1.

Case 1:

$V_i = [5.46, 3.466, 0]\ \frac{km}{s}$

$V_f = [-4.71, -0.74, 0]\ \frac{km}{s}$

$e = 0.849$

$r_p = 1043\ km$

Case 2:

$V_i = [-2.134, 7.024, -2.987]\ \frac{km}{s}$

$V_f = [4.79, 0.81, 6.71]\ \frac{km}{s}$

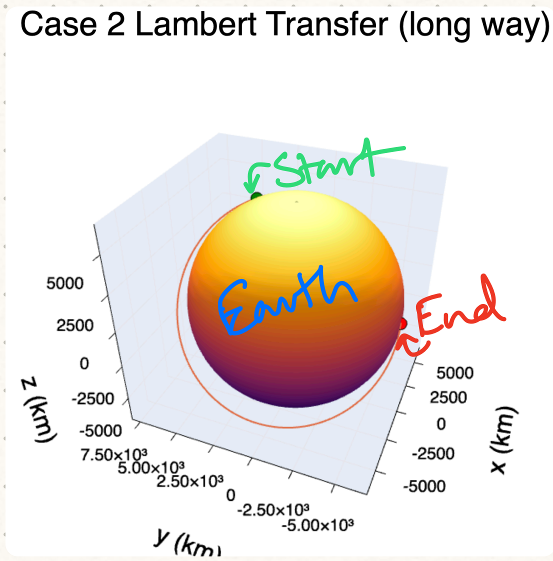
$e = 0.138$

$r_p = 6096\ km$

2. a) $\Delta r = [5.245E-5, 1.507E-5, 7.343E-5]\ km$

b) $\Delta V = [2.33E-9, 1.078E-7, 3.263E-9]\ \frac{km}{s}$

c)



3. $\Delta V_1 = 6.535\ \frac{km}{s}$

$\Delta V_2 = 5.236\ \frac{km}{s}$

$\Delta V = 11.77\ \frac{km}{s}$

4. $v_{est}\ @\ t_2 = [0.738, 2.929, -8.046]\ \frac{km}{s}$

To verify our answer, we can use our position & velocity at t_2 to find the orbital elements, which should give a physical intuition about the orbit itself, hopefully verifying our velocity. As well, we can use our 2BP propogator to analytically check the velocity.

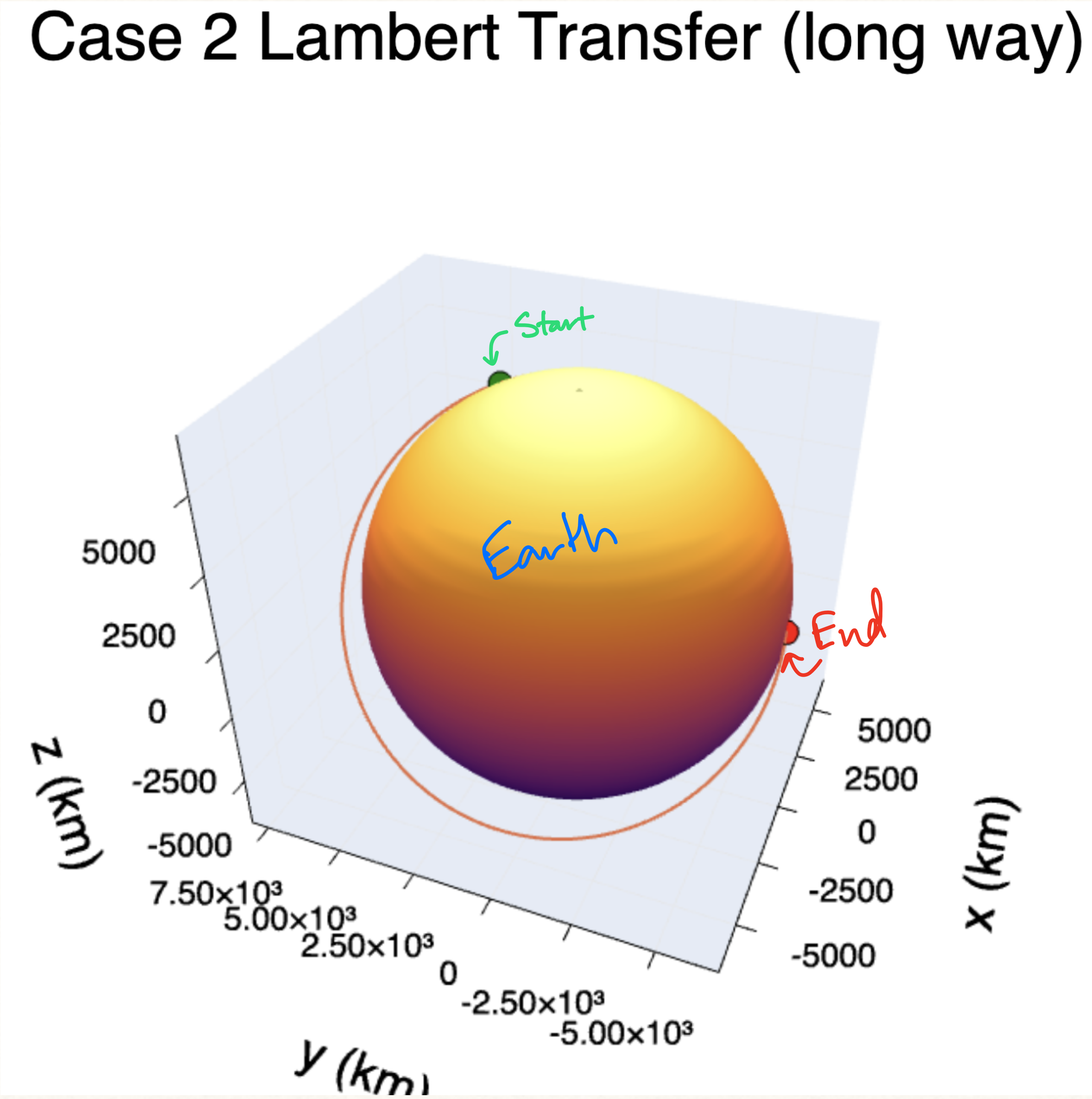
```julia
include("../../../code/sfd.jl")
using .SpaceFlightDynamics

# Case 1: short-way
r1_c1 = [8000.0, 0.0, 0.0]
r2_c1 = [7000.0, 7000.0, 0.0]
TOF_c1 = 3600.0

v1_c1, v2_c1, e_c1, rp_c1 = solve_lambert(r1_c1, r2_c1, TOF_c1; long_way=false)

println("Case 1 (short way):")
println("  v_1 = ", v1_c1)
println("  v_2 = ", v2_c1)
println("  e  = ", e_c1)
println("  r_p = ", rp_c1, " km\n")

# Case 2: long-way, using Earth radius
r1_c2 = [0.5, 0.6, 0.7] .* R_Earth
r2_c2 = [0.0, -1.0, 0.0] .* R_Earth
TOF_c2 = 16135.0

v1_c2, v2_c2, e_c2, rp_c2 = solve_lambert(r1_c2, r2_c2, TOF_c2; long_way=true)

println("Case 2 (long way):")
println("  v_1 = ", v1_c2)
println("  v_2 = ", v2_c2)
println("  e  = ", e_c2)
println("  r_p = ", rp_c2, " km")
```

```
Case 1 (short way):
  v_1 = [5.459317364023696, 3.466008449141647, 0.0]
  v_2 = [-4.705584118347704, -0.7444316050429653, -0.0]
  e  = 0.8486118938193392
  r_p = 1043.4116692745763 km

Case 2 (long way):
  v_1 = [-2.133759073847983, 7.024037548362913, -2.9872627033871755]
  v_2 = [4.792274218490246, 0.8071074997180263, 6.709183905886343]
  e  = 0.13798287545743462
  r_p = 6096.619935743475 km
```

```julia
include("../../../code/sfd.jl")
using .SpaceFlightDynamics
using Plots
plotlyjs()

# Case 1: short-way
r1_c1 = [8000.0, 0.0, 0.0]
r2_c1 = [7000.0, 7000.0, 0.0]
TOF_c1 = 3600.0

v1_c1, v2_c1, e_c1, rp_c1 = solve_lambert(r1_c1, r2_c1, TOF_c1; long_way=false)
sv_c1 = solve_2BP(StateVectors(r1_c1, v1_c1), (0.0, TOF_c1), μ=μ_Earth, int_pts = 500)

r2_c1_diff = r2_c1 - sv_c1[end].r
v2_c1_diff = v2_c1 - sv_c1[end].v

println("Case 1 Final Position Vector Diff: ", r2_c1_diff)
println("Case 1 Final Velocity Vector Diff: ", v2_c1_diff)

# Case 2: long-way, using Earth radius
r1_c2 = [0.5, 0.6, 0.7] .* R_Earth
r2_c2 = [0.0, -1.0, 0.0] .* R_Earth
TOF_c2 = 16135.0

v1_c2, v2_c2, e_c2, rp_c2 = solve_lambert(r1_c2, r2_c2, TOF_c2; long_way=true)
sv_c2 = solve_2BP(StateVectors(r1_c2, v1_c2), (0.0, TOF_c2), μ=μ_Earth, int_pts = 500)

r2_c2_diff = r2_c2 - sv_c2[end].r
v2_c2_diff = v2_c2 - sv_c2[end].v

println("Case 2 Final Position Vector Diff: ", r2_c2_diff)
println("Case 2 Final Velocity Vector Diff: ", v2_c2_diff)

xs = [sv.r[1] for sv in sv_c2]
ys = [sv.r[2] for sv in sv_c2]
zs = [sv.r[3] for sv in sv_c2]

θ = range(0,2π,length=60)
φ = range(0,π,length=30)
x_s = [R_Earth*sin(ϕ)*cos(θi) for ϕ in φ, θi in θ]
y_s = [R_Earth*sin(ϕ)*sin(θi) for ϕ in φ, θi in θ]
z_s = [R_Earth*cos(ϕ)          for ϕ in φ, θi in θ]

plt = plot(
    surface(x_s, y_s, z_s; opacity=0.3, legend=false),
    xlabel="x (km)", ylabel="y (km)", zlabel="z (km)",
    title="Case 2 Lambert Transfer (long way)",
)

plot!(plt, xs, ys, zs; lw=2, label="Transfer arc")
scatter!(plt, [r1_c2[1]], [r1_c2[2]], [r1_c2[3]]; markersize=2, markercolor=:green,
label="Start")
scatter!(plt, [r2_c2[1]], [r2_c2[2]], [r2_c2[3]]; markersize=2, markercolor=:red,
label="End")

display(plt)
```

```
Case 1 Final Position Vector Diff: [1.3030003174208105e-6, 3.03133674606215
2e-7, 0.0]
Case 1 Final Velocity Vector Diff: [5.828511007166526e-10, 2.79416934034770
```

```
75e-10, -0.0]
Case 2 Final Position Vector Diff: [5.245066911882979e-5, 1.507268734712852
2e-5, 7.343093666673104e-5]
Case 2 Final Velocity Vector Diff: [2.3306760965624562e-9, 1.07794643988690
1e-7, 3.262948133908594e-9]
```



```julia
include("../../../code/sfd.jl")
using .SpaceFlightDynamics
using LinearAlgebra

r1 = [8000.0, 0.0, 0.0]
r2 = [7000.0, 7000.0, 0.0]
TOF = 3600.0

v1, v2, e, rp = solve_lambert(r1, r2, TOF; long_way=false)

r1_norm = norm(r1)
v_circ1 = [ 0.0,
            sqrt(μ_Earth / r1_norm),
            0.0 ]

r2_norm = norm(r2)

t_hat2 = [-r2[2], r2[1], 0.0] ./ r2_norm
v_circ2 = sqrt(μ_Earth / r2_norm) .* t_hat2

ΔV1 = norm(v1 .- v_circ1)
ΔV2 = norm(v2 .- v_circ2)
ΔV_total = ΔV1 + ΔV2

println("ΔV at departure (km/s): ", ΔV1)
println("ΔV at arrival   (km/s): ", ΔV2)
println("Total ΔV         (km/s): ", ΔV_total)
```

```
ΔV at departure (km/s): 6.535402184960239
ΔV at arrival   (km/s): 5.235910109612791
Total ΔV         (km/s): 11.771312294573029
```

```julia
include("../../../code/sfd.jl")
using .SpaceFlightDynamics

r1 = [-6_979.49190,   208.846535,  -573.801140]
r2 = [-6_966.33930,   267.474976,  -734.881458]
r3 = [-6_949.96267,   325.979638,  -895.621695]

v2 = solve_gibbs(r1, r2, r3)
println("Estimated velocity at t_2 (km/s): ", v2)
```

```
Estimated velocity at t_2 (km/s): [0.7383384040052391, 2.928550616067919, -8
.046126675284103]
```

```julia
module SpaceFlightDynamics
        using LinearAlgebra
        using DifferentialEquations

        μ_Earth = 398600.4418 # km^3/s_2
        μ_Sun = 1.32712e11 # km^3/s_2
        R_Earth = 6378.1363 # km

        include("./oe_sv.jl")
        include("./2BP.jl")
        include("./kepler.jl")
        include("./lambert.jl")
        include("./gibbs.jl")

        export μ_Earth, μ_Sun, R_Earth
end
```

```
Main.var"##WeaveSandBox#247".SpaceFlightDynamics
```

```julia
stumpff_C2(z::Float64) = z > 0 ? (1 - cos(sqrt(z)))/z : z < 0 ? (cosh(sqrt(-z)) -
1)/(-z) : 1/2
stumpff_C3(z::Float64) = z > 0 ? (sqrt(z) - sin(sqrt(z))) / (z*sqrt(z)) : z < 0 ?
(sinh(sqrt(-z)) - sqrt(-z)) / ((-z)*sqrt(-z)) : 1/6

function solve_lambert(
        r1::Vector{Float64},
        r2::Vector{Float64},
        TOF::Float64;
        μ::Float64 = μ_Earth,
        long_way::Bool = false
)
        # Magnitudes
        r1_norm = norm(r1)
        r2_norm = norm(r2)
        # Transfer angle Δθ
        cos_dθ = dot(r1, r2) / (r1_norm * r2_norm)
        Δθ = acos(clamp(cos_dθ, -1.0, 1.0))
        if long_way
                Δθ = Δθ < π ? 2π - Δθ : Δθ
        else
                Δθ = Δθ > π ? 2π - Δθ : Δθ
        end
        # A-parameter
        A = sin(Δθ) * sqrt(r1_norm * r2_norm / (1 - cos(Δθ)))
        if iszero(A)
                error("Cannot compute Lambert solution: A = 0")
        end

        # Time-of-flight function F(z) = 0
        function F(z)
                C2 = stumpff_C2(z)
                C3 = stumpff_C3(z)
                y  = r1_norm + r2_norm + A * (z*C3 - 1) / sqrt(C2)
                if y < 0
                        return Inf
                end
                return ( (y/C2)^(3/2) * C3 + A*sqrt(y) ) / sqrt(μ) - TOF
        end

        # Solve for z via Newton-Raphson with finite-difference derivative
        z = 0.0
        for _ in 1:200
                Fz = F(z)
                if abs(Fz) < 1e-8
                        break
                end
                δ   = 1e-6
                dF  = (F(z + δ) - F(z - δ)) / (2δ)
                z  -= Fz / dF
        end

        # Compute y, f, g, g
        C2 = stumpff_C2(z)
        C3 = stumpff_C3(z)
        y  = r1_norm + r2_norm + A * (z*C3 - 1) / sqrt(C2)

        f    = 1 - y/r1_norm
        g    = A * sqrt(y/μ)
```

```
        gdot = 1 - y/r2_norm

        # Velocity vectors
        v1 = (r2 .- f*r1) ./ g
        v2 = (gdot*r2 .- r1) ./ g

        # Compute eccentricity and periapsis radius from (r1, v1)
        h_vec = cross(r1, v1)
        e_vec = (1/μ) * ((norm(v1)^2 - μ/r1_norm)*r1 .- dot(r1,v1)*v1)
        e     = norm(e_vec)
        # Semi-major axis from energy
        energy = norm(v1)^2/2 - μ/r1_norm
        a      = -μ / (2*energy)
        rp     = a * (1 - e)

        return v1, v2, e, rp
end

export solve_lambert
```

```
function two_body!(du, u, μ, t)
        # u = [ x, y, z, vx, vy, vz ]
        # du[1:3] = v
        # du[4:6] = acceleration
        @views du[1:3] .= u[4:6]
        r = @view u[1:3]
        r_norm = norm(r)
        @views du[4:6] .= -μ .* r ./ (r_norm^3)
end

function solve_2BP(initial::StateVectors,
                   tspan::Tuple{Float64, Float64};
                   μ::Float64 = μ_Earth,
                   reltol::Float64 = 1e-9,
                   abstol::Float64 = 1e-9,
                   int_pts::Int64 = 2)

        # Pack initial conditions into a 6-vector
        u0 = vcat(initial.r, initial.v)

        # Set up and solve the ODE problem
        prob = ODEProblem(two_body!, u0, tspan, μ)
        sol  = solve(prob, Tsit5(), reltol=reltol, abstol=abstol,
saveat=range(start=tspan[1], stop=tspan[2], length=int_pts))

        return [StateVectors(u[1:3], u[4:6]) for u in sol.u]
end

export solve_2BP
```

```
Error: UndefVarError: `StateVectors` not defined in `Main.var"##WeaveSandBo
x#239"`
Suggestion: check for spelling errors or missing imports.
```

```julia
function solve_gibbs(
        r1::Vector{Float64},
        r2::Vector{Float64},
        r3::Vector{Float64};
        μ::Float64 = μ_Earth
)
        # cross-products
        c12 = cross(r1, r2)
        c23 = cross(r2, r3)
        c31 = cross(r3, r1)

        # N and D vectors
        N = c12*norm(r3) + c23*norm(r1) + c31*norm(r2)
        D = c12 + c23 + c31

        # S vector
        S =  r1*(norm(r2)-norm(r3)) +
             r2*(norm(r3)-norm(r1)) +
             r3*(norm(r1)-norm(r2))

        # scalar prefactor
        factor = sqrt( μ / (norm(N)*norm(D)) )

        # Gibbs velocity at r2
        v2 = factor * ( cross(D, r2)/norm(r2) + S )

        return v2
end

export solve_gibbs
```