```
using Pkg; Pkg.activate("/Users/dhairyagandhi/Downloads/temp/heise")
```

Metalhead Example

base_path = "/Users/dhairyagandhi/Downloads/temp/heise/data"

```
base_path = "/Users/dhairyagandhi/Downloads/temp/heise/data"
```

```
using Metalhead, Flux
```

philip =

```
philip = load(joinpath(base_path, "philip_crop.jpg"))
```

vgg = VGG19()

```
vgg = VGG19()
```

```
"Cardigan, Cardigan Welsh corgi"
```

```julia
classify(vgg, philip)
```

Flux3D Example

```julia
using Flux3D, Zygote, AbstractPlotting, Statistics
```

```
3×2562 Array{Float32,2}:
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0     0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
```

```julia
begin
    dolphin = load_trimesh(joinpath(base_path, "dolphin.obj"))
    src = load_trimesh(joinpath(base_path, "sphere.obj"))
    _offset = zeros(Float32, size(get_verts_packed(src))...)
end
```

```julia
# using GLMakie
```

```julia
# save(joinpath(base_path, "initial_guess2.png"), visualize(src))
```

Normalizing the Target Mesh

```
3×2562 Array{Float32,2}:
 -0.101319335  0.100773014  -0.102468416  …  -0.059885267  -0.061462224  -0.07344546
  0.28716984   0.28668007   -0.15838437      -0.21195507   -0.15824707   -0.17916189
  0.09061999   0.09080836   -0.12292277      -0.64931875   -0.61420053   -0.60773396
```

```julia
begin
    tgt = deepcopy(dolphin)
    verts = get_verts_packed(tgt)
    center = mean(verts, dims=2)
    verts = verts .- center
    scale = maximum(abs.(verts))
    verts = verts ./ scale
    tgt._verts_packed = verts
end
```

Defining the Loss

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

```julia
function loss_dolphin(x::AbstractArray, src::TriMesh, tgt::TriMesh)
    src = Flux3D.offset(src, x)
    loss1 = chamfer_distance(src, tgt, 5000)
    loss2 = laplacian_loss(src)
    loss3 = edge_loss(src)
    return loss1 + 0.1*loss2 + loss3
end
```

Training the Model

```
begin
    θ = Flux.params(_offset)
    nepochs = 10
    for itr in 1:nepochs
        gs = gradient(θ) do
            loss_dolphin(_offset, src, tgt)
        end
        Flux.update!(opt, _offset, gs[_offset])
    end
end
```

0.3716024

```
save(joinpath(base_path, "somewhat.png"), visualize(Flux3D.offset(src, _offset)))
```

```
save(joinpath(base_path, "somewhat.png"), visualize(Flux3D.offset(src, _offset)))
```

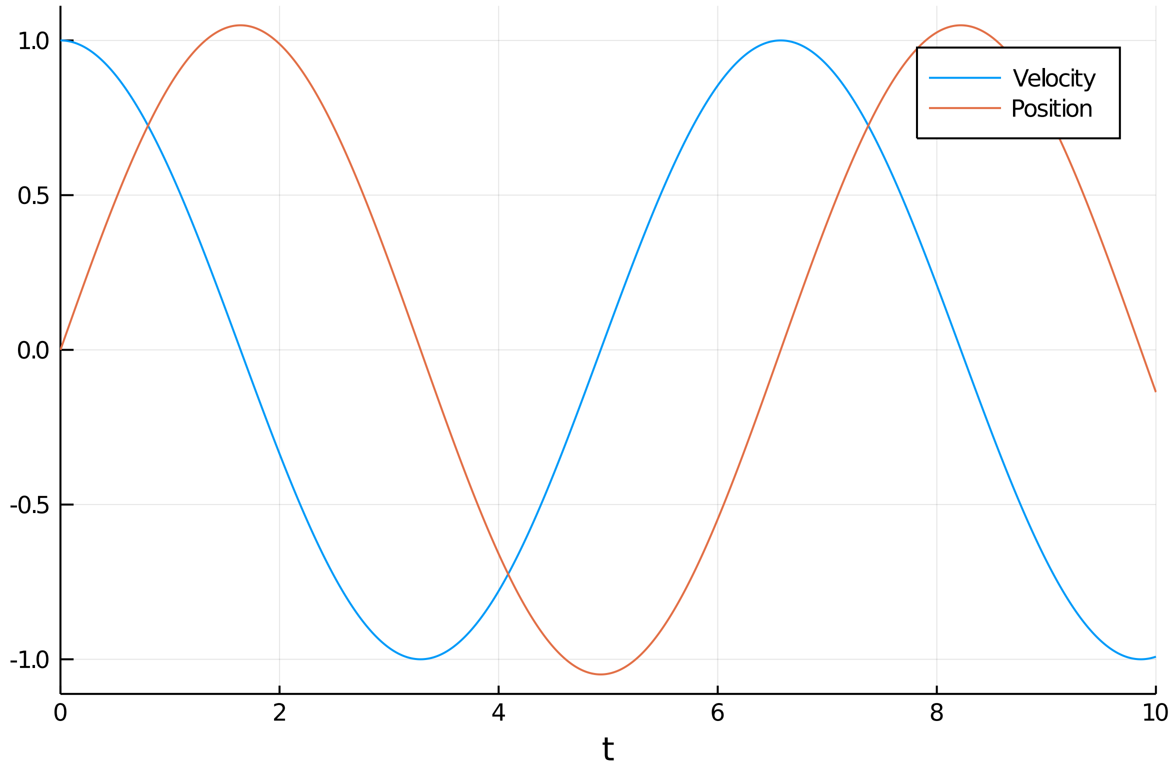
SciML Example - Hooke's Law

Imperfect Spring

$$x'' = -k \times x + 0.1 \times \sin(x)$$

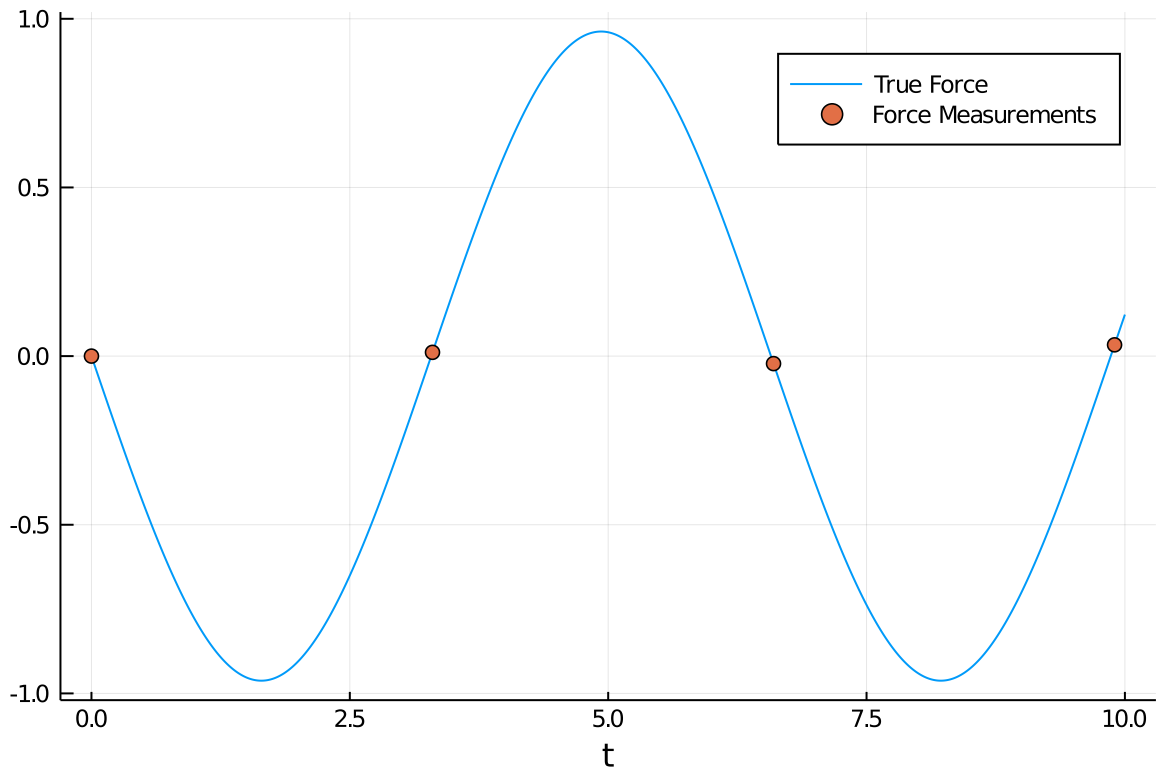
```
using DifferentialEquations, Plots
```

Plot Actual Solution




```
begin
    k = 1.0
    force(dx,x,k,t) = -k*x + 0.1sin(x)
    prob = SecondOrderODEProblem(force,1.0,0.0,(0.0,10.0),k)
    sol = solve(prob)
    Plots.plot(sol,label=["Velocity" "Position"])
end
```

Sampling Datapoints from Actual Solution



```
begin
    t = 0:0.001:1.0
    plot_t = 0:0.01:10
    data_plot = sol(plot_t)
    positions_plot = [state[2] for state in data_plot]
    force_plot = [force(state[1],state[2],k,t) for state in data_plot]

    # Generate the dataset
    t = 0:3.3:10
    dataset = sol(t)
    position_data = [state[2] for state in sol(t)]
    force_data = [force(state[1],state[2],k,t) for state in sol(t)]

    Plots.plot(plot_t,force_plot,xlabel="t",label="True Force")
    Plots.scatter!(t,force_data,label="Force Measurements")
end
```

Define Neural Network and L2 Loss

The neural network is trained to match the force values at every position

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

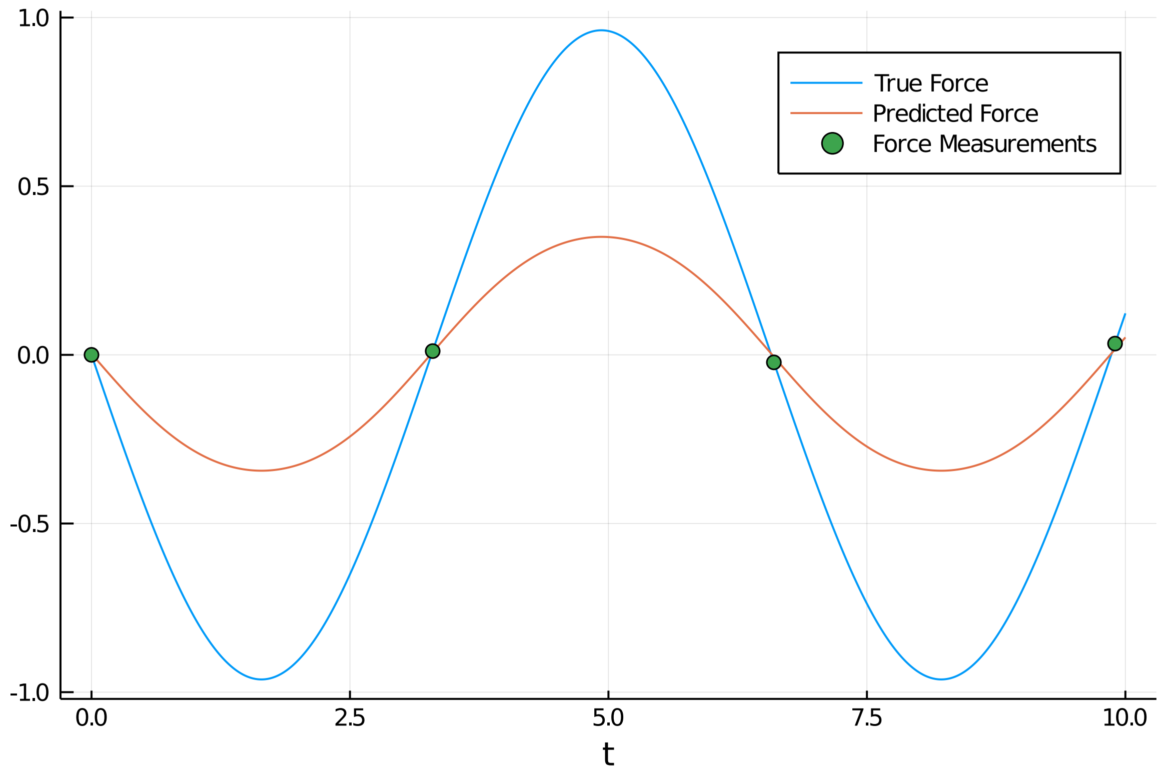
```
begin
    NNForce = Chain(x -> [x],
                Dense(1,32,tanh),
                Dense(32,1),
                first)

    loss() = sum(abs2,NNForce(position_data[i]) - force_data[i] for i in
1:length(position_data))
end
```

Train the neural network

```
begin
    opt = Flux.Descent(0.01)
    data = Iterators.repeated((), 5000)
    Flux.train!(loss, Flux.params(NNForce), data, opt)
end
```

Plot - Trained Model with 4 Datapoints

The trained model fits the datapoints perfectly, but does not capture the real physics

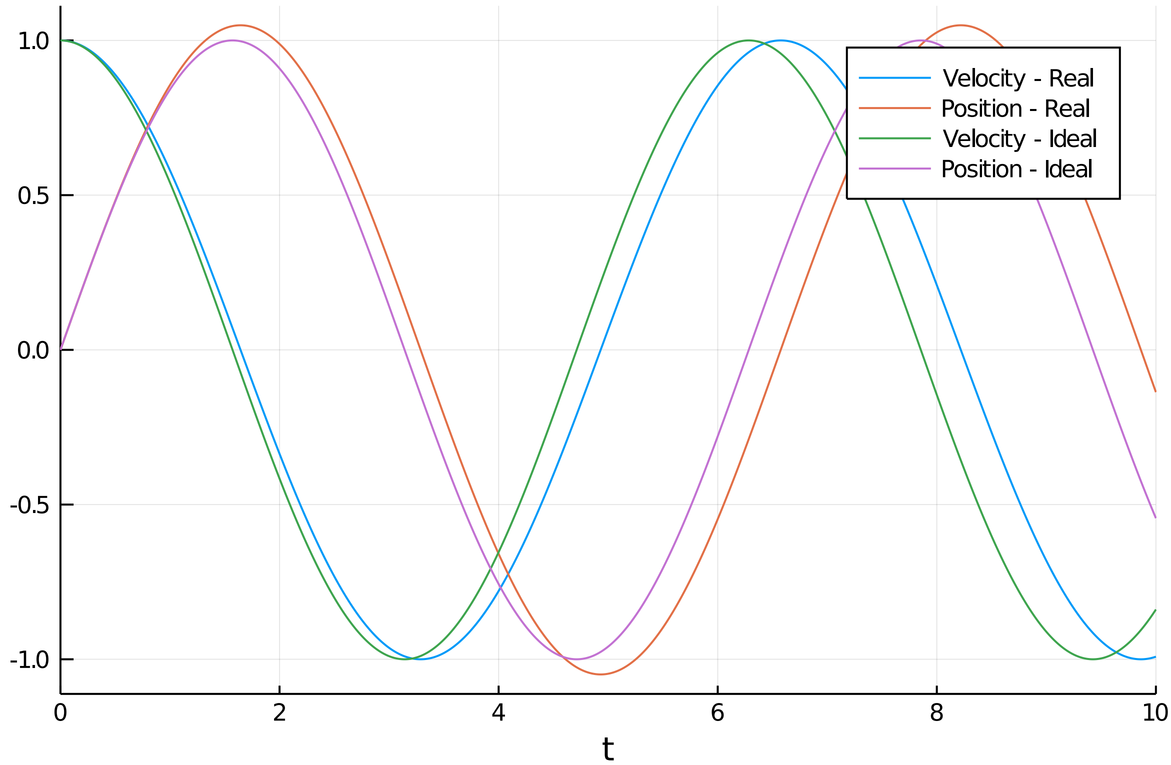



```
begin
    learned_force_plot = NNForce.(positions_plot)

    Plots.plot(plot_t,force_plot,xlabel="t",label="True Force")
    Plots.plot!(plot_t,learned_force_plot,label="Predicted Force")
    Plots.scatter!(t,force_data,label="Force Measurements")
end
```

This shows the neural network fit the force data but doesn't match the force function well enough.

In this case, an extra loss component which combines the constraints of the ideal spring are added

Plot - Ideal Spring vs Real Spring




```
begin
    force2(dx,x,k,t) = -k*x
    prob_simplified = SecondOrderODEProblem(force2,1.0,0.0,(0.0,10.0),k)
    sol_simplified = solve(prob_simplified)
    Plots.plot(sol,label=["Velocity - Real" "Position - Real"])
    Plots.plot!(sol_simplified,label=["Velocity - Ideal" "Position - Ideal"])
end
```

Generate more data assuming ideal spring. This is done by calculating force positions at random points.

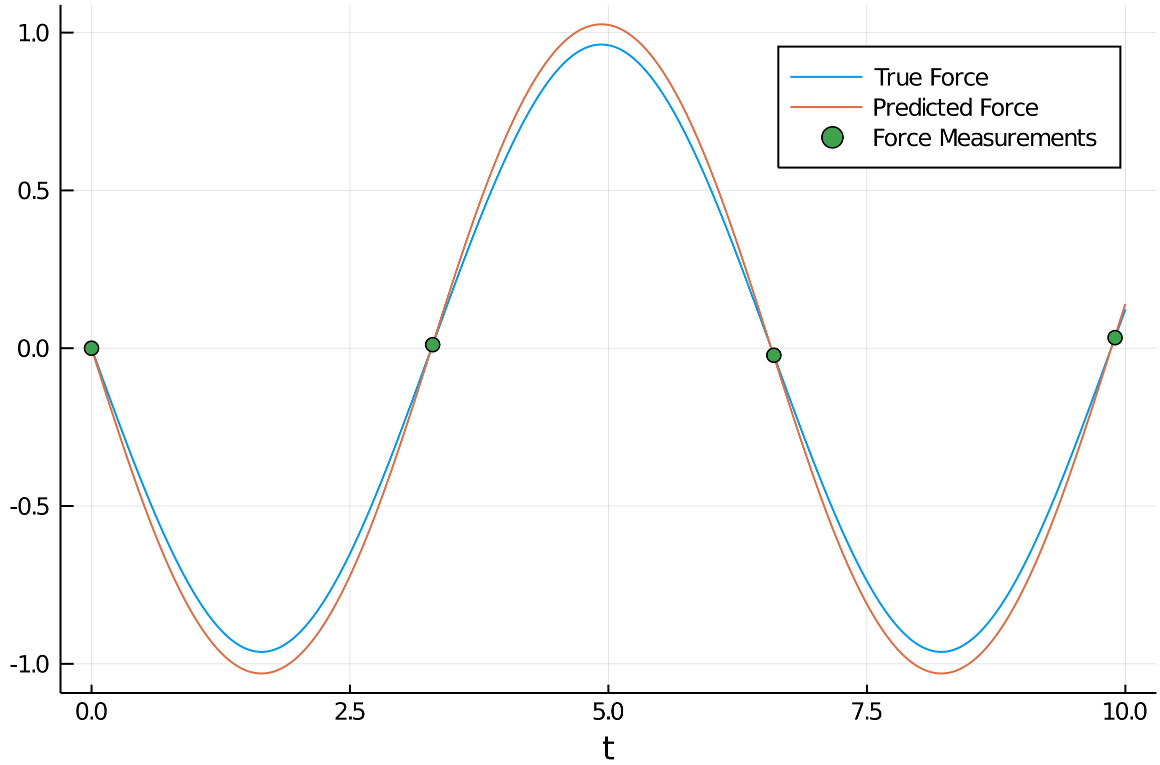
```
composed_loss (generic function with 1 method)
```

```
begin
    random_positions = [2rand()-1 for i in 1:100] # random values in [-1,1]
    loss_ode() = sum(abs2,NNForce(x) - (-k*x) for x in random_positions)

    # Composed loss with weighted ODE loss component
    λ = 0.1
    composed_loss() = loss() + λ*loss_ode()
end
```

Train Neural Network with ODE loss component

```
let
    opt = Flux.Descent(0.01)
    data = Iterators.repeated((), 5000)
    Flux.train!(composed_loss, Flux.params(NNForce), data, opt)
end
```

Plot - Trained Neural Network vs Actual Force




```
let
    learned_force_plot = NNForce.(positions_plot)

    Plots.plot(plot_t,force_plot,xlabel="t",label="True Force")
    Plots.plot!(plot_t,learned_force_plot,label="Predicted Force")
    Plots.scatter!(t,force_data,label="Force Measurements")
end
```

This shows the trained neural network approximating the actual force function very closely.