Unsteady Heat Conduction Equation

$$\frac{\partial T(x,t)}{\partial t} = \alpha \frac{\partial^2 T(x,t)}{\partial x^2}$$

```
begin
    using NeuralPDE, Lux, Optimization, OptimizationOptimJL
    using CUDA, Random, ComponentArrays
    import ModelingToolkit: Interval
    # using OptimizationOptimisers # for Adam
end
```

use_gpu = false

```
use_gpu = false
```

(::Symbolics.Differential) (generic function with 2 methods)

```
begin
    @parameters t, x
    @variables u(..)
    Dxx = Differential(x)^2
    Dtt = Differential(t)^2
    Dt = Differential(t)
end
```

```
[VarDomainPairing(t, 0.0..0.25), VarDomainPairing(x, 0.0..1.0)]
```

```
begin
    #PDE
    alpha = 1
    eq = Dt(u(t, x)) ~ alpha * Dxx(u(t, x))

    # Initial and boundary conditions
    bcs = [u(t, 0) ~ 0.0, # for all t > 0
        u(t, 1) ~ 0.0, # for all t > 0
        u(0, x) ~ (sin(π * x) + 0.5 * sin(3 * π * x) + 0.25 * sin(5 * π * x))] #for all  0 < x < 1

    t_max = 0.25
    x_max = 1.0

    # Space and time domains
    domains = [t ∈ Interval(0.0, t_max),
        x ∈ Interval(0.0, x_max)]
end
```

```
0.008620689655172414
```

```
begin
    # Discretization parameters
    Nx = 30  # Number of spatial grid points
    Nt = 30  # Number of time steps
    dx = x_max / (Nx - 1)
    dt = t_max / (Nt - 1)
end
```

```
begin
    # Neural network
    hidden_size = 16

    chain = Chain(Dense(2, hidden_size, Lux.sigmoid_fast),
                  Dense(hidden_size, hidden_size, Lux.sigmoid_fast),
                  Dense(hidden_size, hidden_size, Lux.sigmoid_fast),
                #   Dense(hidden_size, hidden_size, Lux.sigmoid_fast),
                #   Dense(hidden_size, hidden_size, Lux.sigmoid_fast),
                  Dense(hidden_size, 1))

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

    # strategy = GridTraining([dt, dx])
    Nf = Nx * Nt # number of collocation points for pde evalution
    # Nb = Nx + 2 * Nt # number of points for boundary and initial conditions evaluation
    Nb = Nx + 2 * Nt # number of points for boundary and initial conditions evaluation
    # strategy = StochasticTraining(Nf + Nb, bcs_points=Nb)
    # strategy = QuasiRandomTraining(Nf + Nb, bcs_points=Nb)
    # strategy = QuasiRandomTraining(Nf + Nb, bcs_points=Nb, resampling=false, minibatch=1)
    # strategy = QuasiRandomTraining(Nf + Nb)
    strategy = QuasiRandomTraining(Nf + Nb, bcs_points=Nb, sampling_alg=NeuralPDE.SobolSample())
    # strategy = QuasiRandomTraining(Nf + Nb, bcs_points=Nb, sampling_alg=NeuralPDE.SobolSample(), resampling=false, minibatch=1)
    discretization = PhysicsInformedNN(chain, strategy)

    display(chain)
end
```

```
Chain(
    layer_1 = Dense(2 => 16, sigmoid_fast),  # 48 parameters
    layer_2 = Dense(16 => 16, sigmoid_fast),  # 272 parameters
    layer_3 = Dense(16 => 16, sigmoid_fast),  # 272 parameters
    layer_4 = Dense(16 => 1),           # 17 parameters
)         # Total: 609 parameters,
          #        plus 0 states, summarysize 64 bytes.
```

`@named pde_system =`

$$\frac{\mathrm{d}}{\mathrm{d}t}u\left(t,x\right) = \frac{\mathrm{d}}{\mathrm{d}x}\frac{\mathrm{d}}{\mathrm{d}x}u\left(t,x\right)$$

```
@named pde_system = PDESystem(eq, bcs, domains, [t, x], [u(t, x)])
```

```
BFGS(LineSearches.InitialStatic{Float64}, LineSearches.HagerZhang{Float64, Base.RefValue{
      alpha: Float64 1.0                   delta: Float64 0.1
```

```
begin
    prob = discretize(pde_system, discretization)
    epoch = 0

    callback = function (p, l)
        global epoch
        epoch += 1
        if epoch % 10 == 0
            println("Epoch: $epoch\tLoss: $l")
        end
        return false
    end

    # optimizer
    opt = Optim.BFGS()
    # opt = Adam()
    # opt = Optim.GradientDescent(P=0.01)
end
```

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

```
1 res = @time Optimization.solve(prob, opt; callback=callback, maxiters=10000)
```

```
Epoch: 5310 Loss: 0.44936570161184225
Epoch: 5320 Loss: 0.44471979815290436
Epoch: 5330 Loss: 0.4097078500713332
Epoch: 5340 Loss: 0.35318052118026155
Epoch: 5350 Loss: 0.24534293213703026
Epoch: 5360 Loss: 0.18212370611245865
Epoch: 5370 Loss: 0.15983510293204053
Epoch: 5380 Loss: 0.15200019258764527
Epoch: 5390 Loss: 0.1413307749679522
Epoch: 5400 Loss: 0.13571821849232873
Epoch: 5410 Loss: 0.12560106036255161
Epoch: 5420 Loss: 0.11955064291939033
Epoch: 5430 Loss: 0.11473311301067526
Epoch: 5440 Loss: 0.11205875729315348
Epoch: 5450 Loss: 0.10958424882570222
Epoch: 5460 Loss: 0.10776172247206468
Epoch: 5470 Loss: 0.10590069654942813
Epoch: 5480 Loss: 0.10472861228178883
Epoch: 5490 Loss: 0.10321184318648015
Epoch: 5500 Loss: 0.10223465669460714
Epoch: 5510 Loss: 0.10108951502564628
Epoch: 5520 Loss: 0.09999325375684136
Epoch: 5530 Loss: 0.0981128762844673
Epoch: 5540 Loss: 0.09707476942872562
Epoch: 5550 Loss: 0.09607380807317852
Epoch: 5560 Loss: 0.09533794848393008
Epoch: 5570 Loss: 0.09415727318759233
Epoch: 5580 Loss: 0.09298897231256885
Epoch: 5590 Loss: 0.09176184928060127
Epoch: 5600 Loss: 0.09078597582764283
Epoch: 5610 Loss: 0.08813862187225302
```

```
phi =
  Phi(Chain()             # 609 parameters, (layer_1 = (), layer_2 = (), layer_3 = (), laye
```

```
1 phi = discretization.phi
```

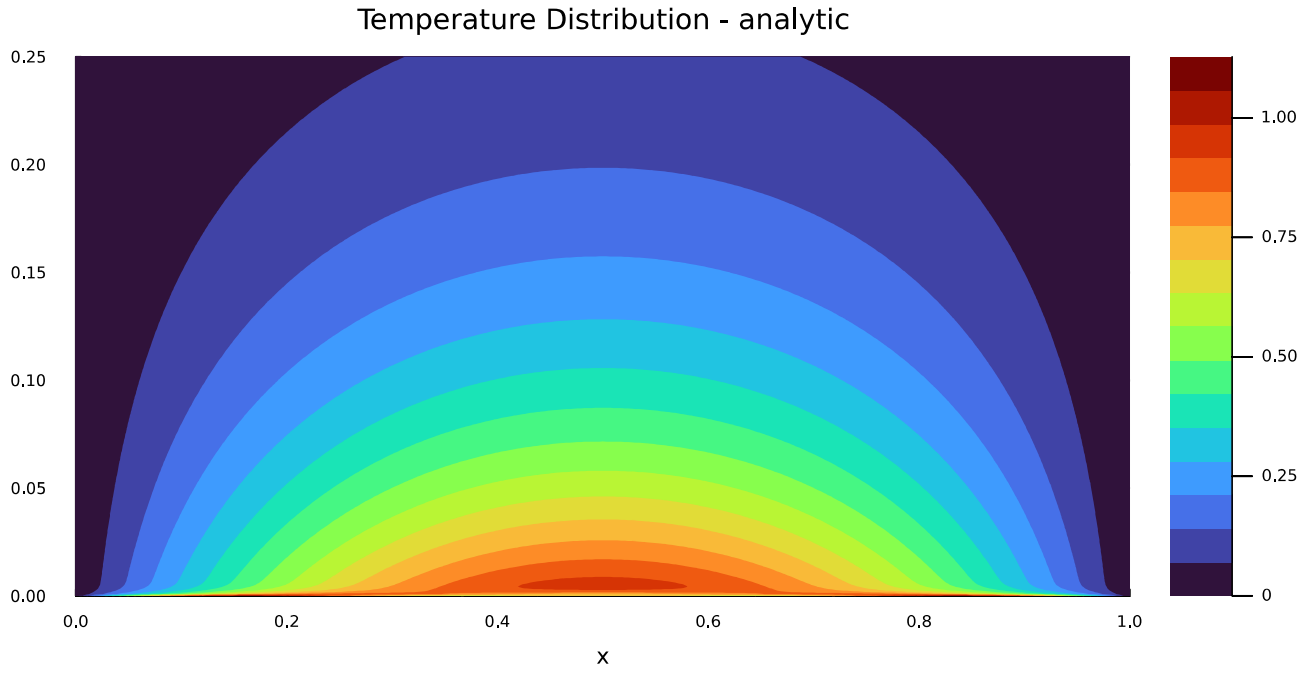
```
1 using Plots
```

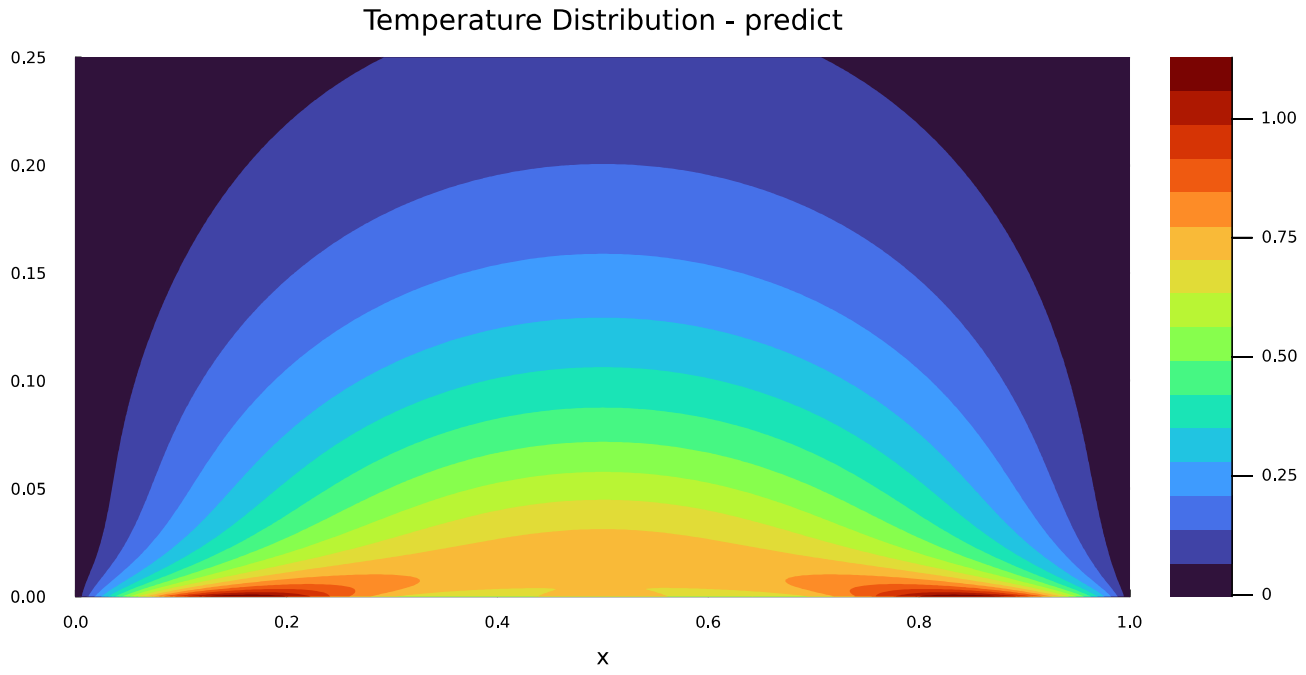
```
analytic_sol_func (generic function with 1 method)
```

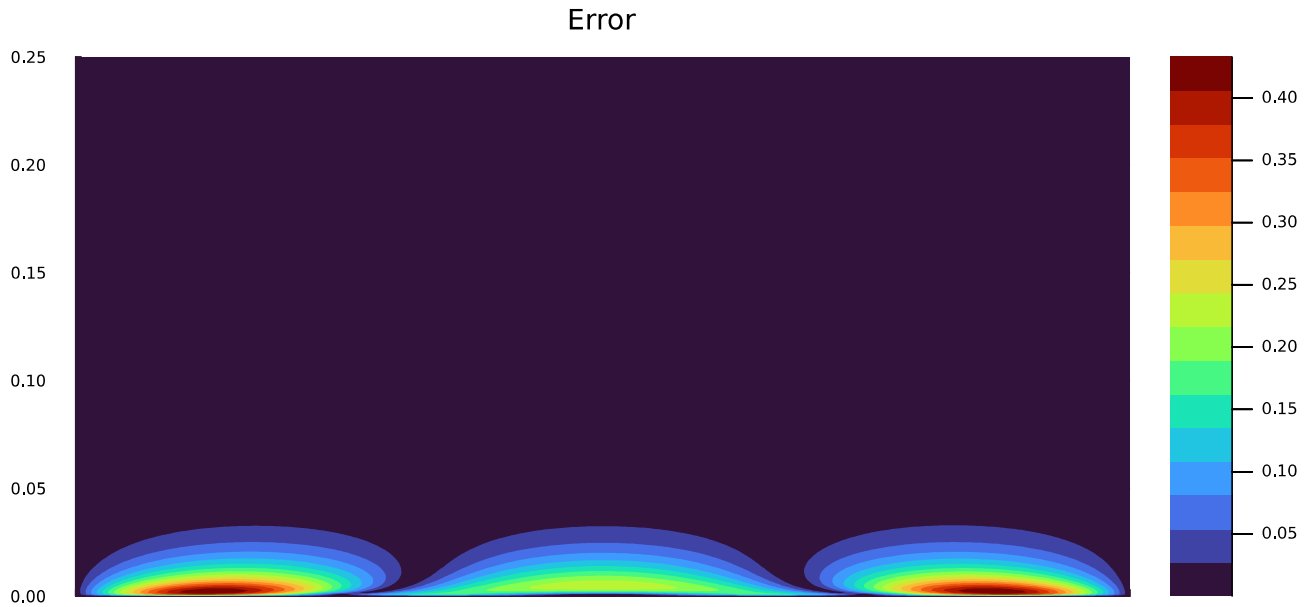
```
begin
    ts = [infimum(d.domain):0.1*dt:supremum(d.domain) for d in domains][1]
    xs = [infimum(d.domain):0.1*dx:supremum(d.domain) for d in domains][2]

    function analytic_sol_func(t, x, alpha)
        term1 = sin(π * x) * exp(-(π * alpha)^2 * t)
        term2 = 0.5 * sin(3 * π * x) * exp(-(9 * π * alpha)^2 * t)
        term3 = 0.25 * sin(5 * π * x) * exp(-(25 * π * alpha)^2 * t)

        T = term1 + term2 + term3
    end
end
```

```
291×291 Matrix{Float64}:
 0.000564397  0.00296404  0.00459859  …  0.00113553  0.00114937  0.00116215
 9.32132e-5   0.0202678   0.0228671      0.00111387  0.00112784  0.00114077
 0.000335725  0.0375601   0.0411156      0.00109215  0.00110624  0.00111931
 0.000720782  0.0548042   0.05931        0.0010704   0.0010846   0.00109778
 0.00106094   0.0719626   0.0774159      0.00104865  0.00106293  0.00107623
 0.00135577   0.088998    0.0953991   …  0.00102692  0.00104128  0.00105467
 0.00160536   0.105873    0.113225       0.00100523  0.00101967  0.00103314
 ⋮                                   ⋱                          ⋮
 0.000913987  0.0901547   0.0951198   …  0.00207732  0.00218215  0.00228892
 0.000898239  0.0727264   0.0767152      0.00214096  0.00224773  0.00235647
 0.000865585  0.0551238   0.0581476      0.00220595  0.0023147   0.00242543
 0.000814303  0.037384    0.0394519      0.00227232  0.00238305  0.00249581
 0.000742502  0.0195443   0.0206627      0.00234008  0.00245283  0.00256762
 0.000648098  0.00164218  0.00181506  …  0.00240925  0.00252405  0.00264091
```

```
begin
    u_predict = reshape([first(phi([t, x], res.u)) for t in ts for x in xs],
                        (length(xs), length(ts)))
    u_real = reshape([analytic_sol_func(t, x, alpha) for t in ts for x in xs],
                     (length(xs), length(ts)))

    diff_u = abs.(u_predict .- u_real)
end
```


Temperature Distribution - analytic
0.25
0.20
0.15
0.10
0.05
0.00
0.0
0.2
0.4
0.6
0.8
1.0
x
1.00
0.75
0.50
0.25
0


Temperature Distribution - predict
0.25
0.20
0.15
0.10
0.05
0.00
0.0
0.2
0.4
0.6
0.8
1.0
x
1.00
0.75
0.50
0.25
0


Error
0.25
0.20
0.15
0.10
0.05
0.00
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05

```
begin
    p1 = contour(xs, ts, u_real', xlabel="x", ylabel="t", title="Temperature
    Distribution - analytic", color=:turbo, levels=15, fill=true, lw=0)
    p2 = contour(xs, ts, u_predict', xlabel="x", ylabel="t", title="Temperature
    Distribution - predict", color=:turbo, levels=15, fill=true, lw=0)
    p3 = contour(xs, ts, diff_u', xlabel="x", ylabel="t", title="Error",
    color=:turbo, levels=15, fill=true, lw=0)

    plot(p1, p2, p3, layout=(3, 1), size=(1000, 1500))
end
```