50.039 Theory and Practice of Deep Learning

W12-S1 About Physics-Informed Neural Networks

Matthieu De Mari

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

About this week (Week 12, an informative lecture about physics-informed neural networks)

1. What is a **Partial Differential Equation (PDE)** and what are **typical uses** of PDEs?
2. Why are PDEs **hard to solve** in practice?
3. What are **Physics-Informed Neural Networks (PINNs)**?
4. What is a **Physics Loss** function and how to write a custom one?
5. How to **train a PINN to solve a given PDE**?
6. Is **Feature Engineering** a good thing to have in PINNs?
7. What are **interesting applications** for PINNs?

Restricted

A reminder on PDEs

Definition (Partial Differential Equation):

A **Partial Differential Equation (PDE)** is an equation, which computes a **function** between **various partial derivatives of a multivariable function**.

The function is often thought of as an "unknown“ for the equation to be solved for, similar to how x is thought of as an unknown number to be solved for in an algebraic equation like $x^2 - 3x + 2 = 0$.

More often than not, it may (but also may not!) include **initial conditions** on top of the partial derivatives equation.

In mathematics, PDEs are notoriously hard to solve and have often been used in Physics, Finance, Optimization and many other fields

Analytical Solution of a PDE

Definition (Analytical Solution of a Partial Differential Equation):

The **Analytical Solution** of a **PDE** is a **function** (or a **set of functions**), written in a closed-form expression, which solve the PDE, along with any additional **initial conditions** that exist on top of the PDE.

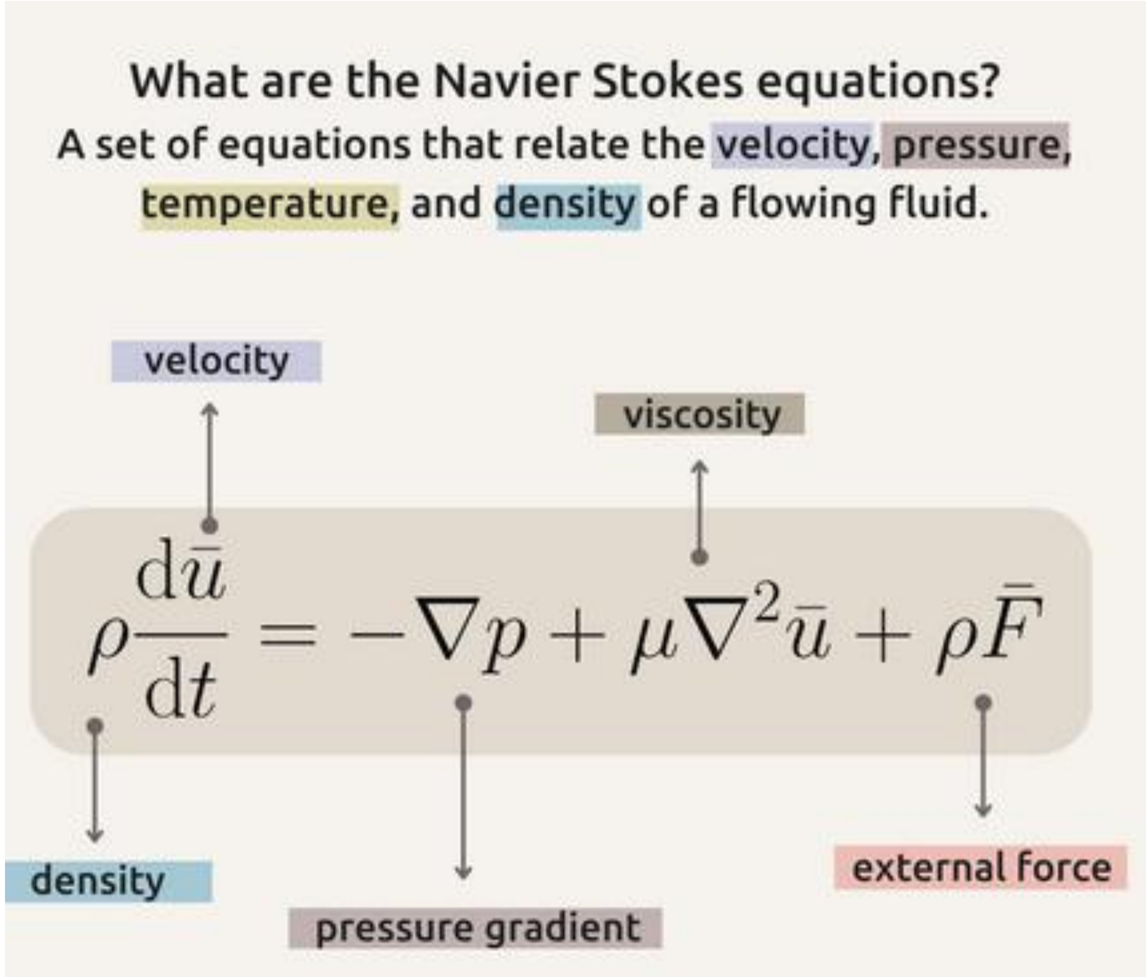
Important note: In most scenarios, except for a few favourable scenarios where certain conditions are met on the PDE, it is simply impossible to find an analytical solution, and we have no other choice but to resort to approximation mechanisms for solving these PDEs.

The Navier-Stokes example

Definition (The Navier-Stokes equations):

The Navier-Stokes equations are PDEs, which are used to describe the motion of viscous fluid substances in Physics.

They are notoriously hard to solve, and in fact, finding an analytical solution to these PDEs happens to be one of the **Millennium Prize Problems**.



The Millenium Prize Problems

Definition (The Millenium Prize Problems):

The **Millennium Prize Problems** are seven well-known complex mathematical problems selected by the Clay Mathematics Institute in 2000.

The Clay Institute has pledged a US$1 million prize for the first correct solution to each problem.

To this day, only one of them has been solved (The Poincaré Conjecture).

- P versus NP.
- Hodge conjecture.
- Riemann hypothesis.
- Yang–Mills existence and mass gap.
- Navier–Stokes existence and smoothness.
- Birch and Swinnerton-Dyer Conjecture.
- ~~Poincaré conjecture.~~

The Mi

Definition
Problems)

The **Millen**
seven well-
mathemati
Clay Mathe

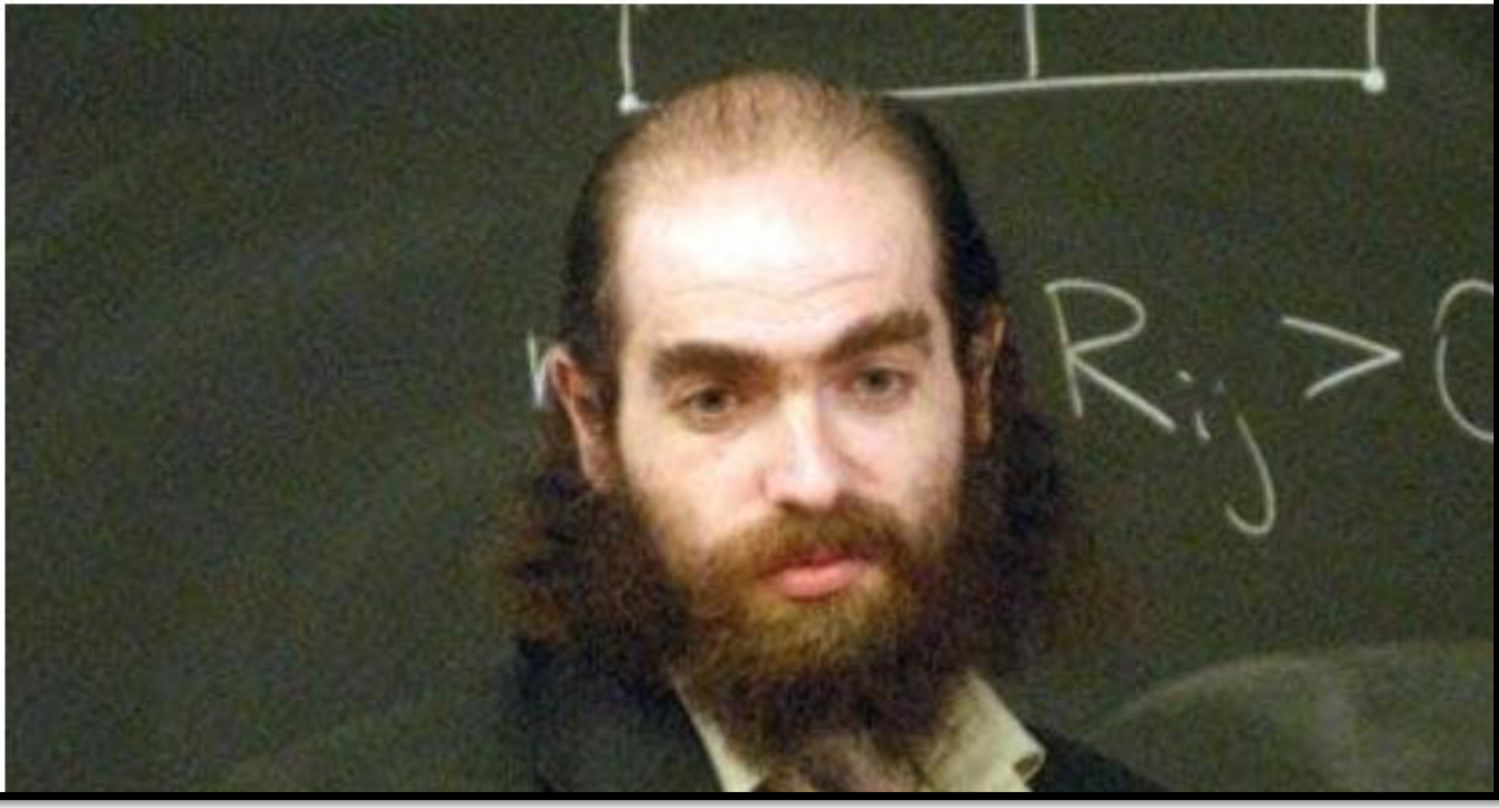
The Clay In
million priz
to each pro

To this day,
solved (The

Mathematician rejected $1 million prize 'because it is unfair'

A Russian mathematician rejected a $1 million prize for solving one of the most challenging problems because he considers it unfair.

01 July 2010 • 5:09pm



e and mass gap.
nce and smoothness.
n-Dyer Conjecture.

Restricted

Quick parenthesis: Why is Navier-Stokes a Millenium Prize Problem anyway?

- The Navier-Stokes equations are fundamental in physics and engineering. We use them in everything from predicting weather, understanding ocean currents, modelling blood flow patterns to understanding aerodynamics and friction in cars/airplanes.

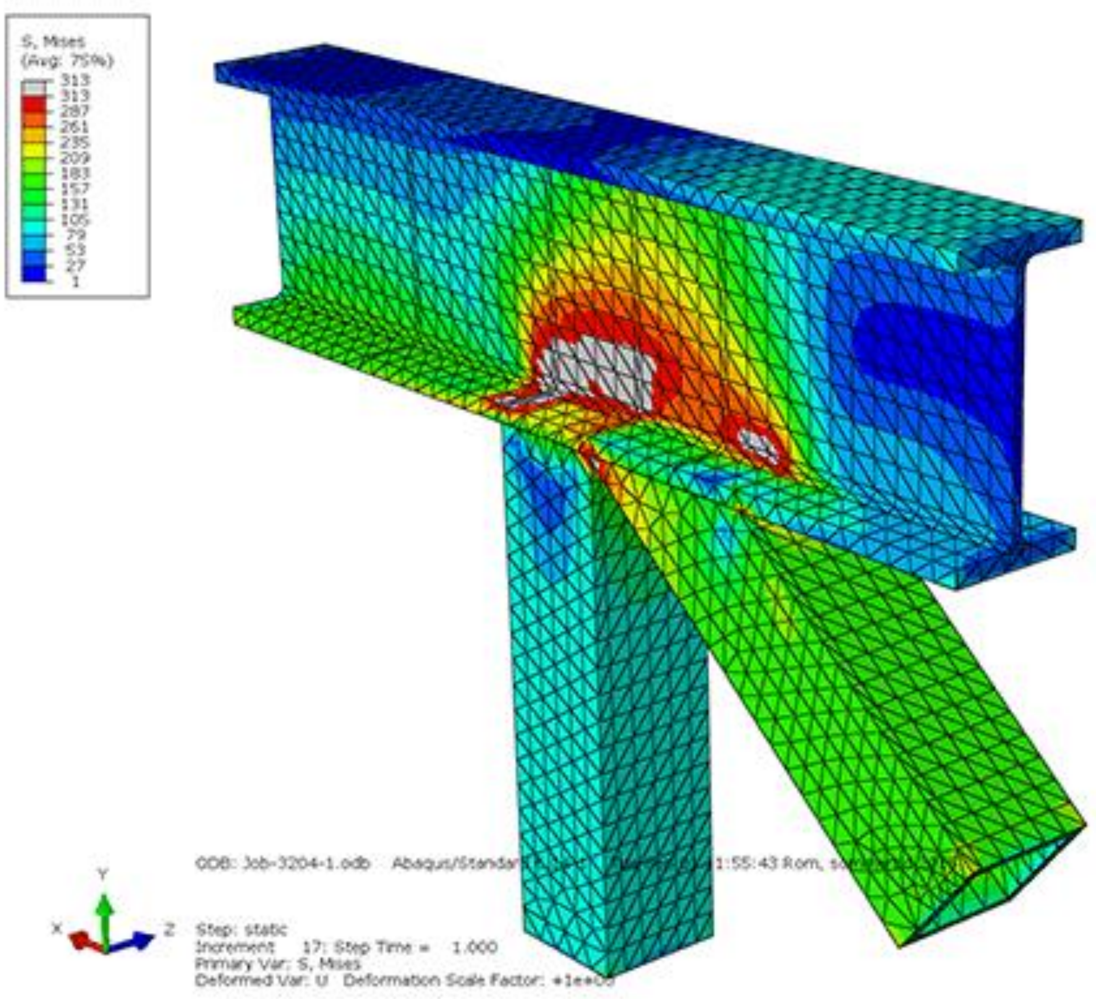
But more importantly,

- It is a **prime example of a PDE that is hard-to-solve**.
- Our understandings of mathematics and PDEs simply are not good enough for solving this type of PDE problems at the moment.
- **Being able to solve Navier-Stokes analytically would require new techniques or insights that could be applied to other challenging problems in mathematics or physics.**

Finite Element Methods and PDEs

Definition (The finite element method – out-of-scope):

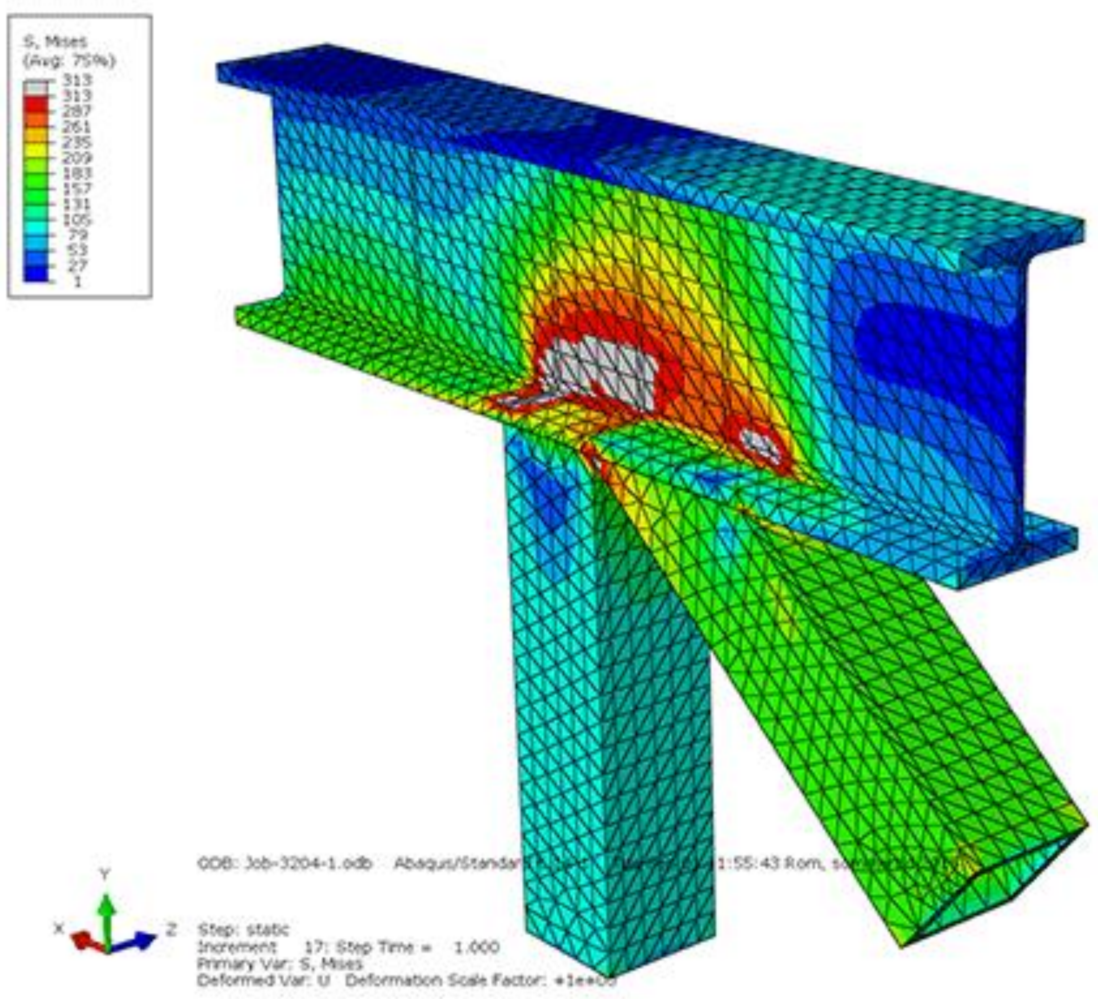
The **Finite Element Method** (**FEM**) is one of the many popular method for numerically solving (i.e. provide an approximation of the analytical solution to) hard-to-solve partial differential equations, for which no analytical solutions have yet been found.



Finite Element Methods and PDEs

How it works (out-of-scope):

- The FEM subdivides a large system into smaller, simpler parts called **finite elements**.
- This is achieved by creating a particular space discretization in the space dimensions, called a **mesh**.
- The method then approximates the "unknown" analytical solution over the domain using **local approximations of derivatives**.



Finite Element Methods and PDEs

While FEM works, it suffers from a few problems:

1. **Complex Geometries**: FEM requires the domain of interest to be discretized into elements. Challenging mesh generation and may require massive computational costs.
2. **High-dimensional Problems**: FEM might become computationally expensive for problems with a high number of dimensions due to a problem called the "curse of dimensionality."
3. **Time dependence issues**: Time-dependent problems might need small time steps for stability, which leads to computational costs.
4. **Parallelization**: While FEM can be parallelized, it often requires sophisticated algorithms and data structures to do so, especially for adaptive meshing in 3D.

Finite Element Methods and PDEs

While FEM works, it suffers from a few problems:

1. **Complex Geometries**: FEM requires the domain of interest to be discretized into elements. Challenging mesh generation and [illegible] require massive computational costs.
2. **High-Dimensional Problems**: FEM might become computationally expensive for problems with a high number of dimensions due to a problem called the "curse of dimensionality."
3. **Time dependence issues**: Time-dependent problems might need small time steps for stability, which leads to [illegible] computational costs.
4. **Parallelization**: While FEM can be parallelized, it often requires sophisticated algorithms and data structures to do so, especially for adaptive meshing in 3D.

Overall, FEM is a powerful method to approximate solutions to PDEs, but it is very expensive to implement, and therefore not fit for many applications (e.g. applications with real-time constraints).

→ Need other solutions!

Neural Networks to the rescue! (again?)

Definition (Physics-Informed Neural Networks):

Introduced in [Karniadakis2021] and [Raissi2019], **Physics-Informed Neural Networks (PINNs)** are a type of universal function approximators based on Neural Networks and Deep Learning.

PINNs can embed the knowledge of any physical laws described by partial differential equations (PDEs).

After training, the PINN can be used to approximate the analytical function g that would be the solution of a given PDE.

Input (t, x) → PINN f trained to approximate the analytical solution g of a given PDE. → Output of NN $f(t, x)$ approximates the analytical solution $g(t, x)$

PINNs vs FEM

PINNs have recently gained interest, because of the following advantages:

1. **Geometry Flexibility**: PINNs do not require a mesh as in FEMs, which can simplify problems with complex geometries.
2. **Universal Approximators**: Neural networks have the capacity to approximate a wide variety of functions (universal approximation theorem! and [Hornik1989]), which can be used to represent complex PDE solutions.
3. **Easily Adaptable**: PINNs can easily be updated or trained on new data, making them adaptable to changing conditions or data-driven problems.
4. **Deep Learning Infrastructure**: The rise of deep learning has led to robust software ecosystems and powerful hardware accelerators, which can be harnessed by PINNs. After training, inference is extremely fast and therefore suitable for applications with real-time constraints.

PINNs vs FEM

PINNs have recently gained interest, because of the following advantages:

1. **Geometry Flexibility**: PINNs do not require a mesh as in FEMs, which can simplify problems with complex geometries.
2. **Universal Approximation**: Neural networks have the capacity to approximate a wide variety of functions (universal approximation theorem!), which can be leveraged to represent complex PDE solutions.
3. **Easily Adaptable**: PINNs can easily be updated or retrained on new data, making them adaptable to changing conditions or data-driven problems.
4. **Deep Learning Infrastructure**: The rise of deep learning has led to robust software ecosystems and powerful hardware accelerators, which can be harnessed by PINNs. After training, inference is extremely fast and therefore suitable for applications with real-time constraints.

Of course, the obvious counterpart is that we need to properly TRAIN the said PINN so that it learns to approximate the analytical solution to the given PDE problem!

And that is not always easy to do!

A toy example of a PDE (Simpler than Navier-Stokes, for obvious reasons!)

A toy example of a PDE

Let us consider this toy example of a PDE.

- Let us denote $T(t)$ the **temperature of my coffee, in °C, at time t**.
- It starts, freshly brewed at 100°C, that is $T(0) = 100$ °C.
  This is the **initial condition** for our PDE.
- The room temperature is constant, and set to $T_{env} = 20$ °C.
- The temperature function $T(t)$ will change over time and its evolution follows the **PDE** below.

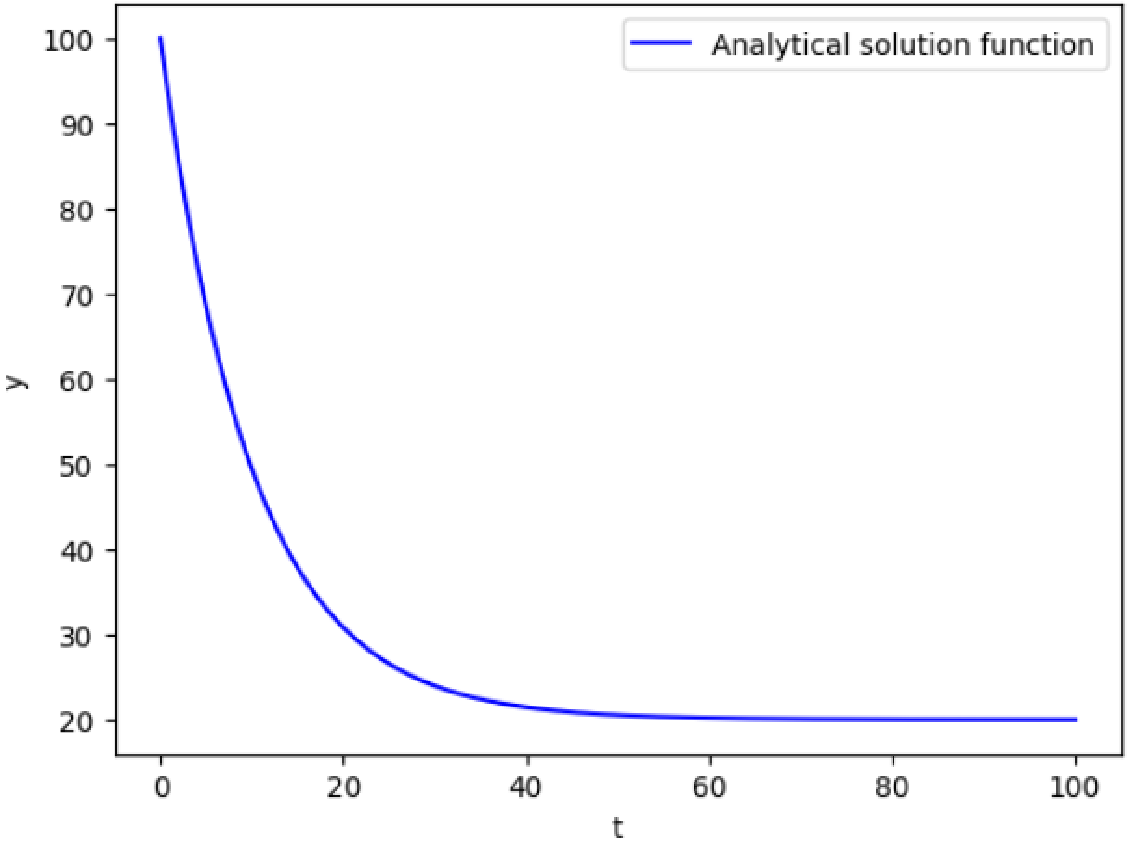
$$\frac{\partial T(t)}{\partial t} = R\big(T_{env} - T(t)\big)$$

- The R constant denotes the temperature dissipation factor (i.e. how fast the temperature changes) and is arbitrarily set to 0.1.

A toy example of a PDE

The PDE problem

$$\frac{\partial T(t)}{\partial t} = R\left(T_{env} - T(t)\right)$$
$$T(0) = T_0$$



Is considered easy-to-solve, as it admits an analytical solution, whose closed-form expression has been known for a while.
It is simply given as:

$$T(t) = T_{env} + (T_0 - T_{env})\exp(-Rt)$$

Defining a class for our PINN

We will train a PINN to solve our toy example PDE.

Input t → PINN f trained to approximate the analytical solution g of toy example PDE. → Output of NN $f(t,x)$ approximates the analytical solution $g(t,x)$

Defining a class for our PINN

We will train a PINN to solve our toy example PDE.

- Define a nn.module class,
- Consists of a few linear layers, with ReLU activation functions,
- And a forward method for inference.

A rather simple model compared to what we have done in the previous lectures.

```
class LinearNN(nn.Module):S
    def __init__(self, num_layers = 5, num_neurons = 128):
        # Start with attributes.
        super().__init__()
        self.num_neurons = num_neurons
        self.num_layers = num_layers

        # Build layers.
        layers = []
        # First layer.
        layers.append(nn.Linear(1, self.num_neurons))
        # Hidden layers will consist of Linear layers and some activation.
        for _ in range(self.num_layers):
            layers.append(nn.Linear(self.num_neurons, self.num_neurons))
            layers.append(nn.ReLU())
        # Finish with one output layer, which is simply a linear and has no activation.
        layers.append(nn.Linear(self.num_neurons, 1))

        # Build the network as a Sequential object of the layers.
        self.network = nn.Sequential(*layers)

    def forward(self, x):
        # Forward method made simple, using the Sequential object from earlier.
        return self.network(x.reshape(-1, 1)).squeeze()
```

```
model = LinearNN().to(device)
```

How to train the PINN?

How to train the PINN to approximate the solution of the given PDE?

- No dataset available.
- This PDE admits an analytical solution, but if we play fair, we should assume we do not know about this solution.
- The only information we have is the PDE itself.

$$\frac{\partial T(t)}{\partial t} = R\left(T_{env} - T(t)\right)$$
$$T(0) = T_0$$

→ Need to come up with a custom loss function, called a physics loss function, to quantify how good the model is at solving the PDE above!

Physics loss function

Definition (The physics loss function):

The **physics loss function** is a custom loss function, which can be used to quantify how good a given PINN model is at solving a given PDE. It requires no dataset, only information about the given PDE.

It typically consists of two parts:

- **A PDE loss function:** quantifies how good the model is at solving the given PDE, with partial derivatives, i.e. $\frac{\partial T(t)}{\partial t} = R\left(T_{env} - T(t)\right)$.
- **An initial condition loss function:** quantifies whether the model satisfies the given initial conditions for the PDE, i.e. $T(0) = T_0$.

These two losses components are then **assembled into a physics loss function**, which is then used to train the PINN.

Restricted

Part 1: The PDE loss

Important observation: If the PINN model f is supposed to act as the solution to the PDE, then the partial derivative of f with respect to t is simply the gradient of the model with respect to its inputs!

This derivative can be computed using **autograd (very nice!), along with the inputs and outputs produced by the model!**

```
# This function will compute gradients with respect to inputs,
# for the given forward pass that produced the respective outputs.
def grad_fun(outputs, inputs):
    return torch.autograd.grad(outputs, inputs, grad_outputs = torch.ones_like(outputs), create_graph = True)
```

```
model_f = LinearNN().to(device)
deriv_f = grad_fun(model_f(t), t)[0]
```

Part 1: The PDE loss

Next, we can

- Define an array t of time values, using a linspace of some sort.
- Compute the outputs $f(t)$ for said time values, using our PINN model,
- Compute the derivatives values $\frac{\partial f(t)}{\partial t}$ using our autograd function for said times values,
- And then check if the PDE equation $\frac{\partial T(t)}{\partial t} = R\left(T_{env} - T(t)\right)$ is met for said time values!

Part 1: The PDE loss

- **Step 2:** Define our linspace array of time values t (here values of t will range from 0 to 100, with 1000 points in array). So $t = (t_1, \dots t_{1000})$

```
# This is the custom loss function we plan to use for the model
def physics_loss(model):
    # Generate x tensor by drawing with a linspace
    t = torch.linspace(0, 100, steps = 1000).view(-1, 1).requires_grad_(True).to(device)
```

Part 1: The PDE loss

- **Step 3:** Forward pass in the PINN model to compute $f(t)$ for all values in the linspace array t.

```
# This is the custom loss function we plan to use for the model
def physics_loss(model):
    # Generate x tensor by drawing with a linspace
    t = torch.linspace(0, 100, steps = 1000).view(-1, 1).requires_grad_(True).to(device)

    # Forward pass
    y = model(t)
```

Part 1: The PDE loss

- **Step 4:** Use autograd to compute $\frac{\partial f(t)}{\partial t}$ for all values in the array t.

```
# This is the custom loss function we plan to use for the model
def physics_loss(model):
    # Generate x tensor by drawing with a linspace
    t = torch.linspace(0, 100, steps = 1000).view(-1, 1).requires_grad_(True).to(device)

    # Forward pass
    y = model(t)
    # Compute the gradient manually
    dy = grad_fun(y, t)[0]
```

Part 1: The PDE loss

- **Step 5:** Compute $\frac{\partial f(t)}{\partial t} - R\left(T_{env} - f(t)\right)$ for all values in the array t. Ideally, we want this quantity to be as close to 0 as possible.

```python
# This is the custom loss function we plan to use for the model
def physics_loss(model):
    # Generate x tensor by drawing with a linspace
    t = torch.linspace(0, 100, steps = 1000).view(-1, 1).requires_grad_(True).to(device)

    # Forward pass
    y = model(t)
    # Compute the gradient manually
    dy = grad_fun(y, t)[0]

    # Compute the loss
    # Val1: checks if the model fits the PDE
    # Val2: checks if the model fits the initial condition
    # Lambda_coeffs: serves the same purpose as in regularization, to indicate
    # the importance of one aspect of the loss (e.g. val1) wrt. the second part (e.g. val2).
    # Note: technically, lambda_coeff is an hyperparameter
    # and we should investigate different values!
    right_hand_side_pde = (R_coeff*(T_env - y)).view(dy.shape)
```

Part 1: The PDE loss

- **Step 6:** To do so, we will attempt to minimize the loss V_1, below.

$$V_1 = \frac{1}{N}\sum_{i=1}^{N}\left(\frac{\partial f(t_i)}{\partial t} - R\left(T_{env} - f(t_i)\right)\right)^2$$

```
# This is the custom loss function we plan to use for the model
def physics_loss(model):
    # Generate x tensor by drawing with a linspace
    t = torch.linspace(0, 100, steps = 1000).view(-1, 1).requires_grad_(True).to(device)

    # Forward pass
    y = model(t)
    # Compute the gradient manually
    dy = grad_fun(y, t)[0]

    # Compute the loss
    # Val1: checks if the model fits the PDE
    # Val2: checks if the model fits the initial condition
    # Lambda_coeffs: serves the same purpose as in regularization, to indicate
    # the importance of one aspect of the loss (e.g. val1) wrt. the second part (e.g. val2).
    # Note: technically, lambda_coeff is an hyperparameter
    # and we should investigate different values!
    right_hand_side_pde = (R_coeff*(T_env - y)).view(dy.shape)
    val1 = torch.mean((dy - right_hand_side_pde)**2)
```

Part 2: The initial condition loss

The initial condition loss is much simpler to compute.

- **Step 1:** Forward pass on the model with a single value corresponding to our initial condition. In pour case, that is $t = 0$, and we get $f(0)$.
- **Step 2:** Check if $f(0)$ matches the initial temperature T_0.
- **Step 3:** To encourage the model to meet this initial condition, we will define the loss V_2 as $V_2 = (f(0) - T_0)^2$. Minimizing V_2 and bringing it to 0 is equivalent to having the model satisfy the initial condition!

```
X_0 = torch.tensor([0.0]).view(-1, 1).requires_grad_(True).to(device)
Y_0 = model(X_0)
GT_0 = sol_fun_torch(X_0).view(Y_0.shape)
val2 = torch.mean((Y_0 - GT_0)**2)
```

Part 3: Assembling the physics loss

We have two losses V_1 and V_2, that must both be brought to a minimum for the model to act as a good approximator for the analytical solution of the PDE.

Let us assemble them into a **physics loss** function L.

$$L = 1 + \lambda_1 V_1 + \lambda_2 V_2$$

Here λ_1 and λ_2 are hyperparameters, weighting the contributions of V_1 and V_2 respectively. We will arbitrarily set them to 1 for now.

The constant 1 value in L, only serves to prevent the loss function to go to zero and make later visuals easier to read, as our training curves will be shown in logarithmic scale later on. It could be removed.

```python
# This is the custom loss function we plan to use for the model
def physics_loss(model):
    # Generate x tensor by drawing with a linspace
    t = torch.linspace(0, 100, steps = 1000).view(-1, 1).requires_grad_(True).to(device)

    # Forward pass
    y = model(t)
    # Compute the gradient manually
    dy = grad_fun(y, t)[0]

    # Compute the loss
    # Val1: checks if the model fits the PDE
    # Val2: checks if the model fits the initial condition
    # Lambda_coeffs: serves the same purpose as in regularization, to indicate
    # the importance of one aspect of the loss (e.g. val1) wrt. the second part (e.g. val2).
    # Note: technically, lambda_coeff is an hyperparameter
    # and we should investigate different values!
    right_hand_side_pde = (R_coeff*(T_env - y)).view(dy.shape)
    val1 = torch.mean((dy - right_hand_side_pde)**2)
    X_0 = torch.tensor([0.0]).view(-1, 1).requires_grad_(True).to(device)
    Y_0 = model(X_0)
    GT_0 = sol_fun_torch(X_0).view(Y_0.shape)
    val2 = torch.mean((Y_0 - GT_0)**2)
    lambda_coeff1 = 1
    lambda_coeff2 = 1
    # Return assembled loss value
    # Also returns val1 and val2 for visualization
    return 1 + lambda_coeff1*val1 + lambda_coeff2*val2, val1, val2
```

Training the PINN

Training the PINN is almost the same procedure as before.

- Forward pass on model (happens in the physics loss function),
- Compute V_1, V_2 and L. as explained earlier.
- Do note that the physics loss does not require a dataset,
- Backprop on L as before, using autograd.
- Using basic Adam optimizer as before (how original!).
- Learning rate decay.
- Keep track of values for training curves later.

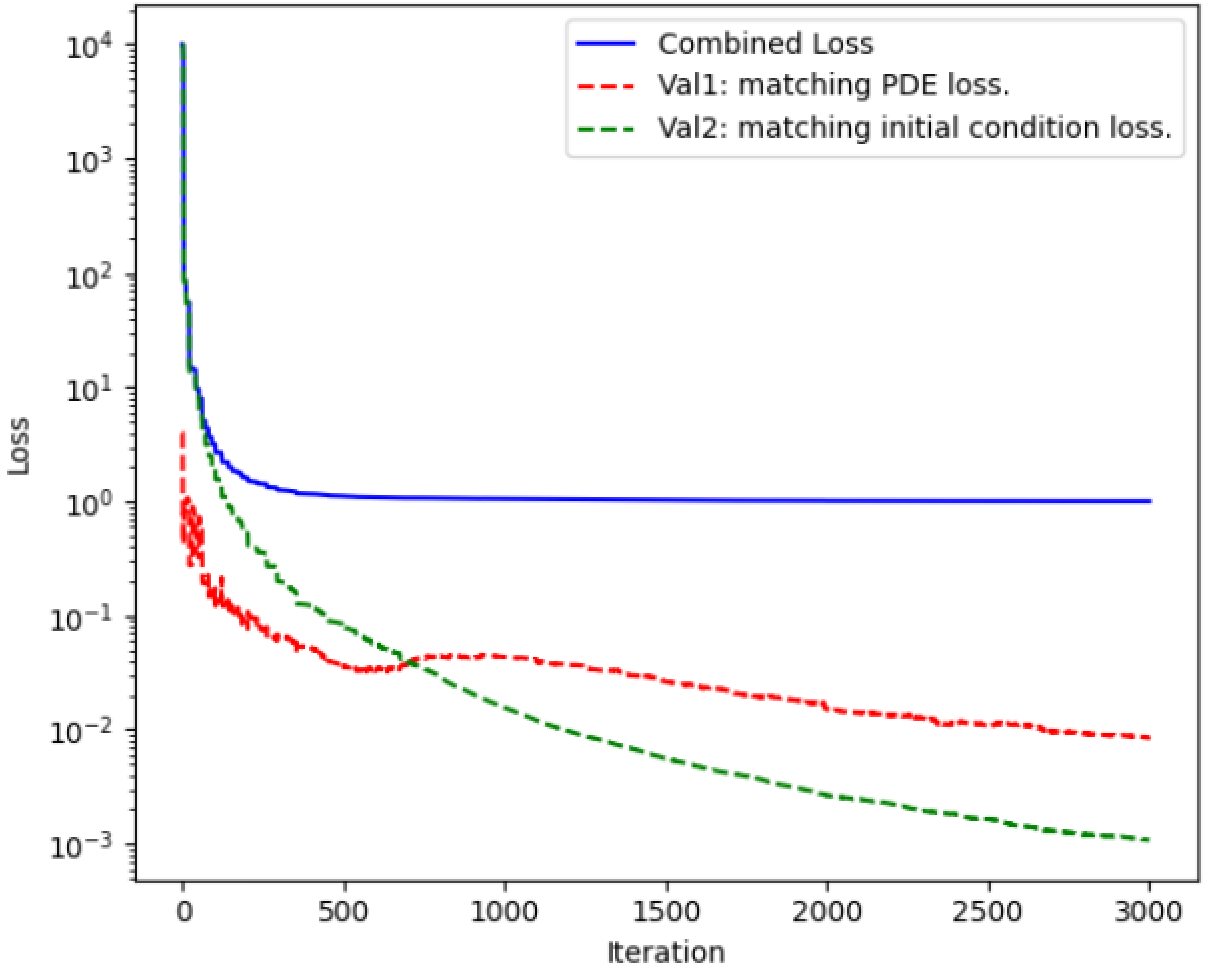
```
# Choose parameters for the training, as before
num_iter = 30001
learning_rate = 1e-3
```

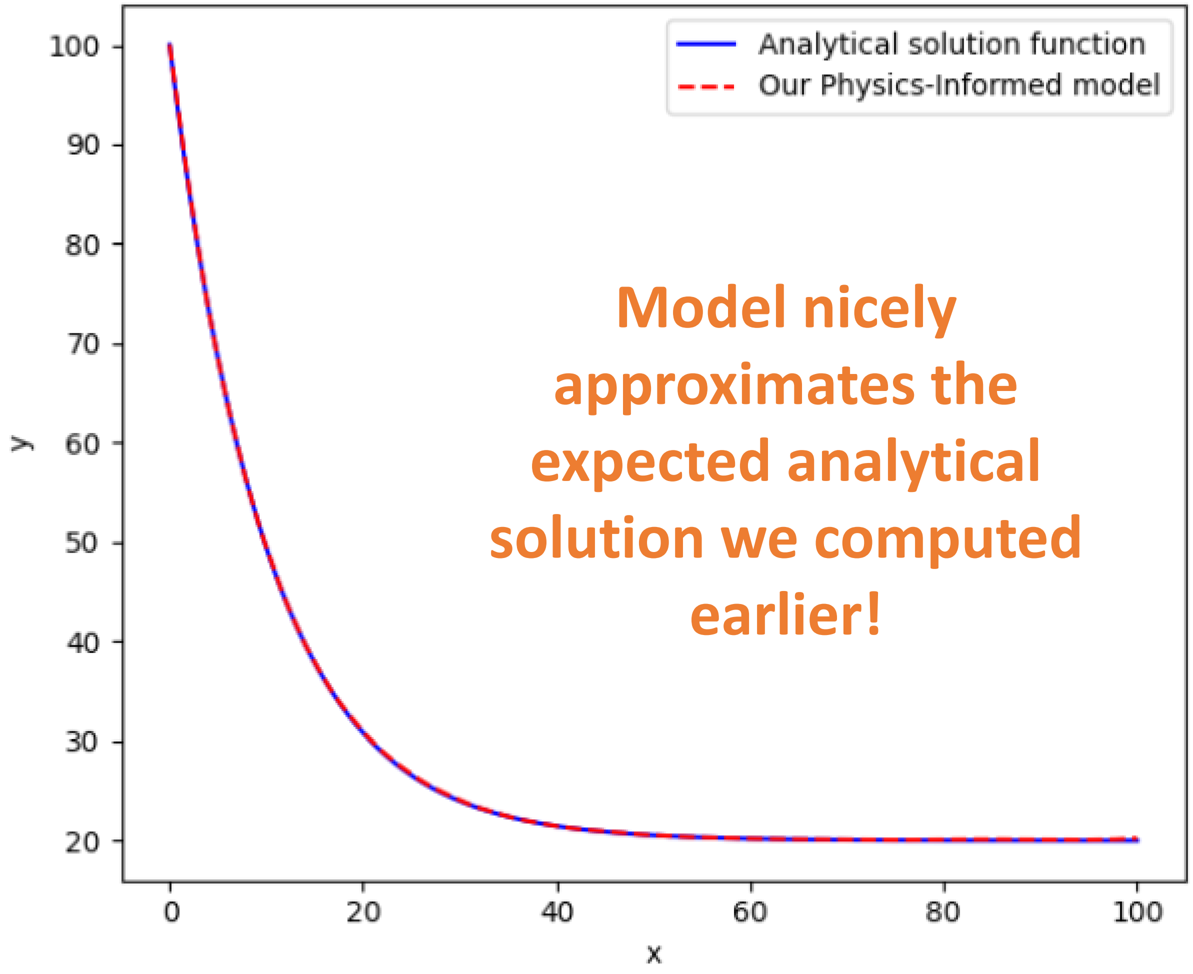
```
### Training Loop
# Reset model
model = LinearNN().to(device)
# Keep track of loss values
loss, val1, val2 = physics_loss(model)
loss_values, val1_values, val2_values = [loss.item()], [val1.item()], [val2.item()]
for i in range(num_iter):
    # Learning rate adjustment (using a decay based on iterations)
    decay = 1 + 10*i//1000
    optimizer = torch.optim.Adam(model.parameters(), lr = learning_rate/decay)
    # Forward pass: Compute the loss
    loss, val1, val2 = physics_loss(model)
    # Zero the gradients
    optimizer.zero_grad()
    # Backward pass: Compute gradient of the loss with respect to model parameters
    loss.backward()
    # Update parameters
    optimizer.step()
    if i % 10 == 0:
        # Add to list every 10 iterations
        loss_values.append(loss.item())
        val1_values.append(val1.item())
        val2_values.append(val2.item())
        if i % 500 == 0:
            # Print progress every 500 iterations
            print(f"Iteration {i}, Combined Loss: {loss.item()}, PDE part: {val1.item()}, Init Cond. part: {val2.item()}")
```

It trains? (V_1 and V_2 go to 0, L goes to 1)

```
Iteration 0, Combined Loss: 10018.4912109375, PDE part: 4.019221782684326, Init Cond. part: 10013.4716796875
Iteration 500, Combined Loss: 9.718744277954102, PDE part: 0.3239942789077759, Init Cond. part: 8.394749641418457
Iteration 1000, Combined Loss: 3.2027595043182373, PDE part: 0.14855265617370605, Init Cond. part: 2.0542068481445312
Iteration 1500, Combined Loss: 2.00662899017334, PDE part: 0.11113713681697845, Init Cond. part: 0.895491898059845
Iteration 2000, Combined Loss: 1.621795415878296, PDE part: 0.09079346060752869, Init Cond. part: 0.5310018658638
Iteration 2500, Combined Loss: 1.4340479373931885, PDE part: 0.07675463706254959, Init Cond. part: 0.3572933077812195
Iteration 3000, Combined Loss: 1.266410231590271, PDE part: 0.06670477241277695, Init Cond. part: 0.1997054517269134 5
Iteration 3500, Combined Loss: 1.2225223779678345, PDE part: 0.05808401852846145 6, Init Cond. part: 0.16443832218647003
Iteration 4000, Combined Loss: 1.173896074295044, PDE part: 0.05169065669178963, Init Cond. part: 0.12220537662506104
Iteration 4500, Combined Loss: 1.1424450874328613, PDE part: 0.04110698401927948, Init Cond. part: 0.10133811831474304
Iteration 5000, Combined Loss: 1.118298053741455, PDE part: 0.03650464490056038, Init Cond. part: 0.0817934200167656
Iteration 5500, Combined Loss: 1.0993268489837646, PDE part: 0.033071864396333694, Init Cond. part: 0.06625502556562424
Iteration 6000, Combined Loss: 1.0892524719238281, PDE part: 0.03310181945562363, Init Cond. part: 0.05615068972110748
Iteration 6500, Combined Loss: 1.0838000774383545, PDE part: 0.03489473834633827, Init Cond. part: 0.04890539124608 04
Iteration 7000, Combined Loss: 1.0764151811599731, PDE part: 0.036936480551958084, Init Cond. part: 0.03947863727807999
Iteration 7500, Combined Loss: 1.0767420530319214, PDE part: 0.04211590066552162, Init Cond. part: 0.03462611511349678
Iteration 8000, Combined Loss: 1.0734248161315918, PDE part: 0.04381300508975983, Init Cond. part: 0.02961186878383159 6
Iteration 8500, Combined Loss: 1.068050742149353, PDE part: 0.04402134194970131, Init Cond. part: 0.02402935177087783 8
Iteration 9000, Combined Loss: 1.0642602443695068, PDE part: 0.04357993975281715 4, Init Cond. part: 0.020680297166109085
Iteration 9500, Combined Loss: 1.0623345375061035, PDE part: 0.044502366334199905, Init Cond. part: 0.017832208424806595
Iteration 10000, Combined Loss: 1.0591691732406616, PDE part: 0.043588023632764816, Init Cond. part: 0.015581161715090275
Iteration 10500, Combined Loss: 1.056633710861206, PDE part: 0.04292408376932144, Init Cond. part: 0.013709674589335918
```

It trains? (V_1 and V_2 go to 0, L goes to 1)





Feature engineering in PINNs

When it comes to PINN, **feature engineering is king**.

- If you suspect that the analytical solution has a certain behavior, for instance exponential, cosine, logarithm, etc.
- Then you can perform feature engineering and provide additional inputs to the PINN.
- For instance, let us pretend that we strongly suspect that the analytical solution of our PDE has a negative exponential behavior.
- We will rework our model so that it takes two inputs, instead of just one, being the time t and $\exp(-t)$.
- Also, amend the physics loss function accordingly.

```python
class LinearNN(nn.Module):
    def __init__(self, num_layers = 5, num_neurons = 128):
        # Start with attributes.
        super().__init__()
        self.num_neurons = num_neurons
        self.num_layers = num_layers

        # Build layers.
        layers = []
        # First layer.
        layers.append(nn.Linear(2, self.num_neurons))
        # Hidden layers will consist of Linear layers and some activation.
        for _ in range(self.num_layers):
            layers.append(nn.Linear(self.num_neurons, self.num_neurons))
            layers.append(nn.ReLU())
        # Finish with one output layer, which is simply a linear and has no activation.
        layers.append(nn.Linear(self.num_neurons, 1))

        # Build the network as a Sequential object of the layers.
        self.network = nn.Sequential(*layers)

    def forward(self, x):
        # Forward method made simple, using the Sequential object from earlier.
        return self.network(x).squeeze()
```

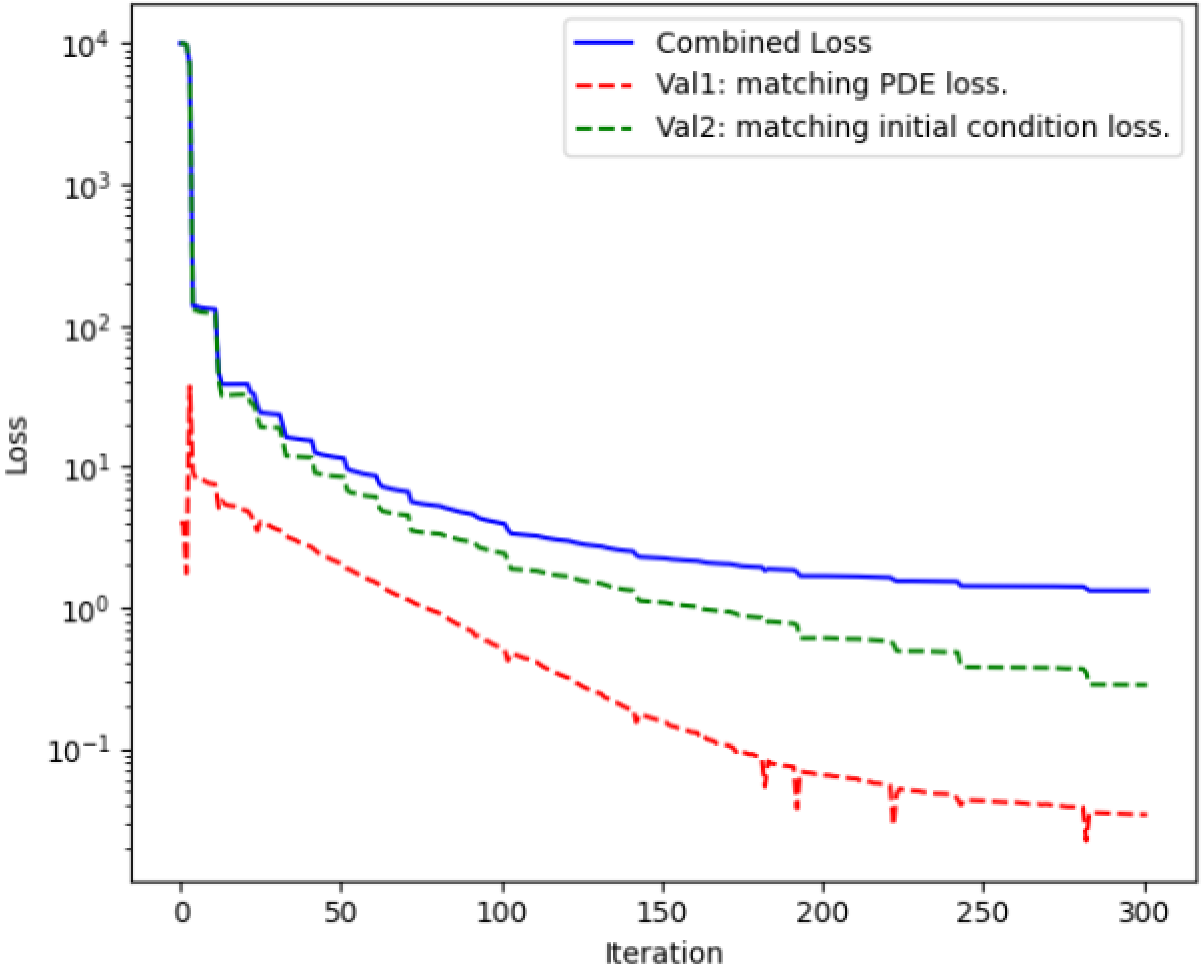
```
# This is the custom loss function we plan to use for the model
def physics_loss(model):
    # Generate x tensor by drawing with a linspace
    t = torch.linspace(0, 100, steps = 1000).view(-1, 1).requires_grad_(True).to(device)
    exp_neg_t = torch.exp(-t)
    input_tensor = torch.cat((t, exp_neg_t), dim = 1)
    # Forward pass
    y = model(input_tensor)
    # Compute the gradient manually
    dy = grad_fun(y, t)[0]
    # Compute the loss
    right_hand_side_pde = (R_coeff*(T_env - y)).view(dy.shape)
    val1 = torch.mean((dy - right_hand_side_pde)**2)
    X_0 = torch.tensor([0.0]).view(-1, 1).requires_grad_(True).to(device)
    exp_neg_X_0 = torch.exp(-X_0)
    input_tensor_0 = torch.cat((X_0, exp_neg_X_0), dim=1)
    Y_0 = model(input_tensor_0)
    GT_0 = sol_fun_torch(X_0).view(Y_0.shape)
    val2 = torch.mean((Y_0 - GT_0)**2)
    lambda_coeff1 = 1
    lambda_coeff2 = 1
    # Return assembled loss value
    # Also returns val1 and val2 for visualization
    return 1 + lambda_coeff1*val1 + lambda_coeff2*val2, val1, val2
```

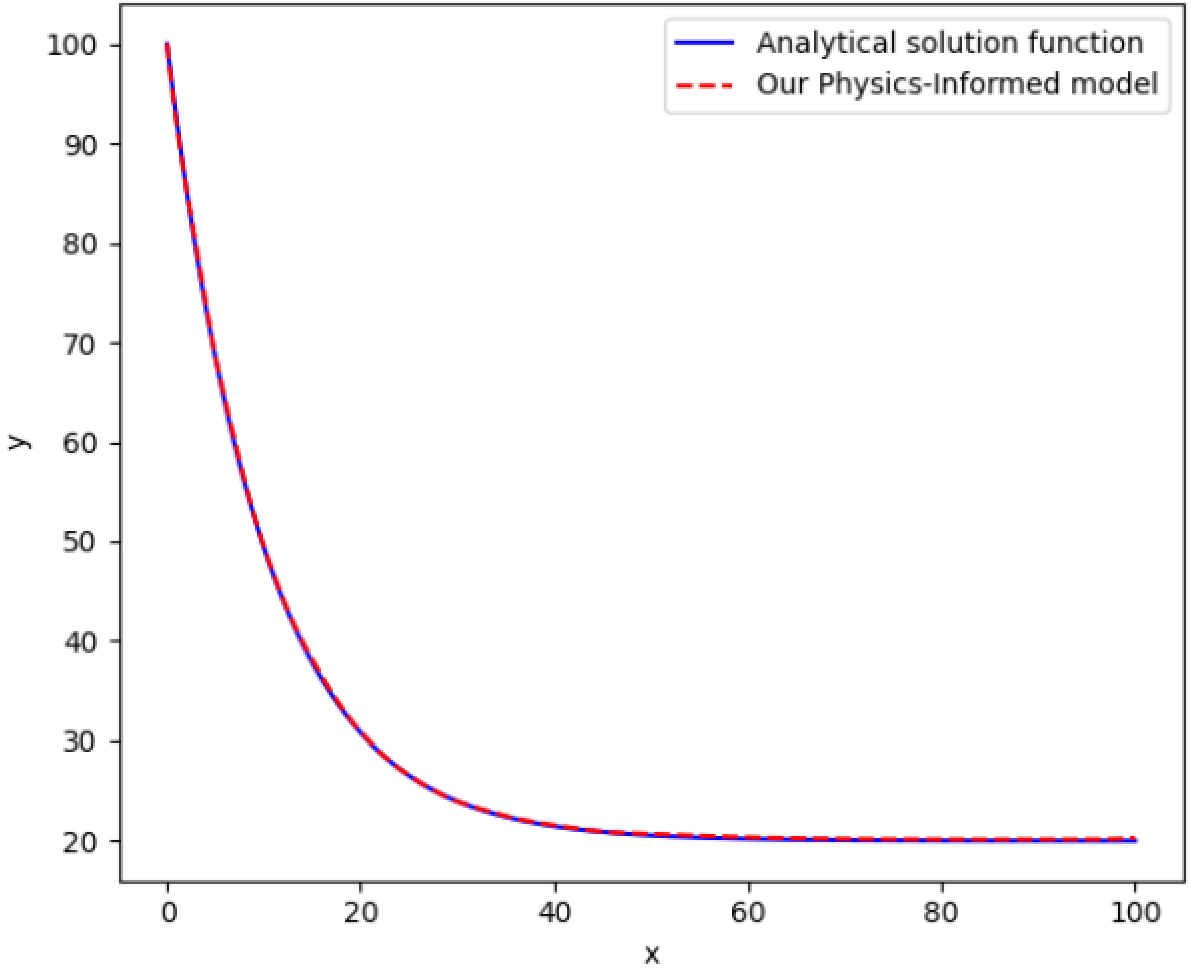
Amend both forward calls accordingly (forward pass for PDE loss calculation and forward pass for initial condition loss calculation)

```python
# This is the custom loss function we plan to use for the model
def physics_loss(model):
    # Generate x tensor by drawing with a linspace
    t = torch.linspace(0, 100, steps = 1000).view(-1, 1).requires_grad_(True).to(device)
    exp_neg_t = torch.exp(-t)
    input_tensor = torch.cat((t, exp_neg_t), dim = 1)
    # Forward pass
    y = model(input_tensor)
    # Compute the gradient manually
    dy = grad_fun(y, t)[0]
    # Compute the loss
    right_hand_side_pde = (R_coeff*(T_env - y)).view(dy.shape)
    val1 = torch.mean((dy - right_hand_side_pde)**2)
    X_0 = torch.tensor([0.0]).view(-1, 1).requires_grad_(True).to(device)
    exp_neg_X_0 = torch.exp(-X_0)
    input_tensor_0 = torch.cat((X_0, exp_neg_X_0), dim=1)
    Y_0 = model(input_tensor_0)
    GT_0 = sol_fun_torch(X_0).view(Y_0.shape)
    val2 = torch.mean((Y_0 - GT_0)**2)
    lambda_coeff1 = 1
    lambda_coeff2 = 1
    # Return assembled loss value
    # Also returns val1 and val2 for visualization
    return 1 + lambda_coeff1*val1 + lambda_coeff2*val2, val1, val2
```

Important note: Gradients are only calculated wrt to t, not $\exp(-t)$!

Training loop does not change, but much faster (3000 iterations instead of 30000!)





Restricted

More advanced stuff on PINNs

This PINN approach can be complexified by

- Taking into account **different types of boundary/initial conditions**.

(Some of them might require to use the autograd function to calculate partial derivatives but we have seen how to do that!)

Table 1: Boundary Condition Types.

Condition		Kind
$u = 0$	homogeneous	Dirichlet or first kind
$u = f(\vec{x}, t)$	non homogeneous	
$\dfrac{\partial u}{\partial \vec{n}} = 0$	homogeneous	Neumann or second kind
$\dfrac{\partial u}{\partial \vec{n}} = g(\vec{x}, t)$	non homogeneous	
$au + b\dfrac{\partial u}{\partial \vec{n}} = 0$	homogeneous	Robin or third kind
$au + b\dfrac{\partial u}{\partial \vec{n}} = h(\vec{x}, t)$	non homogeneous	

More advanced stuff on PINNs

This PINN approach can be complexified by

- Using higher order and/or mixed derivatives.

(If the PDE requires it, use the grad_fun multiple times to compute higher order derivatives!)

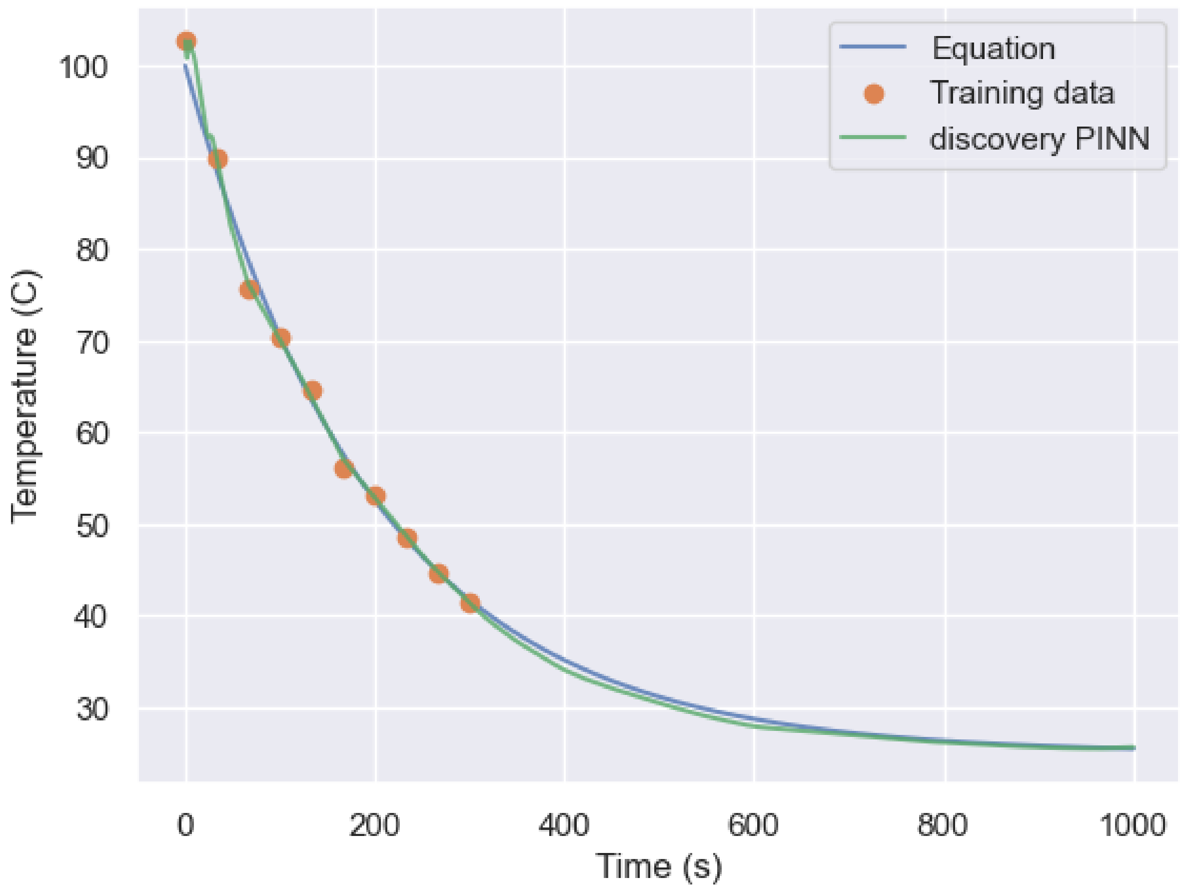
```
# This function will compute gradients with respect to inputs,
# for the given forward pass that produced the respective outputs.
def grad_fun(outputs, inputs):
    return torch.autograd.grad(outputs, inputs, grad_outputs=torch.ones_like(outputs), create_graph=True)

# First derivative
grad_model = grad_fun(model(inputs), inputs)[0]

# Second derivative
second_derivative = grad_fun(grad_model, inputs)[0]
```

More advanced stuff on PINNs

This PINN approach can be complexified by

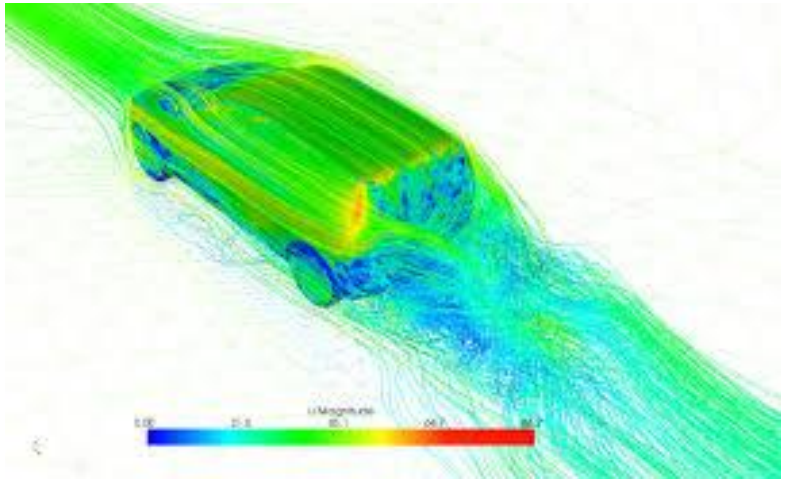
- Using a dataset!

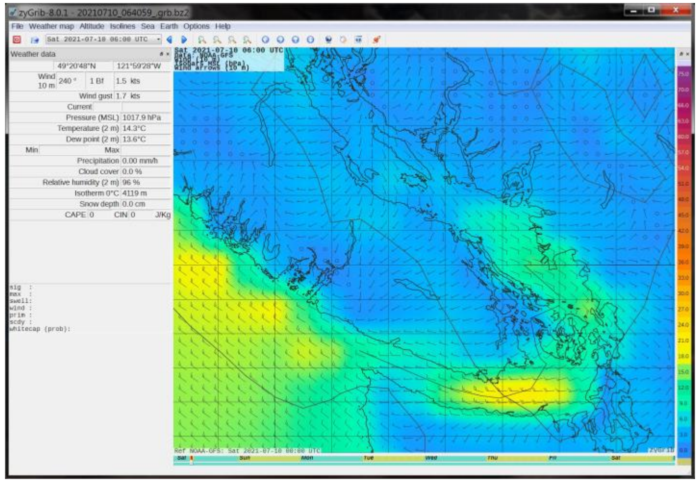
(If you know some values of the analytical function in certain locations, not just the initial conditions, you may use that to help the model train! It will simply require to amend the V_2 definition in the physics loss function to include these additional points!)

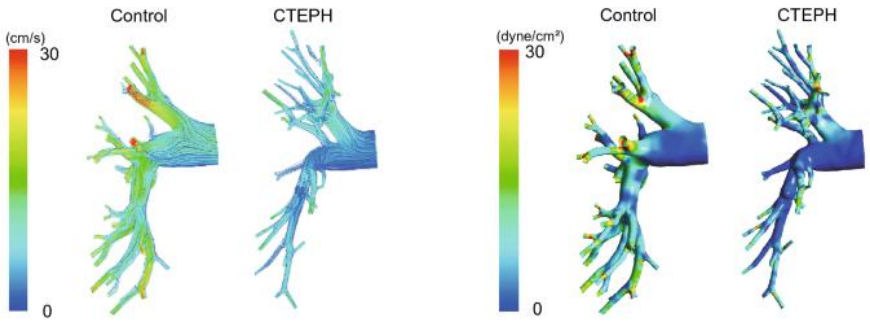


Applications of PINNs

- **Engineering and Fluid Dynamics:** Solving Navier-Stokes equations for simulating fluid flows. Applications in designing aircraft, ships, and automobiles. Modelling heat transfer and energy systems.
- **Environmental Science:** Predicting climate patterns and weather forecasting. Modelling ocean currents and wave dynamics. Groundwater flow and pollutant transport.
- **Medicine and Biology:** Simulating blood flow in arteries (hemodynamics). Modelling tumor growth and drug transport. Bio-mechanics of tissues and organs.

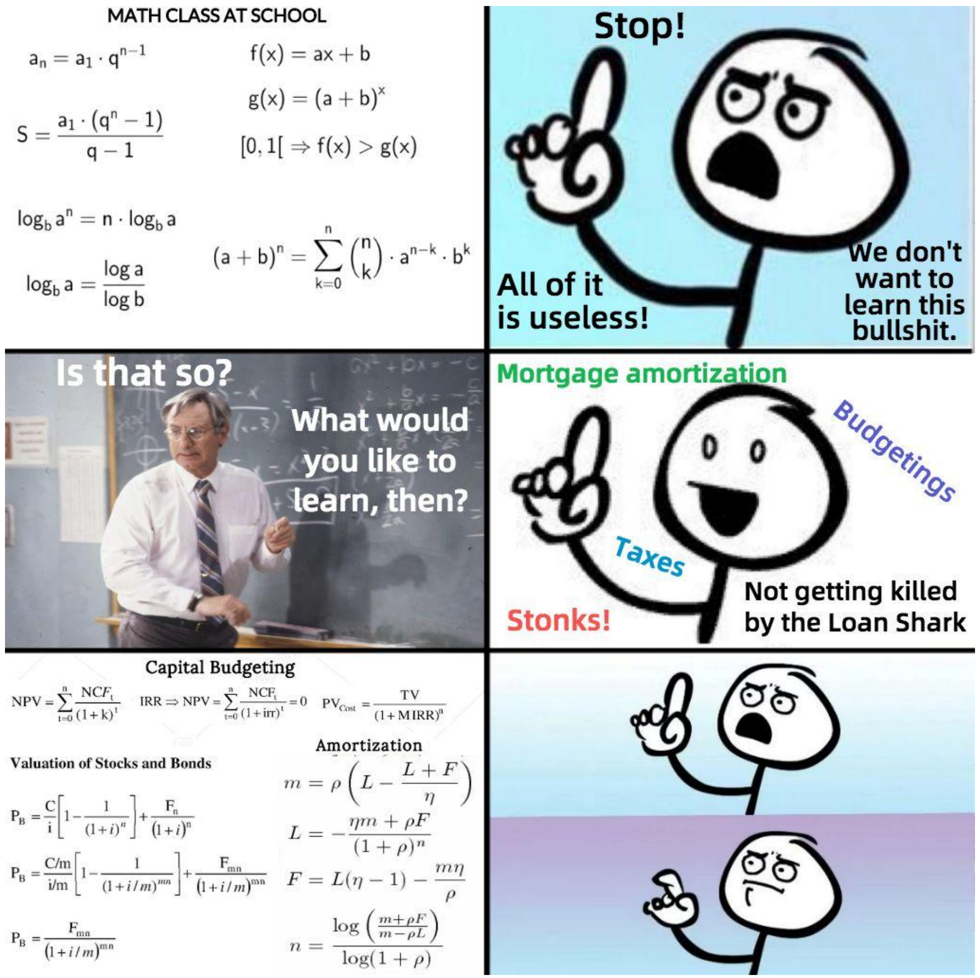


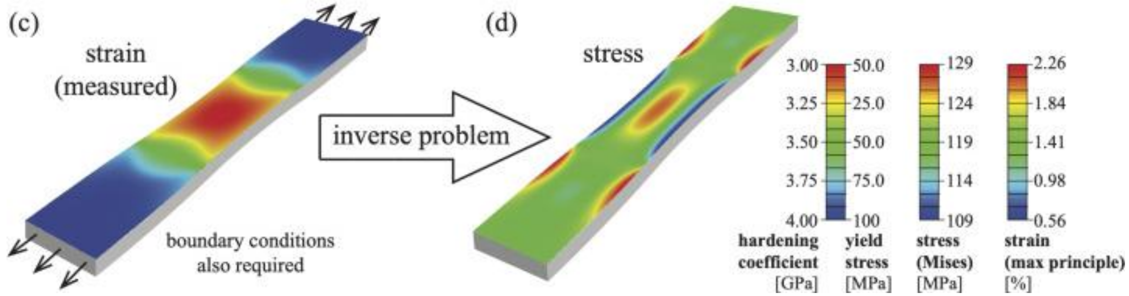




Applications of PINNs

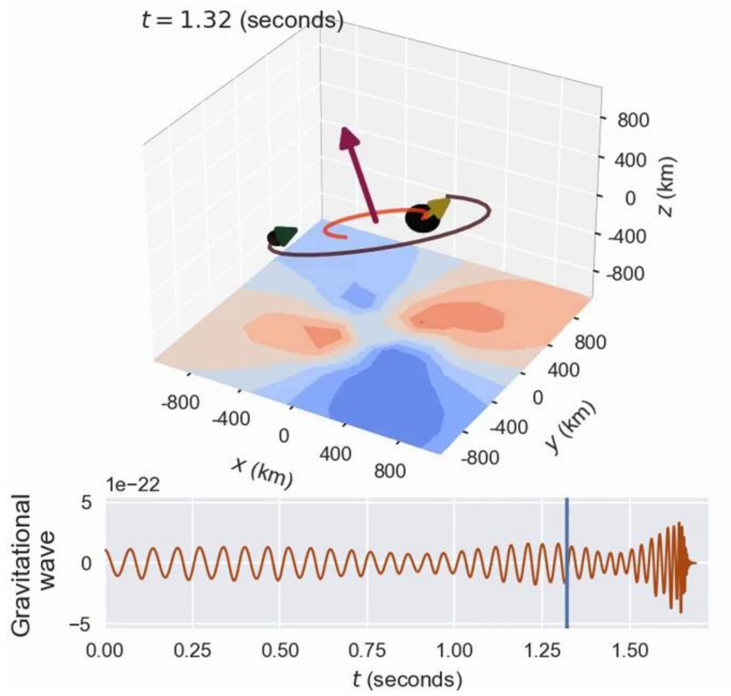
- **Finance:** Solving partial differential equations like the Black-Scholes. Pricing complex financial derivatives. Risk management and portfolio optimization.
- **Structural Mechanics:** Analyzing stress-strain relationships in materials. Solving elasticity and plasticity equations. Simulating failure mechanics and crack propagation.

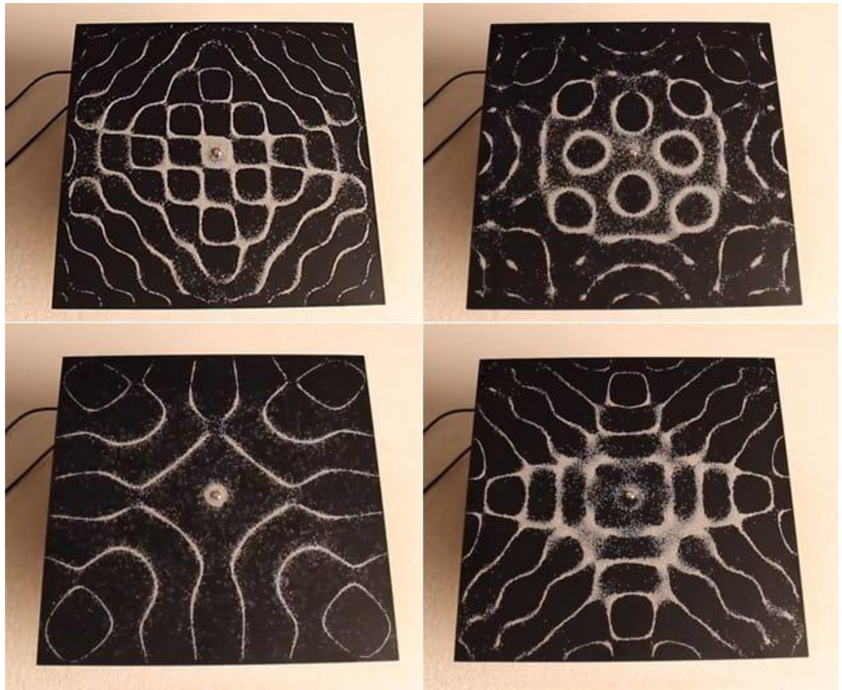




Applications of PINNs

- **Astrophysics and Cosmology:** Modeling gravitational waves and black hole dynamics. Studying the evolution of the universe using Einstein's field equations.
- **Quantum Mechanics:** Solving the Schrödinger equation for complex quantum systems. Modeling quantum transport and materials.
- **Electromagnetics and acoustics:** Simulating electromagnetic and sound wave propagation. Optimizing wireless communication and acoustics.





Conclusion

1. What is a **Partial Differential Equation (PDE)** and what are **typical uses** of PDEs?
2. Why are PDEs **hard to solve** in practice?
3. What are **Physics-Informed Neural Networks (PINNs)**?
4. What is a **Physics Loss** function and how to write a custom one?
5. How to **train a PINN to solve a given PDE**?
6. Is **Feature Engineering** a good thing to have in PINNs?
7. What are **interesting applications** for PINNs?

Learn more about these topics

Out of class, for those of you who are curious

- [Raissi2019] M. Raissi, P. Perdikaris, G.E. Karniadakis, “Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations”, 2019.
- [Karniadakis2021] G. E. Karniadakis, et al., “Physics informed machine learning”, 2021.
- [Hornik1989] K. Hornik, M. Stinchcombe and H. White, “Multilayer feedforward networks are universal approximators”, 1989.

Learn more about these topics

Out of class, for those of you who are curious

- A good tutorial on PINNs by Ben Moseley, “So, what is a physics-informed neural network?”, 2021.

https://benmoseley.blog/my-research/so-what-is-a-physics-informed-neural-network/

- Predicting the market by resolving the Black-Scholes-Merton PDE using Machine Learning? https://www.youtube.com/watch?v=A5w-dEgIU1M