Nagyon jó ötlet most megállni és „lefagyasztani” az architektúrát 💡
Az alábbi összefoglaló kifejezetten arra készült, hogy pár nap múlva gyorsan vissza tudj kapcsolódni.

Nem implementációs részletek, hanem **struktúra + tervezési döntések + indoklás**.

🎯 FŐ VONAL

```
assemble_operator
        ↓
SystemMatrix  (lehet nxm)
        ↓
blokk-összeállítás  [A  B'; B  C]
        ↓
solveMultiFields(K, F; supports=...)
```

Cél:

- egymezős eset változatlan marad
- többmezős természetes kiterjesztés
- nincs fix u,p
- tetszőleges számú mező
- nincs redundáns információ

1️⃣ assemble_operator

Filozófia

Az operátor már most is két mező között épül:

```
assemble_operator(Prow, Pcol, Op_row, Op_col; coefficient=...)
```

Ez természetesen nxm mátrixot ad.

Mit kell mostantól tudnia a visszatérő SystemMatrix-nak?

A visszaadott `SystemMatrix` tartalmazza:

- `A::SparseMatrixCSC`
- `row_problem::Problem`
- `col_problem::Problem`

Egymezős esetben:

```
row_problem = col_problem = P
```

Kereszt operátornál:

```
row_problem = Pp
col_problem = Pu
```

Ez kulcsfontosságú, mert a blokk-összeállítás innen tudja,
melyik mező melyikhez kapcsolódik.

2️⃣ SystemMatrix kibővített koncepció

A jelenlegi:

```
SystemMatrix(A, model)
```

Jövőbeli (logikai szerkezet):

```
SystemMatrix
    A
    row_problem
    col_problem
    problems   # csak blokk esetben
    offsets    # csak blokk esetben
```

Egymezős kompatibilitás

Ha row_problem == col_problem:

```
problems = [model]
offsets = [0]
```

Semmi nem törik.

3️⃣ Blokk-összeállítás

Cél:

```
K = SystemMatrix([A  B';
                  B  C])
```

ahol:

- A: (Pu, Pu)
- B: (Pp, Pu)
- C: (Pp, Pp)

Mit csinál a blokk-konstruktor?

1. Végigmegy minden blokkon
2. Összegyűjti az összes unique Problem-et
3. Meghatározza a sorrendet (first appearance rule)
4. Offseteket számol:

```
offset[i] = sum(ndofs(problems[1:i-1]))
```

5. Összerakja a nagy sparse mátrixot
6. Tárolja:

```
K.problems
K.offsets
```

Eredmény

A solver már nem találgat.
A blokkstruktúra az assembly-ből jön.
Nincs redundáns Pu/Pp megadás.

4️⃣ solveMultiFields

Cél

A mostani `solveField` logikájának többmezős kiterjesztése.

Most:

```
K_ff u_f = f_f − K_fc u_c
```

Ez marad.

Globális működés

Legyen:

- `K::SystemMatrix`
- `F::Vector` (globális rhs)

Solver:

1. végigmegy `K.problems`-en
2. minden mezőhöz:
    - constrainedDoFs
    - prescribed values

3. globális `fixed` és `free` DOF lista
4. redukált solve

A jelenlegi eliminációs módszer teljesen megfelelő.

5️⃣ supports struktúra

Jelenlegi elképzelés:

```
solveMultiFields(K, F,
    supports = [(Pu, [supp1, supp2]),
                (Pp, [supp3])]
)
```

Ez működik, de kicsit kényelmetlen.

6️⃣ Elegánsabb megoldás: field a BoundaryCondition-ben

Most a BC így néz ki:

```
BoundaryCondition("left", ux=0, uy=0)
```

Javasolt bővítés:

```
BoundaryCondition("left", ux=0; field=Pu)
```

vagy

```
BoundaryCondition("left", ux=0; field=:u)
```

Ez nem kötelező.

Előny

A solver egyszerűen:

```
for bc in supports
    P = bc.field == nothing ? single_problem : bc.field
```

Nincs külön supports struktúra.
Nincs mező-lista vacakolás.

Ez nagyon tiszta lenne.

7️⃣ Teljes folyamat példa

```
Pu = Problem(...)
Pp = Problem(...)

A = assemble_operator(Pu, Pu, ...)
B = assemble_operator(Pp, Pu, ...)
C = assemble_operator(Pp, Pp, ...)

K = SystemMatrix([A B';
                  B C])

F = [...]

u = solveMultiFields(K, F;
        supports = [
            BoundaryCondition("left", ux=0, uy=0; field=Pu),
            BoundaryCondition("p_node", p=0; field=Pp)
        ])
```

8️⃣ Mi marad változatlan?

- egymezős solveField
- constrainedDoFs
- freeDoFs

- applyBoundaryConditions!
- sparse solve logika

9️⃣ Stabilitási szempontok

Monolit blokk:

```
[A  B';
 B  C]
```

- robusztus
- saddle-point kompatibilis
- BC elimináció működik
- Schur komplement opcionális később

🔟 Jövőállóság

Ez a struktúra természetesen támogatja:

- Stokes
- Navier–Stokes
- BM
- thermo-mechanika
- u–p hyperelastic
- 3–4–5 mezős multiphysics

Nincs fix u,p.

1️⃣1️⃣ Fő tervezési döntések (összefoglalva)

✅ assemble_operator mindig tudja a row/col mezőt
✅ SystemMatrix tárolja a mező-metaadatot
✅ blokk-összeállítás automatikus
✅ solver nem találgat
✅ BC opcionálisan tud mezőt
✅ egymezős kód változatlan

🎯 A teljes architektúra egy mondatban

Az operátorok tudják, mely mezők között hatnak.
A blokk-konstruktor ebből automatikusan felépíti a globális rendszert.
A solver a SystemMatrix metaadataiból dolgozik.
A BC opcionálisan mező-specifikus.

Nincs duplikált információ.
Nincs fix u,p.
Nincs kézi sorrend.

Ha pár nap múlva visszatérsz, ez a gondolati térkép lesz az alap.

Ha akkor akarod, le tudjuk konkretizálni:

- a pontos adattípust
- a blokk-konstruktor algoritmust
- a solveMultiFields implementációját
- a BC mező-bővítését

Most viszont teljesen jó itt megállni.
Ez az architektúra már nagyon érett gondolat.