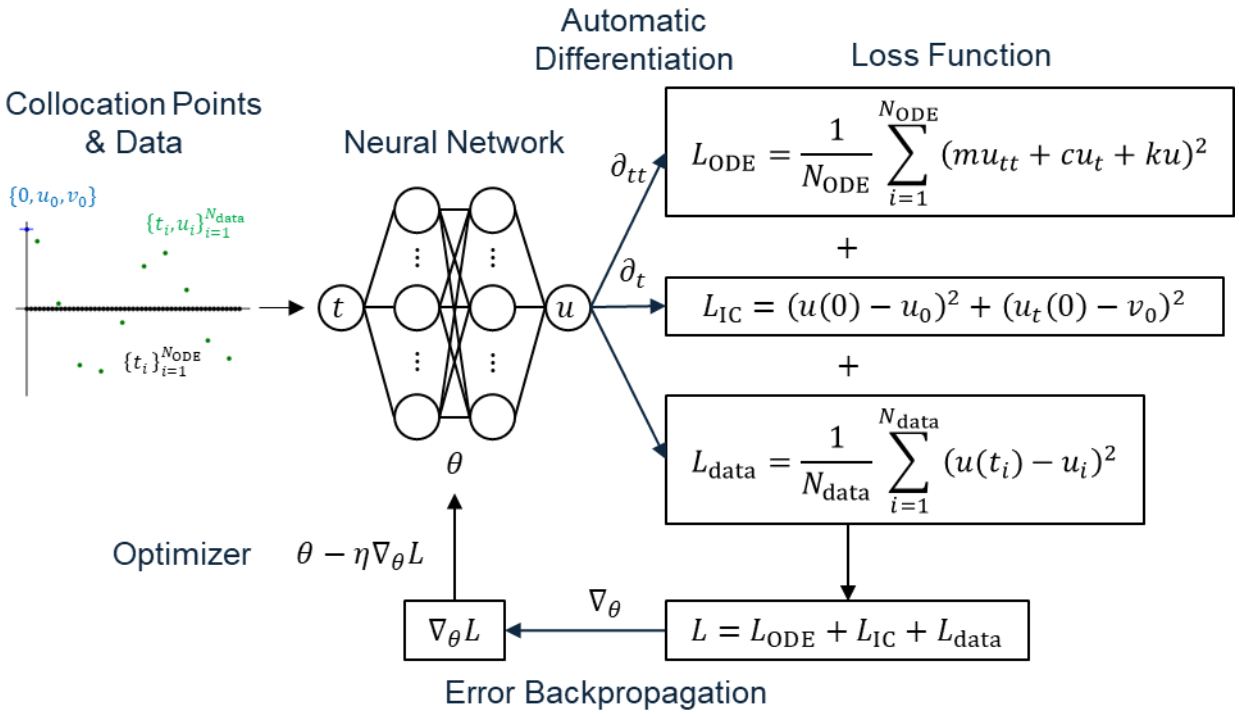
# Physics Informed Neural Network (PINN)

# PyTorch 基本プログラミング




岡﨑　智久

2025 年 9 月 10 日版

# 目次


はじめに 3

## 理論編 4

1. NN と PINN 4

2. PINN の基本構成 6

3. PINN の用法 8

## 実装編 9

0. データセットの準備 10

1. 教師あり学習 12

2. 順解析 16

3. 逆解析 20

4. 一括解 22

## 付録 25

A. PINN 25

B. PyTorch 26

C. Python 27

D. Matplotlib 29

# はじめに

本稿では，Neural Network (NN)により微分方程式系を解析する Physics-Informed Neural Network (PINN)を，初学者が中身を理解した上で実装できるようになることを目指す．標準的なライブラリ以外は自らコーディングすることで，解きたい問題に応じたプログラムを作成する力を身につける．問題設定としては，原論文で扱われた順解析・逆解析に加え，NN の特性を活用した一括解を含んでおり，研究が進んでいる主な用法をカバーしている．その一方で，計算効率・精度向上のために実用上は必須となるモデル設計や最適化の技法はほとんど扱っていない．これらは基本的な実装方法を身につけてから取り組む課題とする．

実装編の各章末に演習問題を用意し，大まかな難易度（煩雑度）を☆〜☆☆☆で示した．実際に手を動かして挙動を確かめたり，より進んだ解析のイメージを掴んだりすることを目的としている．

本稿は分量を抑えるために最小限の記述にとどめている． PINN やプログラミングに関するより詳しい参考文献を以下に挙げる．

【PINN】

・原論文

Raissi, M., Perdikaris, P. and Karniadakis, G. E. (2019). Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations, *Journal of Computational Physics*, 378, 686–707. https://doi.org/10.1016/j.jcp.2018.10.045

・レビュー論文

Karniadakis, G. E., Kevrekidis, I. G., Lu, L., Perdikaris, P., Wang, S. and Yang, L. (2021). Physics-informed machine learning, *Nature Reviews Physics*, 3, 422–440. https://doi.org/10.1038/s42254-021-00314-5

・日本語の解説

岡崎智久 (2025). Scientific Machine Learning 地震学, 地震, 77, 101–120.
https://doi.org/10.4294/zisin.2024-9
(English translation: arXiv preprint. https://doi.org/10.48550/arXiv.2409.18397)

【プログラミング】

・Python

中久喜健司 (2016). 科学技術計算のための Python 入門――開発基礎，必須ライブラリ，高速化.
ISBN: 9784774183886

・PyTorch

宮本圭一郎, 大川洋平, 毛利拓也 (2018). PyTorch ニューラルネットワーク 実装ハンドブック.
ISBN: 9784798055473

# —理論編—

本稿は PINN のプログラミングの基本を習得することを目的とする．よって PINN の実装に必要な知識や定式化を整理するが，数式・計算アルゴリズムの体系的な理論展開は行わない．

# 1．NN と PINN

## 1.0 線形回帰モデル

入出力の対からなる有限のデータ$\{t_i, u_i\}_{i=1}^{N_{\text{data}}}$から未知関数$u = u(t)$を推定する回帰問題を考える．このタスクは**教師あり学習**と呼ばれ，それを解く機械学習は**データ駆動型**モデルと呼ばれる．線形回帰では，未知関数を既知の基底関数の線形結合$u_\theta(t) = \sum_{j=1}^{M} \theta_j \phi_j(t)$により表現し，データに最も適合するパラメタ$\theta = (\theta_1, \dots, \theta_M)$を求める（過剰適合や正則化の議論は省略する）．モデル$u_\theta(t)$がパラメタ$\theta$に関し線形のため（変数$t$に関し非線形であっても）線形モデルと呼ばれる．データ適合度の定量的指標が**損失関数**であり，回帰問題においては以下の平均二乗残差が標準的に用いられる：

$$L_{\text{data}}(\theta) = \frac{1}{N_{\text{data}}} \sum_{i=1}^{N_{\text{data}}} (u_\theta(t_i) - u_i)^2$$

すなわち$L_{\text{data}}$を最小化するパラメタ$\theta^* = \underset{\theta}{\operatorname{argmin}}\, L_{\text{data}}(\theta)$に対し$u_{\theta^*}(t) = \sum_{j=1}^{M} \theta_j^* \phi_j(t)$をモデルの予測値とする．線形モデルでは最適パラメタ$\theta^*$を解析的に計算できる．このように「データ」が与えられたときに「モデル・損失関数」を設計することが教師あり学習の基本である．

## 1.1 Neural Network (NN)

上と同じ回帰問題を考える．ここでは **Neural Network (NN)**と呼ばれる，線形変換と非線形変換の合成で表される，パラメタ$\theta$に関して非線形のモデルを考える：

$$u_\theta(t) = W_3\big(a(W_2 a(W_1 t + b_1) + b_2)\big) + b_3$$

$$\theta = (W_1, b_1, W_2, b_2, W_3, b_3)$$

線形変換の要素$W_i, b_i$が最適化パラメタであり，非線形変換$a$は活性化関数と呼ばれ事前に指定される．極めて柔軟に関数を表現できる上に，既知関数の合成関数のため微分を（理論上は）解析的に計算できる．これをうまく活用したのが後述の自動微分・誤差逆伝播である．NN に対し損失関数を上と同様に定めると，最適パラメタ$\theta^*$を解析的に求めるのは困難である．そこで数値的に最適化する必要があり，**勾配降下法**が用いられる．

## 1.2 Physics-Informed Neural Network (PINN)

回帰問題ではデータの説明を目的とするため，データ残差を損失関数としてきた．しかし，解きたい問題に応じて損失関数を定義することで種々のタスクを実行できる．ここでは微分方程式を解くことを考える．具体的に実装編で扱う減衰振動を考える：

$$m\frac{d^2u}{dt^2} + c\frac{du}{dt} + ku = 0$$
$$u(0) = u_0, \frac{du}{dt}(0) = v_0$$

$u(t)$が未知の解であり，第一式が常微分方程式(ODE)，第二式が初期条件(IC)である．物理定数（$m$: 質量，$c$: 減衰係数，$k$: 弾性定数）や初期値（位置$u_0$，速度$v_0$）は与えられているとする．

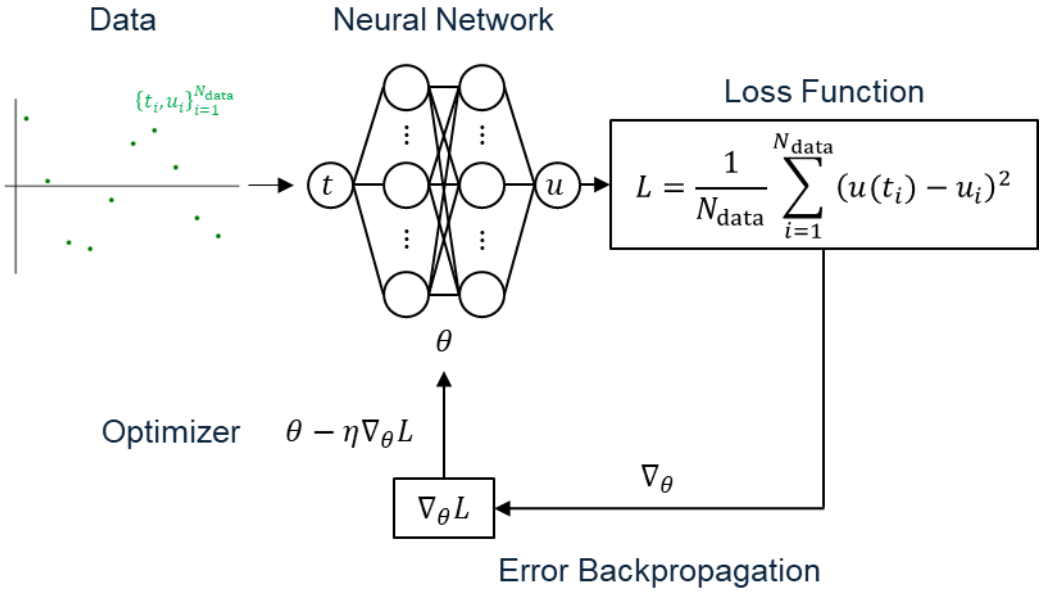
教師あり学習では，予測値をデータに適合させたいのでデータ残差を損失関数とした．同様に，微分方程式や初期条件を満たしたい場合は，それらからの残差を損失関数とすればよい：

$$L_{\mathrm{ODE}} = \frac{1}{N_{\mathrm{ODE}}}\sum_{i=1}^{N_{\mathrm{ODE}}}(mu_{tt} + cu_t + ku)^2$$
$$L_{\mathrm{IC}} = (u(0) - u_0)^2 + (u_t(0) - v_0)^2$$

二式を同時に満たす関数が解なので，和$L = L_{\mathrm{ODE}} + L_{\mathrm{IC}}$を全体の損失関数とする．その最小値 0 を達成するパラメタが得られれば真の解であるが，一般には有限の値にとどまり近似解が得られる．このように物理法則をもとに損失関数を定め NN を最適化するアプローチが，本稿の主題となる**Physics-Informed Neural Network (PINN)**である． Physics-Informed は「データ駆動型」と対概念であり，NN 設計を指す Convolutional, Recurrent 等とは直交する概念である．

# ２．PINN の基本構成

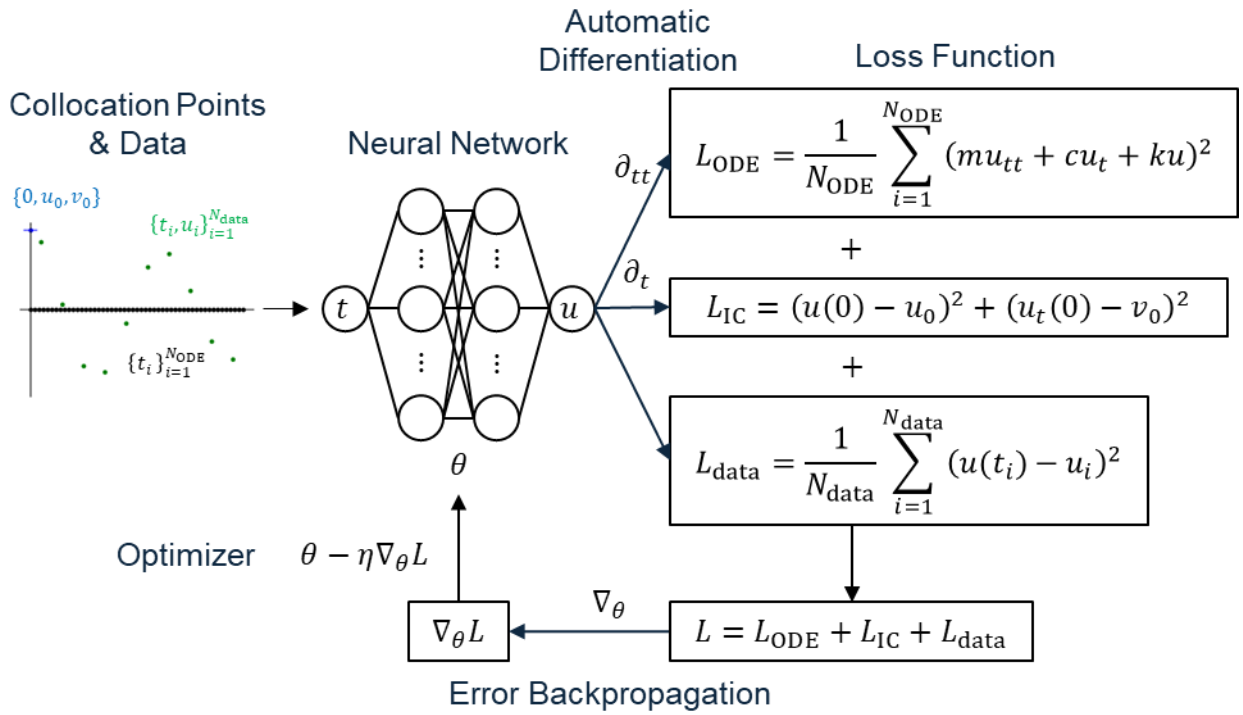
PINN をプログラムとして実装するために，基本的な構成要素を整理する．実装編のソースコードの構成と概ね対応している．




まずデータ駆動型 NN の基本構成を確認する（上図）：

- 教師データ
- NN モデル
- 損失関数
- 誤差逆伝播
- 最適化関数

教師あり学習の基本要素である「データ・モデル・損失関数」を用いる．NN モデルは損失関数を最小化するパラメタを解析的に計算できないため，勾配降下法により最適化する．この勾配計算は深層学習ライブラリに標準的に組み込まれている**誤差逆伝播**により効率的に実行できる．勾配法におけるパラメタ更新幅（学習率）は**最適化関数**(Optimizer)により規定され，その選択は訓練の効率・安定性に大きな影響を与える．深層学習ライブラリには種々の最適化関数が備わっている．




続いて PINN の基本構成を整理する（上図）：

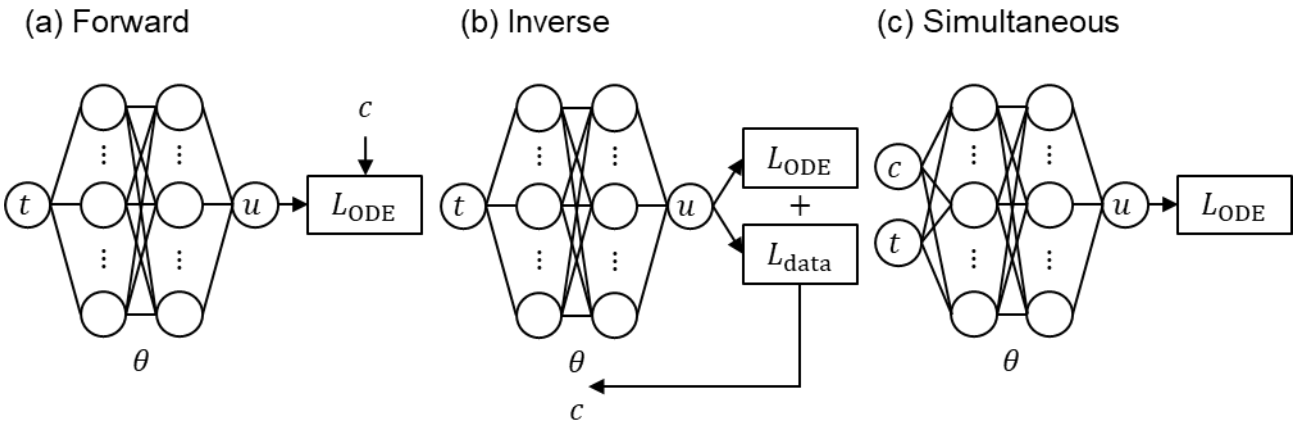
- 選点（&教師データ）
- NN モデル
- 自動微分
- 損失関数
- 誤差逆伝播
- 最適化関数

「教師データ」は順解析・一括解では不要である．データ駆動型 NN との主な相違点は「選点・自動微分・損失関数」である．損失関数に微分方程式や初期・境界条件からの残差が加わり，それらの和を全体の損失関数とする．損失関数に表れる NN 出力の入力変数に関する微分は，深層学習ライブラリに標準的に組み込まれている**自動微分**により精確・効率的に計算できる．さらに，離散的な教師データと異なり微分方程式は連続領域で定義されるため，訓練時に**選点**(Collocation Point)を定める．「NN モデル・誤差逆伝播・最適化関数」はデータ駆動型 NN と基本的に同じである．

教師データは与えられるものとすると，以下のようにモデル構築・訓練を行う．上の項目ほど実装者の寄与が高い．ただし，高度なモデリングではこの限りでない．

- 選点・損失関数：自分で定義する．
- NN モデル・最適化関数：ライブラリのクラス・関数から設計・選択する．
- 自動微分・誤差逆伝播：ライブラリの関数を呼び出す．

# 3．PINN の用法




(a) **順解析**. PINN の最も純粋な用法は，1.2 節で導入したように，完全な方程式系が与えられた下で求解する順解析である．微分方程式の係数や初期・境界条件などの解析条件を完全に指定して解を求める．これは差分法・有限要素法などの数値シミュレーションと同じ問題設定である．従来の NN と異なり教師データを全く用いずに解析できる点が特徴である．

(b) **逆解析**. PINN とデータ駆動型 NN は組み合わせることができる．すなわち$L = L_{\mathrm{ODE}} + L_{\mathrm{IC}} + L_{\mathrm{data}}$を最小化することで物理法則と観測データをともに満たす解が得られる．特に，物理法則に対する知識が不完全（微分方程式の係数や初期・境界条件の一部が未知）の場合に，未知パラメタを推定しつつ解を求める逆解析やデータ同化を実施できる．損失関数と（必要に応じて）NN の構造を変更するだけで順・逆解析を同じ枠組みで実施できる点が PINN の顕著な特徴である．

(c) **一括解**. 順解析ではパラメタ値を完全に指定して求解するため，複数の条件で解析する場合は訓練を繰り返す必要があり計算コストが大きい．そこで，解析条件を NN の入力変数とすることで異なるパラメタ値の解を同時解法できる．このような順解析の拡張を本稿では一括解と呼ぶ（まだ定着した用語がない）．例えば，初期条件に関する一括解は微分方程式の「一般解」に相当する．

3 用法の実装における相違点を減衰振動における減衰係数$c$を例に整理する（上図）．順解析では既知の定数(e.g., $c = 0.1$)を損失関数に代入して解析する．他の値(e.g., $c = 0.2$)の解を得るには NN を再訓練する必要がある．逆解析では$c$が NN パラメタとともに最適化の対象となる．損失関数には現ステップにおける$c$の推定値を代入する．一括解では$c$が NN の入力変数となる．損失関数には$c$の入力値を代入する．逆解析と一括解は混同される場合があるが，逆解析はデータに基づき未知のパラメタ値（真値は一つ）を推定するのに対し，一括解はデータを用いず任意のパラメタ値に関する解を一度に求める．

# ―実装編―

プログラム群“pinn_damped-oscillation”を題材に PINN の実装方法を説明する．最小限のライブラリのみを用いて，できるだけ平易に実装することを目的とする．そのため高度なプログラミング技術（最適化効率・推定精度の向上，構造的なコード構成，プログラムの高速化，GPU の使用，など）には対応していない．

GitHub URL:

【プログラミング環境】

- プログラミング言語：Python（3 系）
- 深層学習ライブラリ：PyTorch
- 基本的なライブラリ：Numpy, Matplotlib

※プログラムの実行時間を計測する場合はライブラリ Time を使用する（付録 C.6）．

【解析対象】

減衰振動（2 階常微分方程式）

- 未知解　$u(t)$　（$u$: 変位, $t$: 時刻）
- 方程式　$m\frac{d^2u}{dt^t} + c\frac{du}{dt} + ku = 0$（$m$: 質量, $c$: 減衰係数, $k$: 弾性定数）
- 初期値　$u(0) = 1,\ \frac{du}{dt}(0) = 0$

⇒理論解　$c < 2\sqrt{mk}$ のとき $u(t) = e^{-\left(\frac{c}{2m}\right)t}\cos\left(\sqrt{\frac{k}{m} - \left(\frac{c}{2m}\right)^2}\,t\right)$ (underdamping)

【実施項目】

1. 教師あり学習
2. 順解析
3. 逆解析
4. 一括解

教師データは誤差なし／ありの 2 種類を準備する．

# 0．データセットの準備 [0a_data.py & 0b_data_simul.py]

教師データ（誤差なし／あり）および精度評価のための真値を作成・保存する.

- 0a_data.py：項目 1~3 の教師データ・真値
- 0b_data_simul.py：項目 4 の真値

以下では 0a_data.py について説明する.

- パラメタ設定

```
m = 1.0  # Mass
c = 0.1  # Damping coefficient
k = 1.0  # Spring constant

noise = 0.05  # Observational noise

num_data = 10  # Number of data points
num_plot = 101  # Number of plot points
```

物理定数：m 質量・c 減衰係数・k 弾性定数

noise：誤差ありデータの観測誤差標準偏差

num_data：教師データ数，num_plot：真値の描画点数

- データ生成

```
# Data generator
def generator(m, c, k, tmin, tmax, num_data, noise=0):
    t = np.linspace(tmin, tmax, num_data)
    gamma = c / (2 * m)
    omega = np.sqrt(k / m - gamma ** 2)
    u = np.exp(-gamma * t) * np.cos(omega * t)  # Exact solution
    u += noise * np.random.randn(num_data)
    return t, u
```

【入力】

物理定数：m 質量・c 減衰係数・k 弾性定数

時間：区間[tmin, tmax]に num_data 個の等間隔グリッド

観測誤差：noise 正規分布の標準偏差（デフォルト値：0）

【出力】

t：時刻

u：変位の真値$u(t) = e^{-\gamma t}\cos(\omega t)$　$\left(\gamma = \frac{c}{2m}, \omega = \sqrt{\frac{k}{m} - \gamma^2}\right)$

※noise が正の場合は各データ独立に誤差を加算．この式は$c < 2\sqrt{mk}$の場合に成立．

```
# Generate data
t_exact, u_exact = generator(m, c, k, 0.5, 9.5, num_data)  # Exact data
t_noisy, u_noisy = generator(m, c, k, 0.5, 9.5, num_data, noise)  # Noisy data
t_truth, u_truth = generator(m, c, k, 0, 10, num_plot)  # Ground truth
```

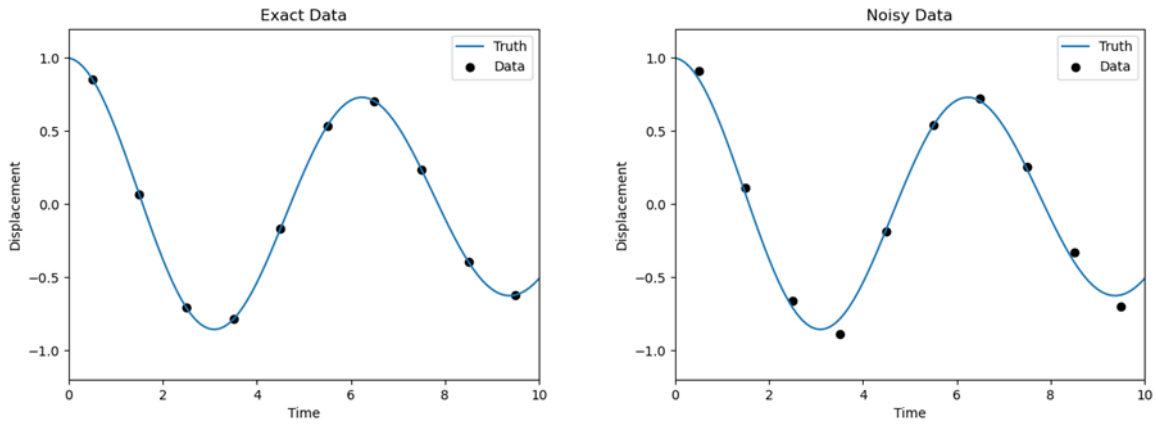
Exact data：10 点[0.5, 1.5, …, 8.5, 9.5]における誤差なしデータ（教師データ用）

Noisy data：10 点[0.5, 1.5, …, 8.5, 9.5]における誤差ありデータ（教師データ用）

Ground truth：101 点[0.0, 0.1, …, 9.9, 10.0]における真値（グラフ描画用）

⇒ファイル名“data_exact.txt”, “data_noisyt.txt”, “data_truth.txt”で保存.

Python, Matplotlib の基本事項は付録 C, D を参照：
インポート・ファイル入出力・画面出力・乱数のシード固定・グラフ描画




模擬データセット．左：誤差なしデータ，右：誤差ありデータ．実線は真値．

演習問題

1．物理定数・観測誤差・観測位置などを変更してデータセットを生成する．☆

# 1．教師あり学習 [1a_nn_exact.py & 1b_nn_noisy.py]

本稿のプログラム・コードは以下の順序で構成されている：

- 解析条件の設定（解析範囲・物理定数・初期値・NN 設計・乱数シード）
- クラス・関数の定義（NN・損失関数・選点）
- データ・モデルの指定（学習データ読込み・NN・最適化関数・選点）
- 訓練（モデルの最適化・モデルの保存）
- 推論（真値の読込み・モデルの予測・誤差評価）
- グラフ描画（解・学習曲線）

基本ライブラリ(PyTorch, Numpy, Matplotlib）と学習データ・厳密解の数値ファイルを読込む他は，プログラムの全体構成を把握しやすいよう単一のソースファイルに記述されている．

1. 教師あり学習と 3. 逆解析のソースコード a, b は教師データ（誤差なし／あり）のみ異なる．

— 解析条件の設定 —

```
# Network parameters
num_epoch = 10000  # Number of training epochs
```

num_epoch：NN の訓練エポック数

— クラス・関数の定義 —

- NN

```
# Neural network
class NN(nn.Module):
    def __init__(self):
        super(NN, self).__init__()
        self.fc1 = nn.Linear(1, 20)
        self.fc2 = nn.Linear(20, 20)
        self.fc3 = nn.Linear(20, 1)

    def forward(self, x):
        x = torch.tanh(self.fc1(x))
        x = torch.tanh(self.fc2(x))
```

```
        x = self.fc3(x)
        return x
```

入出力変数が 1 ノード（各々$t$と$u$に対応），隠れ層が 20 ノード×2 層，活性化関数が Tanh の全結合 NN を構成する．PyTorch における NN 定義については付録 B.4 を参照.

- 損失関数

```
# Loss function
def loss_function(net, x_data, u_data):

    # Data
    u_pred = net(x_data)
    loss_data = torch.mean((u_pred - u_data) ** 2)

    return loss_data
```

【入力】

net：NN モデル

x_data：入力値（時刻$t_i$）

u_data：正解値（変位$u_i$）

【出力】

loss_data：データ残差 $L_{\text{data}} = \frac{1}{N_{\text{data}}} \sum_{i=1}^{N_{\text{data}}} (u(t_i) - u_i)^2$

— データ・モデルの指定 —

- 学習データ読込み

```
# Load data
tu_data = torch.from_numpy(np.loadtxt('data/data_exact.txt', dtype='float32'))
t_data = tu_data[:,[0]]
u_data = tu_data[:,[1]]
```

0a_data.py で作成した入力値・正解値の対を読み込む.

- モデルの指定

```
# Model assign
net = NN()
optimizer = torch.optim.Adam(net.parameters(), lr=0.001)
```

はじめに定義した NN クラスを net と名付け呼び出し，最適化関数 optimizer に Adam を用いる．その引数 net.parameters()は，NN の訓練可能パラメタを更新対象に指定している．

— 訓練 —

```
# Model training
for epoch in range(num_epoch):
    optimizer.zero_grad()
    loss = loss_function(net, t_data, u_data)
    loss.backward()
    optimizer.step()
```

指定された反復回数 num_epoch だけパラメタ更新する．以下のステップは定型文（付録 B.2）：

1. 勾配の初期化
2. 損失関数の評価（定義した関数 loss_function を計算）
3. 誤差逆伝播
4. パラメタ更新

— 推論 —

- モデルの予測

```
# Model prediction
net.eval()
with torch.no_grad():
    u_pred = net(t_true)
```

訓練済み NN net により t_true における予測値 u_pred を計算する．

net.eval()：NN を推論モードに切替え（ドロップアウト・バッチ正規化などの挙動が変化）

with torch.no_grad()：順伝播において勾配計算を行わない⇒省メモリ・計算速度の向上

- 誤差評価

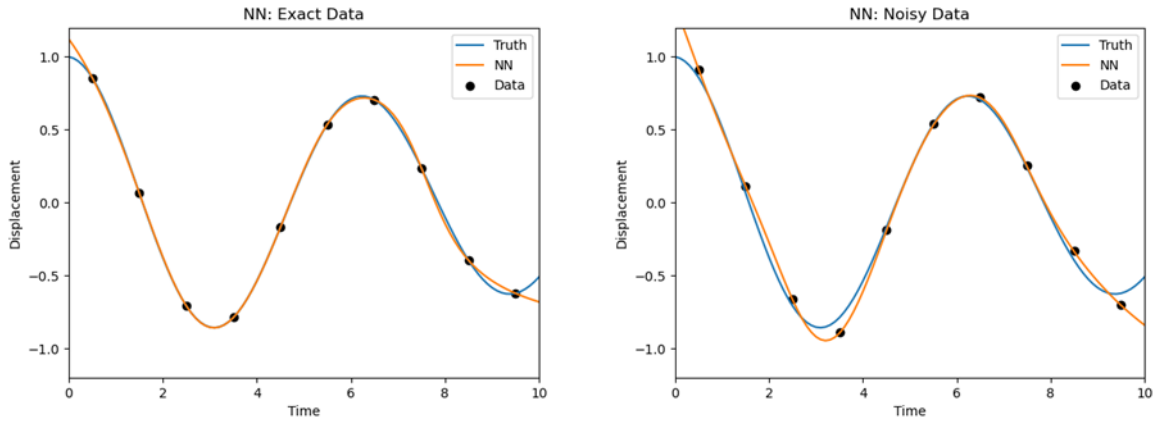
```
# Error
nrm = torch.sqrt(torch.mean(u_true ** 2))
abs_err = torch.sqrt(torch.mean((u_pred - u_true) ** 2))
rel_err = abs_err / nrm
```

`nrm`：真値のノルム $\|u_{\text{true}}\|_2 = \left(\int_{t_{\min}}^{t_{\max}} u_{\text{true}}^2\, dt\right)^{1/2}$

`abs_err`：絶対誤差 $E_{\text{abs}} = \|u_{\text{pred}} - u_{\text{true}}\|_2 = \left(\int_{t_{\min}}^{t_{\max}} (u_{\text{pred}} - u_{\text{true}})^2\, dt\right)^{1/2}$

`rel_err`：相対誤差 $E_{\text{rel}} = \|u_{\text{pred}} - u_{\text{true}}\|_2 / \|u_{\text{true}}\|_2$

Euclid ($L_2$)ノルムを使用する．なお，教師あり学習では損失関数と誤差は類似の量（データ／真値からの二乗残差）だが，PINN では損失関数に微分方程式からの残差が含まれるため本質的に異なる．




教師あり学習による解．左：誤差なしデータ，右：誤差ありデータ．

**演習問題**

1. ネットワーク構造（隠れ層・活性化関数など）を変更する．NN クラスを直接書き換える他，各パラメタをクラスの引数にする方法が考えられる．☆
2. ミニバッチ学習を行う（付録 B.3）．一般に DataLoader を用いる．☆
3. 最適化関数 Adam の学習率を変更する（付録 B.4.1）．例えば Scheduler を用いる．☆
4. 最適化関数に L-BFGS を用いる（付録 B.4.2）．さらに Adam と L-BFGS を組み合わせる．☆☆

# 2． 順解析 [2_forward.py]

本稿では以下を標準設定とする：

- ネットワーク構造：全結合（隠れ層：20 ノード×2 層）
- 活性化関数：Tanh
- 最適化関数：Adam（学習率 0.001）
- 選点：一様グリッド（フルバッチ学習）
- 損失関数の重み：全て 1

上 3 つは NN 一般の，下 2 つは PINN 特有の設定である．こうした設計は最適化効率・解の精度に大きく影響するため，実用上は適切に設定することが重要であり，多数の研究がある（付録 A）．

— パラメタ設定 —

● 問題設定

```
# Physical constants
m = 1.0  # Mass
c = 0.1  # Damping coefficient
k = 1.0  # Spring constant

# Initial conditions
u0 = 1.0  # Initial displacement
v0 = 0.0  # Initial velocity
```

物理定数$(m, c, k)$と初期値$(u_0, v_0)$を指定する．

✧ PINN 順解析では，微分方程式の係数および初期値が指定されている必要がある．データ駆動型の NN とは異なり，数値シミュレーション（差分法・有限要素法など）と同じである．

● モデル・パラメタ

```
tmin = 0
tmax = 10
num_coll = 101  # Number of collocation points
num_epoch = 10000  # Number of training epochs
```

解析時間[tmin, tmax]と選点数 num_coll を設定. 上の設定では$t = \{0, 0.1, 0.2, \ldots, 9.8, 9.9, 10\}$となる．

— クラス・関数の定義 —

✧　NN クラスは教師あり学習と全く同一のため省略する．

- 損失関数

```
# Loss function
def loss_function(net, t_coll, m, c, k, u0, v0):
```

【入力】

net：NN モデル

t_coll：入力変数の選点

m, c, k：物理定数

u0, v0：初期値

```
# ODE
u = net(t_coll)  # u(t)
u_t = torch.autograd.grad(u, t_coll, grad_outputs=torch.ones_like(u),
           create_graph=True)[0]  # du/dt
u_tt = torch.autograd.grad(u_t, t_coll, grad_outputs=torch.ones_like(u_t),
           create_graph=True)[0]  # d2u/dt2
r = m * u_tt + c * u_t + k * u  # ODE residual
loss_ode = torch.mean(r ** 2)
```

【微分方程式】

選点（バッチ）t_coll に対し，モデルの予測値 u を順計算により，その時間微分 u_t, u_tt を自動微分 torch.autograd.grad により計算する．2 階微分は，変数の 1 階微分をさらに自動微分することで得られる．自動微分の詳細は付録 B.1 を参照．微分方程式からの残差 r を計算し，その選点における二乗平均を損失関数 loss_ode とする．

```
# IC
t0 = torch.zeros((1, 1), requires_grad=True)  # t = 0
u0_pred = net(t0)  # u(0)
v0_pred = torch.autograd.grad(u0_pred, t0, grad_outputs=torch.ones_like(u0_pred),
           create_graph=True)[0]  # du/dt(0)
loss_ic = torch.mean((u0_pred - u0) ** 2 + (v0_pred - v0) ** 2)
```

【初期条件】

初期時刻$t = 0$を t0 とし，モデルの初期値 u0_pred を NN 順伝播により，初期速度 v0_pred を自動微分により計算する．与えられた初期値 u0, v0 との残差二乗和を損失関数 loss_ic とする．

```
# Total
loss = loss_ode + loss_ic
return loss, loss_ode, loss_ic
```

【出力】

loss：損失関数$L$（全体）

loss_ode：損失関数$L_{\mathrm{ODE}}$（微分方程式からの残差）

loss_ic：損失関数$L_{\mathrm{IC}}$（初期値からの残差）

- 選点

```
# Collocation points
def uniform_grid(tmin, tmax, num_coll):
    return torch.linspace(tmin, tmax, num_coll, requires_grad=True).view(-1, 1)
```

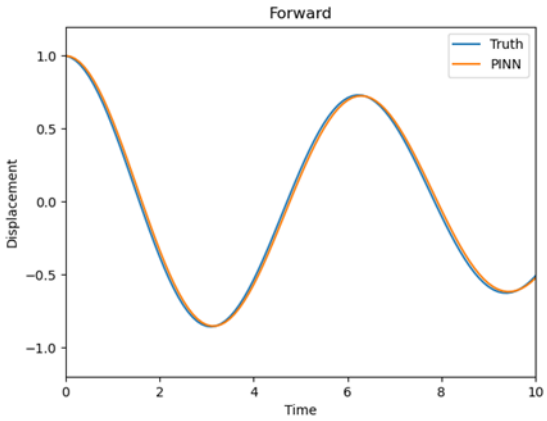
区間[tmin, tmax]に等間隔で num_coll 点のグリッドを生成する（付録 A.1.1）．

— モデルの指定 —

```
# Model assign
net = NN()
optimizer = torch.optim.Adam(net.parameters(), lr=0.001)
t_coll = uniform_grid(tmin, tmax, num_coll)
```

教師データを読み込まない代わりに，選点を定める．

✧ 訓練・推論は基本的に教師あり学習と同様である．




順解析による解.

**演習問題**

1. 訓練途中の NN を用いて推論し，誤差評価や解の描画を行う．☆
2. 解析条件（物理定数・初期値）を変更する．特に臨界・過減衰($c \geq 2\sqrt{mk}$)を解析する．☆
3. **過剰決定問題**：物理法則と誤差なしデータが与えられるとして損失関数$L = L_{\mathrm{ODE}} + L_{\mathrm{IC}} + L_{\mathrm{data}}$を最小化する．収束の早さ・解の精度を教師あり学習・順解析と比較する．また$L_{\mathrm{ODE}}$のパラメタを教師データと異なる値（例えば$c = 0$）に変更するとどうなるか．☆
4. **損失関数の重み**：損失関数を$L = w_{\mathrm{ODE}}L_{\mathrm{ODE}} + w_{\mathrm{IC}}L_{\mathrm{IC}}$として，重み$w_{\mathrm{ODE}}, w_{\mathrm{IC}} \geq 0$を変えて解析し損失関数の値や得られる解を比較する（付録 A.2）．☆
5. **ハード制約**:初期条件を厳密に満たすように NN を構成する（付録 A.3）．このとき損失関数は$L = L_{\mathrm{ODE}}$でよい．☆☆
   ヒント：$u(t) = u_0 + tv_0 + t^2 f(t)$とすれば$u(0) = u_0, u'(0) = v_0$である．
6. **ランダム選点**：エポック毎に選点を一様乱数からサンプリングする（付録 A.1.2）．☆☆

# 3． 逆解析 [3a_inverse_exact & 3b_inverse_noisy.py]

ここでは減衰係数$c$を未知パラメタとして推定する．順解析との違いは，(a)教師データを使用すること，(b)未知パラメタと推定対象とすること，である．(a)は教師あり学習と同様であり，(b)は以下の書き換えで済む．本コードではスカラー値を推定するが（ベクトル値も同様），関数の推定は NN で表現することで実現できる（演習問題 3）．

— 解析条件の設定 —

```
c0 = 1.0 # Initial guess for c
```

推定パラメタの初期値．

— データ・モデルの指定 —

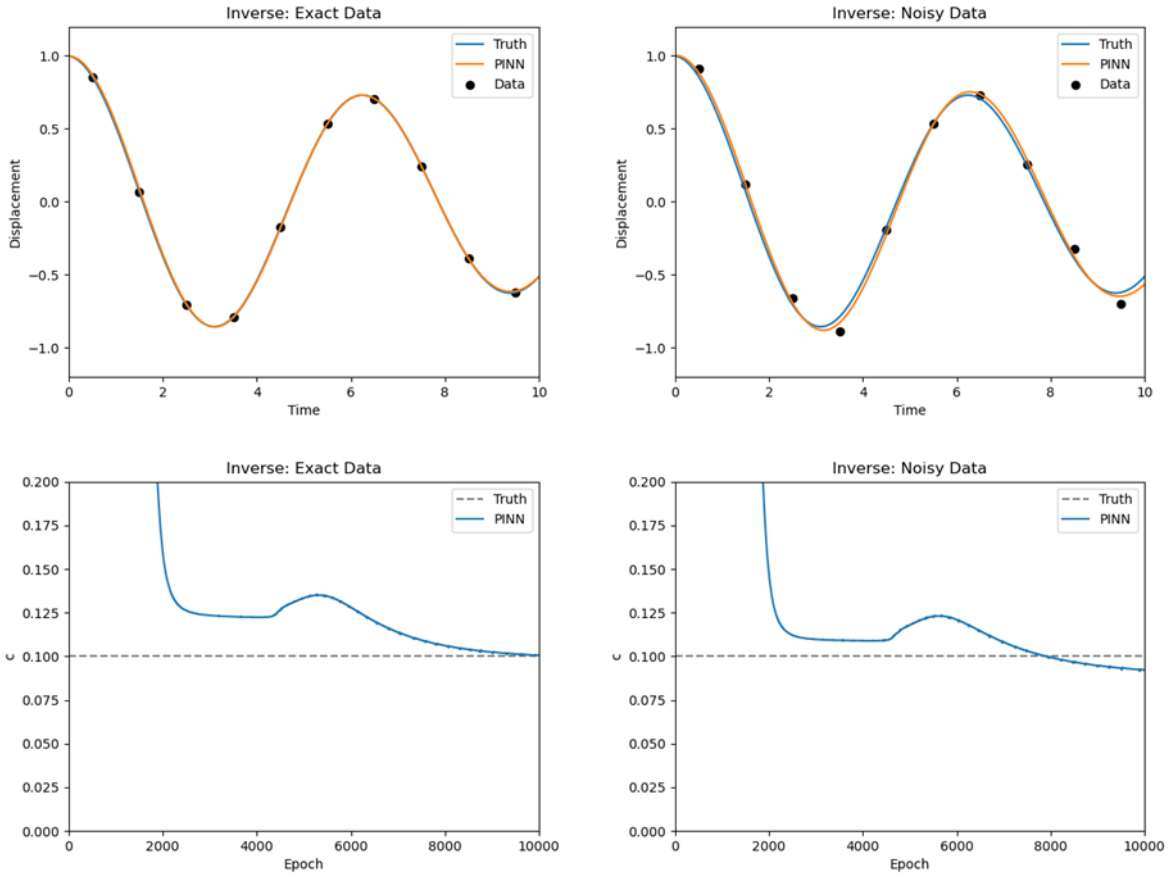
```
c_est = torch.tensor(c0, dtype=torch.float32, requires_grad=True)
params = [c_est]
```

推定パラメタ c_est を初期化し，全体をリスト変数 params とする（今は 1 変数のみ）．

```
optimizer = optim.Adam(list(model.parameters()) + params, lr=0.001)
```

訓練パラメタを NN パラメタ&推定パラメタ params とする．

✧ 損失関数は$L_{\mathrm{ODE}}, L_{\mathrm{IC}}, L_{\mathrm{data}}$の和となる．実装は教師あり学習・順解析を合わせたものであるが，引数 c が定数ではなく，訓練時に更新されるパラメタ c_est となる．




逆解析による解（上段）とパラメタ推定値（下段）．左：誤差なしデータ，右：誤差ありデータ．

## 演習問題

1. 減衰係数$c$と弾性定数$k$を推定する．☆
2. 初期値$u_0, v_0$が未知として減衰係数$c$を推定する．☆
   ヒント：$u_0, v_0$を推定パラメタに加えることもできるが，単に$L_{\mathrm{IC}}$を省略すればよい．
3. **データ同化**：PINN の解析範囲を `tmax=15` などデータ期間より広く指定して解析する．推定値が訓練のエポック数によりどう推移するか．☆
4. **関数の推定**：減衰係数$c(t)$が時間依存するとして推定する．☆☆
   ヒント：変位のNN:$t \mapsto u$と減衰係数のNN:$t \mapsto c$を構成し同時に訓練する．または両者をまとめてNN:$t \mapsto (u, c)$を構成してもよい．

# ４． 一括解 [4_simul.py]

微分方程式を減衰係数$c$の異なる値に対して一括解法する．教師データは用いない．入力変数の増加に応じて NN 構造と選点を変更する．損失関数は順解析と同一であるが，NN 構造の変更に伴いコードを少し書き換える必要がある．

― 解析条件の設定 ―

```
# Physical constants
m = 1.0  # Mass
k = 1.0  # Spring constant

# Analysis range
cmin = 0.0
cmax = 1.0
tmin = 0
tmax = 10

# Network parameters
num_coll_c = 21 # Number of collocation points for c
num_coll_t = 51 # Number of collocation points for t
```

物理定数において$c$は定数でないので指定しない．2 変数$c,t$の解析範囲・選点の個数をそれぞれ指定する．微分方程式の選点数は積(num_coll_c * num_coll_t)となる．

― クラス・関数の定義 ―

- NN

```
# Neural network
class NN(nn.Module):
    def __init__(self):
        super(PINN, self).__init__()
        self.fc1 = nn.Linear(2, 20)   # Input (c, t)
        self.fc2 = nn.Linear(20, 20)
        self.fc3 = nn.Linear(20, 20)
        self.fc4 = nn.Linear(20, 1)
```

入力変数を 2 次元$(c,t)$に変更する.

- 損失関数

```
# Loss function
def loss_function(net, ct_coll, ct_init, m, k, u0, v0):
    # ODE
    u = net(ct_coll).reshape(-1)  # u(c,t)
    u_ct = torch.autograd.grad(u, ct_coll, grad_outputs=torch.ones_like(u),
           create_graph=True)[0]  # (du/dc, du/dt)
    u_t = u_ct[:,1]  # du/dt
    u_tct = torch.autograd.grad(u_t, ct_coll, grad_outputs=torch.ones_like(u_t),
           create_graph=True)[0]  # (d2u/dcdt, d2u/dt2)
    u_tt = u_tct[:,1]  # d2u/dt2
    r = m * u_tt + ct_coll[:,0] * u_t + k * u  # ODE residual: c is an NN input
    loss_ode = torch.mean(r ** 2)

    # IC
    u0_pred = net(ct_init).reshape(-1)  # u(c,0)
    u0_ct = torch.autograd.grad(u0_pred, ct_init, grad_outputs=torch.ones_like
           (u0_pred), create_graph=True)[0]  # (du/dc(0), du/dt(0))
    v0_pred = u0_ct[:,1]  # du/dt(0)
    loss_ic = torch.mean((u0_pred - u0) ** 2 + (v0_pred - v0) ** 2)
```

数式上は順解析と同一だが，減衰係数$c$が NN 入力変数のため ct_coll[:,0]を代入する.

初期値の評価は，順解析では 1 点$t=0$であったが，一括解では区間[cmin, cmax]から選点する.

- 選点

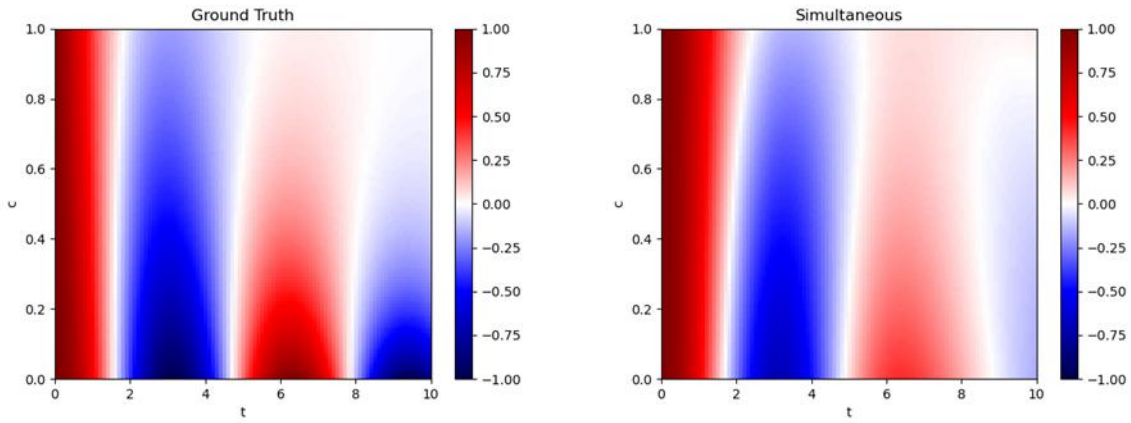
```
# Collocation points
def uniform_grid(cmin, cmax, tmin, tmax, num_coll_c, num_coll_t):
    c_coll = torch.linspace(cmin, cmax, num_coll_c, requires_grad=True)
    ct_init = torch.cat((c_coll.reshape(-1, 1), torch.zeros(num_coll_c, 1)), 1)
    t_coll = torch.linspace(tmin, tmax, num_coll_t, requires_grad=True)
    c_coll, t_coll = torch.meshgrid(c_coll, t_coll)
    ct_coll = torch.cat((c_coll.reshape(-1, 1), t_coll.reshape(-1, 1)), 1)
    return ct_coll, ct_init
```

【出力】

ct_coll：微分方程式の選点$(c,t)$ 2 次元配列(num_coll_c * num_coll_t, 2)

ct_init：初期条件の選点$(c,0)$ 2 次元配列(num_coll_c, 2)

グリッド作成の詳細は付録 A.1.1 を参照.




真値（左）と一括解（右）.

**演習問題**

1. 複数の物理定数・初期値に関し同時解法する．高次元入力では一様グリッドのサイズが大きくなるため，例えばランダム選点を検討する（付録 A.1.2）．☆☆
2. **作用素学習**：時間依存する減衰係数$c(t)$に関し同時解法する．この問題は関数を入出力とする解作用素$A: c(t) \mapsto u(t)$の導出に相当する．☆☆☆
   ヒント：PINN で関数を扱うためには，$c(t)$を例えばスプライン関数の線形結合で表現し，その展開係数を NN への入力変数とする．特定の NN 構造を用いると Physics-Informed DeepONet と呼ばれる作用素学習モデルとなる．
3. **ベイズ逆解析**：減衰係数$c$の逆解析をマルコフ連鎖モンテカルロ法で行う．その際に訓練済みの PINN 一括解を順計算に用いる．このように，一括解は異なるパラメタ値に対する反復計算に有用である．☆☆☆

# —付録—

# A. PINN

## A.1 選点 (Collocation points)

### A.1.1 固定グリッド

- 1 次元グリッド

```
def uniform_grid(tmin, tmax, num_coll):
    return torch.linspace(tmin, tmax, num_coll, requires_grad=True).view(-1, 1)
```

区間[tmin, tmax] に num_coll_t 個の等間隔配置の 1 変数$t$のグリッド格子を作成.

PINN では入力変数で微分するため requires_grad=True とする必要がある.
.view(-1, 1)は NN への入力に合わせて 1 次元配列(N, )から 2 次元配列(N, 1)に整形している.

- 2 次元グリッド

```
def uniform_grid(cmin, cmax, tmin, tmax, num_coll_c, num_coll_t):
    c_coll = torch.linspace(cmin, cmax, num_coll_c, requires_grad=True)
    t_coll = torch.linspace(tmin, tmax, num_coll_t, requires_grad=True)
    c_coll, t_coll = torch.meshgrid(c_coll, t_coll)
    ct_coll = torch.cat((c_coll.reshape(-1, 1), t_coll.reshape(-1, 1)), 1)
    return ct_coll
```

$c$は区間[cmin, cmax] に num_coll_c 個，$t$は区間[tmin, tmax] に num_coll_t 個の等間隔配置の 2 変数$(c, t)$のグリッド格子を作成.

2 行：torch.linspace により 1 次元等間隔グリッド$\mathbf{c} = (c_1, \dots, c_{N_c})$
3 行：torch.linspace により 1 次元等間隔グリッド$\mathbf{t} = (t_1, \dots, t_{N_t})$
4 行：torch.meshgrid により$\mathbf{c} = (c_1, \dots, c_{Nc})$, $\mathbf{t} = (t_1, \dots, t_{Nt})$を 2 次元グリッドに拡張

$$\mathbf{c}_2 = \begin{pmatrix} c_1 & \cdots & c_1 \\ \vdots & \ddots & \vdots \\ c_{Nc} & \cdots & c_{Nc} \end{pmatrix}, \quad \mathbf{c}_2 = \begin{pmatrix} t_1 & \cdots & t_{Nt} \\ \vdots & \ddots & \vdots \\ t_1 & \cdots & t_{Nt} \end{pmatrix} \quad \text{2 次元配列}(N_c,\ N_t)$$

6 行：.reshape(-1, 1)により 2 次元配列$(N_c N_t, 1)$に整形してから torch.cat により連結
（第 2 引数 1 はテンソルの第 1 成分に連結するよう指定）

$$\mathbf{ct}_{\text{coll}} = \begin{pmatrix} c_1 & \cdots & c_{Nc} \\ t_1 & \cdots & t_{Nt} \end{pmatrix}^{\mathrm{T}} \quad \text{2 次元配列}(N_c N_t, 2)$$

A.1.2 乱数サンプリング

A.2 損失関数の重み

A.3 ハード制約

# B. PyTorch

B.1 自動微分

プログラム b1_autodiff.py を例に説明する．

```
def y(x):
    return x ** 2 + torch.sin(x)
```

関数$y(x) = x^2 + \sin(x)$を考える．torch.sin のように Torch の組み込み関数を用いる．

```
x = torch.tensor([-1., 0., 1.], requires_grad=True)
```

自動微分を計算する入力値を定義する．Torch テンソルを用い requires_grad=True とする．

```
y = y(x)
dy_auto = torch.autograd.grad(y, x, grad_outputs=torch.ones_like(y),
           create_graph=True)[0]
ddy_auto = torch.autograd.grad(dy_auto, x, grad_outputs=torch.ones_like(y))[0]
```

1 行：$x = (-1, 0, 1)$における関数$y(x)$の値 y を計算する．

2 行：y から$x = (-1, 0, 1)$における導関数$y'(x)$の値 dy_auto を計算する．torch.autograd.grad を用いる．第 1, 2 引数が微分したい関数と微分する変数である．元々スカラー関数を微分するものであるが，grad_outputs により複数値を同時に自動微分できる．create_graph=True とすることでさらに自動微分を行えるようにする．

3 行：dy_auto から$x = (-1, 0, 1)$における二次導関数$y''(x)$の値 ddy_auto を計算する．このように，高階微分は自動微分を繰り返すことで計算できる．

なお自動微分の第 2 引数は複数指定でき、例えば(x, z)とすると出力は(dy/dx, dy/dz)となる．上のように単一変数で微分する場合、出力が(dy/dx,)なので末尾に[0]として dy/dx を取り出す。

B.2 誤差逆伝播

```
for epoch in range(num_epoch):
    optimizer.zero_grad()
```

```
    loss = loss_function(net, t_data, u_data)
    loss.backward()
    optimizer.step()
```

以下を訓練エポック数 num_epoch だけ繰り返す。

2 行：勾配の初期化

3 行：：データ t_data, u_data と現時点のモデル net に対する損失関数の値を計算

4 行：誤差逆伝播

5 行：最適化関数 optimizer によるパラメタ更新

### B.3 ミニバッチ学習

### B.4 最適化関数 (Optimizer)

#### B.4.1 Adam

#### B.4.2 L-BFGS

### B.5 ネットワーク設計

#### B.5.1 ネットワーク構造

#### B.5.2 活性化関数

#### B.5.3 初期化

# C. Python

### C.1 インポート

```
import torch
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt
```

import により使用するライブラリを呼び出す．as 以下はコード内で使用する名前を定めている．

### C.2 ファイル入出力

- TXT ファイル

```
fe = open('data/data_exact.txt', 'w')
fe.write(str('%.1f'%t_exact[i])+'¥t'+str('%.4f'%u_exact[i])+'¥n')
```

```
fe.close()
```

1 行：関数 open で書き込むファイルを変数 fe と名付けて開く．第一引数がファイル名，第二引数の'w'は書き込みモードを指定している．

2 行：ファイル fe に.write を付けるとファイルに書き込む．数値（Int, Float 型など）を直接書き込めないため str で文字列型に変換している．また小数点以下の桁数を'%.1f'%, '%.4f'%により定めている．'¥t'と'¥n'はタブ区切りと改行．

3 行：ファイルを閉じる．

```
loss = np.loadtxt('result/loss_1a_nn_exact.txt').T
```

関数 np.loadtxt により数値データのファイルを Numpy 配列として読み込む．変数名を loss としている．ファイル内容が完全な配列でなかったり，文字列が含まれたりする場合は要注意である．末尾の.T は配列を転置している．

- CKPT ファイル

```
torch.save(net, 'result/net_1a_nn_exact.ckpt')
```

Torch 変数 net をファイル名'result/net_1a_nn_exact.ckpt'で保存する．

```
net = torch.load('result/net_1a_nn_exact.ckpt')
```

ファイル'result/net_1a_nn_exact.ckpt'を変数 net として読み込む．

### C.3 画面出力

```
print('Epoch', epoch, 'Loss:', str('%.4e'%loss))
```

関数 print により実行画面に出力する．'%.4e'%は指数表記（小数点以下 4 桁）．

### C.4 乱数のシード固定

```
seed = 1
np.random.seed(seed)
torch.manual_seed(seed)
```

Numpy, Torch の乱数シードを固定している．seed の値により調整できる．

### C.5 乱数生成

```
num_data = 10
```

```
np.random.rand(num_data)
np.random.randn(num_data)
```

引数の個数だけ独立に乱数生成.

`np.random.rand`：区間[0, 1]の一様分布

`np.random.randn`：標準正規分布

C.6 実行時間の計測

```
import time

t1 = time.time()
t2 = time.time()
dt = t2 - t1
```

ライブラリ `time` をインポートし `time.time()`により実行時の時刻を取得する．2 つの時刻の差分により実行時間（秒）を計測できる.

# D. Matplotlib

- 折れ線グラフ・散布図

```
plt.figure()
plt.plot(t_truth, u_truth, label='Truth')
plt.scatter(t_exact, u_exact, c='k', label='Data')
plt.xlim(0, 10)
plt.ylim(-1.2, 1.2)
plt.xlabel('Time')
plt.ylabel('Displacement')
plt.legend()
plt.title('Exact Data')
plt.savefig('figure/0_data_exact.png')
plt.close()
```

`plt.figure()`：描画開始

`plt.plot`：折れ線グラフ．第 1, 2 引数が $x, y$ 軸の値．`label='Truth'`はラベル名を指定している.

`plt.scatter`：散布図．第 1, 2 引数が $x, y$ 軸の値．`c='k'`は黒色を指定している.

`plt.xlim`：横軸範囲の指定

`plt.xlabel`：横軸ラベル

`plt.legend()`：ラベル(`'Truth'`, `'Data'`)をグラフ内に表示

plt.title：グラフタイトル

plt.savefig：図の保存．引数はファイル名．

plt.close()：描画終了

- ヒートマップ

```
plt.scatter(ct_true[:,1], ct_true[:,0], marker='s', s=5, c=u_pred,
            cmap='seismic', vmin=-1, vmax=1)
```

marker='s'：マーカーを四角

s=5：マーカーサイズを 5

c=u_pred：u_pred の値による色

cmap='seismic'：カラーマップに'seismic'を使用

vmin=-1, vmax=1：カラー範囲を[-1, 1]に設定