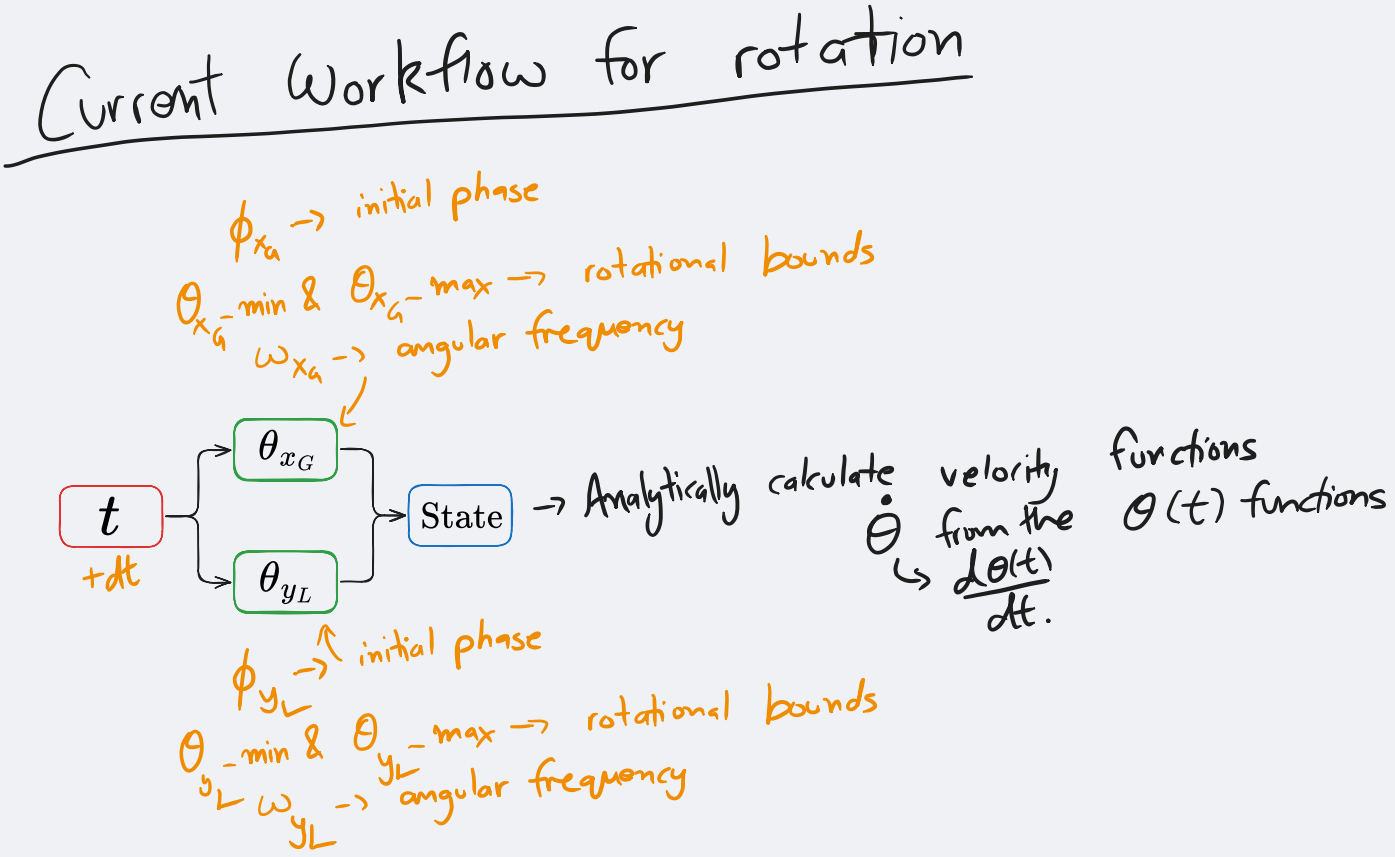
Finding the velocity functions and their transitions

The current version of the dragonfly code is such that I define a certain θ_{x_G} (the angle about the Global X axis) function in time t, as well as a certain θ_{y_L} (the angle about the Local Y axis for the current wing-state, i.e. after the rotation θ_{x_G}) and the with certain time parameter t, just change the location to state $S(\theta_{x_G}, \theta_{y_L})$. I then keep changing t with some value dt to move the animation forward.



Julia code for the current working

```julia
# Define observables
dt = Observable(0.01)

# Global rotation around the x-axis
phi_x_global     = Observable(deg2rad(0))
omega_x_global   = Observable(12)
theta_min_x_global = Observable(deg2rad(0))
theta_max_x_global = Observable(deg2rad(45))

# Local rotation around the y-axis
phi_y_local      = Observable(deg2rad(10))
omega_y_local    = Observable(12)
theta_min_y_local = Observable(deg2rad(-40))
theta_max_y_local = Observable(deg2rad(40))

# Environment variables
```

```
phase_between_front_and_hind = Observable(Float64(pi))          # Phase difference (constant for now)
sync_omegas = Observable(false)                    # Toggle for synchronization
pause     = Observable(false)                    # New toggle for pause


# Map -1,1 to theta_min, theta_max
theta_x_global(t) = map_range(cos(omega_x_global[] * t + phi_x_global[]), (-1, 1), (theta_min_x_global[], theta_max_x_global[]))
angular_velocity_x_global(t) = -omega_x_global[] * (theta_max_x_global[] - theta_min_x_global[]) / 2 * sin(omega_x_global[] * t + phi_x_global[]) # Analytical derivative of theta_x_global

theta_y_local(t) = map_range(sin(omega_y_local[] * t + phi_y_local[]), (-1, 1), (theta_min_y_local[], theta_max_y_local[]))
angular_velocity_y_local(t) = omega_y_local[] * (theta_max_y_local[] - theta_min_y_local[]) / 2 * cos(omega_y_local[] * t + phi_y_local[]) # Analytical derivative of theta_y_local

```

This is a good approach for defining the state of the wings for given bounds, but this does not give you the rotations themselves.

This method has several problems though...

- It does not really calculate the rotation from state 1 to state 2 (we do have the solution for this very easily though, as I will describe below)
- It does not incorporate sudden shifts in the rotational movements for the given position.

You COULD, if you want, use this approach and define the rotational quaternions to go from $State(t_0)$ to $State(t_0 + dt)$ and then apply that rotational operation to solve problem (1).

If state $State(t_0)$ is represented by quaternion q_{t_0} and state $State(t_0 + dt)$ (or any finite value Δt) is represented by quaternion q_{t_0+dt} then the rotational quaternion quaternion q^* defined such that $q^* \, q_{t_0} = q_{t_0+dt}$

> i.e. the rotational operation q^* applied to q_{t_0} gets you to state q_{t_0+dt}) would be given simply by $q_{t_0+dt} \, (q_{t_0})^{-1}$
> ($(q_{t_0})^{-1}$ being the $\text{inv}(q_{t_0})$ which for a unit quaternion is equivalent to $\text{conj}(q_{t_0})$)

It would be nice however to define a rotational scheme that incorporates the rotation rotations $\Delta\theta$ embedded within the working. Such a mechanism would take a rotational velocity function $\dot{\theta}(t)_{x_G}$ and iteratively, for tiny steps dt increment θ_{x_G} (by say Euler's method or Varlet Integration since that is energetically stable).

I think defining such a function would be much nicer. And this might help us change the States with better control over sudden changes in the movement speeds (Problem (2)).

The problem with that though is finding a velocity function that takes you from your current rotational state, to the next for the defined max and min parameters for the θ values while not jumping while transitions.

1. You must define an appropriate velocity function for the given parameter list $\{\phi_{x_G}, \max(\theta_{x_G}), \min(\theta_{x_G}), \omega_{x_G}, \phi_{y_L}, \max(\theta_{y_L}), \min(\theta_{y_L}), \omega_{y_L}\}$. This may be difficult but actually, my current working already does this. The analytically calculated velocity function $\dot{\theta}$ is precisely the function we need. In fact, a plot for

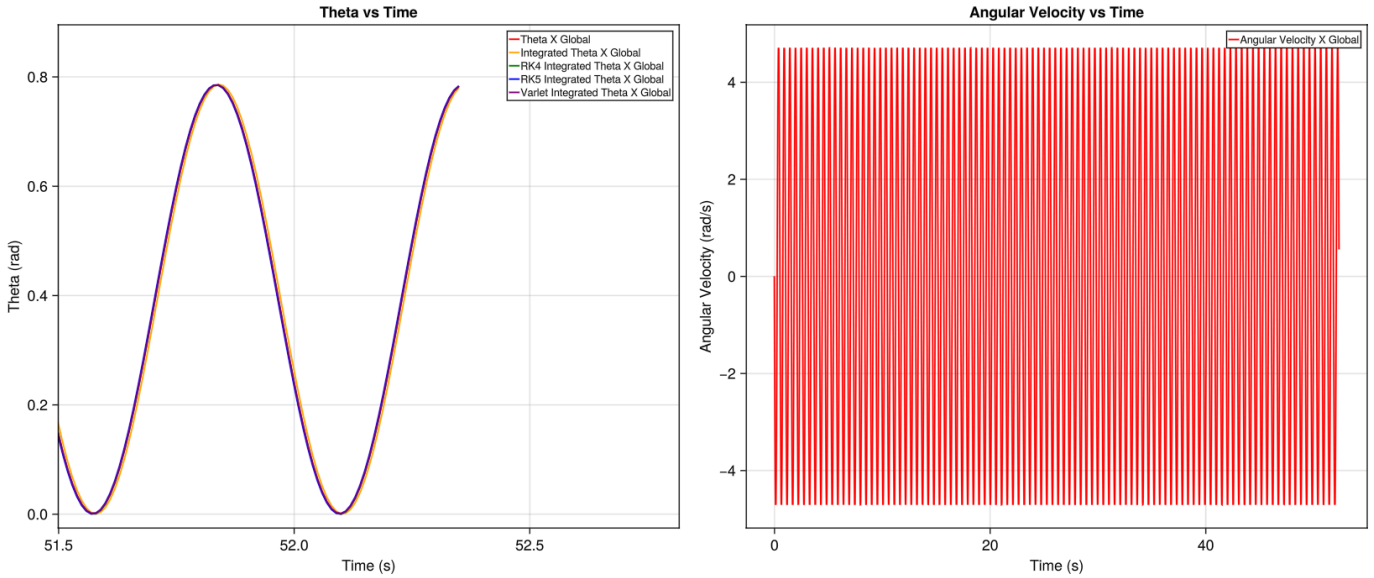
Julia Integrated Theta Values code

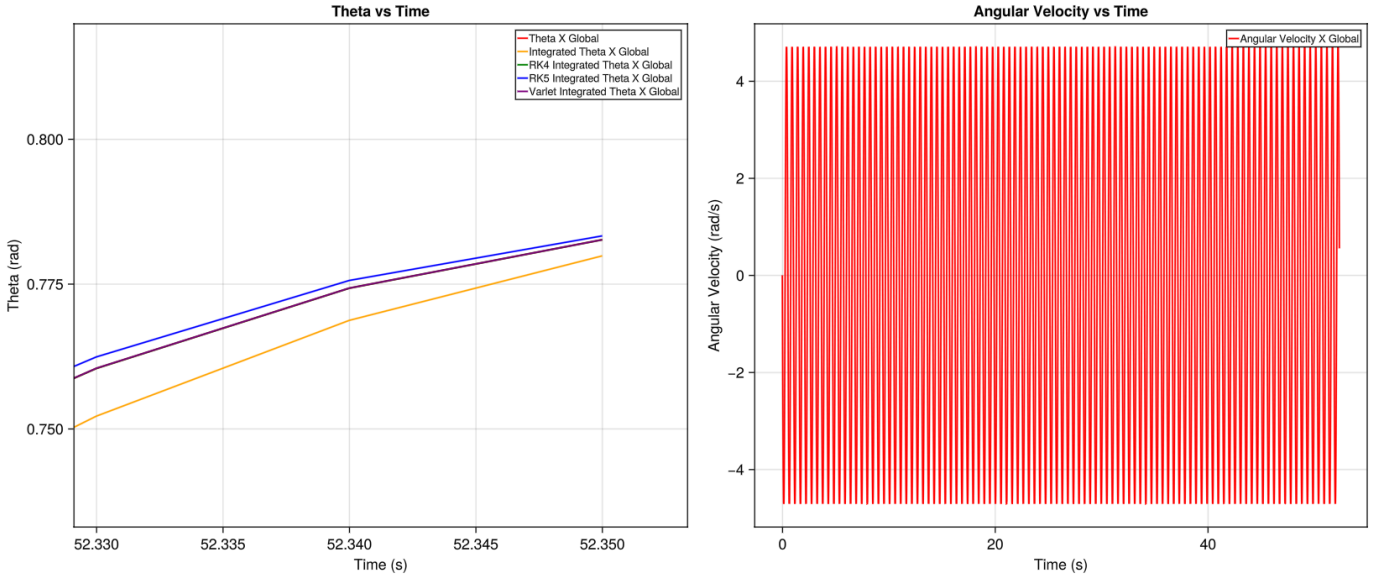
```julia
# Integrated theta value plots
#-----------------------------
# Plot theta, angular velocity and time integrated theta values for a given time range
times = 0:dt[]:100*2π/(lcm(omega_x_global[], omega_y_local[])) # Time vector for the simulation
thetas = theta_x_global.(times) # Calculate theta values for the global x-axis rotation
angular_velocities_x = angular_velocity_x_global.(times) # Calculate angular velocity values for the global x-axis rotation

# Integrating angular_velocity_x_global with time to get the integrated theta values

# Euler's method for numerical integration
integrated_thetas_x_euler = zeros(length(times))
integrated_thetas_x_euler[1] = theta_x_global(times[1]) # Initial value at t=0
for i in 2:length(times)
    # Using angular velocity at the current time point to predict the next value
    x_n_0 = integrated_thetas_x_euler[i-1] # Previous value
    v_n_0 = angular_velocity_x_global(times[i-1]) # Angular velocity at the previous time point
    integrated_thetas_x_euler[i] = x_n_0 + v_n_0 * dt[] # Euler's method update
end

# Calculate the maximum absolute difference to verify integration accuracy
max_diff = maximum(abs.(thetas - integrated_thetas_x_euler))
println("Maximum difference between analytical and integrated theta values: $max_diff")

# Runge-Kutta 4 method for numerical integration
integrated_thetas_x_rk4 = zeros(length(times))
```

```
integrated_thetas_x_rk4[1] = theta_x_global(times[1]) # Initial value at t=0
for i in 2:length(times)
    t_n = times[i-1] # Current time
    x_n_0 = integrated_thetas_x_rk4[i-1] # Previous value
    v_n_0 = angular_velocity_x_global(t_n) # Angular velocity at the previous time point

    k1 = v_n_0 * dt[] # First step
    k2 = angular_velocity_x_global(t_n + dt[]/2) * dt[] # Second step
    k3 = angular_velocity_x_global(t_n + dt[]/2) * dt[] # Third step
    k4 = angular_velocity_x_global(t_n + dt[]) * dt[] # Fourth step

    integrated_thetas_x_rk4[i] = x_n_0 + (k1 + 2*k2 + 2*k3 + k4) / 6 # RK4 update
end
# Calculate the maximum absolute difference to verify integration accuracy
max_diff_rk4 = maximum(abs.(thetas - integrated_thetas_x_rk4))
println("Maximum difference between analytical and RK4 integrated theta values: $max_diff_rk4")

# Runge-Kutta 5 method for numerical integration (Butcher's method)
integrated_thetas_x_rk5 = zeros(length(times))
integrated_thetas_x_rk5[1] = theta_x_global(times[1]) # Initial value at t=0
for i in 2:length(times)
    t_n = times[i-1] # Current time
    x_n = integrated_thetas_x_rk5[i-1] # Previous value
    h = dt[] # Time step

    # Standard RK5 coefficients
    k1 = angular_velocity_x_global(t_n) * h
    k2 = angular_velocity_x_global(t_n + h/4) * h
    k3 = angular_velocity_x_global(t_n + 3*h/8) * h
    k4 = angular_velocity_x_global(t_n + 12*h/13) * h
    k5 = angular_velocity_x_global(t_n + h) * h
    k6 = angular_velocity_x_global(t_n + h/2) * h

    # RK5 update using the correct weighted combination
    integrated_thetas_x_rk5[i] = x_n + (16*k1 + 25*k3 + 25*k4 + 25*k5 + 9*k6) / 100
end
# Calculate the maximum absolute difference to verify integration accuracy
max_diff_rk5 = maximum(abs.(thetas - integrated_thetas_x_rk5))
println("Maximum difference between analytical and RK5 integrated theta values: $max_diff_rk5")

# Varlet Integration method for numerical integration
integrated_thetas_x_varlet = zeros(length(times))
integrated_thetas_x_varlet[1] = theta_x_global(times[1]) # Initial value at t=0
```

```julia
for i in 2:length(times)
    t_n = times[i-1] # Current time
    x_n_0 = integrated_thetas_x_varlet[i-1] # Previous value
    v_n_0 = angular_velocity_x_global(t_n) # Angular velocity at the previous time point

    k1 = v_n_0 * dt[] # First step
    k2 = angular_velocity_x_global(t_n + dt[]) * dt[] # Second step

    integrated_thetas_x_varlet[i] = x_n_0 + (k1 + k2) / 2 # Varlet update
end
# Calculate the maximum absolute difference to verify integration accuracy
max_diff_varlet = maximum(abs.(thetas - integrated_thetas_x_varlet))
println("Maximum difference between analytical and Varlet integrated theta values: $max_diff_varlet")

# Print the maximum differences for all integration methods
println("Maximum differences:")
println("Euler: $max_diff")
println("RK4: $max_diff_rk4")
println("RK5: $max_diff_rk5")
println("Varlet: $max_diff_varlet")

# Create a figure for the plots

fig = Figure()
ax_thetas = Axis(fig[1, 1], title="Theta vs Time", xlabel="Time (s)", ylabel="Theta (rad)")
ax_angular_velocities = Axis(fig[1, 2], title="Angular Velocity vs Time", xlabel="Time (s)", ylabel="Angular Velocity (rad/s)")

# Plot theta and integrated theta values
lines!(ax_thetas, times, thetas, label="Theta X Global", color=:red)
lines!(ax_thetas, times, integrated_thetas_x_euler, label="Integrated Theta X Global", color=:orange)
lines!(ax_angular_velocities, times, angular_velocities_x, label="Angular Velocity X Global", color=:red)
lines!(ax_thetas, times, integrated_thetas_x_rk4, label="RK4 Integrated Theta X Global", color=:green)
lines!(ax_thetas, times, integrated_thetas_x_rk5, label="RK5 Integrated Theta X Global", color=:blue)
lines!(ax_thetas, times, integrated_thetas_x_varlet, label="Varlet Integrated Theta X Global", color=:purple)

# Add a legend to the plots
axislegend(ax_thetas, labelsize=10, patchsize=(10,4), padding=2, colgap=1, rowgap=1, patchlabelgap=1)
axislegend(ax_angular_velocities, labelsize=10, patchsize=(10,4), padding=2, colgap=1, rowgap=1, patchlabelgap=1)
```

```
107
```





After several iterations of the angular-velocity integrated theta calculations, here were the results:

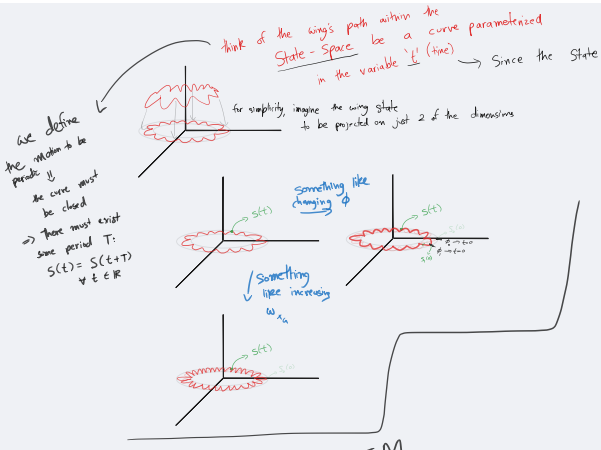
Title="calculation calculation results

```
Maximum differences:
Euler: 0.024038005961633413
RK4: 5.657288949698886e-8
RK5: 0.005827293023867364
Varlet: 0.000942703827175465
```

2. More importantly, you will have to get to the periodic function in a way that begins not from the initial conditions but the current conditions, say time T at which the velocity function is to be changed.
    1. A very crucial point made would be to sort of realize that there may not actually exist a periodic velocity function for the given position that makes the wings oscillate for the

given conditions.
2. In such cases, you would first have to interpolate the wing to the nearest location within the rotation space and curve for the periodically velocity-induced oscillations.

Diagrammatically:



→ Since the State Space is defined by an axis ($\mathbb{R}^3$) & an angle ($\mathbb{R}$), you need 4 Dimensions. (or simply because a quaternion q has 4 independent parameters)

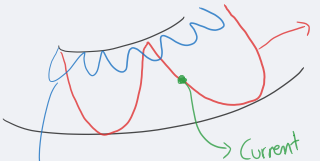
actually, since we use unit quaternions this makes the quaternion to be parameterizable in only 3 variables.

the axis can be uniquely defined by θ, ϕ & R is const = 1 & let the angle of rotation be ω around the Axis (θ, ϕ)

there could be other set of parameterizations such as any of the 24 conversions of Euler angles.

Again though, the mathematics and interpolation b/w states is most convenient (and elegant) with Quaternions)

THE PROBLEM



State path 1 → defined as $\int_0^t \dot{\theta}_1(t') \cdot dt'$

State path 2.
→ Defined for the new set of parameters at the current time
↳ defined as $\int_0^t \dot{\theta}_2(dt') \cdot dt'$

what should be you next position $(\theta + \dot{\theta}\, dt)$

↳ it's a tricky question since the current position → $S_1(t)$ is not even sitting on the curve → $S_2(t')$ so you can not simply say $\theta = \theta_0 + \dot{\theta}_2 dt$

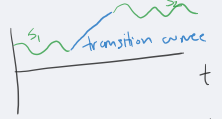
↓

A very simple solution is that you should approach first the closest point on the curve $S_2(t')$ for the point $S_1(t)$ in the State-Space. This can be done by sort of projecting the point on the curve.

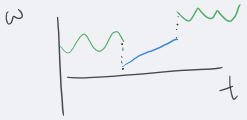
Look up LAGRANGE MULTIPLIERS

A more interesting question is HOW should we approach this point? we certainly wouldn't jump to it. So what would be the $\dot{\theta}$ function (the velocity function) that we should use to transition from curve S_1 to S_2, given our location $S_1(t)$? & How would the transition of the velocities look like?

Hypothetical Design (Base: Must Be improved)



problem with that: velocity function $\dot{\theta}$ may be discontinuous



I am fairly certain that the answer to the problem would be finding the Bézier curve from $S_1(t)$ to $S_2(t)$ for whatever degree you want to conserve the continuity for... for instance a 1st order (linear) Bézier curve would conserve the continuity of just the angles but not the velocity and/or the acceleration. A Quadratic Bézier curve would conserve both angles, and angular velocities.
A **Cubic-Bezier Curve** is generally considered a nice trade off between quality and computational efficiency, which would conserve the continuity of angles, angular velocities and angular acceleration (in principle however, you could compute the curve on which to travel along time, that conserves any higher order derivative)

> Resources on someone who would want to work on that:
> https://en.wikipedia.org/wiki/B%C3%A9zier_curve
> https://youtu.be/jvPPXbo87ds?si=LKlLyS1VTrCTl7wN
> https://youtu.be/aVwxzDHniEw?si=sTrTnfgAMmZQwdkc
