1 Gridap: Grid-based PDE approximations in Julia

1.1 Numerical PDE discretisations

Advanced PDE solvers are *complex*

- Plethora of discretisations: Grad, curl, and div-conforming FEs, hybridisable and virtual elements, discontinuous Galerkin, unfitted and CutFEM schemes
- Complex PDEs: Multiphysics, nonlinear, multiscale, complex constitutive models
- Nonlinear approximation: h-adaptive and p-adaptive methods
- Multiscale methods and multilevel solvers: Strong coupling between functional setting and solvers (not black-box)
- Uncertainty and quantification: intrusive polynomial chaos, (mutilevel) (quasi) Monte Carlo methods, etc
- Nonlinear preconditioning, inverse problems, data-driven parameter identification […]
- Large scale computations: distributed-memory implementations

We want to combine many ingredients!

1.2 Existing libraries

Excellent pool of high-performance libraries: deal.ii, fenics, FEMPAR, MOOSE, libmesh, etc.

- C++ or OO FORTRAN08 (static/compiled languages), some w/ Python interfaces (dynamic languages)
- Excellent if they provide all you need (*user*)
- Far more involved if not (*library developer*)

1.3 Computational math research

PhD students (3-4y), postdocs (1-3y)

- Starting from scratch every time not an option (hard to reach state-of-the-art interface)
- New algorithms to be implemented, may involve extensions of the library core
- Get into these libraries is *very time-consuming*

1.4 Dynamically- vs. statically-typed languages

Dynamically-typed languages:

- **Productivity**: More expressive, no compilation step (problematic for large libraries), interactive development (debugging on-the-fly), better for math-related bugs (no benefit from static compilation), no set-up of environment (compilers, system libraries, etc)

Statically-typed languages:

- **Performance**: Compilers generate highly optimised code

1.5 Solutions

- Dynamic-static combinations: Vectorised PDE solvers in Python + external pre-compiled libraries (NumPy); high-level Python interface of a static PDE library (fenics), etc.
- Constraints over the dynamic code (e.g. vectorisation)
- 2-language barrier: When changes require to get into static library

1.6 Julia: a new paradigm

Aim: Productivity *and* performance

- Productive: Dynamic language
- Performant: Advanced type-inference system + just-in-time (JIT) compilation
- 21st century FORTRAN, designed from inception for numerical computations (MIT, 2011-)
- Solve previous issues: for-loops not a problem, *everything* can be in Julia

Let us give it a try!

1.7 Julia features

- Multiple dispatching paradigm: functions not bond to types, dispatching wrt all arguments (solving multiple inheritance issues)
- Not OO: No inheritance of concrete types (only abstract types), *use composition, not inheritance*, *classify by their actions, not their attributes*…
- Performant Julia code is not obvious: help JIT compiler to infer types, *type-stability* to create performant code

1.8 Julia features

- Package manager is awesome

```
add "Gridap"
```

- Every project comes with its list of dependencies (automatic process)
- Seamless integration with `Github` (register packages, etc)
- Excellent deployment of automatically-generated code documentation in `Github`
- Unit testing and performance tools […]

1.9 Implementing grid-based PDE methods in Julia

Some key decisions:

- Functional-like style i.e. immutable objects, no *state diagram* (just some cache arrays for performance)
- Lazy evaluation of expressions (e.g. implement unary/binary expression trees for types)

1.10 CellFields

The core of `Gridap` are the `CellFields` types

`CellField`: It represent an array of fields (one, vector,…) per cells (e.g. edges, faces, cells in a mesh), where a `Field` provides a physical quantity (n-tensor) per space(-time) point in a mainfold

With these objects, we represent FE functions, FE bases, etc.

We also implement operations:

- Unary operations: e.g. `()`, `×()`, `()`, etc.
- Binary operations: `inner(,)`,`×`, etc.

1.11 Gridap in action

Let us go to Gridap tutorial 1

```
g(x) = 2.0
V0 = TestFESpace(V)
Ug = TrialFESpace(V,g)

f(x) = 1.0
a(v,u) = inner(  (v),  (u) )
b_Ω(v) = inner(v, f)
t_Ω = AffineFETerm(a,b_Ω,trian,quad)

op = LinearFEOperator(V0,Ug,b_Ω,t_Ω)
```

- Nesting objects into other objects via composition (mesh in FE space in FE function + bilinear form (duck typing) + triangulation + quadrature in FE operator…). All objects are immutable
- No numerical computations at this stage, just creating the expression tree (`()` and `inner`)
- Numerically intensive computations deployed here

```
uh = solve(solver,op)
```

1.12 Example: Nonlinear elasticity

Let us take a look at this tutorial

1.13 `Gridap` status

Gridap seed started in Christmas 2018 (1y ago)… trying to find ways to increase productivity in the team. Now we have (big thanks to F Verdugo's amazing work at UPC):

- Lagrangian, Raviart-Thomas, Nedelec […] FE spaces
- Discontinuous Galerkin methods
- Multifield or multiphysics methods
- Interaction with GMesh, Pardiso, PETSc […]

Quite complete documentation, tutorials

Dream: same software for research and teaching!

- One undergrad AMSI project on `Gridap`: from no idea about FEs/coding to MRI data of velocity of patient-specific aorta to pressure fields via in 2 months

- FE tutorials in *MTH5321 - Methods of computational mathematics*

On the way:

- Unfitted FE methods (Martin and Neiva's talks)
- Virtual element methods
- Historic variables for nonlinear solid mechanics
- h-adaptivity (`p4est` interface)

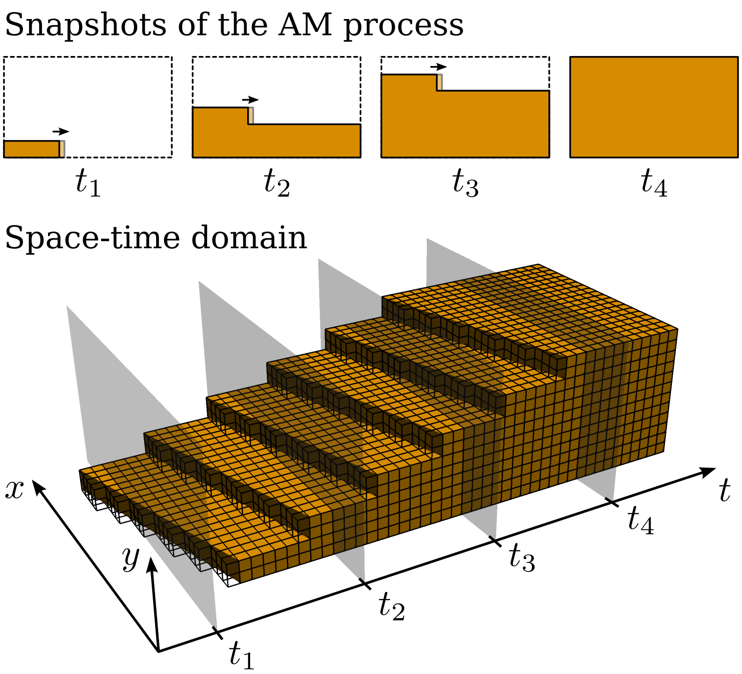

Figure 1: My comments