Mass-Spring Underdamped Oscillator

$$m\frac{d^2x}{dt^2} + c\frac{dx}{dt} + kx = 0$$

$$x(0) = 1$$

$$\frac{dx}{dt}(0) = 0$$

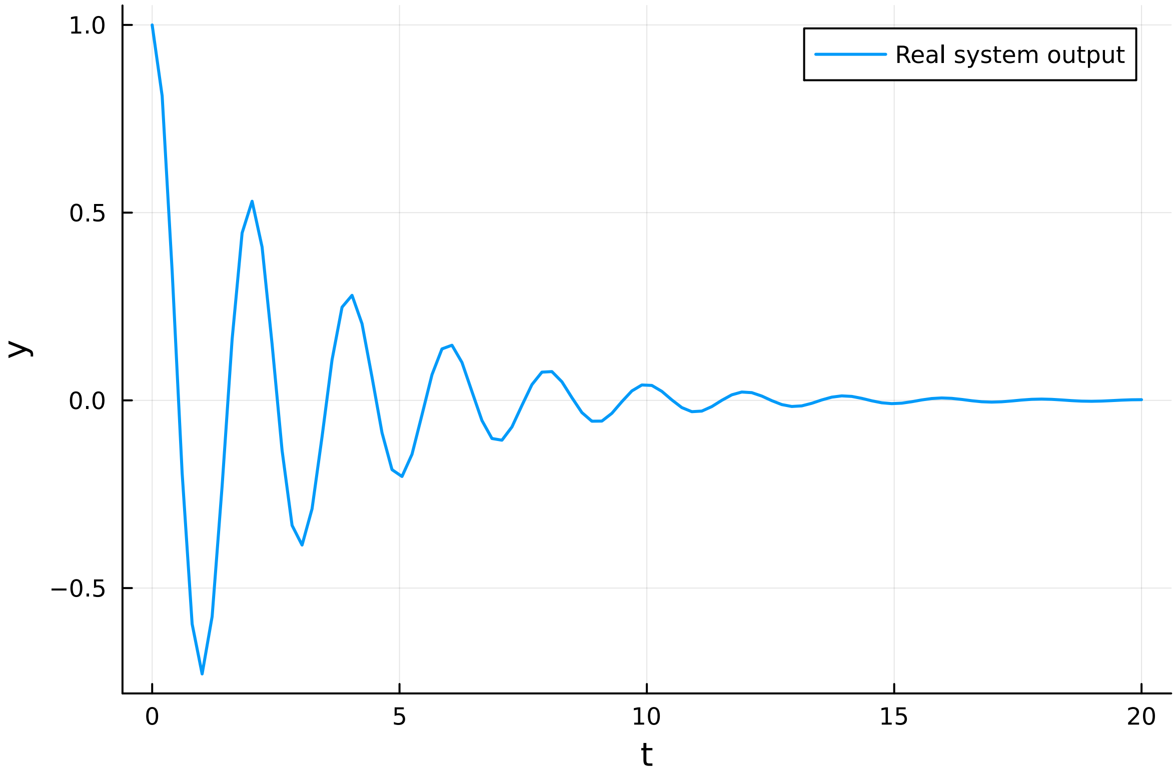
```
begin
    using NeuralPDE
    using LinearAlgebra
    using Plots
    # using Flux
    using Lux, Optimization
    import OptimizationOptimisers
    import ModelingToolkit: Interval
    using CUDA, Random, ComponentArrays
    using OptimizationOptimJL
end
```

```
Differential(t) ∘ Differential(t)
```

```
begin
    @parameters t
    @variables y(..)
    Dt = Differential(t)
    Dtt = Differential(t)^2
end
```

```
use_gpu = false
```

```
use_gpu = false
```

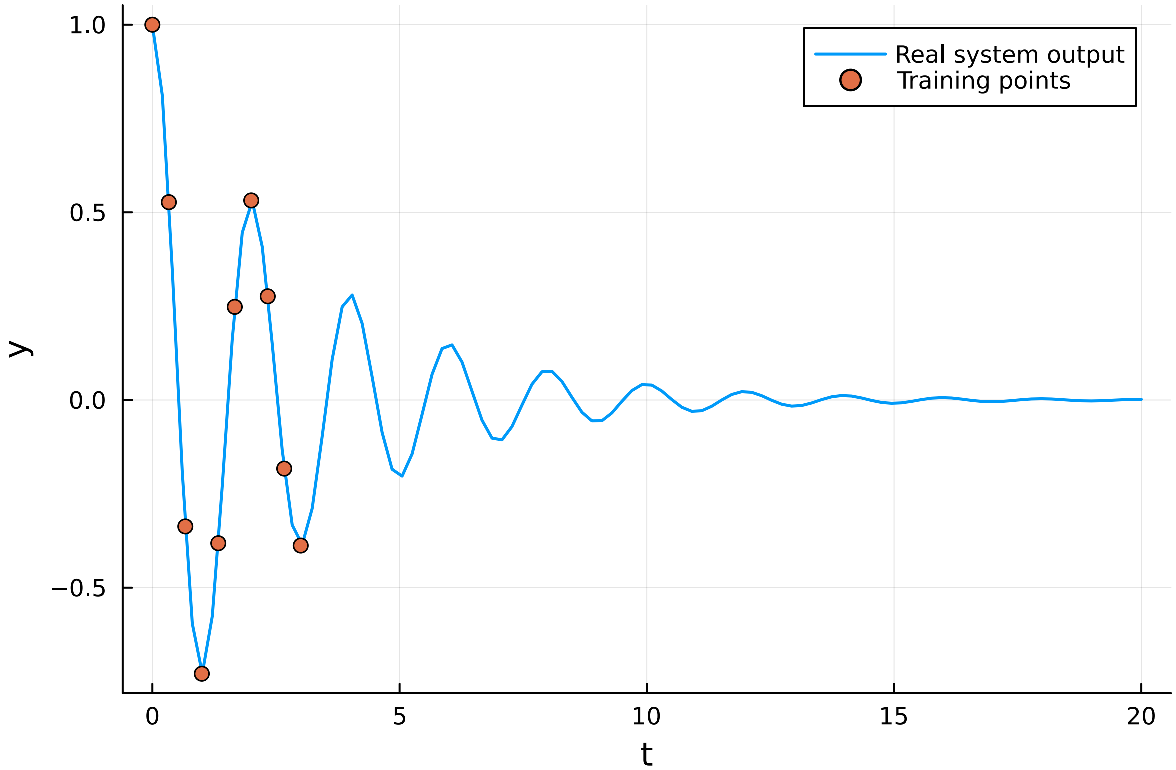



```julia
begin
    # Mass-spring-damper system parameters
    m = 2.0
    k = 20.0
    ζ = 0.1  # Smaller damping ratio for slower energy dissipation
    c = 2 * ζ * sqrt(m * k)

    # Initial conditions
    y0 = 1.0 # Initial displacement
    v0 = 0.0 # Initial velocity

    # Real system
    t_real = range(0.0, stop=20.0, length=100)
    ω0_real = sqrt(k / m)
    ω_d_real = ω0_real * sqrt(1 - ζ^2)
    exp_term_real = exp.(-ζ * ω0_real * t_real)
    osc_term_real = cos.(ω_d_real * t_real) + (ζ / sqrt(1 - ζ^2)) * sin.(ω_d_real * t_real)
    y_real = y0 .* exp_term_real .* osc_term_real

    p1 = plot(t_real, y_real, xlabel="t", ylabel="y", label="Real system output", linewidth=:1.5)
end
```




```julia
begin
    # Time values for training data
    t_u = range(0.0, stop=3.0, length=10)

    # Angular frequency and damping ratio
    ω0 = sqrt(k / m)
    ω_d = ω0 * sqrt(1 - ζ^2)

    # Calculate the exponential and oscillatory components
    exp_term = exp.(-ζ * ω0 * t_u)
    osc_term = cos.(ω_d * t_u) + (ζ / sqrt(1 - ζ^2)) * sin.(ω_d * t_u)

    initial_conditions = [y(0.0) ~ y0,
                          Dt(y(0.0)) ~ v0]

    # Training points
    y_u = y0 .* exp_term .* osc_term

    training_points = [y(t_u_) ~ y_u_ for (t_u_, y_u_) in zip(t_u, y_u)]

    p2 = plot(t_real, y_real, xlabel="t", ylabel="y", label="Real system output",
    linewidth=:1.5)
    p2 = scatter!(t_u, y_u, label="Training points")

end
```

```julia
begin
    # ODE
    f = (y, t) -> m * Dtt(y(t)) + c * Dt(y(t)) + k * y(t)
    eq = f(y, t) ~ 0

    # PINN
    chain = Lux.Chain(Lux.Dense(1, 64, Lux.σ),
                      Lux.Dense(64, 64, Lux.σ),
                      # Lux.Dense(32, 32, Lux.σ),
                      Lux.Dense(64, 1))

    if use_gpu
        ps = Lux.setup(Random.default_rng(), chain)[1]
        ps = ps |> ComponentArray |> Lux.gpu .|> Float64
    end

    # opt = OptimizationOptimisers.Adam(0.01)
    opt = Optim.BFGS()

    # Boundary conditions
    boundary_conditions = vcat(initial_conditions, training_points)

    # strategy = GridTraining(0.1)
    strategy = QuasiRandomTraining(200, bcs_points=length(boundary_conditions)/2)
    discretization = PhysicsInformedNN(chain, strategy)

    domain = [t ∈ Interval(0.0, 20.0)]

    @named system = PDESystem(eq, boundary_conditions, domain, [t], [y(t)])
    prob = discretize(system, discretization)

    epoch = 0

    callback = function (p, l)
        global epoch
        epoch += 1
        if epoch % 100 == 0
            println("Epoch: $epoch\tLoss: $l")
        end
        return false
    end

    display(chain)
end
```

```
Chain(
    layer_1 = Dense(1 => 64, sigmoid_fast),  # 128 parameters
    layer_2 = Dense(64 => 64, sigmoid_fast),  # 4_160 parameters
    layer_3 = Dense(64 => 1),                 # 65 parameters
)         # Total: 4_353 parameters,
          #        plus 0 states, summarysize 48 bytes.
```

res =
SciMLBase.OptimizationSolution{Float64, 1, ComponentArrays.ComponentVector{Float64, Vecto

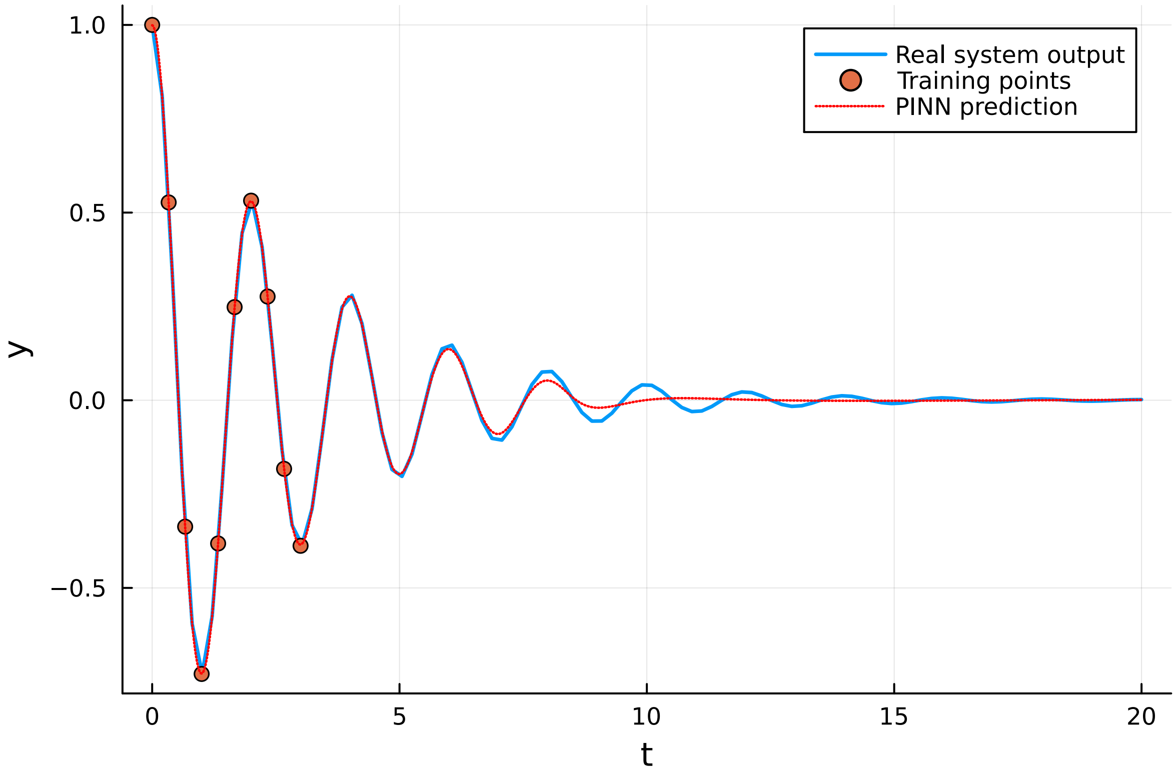
```
res = @time Optimization.solve(prob, opt; callback=callback, maxiters=1000)
```

```
Epoch: 100  Loss: 0.7548978692910726
Epoch: 200  Loss: 0.1976231264759747
Epoch: 300  Loss: 0.07342961578671824
Epoch: 400  Loss: 0.03622208456806477
Epoch: 500  Loss: 0.021201140977161203
Epoch: 600  Loss: 0.012698842893390683
Epoch: 700  Loss: 0.009209591134113574
Epoch: 800  Loss: 0.007598321849814344
Epoch: 900  Loss: 0.006634043870866203
Epoch: 1000 Loss: 0.005845939177140362
312.166092 seconds (305.80 M allocations: 215.230 GiB, 5.18% gc time, 1.42% com
pilation time)
WARNING: both DomainSets and SciMLBase export "isconstant"; uses of it in modul
e NeuralPDE must be qualified
WARNING: both DomainSets and SciMLBase export "islinear"; uses of it in module
NeuralPDE must be qualified
WARNING: both DomainSets and SciMLBase export "issquare"; uses of it in module
NeuralPDE must be qualified
```

[0.999121, 0.998624, 0.997131, 0.994649, 0.991189, 0.98676, 0.981374, 0.975041, 0.967776,

```
begin
    phi = discretization.phi

    t_span = (0.0:0.01:20.0)
    y_predict = [first(phi([t], res.u)) for t in t_span]
end
```




```
begin
    p3 = plot(t_real, y_real, xlabel="t", ylabel="y", label="Real system output",
    linewidth=:1.8)
    p3 = scatter!(t_u, y_u, label="Training points")
    p3 = plot!(t_span, y_predict, xlabel="t", ylabel="y", label="PINN prediction",
    linewidth=:1.2, linecolor=:red, linestyle=:dot)
end
```