1. 引言

研究问题（problem）

物理信息神经网络（PINN）已经成为一种新的学习范式，通过将物理方程约束、边界条件约束、初始条件约束加入到损失函数当中进而求解偏微分方程（PDEs）。尽管这是成功的，但 PINN 由于难以处理多目标优化任务（PDE 残差损失与边界和初始条件损失不在一个量级或者差异较大，训练难度不同），其仍然具有准确性差、收敛速度慢的问题。

解决方案（Innovation）

为解决上述问题，由香港浸会大学和新加坡科技研究局高性能计算研究院合作共同提出了一种新型双平衡 PINN（BP-PINN），通过集成外部平衡和内部平衡来动态调整损失权值，以缓解 PINN 中的两个不平衡的问题，进而提升 PINN 的多目标任务优化能力。

创新点
传统的损失函数权值分配方法：

- 手动分配：效率低，需要大量人工试错改进
- （学习方法）将权重作为超参数在梯度下降中更新：随参数增加，优化复杂度不断增加，训练困难
- （计算方法）基于梯度统计的加权方法：只考虑 PDE 残差与每个初始条件与边界条件的关系，而没有考虑初始和边界条件彼此之间的内部关系（梯度不同造成的训练难度差异）

DB-PINN——将 PINN 中的多优化目标考虑为两个主要目标

- 平衡 PDE 残差损失和内在条件损失
- 平衡内部具有不同拟合难度的条件损失
- 为了平滑的权重更新和稳定的训练提出一种健壮的权重更新策略

2. 论文

2.1 输入数据

对于求解 Klein–Gordon 方程：

$$u_{tt} - u_{xx} + u^3 = 0, \quad x \in [0,1],\, t \in [0,1]$$

- 空间坐标 x
- 时间坐标 t

Wave 方程：

$$u_{tt}(x,t) - 4u_{xx}(x,t) = 0, \quad x \in [0,1],\, t \in [0,1]$$

- 一维空间 x

- 时间坐标 t

Helmholtz 方程：

$$u_{xx} + u_{yy} + u = q(x, y), \quad x, y \in [-1, 1]$$

- 二维空间（x , y）

Burgers 方程：

$$u_t + uu_x = \nu u_{xx}$$

- 一维空间 x
- 时间坐标 t

Navier-Stokes 方程：（方程形式取决于具体问题）

- 二维不可压缩 N-S：二维空间+时间
- 三维不可压缩 N-S：三维空间+时间

Schrödinger 方程：

- 一维：（x ， t）
- 二维：（x ， y ， t）

Heat 方程：

- 输入取决于维度——一维、二维、三维+时间

2.2 相关性计算

（1）PDE 残差损失和条件拟合损失之间的相关性：

条件损失总权重：

$$\mathcal{G} = \sum_{i=1}^{M} \frac{\max\{|\nabla_\theta \mathcal{L}^r|\}}{\overline{|\nabla_\theta \lambda^i \mathcal{L}^i|}}$$

∇_θ表示损失相对于网络参数 θ 的梯度向量　L^r表示 PDE 残差损失

λiLi 表示第 i 个条件损失

（2）条件损失中不同损失的训练难度系数：

$$\mathcal{I}_t = \frac{\mathcal{L}_t}{\mu_{\mathcal{L}_t}}$$

Lt 表示 t 时刻的某一观测条件损失，μLt 表示截止 t 时刻前所有该观测损失的平均值

2.3 针对 DB-PINNs 提出的权重优化策略

由于梯度下降更新具有随机性，瞬时权重值表现出较大的方差。现有的基于梯度的权重方法通常采用指数移动平均（EMA）策略来更新权重：$\lambda^i = (1-\alpha)\lambda^i + \alpha\hat{\lambda}^i$，其中 α 是一个超参数。然而，作者观察到该更新策略仍无法有效处理较大方差，导致瞬时权重值频繁出现突然尖峰，甚至发生算术溢出——当使用标准差或峰度作为统计度量时，此现象尤为明显。为解决这一问题，本文提出了一种鲁棒的免调超参数损失权重更新策略，这种策略使用 Welford 算法，一种用来跟踪观测条件拟合的均值的在线估计方法。使用以下更新规则的平均观测损失向量：

$$\mu_{\mathcal{L}_t} = (1 - \frac{1}{t})\mu_{\mathcal{L}_{t-1}} + \frac{1}{t}\mathcal{L}_t$$

权重更新同理：

$$\lambda^i = (1 - \frac{1}{t})\lambda^i + \frac{1}{t}\hat{\lambda}^i$$

2.4 训练与损失函数

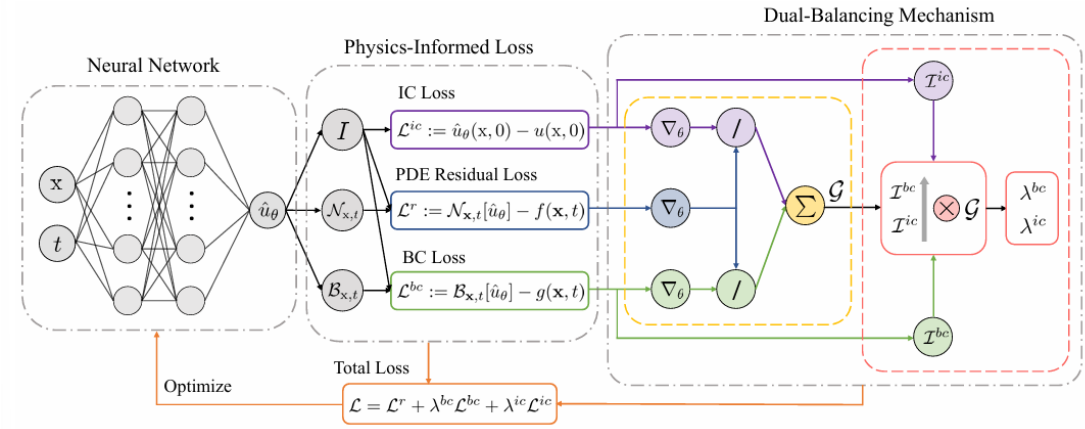
损失函数由数据拟合损失、边界条件损失和初始条件损失构成：

$$\mathcal{L} = \mathcal{L}^r + \lambda^{bc}\mathcal{L}^{bc} + \lambda^{ic}\mathcal{L}^{ic}$$

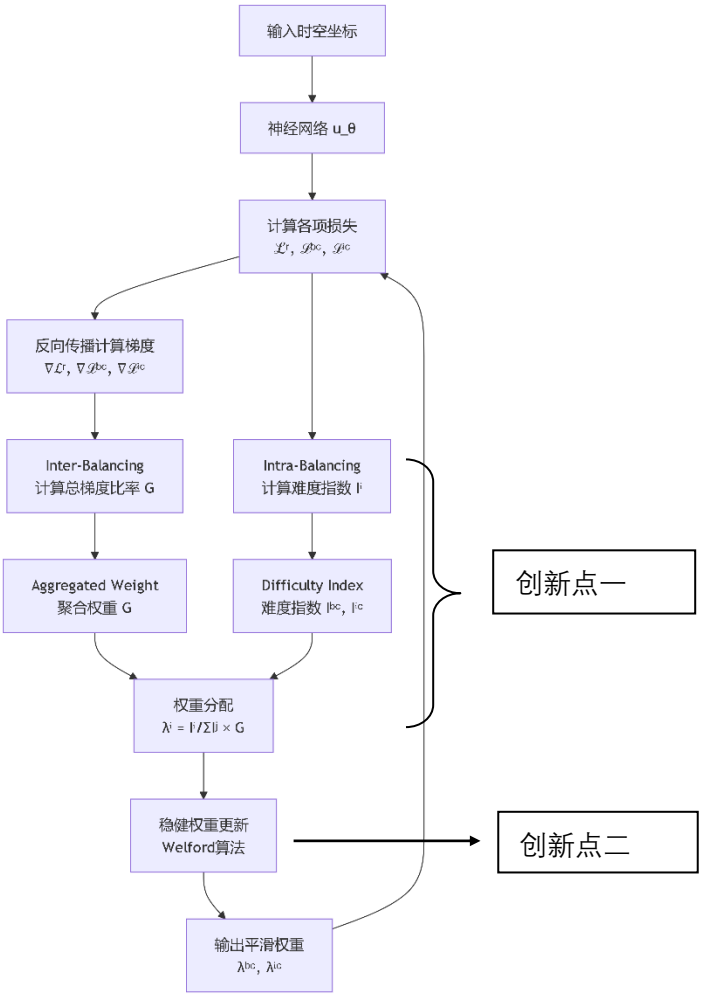
参数 θ 的更新迭代：

$$\theta = \theta - \eta\nabla_\theta\mathcal{L}^r - \eta\sum_{i=1}^{M}\lambda^i\nabla_\theta\mathcal{L}^i$$

2.5 论文中的逻辑框架图



2.6 论文方法流程图



3.项目代码

源码链接： https://github.com/chenhong-zhou/DualBalanced-PINNs

3.1 创建环境（安装说明, 数据集准备，依赖说明，运行配置命令行等）

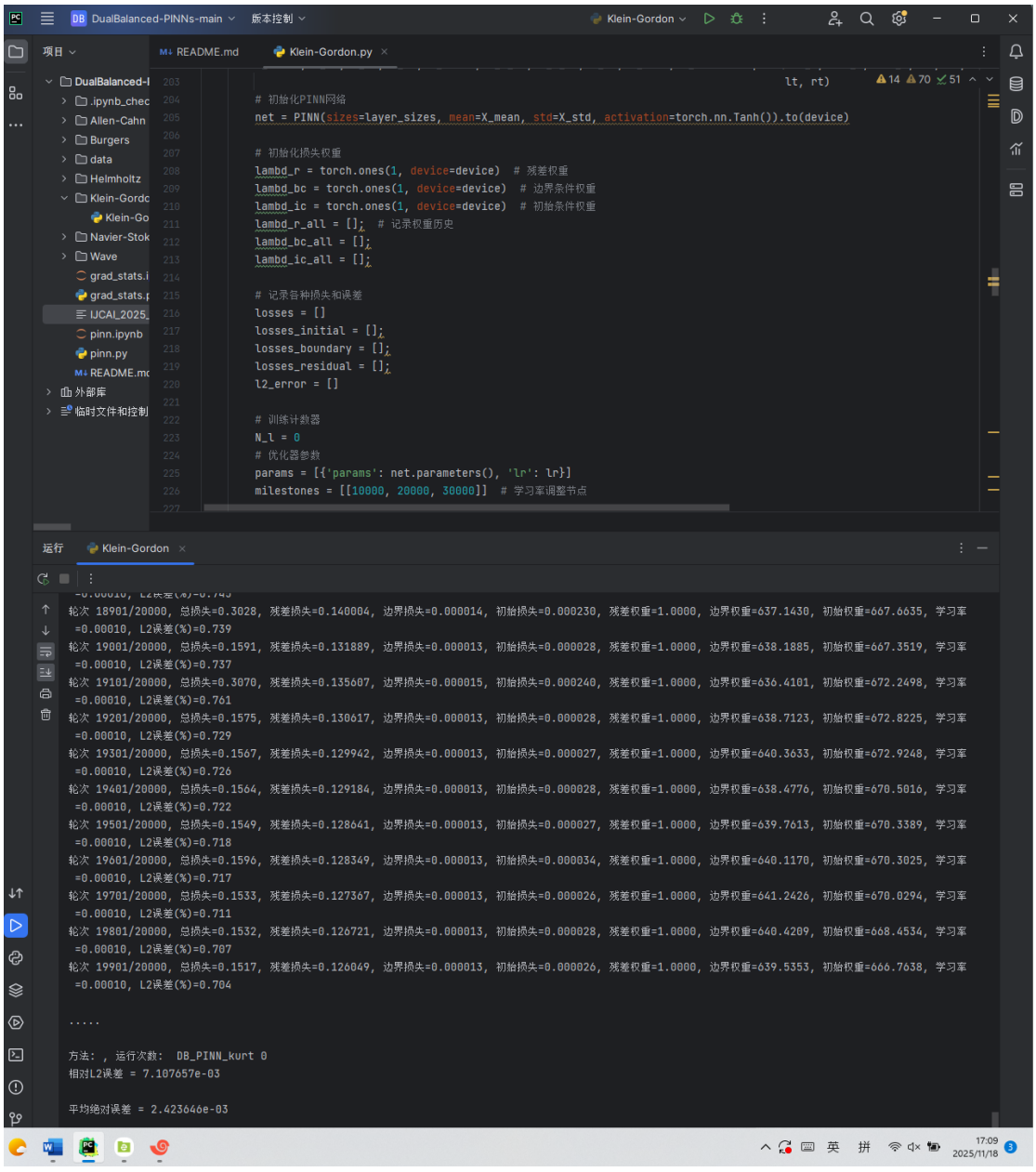
conda create -n PI python=3.10.19 -y
conda activate PI

nvcc --version #检查 cuda 版本

conda install pytorch==2.5.0 torchvision==0.20.0 torchaudio==2.5.0 pytorch-cuda=11.8 -c pytorch -c nvidia #在官网找到合适的版本指令并下载

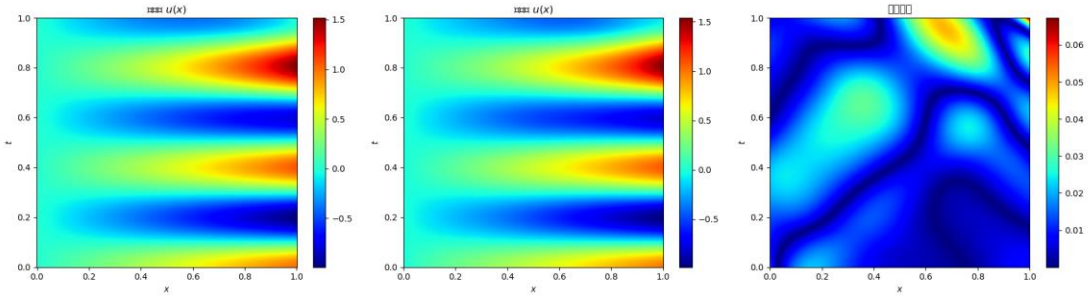
（其余所需包自行通过 conda install 或者 pip install 自行下载）

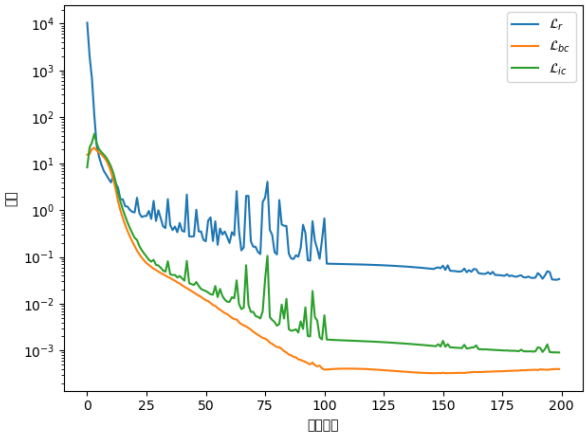
3.2 测试结果

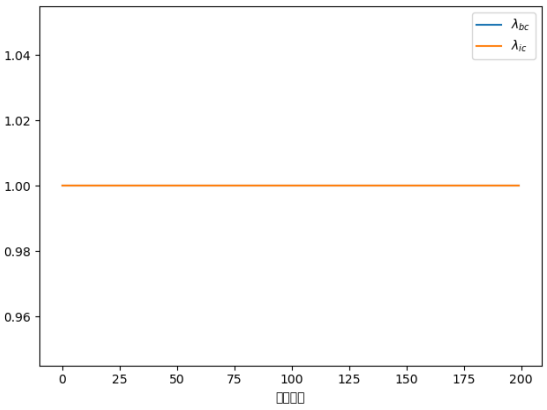


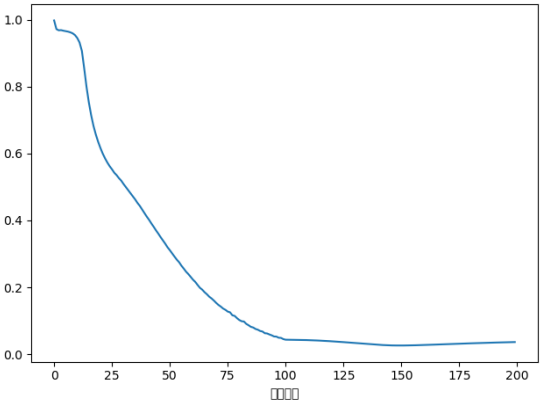
由于原文作者的代码中采用七种不同的损失权重分配方法，每一种都有四张输出图片，共 28 张图片，如果外加七个公式一共将近两百张图片，故这里着重分析介绍文中提到的 Klein-Gordon 方程，其中用于对比的损失权重分配方法分别是：0-等权重，1-均值法，2-标准差法，3-峰度法，DB_PINN 的三种变体（'DB_PINN_mean'，'DB_PINN_std'，'DB_PINN_kurt'），每种方法的四张输出图像依次为 1.精确解、预测解和绝对误差　2. 损失曲线　　3. 学习到的权重　　4，L2 误差

方法 0

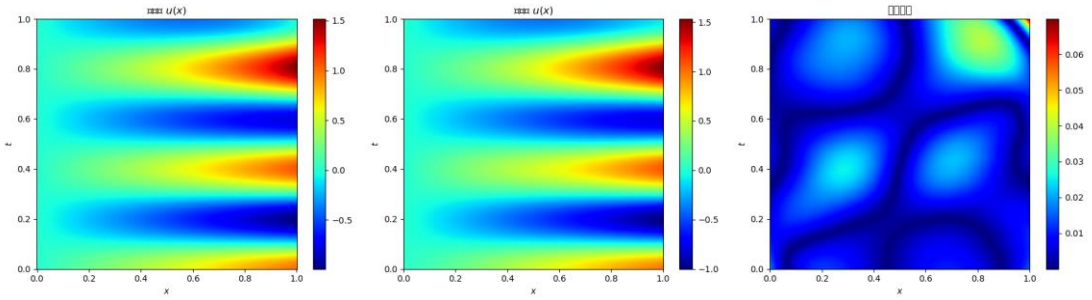

□□□ u(x)
□□□ u(x)
□□□□
x
t

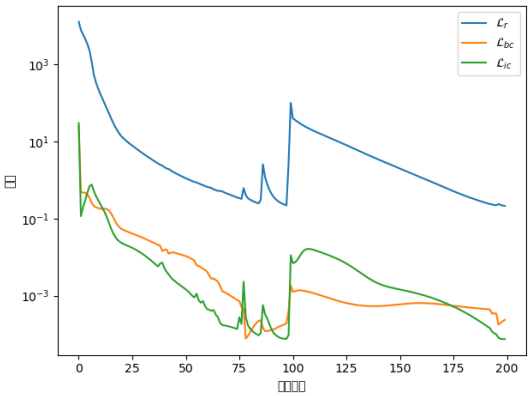

$\mathcal{L}_r$
$\mathcal{L}_{bc}$
$\mathcal{L}_{ic}$
□□
□□□□

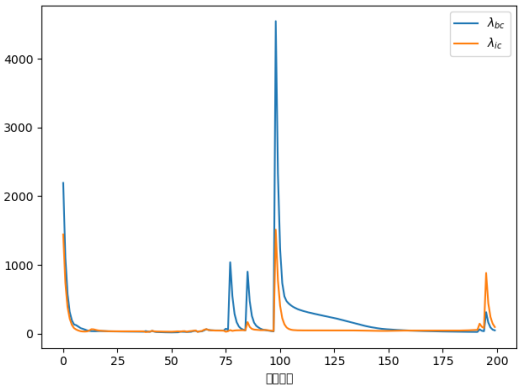

λ_{bc}
λ_{ic}
□□□□

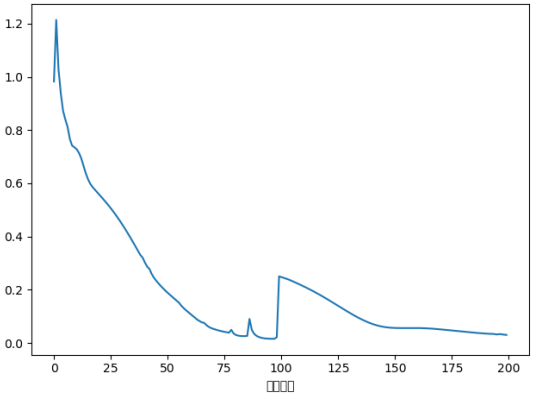

□□□□

方法 1

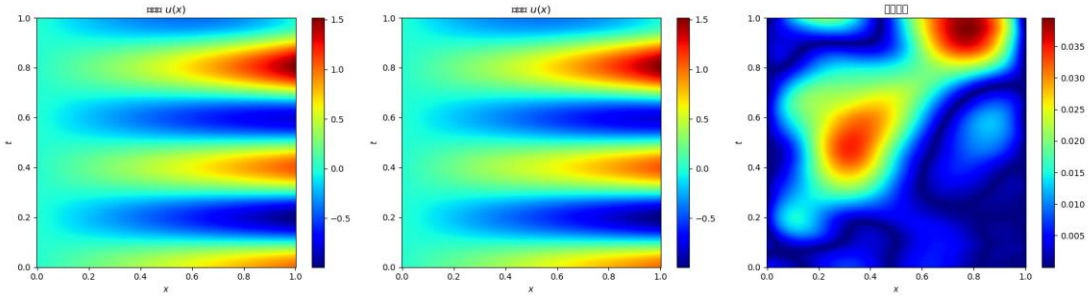


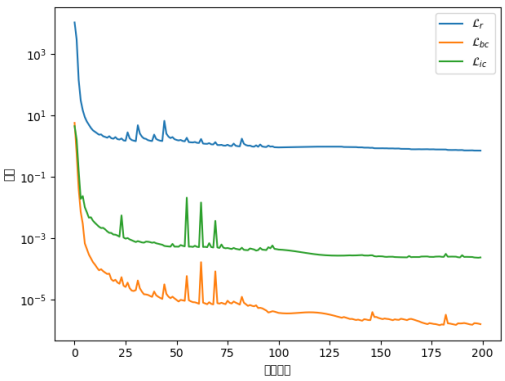


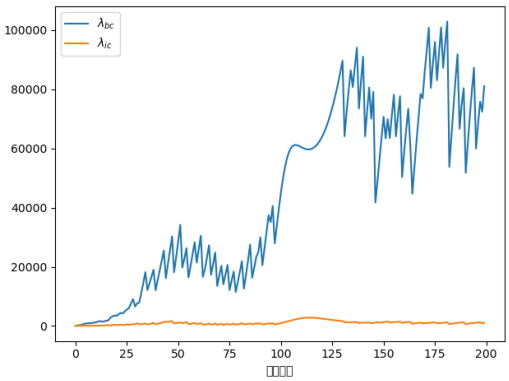


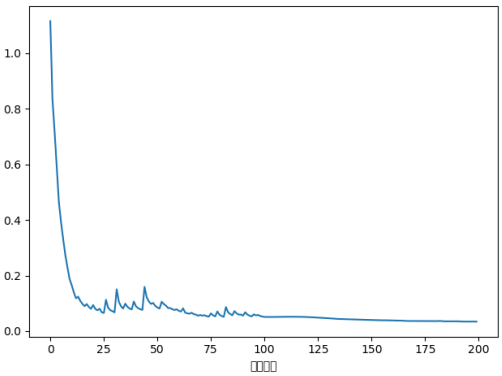


方法 2

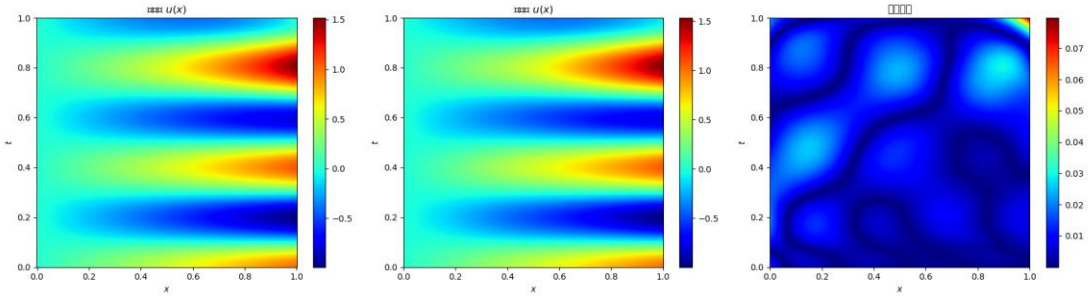


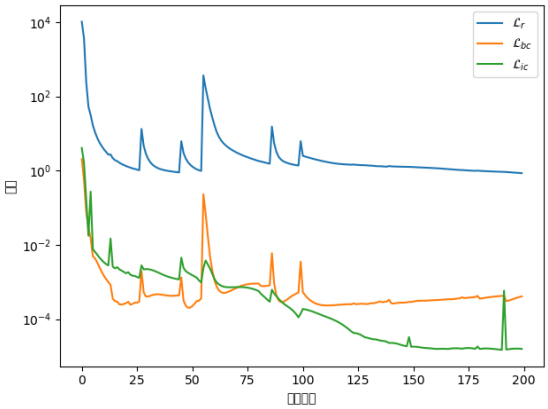

$\mathcal{L}_r$
$\mathcal{L}_{bc}$
$\mathcal{L}_{ic}$
损失
迭代次数

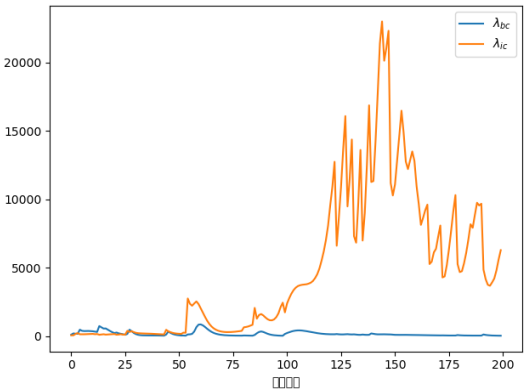

λ_{bc}
λ_{ic}
迭代次数

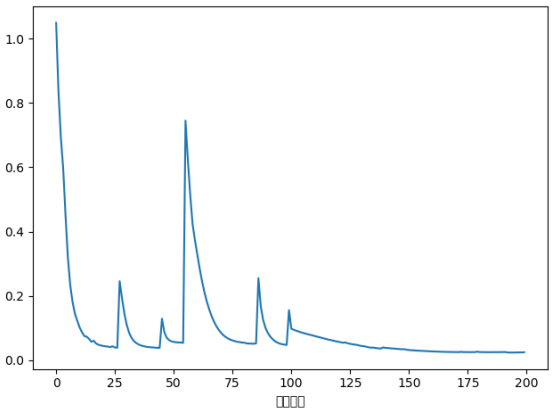

迭代次数

方法 3

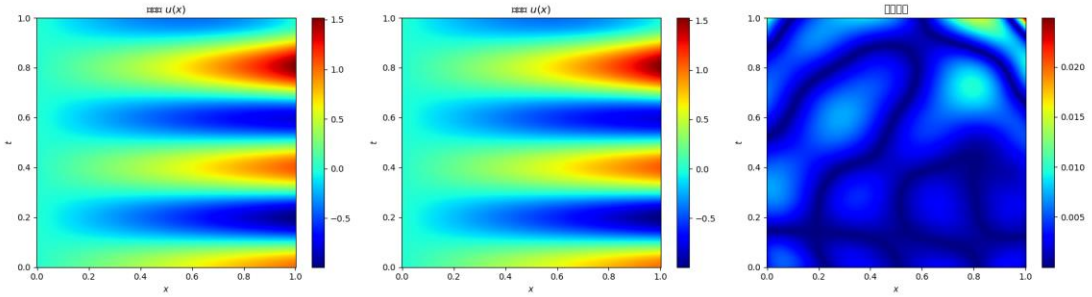


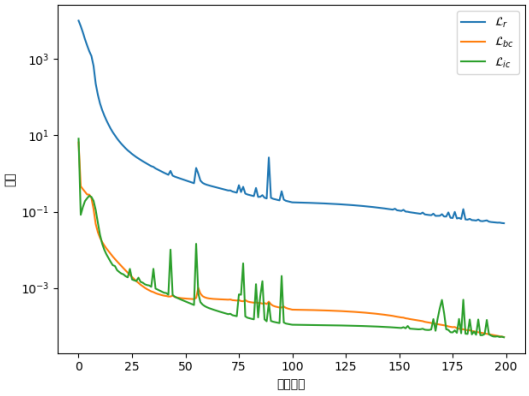


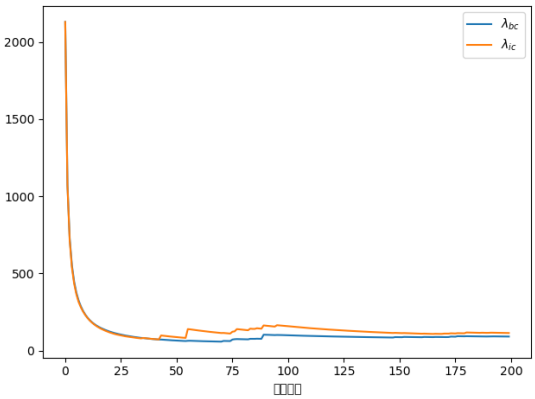


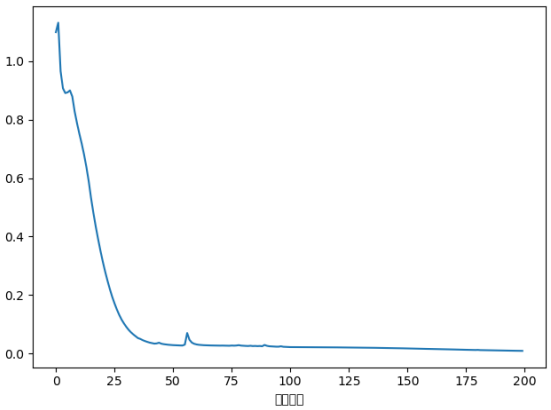


方法 4

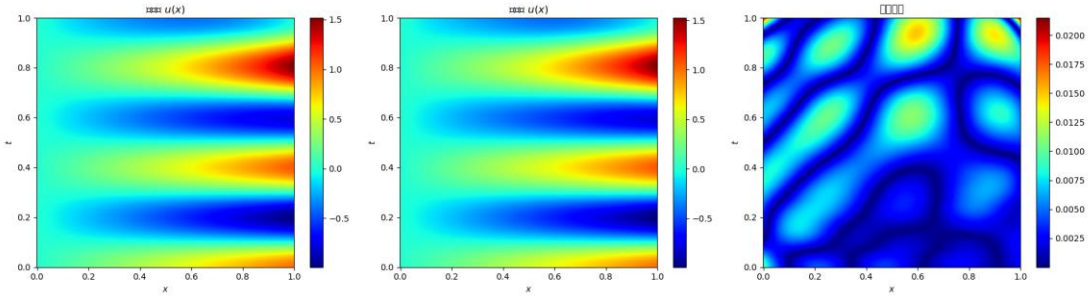


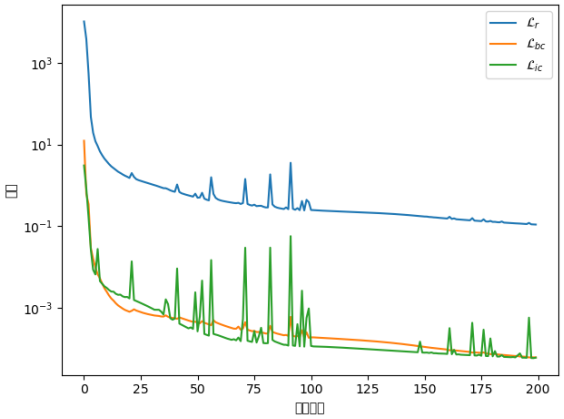


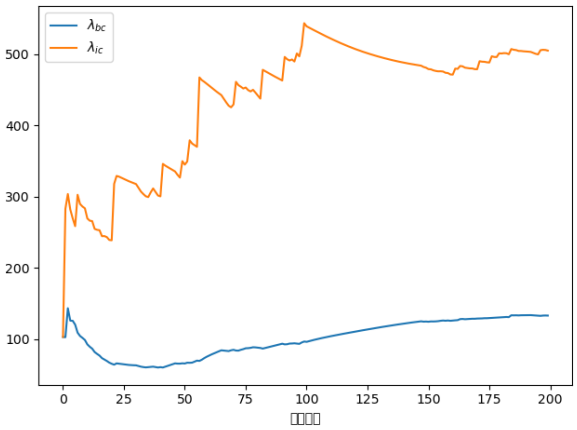


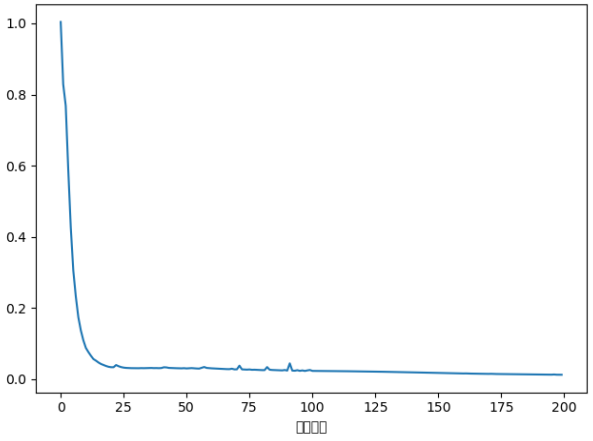


方法 5

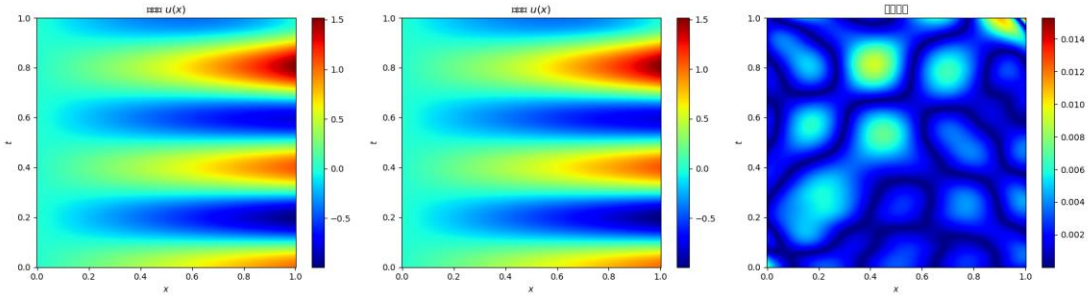


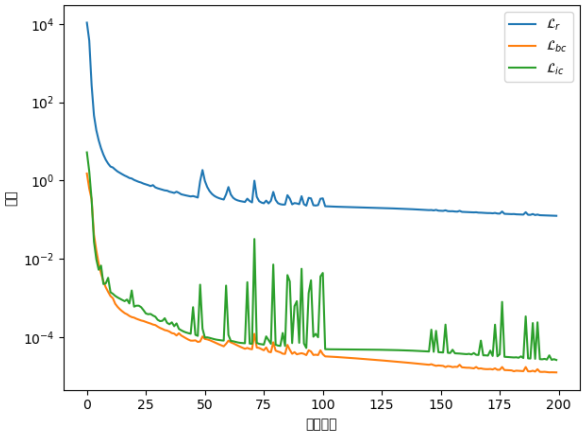

$\mathcal{L}_r$
$\mathcal{L}_{bc}$
$\mathcal{L}_{ic}$
损失
迭代次数

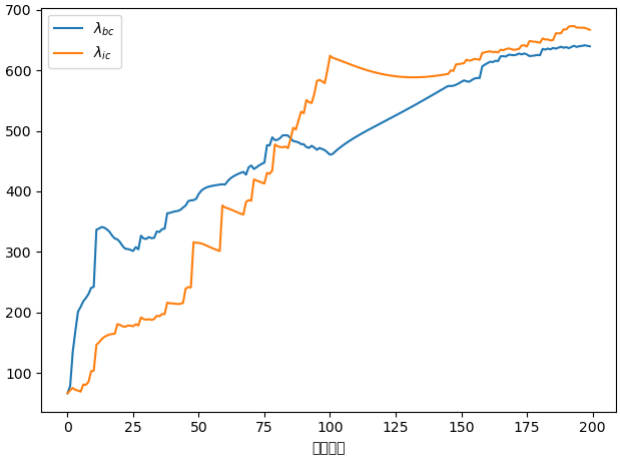

λ_{bc}
λ_{ic}
迭代次数

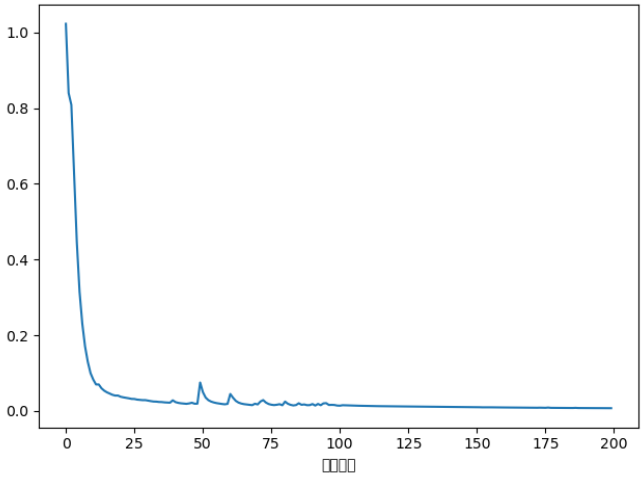

迭代次数

方法 6




$\mathcal{L}_r$
$\mathcal{L}_{bc}$
$\mathcal{L}_{ic}$
损失
迭代次数


λ_{bc}
λ_{ic}
迭代次数


迭代次数

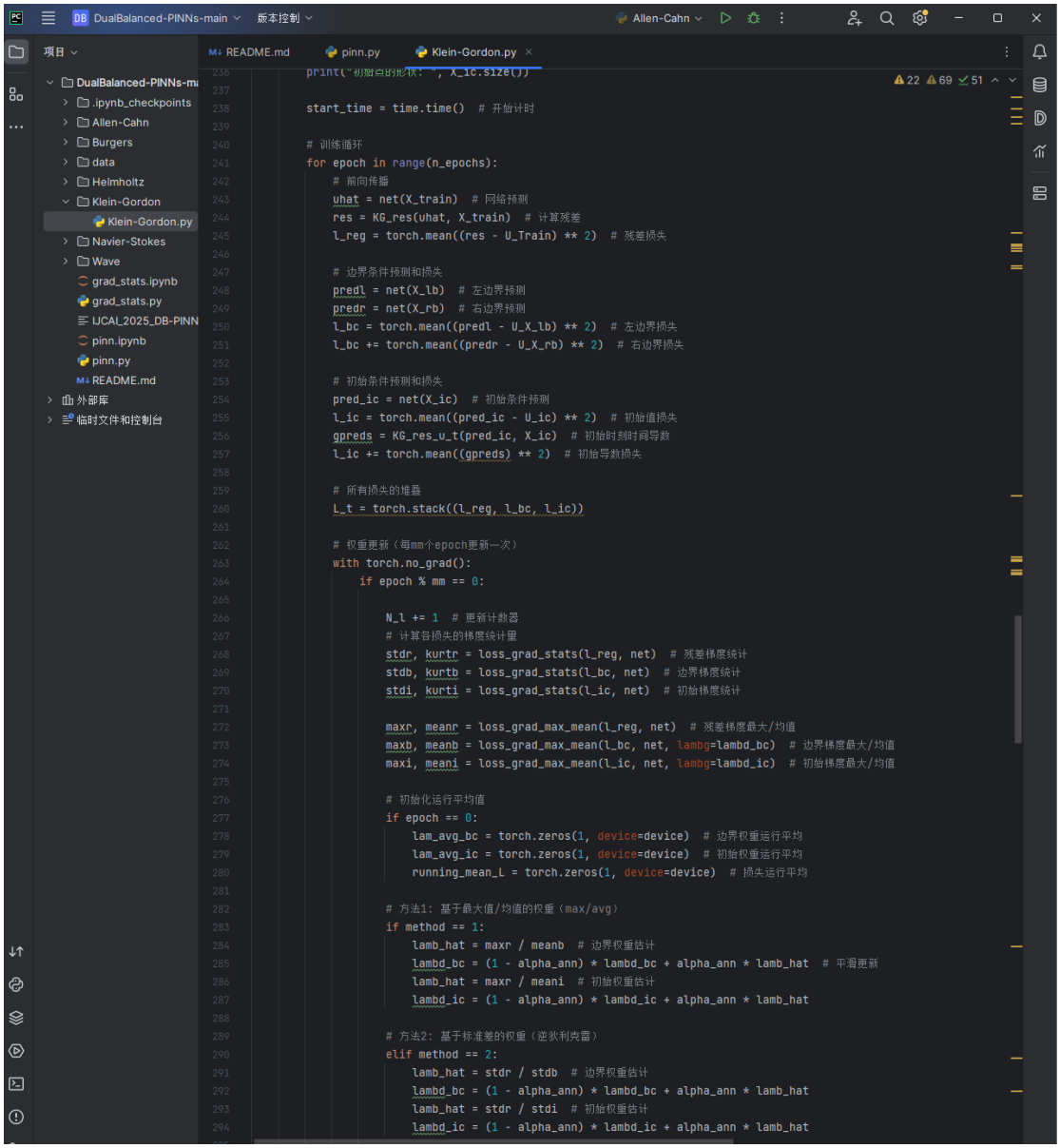
由上述运行结果可知（彩色图片是将前两幅图进行对比，看精确解和预测解的差别大小，图片颜色越接近越准确，第三张彩色图则是看二者的绝对误差，颜色越蓝误差越小，越红误差越大），综合来看，DB_PINN_mean（也就是文中理论部分提出的方法）最优

3.3 论文公式对应代码

注：由于作者通过七个不同的方程进行实验，并在文中只采用三个方程进行实验结果验证（其余为补充实验放在代码里，但总体网络架构基本类似），故以下每个方程的具体代码实现只列举文中多次提到的 Klein-Gordon 方程作为代表

$$\mathcal{G} = \sum_{i=1}^{M} \frac{\max\{|\nabla_\theta \mathcal{L}^r|\}}{|\nabla_\theta \lambda^i \mathcal{L}^i|} \tag{1}$$

在 DualBalanced-PINNs-main/Klein-Gordon.py 文件中的 241-304 行进行了前置运算，并分别在 308、322、338 进行求和得到总权重（文中公式只提到了公式一这一种方法，另两个方法的理论描述由实验部分引出）



```
            # 方法3: 基于峰度的权重
            elif method == 3:
                covr = stdr / kurtr  # 残差变异系数
                covb = stdb / kurtb  # 边界变异系数
                covi = stdi / kurti  # 初始变异系数
                lamb_hat = covr / covb  # 边界权重估计
                lambd_bc = (1 - alpha_ann) * lambd_bc + alpha_ann * lamb_hat
                lamb_hat = covr / covi  # 初始权重估计
                lambd_ic = (1 - alpha_ann) * lambd_ic + alpha_ann * lamb_hat
```

```
        # DB-PINN均值变体
        elif method == 'DB_PINN_mean':
            hat_all = maxr / meanb + maxr / meani  # 总权重预算
            mean_param = (1. - 1 / N_l)  # 运行平均参数
            running_mean_L = mean_param * running_mean_L + (1 - mean_param) * L_t.detach()  # 更新损失运行平均
            l_t_vector = L_t / running_mean_L  # 难度指数（当前损失/历史平均）
            hat_bc = hat_all * l_t_vector[1] / (l_t_vector[1] + l_t_vector[2])  # 按难度分配边界权重
            hat_ic = hat_all * l_t_vector[2] / (l_t_vector[1] + l_t_vector[2])  # 按难度分配初始权重
            # Welford算法更新权重
            lambd_bc = lam_avg_bc + 1 / N_l * (hat_bc - lam_avg_bc)
            lambd_ic = lam_avg_ic + 1 / N_l * (hat_ic - lam_avg_ic)
            lam_avg_bc = lambd_bc  # 更新运行平均
            lam_avg_ic = lambd_ic

        # DB-PINN标准差变体
        elif method == 'DB_PINN_std':
            hat_all = stdr / stdb + stdr / stdi  # 总权重预算
            mean_param = (1. - 1 / N_l)
            running_mean_L = mean_param * running_mean_L + (1 - mean_param) * L_t.detach()
            l_t_vector = L_t / running_mean_L  # 难度指数
            hat_bc = hat_all * l_t_vector[1] / (l_t_vector[1] + l_t_vector[2])
            hat_ic = hat_all * l_t_vector[2] / (l_t_vector[1] + l_t_vector[2])
            lambd_bc = lam_avg_bc + 1 / N_l * (hat_bc - lam_avg_bc)
            lambd_ic = lam_avg_ic + 1 / N_l * (hat_ic - lam_avg_ic)
            lam_avg_bc = lambd_bc
            lam_avg_ic = lambd_ic

        # DB-PINN峰度变体
        elif method == 'DB_PINN_kurt':
            covr = stdr / kurtr  # 残差变异系数
            covb = stdb / kurtb  # 边界变异系数
            covi = stdi / kurti  # 初始变异系数
            hat_all = covr / covb + covr / covi  # 总权重预算
            mean_param = (1. - 1 / N_l)
            running_mean_L = mean_param * running_mean_L + (1 - mean_param) * L_t.detach()
            l_t_vector = L_t / running_mean_L  # 难度指数
            hat_bc = hat_all * l_t_vector[1] / (l_t_vector[1] + l_t_vector[2])
            hat_ic = hat_all * l_t_vector[2] / (l_t_vector[1] + l_t_vector[2])
            lambd_bc = lam_avg_bc + 1 / N_l * (hat_bc - lam_avg_bc)
            lambd_ic = lam_avg_ic + 1 / N_l * (hat_ic - lam_avg_ic)
            lam_avg_bc = lambd_bc
            lam_avg_ic = lambd_ic
```

$$\mathcal{I}_t = \frac{\mathcal{L}_t}{\mu_{\mathcal{L}_t}} \tag{2}$$

在 DualBalanced-PINNs-main/Klein-Gordon.py 文件中的第 311、325 以及 341 行对难度指数进行了计算

```
    # DB-PINN均值变体
    elif method == 'DB_PINN_mean':
        hat_all = maxr / meanb + maxr / meani  # 总权重预算
        mean_param = (1. - 1 / N_l)  # 运行平均参数
        running_mean_L = mean_param * running_mean_L + (1 - mean_param) * L_t.detach()  # 更新损失运行平均
        l_t_vector = L_t / running_mean_L  # 难度指数（当前损失/历史平均）
        hat_bc = hat_all * l_t_vector[1] / (l_t_vector[1] + l_t_vector[2])  # 按难度分配边界权重
        hat_ic = hat_all * l_t_vector[2] / (l_t_vector[1] + l_t_vector[2])  # 按难度分配初始权重
        # Welford算法更新权重
        lambd_bc = lam_avg_bc + 1 / N_l * (hat_bc - lam_avg_bc)
        lambd_ic = lam_avg_ic + 1 / N_l * (hat_ic - lam_avg_ic)
        lam_avg_bc = lambd_bc  # 更新运行平均
        lam_avg_ic = lambd_ic

    # DB-PINN标准差变体
    elif method == 'DB_PINN_std':
        hat_all = stdr / stdb + stdr / stdi  # 总权重预算
        mean_param = (1. - 1 / N_l)
        running_mean_L = mean_param * running_mean_L + (1 - mean_param) * L_t.detach()
        l_t_vector = L_t / running_mean_L  # 难度指数
        hat_bc = hat_all * l_t_vector[1] / (l_t_vector[1] + l_t_vector[2])
        hat_ic = hat_all * l_t_vector[2] / (l_t_vector[1] + l_t_vector[2])
        lambd_bc = lam_avg_bc + 1 / N_l * (hat_bc - lam_avg_bc)
        lambd_ic = lam_avg_ic + 1 / N_l * (hat_ic - lam_avg_ic)
        lam_avg_bc = lambd_bc
        lam_avg_ic = lambd_ic

    # DB-PINN峰度变体
    elif method == 'DB_PINN_kurt':
        covr = stdr / kurtr  # 残差变异系数
        covb = stdb / kurtb  # 边界变异系数
        covi = stdi / kurti  # 初始变异系数
        hat_all = covr / covb + covr / covi  # 总权重预算
        mean_param = (1. - 1 / N_l)
        running_mean_L = mean_param * running_mean_L + (1 - mean_param) * L_t.detach()
        l_t_vector = L_t / running_mean_L  # 难度指数
        hat_bc = hat_all * l_t_vector[1] / (l_t_vector[1] + l_t_vector[2])
        hat_ic = hat_all * l_t_vector[2] / (l_t_vector[1] + l_t_vector[2])
        lambd_bc = lam_avg_bc + 1 / N_l * (hat_bc - lam_avg_bc)
        lambd_ic = lam_avg_ic + 1 / N_l * (hat_ic - lam_avg_ic)
        lam_avg_bc = lambd_bc
        lam_avg_ic = lambd_ic
```

$$\hat{\lambda}^i = \frac{\mathcal{I}_t^i}{\sum_{j=1}^{M} \mathcal{I}_t^j} \times \mathcal{G}, \quad (i = 1, \ldots, M) \qquad (3)$$

在 DualBalanced-PINNs-main/Klein-Gordon.py 文件中的第 312、313，326、327 以及 342、343 行体现

```
306     # DB-PINN均值变体
307     elif method == 'DB_PINN_mean':
308         hat_all = maxr / meanb + maxr / meani  # 总权重预算
309         mean_param = (1. - 1 / N_l)  # 运行平均参数
310         running_mean_L = mean_param * running_mean_L + (1 - mean_param) * L_t.detach()  # 更新损失运行平均
311         l_t_vector = L_t / running_mean_L  # 难度指数（当前损失/历史平均）
312         hat_bc = hat_all * l_t_vector[1] / (l_t_vector[1] + l_t_vector[2])  # 按难度分配边界权重
313         hat_ic = hat_all * l_t_vector[2] / (l_t_vector[1] + l_t_vector[2])  # 按难度分配初始权重
314         # Welford算法更新权重
315         lambd_bc = lam_avg_bc + 1 / N_l * (hat_bc - lam_avg_bc)
316         lambd_ic = lam_avg_ic + 1 / N_l * (hat_ic - lam_avg_ic)
317         lam_avg_bc = lambd_bc  # 更新运行平均
318         lam_avg_ic = lambd_ic
319
320     # DB-PINN标准差变体
321     elif method == 'DB_PINN_std':
322         hat_all = stdr / stdb + stdr / stdi  # 总权重预算
323         mean_param = (1. - 1 / N_l)
324         running_mean_L = mean_param * running_mean_L + (1 - mean_param) * L_t.detach()
325         l_t_vector = L_t / running_mean_L  # 难度指数
326         hat_bc = hat_all * l_t_vector[1] / (l_t_vector[1] + l_t_vector[2])
327         hat_ic = hat_all * l_t_vector[2] / (l_t_vector[1] + l_t_vector[2])
328         lambd_bc = lam_avg_bc + 1 / N_l * (hat_bc - lam_avg_bc)
329         lambd_ic = lam_avg_ic + 1 / N_l * (hat_ic - lam_avg_ic)
330         lam_avg_bc = lambd_bc
331         lam_avg_ic = lambd_ic
332
333     # DB-PINN峰度变体
334     elif method == 'DB_PINN_kurt':
335         covr = stdr / kurtr  # 残差变异系数
336         covb = stdb / kurtb  # 边界变异系数
337         covi = stdi / kurti  # 初始变异系数
338         hat_all = covr / covb + covr / covi  # 总权重预算
339         mean_param = (1. - 1 / N_l)
340         running_mean_L = mean_param * running_mean_L + (1 - mean_param) * L_t.detach()
341         l_t_vector = L_t / running_mean_L  # 难度指数
342         hat_bc = hat_all * l_t_vector[1] / (l_t_vector[1] + l_t_vector[2])
343         hat_ic = hat_all * l_t_vector[2] / (l_t_vector[1] + l_t_vector[2])
344         lambd_bc = lam_avg_bc + 1 / N_l * (hat_bc - lam_avg_bc)
345         lambd_ic = lam_avg_ic + 1 / N_l * (hat_ic - lam_avg_ic)
346         lam_avg_bc = lambd_bc
347         lam_avg_ic = lambd_ic
```

$$\mu_{\mathcal{L}_t} = (1 - \frac{1}{t})\mu_{\mathcal{L}_{t-1}} + \frac{1}{t}\mathcal{L}_t \quad (4)$$

在 DualBalanced-PINNs-main/Klein-Gordon.py 文件中的第 310、324、340 行分别体现（如上图）

$$\lambda^i = (1 - \frac{1}{t})\lambda^i + \frac{1}{t}\hat{\lambda}^i \quad (5)$$

在 DualBalanced-PINNs-main/Klein-Gordon.py 文件中的第 315、316，328、329 以及 344、345 行分别体现（如上图）

$$u_{tt} - u_{xx} + u^3 = 0, \qquad x \in [0,1], t \in [0,1] \quad (6)$$

Klein-Gordon 方程如公式(6)所示，在 DualBalanced-PINNs-main/Klein-Gordon.py 文件中的第 109-129 行代码计算得到

```python
def KG_res(uhat, data):
    x = data[:, 0:1]  # 空间坐标
    t = data[:, 1:2]  # 时间坐标

    poly = torch.ones_like(uhat)  # 用于保持形状

    # 计算一阶导数
    du = grad(outputs=uhat, inputs=data,
              grad_outputs=torch.ones_like(uhat), create_graph=True)[0]

    dudx = du[:, 0:1]  # 对x的一阶导
    dudt = du[:, 1:2]  # 对t的一阶导

    # 计算二阶导数
    dudxx = grad(outputs=dudx, inputs=data,
                 grad_outputs=torch.ones_like(uhat), create_graph=True)[0][:, 0:1]  # 对x的二阶导
    dudtt = grad(outputs=dudt, inputs=data,
                 grad_outputs=torch.ones_like(uhat), create_graph=True)[0][:, 1:2]  # 对t的二阶导

    # Klein-Gordon方程残差: u_tt - u_xx + u^3 = 0
    residual = dudtt - dudxx + uhat ** 3
```

$$u_{tt}(x,t) - 4u_{xx}(x,t) = 0, \quad x \in [0,1], t \in [0,1] \quad (7)$$

Wave 方程如图公式(7)，在 DualBalanced-PINNs-main/Wave.py 文件中的第 115-124 行代码计算得来

```python
def Wave1D_res(uhat, data):
    x = data[:,0:1]
    y = data[:,1:2]
    du = grad(outputs=uhat, inputs=data, grad_outputs=torch.ones_like(uhat), create_graph=True)[0]
    dudx = du[:,0:1]
    dudxx = grad(outputs=dudx, inputs=data,grad_outputs=torch.ones_like(uhat), create_graph=True)[0][:,0:1]
    dudy = du[:,1:2]
    dudyy = grad(outputs=dudy, inputs=data,grad_outputs=torch.ones_like(uhat), create_graph=True)[0][:,1:2]

    residual = dudyy - 4*dudxx + 0*uhat
    return residual
```

$$u(x,t) = \sin(\pi x)\cos(2\pi t) + 0.5\sin(4\pi x)\cos(8\pi t) \quad (8)$$

公式(8)为 Wave 方程精确的解析解，在 DualBalanced-PINNs-main/Wave.py 文件中的第 30-39 行定义

```python
def u_func(X, a = 0.5, c = 2):
    """
    :param x: X = (x, t)
    """
    x = X[:,0:1]
    t = X[:,1:2]
    out = np.sin(np.pi * x) * np.cos(c * np.pi * t) + \
            a * np.sin(2 * c * np.pi* x) * np.cos(4 * c  * np.pi * t)

    return out
```

$$u_{xx} + u_{yy} + u = q(x, y),\ x \in [-1, 1], y \in [-1, 1] \quad (9)$$

Helmholtz 方程如图公式(9)所示，在 DualBalanced-PINNs-main/Helmholtz.py 文件的第 110-122 行中定义成残差形式

```python
def Helmholtz_res(uhat, data):
    """计算亥姆霍兹方程的残差"""
    x = data[:, 0:1]  # x坐标
    y = data[:, 1:2]  # y坐标

    # 计算一阶导数
    du = grad(outputs=uhat, inputs=data, grad_outputs=torch.ones_like(uhat), create_graph=True)[0]
    dudx = du[:, 0:1]  # 对x的一阶导
    dudy = du[:, 1:2]  # 对y的一阶导

    # 计算二阶导数
    dudxx = grad(outputs=dudx, inputs=data, grad_outputs=torch.ones_like(uhat), create_graph=True)[0][:, 0:1]  # 对x的二阶导
    dudyy = grad(outputs=dudy, inputs=data, grad_outputs=torch.ones_like(uhat), create_graph=True)[0][:, 1:2]  # 对y的二阶导

    # 计算源项（强迫项）
    source = - (a_1 * math.pi) ** 2 * torch.sin(a_1 * math.pi * x) * torch.sin(a_2 * math.pi * y) - \
             (a_2 * math.pi) ** 2 * torch.sin(a_1 * math.pi * x) * torch.sin(a_2 * math.pi * y) + \
             lam * torch.sin(a_1 * math.pi * x) * torch.sin(a_2 * math.pi * y)

    # 计算残差：u_xx + u_yy + λu - source
    residual = dudxx + dudyy + lam * uhat - source
    return residual
```

$$q(x, y) = -\pi^2 \sin(\pi x) \sin(4\pi y) - (4\pi)^2 \sin(\pi x) \sin(4\pi y) + \sin(\pi x) \sin(4\pi y) \quad (10)$$

方程(10)为源项，为使 Helmholtz 方程有简单的解析解形式，在 DualBalanced-PINNs-main/Helmholtz.py 文件的第 118-139 行体现

```python
def Helmholtz_res(uhat, data):
    """计算亥姆霍兹方程的残差"""
    x = data[:, 0:1]  # x坐标
    y = data[:, 1:2]  # y坐标

    # 计算一阶导数
    du = grad(outputs=uhat, inputs=data, grad_outputs=torch.ones_like(uhat), create_graph=True)[0]
    dudx = du[:, 0:1]  # 对x的一阶导
    dudy = du[:, 1:2]  # 对y的一阶导

    # 计算二阶导数
    dudxx = grad(outputs=dudx, inputs=data, grad_outputs=torch.ones_like(uhat), create_graph=True)[0][:, 0:1]  # 对x的二阶导
    dudyy = grad(outputs=dudy, inputs=data, grad_outputs=torch.ones_like(uhat), create_graph=True)[0][:, 1:2]  # 对y的二阶导

    # 计算源项（强迫项）
    source = - (a_1 * math.pi) ** 2 * torch.sin(a_1 * math.pi * x) * torch.sin(a_2 * math.pi * y) - \
             (a_2 * math.pi) ** 2 * torch.sin(a_1 * math.pi * x) * torch.sin(a_2 * math.pi * y) + \
             lam * torch.sin(a_1 * math.pi * x) * torch.sin(a_2 * math.pi * y)

    # 计算残差：u_xx + u_yy + λu - source
    residual = dudxx + dudyy + lam * uhat - source
    return residual
```

$$u(x, y) = \sin(\pi x) \sin(4\pi y) \quad (11)$$

Helmholtz 方程精确解如公式(11)所示，在 DualBalanced-PINNs-main/Helmholtz.py 文件的第 37-39 行定义

```python
def generate_u(x):
    """生成精确解：u(x,y) = sin(πx) * sin(4πy)"""
    return np.sin(a_1 * np.pi * x[:, 0:1]) * np.sin(a_2 * np.pi * x[:, 1:2])
```