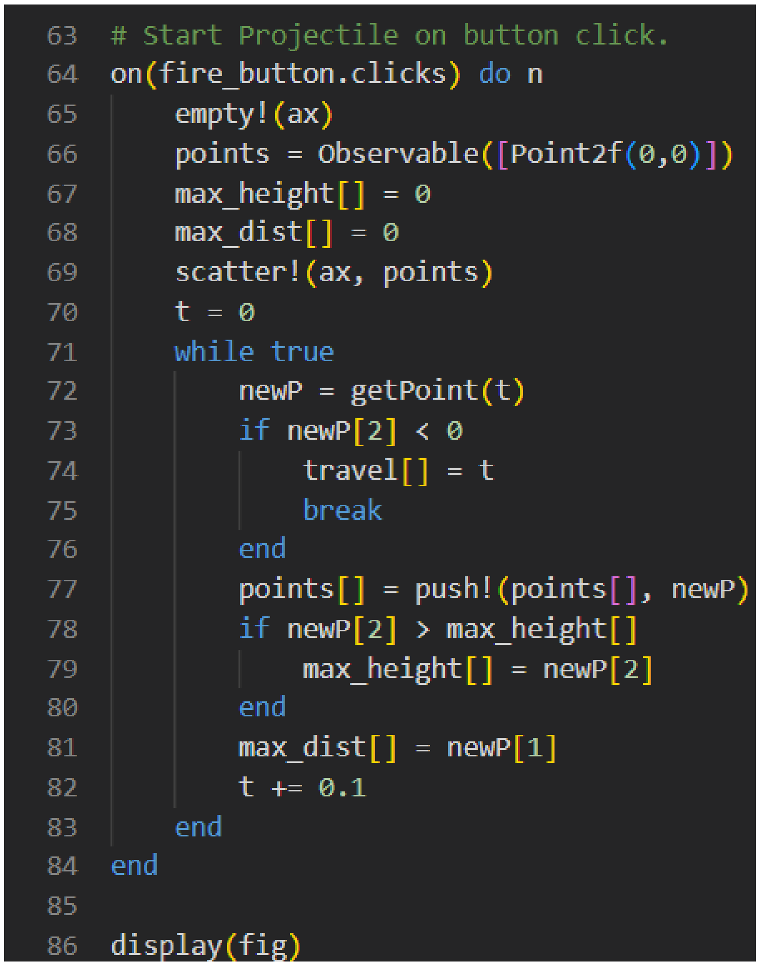
   - $y(t)$ is the vertical position of the projectile at time t,
   - v_0 is the initial launch velocity,
   - θ is the launch angle,
   - g is the acceleration due to gravity (approximately 9.8 m/s²), and
   - t is time.

These equations are implemented in the *getPoint*() function, that is used to get the x and y coordinates given a time *t*.



```julia
# Get projectiles position at time t
function getPoint(t)
    V₀ = power_slider.value[]
    θ = angle_slider.value[]
    g = 9.8
    x = V₀ * cosd(θ)t
    y = V₀ * sind(θ)t - 0.5g * t^2
    return Point2f(x, y)
end
```

The last step to implement is creating the projectile's path once the user clicks the fire button. Using the clicks observable tied to the button in the figure, we can start the projectile calculations whenever the button is pressed. The plot is first cleared to make room for the projectile motion. The points observable is created to hold the newly created points and is passed into a scatter plot. New points are created at every tenth of a second. The max distance is updated and the max height is checked every step. The simulation stops when a newpoint is created whose y value is less than 0 and the travel time is set.



```julia
# Start Projectile on button click.
on(fire_button.clicks) do n
    empty!(ax)
    points = Observable([Point2f(0,0)])
    max_height[] = 0
    max_dist[] = 0
    scatter!(ax, points)
    t = 0
    while true
        newP = getPoint(t)
        if newP[2] < 0
            travel[] = t
            break
        end
        points[] = push!(points[], newP)
        if newP[2] > max_height[]
            max_height[] = newP[2]
        end
        max_dist[] = newP[1]
        t += 0.1
    end
end

display(fig)
```

The figure is displayed to the user where they can test different initial velocities and angles. I initially planned for the path of the projectile to be updated in real time creating a smooth animation, but the figure doesn't update until the button observable is complete.