`PINN` class for a 3D problem. This class will encompass the neural network for displacement, the computation of strain from displacement, the computation of stress from strain using the constitutive relations, and the computation of the divergence of stress to enforce equilibrium.

Here's a rough outline:

```
import torch
import torch.nn as nn
import torch.autograd as autograd

class PINN(nn.Module):
    def __init__(self):
        super(PINN, self).__init__()

        # Define the neural network layers for displacement
        self.fc1 = nn.Linear(3, 50)  # Assuming input is (x, y, z)
        self.fc2 = nn.Linear(50, 50)
        self.fc3 = nn.Linear(50, 3)  # Output is (u, v, w) displacements

    def forward(self, x):
        # Forward pass to compute displacements
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        displacement = self.fc3(x)
        return displacement

    def compute_strain(self, positions):
        # Compute strain components using autograd
        displacements = self.forward(positions)
        epsilon_xx, = autograd.grad(displacements[:, 0], positions[:, 0],
grad_outputs=torch.ones_like(displacements[:, 0]), create_graph=True)
        epsilon_yy, = autograd.grad(displacements[:, 1], positions[:, 1],
grad_outputs=torch.ones_like(displacements[:, 1]), create_graph=True)
        epsilon_zz, = autograd.grad(displacements[:, 2], positions[:, 2],
grad_outputs=torch.ones_like(displacements[:, 2]), create_graph=True)
        epsilon_xy, = autograd.grad(displacements[:, 0], positions[:, 1],
grad_outputs=torch.ones_like(displacements[:, 0]), create_graph=True)
        epsilon_xz, = autograd.grad(displacements[:, 0], positions[:, 2],
grad_outputs=torch.ones_like(displacements[:, 0]), create_graph=True)
        epsilon_yz, = autograd.grad(displacements[:, 1], positions[:, 2],
grad_outputs=torch.ones_like(displacements[:, 1]), create_graph=True)
        return epsilon_xx, epsilon_yy, epsilon_zz, epsilon_xy, epsilon_xz,
epsilon_yz

    def compute_stress(self, epsilon, C):
        # Compute stress components using constitutive relations
        # Assuming you have a function 'constitutive_relation' that computes
stress from strain
        sigma = constitutive_relation(epsilon, C)
        return sigma

    def compute_loss(self, positions, C):
```

```
        # Compute divergence of stress and enforce equilibrium
        epsilon = self.compute_strain(positions)
        sigma = self.compute_stress(epsilon, C)
        div_sigma_x, = autograd.grad(sigma[:, 0], positions[:, 0],
grad_outputs=torch.ones_like(sigma[:, 0]), create_graph=True)
        div_sigma_y, = autograd.grad(sigma[:, 1], positions[:, 1],
grad_outputs=torch.ones_like(sigma[:, 1]), create_graph=True)
        div_sigma_z, = autograd.grad(sigma[:, 2], positions[:, 2],
grad_outputs=torch.ones_like(sigma[:, 2]), create_graph=True)

        # Loss function to enforce equilibrium (divergence of stress equals zero)
        loss = torch.mean(div_sigma_x**2 + div_sigma_y**2 + div_sigma_z**2)
        return loss

# Helper function for constitutive relation (modify as needed)
def constitutive_relation(epsilon, C):
    # Compute stress from strain using the constitutive matrix C
    # Modify this function based on your specific constitutive relations
    sigma = torch.matmul(C, epsilon)
    return sigma

# Instantiate the PINN model
model = PINN()
```

This is a basic structure for the `PINN` class. You'll need to adjust the neural network architecture, the constitutive relation function, and other details based on your specific problem and requirements.

¡Claro! `autograd.grad` es una función poderosa en PyTorch que nos permite calcular derivadas de tensores con respecto a otros tensores. Es esencialmente una forma de calcular el producto del Jacobiano por un vector, sin tener que calcular explícitamente el Jacobiano en sí.

¿Qué es el Jacobiano?

Dado un vector de funciones (\mathbf{f}(\mathbf{x})) con respecto a un vector de variables (\mathbf{x}), el Jacobiano (\mathbf{J}) es una matriz donde el elemento en la fila (i) y la columna (j) es la derivada parcial de la función (i)-ésima con respecto a la variable (j)-ésima:

[
\mathbf{J} = \begin{bmatrix}
\frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \dots \\
\frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \dots \\
\vdots & \vdots & \ddots \\
\end{bmatrix}
]

¿Qué hace `autograd.grad`?

La función `autograd.grad` de PyTorch calcula derivadas de tensores con respecto a otros tensores. Específicamente, si tienes un tensor de salida (\mathbf{y}) y un tensor de entrada (\mathbf{x}),

`autograd.grad` te permite calcular el producto (\mathbf{J} \cdot \mathbf{v}) para un vector dado (\mathbf{v}), sin tener que calcular el Jacobiano (\mathbf{J}) explícitamente.

Ejemplo:

Supongamos que tienes una función (\mathbf{y} = \mathbf{f}(\mathbf{x})) y quieres calcular el producto del Jacobiano por un vector (\mathbf{v}). En PyTorch, esto se haría así:

```
y = f(x)
v = torch.tensor([...])  # Tu vector v
Jv, = autograd.grad(y, x, grad_outputs=v, create_graph=True)
```

El argumento `grad_outputs` es el vector (\mathbf{v}) con el que quieres multiplicar el Jacobiano. El resultado, `Jv`, es el producto del Jacobiano por (\mathbf{v}).

Ventajas:

El cálculo explícito del Jacobiano puede ser costoso en términos de memoria y tiempo de cálculo, especialmente para redes neuronales grandes. Al usar `autograd.grad` para calcular directamente el producto (\mathbf{J} \cdot \mathbf{v}), puedes evitar este costo y hacer que el cálculo sea más eficiente.

En resumen, `autograd.grad` es una herramienta esencial en PyTorch que te permite calcular derivadas de manera eficiente, especialmente útil para aplicaciones como las redes neuronales informadas por la física (PINN).

La divergencia de un tensor de segundo orden (tensor 2D) no se define directamente como lo hace para un campo vectorial. Sin embargo, lo que a menudo se quiere en el contexto de la mecánica de medios continuos es la divergencia de un campo tensorial, que es un campo vectorial resultante de tomar la divergencia de cada fila (o columna, dependiendo de la convención) del tensor en cada punto.

Dado un tensor de segundo orden (\mathbf{T}) en 3D, cuyas componentes son (T_{ij}), la divergencia de (\mathbf{T}) se define como:

$$\text{div}(\mathbf{T}) = \begin{bmatrix} \frac{\partial T_{11}}{\partial x} + \frac{\partial T_{12}}{\partial y} + \frac{\partial T_{13}}{\partial z} \\ \frac{\partial T_{21}}{\partial x} + \frac{\partial T_{22}}{\partial y} + \frac{\partial T_{23}}{\partial z} \\ \frac{\partial T_{31}}{\partial x} + \frac{\partial T_{32}}{\partial y} + \frac{\partial T_{33}}{\partial z} \end{bmatrix}$$

A continuación, te proporciono una función en PyTorch que calcula la divergencia de un tensor de segundo orden en 3D:

Esto está más mal que bien >

```python
import torch.autograd as autograd

def divergence_2D_tensor(T, x):
    """
    Calcula la divergencia de un tensor 2D en 3D.

    Parámetros:
    - T: Tensor de PyTorch de tamaño (3, 3) representando el tensor 2D.
    - x: Tensor de PyTorch de tamaño (3,) representando las coordenadas (x,
y, z).

    Retorna:
    - div_T: Tensor de PyTorch de tamaño (3,) representando la divergencia
del tensor T.
    """
    div_T = torch.zeros(3)

    for i in range(3):
        div_T[i], = autograd.grad(T[i, 0], x[0], retain_graph=True,
create_graph=True)
        div_T[i] += autograd.grad(T[i, 1], x[1], retain_graph=True,
create_graph=True)[0]
        div_T[i] += autograd.grad(T[i, 2], x[2], retain_graph=True,
create_graph=True)[0]

    return div_T
```

Para usar esta función, asegúrate de que el tensor `x` tenga `requires_grad=True` para que PyTorch pueda calcular las derivadas con respecto a él.

Dudas

- **¿Por qué solo tenemos tensiones sobre nos nodos de la superficie?**
- **¿¿Pasa algo si no aplico las condiciones de contorno solo en los nodos de las fuerzas, sino que lo aplico en todos??**
  No me parece tan loco, puesto a saber las tensiones, las impongo todas. De hecho imponerlas solo en la zona de fuerzas no termina de convencerme. Si esa fuerza la quisiéramos imponer correctamente, pues no se, quizá podríamos descomponer la tensión y solo indicarle la tangente a la superficie....
  Por el momento:

1. Aplicar las tensiones como NBC solo en los nodos de las fuerzas, pero en todos!!!
2. Aplicar las tensiones sobre todos o algunos de los nodos conocidos como BC.
3. La data son desplazamientos de todos los nodos conocidos

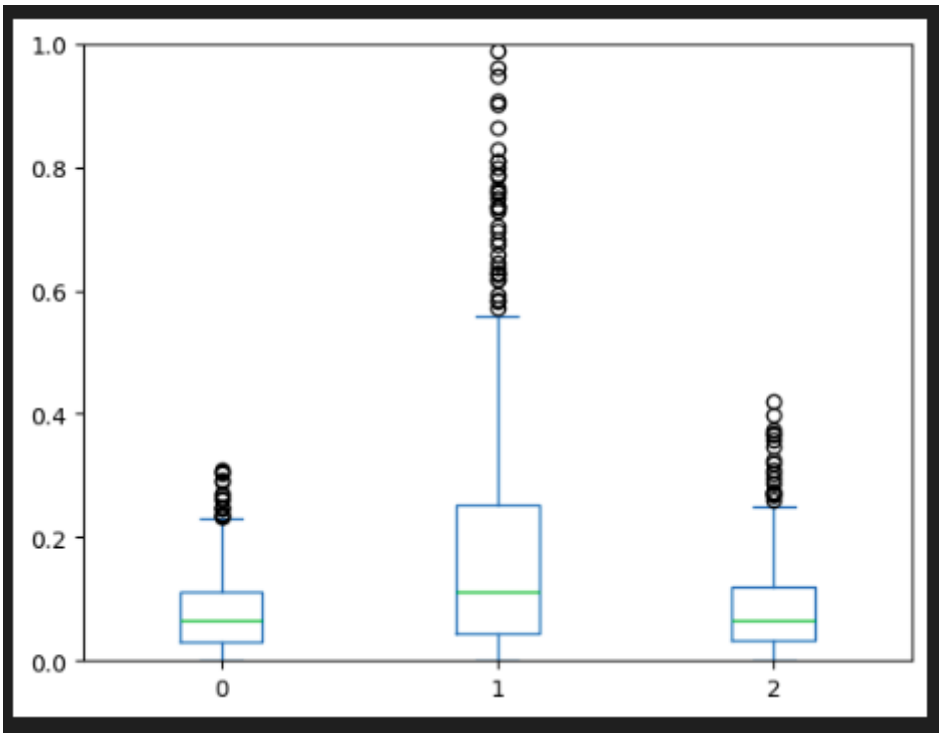
Sobre la version mixedForm:

1. Imponer la EqsConst sobre todos los nodos del dominio, o almenos los de la superficie (yo diría todos, los collocation points).
2. Las BC siguen siendo las sigmas y los puntos fijos. ¿O no pongo las sigmas, que ya están impuestas en la loss EXP?
3. La data son desplazamientos y tensiones de la superficie.

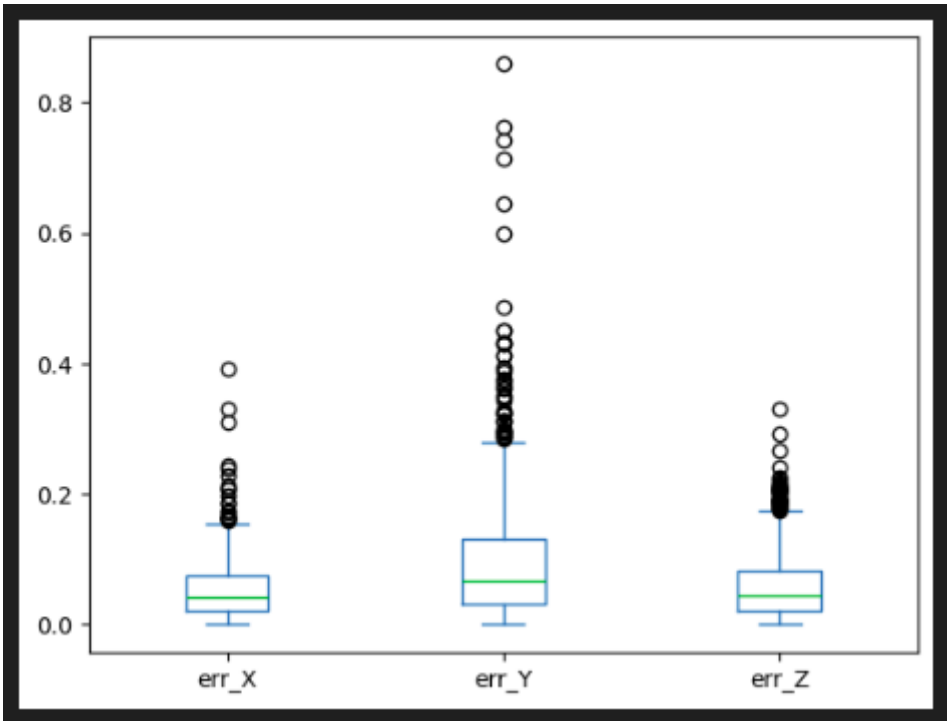
Podría ser interesante, sacar nodos interpolando ente nodos, y plotear igualmente en las superficies el valor dado por la red en esos puntos. Así a modo de aumentar la precisión de la simulación.

Está pendiente el tema de las normalizaciones...

Caso mixto



Esto es con lo de siempre.



Metindole el tema de la Ec constitutiva va mejor.