```
Main.workspace2.OptimalSpacecraftTrajectories
```

```julia
begin
    using Plots ✓ , PlutoUI ✓ , LinearAlgebra ✓ , Symbolics ✓ ,
DifferentialEquations ✓ , DataFrames ✓ , CSV ✓
    include("/home/burtonyale/Documents/repos
/OptimalSpacecraftTrajectories/src/OptimalSpacecraftTrajectories.jl")
    import .OptimalSpacecraftTrajectories
    const OST = OptimalSpacecraftTrajectories
end
```

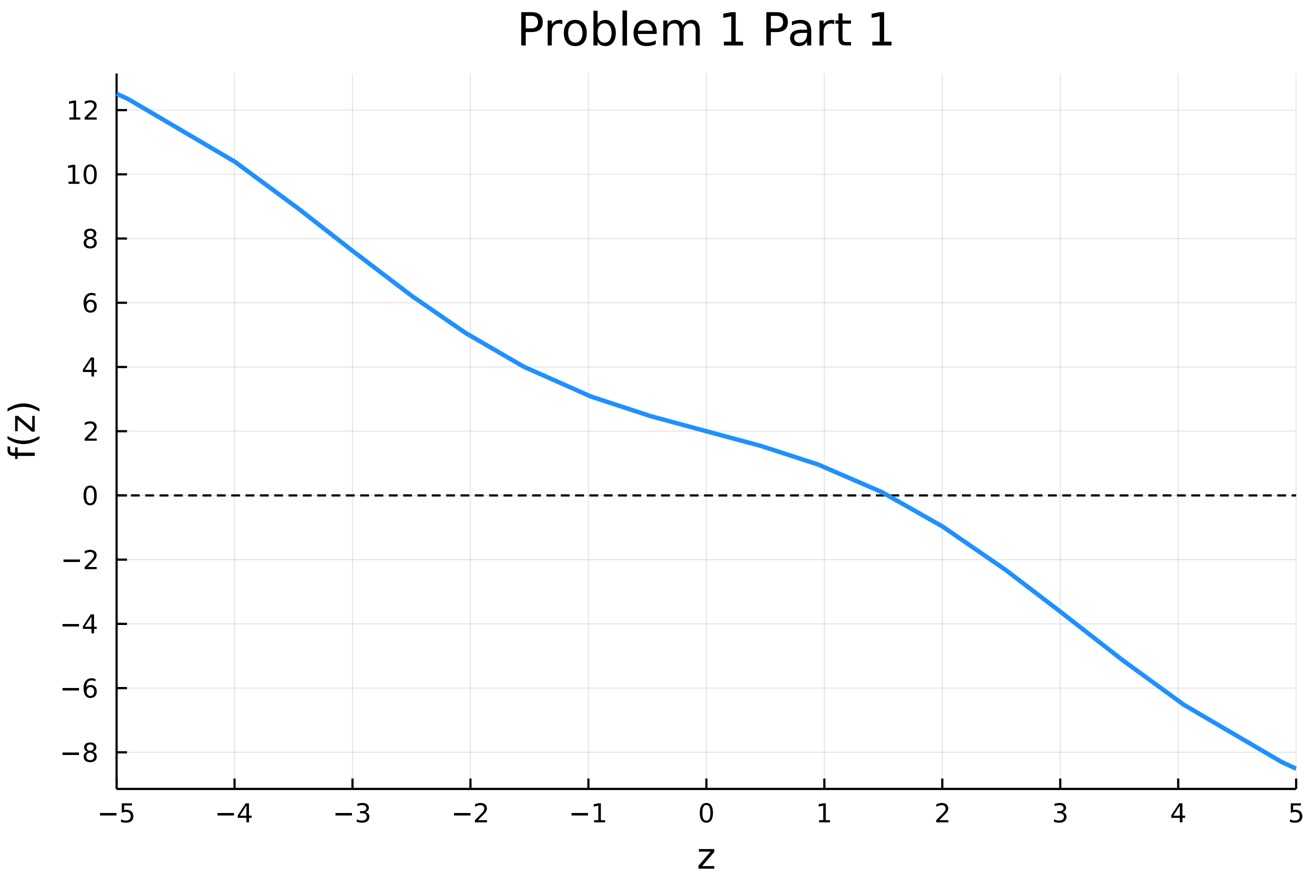
Problem 1

Part 1

```julia
begin
    # ORIGINAL FUNCTION
    F = (x⃗, z) -> 2*z*sin(x⃗[1]*x⃗[3] + x⃗[2]) + sin(z)*cos(x⃗[1] + x⃗[2]
x⃗[3]) + 2

    # SETTING X VAL
    x⃗₀ = [1.0, 2.0, 3.0]

    # X VAL SET
    Fz(z) = F(x⃗₀, z)
    md"## Part 1"
end
```




```
begin
    plot([-5, 5], [0, 0], linestyle=:dash, color=:black, label="", xlim=(-5, 5), fmt=:png)
    plot!(Fz, -5, 5, color=:dodgerblue, lw=2, label="", dpi=200, xticks=-5:5, xlabel="z", yticks=-10:2:14, ylabel="f(z)", title="Prob
Part 1")
    # png("hw2p1.1.png")
end
```

Part 2

f(z) = 0 @ z = 1.543295599106779

Using Roots.jl package

```
begin
    # δ, z = root_solve(ƒ_z, 0.0, 3, 1e-16)
    # plot([-5, 5], [0, 0], linestyle=:dash, color=:black, label="",
    #   title="Problem 1 Part 2: 3rd Order Recursive")
    # plot!(ƒ_z, -5, 5, color=:dodgerblue, lw=2,
    #   label="f(z) = 0 @ z = $(round(z[end], digits=6))", dpi=200,
    #   xticks=-5:5, xlabel="z", yticks=-10:2:14, ylabel="f(z)")
    # scatter!([z[1]], [ƒ_z(z[1])], marker=:star, color=:red,
label="Initial Guess")
    # scatter!(z[2:end], ƒ_z.(z[2:end]), color=:red, label="Number c
Iterations = $(length(δ))")
    using Roots
    z_actual = find_zero(Fz, 0.0)
    md"""
    ### f(z) = 0 @ z = $(z_actual)

    Using Roots.jl package
    """
end
```

Part 3

root_solve_recursive (generic function with 2 methods)

```
function root_solve_recursive(f, x*, order, minerror = 1e-3)
    # GENERATING SAVED VARIABLES
    Δz = [Inf]
    x_out = [x*]

    # ROOT SOLVING
    # while abs(Δz[end]) > minerror
        # FINDING DERIVATIVES
        f₀ = f(x*)
        f′ = OST.cntrDiff[5][1](f, x*, 1e-5)
        f″ = OST.cntrDiff[5][2](f, x*, 1e-3)
        f‴ = OST.cntrDiff[5][3](f, x*, 1e-3)

        # GENERATING TAYLOR SERIES
        δz = -f₀/f′
        δ²z = (-f″*f₀^2)/f′^3
        δ³z = (f₀^3 / f′^5)*(f′*f‴ - 3*f″^2)
        if order == 1
            Δz₀ = δz
        elseif order == 2
            Δz₀ = δz + 0.5*δ²z
        else
            Δz₀ = δz + 0.5*δ²z + 1/factorial(3) * δ³z
        end
        push!(Δz, Δz₀)
        x* += Δz₀
        push!(x_out, x*)
    # end
    deleteat!(Δz, 1)
    return Δz₀
end
```

root_solve_quad (generic function with 1 method)

```
function root_solve_quad(f, x⋆)
    # FINDING DERIVATIVES
    f₀ = f(x⋆)
    f′ = OST.cntrDiff[5][1](f, x⋆, 1e-5)
    f″ = OST.cntrDiff[5][2](f, x⋆, 1e-3)
    f‴ = OST.cntrDiff[5][3](f, x⋆, 1e-3)

    if f′^2 - 2*f″*f₀ < 0; return 0; end

    Δx = [(-f′ - sqrt(f′^2 - 2*f″*f₀))/f″, (-f′ + sqrt(f′^2 -
2*f″*f₀))/f″]

    if abs(f(x⋆ + Δx[1])) < abs(f₀)
        return Δx[1]
    else
        return Δx[2]
    end

end
```

root_solve_halley (generic function with 1 method)

```
function root_solve_halley(f, x⋆)
    # FINDING DERIVATIVES
    f₀ = f(x⋆)
    f′ = OST.cntrDiff[5][1](f, x⋆, 1e-5)
    f″ = OST.cntrDiff[5][2](f, x⋆, 1e-3)
    f‴ = OST.cntrDiff[5][3](f, x⋆, 1e-3)

    Δx = (2*f₀*f′)/(f″*f₀ - 2*f′^2)

end
```

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

```
function root_solve_laguerre(f, x*, n)
    # FINDING DERIVATIVES
    f₀ = f(x*)
    f′ = OST.cntrDiff[5][1](f, x*, 1e-5)
    f″ = OST.cntrDiff[5][2](f, x*, 1e-3)
    f‴ = OST.cntrDiff[5][3](f, x*, 1e-3)

    D₁ = f′/f₀
    D₂ = f″/f₀ - D₁^2


    if (1-n) * (D₁^2 + D₂*n) < 0; return 0; end

    q = [-n/(D₁ - sqrt( (1-n) * (D₁^2 + D₂*n) )),
         -n/(D₁ + sqrt( (1-n) * (D₁^2 + D₂*n) ))]
    if abs(q[1]) < abs(q[2])
        return q[1]
    else
        return q[2]
    end
end
```

a) First Order Recursive Error: 0.00048566623763979244

b) Second Order Recursive Error: -1.0526718867920337e-5

c) Third Order Recursive Error: 2.0692240321196 21e-7

d) Quadratic Error: 4.1154893160033623e-7

e) Halley Error: -4.966159171448936e-6

f) Laguerre $n = 1$ Error: 0.00048566623763979244

g) Laguerre $n = 2$ Error: 4.1154893160033623e-7

h) Laguerre $n = 3$ Error: -9.107433529553788e-7

Part 4

converge_root_solve (generic function with 1 method)

```
function converge_root_solve(f, x⋆, solver; output_type=:x,
min_convergence=1e-16)
    # SETUP
    Δx = []
    x = []

    # FIRST ITERATION
    push!(Δx, solver(f, x⋆))
    x⋆ += Δx[end]
    push!(x, x⋆)

    # SECOND ITERATION
    iters = 1
    # while (abs(x⋆ - z_actual) > min_convergence)
    while abs(Δx[end]) > min_convergence
        δx = solver(f, x⋆)
        if abs(real(δx)) < min_convergence || isnan(δx); break; end
        # if ((δx < min_convergence) & (Δx[end] < min_convergence))
isnan(δx); break; end
        push!(Δx, δx)
        x⋆ += Δx[end]
        push!(x, x⋆)
        iters += 1
        # if iters > 30
        #   break
        # end
    end

    while length(x) < 10
        push!(x, 0)
    end

    if output_type === :x
        return x⋆
    elseif output_type === :Δxx
        return Δx, x, x .- z_actual
    end
    # return x, Δx, x⋆
end
```

"hw2p1.4_0.csv"

```
begin
    solvers = [root_solve_halley,
        root_solve_quad,
        [laguerre(f, z) = root_solve_laguerre(f, z, n) for n = 1:3].
        [recursion(f, z) = root_solve_recursive(f, z, n) for n = 1:3
    outputs = []
    df = []
    for z₀ = [1.5, 0.0], i = 1:length(solvers)
        push!(outputs, converge_root_solve(Fz, z₀, solvers[i],
output_type=:Δxx, min_convergence=1e-16))
        push!(df, DataFrame(δz = outputs[end][2], z = outputs[end][2
err = outputs[end][3]))
        CSV.write("hw2_$i$(string(solvers[i]))$z₀.csv", df[end])
    end
    df
    CSV.write("hw2p1.4_0.csv", DataFrame(halley = df[9][!, 3], quad
df[10][!, 3], lag1 = df[11][!, 3], lag2 = df[12][!, 3], lag3 = df[13
3], rec1 = df[14][!, 3], rec2 = df[15][!, 3], rec3 = df[16][!, 3]))
end
```

Problem 2

$$P = \cos\left(-x_1 x_2 x_3 + z^2\right) \qquad \mathbf{x} = [1,\, 2,\, 3]^T \qquad z_0 = 0$$

Part 1

cplxDiff (generic function with 1 method)

```
function cplxDiff(f, x₀, h)
    f′ = []
    L = length(x₀)
    for i = 1:L
        H = zeros(Complex{Float64}, L)
        H[i] = h*im
        push!(f′, imag(f(x₀ + H))/h)
    end
    return f′
end
```