Table of Contents

Inverse Problem.

Let us say we have a black box $f(x)$ that models a system.

> We want to find a value x such that $f(x) = t$.

It can be formulated as an optimization problem by opimixing the loss

$$L(x) = |f(x) - t|$$

We can use the gradient decent to optimize the loss function if it is differentiable.

Differentiable Programming

One such example of a black box function is given by a Differntial equation solver. Thanks to the new Automatic differntiation library Zygote.jl, which is now the default AD in flux, it can do source to source automatic differntiation. It can now imlement AD on functions written in Julia, In particular, on any Differntial equation solver. This is in the spirit of **Differntiable programming**, a new paradigm in computation that enable complete code to be differntiated. This is enabling architecture beyond neural networks by implementing different solvers inside as layers.

Parameter optimization

Let us consider one example of the inverse problem using DiffEqFlux.jl. Consider a system of equations with unknown parameters.

> Can we find the parameters with low amount of data?

Lorenz system

Consider the Lorenz equations

$$x' = \alpha y - \alpha x$$

$$y' = x * (\varrho - z) - y$$

$$z' = x * y - \beta * z$$

Problem

> What parameters will stabilise x to 5?

We will start with a random parameter and decent towards the ones that brings X closer to 5.

```
lorenz! (generic function with 1 method)
```

```
begin
    using DifferentialEquations, Flux, DiffEqFlux, Plots

    function lorenz!(du, u, p, t)
      x, y, z = u
      α, β, ρ = p
      du[1] = dx = α*y - α*x
      du[2] = dy = x*(ρ-z) -y
      du[3] = dz = x*y - β*z
    end
end
```

```
Float64[1.0, 3.0, 2.0]
```

```
begin
    u0 = [1.0, 1.0, 1.0]
```

```
    tspan = (0.0, 10.0)
    tsteps = 0.0:0.1:10.0
    p = [1.0, 3.0, 2.0]
end
```

Visualize

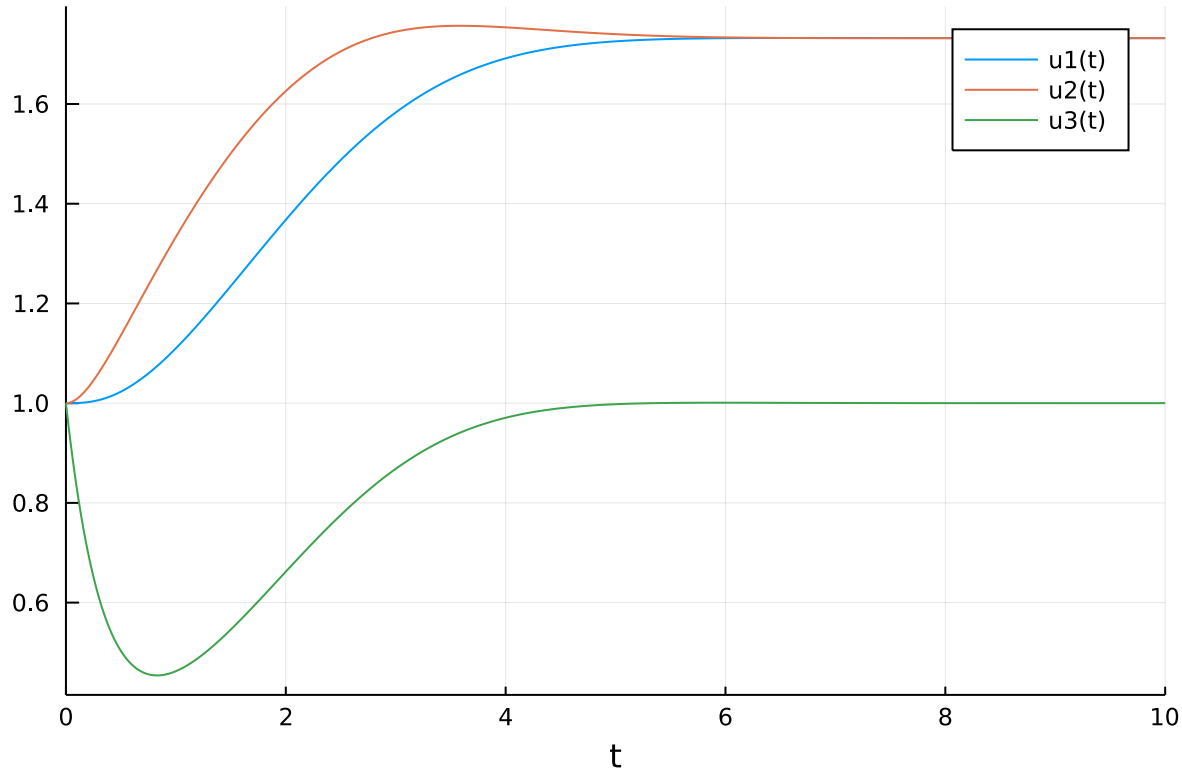
	timestamp	value1	value2	value3
1	0.0	1.0	1.0	1.0
2	0.0771957	1.00014	1.00538	0.862319
3	0.197077	1.002	1.03012	0.704345
4	0.336078	1.00844	1.07402	0.584419
5	0.512407	1.02451	1.14097	0.49882
6	0.71633	1.0538	1.22292	0.458691
7	0.957852	1.10074	1.31749	0.458969
8	1.2379	1.16677	1.41811	0.494689
9	1.571	1.25389	1.52158	0.561602
10	1.96356	1.35829	1.61839	0.653385

more

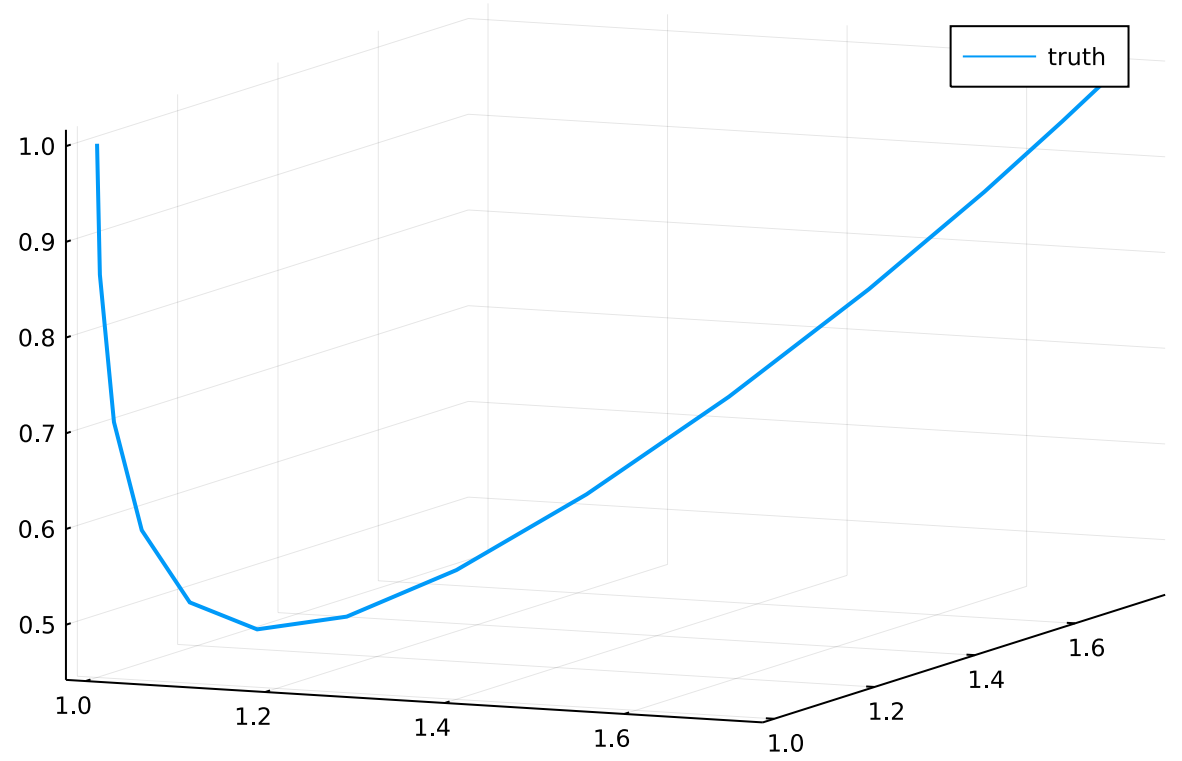
```
begin
    prob = ODEProblem(lorenz!, u0, tspan, p)
    sol = solve(prob, Tsit5())
end
```

```
Float64[1.0, 0.862319, 0.704345, 0.584419, 0.49882, 0.458691, 0.458969, 0.494689, 0
```

```
sol[3,:]
```




```
begin
    plot(sol)
end
```




```
plot(sol[1,:], sol[2,:], sol[3,:], linewidth = 2, label = "truth", legend = true)
```

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

```
begin
    params = Flux.params(p)
```

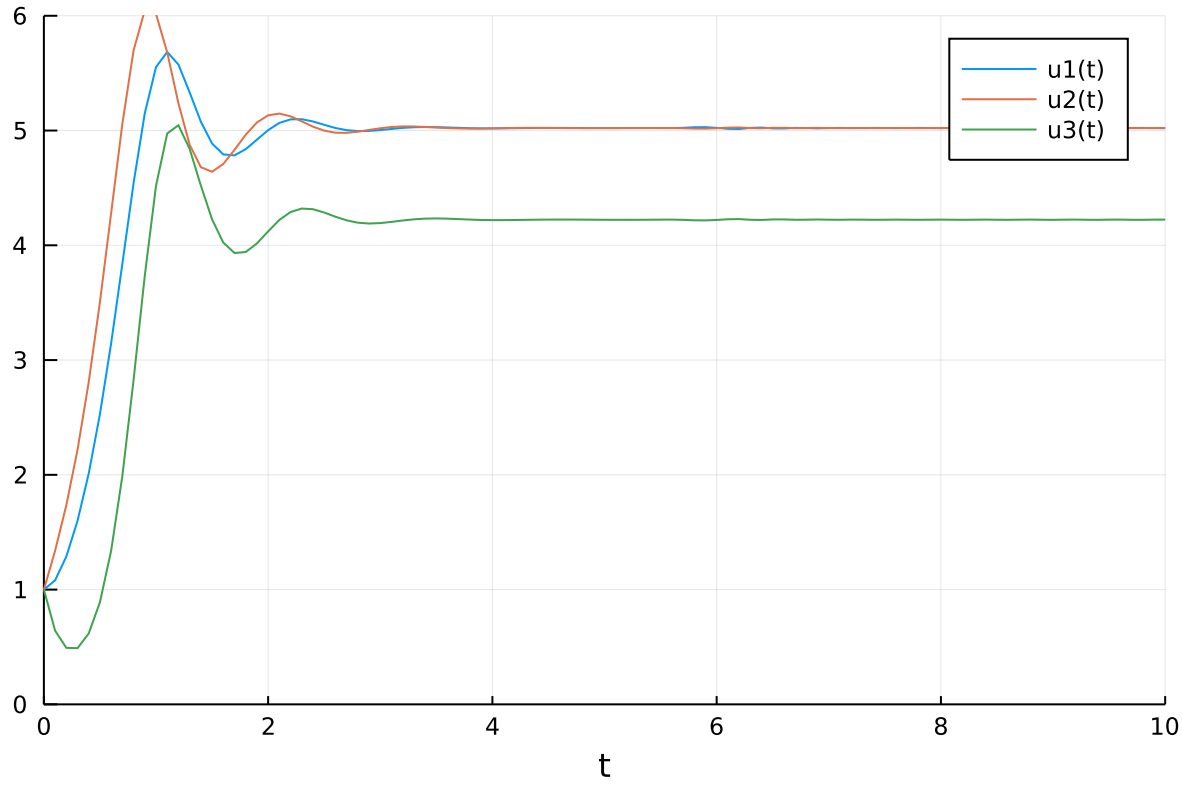
```
function predict_rd() # Our 1-layer "neural network"
  solve(prob,Tsit5(),p=p,saveat=0.1)[1,:] # override with new parameters
end

loss_rd() = sum(abs2,x-5 for x in predict_rd()) # loss function

end
```

```
begin
    data = Iterators.repeated((), 100)
    opt = ADAM(0.1)

    Flux.train!(loss_rd, params, data, opt)
end
```




```
plot(solve(remake(prob,p=p),Tsit5(),saveat=0.1),ylim=(0,6))
```

new_sol =

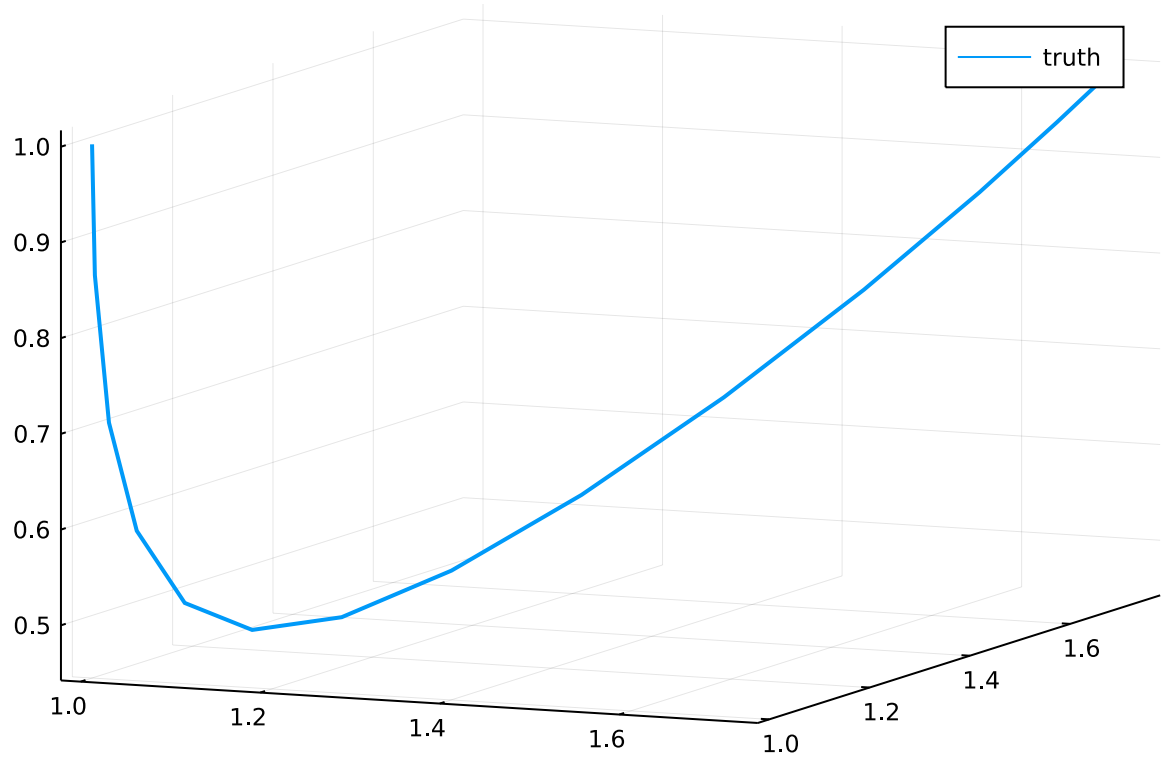
	timestamp	value1	value2	value3
1	0.0	1.0	1.0	1.0
2	0.0771957	1.00014	1.00538	0.862319
3	0.197077	1.002	1.03012	0.704345
4	0.336078	1.00844	1.07402	0.584419
5	0.512407	1.02451	1.14097	0.49882
6	0.71633	1.0538	1.22292	0.458691
7	0.957852	1.10074	1.31749	0.458969

	timestamp	value1	value2	value3
8	1.2379	1.16677	1.41811	0.494689
9	1.571	1.25389	1.52158	0.561602

```
new_sol = solve(prob,Tsit5())
```

```
Float64[5.83646,  5.97144,  5.22294]
```

```
p
```




```
plot(new_sol[1,:], new_sol[2,:], new_sol[3,:], linewidth = 2, label = "truth", legend = true)
```